IBM Rational ClearCase 7.0: Master the Tools That Monitor, Analyze, and Manage Software Configurations

Take a deep dive into extending ClearCase 7.0 to ensure the consistency and reproducibility of your software configurations

Foreword by Lars Bendix, Ph. D., ETP, Lund University, Sweden

Marc Girod **Tatiana Shpichko**

[PACKT] PUBLISHING enterprise professional expertise distilled

IBM Rational ClearCase 7.0: Master the Tools That Monitor, Analyze, and Manage Software Configurations

Take a deep dive into extending ClearCase 7.0 to ensure the consistency and reproducibility of your software configurations

Marc Girod

Tatiana Shpichko

[PACKT] PUBLISHING enterprise professional expertise distilled

BIRMINGHAM - MUMBAI

IBM Rational ClearCase 7.0: Master the Tools That Monitor, Analyze, and Manage Software Configurations

First published: April 2011

Production Reference: 1190411

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-849680-12-7

www.packtpub.com

Cover Image by Tatiana Shpichko (tanya.shpichko@gmail.com)

Credits

Authors
Marc Girod
Tatiana Shpichko

Reviewers
Fernán Izquierdo
Torben Rydiander

Development Editor
Rukhsana Khambatta

Technical Editor
Neha Damle

Indexer
Rekha Nair

Foreword
Lars Bendix

Editorial Team Leader
Vinodhan Nair

Project Team Leader
Priya Mukherji

Project Coordinator
Shubhanjan Chatterjee

Proofreader
Aaron Nash

Graphics
Geetanjali Sawant

Production Coordinator
Shantanu Zagade

Cover Work
Shantanu Zagade

Foreword

My first encounter with software configuration management was way back in the eighties while at university – and way before I knew that it was called software configuration management. We were doing a student project and were five people working on this group project. I was coding away, slipping into experiments that eventually took the wrong directions – now where was that undo button? We were stepping on each other's toes overwriting changes from others or updating files with unpleasant and surprising side effects. All this hampered our productivity and caused us to have to work late nights to meet our deadline.

The professor, who later was to become my Master's thesis supervisor, told me that it was called software configuration management and that I would get the Nobel Prize in Computer Science if I solved the problem. He was involved in a start-up suffering more or less the same kind of problems. When I graduated I got a job in the industry – but problems persisted. We compiled older versions of modules, forgot to link in a couple of files in the executable or linked versions that had not been re-compiled. Often not causing big disasters, but still a mess that created a general feeling of confusion and uncertainty and caused quite a lot of rework. Apparently there was something they had not taught us at university. So after a while I returned to the university to do research and teaching (and from time to time help out companies) in software configuration management.

Later on I learned that (software) configuration management to companies (and standards organizations) meant something slightly different. It was more of an administrative and bureaucratic control activity with an emphasis on managing changes, preserving the integrity of the product and providing traceability between artefacts. This is also important for a project. However, it is comparable to putting brakes on the project in order to avoid that it runs out of control, whereas the software configuration management I had known was more like providing a team with a powerful accelerator. If you want to guide a project to a successful result you will need both an accelerator and a brake – and to know when and how to apply each of them (actually some successful Formula-1 drivers apply both at the same time going out of sharp turns). Books that explain about (SCM)brakes and how to use them are plenty and 13 a dozen whereas there is written surprisingly little about (SCM)accelerators – it seems to be practiced by people that are too busy to be able to "preach" it.

This is why this book is so valuable to practitioners as it deals with accelerators. How you avoid all those small mistakes that create a big mess and make your working life miserable. Software development is by nature most often the collaborative effort of a team of people. This team may be located locally or it may (as it happens more and more frequently because experts are hard to come by locally) be a distributed team. No matter what, people should be allowed to be productive and to work together as efficiently as possible. Build management with fast and reproducible builds is important for traditional development, but becomes even more important if you want to go with agile development methods. Imagine what would happen to the Test-Driven Development cycle of "write unit test(s), build the system, run the system, write code" if the "build the system" phase would take two hours to complete. Agile methods thrive on quick, immediate feedback (one of Kent Beck's fundamental values for Extreme Programming) and therefore demand fast (and reliable) builds.

During my years as a teacher, I have time and again met students who brought the latest version of a document or some code with them to our supervising meetings instead of the version they had sent me one or two days earlier. Every year I experience the thrill of seeing students on our second year project course step on each other's toes and pull the rug from under each other – at least in the beginning of the project period. From what I have seen during years of interaction with industry the situation in many companies is not that much better. Apparently good software configuration management behaviour is not baked into our DNA – and sadly neglected in the teaching of software engineering at most universities.

This is why this book is so valuable to everyone involved in software development. Most often if we follow our individual instincts we tend to ignore the communication and co-ordination that has to take place in a team effort and confusion and misunderstanding will be the consequence. Key concepts like integrity, reproducibility and traceability – good, old-fashioned configuration management concepts – are fundamental building bricks in any solution to that situation. So easy to pronounce, yet so difficult to implement and practise – Marc and Tanya take you by the hand and show you how.

I did not get my Nobel Prize, nor will this book earn Marc and Tanya the Nobel Prize in Computer Science. Not because the book isn't good – it is excellent – but because it is too practical and hands-on. Marc and Tanya are a strong team with many years of practical experience from both software configuration management and ClearCase usage. They are both driven by a strong passion for doing things the right way – and a deep curiosity for how to use a tool for most benefit and for exploring and pushing the boundaries of a tool's capabilities. ClearCase is a wonderfully powerful tool – but also somewhat difficult to master in particular for its more advanced features. I have seen students at computer labs shoot off both their feet in two seconds flat – and still walk away at the end of the lab thinking "what an awesome tool with all this cool stuff that it can do" – and they only scratch the surface in their labs.

This is why this book is so valuable because it brings together two capacities – one explaining the use of the other. If you go for a tool like ClearCase it should be because you want and need something more than "the ordinary SCM tool" can offer you. Marc and Tanya explain you how to take full advantage of ClearCase (and some of its more advanced capabilities) through a series of hands-on examples.

If you do not use ClearCase you may still want to read this book. Just skip the specific details and code examples related to ClearCase and enjoy the general guidelines and principles about software configuration management that Marc and Tanya introduce and explain. At the end of the book you'd probably wish you did use ClearCase.

Lars Bendix, Ph. D., ETP
Lund University, Sweden
March, 2011

Lars Bendix is an associate professor at the Department of Computer Science, Lund University, Sweden where he teaches a dedicated course on software configuration management. His main research interest is software configuration management and how it can be used to support software development processes. He is also interested in agile processes and their pedagogical use in software engineering education. He received a Master's Degree in Computer Science from Aarhus University, Denmark in 1986 and a PhD Degree from Aalborg University, Denmark in 1996.

About the Authors

Marc Girod grew up and graduated in France (MSci - Supélec 1983). He moved to Finland, where he lived for over 20 years, and then to Ireland. He worked first as a software developer, later as a ClearCase administrator and build engineer. He is currently working for LM Ericsson in Athlone, Ireland. He invested similar passion into several technologies over his career: Transputers and Occam2, C++, and since 1994, ClearCase. He is interested in contemporary philosophy (existentialism and postmodernity).

Tatiana (Tanya) Shpichko was born in Moscow, Russia. She got an early interest for mathematics and programming, which developed into a lifelong passion in the secondary mathematical school number 179. She graduated from Moscow State Technical University of Electronics and Mathematics with an MSci degree in Computer Science. Tatiana started with software development back in 1989, and has been programming in Pascal, C, C++, and Java. She first started to use ClearCase as a developer and then moved to work as a ClearCase administrator in 2005.

Tatiana has lived and worked in Finland for the past 10 years.

Tanya and Marc share a passion for scuba diving and underwater photography.

To each other, and to all the octopi, morays, and whale sharks of the world oceans. And to Sergey and Dominique, who have been helping and inspiring us all the way along.

About the Reviewers

Fernán Izquierdo (Madrid, 1981) is a technology and strategy freelance consultant with a strong background in software solutions and marketing strategies. Thus, in order to approach problems in the most practical but creative ways, he has received a very multidisciplinary education. This education includes Master Degrees in Telecommunications Engineering, Marketing, Cognitive Systems, and Interactive Media; a Master in International Business Administration (iMBA); and a Master Diploma in Filmmaking.

As a technology consultant, in the past few years, he has managed and led consultancy teams for IBM Software Services. All IBM final customers formed part to the Fortune Global 500 biggest companies. His tasks included technical support of the sales team, project planning, adaptation of IBM software products to customer needs, technical implementation of the solution in customer site, customer training, and post-sales support.

Besides, he has also worked for the United Nations Headquarters in New York, in the development of the NGOs worldwide web management system for the UN Department of Economic and Social Affairs. Other employees include the Dublin Institute of Technology and Zero Um Engenharia in Brazil. He has published 5 IEEE technical papers until 2011.

As a strategic marketing specialist, he has designed the worldwide commercial launch of the Reactable musical instrument in 2009, by Reactable Systems. His tasks included defining sales and marketing strategies and creating campaigns accordingly as well as customer and partners (communication, branding and supply chain) relations management. At the moment (2011), he is working for the Spanish Embassy in Tokyo as a marketing and international trade advisor.

He also believes that rich personal experiences are vital for a full life, both personally and professionally. Therefore, until 2011 he has lived in Spain, The Netherlands, Norway, Ireland, Brazil, United States, and Japan. This trajectory has resulted in him being fluent in Spanish, English, Catalan, Portuguese, and Japanese. Moreover, he has obtained more than nine fellowships around the world.

I would like to thank everybody who has shared a smile with me anytime anywhere. Maybe that smile led to a thought, the thought led to a discovery, the discovery led to a decision, the decision led to an action…and the action led to a word in this book.

Torben Rydiander has since 1984 worked with IT in the financial area in Denmark. Torben has since 2000 worked as Tool Specialist, with main focus on Change and Configuration Management, in the largest bank in the Nordic countries. The tools are mainly IBM Rational tools such as Software Architect, ClearCase, and ClearQuest. Torben is at the moment evaluating Rational Team Concert and hopes to implement it in the bank in 2011. His spare time is spent on golf and with his grandchildren.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

http://PacktLib.PacktPub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Instant Updates on New Packt Books

Get notified! Find out when new books are published by following *@PacktEnterprise* on Twitter, or the *Packt Enterprise* Facebook page.

Table of Contents

Preface

Both of us are essentially developers. One way or another, we took charge of building the software for our colleagues, and gradually moved into making this our main job. We have been involved in support and maintenance; evaluation, planning, and advocacy; installation and updates; design and implementation of customizations, and even to some extent, in design and implementation of enhancements and prototyping what could be a new SCM tool. Fundamentally, we remained developers.

We know that developers are creative and passionate people who do not like to be disturbed and who want to focus on their work, which is often at the limit of their own skills. These people are likely to leave to others mundane issues of planning, organization, politics. This is of course dangerous and often abused. One problem is that under the word *communications* one too often means either propaganda, or secret diplomacy and information concealing. Information doesn't flow well through the mediation of people who share neither the concerns nor the competences. This is not only an issue of conspiracy: there is something structural in the fact that competences, and even the concerns, are not being shared. Competences get acquired slowly, as a by-product of dedication and investment, and this process inevitably results in specialization. It is not simply a matter of e-learning.

We are deeply concerned about the future, the present, and the past of SCM. For what we witness, neither the efficiency nor the quality of software development have improved in the last period, and the trends are bleak.

Main stream solutions are always dull. It becomes worrisome when one can identify forces that drive them towards obviously wrong directions. This is how we analyze the excessive value given to *visualization*, and which in fact aims at making ignorance socially acceptable. Visual interfaces hide more than they show, and they do it on generic bases, which does *not* serve the specific needs of specialists. Their opacity is often overwhelmingly hard to reverse-engineer.

ClearCase was not a main stream tool. If it became one, it is at the expense of losing what made it valuable.

Evolution, in the absence of progress, works by throwing away the species which couldn't prove themselves useful, and bringing in new candidates at random. This is a fate we'd like to save ClearCase from; we feel that there is something there to keep. Communicating this to our peers, developers, is the ambition of this book. We found in writing it that despair didn't prevent some fun. We hope to be able to communicate some of it too!

What this book covers

The Teaser is there to raise questions about what to expect from a tool which pretends to revolutionize Software Configuration Management.

Chapter 1 – Using the Command Line presents and attempts to justify one important bias, and focus of this book: the power of the command line interface, as opposed to GUIs, and the sense of security and control it offers to the end user.

Chapter 2 – Presentation of ClearCase guides the reader into a review of the scenery, as light as possible: what are the main elements which build up the originality of ClearCase, and with respect to what historical background.

Chapter 3 – Build Auditing and Avoidance is the beef of this book, upfront! It is a technical chapter which explains what to pay attention to in order to get a real benefit from ClearCase. It explores the core around which revolves the conceptual integrity of the product.

Chapter 4 – Version Control drives back to the bread-and-butter of Software Configuration Management, showing that an exceptional tool as ClearCase must *also* provide the basic functionalities offered by its competition.

Chapter 5 – MultiSite Concerns brings the focus on issues that it is expensive to ignore at the time of early decisions, and to be reminded later. We show that provided these concerns are well understood, they build up no restrictions, and may on the contrary guide to elegant as well as robust designs.

Chapter 6 – Primary Metadata is a guide to an efficient yet simple use of the main tools, labels and branches, that ClearCase provides to manage—identify, discriminate, access—the user data during the critical phases of the development.

Chapter 7 – Merging tries to make essential simplicity emerge from the artificial complexity of common misuses. Merging is a necessary functionality, which only becomes frighteningly complex when abused.

Chapter 8 – Tools Maintenance turns to the important concern of managing the development environment, in order to guarantee the consistency and reproducibility of builds. It shows how ClearCase offers there again a powerful, elegant, and efficient solution to a hard problem.

Chapter 9 – Secondary Metadata makes some well informed choices among the profusion of weapons available under the ClearCase umbrella. It explains some sophisticated functionalities in detail, and debunks a few popular myths.

Chapter 10 – Administrative Concerns guides the end user into what she needs to know from the ClearCase administrator's tasks, which may not all be useful every day, but should be easily reachable when needed.

Chapter 11 – MultiSite Administration focuses on aspects of which even end users should have a basic understanding, in order to form correct expectations and to collaborate efficiently in distributed environments.

Chapter 12 – Challenges turns to some alleged weaknesses of the ClearCase model, and especially to the apparent mismatch of its build model with the main stream process of Java development. It argues that the foundation is sound and well worth being further developed to catch up with functionalities offered by other tools.

Chapter 13 – The Recent Years' Development is a critical review of the trends that have steered the ClearCase product. In our view, what made the exceptional value of ClearCase was lost from sight, and it is urgent to bring it back in the limelight.

Our Conclusion – ClearCase Future ends up as an invitation to collaborate on an open development of the most elegant functionalities brought by ClearCase and developed in this book.

What you need for this book

This book doesn't *require* any particular competences or knowledge. Its focus on practical examples will however make sense only for readers with access to a ClearCase development environment. Prior experience of programming languages and of collaborative software development, including building, versioning, possibly tool installation and maintenance, will undeniably help to share our sense of elegance.

Open mind and critical spirit towards authority, main stream and tradition, are probably the most valuable assets we hope for in our readers.

Who this book is for

If you are a developer who wants to use ClearCase for software development then this book is for you. This book is not for experts, but it will certainly challenge your technical skills.

While ClearCase concepts and tools are presented for those unfamiliar with the tool, an ability to use the command line as well as some fluency with shell scripting will certainly be an advantage.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
#include <stdio.h>
#include "wstr.h"
void wstr(const char* const str) {
  printf("%s\n", str);
}
```

We showed a reasonable amount of code, and of transcripts. We faced there the common problem of having to wrap long lines of text in a way which would allow our reader to understand what the original lines actually were.

We chose to cut the lines ourselves, to pad the first line with '#'s to the right margin (72 characters), and to right-align any continuation lines.

Italics will be used for general emphasis and less important concepts.

References to web pages are marked as `URLs` in the chapters' text, and the explicit URL list along with the chapters' and corresponding page numbers can be found from the Appendix.

New terms and **important words** are shown in bold.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or e-mail `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.PacktPub.com`. If you purchased this book elsewhere, you can visit `http://www.PacktPub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

Teaser

Let's start with a scenario, backed by a few brief transcripts. They demonstrate some functionality, which is not traditionally expected from an SCM tool, because it is supported only by ClearCase. Wait until the end of this part for a detailed review, and see the annex for the full sources of `Makefile`, `pathconv`, and `VobPathConv.pm`.

Many details of this *teaser* may remain obscure at this stage. We shall come back to them in the body of the book, and shall use this example to illustrate some issues, and the solutions we propose for them.

In our conclusion, we'll ask you to think of the unique functionalities of ClearCase, these which make it glow in the dark of today SCM, and to ask yourself: *How could I tease a smart colleague, not necessarily aware of ClearCase, into wishing they were available?*

And at this point, we wish you'll be able to implement ...about this.

But first the scenario

We intend to edit a file our own application uses—be it for a fix or an enhancement: it doesn't matter.

We won't show our application, but will focus on this file: `VobPathConv.pm`.
It is not one of the files that directly implement the main logic of our application, but rather a resource we use. In our case, it is a **Perl** *module* (a library), originally developed for a non-related purpose, and providing (among others) a function, `localtag2tgt`, returning the **vob tag** in a remote **region** (this of our **registry** server, typically UNIX) corresponding to a vob tag in the local region (may be either UNIX or Windows).

Many strange words, but do not panic! We'll review them in *Chapter 2*, *Presentation of ClearCase*. It is enough for our purposes to consider that this module serves an ancillary purpose related to portability between environments (a vob tag is the path under which a ClearCase database is mounted as a file system).

We noticed that the function returns an empty string in case of error (no corresponding path found), and consider making the function *die* (that is abort, in Perl terms) instead.

Here is a first transcript (more on **versions**, **branches**, and **rules** in *Chapter 4*, *Version Control*):

```
$ cd /vob/perl/lib/site-lib/5.10.0/ClearCase

$ cleartool ls VobPathConv.pm
VobPathConv.pm@@/main/imp/4                Rule: APP [-mkbranch mg]
```

We notice that the latest published version of the module bears a **label** we didn't place ourselves (labels will be covered in *Chapter 6*, *Primary Metadata*, although we will keep using them before).

`APP` is a **label type** of our own, but `JOE_1.35` is not!

```
$ cleartool des -fmt "%l\n" VobPathConv.pm
(APP, APP_2.57, JOE_1.35)
```

Looking at the label type, we find an **attribute** (of type `ConfigRecord`—not a predefined type: this might not make full sense until *Chapter 9*, *Secondary Metadata*, but it will then) pointing to the full path of a **derived object** (see *Chapter 3*, *Build Auditing and Avoidance*), which was used to apply it:

```
$ cleartool des -aattr -all lbtype:JOE_1.35@/vob/perl
JOE_1.35
 Attributes:
   ConfigRecord = "/vob/apps/joe/t/all.tests@@--04-01T07:51.345765"
```

We check that we can access this vob. We **mount** it. The derived object has not been **scrubbed** (we'll discuss scrubbing in *Chapter 10*, *Administrative Concerns*):

```
$ cleartool mount /vob/apps
$ cleartool lsdo /vob/apps/joe/t/all.tests@@--04-01T07:51.345765
--04-01T07:51.345765 "all.tests@@--04-01T07:51.345765"
```

We examine the **config record** and find a **makefile** referenced there (config records come in *Chapter 3*):

```
$ cleartool catcr /vob/apps/joe/t/all.tests@@--04-01T07:51.345765
Derived object: /vob/apps/joe/t/all.tests@@--04-01T07:51.345765
Target all.tests built by mg.user
Host "tarzan" running Linux 2.6.18-128.el5
Reference Time 2010-04-1T07:51:07Z, this audit started ###############
                                                    2010-04-01T07:51:07Z
View was vue.fiction.com:/views/jane/joe.vws
Initial working directory was /vob/apps/joe/t
----------------------------
MVFS objects:
----------------------------
/vob/apps/joe/t/Makefile@@/main/jn/1 <2009-10-06T13:02:29+01>
/vob/apps/joe/t/all_tests@@--04-01T07:51.345765
/vob/apps/joe/t/pathconv.cr@@--04-01T07:51.345754
----------------------------
non-MVFS objects:
----------------------------
/opt/rational/clearcase/etc/builtin.mk <2008-04-30T15:50:51+01>
----------------------------
Build Script:
----------------------------
      touch all.tests
----------------------------
```

We can set our **config spec** to use the label (more on config specs in *Chapter 2*—these are sets of rules to select versions):

```
$ ct catcs
element * CHECKEDOUT
mkbranch mg
element * JOE_1.35
```

This allows us to run the test case, using the same build script as recorded (**winkin** will be developed in *Chapter 3*):

```
$ cd /vob/apps/joe/t
/vob/apps/joe/t> clearmake all.tests
Wink in derived object "pathconv.cr"
Wink in derived object "all.tests"
/vob/apps/joe/t> clearmake all.tests
`all.tests' is up to date.
```

We check out the file (`VobPathConv.pm`) and edit it: when the vob tag path is unknown, instead of returning an empty string, set the function to `die`. Here is the `diff` (this is only the first fix):

```
< return (-d $tag? $tag : '') if $locreg eq $tgtreg;
> if ($locreg eq $tgtreg) {
>   die unless -d $tag;
>   return $tag;
> }
```

We run the test case again and see what this broke:

```
$ clearmake
   ./pathconv >/dev/null
Died at /vob/perl/lib/site-lib/5.10.0/ClearCase/VobPathConv.pm #####
                                                line 63, <GEN1> line 16.
*** Error code 2
clearmake: Error: Build script failed for "pathconv.cr"
```

There was, in `pathconv`, a test for this case, and now this test fails:

```
my $unknowntag = '/vob/foo';
$res = localtag2tgt($unknowntag);
$rc |= $res? 1:0; #expecting empty string
```

This is obviously plain regression testing—our change broke backwards compatibility and the test shows it.

We ought to note that it is *smart* regression testing: execution only takes place when something depended upon has changed and is otherwise avoided (winked in or up to date).

But there is even a more interesting quirk, and it is that *we* didn't spot this requirement: *our customer* did!

And she told us about it in a very clear and efficient manner, preserving two-way information hiding: we need to be aware of only the relevant bits of each other's concerns.

Irrespective of whether or not our relationship is commercial, it is our interest to keep our customers, and thus their customers, happy. By using our code, they raise its value, potentially also in the eyes of others.

This will result in better, and faster, feedback for us: an improved flow of advance notices of changes to take into consideration. We will be closer to the goal of driving the main stream, instead of following trends. This is always a two-way collaboration, not only a producer-consumer relationship—success is a win-win case.

We could now contact the author, and try to negotiate the case for our changes. But this is a bad idea in many respects:

- It disregards the fact that she already did her part of the work, so that we are already bound by backwards compatibility.
- She didn't disturb us, so why should we disturb her?
- In any case we would have to wait for her reply, then we'd rather restore a stable situation first.

So, let's revert our change (in this respect). We may even implement the new *die* functionality as a configurable option.

Time to review the exhibited functionality?

Let's sum it up:

- The possibility to communicate with test cases, elements of the *expected* behavior of software components, from the user to the maintainer, that is, upstream of the traditional information flow.
- The possibility for the maintainer to verify whether these users' *expectations* are still met after he makes some modifications, but before he publishes them; not needing to ask for feedback and wait for replies, that is, support for anticipated and canned feedback.

This applies to regression testing, but of course not only to it. It concerns tests as a special case of executables in general, that is, artifacts we can *run*, instead of artifacts we have to *read* (which is always more difficult, more involving).

The SCM tool may thus implement a new kind of very effective communications:

- *Horizontal*, that is, directly between interested peers, without the mediation of any organization
- *Bi-directional*, that is, not only top-down, from producers to consumers, but also bottom-up

- *Objective*, that is, freed from the need for, and the risks of, *interpretation*
- And last but not least, about intentions and expectations, not only *after-the-fact*!

Some people may think that this is remote from the primary concerns of SCM, which would be some kind of accounting and assurance about data availability. We believe that such judgments would be short-sighted, and out-of-step with the reality of today's software—communications is *the main concern* of SCM. Software development is about making choices in a context of incomplete certainty, so that it is all about convincing others and making commitments.

1
Using the command line

Something the previous chapter (**Teaser**) displayed already is our focus on the UNIX command line, as a primary working environment.

We see it as a means to maximize power and management over one's work. This gives access to scripting, in portable and efficient ways, as well as to accurate and relevant documentation.

We will devote a few words of justification to the command line interface. This will be followed with a few hints on Perl usage, and on tool configuration.

Rationale (pun intended)

There is a general perception that graphical user interfaces are a sign of modernity (in a positive sense). Also that they free the user from having to learn the syntax of tools, and thus bring in simplicity and help her to focus on her own tasks.

We do not think this is true. GUIs hide more than they show, and introduce discontinuities, thus making it more difficult to reverse engineer the hidden assumptions of their authors.

A possible objection is that GUIs and the command line would be addressed to and used by different groups of people, maybe in different roles. Let's stress that this only enforces our point. We do not believe anyway in such assertions, our experience being that:

- There is no task that couldn't be better carried with appropriate, possibly self-configured, command-line tools than with a GUI; we reject the idea that either would, for instance, be better or worse adapted to *advanced* tasks.
- The use of a GUI encourages a *back-seat driver's* attitude, the dispatching of one's responsibility to others, which fights the spirit and intent of **Software Configuration Management**.

- Encouraging practices that might lead groups of people, who are meant to collaborate on common goals, to ignore one another, is a dangerous idea.
- The command line is not a monolithic tool that would have to be *mastered* as a whole; it is easily extendible via scripting.

But be it one way or the other, GUI users want to use them, not to read books. Besides, the existing literature already covers them extensively. We thus propose to cover the other half of the world.

Finally, we do not intend to compete against IBM, which nowadays provides instructions on clicking on the right (or sometimes the left) button as videos.

Against intuition

GUIs aim at being *intuitive*. We do not deny them some success in this direction, which explains the satisfaction of many users. We only believe this is a misguided strategy, conflicting with the goal of *making sense* of one's, and others' work, and this for several reasons:

- Intuition doesn't scale**.** There is a limit in terms of complexity to what may feel intuitive to anyone.
- Intuition doesn't communicate: intuition is an event, and not a process.
- Intuition is hard to question or to validate.
- Intuition is only for humans; it is awkward for tools.

We'll follow Nobel Prize winner Daniel Kahneman (***Maps of Bounded Rationality: Psychology for Behavioral Economics*** in stating:

Intuition and reasoning are alternative ways of solving problems.

And we'll just opt for reasoning.

Some graphical tools such as maps are obviously very useful. But as Kahneman notes, consulting a map involves reasoning, not intuition. A large source of intuition in common GUIs is based on metaphors such as the desktop or ... the map (especially following *Google Earth*), which appeal to experiences familiar to end users, precisely because they are remote to *software*.

The continuity of reasoning

Let's take the perspective of users who intend to learn as much as they need, and not necessarily more but in a continuous, that is manageable, way. Users who intend to share their experience with others, by comparing their progress gradually, and reproduce the steps they take, at their own pace.

Reasoning, and building up one's understanding, is a process that takes time and goes through states. It is useful and safe to identify and validate these states with others: this defines a path. There may of course be many roads from one point to another, and nobody is forced to take the same road as anybody else, but it is essential to identify the roads in a way allowing one to compare them. This identification operates in practice on discrete states. It may feel paradoxical, but continuity is a result of using discrete representations instead of more *continuous* graphical ones.

Text is well-suited to record the progression of reasoning, in a way easy to trace by the next pilgrim.

Recording achieves an essential *SCM* concern: *re-produce* rather than produce (from scratch, re-inventing the wheel) and *re-use*. We think that one ought to use an SCM tool (ClearCase) in an SCM way! One's investment will be rewarded.

Another important issue is the possibility to get access to the specific outputs of different tools, and to be able to analyze them by parsing, searching, and comparing relevant details rather than being stuck by the opacity of a pop-up window. Nowadays, the *production* of documents, be it text or not, is assisted in frightful ways, which puts the *reader* to face ever more difficult challenges. Plain text is her best chance to practice *active* reading, that is, to decide precisely what information she wants to focus upon and to actually make it emerge out of the surrounding noise. Passive readers most often have the information they need at their disposal, but fail to notice it; they are instructed by experience to avoid spending the effort necessary to read it!

The users we target are not alone, and their reading is well assisted by powerful tools—text being still significantly better supported than graphics. Not only is there a profusion of existing tools, but there is a world of scripting that allows the user to maintain her own ad hoc tools.

Illustrations

French sociologist Pierre Bourdieu explained in *On Television* that communications takes time; instant communications is no communications at all. Graphical illustrations are often misleading in this respect: they tend to overspecify a situation for the purpose of giving a representation of one single aspect. The constraint of producing an image implies to complete the useful parts with details, which otherwise would be left undetermined. We'll pay a special effort to avoid this mistake, and will therefore restrict ourselves to producing textual illustrations, that ought to be read from a beginning to an end, and therefore lend themselves to critical analysis from a careful reader.

Text, shell, and terminal

When we think of text, we think at the same time of the format of files and of tools to operate on the system: **terminals** and **shells**. The files are the natural logs of this system operation. Of course, most text editors allow us to modify text in arbitrary order, but each text file implements the convention of time—from its beginning to its end. Note how there is nothing similar with graphics (apart for cinema).

Storing commands as text files and processing text from files in the same way as from any text stream is called **scripting**. The boundary is not meant to be obvious, and we will have an example below.

This brings us to the point that *text* is not as straightforwardly and universally defined as one might naively assume; it is subject to encoding conventions, and rendering formats (for example, to support colors or fonts). The goal of preserving contents through text-to-file-and-back transformations, leads one to restrict the definition to the barest (restricted character set, single font, single color). We still have to pay attention to one crude issue: incompatible end-of-line conventions. There are actually three: UNIX (one character, *linefeed*—ASCII 10 decimal), Windows (a two character sequence: *carriage return* linefeed—ASCII 13 - 10), and Mac (carriage return, #ASCII 13, alone). One way is to take care of conversions, and the problem is to decide at what point. One option, which we'll touch in *Chapter 2*, *Presentation of ClearCase*, is to delegate it to the view. The other, clearly inferior option, is to use checkin triggers. The best is to raise the level of understanding of users, and to have them stick to tools consistent in this respect, that is, in practice avoid tools that enforce the Windows convention.

The *shell* is the program in charge of executing interactive commands (on UNIX, it is distinct from the *terminal*). It presents annoying differences between Windows and UNIX such as:

- The use of back versus forward slashes as separators in directory hierarchies (including in the format of ClearCase *vob tags*—see Chapter 2). Backslashes are used in UNIX as *escape* characters to prevent the evaluation of special characters. The result is that they get consumed by the shell, so that to preserve them, one needs to escape (`\\word`) or otherwise *quote* them (`"\word"`);
- Quoting has two flavors on UNIX but only one on Windows. Double-quotes (`"word"`) allow the evaluation of variables in the quoted text; on UNIX, single quotes ('word') make it possible to prevent this.
- The UNIX file systems are all *rooted* from a common directory, marked as `/`. This is not the case on Windows where the situation is more complex: while file systems (rooted in \) are usually *mapped* to single letter *drives* (followed with a colon, for example, `C:`). On the other hand, a competing syntax may be used to access named *shares*, prefixed with double backslashes and host names (for example, `\\frodo\nest`).
- Shells support the expansion of *environment variables*, with different syntaxes: `$VAR` on UNIX and `%VAR%` on Windows. The definition syntax varies even between UNIX shells.
- Less visible, but let's mention it: the separator used in list variables such as `$PATH`/`%PATH%` differs from `":"` (UNIX) to `";"` (Windows). We'll have an example later with `MANPATH`.

This not so brief review should convince us that it is challenging to write simple commands, even just to run programs, in a way that would be portable (that is, stable) across the UNIX/Windows boundary. This issue is usually crucial in almost any organization using ClearCase: a typical environment comprises a large number of servers and workstations based on a variety of platforms with a tendency to run the servers under UNIX and the end-user workstations under both UNIX and Windows. May this be our reason to introduce Perl, as an answer to the portability issue.

Perl

In this book, as already mentioned in the *Teaser*, we'll focus on Perl as a scripting language for our examples. There are many reasons for this choice.

We already mentioned portability: Perl is available on all the platforms on which ClearCase is supported, and allows (with the help of a few suitable **modules**) to factor away and to hide the platform idiosyncrasies.

A second reason to choose Perl over other scripting languages (first of which the UNIX shells) is the existence of a debugger, and the model of compiling first and running only code that has satisfied a first check. Both of these are significant advantages to cover a range of users extending to the administrator. Note that the debugger may be used interactively from the command line (`perl -d -e1`), and "one-liner" commands are possible, not to force you to save scripts to files.

Then could come the existence of a mass of experience, both via the CPAN library network (`http://cpan.org`), and the many newsgroups and forums (see perlfaq2, below), thus giving access to a culture of communications and excellence.

Finally, one must mention that Perl has been used by the developers and distributors of ClearCase. However, at this point, we would recommend that the user avoids the apparent facility of using the instance of perl that is part of the ClearCase distribution and instead maintains her own installation in a ClearCase vob (obviously sharing it network-wise with her collaborators—see *Chapter 8*, *Tool maintenance*, and note that we implied this in the *Teaser*).

IBM has relatively recently improved its support for Perl, with a consistent ratlperl in the `common` tools directory. However, this doesn't answer the following points:

- It is not intended for the end users: it is tailored for the needs of ClearCase tools. Remember IBM is not committed to supporting it in any way.

- It is still a relatively old version and comes with a minimum set of modules, so you'll sooner or later get into limitations (check the available modules with `egrep ^=head2 perllocal.pod`, where the `perllocal.pod` is located in the platform subdirectory, which you get with `ratlperl -V`).
- You might think of installing modules yourself, but:
    - If you need to compile them, the compiler needed to build it is not provided—you'll fail to install even ***ClearCase::CtCmd*** (which is a Perl module provided by IBM on CPAN)!
    - Your changes will be washed away at the next ClearCase installation.
    - You'd need to make the same changes locally on every host, thus to have admin rights there.
    - In fairness, there exists in *perlfaq8* advice on *How do I keep my own module/library directory*, which makes it possible to maintain modules outside of the Perl installation, forcing the users to reference the shared library path.

We want to stress that Perl is a means to open ClearCase to its users, to enable them to tailor it to their needs. This is true for every user, not only for administrators with special needs! The main need it serves is it gets one freed from the GUI.

Perl documentation

One thing Perl encourages you (the user) to do is extend the Perl documentation, and does this by lowering the threshold for producing quality documentation. Although on a complete installation, the perl documentation will be available as man pages, perl offers a format, **pod**, and a tool **perldoc** to achieve these goals. This displays another argument in favor of text: *less is more*.

Perl provides a possibility to search through its documentation using the same perldoc tool, so that one can search through Perl FAQ, functions or modules:

```
$ perldoc -q 'regexp' # Searches free text in a form of regular expression
$ perldoc -f perl_function_name
$ perldoc -m perl_module_name
$ perldoc perlfaq2
```

Windows command prompt and alternatives

For Windows, the default terminal choice is the *Windows cmd*. It has a limited functionality, for example, no logging, limited buffering, few hot keys, a clumsy (first arbitrary choice) tab completion, and so on. But as restricted as it is, to use ClearCase, it is still much more powerful, and more functional than the GUI. Even some quite basic ClearCase commands (for example, create a symbolic link in the vob root directory, change file permissions, remove labels from a version, and so on) are only available on the command line.

Here are some useful hints on configuring the Windows cmd to make it more usable:

- Enable the Quick edit mode. Locate `HKEY_CURRENT_USER` -> `Console` -> `QuickEdit` in Registry Editor (`Regedit`) and set its value to 1. This enables easy copy-pasting (helpful in the absence of any other text handling utilities/ keys): select with the mouse and press *Enter* for copying, and right-click to paste. This may also be set and unset in the command prompt window, in the **Properties** pane of the menu, by right-clicking the menu bar.
- Set appropriate *Screen Buffer Size* and *Window Size*. This also affects the ease to copy and paste!

Here is a small example of what one can do in Windows command line prompt:

```
> cleartool startview myview
> cleartool mount \myvob
> cd m:\myview\myvob\dir1
> cleartool ln -s ../dir2/file my_symlink
> cleartool mklbtype -nc LBL
> cleartool mklabel LBL ../dir2/file
> cleartool rmlabel LBL_1 ../dir2/file
```

This transcript creates a symbolic link, then creates a label type, applies a label to a versioned file, and removes another label from the same version.

A far better choice of a terminal for Windows would be **Cygwin**. Even the default Cygwin terminal window uses GNU bash shell, with all the expected bash features supported (such as Emacs key bindings, bash profile, and so on). Installing the Cygwin `ssh` package, one could use it conveniently from the same Cygwin terminal.

The terminal we use ourselves for both UNIX and Windows is GNU Emacs shell mode. Originally a text editor, GNU Emacs has developed, by the virtue of the genericity of text, into a full-blown integrated environment.

In Windows, one of the options is again to obtain Emacs as a Cygwin package, and to use it in Cygwin/X mode.

Emacs' ability to combine shell, file editing, man, and info modes, powerful multi-window and multi-frame system, unlimited customizing options, powerful logging, text-and-command search, and the ability to edit remote files via tramp/ssh, makes it a powerful choice, even for "normal" users, not to mention administrators. The pattern of using Cygwin Emacs is not without minor pitfalls such as the absence of the out of the box tab completion and `Ctrl+C` process signaling over ssh.

Let's compare Cygwin with a UNIX *terminal*.

Here is an example of what one can do in a UNIX terminal:

```
$alias ct=cleartool
$export DO='"do@@--04-11T14:50.3375"'
$ for i in $(ct lsvob -s); do if ct des lbtype:LBL@$i >/dev/null 2>&1; \
  then echo mkattr -rep ConfigRecord \'$DO\' lbtype:LBL@$i; fi; \
 done | ct
```

This UNIX shell code goes through all the ClearCase vobs, checking whether a label type `LBL` exists there. Every instance of `LBL` gets an attribute of type `ConfigRecord` and value `do@@--04-11T14:50.3375`.

This demonstrates the use of *pipes*, that is, a model of text as a stream, produced by one process and consumed by another—here cleartool. It also shows the use of basic programming language constructs such as *loops* and *conditionals* as part of the (here *bash*) shell, as well as the definition and use of an environment variable, together with a quoting issue (the `mkattr` function of cleartool requires that string values are enclosed in double quotes, which themselves have thus to be preserved from the shell). Note that the conditional tests the return code of a cleartool function (`describe`), both stream outputs (*standard* and *error*) of which are ignored by being sent to a special device, `/dev/null`, with a file interface. Note the use of `ct` as a shorthand for cleartool—we shall keep using it! Aliases need to be marked as expanded with the `shopt` command in order to work in non-interactive mode, as in a pipeline.

Now we can illustrate the annoying differences in Windows and UNIX shells explained earlier (in the *Text, shell, and terminal* section of the chapter). The UNIX transcript shown above would not work as such on Cygwin. The vob tags returned by the (Windows ClearCase) `lsvob` function would be rooted in backslashes, which Cygwin bash would evaluate as escape characters and thus remove. We would have to restore them, for example, as follows:

```
for i in $(ct lsvob -s); do if ct des lbtype:LBL@\\$i >/dev/null 2>&1;
then echo mkattr -rep ConfigRecord \'$DO\' lbtype:LBL@\\$i; fi; done | ct
```

This may seem as a tiny issue at first glance, but errors like this are enough to break a script or a build system, and chasing them soon takes enormous efforts in implementation, debugging, and testing.

As we already said, Perl allows us to bridge this gap, for example, with a cleartool wrapper as ***ClearCase::Wrapper::MGi***, itself building upon ***ClearCase::Wrapper*** and other modules. A wrapper installs itself as a Perl script—here `cleartool.plx` in `/usr/local/bin`, and may be aliased as `ct` instead of `cleartool`. It extends and corrects the cleartool behavior, falling back to the standard cleartool for the non-extended functionality.

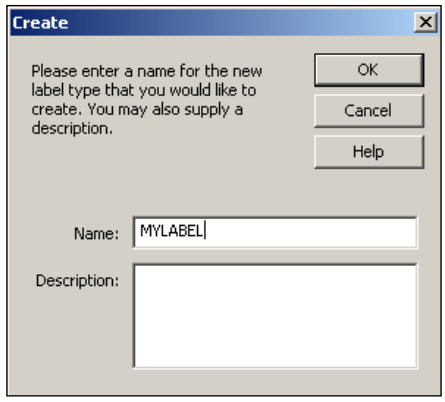
Now, with the help of ClearCase wrapper, one can run *exactly* the same original transcript on both platforms, UNIX and Cygwin Windows.

GUI versus text mode

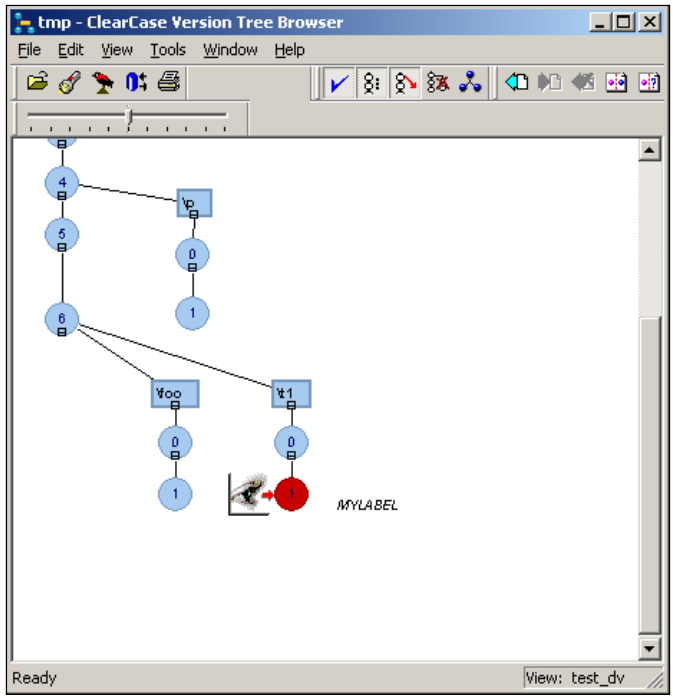
Let's now have a small (biased) illustration on different modes of working—in ClearCase GUI and in ClearCase command line. Suppose, one would like to apply a label recursively to a directory hierarchy in ClearCase, containing a few subdirectories and hundreds of files.

First, a GUI scenario:

1. Open the Rational ClearCase Explorer (`C:\Program Files\Rational\ClearCase\bin\clearexplorer.exe`).
2. Select the view.
3. Mount the vob.
4. Click on the vob's root directory and select **Explore types**.
5. Double-click on **label type**.
6. Go to the **Type** menu and select **Create**.
7. Type label type name as **MYLABEL** in the **Name** field. Click **OK**.



8. Get back to the vob and locate the directory you want to label.
9. Right-click on the selected directory and select **Version tree**.
10. Right-click on the version selected by the view config spec and marked by the "eye" icon. Select **Apply label**.
11. Select **MYLABEL** from the list. Click **OK**.
12. Repeat the three last steps for each element under the selected directory (you might have to repeat the steps a hundred times…).



Now, let's look at the command-line scenario:

```
$ ct setview myview
$ ct mount /vob/myvob
$ cd /vob/myvob/mydir
$ ct mklbtype -nc MYLABEL
$ ct mklabel -rec MYLABEL
```

And we are done (feel the difference)!

One might object that it is possible to add a functionality such as `mklabel -recurse` to the suitable menus of the ClearCase Explorer. The tool to perform this is the ClearCase Context Menu Editor.

Of course, one might also object that our examples are not fair, and they are not. We stand for our point—it will always be easier to find yet a better hammer to hit any given nail using a command line than using a GUI (be it merging files or applying labels). Beyond the fact that finding a *wizard* might be non-obvious and disruptive of the user's flow of mind, the magic this one resorts upon will fall short eventually and require then reverse engineering knowledge. The user is, in fact, only buried one step deeper.

ClearCase documentation

ClearCase has a comprehensive documentation set in the form of so-called **ClearCase man pages**.

ClearCase shares, in a way, the same philosophy as Perl concerning its man pages. A ClearCase man page refers basically to a file coming along with the ClearCase installation, residing on a local machine and containing a relatively small piece of ClearCase documentation, dedicated to a single command or topic. Reading ClearCase man pages may be achieved in a standard UNIX way, using the `man` utility (or any other similar tools, for example, as part of GNU Emacs). It may also happen using the **man** function of the cleartool utility. On Windows platforms, this will result in requests to one's browser to display equivalent HTML pages.

Using UNIX man for accessing the ClearCase man pages requires little configuration. The man pages reside under `/opt/rational/clearcase/doc/man`; one needs thus to add this directory to the `MANPATH` environment variable. There is a special prefix to name the ClearCase man pages: `man ct_<ClearCase command name>` (sometimes with an abbreviation).

```
$ MANPATH=$MANPATH:/opt/rational/clearcase/doc/man
$ man ct_co
$ man ct_describe
```

These commands would display the man pages for the `checkout` (`co`) and `describe` commands respectively.

Alternatively, as mentioned earlier, and just like with the Perl `perldoc` command (which also performs as an alternative to `man`), one can use `cleartool man` (`MANPATH` setup not needed):

```
$ cleartool man co
$ cleartool man des
```

The distribution (both on UNIX and Windows) also offers a set of useful ClearCase books in PDF or HTML format (also available for reference purposes from the IBM website). They are located at `/opt/rational/clearcase/doc/books` in UNIX and at `<installation drive>:\Program Files\Rational\ClearCase\doc\books` in Windows.

There is finally a useful cleartool functionality (only on UNIX): **cleartool apropos**. It displays a brief summary of cleartool man pages matching a query to be specified as a regexp:

```
$ ct apropos 'label.*version'
mklabel Attaches version labels to versions of elements
rmlabel Removes a version label from a version
```

ClearCase does not support the users in extending this documentation (as Perl does with the *pod* markup language).

ClearCase online help is also available at the IBM Rational information center: `http://publib.boulder.ibm.com/infocenter/cchelp/v7r0m1/index.jsp`.

Note that there are separate resources there for each ClearCase version; this particular one is for the Rational ClearCase 7.0.1 family, although it is not as specific as one might think, and applies to other versions as well. We once submitted an RFE requesting to allow the user to assert changes between these docs, which was rejected.

Here are some ClearCase-related Internet resources and forums:

CM Crossroads Wiki
`http://www.cmwiki.com`

CM Crossroads ClearCase Forum
`http://www.cmcrossroads.com/forums?func=showcat&catid=31`

IBM Rational ClearCase forum
`http://www.ibm.com/developerworks/forums/forum.jspa?forumID=333`

Summary

We have reviewed some issues faced by the team player in trying to record her work and experience in a way that might be reproducible or even comparable by others. With all this set up, we are better equipped to dive into our actual topic: ClearCase.

2
Presentation of ClearCase

Soon 20 years after its introduction, ClearCase is still one of the major **Software Configuration Management** tools on the market.

The concept of SCM is however not as clearly defined and agreed upon as some might pretend. ClearCase itself actually challenged any prior definition significantly by introducing novelties to what SCM might mean. These novelties are, as we'll try to show, what makes of ClearCase, still today, an original and valuable product. Instead thus of risking to fix too early the meaning of SCM, we'll offer a historical perspective, and aim at refining toward our conclusion how ClearCase contributes to define a concept both consistent and useful to the wide community of software users. On this path we choose however to consider SCM essentially as a *concern*, which might and should be shared virtually by anybody, not a concept for the exclusive use of specialists.

Our initial presentation will:

- Offer the above mentioned initial historical setup
- Focus upon the features that make ClearCase stand out
- Describe the elements of the implementation which will be referred to in the following chapters

In this book, we'll cover mostly what became known as **base ClearCase**, and leave the UCM extension (Unified Change Management). Justifications for the soundness of this choice will become clearer and clearer through the reading. Let's state in addition that UCM is already well, and exclusively, covered by existing literature.

SCM history

> *There's one thing history teaches us, and it is that it teaches us nothing. — Author forgotten*
>
> *We have one duty towards history, and it is to rewrite it. — Oscar Wilde*

A brief perspective of SCM history helps to debunk some myths about what to expect from the different tools. It is commonplace to read matrix comparisons, based on checklists of *standard* features. We try to show here that such comparisons are based on unsubstantiated assumptions: there is no "high-level" definition of SCM, which would be applicable to the tools. At least, the most useful functionalities offered by ClearCase cannot be expected on such bases: this is what we'll review next.

CM, *Configuration Management*, was born twice independently, and a third time as a synthesis. It happened first, in the automotive industry, in the 40s, as an accounting discipline. The second birth is this of **version control** or "source code management", and took place in the early steps of the software industry, in the 60s.

The record of this second birth is a bit murky, especially as it spreads over a period of more than ten years, and the people it involves undoubtedly knew and interacted with each other before they published anything. Let a slight bit of straightening help making it clearer, even if it may seem to conflict with some chronological evidence, as we mention it below.

In its early stage, software development was pure creation *from scratch*. This gave an extreme importance to sources: no source, no binary. Of course, this bias, although still strong in oldtimers' mind, doesn't make much sense nowadays anymore: we all get most of software in executable form first, and most often its functionality is the only (OK: the main) thing which matters. Only later, possibly, we become interested in reading the source code, maybe to make a fix or an enhancement.

Comparing text sources was made handy with the `diff` tool. Then the reverse tool was invented: `patch`, which allows you to apply diffs to a similar (but not identical) version of the original text. These tools made it possible to implement cheap (in terms of space consumption) personal backup. Soon again, it was understood that this backup constituted a trace of the recent evolution, which it was valuable to share. Let's admit it: it is not exactly the way it went. For ten years before Larry Wall actually wrote `patch`, people used `ed` line editor scripts, and this is what went into the first version control tools.

The third moment is this of the "official" birth of **SCM** (at least, with this acronym), as the merger of the two traditions, and it happened as early as in the 70s.

We have again to acknowledge that this presentation of history is sometimes received as outrageous, especially by proponents of the first *CM* tradition.

One cause of problems with this history (for example, SCCS was in fact known earlier than diff and patch), is that things became gradually simpler only after the inception of C and UNIX. But this didn't happen at once; several tools had already been developed for years in other languages, on other platforms, and were only later converted to C and ported to UNIX. The important date concerning every one of these tools is thus maybe not when it was created, but when its propagation reached people who made it successful.

ClearCase originality

Two main ideas make the originality of ClearCase, as an SCM tool:

- Presenting the version database via a file system interface (distinct from a sandbox or working copy)
- Auditing build transactions, and recording the system calls for opening file objects, to produce a graph of dependencies, and to base on it an identification mechanism

The two ideas are strictly speaking independent. ClearCase however builds upon their interaction.

Virtual file system

It is not obvious for people with an experience with other tools, and the prejudice of the reference to an abstract concept of *version control* (or source control), to give value to these ClearCase specificities.

Accessing the content of the repository **in-place** (as opposed to in a copy) allows developers to benefit from the tool management *earlier*: during the edition phase itself, and not only after the artifacts have reached a certain status (say, *buildable*, or *tested*), and are thus more stable. It should be quite obvious that the less stable the artifacts are, the more useful the management!

Work performed in a **dynamic view** (the device to exploit this virtual file system abstraction: more on this below) is accessible for inspection by others (on *their* terms), which brings continuity. Developers can access one another's dynamic views explicitly, but the most interesting and important ClearCase feature is the *implicit* artifacts' (so-called **derived objects**) sharing, performed by the tool itself. This is decoupled from deliveries that imply an explicit decision and judgment from their respective authors: a statement about readiness with respect to conventional expectations or general usefulness.

Auditing, winkin

Derived objects are even more variable than source files (because they collect multiple dependencies). The decisive functionality of ClearCase is a mechanism allowing to share these, implicitly, and under the control of the tool. This certainly has a potential for saving time and space (avoiding needless builds, and copies), but the most valuable is that it sets up a foundation for managing complexity. It does it in two ways: first, by reducing the overall amount of separately distinguishable artifacts, and making real differences emerge (that is, raising the signal/noise ratio); then by introducing an objective structure (the dependency graph) into the space of configuration items.

The mechanism at the heart of **clearmake** (the build tool of ClearCase, focus of *Chapter 3*, *Build Auditing and Avoidance*) is based on auditing, and recording as dependencies files accessed during the execution of commands, such as those involved in builds. This happens by intercepting system calls. The records may be used later to decide that an existing artifact matches the requirements for a new one, the production of which may thus be avoided. Clearmake then offers, **winks in**, the old product in place of a needless new duplicate.

ClearCase is not exactly the only system building on these ideas: one can also mention ***Vesta***, and (for winkin) ***Audited Objects***, by David Boyce.

Also, let's note that recent developments of ClearCase do *not* build on those ideas, and thus depart significantly from what makes base ClearCase attractive: snapshot views do *not* most of the time give access to the versioned database (they do it in a restricted way, for example when using `ct find`, which connects to the view server, and fails if it cannot), and do *not* support build auditing. Note the use of `ct` to alias **cleartool**, as advertised in *Chapter 1*, *Using the Command Line*.

One problem is that these features have a price (mostly in terms of requirements on the network), and unless one uses them, this price becomes a penalty, comparing ClearCase to its competitors.

In addition, optimizing winkin involves collaboration from the users, and thus orients their effort. The same may be told of SCM in general (users may easily defeat any kind of manageability by choosing adverse practices: useless duplication, coarse granularity), but sophistication is even more dependent on user commitment. We'll focus in this book on ways to exploit sophistication to its full potential, and over.

The main concepts

ClearCase defines several sub-functions that build up a network of server-client relationships. These functions may be mapped onto different hosts. On every host running a ClearCase functionality, we'll find one *location broker* (`albd_server`) process, which plays the role of a postman. It is always accessible via a *well-known* address, at port 371, and will answer requests concerning the location of processes in charge of the other functions. Note that standalone installations, where all functions are run on a single machine, are possible.

A command line utility, `aldb_list`, is available to check that this essential process is up and running, and, from remote hosts, is reachable.

Its response will be a list of all ClearCase processes known on the host (which may be long). This may often be ignored (by being sent to `/dev/null`): then a one line status summary is sent to `stderr`, which gets displayed (since it is not redirected by the previous).

```
$ /opt/rational/clearcase/etc/utils/albd_list badh >/dev/null
noname: Error: Unknown host 'badh': Host not found
cannot contact albd

$ /opt/rational/clearcase/etc/utils/albd_list goodh >/dev/null
albd_server addr = 100.1.2.48, port= 371
```

Note that it is not enough to guarantee full operability (as other ports are needed for most operations).

We'll now review the main distinctive concepts, and the ways they are implemented:

- Vob
- Dynamic view
- Registry, region, site
- Config spec

Vobs and views

VOB stands for *Versioned Object Base*. Vobs implement in ClearCase the shared repositories for arbitrary data. Views, although nothing prevents one from sharing them, typically support the stateful access of multiple vobs by a single user.

Vobs and views are databases (embedded, object-oriented, not table based). At any time, the databases are kept in the memory of server processes (`vob_server`, `vobrpc_server`, `view_server` and `db_server`), but they are also saved to files, in storage areas. The memory images are read from the files, and the caches flushed back to disk at various points, for example, for backup purposes. In the storages, we find some pools with the main data, the database files, as well as some index files.

The databases are however mostly exposed as **mvfs** (for multi-version) file systems, a technology building upon *nfs* (and adapted to *cifs/smb* for Windows). Vobs are thus **mounted**, or **mapped**, while views need to be **started** or **set**. Both are known via their **tags**, which can be thought of as **mount points**. Only, the vob tag is a full mount point (which is a full path) whereas the view tag is only a subdirectory of the view root (`/view` on UNIX and `\\view` on Windows). Then, the mounted file system is actually a combination of both a vob and a view. On UNIX, it is customary (not mandatory) to mirror the view hierarchy and use a common root under which to mount all vobs (typically, `/vob`).

Note that neither this root nor the full vob tags are themselves versioned: they may be considered as common stubs, shared by all paths of all versions. They'd rather be kept very stable, and thus their names should not be too specific. In the example below, for a vob having `/vob/myvob` tag, one can easily rename any subdirectory under `/vob/myvob` with `ct mv` command, but it is not possible (or not recommended) to rename either the vob tag directory `/vob/myvob` itself, or the common vob mount root `/vob`. To be more precise, this is possible, but not *under ClearCase*, and creating a new tag would affect the history as well as the future.

```
$ ct lsvob /vob/myvob
* /vob/myvob /vobstorage/myvob.vbs public
$ mount | egrep 'myvob|vobstorage'
vobhost:/rootdir/vobstg on /vobstorage type nfs
vobhost:/rootdir/vobstg/myvob.vbs on /vob/myvob type mvfs
```

This shows the vob mount point: `/vob/myvob` (which is also the vob tag) of the MVFS filesystem physically stored on the host `vobhost` at `/rootdir/vobstg/myvob.vbs` **vob storage** directory. Note the `*` on column 1 of the output: this tells us the vob is actually *mounted* and not merely registered together with this tag. Be aware of the fact that one cannot "access directly" a vob on its vob storage and for example, check out something from there. The only possible way to access the vob is via a view, as explained below in this chapter.

```
$ ct lsview myview
* myview /viewstorage/joe/myview.vws
$ mount | grep -r "/view |viewstorage"
viewhost:/rootdir/viewstg on /viewstorage type nfs
localhost:/dev/mvfs on /view type mvfs
$ ls -ld /view/myview
drwxr-xr-x 38 joe jgroup 4096 Feb 15 11:16 /view/myview
```

And this shows the view mount point: `myview` (also the view tag), which is a subdirectory of the *root* MVFS filesystem mounted at `/view`; and the actual view storage directory is located on the `viewhost` at `/rootdir/viewstg/joe/myview.vws` directory. When the view is *set* (on UNIX) ClearCase performs a `chroot` system call to set the root directory to `/view/myview`, which explains that all vobs, but also all the contents of the UNIX root are found there. This obviously includes `/view` itself, which needs to be taken into consideration if recursively navigating directories with UNIX `find`. A typical case is this of running the GNU utility `updatedb` from a crontab, to support the `locate` tool. Again, the view is already started: the `view_server` process is running.

A vob is thus a multi-version repository, which stores in addition shared derived objects, and meta-data.

Views have some additional storage for private (non-shared) objects, but otherwise work as a filter on the vobs, presenting to the user and her tools, a consistent selection of the versions available from the vobs. They implement the support for the abstract concept of **software configuration**, and the **exclusion principle** this one satisfies: at most one version of every configuration item. Note that it is typical that one uses in the same time multiple vobs, but one single view. We'll pay more attention below to the mechanism used to express the version selection: the **configuration specification**, or **config spec**. Some operations (examining well-defined objects, like metadata) do not require to mount the vobs or set the view. But one needs to do both operations (`mount`, `setview`) to be able to navigate the vob as a file system, otherwise it shows empty (the normal default state of the vob mount point).

```
$ ct pwv
Working directory view: ** NONE **
Set view: ** NONE **
$ ct lsvob /vob/myvob
/vob/myvob /vobstorage/myvob.vbs public
$ ls -la /vob/myvob
total 10
dr-xr-xr-x 2 root root 1024 May 7 22:31 .
drwxr-xr-x 126 root root 4096 May 5 10:53 ..
```

No view is set, `myvob` is not mounted and the vob mount directory shows empty.

```
$ ct mount /vob/myvob
$ ls -la /vob/myvob
total 10
dr-xr-xr-x 2 root root 1024 May 8 12:11 .
drwxr-xr-x 126 root root 4096 May 5 10:53 ..
```

Just mounting the vob without setting the view is not enough.

```
$ ct setview myview
$ ct pwv
Working directory view: ** NONE **
Set view: myview
$ ls -la /vob/myvob
total 18
dr-xr-xr-x 2 vobadm jgroup 1024 Oct 4 12:21 .
drwxr-xr-x 126 root root 4096 May 5 10:53 ..
drwxrwxr-x 3 joe jgroup 24 Aug 24 2009 dir1
drwxrwxr-x 4 smith jgroup 46 Feb 1 2010 dir2
```

Now the vob is mounted and the view is set: the vob content is displayed according to the view configuration.

Note that these examples are UNIX specific: on Windows, one cannot *set* a view (the command is not supported), one has to start it, and (typically) to map it to a drive letter, so that one would have to choose different operations to display related effects. It is however typical that Windows users find vobs unmounted, since starting ClearCase there doesn't mount the public ones, as it does on UNIX.

The bulk of the data is not stored *in* the databases, but in **storage pools**, directory hierarchies hosting various kinds of **containers** (the files with the actual data). The view and vob processes will actually produce and maintain containers matching the requirements of the use context, and hand these to the external tools (editors, compilers, etc), that need thus not be aware of the SCM system. This also ensures that the performance is not significantly affected by using files *from a vob*, since in fact, the files are in the end used from a standard storage. Finally, the processes mentioned above run, and the files are stored, on server hosts: view, vob, or storage servers.

We'll not be back to the containers (source, cleartext, DO), and to the storage in general (protections) until *Chapter 10*, *Administative Concerns*.

Deeper into views

A first and obvious virtual file system's characteristic is that it offers dynamic access to the repository: there is no need to download/unload the actual data when setting or changing the view configuration, as the dynamic view works as a filter selecting versions according to the view's configuration specification and displaying them under the vob mount point.

```
$ ct setview view1
$ ls -l /vob/myvob
drwxrwxr-x 3 joe jgroup 26 Sep 26 2006 dir1
drwxrwxr-x 3 joe jgroup 49 Sep 26 2006 dir2
drwxrwxr-x 4 joe jgroup 517 Apr 9 20:45 lost+found

$ ct setview view2
$ ls -l /vob/myvob
drwxrwxr-x 3 joe jgroup 24 Jun 17 2008 dir1
drwxrwxr-x 3 joe jgroup 46 Jun 17 2008 dir2
drwxrwxr-x 4 joe jgroup 517 Apr 9 20:45 lost+found
drwxrwxr-x 3 joe jgroup 23 Jun 17 2008 dir3
```

Why does the same vob directory content look different? - Because the configurations of `view1` and `view2` are different (see more on config spec below).

Versioning mechanism

Let's also mention that (dynamic) views support path extension syntaxes to access if need-be arbitrary versions (the ones not selected otherwise by the current view configuration):

- One such syntax (**view extended path**) involves explicitly referring to a view different than the *current* one (the one *set* previously)
- The other (**version extended path**) uses *decorations* to specify the versions of path elements. The use of such extensions is signaled by a `@@`, which enters the space of versions of the element, presented as a directory tree. This `@@` works as a door into the hidden space: it need not be mentioned again, and all the names beyond it are interpreted within the space
- They may be mixed

Let's notice that we already met a few such cases in the *Teaser* chapter.

The view extended path:

```
$ cd /view/joe/vob/perl/lib
```

Here the view tag is `joe`, and the vob tag is `/vob/perl`, and `/vob/perl/lib` is a directory within the vob.

The version extended path:

```
$ ls -ld /vob/perl/.@@/APP/lib/APP
```

Here the vob tag is `/vob/perl`, `/vob/perl.@@/APP` signifies a version of the `/vob/perl` root directory labeled with `APP` label, and it is followed by `lib/APP` denoting version of the lib subdirectory also labeled by the same label `APP`. Note that vob roots display an exceptional behavior in requiring there an explicit mention of the `.` directory, to which the label applies. This is not needed for other directories but the vob root.

And another variant of the version extended path:

```
$ cat /vob/apps/joe/t/Makefile@@/main/jn/1
```

Here the vob tag is `/vob/apps`, and `/main/jn/1` signifies version 1 of the branch `/main/jn` of the `/vob/apps/joe/t/Makefile` element. Technically, `jn` is *cascaded* from the `main` branch. Note that `main` is a predefined branch type, and that there exist one and only one main branch on every element (which may at most be *renamed* or *chtyped* to another name).

Let's mention one extreme, but at times useful, case: the *fully qualified path*. This completely bypasses the config spec, ensuring that the path will works in any context. Note that it requires a view, which must be started:

```
$ ls -ld /view/joe/vob/perl/.@@/APP/lib/APP
```

Of course, the default use of views aims at accessing these same versions as if they were the only ones, as if the **elements** were plain file system objects (files or directories):

```
$ ct setview myview
$ ls -ld /vob/perl/lib
drwxrwxr-x 3 joe jgroup 32 Sep 01 2009 lib

$ ls -l /vob/apps/joe/t/Makefile
-r-xr-xr-x 3 joe jgroup 496 Oct 05 2009 Makefile
```

So, in ClearCase virtual filesystem, a file name is common to a whole family of instances. These instances may be versions of an element, or, probably more interestingly, instances of a derived object. Also, as we have already seen in the examples above, the view-selected version, or actually *any* version of the element can be accessed by *any* standard operating system tools, and not only by ClearCase-specific ones:

```
$ ct setview myview
$ ct ls /vob/utils/hello
/vob/utils/hello@@/main/5 Rule: /main/LATEST
$ perl /vob/utils/hello
Hello, world!
```

We have executed the version of the perl script `hello`, which was selected by the `myview` view. The actual selected version is `/main/5`, which means version 5 on the branch main.

```
$ perl /vob/utils/hello@@/main/br1/2
!dlrow ,olleH
```

And now we have executed another version of the `hello` Perl script, which was not selected by the view by specifying it explicitly: `/main/br1/2`, which is version 2 on the branch `/main/br1`.

The ClearCase versioning mechanism is *transparent* to the end user and to operating system tools.

It is essential (thinking again of derived objects primarily) that this property extends to private data and repository data: the location in the virtual file system gives access to the set via an instance selected by the view, yet any version may be accessed explicitly using an extended notation (introduced with `@@`).

This is how the derived objects are versioned (note the slightly different syntax):

```
$ cd /vob/test
$ ls -l
-rw-r--r-- 1 joe jgroup 0 Apr 11 14:50 all.tests
-r--r--r-- 1 sara jgroup 135 Apr 11 14:50 Makefile
-r-xr-xr-x 1 sara jgroup 381 Apr 10 14:55 pathconv

$ ct lsdo all.tests
--04-11T15:52 "all.tests@@--04-11T15:52.3380"
--04-11T14:50 "all.tests@@--04-11T14:50.3375"
```

This lists all versions of the derived object `all.tests`.

The non-selected versions can be accessed directly using the same `@@` syntax as for the version-controlled elements:

```
$ ct des all.tests@@--04-11T15:52.3380
```

Views properties

Views have some properties: one of them (**text mode**) concerns the presentation of text, and the conversion of end-of-line sequences, that we mentioned in the last chapter. The typical application, in views intended for use with Windows tools, is to add the expected carriage returns to lines of files stored in UNIX mode. This occurs at *cleartext generation* (more on this in Chapter 10), even for read-only access (often from tools, including compilers). One can see that this is superior to the option of explicitly modifying the data at checkin (and checkout). The downside is to restrict the views once and for all to be exclusively used on either platform (otherwise, views may be tagged in two regions: in both Windows and UNIX, and thus shared, with some possible benefits). Obviously, and mostly for symmetry, other patterns of use are possible, including the default transparent mode: no conversion.

The commands to examine and change those properties are:

```
$ ct lsview -l -prop -full <view tag> | grep Text
```

where the output text signifies the following modes:

`Text mode: unix` - transparent (default)

`Text mode: msdos` - insert carriage returns (when presenting to the user the stored text, and remove them when storing)

`Text mode: strip_cr` - strip carriage returns

Note that these view's text mode properties cannot be modified after the view has been created (even with the `ct chview` command).

We'll leave vob creation (a somewhat rare event, even if users are encouraged to practice creating and maintaining private vobs of their own) for later, but let's mention the commands to create and remove views that should be used much more often and by everyone:

```
$ ct mkview -tag joe -stg -auto
$ ct rmview -tag joe
```

The first form implies the existence of pre-defined **view storage locations**, which frees the user from storage issues concerns (and thus disk space, and backup). Such pre-defined view storage locations can be created and listed by the following commands:

```
$ ct mkstg -view ...
$ ct lsstg -view
```

In case there are a few view storage locations defined for the same site, one of them is picked up at the view creation while using the `-stg -auto` option (the algorithm to select one location among others is documented in ***technote 1147041:*** for dynamic views, it supports one way to exclude some locations, favors local ones, and otherwise picks one at random—unfortunately not taking details such as the version of ClearCase on the server into consideration).

The three text modes mentioned above can be specified using the `-tmo` option (in the `ct mkview` command) with one of the following arguments: `transparent`, `insert_cr` or `strip_cr`.

The most important way in which the user affects her views, is however the value of her **umask** at the time of creation (note by the way that views are meant to be created by their actual user, and not by *root*). The umask is in UNIX a bit pattern used to restrict the access rights of files created, from a basic (`777`) pattern: *read/write/execute* (one bit for each), for *owner*, *group*, and *world*: altogether 9 bits. The value `777` (in octal) represents thus a full pattern: every bit set for every entity, since `7` is `4 + 2 + 1`. The umask value is subtracted from it and the result is used to set the view permissions. In practice, the disputed issue is to decide whether or not to grant write access to group, that is to choose between the values `002` and `022` for the umask: filter nothing (`0`) away from the owner, and *write* from "world" (`2`), but choose to treat one's collaborators along to the former or the latter model:

```
$ umask
022
# The view permissions would be then 755
$ ct mkview -tag test1 -stgloc -auto
...
It has the following rights:
User : joe: rwx
Group: jgroup : r-x
Other: : r-x
```

```
$ umask
002
# The view permissions would be then 775
$ ct mkview -tag test2 -stgloc -auto
$ ct lsview -l -prop test2 | egrep "Owner|Group|Other"
Owner: joe : rwx (all)
Group: jgroup : rwx (all)
Other: : r-x (read)
```

Note that the *execute* right is necessary on directories.

The view write permission allows the user to modify the state of the view, such as by setting the config spec, or by checking files out or in.

The current value of the umask still governs the permissions set to files (view-private).

ClearCase modifies the write access of elements according to whether they are checked in (read-only), or checked out (writable). In a read-only view, checkouts are forbidden. Note that one cannot create a read-only view by setting one's umask to `222`. One has to use the already mentioned `chview` tool.

The issue of the group write permission is met in practice in three cases:

- The one we just mentioned: the view creation. Users are typically afraid that views created with a permissive mask might be shared (usually not a good idea), but this also allows to fix problems during vacations (there is a range of options: checkin, uncheckout, unreserve, remove view, with different restrictions and applying to different contexts, including snapshot views and Windows; see Chapter 10 for more details), or to chase stale file handles in some but not all views.
- For winked in derived objects: DOs produced under a restrictive umask will be read-only for the other members of the group. They may still be winked in, and may still be deleted (depending on the access rights to the parent directory), but they cannot be overwritten. This means that builds will eventually fail (after having first succeeded and winked in some results): clearly a nuisance. This is fortunately easy to solve, by using the CCASE_BLD_UMASK makefile macro, thus overriding the view umask under clearmake.

- For directory elements: directories created with a restrictive umask will not be modifiable by others, who will not be able to create new elements, checkout existing ones without redirecting the data elsewhere, or create derived objects and even simple view-private files. This may seem like a mine field, as the problems are typically not detected at once. One may of course (with appropriate rights) change the protection of elements, but one may meet further issues in presence of MultiSite: protection events are subject to settings in the replica objects, and may be filtered. We'll touch this in due course (see *Chapter 11*, *MultiSite Administration*).

Registry, License, and even Shipping servers

The ClearCase installation is concentrated (apart for the data itself: vobs and views). In UNIX, these are found under two directories: `/opt/rational/clearcase`, and `/var/adm/rational/clearcase`; in Windows, both parts are by default under `\Program Files\Rational\ClearCase`, the latter concentrated as a `var` hierarchy below this root. The former part is considered read-only and not preserved during upgrades (installations). Note that there is also a `common` tree (`/opt/rational/common` in UNIX and `C:\Program Files\Rational\Common` in Windows) with some read-only non-preserved contents.

View and vobs are **registered** in a central location, a directory `/var/adm/rational/clearcase/rgy` on a dedicated server (optionally backed up on another host), in a few flat text files collectively known as the registry. They contain records of the *objects* themselves and of the *tags* used to access them. The realm of a single registry maps to the span of **regions** (also recorded in the registry), and (possibly with subtle differences, rather to be avoided) with this of **sites** (in MultiSite context). The relevant commands are:

```
$ ct hostinfo -l
$ ct lsregion
```

One typical use of regions (see our *Teaser* chapter), is to support Windows and UNIX environments over the same vobs (and even optionally views). For reasons explained in the previous chapter (the different path naming schemes), a vob must be tagged in both in different ways, and regions make this possible. Each client host (even the registry host itself) can only belong to a single region, and can only access vobs and view with tags in this region. This is configured in a `config/rgy_region.conf` text file, under the `/var` hierarchy (not to be confused with the `rgy/regions` file, on the registry host, in which the names of the various regions are stored). But a ClearCase client is allowed to query the tags used in other regions, and to check the identity of a registry object by using its **uuid**: universally unique identifier.

Another function is typically localized on a separate server: holding the license database. ClearCase supports two alternative licensing schemes, functionally intended to be equivalent (*floating* licenses, per simultaneous user): a traditional, proprietary one, based on a simple flat file, and a more recent one, using the *de facto* standard FLEXlm, and supported with tools to provide duplication, backup, and monitoring (these functionalities may be achieved with the former, but only using ad hoc tools).

One last kind of servers (hosts) may occasionally be met: shipping servers. Their function relates to MultiSite, and whereas it is made necessary for crossing firewalls (with restrictions in the opened port ranges), shipping servers may also be used for other purposes (for example, buffering or dispatching between multiple vob servers on a site). See thus Chapter 11.

Config specs

ClearCase offers (or should we say *offered*, from the point of view of the UCM extension, which, as told earlier, we'll ignore for now) a mechanism to select systematically and implicitly consistent sets of versions for multiple elements, across many vobs. This mechanism is called **configuration specification**, and is a property of views. Views are thus considered as the ClearCase implementation of the abstract concept of **software configuration**, already mentioned.

The specification happens through an ordered list of rules evaluated in turn for every accessible element until one matches. Only the first match matters. Note that it is often a different rule which matches in the context of different elements. A rule selects a version (out of many) of an element.

To check which rule actually applied for a given element, use the `ls` command:

```
$ ct ls -d foo
```

The `-d` flag in the above example is useful if `foo` is a directory. What matters, for build avoidance purposes, is the result of the selection, the version selected, and not the rule used (hence, not the config spec). Config specs should be considered merely as convenience tools. However, to remain convenient, they have to be well understood, and thus to be kept simple.

It is a common mistake (sanctified by UCM) to follow the slippery slope of making them too complex, by piling up many ad hoc rules specific to small contexts, and finally to 'free the user from this complexity' by generating them: one has once again thrown the baby with the bath water.

The specification ought to be consistent, and this consistency is best achieved by limiting the number of rules, and thus augmenting their scope: less is more. Then, there is no reason anymore to generate the config specs: the main result achieved by doing this is that the user doesn't feel that consistency is her concern.

A set of simple commands is provided to operate on config specs as text. Here are the man pages:

```
$ ct apropos 'config spec'
catcs Displays the config spec of a view
edcs  Edits the config spec of a view
setcs Sets the config spec of a view
```

In addition to these, the man page documenting the syntax is `config_spec` (`ct man config_spec`).

The most typical example of config specs is:

```
element * CHECKEDOUT
element * /main/LATEST
```

This is called the **default config spec**, and may be restored in a given view, with the command:

```
$ ct setcs -default
```

Beware however:

- The previous config spec (in place before running this command) is lost: ClearCase does nothing to version it, or back it up
- This will actually set the config spec stored on the local host as `/opt/rational/clearcase/default_config_spec` (one might consider it a breach of the principle of least surprise to change it, but it may still be the case...)

Seldom (but see below for one case) will you *not* want to select versions you'd have checked out yourself (what the first line in the default config spec says).

The second line will however on the contrary seldom be sufficient for your needs. It is valid only as a default, ensuring that you'll access at least one version of every element you can reach: there is by construction always a `main` branch in every element (unless you have changed its type, or renamed the main type in the vob—again a questionable move if you intend not to surprise your users), and there will always be a `LATEST` version on this branch, even if it may be version 0, which is an empty one. Note however the more important restriction at the end of the above

sentence: "every element *you can reach*". An element is reachable only from a certain path (or more, in case of hard links); it is not uncommon that one of the directories on this path will be selected in a version not containing (yet or anymore) the next name in the path chain: some elements may thus not be reachable at all, in a the current view, using the current config spec!

Let's rather consider the following, more useful, config spec:

```
element * CHECKEDOUT
element * .../branch/LATEST
mkbranch branch
element * LABEL
element * /main/0
```

It anticipates on *Chapter 6, Primary Metadata,* for the concepts of labels (here in UPPERCASE) and branches (lowercase). The first line selects the user's checked out versions.

The second line assumes the user is working in an own branch (*`branch`*), and her check-ins are always made to a distinct *workspace* in the **version tree**, in which she has write access. This device is meant to support parallel development. By this line in the config spec, the check-outs will also be done from the `branch` in case it already exists for the element.

The third line specifies the beginning of the "branching off" section: that is, for those elements that do not have the branch `branch` yet, it will be created from the element version labeled by the label `LABEL`, as specified on line four. Typically, such a label denotes a code baseline, for example, some stable release, which you'd like to use as a base for the further development.

The last line is mainly for creating new elements: any element has the default version 0 on the main branch (`/main/0`), and it will be branched off to the version 1 on the branch `branch` right away by this rule. The same rule will however also catch the case of elements that were not part of the baseline (that is, not labeled by `LABEL`), ensuring that no discontinuity will arise from taking existing versions, which might introduce conflicts with other files.

As one may see, the main concern is to ensure continuity of work via consistency of the rules: avoid that files might appear or disappear, or that the versions selected might differ in unexpected ways, as a result of one's actions (checking out or in, creating a branch, applying a label to a version currently selected).

```
element * LABEL -nocheckout
```

This last example is one in which own checkouts are excluded (both producing any, and selecting any which would have been produced in the same view before setting this config spec): clearly this is suitable for a read-only access of existing versions, but possibly also for building, hence for creating derived objects, and thus "writing" in this sense. This kind of setup is useful to reproduce a configuration recorded as one single label type; maybe for validating it, making sure that nothing critical was forgotten (such as environment variables, view private files, or files outside the vobs, which would affect the build one would want to replay).

There are several other functionalities available to edit config specs (see the `config_spec` man page):

- **Scope patterns**. We recommended to make rules as general as possible. There may however be some restrictions to the extent of this advice, such as the fact that metadata (and thus label and branch types) is vob local. Some rules may thus make sense only in the context of certain vobs, but not of others. Scope patterns extend (or rather restrict) the `*` character we have used until now. Scope rules based on vob tags pose an interoperability problem: the vob tags are only guaranteed to be valid in the current region. The solution is to use a syntax mentioning the vob family uuid (with the tag as a useful comment):

  ```
  element "[899d17d4.769311d9.9735.00:01:83:10:fe:64=############
                                                  /vob/apps]/..." LABEL
  ```

  There is another use: work around the exclusion principle, and allow to select two versions of the same element in the same view. This is clearly an exception, probably temporary, but it is essential to be able to allow it: the user creating a new element instead of a new version (*evil twin*) would even be worse, and cannot easily be prevented. The idiom is to use a hard link and a different name (or path), and to add an exceptional rule for the alternate name.

  ```
  element libfoo.so.26 FOO_26
  element * LABEL
  ```

  This example assumes that the current version of `libfoo.so` is selected by the generic rule with `LABEL`, and preempted in the special case of the `libfoo.so.26` name (a hard link of the `libfoo.so` element).

- **Include rule**. It is possible to share config spec fragments, by including them into one's config spec. This is obviously the mark of a suspicious complexity, but again, may be justified in certain contexts. Note a non-intuitive problem when the included fragments are modified: the changes do not propagate to the user config spec until she re-compiles it, which happens with the command:

  ```
  $ ct setcs -current
  ```

In doubt, one may enquire when was the config spec last compiled with:

```
$ ct lsview -l -prop -full
```

grepping for the `Last config spec update` row, or by looking at the time stamp of the `.compiled_spec` file in the view storage directory.

A second problem may surprise users: the versions of the files actually included are all evaluated using the config spec of the current view, from where one runs the `setcs` or the `setview:` any rules found earlier in the same config spec do not affect this selection! This is documented, simple to understand in theory, but counter-intuitive in practice.

- **Time clauses**. One may try to protect oneself against changes that happened after a certain time stamp, and which would creep in via the `LATEST` builtin pseudo-label type: in effect, to freeze the time at a date in the past. Let's bring your attention to a gotcha under MultiSite: one may learn new things about one's past! This may happen via importing a delayed packet. The technique of using time clauses, which seems careful, is thus not a panacea. We'll come back to MultiSite concerns, and how to deal with them safely in Chapter 5.
- **Block rules**.The last *goody* in the somewhat too rich config spec syntax is an exception to the too simple description with which we opened this paragraph: *block* rules—meta rules that apply onto other rules, bracketed within a block. This may be used for `mkbranch` and `time` clauses, and, let's admit it, rather helps introducing clarity than the other way around.

This review of config specs did not intend to be exhaustive: for this, we refer our reader to the man page. What we do want to stress is that config specs may be written, and even read by users, unless one errs on the side of excessive complexity. The need to version config specs (either by use of the include clause, or by using the `setcs` command to set one's config spec from a file) should be felt as a red flag. The opposite strategy is clearly wiser: design one's config spec so that it may be stable, and remain small and simple. This may be achieved by using **floating labels** (see Chapter 6).

Summary

This concluded our review of the basic concepts:

- The virtual file system metaphor
- Auditing and winkin
- Vobs and views
- Registry, license and other servers
- Config specs

Remember that we also opened up a perspective on thinking of SCM at large, which we'll nourish during our travel through ClearCase.

The scenes are ready? The play may start!

3
Build Auditing and Avoidance

In this chapter, we'll focus on **derived objects**, and on producing and managing them. We believe that it is what makes *the* competitive advantage of ClearCase, and this makes this chapter, despite its apparent technicality, the most important one. Contrary to the tradition of version control, which still dominates the mindset of most of its competitors, ClearCase allows to *manage directly* what is most valuable to the end user—the products and not the ingredients. This applies to many realms and to the various stages of production: building executables out of *source code* as a particular case, but also, for example, packaging the software, testing, producing documentation, etc. We'll analyze how the user and the tool can collaborate to make management possible and effective.

In this chapter, we are going to cover the following topics:

- Configuration records: how to produce them and to use them; how to optimize makefiles, avoiding various pitfalls, to not only reduce build times, but most importantly, to maximize the stability and the sharing of the derived objects
- Validation: how to use the config records for analysis purposes; how to examine the records of individual derived objects, to reproduce the builds and to assert the quality of the reproduction
- Tying some knots: how to deal with some specific requirements, questioning some "best practices"; we'll finally mention leveraging these techniques to a wider perspective

Configuration records

ClearCase offers to record the software configuration actually used during build transactions. But all records are not equally useful: it depends on the user to help ClearCase to produce records that may become the first source of information concerning the development.

Flat or hierarchical: clearaudit vs. clearmake

To introduce configuration records, the simplest is to first take the **clearaudit** tool. This performs the basic function of recording (**auditing**) the files opened during the hierarchical execution of a process and of the subprocesses directly or indirectly run by it. It does this however only on files located in a vob. Let's thus suppose we are in a view, and under a vob mount directory. We start a clearaudit session, concatenate a couple of files (to `/dev/null`), create one new empty `tag` file, and exit. Then we look at the result as recorded in the **config record** of the `tag` file.

```
$ cd /vob/test/caudit
$ clearaudit
$ cat VobPathConv.pm >/dev/null
$ cat /etc/passwd >/dev/null
$ touch tag
$ exit
$ ct catcr tag
Derived object: /vob/test/caudit/tag@@--05-15T06:19.20613
Target ClearAudit_Shell built by mg.user
Host "sartre" running Linux 2.6.18-128.el5
Reference Time 2010-05-15T06:18:35+01:00, this audit started ############
                                                2010-05-15T06:18:35+01:00
View was vue.fiction.com:/views/mg/mg.vws
Initial working directory was sartre:/vob/test/caudit
----------------------------
MVFS objects:
----------------------------
/vob/perl/lib/site-lib/5.10.0/ClearCase/VobPathConv.pm@@/main/imp/4
<2009-06-21T12:55:49+01:00>
/vob/test/caudit/tag@@--05-15T06:19.20613
```

Comparing to what we got in the *Teaser* chapter, let's note that this case is simpler. In *Teaser*, we did produce a file, `all.tests`, to which the config record got attached; we practiced using the `ct catcr` command. Now again, we need to create one such file, unless our rule modifies one suitable file (in a vob), in which case the config record already gets attached to it and we do not need an extra `tag` file.

We note that `/etc/passwd` did not get recorded in the `tag` config record. It is not a vob object, so recording it (as we did for the `builtin.mk` makefile component in the *Teaser* chapter) would require an explicit mechanism (in the *Teaser*, it was a special makefile macro, which will look into later).

There is thus no *non-MVFS objects* section, and neither any *Build Script* one. Something more important but less obvious—this record is *flat*. Note that the `VobPathConv.pm` module was not mentioned in the derived object `all.tests` config record we have seen in the *Teaser* chapter; and yet it *was* recorded. This is because the record produced by **clearmake** was actually *hierarchical* (which we did not display), and it is `pathconv.cr`, which was referenced explicitly in the `all.tests` config record. This is the main difference between clearaudit and clearmake config records: there are no hidden parts in this clearaudit record; it is all there, flat.

Maybe you wonder where the config record itself is actually stored? The important point here is that although the `tag` file (even if empty) is, at least at this point, stored in the view storage (it is view private data—or lack thereof), the config record is already public, and therefore stored in the vob database (you can detect its existence with the **countdb** utility, which we'll look at in *Chapter 10*, *Administrative Concerns*).

What matters next for us is to understand that audited files may be thought of as **dependencies**, and to see that this makes even more sense in presence of a multi-version database than of a plain filesystem. We shall leave outside our scope how one might compensate the absence of such a database, by mapping the recorded file names to versions in an external repository, in an attempt to decouple the function of clearmake from ClearCase.

The challenge is to use a full list of dependencies, together with the script used to produce a given derived object, as a means to unequivocally **identify** it, and by identification, to understand the most basic function of Software Configuration Management (which one remembers, was once borne as some kind of accounting). This identification is implicit (provided by the system), and operational (the system may use it efficiently).

Let's note that it sets a few requirements:

- The production must be deterministic and causal, grounded on file objects (random behaviors, or dependence on time are excluded)
- All the dependencies must be themselves identified (in the context of ClearCase, files must be in vobs or explicitly designated, elements must be checked in)
- The execution of the script itself must be atomic (closed over oneself: interactions with other processes will cause problems)

Note that the concept of dependency has a precise meaning in the context of makefiles (where it is sometimes also termed as prerequisite).

Makefiles will also allow us, as mentioned earlier, to produce hierarchical config records. The interest lies in the reuse of components, and in their increased stability: coarse components become dependency bottlenecks. Fine granularity, on the contrary, leads to fewer dependencies and therefore to better stability expectations.

Makefile syntaxes—compatibility modes

Makefiles are an old technology, once again popularized by UNIX. It also has slightly diverged in the various environments, and therefore comes in several variants. clearmake has historically taken three parallel strategies:

- Support a minimum common syntax
- Support a compatibility mode with the local platform syntax (Solaris, HP-UX...)
- Support the syntax of **GNU make**, as this soon became available everywhere

This third strategy has clearly come as the winner, and it is the one we recommend and will use.

GNU make has evolved over the years, and clearmake has followed. It offers the richest set of built-in functions, which allows to avoid spawning sub-shells, and thus: depend on the environment, require extra complexity, and incur performance penalties.

It also supports two flavors of variables: *recursively* or *simply* expanding, with the latter offering again opportunities for simplifications and performance improvements in many cases.

The GNU compatibility mode is also the only one supported on Windows. It is invoked on the command line by using the `-C gnu` option, but may also be driven by an environment variable.

We did *not* use it in our *Teaser*, for simplification, as we would have had to explain the presence of one extra makefile component in the config record.

There is a large body of documentation and of expertise about GNU make. This is a valuable asset. Also, it is wise to retain compatibility with the GNU make tool, that is, to ensure that one is able to build using it. The benefit may be to be able to get a wider assistance while tracing difficult problems, or to have a second tool when doubting the first one. Remember however that whatever the similarities, essential differences remain. Not all advice concerning build systems tuned for GNU make is relevant or even valid when applied to clearmake. One good example is the historical controversy concerning a famous paper by Peter Miller: ***Recursive Make Considered Harmful***. While the problems described in it are real, the proposed solution, monolithic top-down build systems, is not in line with our goal of optimizing **winkin**, which obviously could not be a requirement for the author. We'll leave it to the reader to check that our solutions do address Miller's issues, using a radically different strategy.

A first case with clearmake

Our goal will be to describe ways to build hierarchical subsystems, in a way optimizing the stability of the various components and maximizing winkin. We shall grow this system from a first simple one. We'll take our example in C code, but keep it as trivial as possible**:** we'll restrict our functionality to the traditional *Hello World* greeting.

The most natural way to compose a system from components written in C is by creating and using shared libraries.

So, here is a subsystem producing a shared library, containing a function (quite redundant...) to print a single string. The source code is comprised of one header file, and one source file; `wstr.h`:

```
void wstr(const char* const str);
```

and `wstr.c`:

```
#include <stdio.h>
#include "wstr.h"

void wstr(const char* const str) {
  printf("%s\n", str);
}
```

We are, however, more interested in the Makefile:

```
CCASE_BLD_UMASK = 2
SRC = wstr.c
OBJ = $(SRC:.c=.o)
LIB = libwstr.so
```

```
BIN = /vob/tools/bin
CC = $(BIN)/gcc
LD = $(BIN)/ld
CFLAGS = -Wall -fpic
.MAKEFILES_IN_CONFIG_REC: $(LIB)

all: $(LIB)

nolib \
$(LIB): $(OBJ)
        $(LD) -shared -o $@ $(OBJ) -lc
```

Here, we recognize two sections:

- Macro definitions: a name and a value assigned to it
- Rules: a *target*, followed by *dependencies*, and optionally below, indented with TAB characters, with the rule proper, i.e. with a shell script supposed to produce the target

Using clearmake, one need not explicitly name *static* dependencies to elements (as e.g. to `wstr.h` in the example above), as they will be discovered and checked automatically. On the contrary, dependencies on other derived objects are mandatory. They are termed in ClearCase documentation (notably the excellent ***Guide to Building Software***, available both from the IBM website, but only for version 7.0.0 – at least at the time of writing, and as part of the distribution) **build order dependencies**, and this term describes well their essential function, driving the navigation by the make tool of the dependency tree.

Maintaining this list will be the major focus of our attention, and the main tool to ensure the quality of the produced config records.

The rule script is idiosyncratic to our environment: the `-shared` flag and the explicit use of the `-lc` standard C library, as well as the mandated use of the linker (instead of the main *gcc* driver) would be different in some other environment and with other tools. The `-o` flag to specify the output, as well as the `$@` make macro to name the target are more generic.

Let's note that this makefile relies upon the definition of *implicit* (or *pattern*) rules, needed here to perform the compilation proper, i.e. to produce the object file, `wstr.o`, from the source file, `wstr.c`. One advantage of using the C language is that this rule is standard. It uses the `CC` and `CFLAGS` macros, respectively for the compiler and compilation flags. Some of these definitions could (should) be extracted into common makefile components, to be included by similar subsystems:

- `OBJ`, as it is completely generic, and computed from the value of `SRC`.

- `CC` and `LD`: we are using here the Gnu C compiler and linker. These are found from a vob.
- `CFLAGS` is set here to produce *position independent code*, i.e. is only suitable for objects packaged in shared libraries.
- `.MAKEFILES_IN_CONFIG_REC` is technically a special target, although it behaves more like a macro. We met this already in our *Teaser* makefile (found as code from the publisher web site). We'll devote the next paragraph to it. For the time being, and assuming only one library per makefile and one makefile per directory, one could extract it as well. One could also work around these assumptions, but this presents tradeoffs in terms of simplicity, and we'll not do it here.
- `$(LIB)` is also fully generic, depending only on macros. Note the `nolib` guard, ensuring syntactic correctness of the makefile in the case the macro would not be defined.
- Next, we could decide that all the C files found in our directory are actually meant to be linked to our library. This requires that we move away test and any extra files, which is probably advisable anyway, and will help others to make sense of our code. Assuming this, one might use the `wildcard` GNU make function to yield a generic definition for `SRC` in our standard makefile component.

These extractions would leave us with a local makefile such as:

```
STDDIR = /vob/bld/std
LIB = libwstr.so
include $(STDDIR)/stdlibdefs.mk

all: $(LIB)

include $(STDDIR)/stdrules.mk
```

We would still have to define the path to the standard makefile components, `STDDIR`, and the name of the library produced locally, `LIB`. And we would have to define the default target (the first non-special target lexically met), `all`, which is actually equivalent to its dependencies, as having no rule of its own. This target is the one built when invoking clearmake without arguments.

The purpose of these extractions should be clear, although it is twofold:

- Avoiding accidental differences, such what could happen if defining the same list several times, with its tokens in different order, in the makefile: making sure that any information is only represented once.
- Managing by differences, that is, raising the signal/noise value of the local makefile, so that errors would be easier to spot and to fix; only essential information should be presented, if possible.

Our case is still too simple: something we miss here, because we did not meet this need, is to include directories to resolve precompiler directives. These would typically arise from indirect dependencies upon other subsystems. The header files should be included during the compilation of the local source files, but the related libraries should be linked in once (and only once) in the resulting executable. This is an addition we should make into the `CFLAGS` variable, which we moved now into a shared makefile component. This addition may in fact be a boilerplate, and computed from a `LIBS` macro which is only empty in our first simple case. The only problem is that our value of `CFLAGS` was specific to shared libraries (the position independent code), whereas this generic addition could in fact be more general, i.e. implemented in a `stddefs.mk` component, which would itself be included from our `stdlibdefs.mk` one. The point is that we'd like to *modify* (to append to) the value of the generic `CFLAGS`. This is an indication for switching to the *directly expanded* flavor of make macros, which we alluded to in the paragraph on compatibility mode, since modifying a *recursively expanded* macro leads to an infinite regress.

This yields the following state of the standard makefile components, `stdlibdefs.mk`:

```
include $(STDDIR)/stddefs.mk
CFLAGS := $(CFLAGS) -fpic

.MAKEFILES_IN_CONFIG_REC: $(LIB)
```

`stdpgmdefs.mk`:

```
include $(STDDIR)/stddefs.mk
.MAKEFILES_IN_CONFIG_REC: $(PGM)
```

and `stddefs.mk`:

```
CCASE_BLD_UMASK := 2
BIN := /vob/tools/bin
CC := $(BIN)/gcc
LD := $(BIN)/ld
SRC := $(wildcard *.c)
OBJ := $(SRC:.c=.o)
LDIRS := $(patsubst %/,%,$(dir $(LIBS)))
LDIR := $(addprefix -L, $(LDIRS))
LNAM := $(addprefix -l, $(patsubst lib%.so,%,$(notdir $(LIBS))))
LINC := $(addprefix -I, $(LDIRS))
CFLAGS := -Wall $(LINC)
```

Note the assignment syntax: this is now a direct assignment and it is only evaluated once, which allows to use the previous value of the macro without fearing the infinite regress.

We changed all the assignments to be direct, with the consequence that their lexical order now matters.

We used GNU make functions liberally, to build lists of options for the tools, from the same initial values. Note that `LIB` and `LIBS` (which we do not actually use yet) are now the only ones under the responsibility of the user.

And we need the following rules (`stdrules.mk`):

```
nopgm \
$(PGM): $(OBJ) $(LIBS)
        $(CC) -o $@ $(OBJ) $(LDIR) $(LNAM)

nolibs \
$(LIBS):
        @cd $(dir $@); $(MAKE)

nolib \
$(LIB): $(OBJ) $(LIBS)
        $(LD) -shared -o $@ $(OBJ) -lc
```

From this file, we only needed the `$(LIB)` rule so far, to which we just added a dependency to potential libraries it would depend upon. This macro being empty in this context does neither affect our build, nor result in any include flag in the compiler options.

But it is time now to turn to the next step, and to *use* this library we just built, precisely via this mechanism. We shall thus use the two additional rules presented right above.

Recording the makefiles

The `.MAKEFILES_IN_CONFIG_REC` special target does what its name says—recording the makefiles in the config record of the objects named as its dependencies. The makefiles are needed in order to reproduce a build, so that it is convenient to record them. However, every target typically requires only a small subset of the information makefile contains— the rule associated with it, with all the macro expanded. If this rule doesn't match textually the script used for a next invocation, clearmake will assume that a different recipe produces another cake and will run it.

But then why *not* to record them systematically? And why using a special mechanism? The answer is simple: makefiles tend to constitute dependency bottlenecks, that is, to constitute a dependency of each and every of the objects they are meant to build and are therefore very volatile. Makefile might be subject to change every time new objects are added. They share this property with directories, and actually with most other containers (such as archives or packages from where to pull out whole installations).

This is why makefiles deserve a special treatment. Like directories, they will (but only when using this special target) get recorded, but will be ignored for dependency matching purposes. Although ignored, of course as file versions and not as expanded scripts: these will still have to match as described above.

Note that in our examples, we do not record makefiles in all the config records, but only for programs and libraries.

As a side remark, recording makefiles with the built-in rules and definitions (for GNU compatibility mode they are the following files under the ClearCase installation directory `/opt/rational/clearcase`: `etc/gnubuiltin.mk` and `etc/gnubuiltinvars.mk`) in this way gives us a good example of *explicit* recording of files outside any vob—the full paths and timestamps are recorded. Of course, all the other makefiles, found in the vobs, are recorded as well, in the *MVFS objects* section of the config record:

```
$ ct catcr hello
Derived object: /vob/apps/hello/helloc/hello@@--05-23T13:15.178584

----------------------------
MVFS objects:
----------------------------
/vob/apps/hello/helloc/Makefile@@/main/3 <2010-05-23T13:15:53+03:00>
/vob/apps/hello/helloc/hello@@--05-23T13:15.178584
/vob/apps/hello/helloc/hello.o@@--05-23T13:15.178583
/vob/bld/std/stddefs.mk@@/main/1 <2010-05-21T01:19:15+03:00>
/vob/bld/std/stdpgmdefs.mk@@/main/1 <2010-05-23T13:10:09+03:00>
/vob/bld/std/stdrules.mk@@/main/1 <2010-05-21T01:19:17+03:00>
/vob/apps/hello/wstr/libwstr.so@@--05-23T12:39.178560
----------------------------
non-MVFS objects:
----------------------------
/opt/rational/clearcase/etc/gnubuiltin.mk <2008-03-07T02:26:59+02:00>
/opt/rational/clearcase/etc/gnubuiltinvars.mk <1998-12-22T02:47:15+02:00>
```

Using remote subsystems

Let's note that our library does not depend so far upon its client code (`hello.c`), and it is good so! It is purely offered. This is achieved by not defining the library and the client code build together in, for example, some common bulky "project" makefile. We shall thus use it from a different, remote directory. We should, in fact, keep the dependency as it is—one way—and avoid creating a top-down architecture that would preclude later reuse of valuable components in other contexts than the ones in which they were initially developed.

Our code will still be trivial, `hello.c`:

```
#include "wstr.h"

int main() {
 wstr("Hello World");
 return 0;
}
```

Again, we'll focus our interest onto the `Makefile`:

```
PGM := hello
LIBS := /vob/apps/hello/wstr/libwstr.so
STDDIR := /vob/bld/std
include $(STDDIR)/stdpgmdefs.mk

all: $(PGM)

include $(STDDIR)/stdrules.mk
```

This should strike as very similar to the former, partly thanks to the extraction of shared makefile components.

We note the use of full and not relative paths. This answers the concern of avoiding to propagate a dependency on the directory of invocation, into the name of the invoked target—being fully qualified, the target name will be the same, independently from where it is being invoked. This concern anticipates on reuse from other possible directories.

Next, we'll notice our using the same `stdrules.mk` file as previously, albeit in it, the two rules we did not use so far.

The `$(PGM)` rule in itself is simple—it links its arguments into an output named as the target (using the `-o` option and the `$@` macro, in the same way as our earlier `$(LIB)` rule). It uses now our `$(LDIR)` and `$(LNAM)` macros. These macros act in a similar way upon `$(LIBS)`, from which they extract lists of directories and filenames respectively, that they format then into options as required by the linker syntax, prefixing the directories with `-L` and the short library identifiers with `-l`.

Its dependency list is what interests us now: or more precisely, not the `$(OBJ)` part, which is again common with the previous case, but the `$(LIBS)` one. `$(LIBS)` is a list , every item of which is produced by our second rule. It is reduced in our example to just one item: `/vob/apps/hello/wstr`. Every call of the rule performs a remote (also named *recursive*) build invocation, i.e. it switches directory to the parent of a library and explicitly invokes `$(MAKE)`.

This will, in fact, result in using our previous case! Here is the transcript:

```
$ clearmake -C gnu
 /vob/tools/bin/gcc -Wall -I/vob/apps/hello/wstr -c hello.c -o hello.o

clearmake[1]: Entering directory `/vob/apps/hello/wstr'
 /vob/tools/bin/gcc -Wall -fpic -c wstr.c -o wstr.o

 /vob/tools/bin/ld -shared -o libwstr.so wstr.o -lc

clearmake[1]: Leaving directory `/vob/apps/hello/wstr'

 /vob/tools/bin/gcc -o hello hello.o -L/vob/apps/hello/wstr -lwstr
```

So, great satisfaction: we made it! Our macros expanded as expected, and our result (hello) works according to specification (note that we have to tell the system where to look for the shared library, using the standard shell macro `LD_LIBRARY_PATH`, when executing it):

```
$ export LD_LIBRARY_PATH=/vob/apps/hello/wstr
$ ./hello
Hello World
```

But, let's try to build again, now with the verbose flag:

```
$ clearmake -C gnu -v
Cannot reuse "/vob/apps/hello/wstr/libwstr.so" - build script mismatch
<<< current build script
>>> "/vob/apps/hello/wstr/libwstr.so" build script
***************
1c1
< /vob/tools/bin/ld -shared -o libwstr.so wstr.o -lc
---
> @cd /vob/apps/hello/wstr/; clearmake
***************

======== Rebuilding "/vob/apps/hello/wstr/libwstr.so" ========
clearmake[1]: Entering directory `/vob/apps/hello/wstr'
`all' is up to date.
clearmake[1]: Leaving directory `/vob/apps/hello/wstr'
============================================================

`all' is up to date.
```

We built the library, but had to rebuild it the next time!? This was not the intention, as nothing has changed that would justify rebuilding. Remember that this may be costly, but also ruins our identification goals.

What is the problem? The transcript tells it very clearly: we have two different rules to produce the `libwstr.so` library—the local one and the remote one. Both concern the same target name.

Remote dependencies

A first idea would be to discriminate between the two target names—the local one (the library name), and the remote one which could be the same with a postfix, e.g. `_build`. The `$(LIBS:=_build)` rule would, however, become a *pseudotarget* in the `stdrules.mk` makefile:

```
nopgm \
$(PGM): $(OBJ) $(LIBS:=_build)
        $(CC) -o $@ $(OBJ) $(LDIR) $(LNAM)

nolibs \
$(LIBS:=_build):
        @cd $(dir $@); $(MAKE)

nolib \
$(LIB): $(OBJ) $(LIBS:=_build)
        $(LD) -shared -o $@ $(OBJ) -lc
```

And it would force the *actual* rebuild of any target depending upon it, with the following message in the verbose build output:

```
Must rebuild "hello" - due to rebuild of subtarget ######################
                                  "/vob/apps/hello/wstr/libwstr.so_build"

$ cd helloc
$ clearmake -C gnu -v
...
Must rebuild "hello" - due to rebuild of subtarget ######################
                                  "/vob/apps/hello/wstr/libwstr.so_build"

======== Rebuilding "hello" ========
 gcc -o hello hello.o -L/vob/apps/hello/wstr -lwstr
Will store derived object "/vob/apps/hello/helloc/hello"
============================================================
```

The cure might seem to be to `touch` such a file (`libwstr.so_build`) at the end of the remote rule (`$(LIBS:=_build)`), but this wouldn't work either. This is because now the target would not itself have **build order dependencies**, and thus, in effect, hide any changes to the library `libwstr.so`, i.e. such changes that would *not* cause the rebuild of the `hello` application.

This appeals thus for a more radical fix: using a layer of pseudotargets, avoiding to depend upon them, but carefully collecting the resulting records.

By "collecting the records", we mean opening the files to which records are attached in the context of a rule and producing a new *tag* file.

The first part, we'll do by reading one line (a minimal amount, maybe not so small in the case of binaries; GNU head has an option to read one byte) with the `head` utility, sending it to `/dev/null`. We shall make a `STAT` macro for this purpose. The second part, we'll do by removing and then touching a tag file (in order to avoid carrying forward dependencies of the previous version).

The changes affect all the shared makefile components. In fact, since we can concentrate the config recording to tag files, and decide to have one per directory (for simplification), with a standard name of `build.tag`, we can move the "special target" (`build.tag`) together with the other targets to `stdrules.mk` and get rid completely of `stdpgmdefs.mk`. The new makefile components are now `stddefs.mk`, `stdrules.mk` and `stdlibdefs.mk`.

`stddefs.mk`:

```
CCASE_BLD_UMASK := 2
BIN := /vob/tools/bin
CC := $(BIN)/gcc
LD := $(BIN)/ld
SRC := $(wildcard *.c)
OBJ := $(SRC:.c=.o)
LDIRS := $(patsubst %/, %, $(dir $(LIBS)))
LDIR := $(addprefix -L, $(LDIRS))
LNAM := $(addprefix -l, $(patsubst lib%.so, %, $(notdir $(LIBS))))
LINC := $(addprefix -I, $(LDIRS))
LTAGS := $(addsuffix build.tag, $(dir $(LIBS)))
CFLAGS := -Wall $(LINC)
STAT := head -1 > /dev/null
RECTAG := rm -f build.tag; touch build.tag
```

The new `$(LTAGS)` macro lists the `build.tag` files in all the `$(LIBS)` directories.

`stdrules.mk`:

```
.MAKEFILES_IN_CONFIG_REC: build.tag

build_pgm: $(LIBS:=_build)
 @$(MAKE) TGT=$(PGM) build.tag

build_lib: $(LIBS:=_build)
 @$(MAKE) TGT=$(LIB) build.tag

nolibs \
$(LIBS:=_build):
 @cd $(dir $@); $(MAKE) build_lib

nopgm \
$(PGM): $(OBJ)
 $(CC) -o $(PGM) $(OBJ) $(LDIR) $(LNAM)

build.tag: $(TGT)
 @if [ -z "$(TGT)" ]; then $(MAKE) -f $(MAKEFILE); fi; \
 $(STAT) $(TGT) $(LTAGS); \
 $(RECTAG)

nolib \
$(LIB): $(OBJ)
 $(LD) -shared -o $@ $(OBJ) -lc
```

We set `$(TGT)` on the invocation command lines in order to parameterize the generic `build.tag` target.

`stdlibdefs.mk`:

```
include $(STDDIR)/stddefs.mk
CFLAGS := $(CFLAGS) -fpic
```

The `helloc/Makefile` gets slightly modified accordingly (`$PGM` is replaced by `build_pgm` as the `all` target dependency, i.e. in fact as its alias).

`helloc/Makefile`:

```
PGM := hello
LIBS := /vob/apps/hello/wstr/libwstr.so
STDDIR := /vob/bld/std
include $(STDDIR)/stddefs.mk

all: build_pgm

include $(STDDIR)/stdrules.mk
```

This solves the problem at hand—the `build.tag` files get reevaluated every time (but actually even they are either wink or stay untouched). But the real derived objects are stable—the actual rebuild involving the compiler never gets executed unnecessarily:

```
$ clearmake -C gnu -v
No candidate in current view for "/vob/apps/hello/wstr/libwstr.so_build"

======== Rebuilding "/vob/apps/hello/wstr/libwstr.so_build" ========
clearmake[1]: Entering directory `/vob/apps/hello/wstr'
No candidate in current view for "build_lib"

======== Rebuilding "build_lib" ========
clearmake[2]: Entering directory `/vob/apps/hello/wstr'
`build.tag' is up to date.
clearmake[2]: Leaving directory `/vob/apps/hello/wstr'
========================================================

clearmake[1]: Leaving directory `/vob/apps/hello/wstr'
========================================================

Must rebuild "build_pgm" - due to rebuild of subtarget ###################
                                  "/vob/apps/hello/wstr/libwstr.so_build"

======== Rebuilding "build_pgm" ========
clearmake[1]: Entering directory `/vob/apps/hello/helloc'
`build.tag' is up to date.
clearmake[1]: Leaving directory `/vob/apps/hello/helloc'
========================================================
```

Yet the actual rebuild happens in case any of the dependencies change; however, the situation is not satisfactory as of now.

Multiple evaluation of dependencies

We need to make our case one step more complex yet, by adding a second library/subsystem. The function we add now prints a string in reverse order, and as it happens, depends itself on out first library.
Here is the source code, `srev.h` and `srev.c`.

`srev/srev.h`:

```
char* srev(const char* const s);
void wrev(const char* const str);
```

srev/srev.c:

```
#include <stdlib.h>
#include <string.h>
#include <wstr.h>
#include "srev.h"

char* srev(const char* const s) {
  int i = 0;
  int j = strlen(s);
  char* r = malloc(j);
  while (j) {
   r[i++] = s[--j];
  }
  r[i] = 0;
  return r;
}

void wrev(const char* const str) {
  char* p = srev(str);
  wstr(p);
  free((int*)p);
}
```

and the makefile:

```
# srev/Makefile
ROOT := /vob/apps/hello
LIB := libsrev.so
STDDIR := /vob/bld/std
LIBS := $(ROOT)/wstr/libwstr.so
include $(STDDIR)/stdlibdefs.mk

all: build_lib

include $(STDDIR)/stdrules.mk
```

We slightly modify our main function in `hello.c` and its `Makefile` to use the new functionality:

helloc/hello.c:

```
#include "wstr.h"
#include "srev.h"

int main() {
  const char* const s = "Hello World";
  wstr(s);
  wrev(s);
  return 0;
}
```

helloc/Makefile:

```
PGM := hello
ROOT := /vob/apps/hello
STDDIR := /vob/bld/std
LIBS := $(ROOT)/wstr/libwstr.so $(ROOT)/srev/libsrev.so

include $(STDDIR)/stddefs.mk

all: build_pgm

include $(STDDIR)/stdrules.mk
```

We build successfully, using the unchanged shared makefile components, add the new path to the `LD_LIBRARY_PATH` environment variable and run the application:

```
$ ./hello
Hello World
dlroW olleH
```

It looks right. But nevertheless, there is something wrong.
What is the problem? Let's attempt a rebuild:

```
$ clearmake -C gnu
clearmake[1]: Entering directory `/vob/apps/hello/wstr'
clearmake[2]: Entering directory `/vob/apps/hello/wstr'
`build.tag' is up to date.
clearmake[2]: Leaving directory `/vob/apps/hello/wstr'

clearmake[1]: Leaving directory `/vob/apps/hello/wstr'

clearmake[1]: Entering directory `/vob/apps/hello/srev'
clearmake[2]: Entering directory `/vob/apps/hello/wstr'
clearmake[3]: Entering directory `/vob/apps/hello/wstr'
`build.tag' is up to date.
clearmake[3]: Leaving directory `/vob/apps/hello/wstr'

clearmake[2]: Leaving directory `/vob/apps/hello/wstr'

clearmake[2]: Entering directory `/vob/apps/hello/srev'
`build.tag' is up to date.
clearmake[2]: Leaving directory `/vob/apps/hello/srev'

clearmake[1]: Leaving directory `/vob/apps/hello/srev'

clearmake[1]: Entering directory `/vob/apps/hello/helloc'
`build.tag' is up to date.
clearmake[1]: Leaving directory `/vob/apps/hello/helloc'
```

The problem is that we went twice in the `wstr` directory.

Not a big deal? Unfortunately, it is: this is a major nuisance if the system is not as trivial as ours. Complex builds will take significant time just for clearmake to reach the conclusion that they are up to date. The number of evaluations will only grow, and it will grow fast as the number of paths to reach the leaf nodes from the root in the dependency graph. It will actually affect more fine grained designs than coarse grained ones, and thus encourage people to throw management concerns away.

What is important for us is to avoid the trivial solution of hardcoding one particular traversal of the graph, on the ground that it allows some low-level optimizations.

The solution, however, is not particularly difficult on the basis of our system as it is designed so far! We should note that the problem affects only a single target: `$(LIBS:=_build)`. What we want is to avoid invoking its rule a second time, if it has been invoked once. We must thus keep track of the first time it gets invoked. The natural way is by creating a temporary file. We must do it in a location which doesn't depend on the caller, and in a way which doesn't itself get recorded so that nothing depends on it.

The best location is the simplest one—the directory of the remote target.

The `$(LIBS:=_build)` doesn't itself produce any derived object which would be recorded. So the second requirement is easy to fulfill. We may even add an ounce of optimization by telling clearmake not to record anything, using the `.NO_CONFIG_REC` special target.

The next question is the name of the file: it should be unique, to avoid collisions between possible concurrent builds. The usual trick is to use the process id of the top makefile (the one directly invoked by the user). This *pid* is guaranteed to be unique on the host, so we may add to it the host name, which will protect us against builds running on different machines as well as anticipate on the option of using *distributed builds* (see later). We have to propagate this information through the chain of make invocations. Again, this is standard functionality: we may define this macro in such a way that it gets overridden by the value set in the command line invocation.
The last concern is to clean up these files. We shall do this at the end of a successful build, taking care not to overwrite the possible error code returned in case of failure.

Here are the changes to the two shared makefile components, `stddefs.mk`:

```
BUILD := $(shell hostname).$(shell echo $$$$)
LBLD := $(addsuffix $(BUILD), $(dir $(LIBS)))
```

and `stdrules.mk`:

```
.NO_CONFIG_REC: $(LBLD)
...
build_pgm: $(LIBS:=_build)
        @$(MAKE) BUILD=$(BUILD) TGT=$(PGM) build.tag \
        && rm $(LBLD)

build_lib: $(LIBS:=_build)
        @$(MAKE) BUILD=$(BUILD) TGT=$(LIB) build.tag

nolibs \
$(LIBS:=_build):
        @if [ -z "$(BUILD)" ]; \
         then echo stddefs.mk not included; exit 1; fi; \
         if [ -r $(dir $@)$(BUILD) ]; then exit 0; \
         else touch $(dir $@)$(BUILD); \
         cd $(dir $@) && $(MAKE) BUILD=$(BUILD) build_lib; \
        fi
```

Validation

The result we reached now is simple (all is relative), and certainly too simple to answer all the requirements one might have. For example, one might criticize the fact that the include directives are set once per directory, and not once per file. Also that we create a shared library per directory, which may result in a large number of them. These critiques are valid, but can be addressed, subject to tradeoffs. For instance, the latter could be handled by making intermediate archive libraries, and building the shared libraries from them.

Another batch of critiques might focus on robustness, and error handling and reporting. For example, how would this system react in presence of cycles in the dependency graph (the same subsystem being referenced at different levels in the recursive traversal)?

This introduces us to the tool: **ct catcr -union -check**.

Error reports and their analysis

The answer is that the build might work, and even the resulting executable might as well. Yet, a real problem is lurking with potential consequences, which should rather be detected early, while debugging strange behaviors in the produced deliverables. The transcript might be (we insert numbers in column 1, so that we may refer to the precise reports in the following):

```
$ ct catcr -union -check build.tag
----------------------------
MVFS objects:
----------------------------
1)
Object has multiple versions:
First seen in target "build.tag"
 7 /vob/apps/hello@@/main/mg/11 <2010-05-18T18:39:03+01>
First seen in target "libwstr.so"
 2 /vob/apps/hello@@/main/mg/10 <2010-05-16T16:09:18+01>
2)
Object has multiple versions:
Object has no checked-in element version:
First seen in target "build.tag"
 3 /vob/apps/hello/wstr/Makefile <2010-05-20T09:30:46+01>
First seen in target "libwstr.so"
 1 /vob/apps/hello/wstr/Makefile@@/main/mg/1 <2010-05-16T14:22:30+01>
3)
Object has multiple versions:
First seen in target "build.tag"
 2 /vob/apps/hello/wstr/libwstr.so@@--05-23T12:39.178560
First seen in target "hello"
 3 /vob/apps/hello/wstr/libwstr.so@@--05-25T19:44.178752
```

This transcript is the result of introducing the following addition to `wstr/Makefile`:

```
ROOT := /vob/apps/hello
LIBS := $(ROOT)/srev/libsrev.so
```

In this case, this addition is a pure fake: defining the `LIBS` macro results in a dependency which the code doesn't require. It will thus be easy to fix.

Let's note that we draw now the benefit of having produced a "top level tag file", to which the config record of the whole build got attached. We would like to highlight again the fact that it is not an artificial *global* top, from which any build would have to be run. Our top level is a bottom-up result, i.e. the opposite to an original intention, a "project" top. The risk is to tie to the payload of every subsystem the context in which it was—often accidentally—developed. Such a price tag may effectively defeat the benefits of sharing and reuse. In other words, the build of a subsystem made in the context of a project should be re-usable as such in the context of any other project.

Let's review the analysis provided by `catcr -check`:

- *Directory elements.* Error 1) in the transcript above is: `Object has multiple versions`. This is typical of dependency cycles—the current build remembers of the previous one and, if some objects are modified between the two, the recorded versions will not match. In the case of `/vob/apps/hello@@/main/mg/11`, it is however a directory, and as we already mentioned it, directories are ignored for dependency matching purposes. What this means is that this precise case is not the symptom of a cycle, but an unavoidable difference. We'd rather skip it, or in fact, have `catcr -check` skip it for us, so that we may focus on critical errors first (by default, `catcr` will not list directory versions, but `catcr -check` will report their differences). This can be done by using an additional option, `-type f`, for skipping directories and checking file objects only. One must add that real problems may, at times, lurk behind such differences between directory versions. clearmake will pay attention to some (removing from a directory an entry name used in the previous build), but not all (e.g. adding an entry name which would preempt one used previously if the build was run again).
- *Makefile elements*. The next report 2) is a double one: `Object has multiple versions` and `Object has no checked-in element version`. This is a special case of the previous kind, now concerning a file element. We built before checking in (perfectly normal), but this older build event was not overwritten. Referencing a checked out version usually prevents build reproduction (other views have no access to the data, as it is private to our view). This being an error depends upon the state of our build: it tells us that this build is not shareable. `Makefile` is however a makefile. Our case is thus similar to the previous one with directories—makefiles are ignored for dependency matching purposes, and may be found in multiple versions (as builds using a previous version might not have been invalidated). The unfortunate aspect is that `catcr` does not provide us with a convenient flag to ignore these (usually spurious) reports.
- *Derived objects.* At last, error 3) shows us a real problem: `Object has multiple versions`. This is a derived object this time. So, for some reason this derived object was invalidated and rebuilt, but only in a partial context; the rest of our system still depends on the previous version of this derived object and has not been rebuilt accordingly if there was a real need. We must thus investigate where the problem is.

To investigate case 3), we will use a handy tool, `grepcr`, performing a recursive search in the config records, and thus providing invaluable help for locating cycles in the makefiles. The `grepcr` prints derived objects hierarchies: those that reference the problematic DO in question, `libwstr.so`:

```
$ grepcr /vob/apps/hello/helloc/build.tag libwstr.so
/vob/apps/hello/srev/build.tag@@--05-25T20:41.178893:
 /vob/apps/hello/wstr/build.tag@@--05-25T19:44.178753:
   /vob/apps/hello/srev/build.tag@@--05-25T19:03.178745:
     /vob/apps/hello/wstr/build.tag@@--05-23T14:29.178606:
       /vob/apps/hello/wstr/libwstr.so@@--05-23T12:39.178560:
```

What do we see here? There is clearly a cycle between the `srev` and `wstr` subsystems!

It should be easy now to locate the fake addition to the `wstr/Makefile`, resulting in the cycle and to remove it from there.

There are no other kinds of errors in this transcript, but we already saw that fixing the problem (removing the cycle) will not yield us a fully clean transcript, although a sane situation was restored. This minor *false negative* cannot be avoided:

```
$ ct catcr -union -check -type f build.tag
----------------------------
MVFS objects:
----------------------------
Object has multiple versions:
First seen in target "build.tag"
 7 /vob/apps/hello@@/main/mg/11 <2010-05-18T18:39:03+01>
First seen in target "libwstr.so"
 2 /vob/apps/hello@@/main/mg/10 <2010-05-16T16:09:18+01>
Object has multiple versions:
Object has no checked-in element version:
First seen in target "build.tag"
 3 /vob/apps/hello/wstr/Makefile <2010-05-20T09:30:46+01>
First seen in target "libwstr.so"
 1 /vob/apps/hello/wstr/Makefile@@/main/mg/1 <2010-05-16T14:22:30+01>
```

Here is one more possible error report:

```
Element has multiple names: (OID: ########################################
                                  b966d58d.645d11df.97a2.00:0b:db:7d:45:e7)
 /vob/apps/hello/wstr/stddefs.mk (first seen in libwstr.so)
 /vob/bld/std/stddefs.mk (first seen in build.tag)
```

This occurs in case a file element (`stddefs.mk` in this case) was moved to a different directory. Some derived objects used it from its old location. As we mentioned in our comment of 1), such changes, although only in directories, would force a rebuild, if the object itself was not a makefile. As in the previous case, because the version change did not force a rebuild, the previous version stayed recorded, which results in a spurious error report.

Let's mention that the `catcr -union -check` tool may also be used to assert the consistency of several independently built objects, by specifying multiple arguments. This may be very valuable before attempting an integration.

State of derived objects and reference count

As you produce a derived object, in a standard dynamic view, it is first private, non-shared (but shareable), with a reference count of `1`:

```
$ ct setview view1
$ clearmake
 echo ddd> d.out

$ ct lsdo -l d.out
2010-05-27T16:47:52+03:00 Joe Smith (joe@hostname)
 create derived object "d.out@@--05-27T16:47.179000"
 size of derived object is: 4
 last access: 2010-05-27T16:47:52+03:00
 references: 1 => hostname:/viewstorage/view1.vws
```

If you remove it, the data is lost, but the record remains in the vob database, with a reference count of `0`. You can assess its existence with the `lsdo` tool and the `-zero` flag, or with the countdb one (as already noted while speaking of clearaudit):

```
$ rm d.out
$ ct lsdo -zero -l d.out
2010-05-27T16:47:52+03:00 ???.???
 create derived object "d.out@@--05-27T16:47.179000"
 size of derived object is: 4
 last access: 2010-05-27T16:47:52+03:00
 references: 0
```

Note that such a derived object, which is not **shared**, and with zero references, would always get re-created by a subsequent build (as its data is lost):

```
$ clearmake -d
...
No candidate in current view for "d.out"
>>> 16:55:10.913 (clearmake): Shopping for DO named "d.out" in ##########
                                          VOB directory "/vob/test/tmp@@"
```

```
>>> 16:55:10.943 (clearmake): Removed 0-ref, no-data heap derived #######
                                        object "d.out@@--05-27T16:54.179001"

======== Rebuilding "d.out" ========
 echo ddd> d.out
```

If there exists one reference, and a second view attempts to produce the same derived object, in a context in which identical same dependencies get selected, the derived object **winks in** the new view. As a matter of fact, it first gets **promoted** to the **shared** status, its data gets copied to the vob derived object pool, and the new view gets a reference to it.

Examining the object (from either view) with the `lsdo` tool, we can now see that its status is *shared*, and its reference count `2`:

```
$ ct setview view2
$ clearmake
Wink in derived object "b.out"
$ ct lsdo -l b.out
2010-05-27T16:32:59+03:00 Joe Smith (joe@hostname)
 create derived object "b.out@@--05-27T16:32.178992"
 size of derived object is: 0
 last access: 2010-05-27T16:32:59+03:00
 references: 2 (shared)
 => hostname:/viewstorage/view1.vws
 => hostname:/viewstorage/view2.vws
```

Note that we achieve the same result by using the `winkin` command instead of building. If we first promote a derived object using the `winkin` command, and remove it from the view, only then its data is not lost as it is already stored in the vob derived object pool. Such a derived object, despite its null reference count, may still wink in to other views and thus avoid getting rebuilt.

Removing derived objects

Derived objects get automatically scrubbed periodically once they are not referenced anymore. The details may be tuned using the *scheduler*; we'll pay it some attention in *Chapter 10, Administrative Concerns*.

There are cases when you might want to remove some DOs. But as seen previously, until they have been scrubbed, a null reference count doesn't prevent them from winking in back to life. What you might thus want to do, after the reference count has dropped to 0, is to force the scrubbing. There is however a simpler tool than the *scrubber* (used by scheduled jobs) itself: `rmdo`. However, one should only use `rmdo` once the DO has been removed from all the views referencing it, and its reference count has therefore dropped to 0. This requirement is not easy to meet in practice,

when the views of many users are involved. Bypassing it will result in *internal error* reports to the view owners and in the logs.

One reason for removing derived objects could be to reach a situation where `catcr -union -check` reports no errors. This is an optimal baseline to be ready to detect errors easily, if they get introduced.

There are scenarios in which this optimal situation may become spuriously disturbed—by producing equivalent derived objects. This may happen if two concurrent builds are started at nearly the same time. In such a case, neither has yet produced the results, which the other could have winked in. A good design will make such cases of race conditions infrequent, but one cannot avoid them completely. The most common way to produce this adverse situation is by releasing a source code baseline without providing a build to be winked in. We'll see in below in the *Litmus Test* paragraph, how the release procedure may avoid this pitfall. Another way to produce identical DOs is to use *express builds* (i.e. views producing non-shareable derived objects), or some options of clearmake such as `-V` (view only) or `-u` (unconditional), and then to convert the DOs to shareable status (with `winkin` or `view_scrubber -p`, or by checking them in—see later). Such practices are misguided: they give a short term benefit, for a long term (higher level) penalty.

The problem with equivalent DOs is that they introduce spurious differences between config records using different instances out of sets of equivalent ones, and thus pollute the derived object pool. These differences are hard to tell apart, and may thus hide real problems. There is an unfortunate long term bug in clearmake that leverages this issue: when **shopping for derived objects**, clearmake will prefer the *newer* of two equivalent derived objects, therefore propagating a locally created instability in a viral effect. The version of a derived object present in a given view will be validated, but other views will use the newer copy. The result is that the combined system will report differences that will not get cleaned up automatically—they require careful human intervention (to remove the offending duplicate DOs, as explained earlier), or a radical change in basic dependencies.

Here is an illustration of the equivalent derived objects bug:

```
$ ct setview view1
$ ct lsdo wstr.o
--05-23T12:39 "wstr.o@@--05-23T12:39.178559"

$ ct setview view2
$ cd /vob/apps/hello/wstr
$ clearmake -C gnu -u
 /vob/tools/bin/gcc -Wall -fpic -c wstr.c -o wstr.o

 /vob/tools/bin/ld -shared -o libwstr.so wstr.o -lc
```

```
$ ct lsdo wstr.o
--05-27T14:44 "wstr.o@@--05-27T14:44.178969"
--05-23T12:39 "wstr.o@@--05-23T12:39.178559"
$ ct ls wstr.o
wstr.o@@--05-27T14:44.178969

$ ct setview view3
$ cd /vob/apps/hello/wstr
$ clearmake -C gnu -d
...
No candidate in current view for "wstr.o"
>>> 15:27:46.791 (clearmake): Shopping for DO named "wstr.o" in VOB #####
                                        directory "/vob/cifdoc/test/wstr@@"
>>> 15:27:46.798 (clearmake): Evaluating heap derived object ##############
                                            "wstr.o@@--05-27T14:44.178969"
Wink in derived object "wstr.o@@--05-27T14:44.178969"
...
$ ct ls wstr.o
wstr.o@@--05-27T14:44.178969
```

As we can see, in this case when the equivalent derived object has been explicitly created with a `clearmake -u` command, clearmake chooses the "heap derived object", that is, the newest from the two available ones.

Dependencies on the environment and on tools

At this point, we found that the excellent `catcr -union -check` tool has some possible drawbacks—it may report *false negatives* that it is not trivial to get rid of, or even to interpret safely. At the very least, it requires some work and attention.

The unfortunate truth is that it may also err on the other side, and fail to report some problems that could prevent the reproduction of recorded events. A typical cause of such hidden dependencies is the user environment. This is a difficult question as clearmake cannot determine what variables affect where. What is possible is to make sure that environment variables are explicitly set in the makefiles, so that builds get protected from user settings (and that the user is free to set her environment as she pleases). The problem is to determine what variables to define, and for this, we know nothing better than heuristics, and trial and error. So, as soon as an impacting variable has been spotted, define it away—the only shame is to hit the same pitfall a second time.

One particular environment variable is of course `PATH`, which determines the algorithm used to find tools that would not be defined with their full path. It is a good idea to set `PATH` in the makefiles, and a better one is to define the full path of one's tools (define a macro for every tool). One might think that defining `PATH` is sufficient, but it leaves the door open to finding variants of the same tools in different places on different hosts.

Of course, a full path is not enough to determine tool, or its version. One might be tempted to hardcode the version of the tools in their name or their path. This is usually a poor practice, and tends to prevent changes, as the names often spread to several places— scripts, makefiles, documentation, symbolic links. There is no syntax for aliasing such inconvenient names (hence performing the reverse task for one's convenience) that would suit all the contexts of use.

Another easy solution may seem to be accessing the different versions of tools using symbolic links. Symbolic links deserve a special note. They tend to be overused by people without SCM background. One major problem with symbolic links is that they are not a object of their own (apart in one special corner case related to the `lost+found` directory of replicated vobs)—one cannot version them. They are data of directory objects, and as we saw already, directories are ignored for DO matching purposes, except in some limited cases. In addition, symbolic links introduce potentialities for spurious differences in recorded build scripts. Only for these reasons, one should avoid them.

If one wants to discriminate tools, one generic way is to store some specific data (e.g. cksum or version output) about them in makefile macros, and to use these macros in the build rules (even artificially, e.g. by echoing their value to `/dev/null`).

By far the best solution to record tools is however to maintain them in vobs, under common names that would not be tied to a particular version. We'll devote *Chapter 8*, *Tool Maintenance*, to this issue.

Reproducing the build

Fortunately, there is an easy way to make sure one's build is reproducible, and it is to try.

To do it, the first task is to complete the packaging, by applying a convenient label, which will serve as handle (be this our introduction to labels, although we'll be back on their subject in *Chapter 6*, *Primary Metadata*).

First, we need to create label types in every vob referenced. We'll use the config record to tell us which vobs:

```
$ ct catcr -type d -s -flat build.tag | \
perl -nle 'print"mklbtype -nc TYPE\@$_" if s:/\.@@.*$::' | cleartool
```

This command *flattens* the config record, retaining only directory versions. The list is piped to the perl command line invocation that filters vobs root directories only (only they have a dot in their version extended representation: `/vob/tag/.@@/version`) and applies a simple pattern replacement for striping the version information. Using the result, it prints commands to create the label type in every vob, and the output is piped again to cleartool for execution.

This only assumes we attach no comment to the new `TYPE`. This way, we shall create one label type per vob, and these will all have the same (and each of them is the vob local) name. There would be an option to link these types together, using so-called **hyperlinks**, and thus to make them possibly global. This would, however, open a discussion, which we'll defer until *Chapter 9*, *Secondary Metadata*.

What remains now is to apply the label:

```
$ ct mklabel -con build.tag TYPE
```

This command will suffice to traverse all the vobs. It will give us errors if several different versions of the same elements are met during the labeling process, which, as we already decided, might be acceptable in the case of directories and of makefiles. In such occurrences, the label will go to the *latest* version in case the both versions are on the same branch (which may be trivial and thus safe to decide, or may not be so, depending on the topology of the version tree). This is the best possible choice—the differences have been considered by clearmake.

There is of course one case in which this will not work—if we have intentionally used different versions of some elements in different contexts (either by building separately, or by using hard links and exceptional scoped rules based on the alternate names, as alluded in the previous chapter). In such cases, a label will not do: this has to be on at most one version. But exceptions are exceptions, and it is fine to be reminded of them.

The main issue against labeling is the time it takes, but we believe this should rather be addressed than worked around.

Let's however mention one workaround which relates to our current topic: there is a config spec syntax to allow using a config record directly (that is `-config do-pname` option in the view config spec), instead of, as we propose here, via a label applied using it. This does of course shortcut the application! We lack experience of using this rule in practice, and have always preferred to retain a label type as a concrete handle to reproduce a software configuration.

We do, however, record as an **attribute** (let's leave this aspect to Chapter 9) of the label type, the identity of the config record. This identity should allow a different user to retrieve the config record if it is still available, and otherwise, to produce it again. We deal with the latter concern in the following, using the label type itself. The former supposes that one records both the *full path* (guaranteed to be accessible using the label in one's config spec), and the *id of the derived object* in the label type attribute value. ClearCase provides us indeed with a unique identification for every derived object produced, or actually even with two of them:

```
$ ct des -fmt "%n\n%On\n" build.tag
build.tag@@--05-20T09:37.20925
e463cd5e.63ea11df.946a.00:15:60:04:45:5c
```

The `-fmt` option is a standard way with many cleartool commands (for example, `describe` command, which we'll use heavily) to define the format in which one expects the output. It is especially useful to produce synthetic results on a single line, and therefore suitable for grepping. On the contrary, we used it here to produce two lines: the first with the *DO-ID* of the derived object and the other with its **oid**. The former is obviously specific to derived objects. It is human oriented, a compound of a file name, with a time stamp. This doesn't make it memorizable (a wished property of names) but it makes it understandable and easy to relate to other such names. Its weakness lies as often in its *user friendliness*: the timestamp in it is not guaranteed to be stable. On the contrary, it will change after one year to explicitly mention the year. This is what leads us to store at least in addition, the latter line, the oid.

This is, on the contrary, a low-level **object id**, a kind shared with all ClearCase objects, elements, or types. Note that this is not the lowest possible id: there also exists a seldom met, e.g. in some error reports, *dbid* at the underlying database level.

This oid is stable; however, it is not sufficient alone, and not only from a cognitive point of view: reconstructing the original DO-ID from it is not trivial. One can use the following to retrieve the original DO-ID timestamp part:

```
$ ct des -fmt "%n\n" oid:e463cd5e.63ea11df.946a.00:15:60:04:45:5c
e463cd5e.63ea11df.946a.00:15:60:04:45:5c@@--05-20T09:37.20925
```

And as for the file name part of the DO-ID, we need to keep track of it ourselves, and that is why we record it separately as well:

```
$ ct des -fmt "%Na\n" lbtype:TYPE@/vob/apps | tr ' ' '\n'
ConfigRecordDO="build.tag@@--05-20T09:37.20925"
ConfigRecordOID="e463cd5e.63ea11df.946a.00:15:60:04:45:5c"
```

Now that we have the labels in place, we may want to lock the label types as a proof that nothing could affect them after the time stamp of the locking event.

The label type `TYPE`, which we have just created, attributed, and applied, can now serve two different but related purposes: first, for the build reproducibility (see the quick-check *Litmus test* below), and the second, for fetching the derived object, and the whole software configuration along with it by explicitly winkin from the command line:

```
$ ct winkin -r build.tag@@--05-20T09:37.20925
```

This will be useful for our "customers", to access our proposal with the least effort; but we won't get deeper into that right now.

Litmus test

We can now create a new view, and set a one-line config spec using the label type as its single rule:

```
element * TYPE
```

What we want to do is to build using the same makefile and our target is to produce the DO in the first place. And what we expect to witness is a full winkin and build avoidance.

The point is: if the build *does* build anything, then something went wrong in the recording —we do not have the exact same dependencies.

At this point we do have data to examine; we may run `catcr -union -check` on the two topmost build tags and narrow the differences down using out `grepcr`.

Note that this test, however good, cannot prove we have no hidden dependency on the environment. The winkin is only as good as the record is—a full winkin is not a proof that clearmake could have built what it winked in.

This procedure may seem heavy, but it all depends on the frequency of mismatches, and after a first investment in tuning the makefiles, and there, especially the dependency lists, the whole system will converge to a stable one, with minor incremental issues bound to the latest changes.

This may also serve as the basis for the further enhancements, just as we did illustrate in our *Teaser*.

In that case the label type also helps the new developer to find out, *how* the build was actually made, that is, to find the Makefile along with its location and the target!:

```
$ ct des -fmt "%Na\n" lbtype:TYPE@/vob/apps | tr ' ' '\n'
ConfigRecordDO="build.tag@@--05-20T09:37.20925"
ConfigRecordOID="e463cd5e.63ea11df.946a.00:15:60:04:45:5c"

$ cleartool catcr build.tag@@--05-20T09:37.20925
Derived object: build.tag@@--05-20T09:37.20925
Target build.tag built by joe
Host "tarzan" running Linux 2.6.18-128.el5
Reference Time 2010-04-1T07:51:07Z, this audit started ###################
                                                    2010-04-01T07:51:07Z
View was vue.fiction.com:/views/jane/joe.vws
Initial working directory was /vob/apps/hello/helloc
----------------------------
MVFS objects:
----------------------------
/vob/apps/hello/helloc/build.tag@@--05-29T15:29.179120
/vob/apps/hello/helloc/Makefile@@/main/cif/5 <2010-05-24T19:16:03+03:00>
/vob/apps/hello/srev/build.tag@@--05-29T15:29.179117
/vob/bld/std/stddefs.mk@@/main/cif/3 <2010-05-25T16:58:25+03:00>
/vob/bld/std/stdrules.mk@@/main/cif/8 <2010-05-29T15:25:48+03:00>
/vob/apps/hello/wstr/build.tag@@--05-29T15:28.179108
----------------------------
non-MVFS objects:
----------------------------
/opt/rational/clearcase/etc/gnubuiltin.mk <2008-03-07T02:26:59+02:00>
/opt/rational/clearcase/etc/gnubuiltinvars.mk <1998-12-22T02:47:15+02:00>
----------------------------
Variables and Options:
----------------------------
LTAGS=/vob/apps/hello/wstr/build.tag /vob/apps/hello/srev/build.tag
MAKE=clearmake
MAKEFILE=Makefile
RECTAG=rm -f build.tag; touch build.tag
STAT=head -1 > /dev/null
TGT=
----------------------------
Build Script:
----------------------------
 @if [ -z "" ]; then clearmake -f Makefile all; fi; ####################
                      head -1 > /dev/null /vob/apps/hello/wstr/build.tag
         /vob/apps/hello/srev/build.tag; rm -f build.tag; touch build.tag
----------------------------
```

Then we know we need to use the `Makefile` makefile from the `/vob/apps/hello/helloc` directory (note the line, `Initial working directory was /vob/apps/hello/helloc`, in the transcript above), and that the makefile target is `build.tag` (note the line, `Target build.tag built by joe`, in the transcript).

What is more important is that this ought to describe the situation following a release: any *official* or just *higher level* build using our latest contribution should only *validate* it, i.e. promote it as the baseline status for others. The most convincing way in which it could achieve this is by **building nothing anew**. One obvious condition for this to be possible is that the release procedure itself should provide a build suitable for winkin. The build is a necessary part of the release, and it must take place before updating the baseline.

We'll have to show in Chapter 6 how to avoid breaking this with adverse release processes.

Let's stress here how this opposes most traditional strategies, in which the build manager confidence is grounded in the fact that *she* built everything in *her* controlled build environment. In the context of clearmake, we may ground confidence on **transparency** and **management**, instead of on *opacity* and *control*! We may unleash the benefits of collaboration.

One result we obtained is the identification by the system of the derived objects, as members of a family, which allows them to be elected as a possible, if not the preferred, ClearCase implementation of the concept of configuration item. We cannot avoid to reckon that the tight mapping of the DO family identity to a full path name sounds like an unfortunate overspecification.

We reached here to the logical conclusion of this chapter, which we'll wrap up nevertheless once again before the beginning of the next one.

However, before concluding, we must handle a few issues that did not find a suitable place in our presentation.

Tying some knots

Some issues could not find their natural place in the flow of our presentation, but must be dealt with nevertheless.

Ties between vobs and views

After we have produced shared derived objects, and these have thus been moved to some vobs' derived object pools, we broke the clean separation between vobs and views. We can easily assess this by running the `describe` command with its `-long` flag on some vob object:

```
$ ct des -l vob:.
versioned object base "/vob/apps"
...
 VOB holds objects from the following views:
...
vue.fiction.com:/views/mg/mg.vws [uuid ##################################
                                  d2d16962.837211de.9736.00:01:84:2b:ec:ee]
vue.fiction.com:/views/mg/mg2.vws [uuid ##################################
                                  abcb9897.604e11df.8511.00:01:84:2b:ec:ee]
...
```

This is certainly a minor detail which may be ignored most of the time. It will, however, affect removing of views and resynchronizing them with the vobs after a recovery from backup or another replica. We'll be back to the former in Chapter 10.

Distributed or parallel builds

The issue of builds distribution is *in*: it brings along with load balancing, significant hopes of performance gains and optimal resource utilization. It is supported by clearmake using `-J` options standard to other make tools and to Gnu make in particular (the option is `-j` in Gnu make). There are, however, pros and cons.

Let's set up an environment for distributed builds and collect some data. We need to create a file with a list of hosts in our home directory, with a name starting with `.bldhost` and a suffix matching an environment variable `CCASE_HOST_TYPE` (which would rather be a makefile macro), and to ensure that the remote shell used by clearmake is enabled for our use, between these hosts:

```
$ export CCASE_HOST_TYPE=test
$ cat ~/.bldhost.test
sartre
beauvoir
$ ll /opt/rational/clearcase/etc/rsh
lrwxrwxrwx 1 root other 12 Aug 3 2007 /opt/rational/clearcase/etc/rsh ###
                                                              -> /usr/ucb/rsh
$ cat <<eot> ~/.rhosts
> sartre
> beauvoir
> eot
$ /opt/rational/clearcase/etc/rsh beauvoir echo hello
hello
```

Instead of configuring `rsh`, we could as well have set the `CCASE_ABE_STARTER_PN` environment variable to point to, for example, `ssh`, and used it.

With this setup completed, we are ready to fire the first try of the distributed build, which produces some error though:

```
$ clearmake -C gnu -J 2
Build host status:
 Host sartre unacceptable: only 36% idle, build hosts file requests #####
                                                                  50% idle.
...
```

What is the problem? Well, we need to fine-tune the default 50 percent idleness limit (a lower value doesn't seem unreasonable):

```
$ cat ~/.bldhost.test
-idle 30
sartre
beauvoir
```

Then for the purpose of the test, as explained earlier, we removed all the derived objects, both from all views and, as vob owner with `rmdo`, from the derived object pool.

We ran the following command, thus under a utility measuring the time spent (in user and kernel spaces, then from a *wallclock* perspective: *real*). We slightly skim the output:

```
$ time clearmake -C gnu -J 2
Rebuilding "/vob/apps/hello/wstr/libwstr.so_build" on host "sartre"
Rebuilding "/vob/apps/hello/srev/libsrev.so_build" on host "beauvoir"

======== Finished "/vob/apps/hello/wstr/libwstr.so_build" on host #######
                                                        "sartre" ========
...
======== Finished "/vob/apps/hello/srev/libsrev.so_build" on host #######
                                                      "beauvoir" ========
...
======== Finished "/vob/apps/hello/wstr/libwstr.so_build" on host #######
                                                        "sartre" ========
...
======== Finished "build.tag" on host "sartre" ========
============================================================

clearmake[1]: Leaving directory `/vob/apps/hello/helloc'
============================================================

real 0m5.735s
user 0m0.515s
sys 0m0.995s
```

The similar time test for a non-distributed build fires the following figures:

```
$ time clearmake -C gnu
...
real 0m2.454s
user 0m0.476s
sys 0m1.087s
```

The results (on such a small build system) are quite clear:

- The build passes and produces identical results to the non-distributed one
- There is an overhead to distribute the builds: the shortest times achieved even with the `-J` flag are met when the actual building happens on one single host (because other hosts are temporarily overloaded)
- The overhead is comparatively much larger in subsequent builds (i.e. when the build process is mostly just asserting that nothing needs to be built)

The first point cannot be neglected: a correct build system, with dependencies completely described, should be distributable. As a matter of fact, distributing the build is a sound test of the build system. An incorrect build with insufficient dependencies will fail only randomly, depending on which targets are distributed to different hosts, so that a one-time success is no guarantee.

The second and third points are obvious in afterthought, but must be kept in mind: the overhead will penalize a user building a small system. But it will also penalize all users on average—there is a bounty paid to distributing builds. This bounty may be tolerable if there are computing and networking resources in excess, but it becomes unjustified under heavy load. One might also guess that the overhead penalizes fine-grained systems more than coarse-grained ones, but we do not have facts to back this guess.

A break-even is to be expected for a certain size of the build, which is neither easy to compute nor stable. One could expect that distributing builds may win on building from scratch, but one should not be surprised to see distributed builds compare poorly on incremental builds, which should be the main target of our SCM focus: **manage by differences**.

Let's however admit that parallel building doesn't work well with recursion in clearmake: until version 7.1, the value given by the user via the `-J` flag or the `CCASE_CONC` variable (the maximum number of concurrent builds) is propagated and implicitly reused independently by every recursive clearmake invocation, potentially resulting in an exponential explosion of build jobs; in 7.1, the value is is not propagated, so that only the initial build targets may be distributed. In order to fix this problem, one may consider computing a `JVALUE` parameter, and use it to spawn sub-makes with: `$(MAKE) -J $(JVALUE)`.

The computation will need the original value of CCASE_CONC (propagated e.g. as ORIG_CONC, since CCASE_CONC is locally overridden by the parent -J option), as well as a count of the processes currently running, obtained with something similar to $(shell pgrep clearcase/etc/abe). Note that *abe* (audited build executor) processes are launched for build jobs on remote hosts, not for multiple jobs on the same host (if one uses the same host multiple times in one's bldhost file, for example, to take advantage of a cluster architecture). In this latter case, one would have to monitor something else than the number of abe processes, maybe just clearmake ones.

Here is a code example, using wc (UNIX word count) to count the number of clearmake processes, and dc (UNIX desk calculator) to calculate the value of the JVALUE parameter:

```
ifeq ($(MAKE), clearmake)
   ORIG_CONC := $(CCASE_CONC)
   LCONC = $(shell pgrep clearmake | wc -l)
   JVALUE = $(shell p=`echo $(ORIG_CONC) $(LCONC) - p | dc`; \
     if [ $$p -lt 0 ]; then echo 0; else echo $$p; fi)
   OPTS = ORIG_CONC=$(ORIG_CONC) -J $(JVALUE)
endif
```

One would use this value explicitly in the remote invocations (only on pseudo targets: beware recording it into the config records!):

```
    $(MAKE) $(OPTS)
```

Note also that such measurements may easily be fooled if started concurrently (thus too early to take each other into account). This is why GNU make actually uses a more robust technology, based on serialized IPC.

In some cases (such as precisely, taking advantage of multi-processor or cluster architectures) such complexity (or better) might be worth the while. Otherwise, understanding these aspects, one may tune the values of CCASE_CONC to get some benefit of distribution for one's system.

Let's conclude our critique with the following note: using derived objects already built by others (winking them in), *is* actually a form of distributed build! This is true at least from the point of view of the resource utilization.

Staging

The practice of making elements of one's critical deliverables and checking in new versions, also referred to as *staging*, is often recommended, even in IBM/Rational own documentation.

Creating an element, checking out, and checking in a version, will be the topic of the next chapter.

Doing it inside the build brings in a few additional issues, such as:

- Accessing the checked-in versions in a view in which `LATEST` has been locked at the time of starting the build (as with a `-time` clause in a config spec): this is best done by using a label, which must first be applied, which leads to the next problem.
- Accessing checked-out versions with commands such as `mklabel`: one needs to use an explicit `@@/main/CHECKEDOUT` extension, meaning that one has to record or to compute the branch from where the version was checked out.
- Using the additional `-c` option of clearmake to check out the derived object versions prior to modifying them.

Several reasons may be invoked in favor of staging:

- Ensuring that critical deliverables won't be lost (especially after they have been delivered to customers).
- Providing a performance benefit, by avoiding configuration lookup.
- Sharing derived objects with MultiSite.

The main price is, however, to create alternative dependency trees: ones with opaque nodes. The makefile system stops being transparent.

Of course, the `catcr -union -check` consistency test is still available—checked in derived objects remain derived objects. But asserting the consistency of the build becomes a political issue: it is not collaborative anymore.

Staging is thus not downright incompatible with derived object management under clearmake, but it surely competes against it.

We believe the concerns listed above may be addressed in other ways:

- Shared derived objects won't get scrubbed if they remained referenced. One may easily keep a view with a reference to the top of a config record hierarchy, which will ensure the persistence of the critical assets.
- We believe that we sufficiently addressed the performance issues and showed that the gain of staging is at best only minimal against a well-tuned build system.
- Sharing derived objects across sites is not supported by MultiSite. Sharing the data may be uselessly expensive: network bandwidth has progressed greatly, but latency has not, and is not expected to, under the current laws of physics.

- In any case, even bandwidth has not progressed at the same speed as CPU and clocks. It does make sense to share config records, for comparison purposes, which is possible with a `-cr` option of the `checkin` tool. This option has the same issues as staging proper, with placing references to the derived objects into journaling records (known as *oplogs*) used for the replication. We shall come back to this in *Chapter 5*, *MultiSite Concerns*.

Application to other tasks than mere builds

In the spirit of our *Teaser*, we want to stress that the techniques developed in the chapter may be applied to other tasks than the ones understood with a narrow acceptation of *building*. In general, they may be used to avoid running again and unnecessarily arbitrary tasks, by managing the artifacts these ones produce. An excellent example of a domain of application is test avoidance, and especially regression testing.

Summary

We are aware that we dealt with a mass of technical details. We believe these details were essential to satisfy the extremely high expectations we place on a *full* use of the avoidance and winkin machinery that ClearCase offers us. In the precision with which the dependencies are described lies the scalability and the economy of the recording, and hence eventually, the justification for the strong points we'll make in our conclusion.

We would like however to bring our reader's attention, away from those technical details, to the *paradigm shift* that has taken place: from *producing deliverables*, to **reproducing** *a task* performed elsewhere by somebody else. *Formal sameness* is now detected by the *system* (clearmake), and guaranteed by avoiding to merely produce *again* the same, or similar in some loose sense, deliverables. The benefit is not so much a matter of build speed or of space saving. It lies in the **stability** that could be obtained, and communicated between developers, for the incremental steps of the development process.

This is the insight we'll have to carry with us while exploring the rest of ClearCase.

4
Version Control

Dear Reader,

You have already entered the second part of our book. We stated our main thesis (didn't you notice?). What remains for us is to make the dream live! It will be a feast and a fight, of minute details, on a narrow path between slippery slopes.

The CM tradition was built on top of versions of source artifacts, crafted by hand: its world was a construction ex nihilo. By contrast, our world is always already there. Furthermore, it is not a cookbook trying to blow a sparkle of life into an assembly of carbon atoms: on the contrary, it is a frenetic fight to make sense of a universe of life all around us, and which we are part of. The only stability there is the one we are responsible for: meaning. ClearCase made it possible to put SCM back on its feet, after so many years of its standing on its head.

Derived objects are found in sets of instances: they meet there the fundamental requirement for **configuration items**, as set by the theoretical SCM framework. In fact *they* represent the prototype. Elements are what ClearCase names the hand crafted source artifacts. They can be considered as degenerate derived objects: the leaves of the dependency tree. Contrary to the derived objects, which build up a deep structure (the dependency tree), they are found at the surface of the software configuration space: they are flat, and therefore opaque from a *generic* point of view (but we do not forbid *specific* analysis, using file formats aware tools, and interpreting the data in ways that make structure arise).

We'll focus in this chapter on this small subset of the configuration items: the most simple, the most humble ones.

Historical concerns are neither jeopardized nor forgotten. We'll show here how they are supported by ClearCase, in ways that only minimally depart from standard expectations:

- Making new elements
- Check out and check in, at least in simple cases, and the subtleties of managing memory
- Looking for differences between versions
- Answering some misguided critiques

Making elements

Elements, i.e. potential families of versions, may be created either empty or from view private files (but not from private directories). The command is `mkelem`, and there exists an `mkdir` shortcut for directories (equivalent to `mkelem -eltype directory`). The same operation, if carried out from the GUI, is termed *add to source control*.

Whether or not obvious, let's note that one needs to **checkout** the parent directory before adding new elements, and to **check** it **in** afterwards before others may access the result of the operation: one cannot create an element without assigning it a path. Note that a newly created vob comes with a checked in root directory.

We noticed a restriction: *not from private directories*. It is worth mentioning that many such orthogonality glitches have been ironed away by David Boyce, in his wrapper (or actually wrappers) available from CPAN: ***ClearCase::Wrapper*** (and ***ClearCase::Wrapper::DSB***). Let's take this as an example: he offers `mkelem` a `-recurse` option (among others) which, one soon forgets, is missing from the original cleartool.

If adding several elements to the same directory, it is usually better to check out the directory once, add all the elements, and check in the directory once. This creates fewer events in the history, results in a smaller *version tree* of the directory, and is simply faster.

The `mkelem` command has flags that allow to deal with details, which otherwise would require some user interaction:

- Giving a comment (`-c`) or not (`-nc`). The comment is duplicated between the element and the first version if checked in simultaneously.

- Avoiding to check out the new element from the initial version `/main/0` (`-nco`). The default behavior is to check out the initial version and remove the view-private copy of the data. Note that the `/main/0` version is always empty, and in case the `-nco` option was used along with a non-empty view-private file, the element with the specified name gets created, and the view-private file is renamed to yet another view private file with the additional extension `.keep` (in order not to **eclipse** the element created in its `/main/0` version).
- Checking in the initial version *(*`-ci`), which got checked out by default, and create the version 1 on some branch (depending on the **config spec**).

In relation with the default checkout behavior, it is important to set the config spec correctly. It must contain the following line in order to make element creation possible:

```
element * /main/0
```

This line can also be:

```
element * /main/LATEST
```

We mentioned this in the presentation of config specs in Chapter 2.

As any element gets initiated with (an empty) version 0 on the branch `main`, and in order to check it out (may be to another branch), it must be selected by the config spec.

Elements have a type, which affects the way they are stored and the way their versions may be compared. This type is determined from their name, and possibly their contents, unless it is given explicitly with an `-eltype` option. We'll develop on these in *Chapter 9*, *Secondary Metadata*, but may already mention the existence of a **magic file**, with rules to determine the type of the new element. The default ClearCase magic file (`/opt/rational/clearcase/config/magic/default.magic`) falls back to `compressed_file` if no other suitable match was found.

Mass creation of elements requires other tools:

- *clearfsimport*, if importing from a flat directory structure
- Tools of the *clearexport* family, followed with *clearimport*, if importing version trees from other tools to ClearCase
- *cleartool relocate* to export data from one ClearCase vob to another

We'll cover those in *Chapter 8*, *Tools Maintenance*, and *Chapter 10*, *Administrative Concerns*. The development model encouraged by ClearCase is continuous, with early publication (under user/consumer control) of small increments. One easily understands that the issue of mass creation should be considered rather exceptional. Again, it is worth mentioning that a tool far more flexible and configurable than *clearfsimport* is David Boyce's *synctree* (see Chapter 10).

The last source of elements is other replicas, but this aspect belongs to *Chapter 5*, *MultiSite*.

If you remember our *Presentation of ClearCase* (*Chapter 2*), you'll recall that a critical aspect of the creation of a new directory is the *umask* of the user: this drives the ability for others to produce derived objects or to add new elements to this directory, and even to checkout existing elements without complex syntax (see later). In most cases, directories should be *writable to group*.

Checkout and checkin

Elements under ClearCase are fully accessible (but read-only) as **checked in**, e.g. to tools such as compilers. **Checking out** implies thus in general terms an intention to *modify* an element (we'll discuss later in this chapter the special case of tools requiring that some files be kept writable). A version checked out becomes a view-private file and is not shared with others (unless via a view-extended path) until it's checked in. This is why checking out is at best deferred as late as possible, and also done on a file basis, as needed. In the same way, checking in should happen as soon as possible, as it provides a fine-grain backup: it is easily decoupled from *delivery* to others. The set of checked out files should therefore be, at any moment, minimal, and give an indication of the precise focus of one's current work. So, we can say that the rule of thumb in ClearCase is: *Check in often, check in soon!*

We mentioned earlier that in order to check out a file, one needs to be allowed to save its data in a directory. If the parent directory is owned by another user, and is not group writable (assuming you belong to the same group), checking out will fail. You need then to specify another directory where to save the data, with an `-out` option. When the time comes to check in your changes, you can use the `-from` option.

Checking in an identical copy of a text file is suspected to be an error (it doesn't make much sense, as it just clobbers the version tree, and gratuitously invalidates derived objects that would have been built using the first version). To force ClearCase to accept this, you must use an `-identical` flag. Note though that this behavior depends on the element type manager: if the type manager does not compute version-to-version **deltas** (contrast `text_file` versions which are stored as deltas—see Chapter 9—to `compressed_file` ones which are not) a new (even identical) version is always created for the element. Most of the time, what makes sense is however to undo the checkout (you changed your mind and actually do not need to modify the file). Then the correct option is obviously to **uncheckout** the element. The fact that this is a distinct command is a minor nuisance, especially when you process several files at a time. David Boyce's wrapper serves this with a `-revert` flag to `checkin`.

There is obviously more to say about checking out and back in, in the context of a branching strategy, but we'll defer it until *Chapter 6*, *Primary Metadata*, after we have taken into account the requirements that may arise from MultiSite.

Versioned directories

Early version control systems only supported text files. It soon became obvious that names are equally important as any other data, and that the pressure to change them, as well as the layout of file systems, made **directory versioning** necessary. The only aspect of directory data which is versioned in ClearCase is the name of entries: the size and the time stamp of files vary freely (they are those of the versions, and thus depend on the selection), the ownership and the access rights are common to the whole elements, and thus non versioned—shared among all the versions.

In order to rename a directory entry (itself a file, a symbolic link, or a directory), to create it, or to remove it, one needs to first check out the parent directory. To move a file, one needs to check out both the source and the target directories. And conversely, the modification is private until the directories are checked in back.

lost+found

At the root of every vob, in the `/main/0` version of it, there is a `lost+found` directory (there is one also at times in the `.s` directory of the view storage: see below).

This is a special directory, which has only a `/main/0` version and cannot be checked out. One may *remove* its name from a new version of the root directory, but this doesn't affect its real existence (as it is always accessible via the version-extended path `<vobtag>/.@@/main/0/lost+found`), so this is therefore a questionable move.

Elements are moved there when they are not referenced from anywhere anymore (they are then called *orphans*).

The simplest scenario is the following: the user checks out a directory, creates a few elements there, and then changes her mind (maybe the intended elements have already been created elsewhere). She unchecks out her directory: the newly created elements are moved to `lost+found`.

In this process, the files get renamed, in order to avoid possible name clashes: a compact form of the element's oid is appended to the original base name:

```
$ ct unco -rm .
cleartool: Warning: Object "a" no longer referenced.
cleartool: Warning: Moving object to vob lost+found directory as ########
                                        "a.90d36f02668949f7b000811fe3624a51".
$ cd lost+found
$ ls -ld a.90d36f02668949f7b000811fe3624a51
drwxrwxrwx 2 joe jgroup 0 Feb 20 18:09 a.90d36f02668949f7b000811fe3624a51
$ ct des -fmt "%On\n" a.90d36f02668949f7b000811fe3624a51@@
90d36f02.668949f7.b000.81:1f:e3:62:4a:51
```

Note that every version has a distinct oid: in order to get the element oid, we had to append `@@` to the name.

Moving unreferenced files to `lost+found` is a safety feature on the behalf of ClearCase: it protects the user from the unexpected consequences of innocuous commands.

Other scenarios are probably less innocuous, as they may include removing versions of directories or even whole directory elements.

There are typically two things one might want to do with files in `lost+found`:

- Restore them, if they were actually valuable. This is easily done with the `mv` (move) command, using the basename and dropping the oid, but provided one knows where to restore them, as the original path information is lost:

  ```
  $ ct co -nc .
  $ ct mv /vob/apps/lost+found/a.90d36f02668949f7b000811fe3624a51 a
  $ ct ci -nc .
  ```

- Remove them completely. This is a good practice, not only to save disk space, but also to make it easier for others to manage their own files in the future.

The tool to perform the cleanup is **rmelem** (remove element). Removing elements destroys information in an irreversible way, and should thus not be done lightly. But precisely for this reason, this tool is safe to use: apart for administrators, only element owners may use it, and only if strict conditions apply (no metadata attached to any version, and all branches created by the owner):

```
$ ct mklabel LABEL foo
Created label "LABEL" on "foo" version "/main/br/1".
$ ct rmelem -f foo
cleartool: Error: Element "foo" has labeled versions
cleartool: Error: You must be the VOB owner or privileged user to #######
                                                    perform this operation.

$ id -un
joe
$ sudo -u adm cleartool mkbranch -nc br2 bar
Created branch "br2" from "bar" version "/main/5".
$ ct rmelem -f bar
cleartool: Error: Element "bar" has branches not created by user
cleartool: Error: You must be the VOB owner or privileged user to #######
                                                    perform this operation.
```

This means that in practice, one can only use it to correct a mistake just after it occurred, and before any other user actually used the element. Therefore, administrators who disable the use of `rmelem` are misguided and do create more problems than they solve.

Note that in order to use rmelem, especially in `lost+found`, you do not need to check out the parent directory; but you do need uncheck out the element (if it happens to be checked out) before it can be removed.

```
$ ct rmelem -f a.90d36f02668949f7b000811fe3624a51
cleartool: Error: Can't delete element with checked out versions.
cleartool: Error: Unable to remove element ##############################
                                    "a.90d36f02668949f7b000811fe3624a51".

$ ct unco -rm a.90d36f02668949f7b000811fe3624a51
$ ct rmelem -f a.90d36f02668949f7b000811fe3624a51
Removed element "a.90d36f02668949f7b000811fe3624a51".
```

In the above scenario, the user should have removed her elements before unchecking out the directory.

Cleaning up `lost+found` of others' files is an administrative procedure, and we'll handle it in Chapter 10. Let's however warn the user of two things:

- One should *not* use rmelem recursively from `lost+found` (on the output of a `find` command): the fact that a directory is not referenced anymore does not guarantee that its contents is not.
- It is normal for the number of cataloged elements to *grow* as a result of removing some directories: their content was not *yet* in `lost+found`, as it *was* referenced; not so anymore after one removed their parent directory.

The view `lost+found` directory, which we already mentioned, is located in the view storage directory, inside the `.s` subdirectory: `view-storage.vws/.s/lost+found`. Unlike the vob `lost+found`, it does not exist by default. It only gets created when user executes `cleartool recoverview` command to make the *stranded* files available explicitly. Stranded view-private files are files, whose VOB pathname is not accessible at the moment (e.g. in case the vob in question was unmounted or removed), or whose directory name inside the vob is not available if e.g. not selected by the view's current config spec. Such stranded files can sometimes become available again, for example, if the vob gets mounted (the former case) or the view config spec changes to select the file's parent directory (the latter case). The user can also decide to move the stranded files explicitly to her view's storage `lost+found` directory by using the recoverview command with `-sync`, `-vob`, or `-dir` options. The moved files are accessible with normal operating system commands.

Removing files

As an executive summary of the previous paragraph, it is important for the user to understand the difference between:

- Removing a name from a directory (`rm` command, short for `rmname`)
- Removing an element (`rmelem` command)

The former is reversible: it boils down to making a different version of a directory. The latter is not (unless by recovering from backup, but this involves restoring the state of the full vob, thus losing events more recent than the destruction of the element).

Restoring a file removed from a directory version does not require any special privilege or extraordinary skills. We'll review below the tools and the procedure, but first a couple of prerequisites: a recap on view extended path and a brief review of the version tree.

Looking at the view extended side of things

If we take a closer look at a particular version of some directory (using the version extended syntax), two interesting observations can be made:

- All the entries seem to be directories (even if the entry name refers to a file element)
- The timestamps and sizes do not directly relate to those in the directory listing inside a view:

```
$ ct ls -d .
.@@/main/mybranch/20 Rule: .../mybranch/LATEST
$ ls -l .
-r--r--r-- 1 joe jgroup 133 Jun 1 2010 file1
drwxrwxrwx 2 vobadm jgroup 0 Oct 31 2009 dir1

$ ls -l .@@/main/mybranch/20
dr-xr-xr-x 1 joe jgroup 0 May 18 2010 file1
drwxrwxrwx 2 vobadm jgroup 0 Jan 15 2009 dir1
```

The `file1` element represented as a directory, is actually the whole version tree of this element referred by its family name, `file1`, and it is the view's job to resolve it to a single actual version. We can navigate explicitly through the whole version tree to get to the same version as selected by the view:

```
$ ct ls file1
file1@@/main/mybranch/6 Rule: .../mybranch/LATEST
```

We would also find out that the element's "strange" date and size shown in its directory version listing are actually the date and size of its initial version, `/main/0` (given for reference purposes only, as indication of the version tree "root"):

```
$ ct ls file1@@/main/0
-r--r--r-- 1 joe jgroup 0 May 18 2010 file1@@/main/0

$ ls -l .@@/main/mybranch/20/file1/main/mybranch/6
-r--r--r-- 1 joe jgroup 133 Jun 1 2010 ##################################
                                .@@/main/mybranch/20/file1/main/mybranch/6
```

Version tree

One gets a good grasp of an element by displaying its **version tree**:

```
$ ct lsvtree stddefs.mk
stddefs.mk@@/main
stddefs.mk@@/main/0
stddefs.mk@@/main/mg
stddefs.mk@@/main/mg/1 (MG_3.38)
stddefs.mk@@/main/mg/5 (MG_3.39)
stddefs.mk@@/main/mg/6 (MG, MG_3.40)
stddefs.mk@@/main/mg/8
stddefs.mk@@/main/mg/foo
stddefs.mk@@/main/mg/foo/2
stddefs.mk@@/main/mg/p
stddefs.mk@@/main/mg/p/1
stddefs.mk@@/main/mg/p/p1
stddefs.mk@@/main/mg/p/p1/1 (TTT)
```

One recognizes the version extended path syntax, branches, and versions (leaves ending in a number), and as the last decoration in the default view, the list of labels applies to every version. Note that by default, only "interesting" (first, last, labeled, base of branches) versions are shown. One gets the same output for directory objects.

Let's make a note concerning the topological structure of the version tree: there may be many branches, cascaded on top of each others at various depths, but there is only one *main* (even if the type may be changed or renamed, with the risk of surprising users): this is something one cannot change. The main branch serves as basis for all the other branches. Otherwise, the main branch is not in any way more important than the other branches from the user perspective. The ClearCase novices often tend to exaggerate the importance of the main branch and are inclined to think that all of the development must be done in the main branch or at least eventually merged to it. Nothing of that kind! In ClearCase no branch is more equal than others!

This tool also offers a graphical option (`-g`) which, in our experience, is very popular among users; although we suspect it to be responsible for the creation of many evil twins —see later— as it allows the user to bypass, without warnings, the config spec selection. In the spirit of our *Chapter 1*, *Using the command line*, let's however note what (in addition to display space) one loses by using it: line orientation, i.e. the possibility to pipe to other tools, to filter and tailor one's output:

```
$ ct lsvtree -s stddefs.mk | \
perl -nle 'print qq(des -fmt "%Vn %Nc\\n" $_) if m%/[1-9]\d*$%' | \
  cleartool
/main/mg/1
/main/mg/5
/main/mg/6 optimized
/main/mg/8 tools in vobs and build.tag rule
/main/mg/foo/2
/main/mg/p/1 Peter's enhancement
/main/mg/p/p1/1
```

These are the perl-formatted commands for cleartool, instructing it to show only the version and the comment (if any), on the same line, this exclusively for non-0 versions (skipping the branches). Note the use of the powerful `-fmt` option, documented in the `fmt_ccase` man page. The focus is thus on showing only relevant information, in a minimalistic way.

We even actually use an `lsgen` tool, from a `ClearCase::Wrapper::MGi` CPAN module, to show only the last relevant *genealogy* of the version currently selected (that is the versions that have contributed to it), in reverse order from the version tree.

Recovering files

The first step in recovering a file is to find a former directory version in which it was still referenced. Several tools could be used to explore the version tree. We find that for directory elements, `lshistory` is more convenient than `annotate`. One has again to filter the relevant information, and thus first to present it in a suitable format. In the example which drove our previous chapter, we had, at some point, a "definitions" makefile for programs (`stdpgmdefs.mk`), which later became useless and was removed. In what directory version exactly?

To answer this, we'll use a command made of three parts in a pipe:

- Produce the history data
- Normalize the lines of the output
- Filter

The second part is unfortunately complex, especially as a one-liner: it is however boilerplate. The output of `lshistory` for directories in not line oriented: for checkin event, the version is displayed (we actually output first, before the version, a numeric time stamp for sorting purposes, which we drop at print stage), followed with the comment recorded. The latter is a multi-line list of differences between versions. The role of the Perl script is to reproduce the prefix information found on the first line of every event report, to the beginning of every following one. It performs its task in a very typical way for Perl scripts: by creating a *hash*, i.e. a data structure indexed by keys. The keys here are the prefixes (i.e. the time stamp plus version) information, and the data is a list of all the file changes. When the input is exhausted, in an `END` block, the script prints out its data structure, in two nested loops, with the outer one sorted on the time stamps.

```
$ ct lshis -d -fmt "%Nd %Vn:%c\n" . | \
 perl -nle 'next unless $_;
  if(/^(\d{8}\.\d{6} [^:]+):(.*)$/){
    $k=$1;push @{$t{$k}},$2}else{push @{$t{$k}},$_};
  END{for $k(reverse sort keys %t){
    for(@{$t{$k}}){$k=~s/^[\d.]+ (.*)/$1/;print"$k $_"}}}' | \
 grep stdpgmdefs.mk
/main/mg/2 Uncataloged file element "stdpgmdefs.mk".
/main/mg/1 Added file element "stdpgmdefs.mk".
```

The file was thus present in version 1, but removed already in version 2 of the `/main/mg` branch of the current directory.

Next, we check out the version of the directory into which we want to resurrect the file, and use the `ln` tool (link) to duplicate the entry found in version 1. We thus create a **hard link** (by opposition to *soft*, synonymous to *symbolic*, which the same tool would produce if used with an `-s` flag). "Hard link" is the technical term, in the UNIX tradition, for the *name* used to record an entry in a directory object. The same file object may thus be hard linked multiple times in different directories, or different versions of a directory, under the same or different names.

```
$ ct co -nc .
$ ct ln .@@/main/mg/1/stdpgmdefs.mk .
$ ct ci -nc .
```

Giving the directory as a target, as we did, we retain the original name of the file—`stdpgmdefs.mk` in our example here.

We could, in theory, have used the merge tool (we'll come back to it in *Chapter 7, Merging*). However, this would have offered us *all* the changes, interactively if we selected so. The number of such changes is variable, and may be large (we could check in advance, and even script the interaction). In the end, merge would however (for directories) create a hard link in exactly the same way.

Hard links

We have already mentioned **hard links** a couple of times. We saw that they may be used to select several versions of the same element (with scoped rules in the config spec), and that they may be used to restore removed files.

The *long* output of the `ls` command in UNIX mentions the number of hard links pointing to the same file (and thus in a vob, to the same element):

```
$ ls -dl bin
drwxr-xr-x 2 admin locgrp 1564 Jan 9 2009 bin
```

In case of directories, the number starts from 2 because of the "." alias, and is incremented by 1 for every sub-directory because of the ".." alias there.
These two special cases are handled in a special way, and never result e.g. in labeling errors (attempting to label multiple times).

Until a recent version of ClearCase, there was no direct support for finding the other names of a given element. This is now fixed, with two possible syntaxes:

```
$ ct des -s -aliases -all t1
t1@@/main/3
 Significant pathnames:
 /vob/tools/tls@@/main/1/t1
 /vob/tools/tls@@/main/1/t2


$ ct des -fmt "%[aliases]Ap\n" t1
/vob/tools/tls@@/main/1/t1 /vob/tools/tls@@/main/1/t2
```

Evil twins

This issue of recovering files brings us to the topic of **evil twins**, that is, the consequence of not caring for possible file recovery, while facing for a second time, the need for the existence of a file one removed at some point. First, let's stress that this plague is not an exclusive disease of ClearCase, quite on the contrary. The basic scenario is thus the following (only the barest bones!):

```
$ ct co -nc .
$ echo first > foo
$ ct mkelem -nc -ci foo
$ ct ci -nc .
```

then later:

```
$ ct co -nc .
$ ct rm foo
$ ct ci -nc .
```

and later again:

```
$ ct co -nc .
$ echo second > foo
$ ct mkelem -nc -ci foo
$ ct ci -nc .
```

The problem is that the two instances of `foo` are different elements that accidentally share the same name (in these two versions of the same directory). From a ClearCase point of view, they have nothing in common (as their name is not one of their properties): they have a different oid, a different version tree, a different history... You may recover the first one (under a different path name) and select arbitrary versions of both in the same view (i.e. the same software configuration), without ClearCase being able to inform you of possible inconsistencies. If they do share some commonalities from a semantics point of view (the user's intentions), an opportunity to communicate this to the tool was missed.

This last characterization goes deep into the spirit of SCM (or of the ClearCase supported new generation thereof): one must *use* the tool in order to benefit from it. This doesn't mean that the tool drives and the user obeys, but it tells that one doesn't quite get the power of a Ferrari if one drags it with oxen: a sophisticated tool serves its users in relation to their investment in it.

Something else ought to be understood from this abstract characterization: *evil twins* are not just a matter of the trivial scenario depicted above—this is just a particular case of a more general and deeper issue. Evil twins are not a topic to be handled away with crude (or even clever) *triggers* (more, or may be less, on them in *Chapter 9, Secondary Metadata*). It is utterly irrelevant to the real issue that the twins are *files* bearing the same *name* in the same *directory*: what matters is that the user missed an opportunity to represent a commonality in a way which would elect tool support.

And this is all what SCM is about: make it possible for the user to focus on creative activity, letting the tool take care of reproducing anything already done, earlier or elsewhere.

So... you meet evil twins, what can you do? Unfortunately, in the general case, nothing! Pick the one and hide the other away. But removing it (with `rmelem`) will break any reference to it, and especially any config record having used it. It is thus a matter of convention to decide whether to follow our earlier advice and leave a track of one's own uses, by applying a label: *to read is to write*—as you change the status of what you read, so why not make it explicit? In such a case, you might remove versions and even elements not bearing any *active* label (bound to a config record). Save the data as a new version of the twin retained (in a suitable branch, which sets requirements that we'll address in Chapter 6), and possibly move the *passive* labels (and other metadata) there. Such decisions obviously are subject to trade-offs and are difficult to justify.

Eclipsed files

We mentioned the word **eclipsing** in the paragraph on mkelem. This situation results from a conflict between a view private file and an element bearing the same name. It may happen in different situations, but the most likely scenarios involve changing one's config spec, thus making accessible some elements that were not available previously. In case of such a conflict, the view private file wins, i.e. *eclipses* the vob element.

This may easily be detected with the `ls` command:

```
$ ct ls foo
foo
foo@@ [eclipsed]
```

Of course, `catcr -union -check` would also complain about this file if it was used in a build.

It is simple to delete or rename away such files. Another way to handle them is to use the same method as for *stranded* files (see earlier, in the *lost+found* paragraph), with the following command or a variant thereof:

```
$ ct recoverview -dir $(ct des -fmt "%On\n" .@@) -tag mg
Moved file /views/marc/mg.vws/.s/lost+found/8000043e4c0d3fc3foo
$ rm /views/marc/mg.vws/.s/lost+found/8000043e4c0d3fc3foo
```

Writable copies

Eclipsing files may be an answer to a question we set aside at the beginning of this section: making writable copies of element versions, for tools requiring it. This is arguably a better answer than keeping the versions *checked out unreserved* (which is nevertheless still an option).

The third option which has our preference is to create a branch of the directory, from which to remove the elements and to place the view private copy there. The reason for our preference is the clarity of using a dedicated branch type, allowing for some sharing (not the actual data) among collaborators using the same tool. It also makes it convenient to compare the different copies (although with the checkout unreserved variant, one might use some other view for reference).

Differences and annotations

The analysis of changes we did in the case of directories is of course available for text files as well. We say "text files", but it is actually a matter of element types and their type manager support for the related functionality. As already promised, we'll treat element types in more detail as part of the secondary metadata in Chapter 9. Let's say now that the implementation of basic functionality is shared, so that `text_file` represents in fact more than itself and even more than its own sub-types.

As our readers may guess, we would not recommend the use of the graphical option of the `diff` tool, which results in putting all the responsibility for spotting interesting data onto the user, making it impossible to use any fancy tool to assist her. The default output of `cleartool diff` also surprises us, which presents the differences side by side, thus truncating the lines (under usual circumstances).

We mostly use the `-diff` (but `-serial` goes too) option to retain all the data in a greppable format.

Note that one may of course use external tools, such as `ediff-buffers` under GNU emacs, to compare multiple versions or add colors while retaining the advantages of text mode.

The `-pre/decessor` option might be the single reason to prefer the `cleartool diff` command to external diff tools (so handy that it is even implicit with ***ClearCase::Wrapper***):

```
$ ct diff -diff -pred stddefs.mk@@/main/mg/4
9c9
< LTAG := $(patsubst %so,%.tag,$(LIB))
---
> LTAG := $(patsubst %.so,%.tag,$(LIB))
```

The angle bracket prefixes obviously distinguish the two versions being compared; the line numbers and the type of change are expressed in compact format.

The `annotate` tool is less frequently used (having no counterpart in standard UNIX, although similar functions exist in several version control tools).

Let's admit that its interface is (powerful yet) cumbersome, so that one doesn't want to compute the suitable arguments to make the output greppable every time:

```
CLEARCASE_TAB_SIZE=2 ct annotate -out - -nhe -fmt \
 "%Sd %25.-25Vn %-8.8u,|,%Sd %25.-25Vn %-8.8u" -rm -rmf " D "\
 stddefs.mk | grep LTAGS
2010-05-16 /main/mg/3 marc  D LTAGS := $(patsubst %.so,%.tag,$(LIBS))
2010-05-16 /main/mg/5 marc LTAGS := $(addsuffix build.tag, $(dir ########
                                                          $(LIBS)))
```

We were interested in the definition of the `LTAGS` macro: this tells us that the current definition was introduced in version `/main/mg/5`, by `marc` on May 16.

Grepping is really essential with this tool because of the amount of data it may generate: annotate is in general too verbose to be useful otherwise!

This implies that you:

- Don't want to "elide" prefixes from any line
- Want to get rid of headers

You also typically want to align the content on the same column, after a prefix, and thus:

- Set fixed sized formats, relatively short
- Set them wide enough to be useful, and truncate the versions from the left (align to the right)
- Use a short tab size (2)
- Use a short format to indicate deletions: "D"

We have used this format with very little variation: mostly had to add the `-nco` flag when needed, and to trade the `-rm` one for `-all`, to get a view of other branches.

Misguided critiques

The subversion Red Book presents some surprising rhetoric in favor of the model it supports. It designates as *Lock-Modify-Unlock*, the alternative model, which would thus be, for naive readers, this of ClearCase (among other SCM products). The subversion model is elected to similar abstraction under the label: *Copy-Modify-Merge*. The problems attached, according to subversion authors, to the former model, are fortunately trivially and routinely avoided by ClearCase users: these are problems of the `/main/LATEST` syndrome, that is, serialization problems bound to the use of one single branch (per element).

A ClearCase model naturally aims at avoiding the negative aspect of *locking* (in fact **reserving** in ClearCase terms—locking being reserved for some other usage), that is, modifying the state of the system for the next user. On the other hand, it pays attention to the beneficial aspect of reserving: communicating to others an intention and the focus of an ongoing task.

ClearCase does offer a concept of unreserved checkout, but there is hardly any reason to use it (one should use branches). Anecdotic functionality justified by the mere fact it was cheap to implement. There also is an `unreserve` command to move a checked out version out of the way. But on the second thought, why would one try to undo what somebody else did and then ... to do exactly the same thing oneself? Perhaps a better idea would be to branch off to an own branch (see Chapter 6).

Another trendy commercial in favor of subversion rides the wave of *atomic commit*. This was even so powerful as to force some kind of support to be introduced in a recent version of ClearCase (7.1.1), on *customer demand*! No special support was ever needed in ClearCase to allow for atomic publication: this is the domain of labels. The concept of atomic commit builds on many wrong assumptions: that checkin and publication are necessarily coupled; that checkin always involves a mass of files, and thus a major discontinuity. These abominations should be corrected where they happen, instead of resorting to smoke puffing games.

Summary

This chapter was lighter than the previous one, wasn't it? We showed that ClearCase, despite being a ground-breaking monument in the history of SCM, does also handle neatly the most basic and humble functionalities of:

- Making new elements
- Making new versions of existing ones
- Giving a clear and manageable (lending itself to tailoring) access to the resulting complexity

All this without resorting to Shoot-Yourself-In-The-Foot GUIs and other fads of some misguided competition.

The next chapter is again more important in terms of ClearCase originality. It shows how the requirements set upon the local network, by the build avoidance mechanism, led ClearCase architects to design the precursor of all distributed SCM systems. In many ways, still ahead of its successors.

5
MultiSite Concerns

Not everybody is concerned with distributed development. However, it is wise to anticipate the move to a multisite environment, and to ensure that it would not result in a major discontinuity. There are many scenarios, most of which unpredictable from the developers' point of view, in which a **MultiSite** environment may become relevant: outsourcing, acquisitions, mergers, splits. It is important to note that some of them are incompatible with a central point of control. It is thus wise to design for flexibility.

Other concerns may lead to partition one's network, and to design a MultiSite solution, even on the same physical site. We'll see some issues related to performance in the next paragraph.

Imagine at any rate that a need arises suddenly: will it break the original single-site setup? Will this need to be redesigned completely? Or can the users continue working just as they have got used to so far? If the environment remains stable over such a change in development scale, we have a strong indication of a proper original setup, optimized for collaboration.

Our agenda for this chapter is:

- Push or pull?
- Replicas: avoid depending on mastership.
- Global types: yes. Admin vobs: no!
- Shortcomings of ClearCase MultiSite.

Distribution model

There is a common debate concerning the pros and cons of centralized versus distributed configuration management systems. Beyond the fact that it concerns mostly version control systems, this dichotomy actually misses the case of ClearCase, which is often wrongly considered as a representative of the former camp. The incorrect assumption, which adds to the common ignorance of MultiSite, is to tie distribution to a *pull* model (such as in *git* or *Mercurial*: users have to get, i.e. explicitly download the software they are interested in from remote sites), whereas the push model is associated with the centralized systems (such as CVS or *subversion*: users *commit*, i.e. explicitly upload their changes to the central server).

ClearCase MultiSite is a distributed system with an implicit *push* model: remote changes are made available to the user in the background, **asynchronously**.

This is a departure from the common culture of *synchronous* communications, and is probably felt as a loss of *control*, while the gain of *management* is not clearly perceived.

Synchronization, that is, reaching the *same state* in the *same time* (or meeting the same states in the same order), is only ever obtained by *waiting*, that is, adjusting to unavoidable time differences. This is inescapably a loss of efficiency and of scalability. The sophistication of the interactions between the ClearCase clients and servers (especially during builds: advertising the creation of derived objects, promoting local ones for remote winkin) made it clear to the architects of ClearCase that synchronized behavior was not an option.

Now, vobs are databases, the consistency of which is guaranteed by journaled transactions. There is a clear mismatch! Clearly every vob server has its own time, but this time is *local*, and there is no *global time*: the situation is one of relativity, similar to this of Einstein's mechanics. Propagation times are significant compared to CPU speeds, and cannot be ignored.

In fact, if we consider the normal behavior of ClearCase, let's say during a build, we are writing to a vob database, and all these events have to be serialized, that is, we are constantly acquiring database locks, executing transactions, and releasing the locks. This happens between the client and the server, so that the duration of the process includes the round-trip time of the IP packets between the two hosts, and this time is incompressible! It is bound to the speed of the signal and the physical distance (even if in practice, the propagation of the signal is the slowest due to its treatment in switches, routers, and firewalls). The most important fact in the communications is thus not the bandwidth, which has been

increasing steadily, but the *latency*. In fact, measured in clock cycles (and thus in opportunities lost while waiting), the impact of latency is continuously getting more serious.

This is why ClearCase is optimally used within a 0-hop LAN, and any activity involving remote hosts should happen in distinct sites, and thus involve MultiSite. The trade-offs unfortunately involve a loss of functionality over MultiSite, and of MultiSite scalability.

Multitool, and MultiSite Licenses

ClearCase MultiSite is presented as an add-on product, with distinct licenses: one needs a MultiSite license as soon as one accesses a replicated vob. This is of course a major discontinuity and a questionable commercial decision, which may have limited the historical use of ClearCase.

Being a distinct product, it came with its own tool: **multitool**, (which we'll alias as `mt`) for running MultiSite specific commands. Over time, several multitool commands have migrated to cleartool as well (and backwards such as `cd` and `pwd`), as using both is often inconvenient (especially when using a background session to which to pipe commands: then one would need two such sessions, and need to keep the two in sync). There remain however some commands that require the use of multitool. We'll use some below.

Replicas and mastership

The solution to the dilemma is for every site to have, not an identical copy of the vobs, but a **replica** of it: this replica is a workspace which can only be modified locally, and is prevented by the system from diverging from the other replicas. In the absence of a global time, it would be meaningless to say that the data is the same: by the time one would have asserted the state on one site and brought this information to another site for comparison, it would be unavoidable that this state would have changed in either or both replicas. What is important is that the differences do not grow, and that the *past*, up to a recent date, is common.

This is implemented in the concept of **epochs**: ever growing figures that represent high-water marks: the state reached from the point of view of every replica.

We'll review here the commands that may concern users, interested in tracing the origin or the status of changes. We'll have a later *Chapter 11*, *MultiSite Administration*, on administrative aspects.

First, know who you are and who are the partners in communication, both by name and by virtue of the servers:

```
$ ct des -fmt "%[replica_name]p\n" vob:.
wonderland
$ ct lsrep -fmt "%n %[replica_host]p\n"
wonderland beyond.lookingglass.uk
sky alpha.centauri.org
```

Then find out when was the last import and what is the resulting status in terms of the epoch number of the remote replica.

```
$ ct lshis -fmt "%d %o\n%c" -last replica:wonderland
2010-06-11T16:10:40+01 importsync
Imported synchronization information from replica "sky".
Row at import was: sky=377 bigbrother.deleted=565 wonderland=1206
$ mt lsepoch wonderland | grep '(sky)'
 oid:6ba49d09.011621db.8ab1.00:01:93:10:fe:84=378 (sky)
```

If we have connectivity to the remote server, we can compare with its own values:

```
$ albd_list alpha.centauri.org > /dev/null
albd_server addr = 123.1.2.3, port= 371

$ multitool lsepoch -actual sky | grep '(sky)'
 oid:6ba49d09.011621db.8ab1.00:01:93:10:fe:84=378 (sky)
```

This is however unlikely. You have a better chance from your vob server (it must be connected to some ClearCase server, but it may not be directly to the remote vob server, rather an intermediate shipping server). How it is connected, at least from the point of view of shipping its own packets, will show with:

```
$ cd /var/adm/rational/clearcase/config
$ egrep '^[^#]*ROUTE' shipping.conf
```

If this shows nothing, then the connection is direct—to any hosts one tries to ship to, which may only be a subset of the servers hosting replicas of your vob (in case a delegation scheme is in place, using one kind of *hub*). This is, assuming the setup uses the ClearCase **shipping_server** mechanism, which is likely, but not necessary. This setup guarantees a high degree of transparency and orthogonality between the ClearCase servers having the direct connections to one another (the prerequisite is that the above mentioned `albd_list` command succeeds). The following commands will allow you to investigate the history and the schedules:

```
$ ct getlog -host alpha.centauri.org -inquire
vob ClearCase vob_server log
view ClearCase view_server log
sync_import MultiSite import synchronization log (unformatted)
```

```
shipping_receipt Multisite shipping receipt log
shipping MultiSite shipping_server log
...
```

This command shows all the various ClearCase logs that are accessible to you from the remote host (`alpha.centauri.org`). All these logs can be fetched by the same `cleartool getlog` command, and each particular log can be referenced by its name specified in the output of the `-inquire` command above, e.g.:

```
$ ct getlog -host alpha.centauri.org shipping
============================================================
Log Name: shipping Hostname: alpha.centauri.org Date: #######
                                    2010-06-23T10:00:37+01:00
Selection: Last 10 lines of log displayed
------------------------------------------------------------
```

This command returns a list of all the scheduled ClearCase jobs on the remote host:

```
$ ct sched -get -host alpha.centauri.org
```

The concept of **mastership** guarantees that inside every object, there is an area exclusively assigned to the local replica, in which local users may write without consideration for anything that might happen on other sites. The price of considering this would be exorbitant: it would be synchronization. Note that mastership works like a dimension orthogonal to any structure of the software configuration.

```
$ ct des -fmt "%[master]p\n" brtype:main
sky@/vob/apps
```

This preceding example command line shows that the mastership of the main branch type belongs to the `sky` replica of the `/vob/apps` vob.

We wrote inside *every object*, and this works well for elements, because it suits the concept of branches, which may be used for this purpose (the summary will wait until next chapter). But it doesn't work that well for **metadata**: for metadata, we need to decide whether to resort to conventions, or whether the item (usually a type) is stable enough (remember that the concern is only relevant for modifications) to be shared between the replicas.

Avoid depending on mastership

It is possible to change the mastership of objects. As administrator, it is always possible to grant one's mastership to some other replica, and to *send* it. Note that it takes time, and as such, is irreversible. Once you have sent the mastership, you depend on the other site to send it back to you. Or it is also possible to configure the system so that one may request the mastership of certain objects. This does also require time. In both cases, the underlying technology is this of creating a

sync packet, containing all the **oplogs** with numbers below the event one is interested in, to **ship** this packet, and to **import** it at the destination. The `reqmaster` command tries to take some shortcuts to make it faster, but whether it is possible depends on the network configuration, and ultimately is not always safer (one still needs that all the previous oplogs sent previously have been delivered and imported, in order to be able to import *the* oplog...).

We are not trying to paint things black: this is possible, and even works most of the time. And when it doesn't, it is possible to fix. But, it is always better not to depend on it at all! Even more: suppose some amount of mastership transfers happens, one is better to avoid by all means depending on anything related to it, just because it opens the door to surprises and instability: it introduces a global state of the system.

So, let's not change mastership. The remaining danger is unfairness: we may still easily be left with some sites *more equal than others*. One site owns *the* main branch type (do you remember? We insisted that *topologically*, there is only one in the version tree). If the process mandates that the official versions must be found on *main* branches, then one site is structurally in a different position than others! The situation doesn't change an iota by selecting any other type for the *integration* branches.

So, our goal will be to answer the simple question: how to avoid depending on mastership?

The answer is surprisingly simple: read from anywhere, write only to objects (that is, to branches in the case of elements) you know you master, because you created them. Share and publish **in-place**.

In this radical simplicity, this common sense rule defeats 90% of the processes usually presented as best practices, including the infamous UCM.

This rule rejects all concepts of *main* or *integration* branches: never write there, especially for publication!

Branches

Create branches: this is always possible, if you use a type you created. So, in order to shoot yourself in the foot, decide to use a predefined type. As long as you don't, you are free. Branch types should thus not be bound (hardwired) to an intention: keep them a commodity. Stick to this from the beginning, and neither MultiSite nor mastership can force you to change your way.

Do not design a hierarchy: not everybody would be allowed to create the lower branches in elements where they wouldn't exist.
Avoid cascading: prefer branching off main branches (or whatever the root branch type is named). One cannot cascade forever: sooner or later, one meets system limitations. Choose a policy which preserves the topology of the system for the next user: the *bush* model.

Labels

If branch types should be kept free from meaning, it is that semantics are better held by label types. The cause of the fundamental asymmetry in ClearCase between branches and labels is simple: labels can be aliased, whereas branches cannot:

```
$ ct des -s foo@@/AAA foo@@/BBB | sort -u
foo@@/main/3
$ ct des -fmt "%Nl\n" foo@@/main/3
AAA BBB
```

MultiSite brings however again some far reaching constraints. The namespace of types is flat and shared among all replicas. This means that any given name may be reserved by the fastest replica, and preempt the ability of others to use it (there is a mechanism to resolve conflicts afterwards, but nobody wants the discontinuity implied by such occurrences). Furthermore, because of the *asynchrony* of replication, it is possible for this event to occur *retroactively:* every time a packet gets imported, it is the past that changes (note by the way, that this affects time rules in config spec as well as labels). Of course, the probability of such accidents is small, but it is a good idea to use conventions to make it smaller yet. Conventional namespaces, or pre-assigned families of names, play this role.

Another issue related to MultiSite is that while the same label name may be associated across several vobs to the same *baseline* (that is, reference software configuration), sync packets are per vob, and independent from others. The result is that there is no guarantee that a baseline drawn across several vobs will be synchronized atomically or even only consistently in time.

To wrap up the considerations brought by MultiSite, we advise to:

- Avoid *sharing* label types: to put it clearly, avoid `mklbtype -shared`.
- Use only *local* label types, that is, types created locally. This refers both to *applying* them (one cannot apply unshared types mastered on other replicas), and using them directly in config specs (you never know whether you imported *all* the "make label" events). Note that you may (and should) use remote labels as a basis to apply local ones, for integration purposes.

- Acquire conventional sub-domains (maybe only prefixes) and create new types within them (to avoid collisions).

A last piece of advice which only marginally relates to MultiSite: we recommend to lock label types, especially the ones you intend to move (`mklabel -replace`). Lock them, and unlock them as needed, of course: this gives a simple guarantee that they have been stable after a certain time stamp. Locks are not replicated (and it doesn't make much sense to lock a type mastered elsewhere, and which one therefore cannot apply), apart for *obsolete* locks (`lock -obs`). Obsoleting types requires mastership.

Other types

We shall deal with other metadata types in more detail in *Chapter 9, Secondary Metadata*, but we need to anticipate this slightly here. Similar considerations as reviewed for label and branch types apply roughly. The option of sharing types is reasonable for stable types, created once for all, and only used later. This results in less types and in smoother collaboration. The trade-offs depend however clearly on usage patterns: for example, attribute types bound to label ones (we'll give examples already in the next chapter) should clearly *not* be shared, and their name should also be taken within conventionally pre-acquired domains.

The case of element types is special and needs to be noted, when custom element type managers are defined and installed. ClearCase MultiSite does nothing whatsoever to replicate the managers (the built-in tools are installed under `/opt/rational/clearcase/lib/mgrs`, which makes it the natural place to put custom additions) to other sites: this responsibility is left entirely to administrators, and this meta-responsibility to the users... But no bother: one will be reminded of the existence of element types managers at the time of failing to import sync packets on a remote site!

```
multitool: Error: Operation "create_element" unavailable for manager ####
                                                                    "bar"
   (Operation pathname was: ##############################################
                    "/opt/rational/clearcase/lib/mgrs/bar/create_element")
multitool: Error: Unable to store new version.
multitool: Error: Unable to replay oplog entry 1800: error detected by ##
                                                      ClearCase subsystem.
1800:
op= mkelem
```

Global types and admin vobs

We saw that types are vob specific, and also that config spec rules based on them actually use their names only. This situation is clearly suboptimal, and may be improved by using **global** types, that is, creating types with the `-global` flag, which allows one to set up hyperlinks of `GlobalDefinition` types. This affects the way types are locked, renamed, and removed (together). This, however, creates a relationship between vobs:

```
# Site 1
$ ct mklbtype -nc -global FOO@/vob/server
$ ct mklbtype -nc FOO@/vob/client
$ ct mkhlink GlobalDefinition \
 lbtype:FOO@/vob/client lbtype:FOO@/vob/server
```

Describing the type in either vob will now show the same information: the one stored on the server side, i.e. about the global type:

```
$ ct des -ahl -all lbtype:FOO
FOO
 Hyperlinks:
 GlobalDefinition <- lbtype:FOO@/vob/client
```

There are, however, two different objects: on the client side, the type is a local copy, which may be checked using the `-local` flag:

```
$ ct des -ahl -all -local -fmt "%[type_scope]p" \
 lbtype:FOO@/vob/client lbtype:FOO@/vob/server
global Hyperlinks:
 GlobalDefinition <- lbtype:FOO@/vob/client
local copy Hyperlinks:
 GlobalDefinition -> lbtype:FOO@/vob/server
```

The MultiSite replication system does nothing particular to support (wasn't originally designed to support) the consistency of this inter-vob relationship. This means that one may have a replica of one vob (`/vob/server`, hosting the global definition), but not of the other (`/vob/client`), and the hyperlink will then show up as "dangling":

```
# Site 2
$ ct des -ahl -all lbtype:FOO@/vob/server
FOO
 Hyperlinks:
 ? <- <object not available>
```

One may get more information (the oid of the link type, link object, vob, and the other end object) with `ct dump`, although this amounts to what was contained in the `ct des -local` report.

As such, this doesn't cause any problem, beyond some confusion among remote users.

This may become annoying only if they decide to clean up the glitch, and use the following command as vob owner on their site:

```
# Site 2
$ ct checkvob -hlink lbtype:FOO
```

The problem being that this will succeed and thus destroy your information! Even when this happens, it leaves a trace in the history, which may be used to communicate and avoid the same problem in the future. To be precise, the information is not destroyed: the hyperlink just gets detached from the `/vob/server` label type (on both sites) and remains attached to the `/vob/client` label type:

```
# Site 1
$ ct des -ahl -all -local -fmt "%[type_scope]p %n\n" \
 lbtype:FOO@/vob/client lbtype:FOO@/vob/server
local copy FOO
 Hyperlinks:
 GlobalDefinition -> lbtype:FOO@/vob/server
global FOO
```

Note that the situation is different, if, on contrary, the non-replicated vob is the one hosting the global definition (`/vob/server`). In that case, the error report is the one as shown here:

```
# Site 2
$ ct des lbtype:BAR
cleartool: Error: Unable to find replica in registry for VOB with #######
                      object ID:"046d37bf.7ac511df.9a89.00:01:84:a9:f4:34"
cleartool: Error: Unable to locate versioned object base with object ####
                           id: "046d37bf.7ac511df.9a89.00:01:84:a9:f4:34".
cleartool: Error: Trouble finding the global definition for local type ##
                                                                    "BAR".
$ ct des -local -ahl -all lbtype:BAR
BAR
 Hyperlinks:
 GlobalDefinition -> <object not available>
```

In this case, the `checkvob` command will fail, as will any `rmhlink` one (and `lstype` will complain and return an error code, which may affect scripts):

```
$ ct checkvob -hlink lbtype:BAR
Unable to determine if the following hyperlink is intact.
GlobalDefinition@129@/vob/client lbtype:BAR@/vob/client -> <object not ##
                                                                  available>
Delete it? [no] yes
cleartool: Error: Unable to find replica in registry for VOB with #######
                     object ID:"d5b61d1f.c73211db.8b18.00:16:35:7f:04:52"
cleartool: Error: Unable to locate versioned object base with object ####
                           id: "d5b61d1f.c73211db.8b18.00:16:35:7f:04:52".
cleartool: Error: Unable to perform operation "remove hyperlink" ########
                                    in replica "sky" of VOB "/vob/client".
cleartool: Error: Master replica of hyperlink is "wonderland".
cleartool: Error: Unable to remove hyperlink ############################
  "GlobalDefinition@129@/vob/client lbtype:BAR@/vob/client -> <object not
                                                                 available>".
```

Now it actually obeys the mastership rules unlike in the previous case we saw.

One might say that this encourages the anti-social behavior of not replicating the source of one's information, but such a strategy would be cumbersome and unsustainable in the long range, if wanting to create global `lbtypes` from different server vobs.

In summary, global types are thus a useful feature, even if a low-level one, and as such supported in extremely minimal ways: even the `cptype` command would not create a client copy of a global type—one must create the `GlobalDefinition` hyperlink explicitly!

Built on top of them, there exists the more ambitious concept of *admin vobs*. Admin vobs build up a hierarchy of vobs, linked by hyperlinks of type `AdminVOB`, and offering shared definitions of global types missing in the lower vobs of the tree as they are being requested (for example, while running `ct mklabel` for a type not yet defined in the current vob). Note that for this mechanism to work, one still has to create the types explicitly as `-global`, and in the admin vob: types created in the client vobs are by default `-ordinary`.

The same issue as previously mentioned for `GlobalDefinition` hyperlinks exists with `AdminVOB` ones, but its consequences may be more severe: it is possible that packets for the replica of a client vob cannot be successfully imported in the absence of the admin vob.

Admin vobs bring though a conceptual problem: they bind all sites to one and the same hierarchy, and thus at least to a common set of vobs. In this respect, they defeat the fairness of the MultiSite design. If we remember the discussion which started this chapter, about the various scenarios along which one might get pushed to use MultiSite, one clearly sees that such uniformity doesn't suit the flexibility requirements considered. It is easy to conceive scenarios in which one would "inherit" admin vob hierarchies from various origins, which would be utterly cumbersome to merge, thereby leading to chaos.

Our recommendation concerning admin vobs is thus, based on MultiSite concerns, to avoid using them at all! On the contrary, global types may be used: some scripting will raise up the functionality to a reasonable and useful level.

Shortcomings of MultiSite

We saw in the beginning of this chapter that there could be advantages in partitioning a large local network into several *sites* (in the MultiSite sense of the word), and we alluded to trade-offs concerning the loss of functionality.

It is now time to clarify these trade-offs, that is, to give reasons not to use MultiSite in spite of the benefits this might bring.

Our reader will not be surprised if we stress here the complete lack of support for derived objects. This is sometimes worked around by naive users through *staging*, that is, by checking them in as *versioned derived objects*, partly for the purpose of replicating them; we already mentioned this in *Chapter 3*, *Build Auditing and Avoidance*. We believe that this is throwing the baby with the bath water. Versioned derived objects are certainly still derived objects, in that they retain their config records, but they lose what makes config records valuable: that clearmake uses them to implicitly *manage* derived objects—clearmake identifies derived objects by their config records, which allows it to avoid their useless duplication. This shift of responsibility from users, or administrators, to the tool, thus freeing human resources for more creative and more complex tasks, is what potentially makes ClearCase a next generation SCM system.

Furthermore, as we already mentioned, we do not believe there is much value in replicating the data of derived objects: the makefile system should be optimized to make it faster to build locally than to download binaries. At any rate, one shouldn't trust binaries that one cannot build oneself, or rather, that one cannot analyze locally. At the SCM level, such analysis can only be generic, that is, based on comparison. This is the road to enhancement: MultiSite should replicate the config records, as config records. This means that no derived object data would need to be replicated,

but the config records would be available for tool supported comparisons. As we already noted, at the moment one can only emulate this by explicitly checking in the config record:

```
$ ct diffcr foo .@@/main/cr/1/foo/main/mg/1
< Target foo built by mg.user
> Target foo built by mg.user
< Reference Time 2010-06-20T14:48:26+01, this audit started ##############
                                                    2010-06-20T14:48:26+01
> Reference Time 2010-06-20T14:44:32+01, this audit started ##############
                                                    2010-06-20T14:44:32+01
----------------------------
MVFS objects:
----------------------------
< /vob/test/t@@/main/mg/1 <2010-06-20T14:44:45+01>
> /vob/test/t@@/main/cr/1 <2010-06-20T14:43:28+01>
----------------------------
< /vob/test/t/foo@@--06-20T14:48.16247
> /vob/test/t/foo@@/main/mg/1
```

Note that in the above transcript, one object is a derived object and the other a derived object version, actually checked in with the `-cr` option.

Note also that we checked it in in a special branch `cr`, conventionally reserved for this purpose, of its parent directory, which allowed us to retain the exact path name, whereas avoiding to leave a read-only file in the way of our normal builds. To update this element, we use a different view, in order to avoid altering the original DO and test the effects of the `-cr` option:

```
$ ct setview v2
$ ct co -nc -nda foo
$ ct ci -nc -cr -from /view/v1$(pwd)/foo -rm foo
Checked in "foo" version "/main/mg/2".
$ ct des -fmt "%[DO_ref_count]p %[DO_kind]p\n" foo
 unshared
$ ct co -nc -nda foo
$ ct ci -nc -from /view/v1$(pwd)/foo -rm foo
Checked in "foo" version "/main/mg/3".
$ ct des -fmt "%[DO_ref_count]p %[DO_kind]p\n" foo
 shared
$ ct setview v1
$ ct des -fmt "%[DO_ref_count]p %[DO_kind]p\n" foo
1 unshared
```

The conclusion is surprising (to us): neither the checked in config record (not promoted to the *shared* status) nor the plain checked in DO (which, conforming to the documentation, *is* promoted to the shared status) has a **reference count** or affects the reference count of the original object (displayed last).

What this shows is a relatively recent change in ClearCase, which we had not noticed: until version 2003.06, the journaling record for checkin events used to have a reference to the derived object, which was only released by *scrubbing oplogs* (we'll come back to oplogs scrubbing in *Chapter 10*, *Administrative Concerns* and *Chapter 9*, *Secondary Metadata*). It used to be a common cause of problems, because users would want to remove versions of large binaries, which resulted in dangling pointers inside the oplogs and errors while importing sync packets. It seems that the versioned derived object is now a full duplicate of the original DO (with the result that it is being shared or not is rather pointless), which is a quite radical fix.

Versioned derived objects are thus simply not visible to `lsdo` and cannot (probably) be winked in:

```
$ ct lsdo foo
--06-20T15:27 "foo@@--06-20T15:27.16250"
```

On the remote site:

```
$ ct des -fmt "%n, %m, %[DO_ref_count]p, %[DO_kind]p\n" foo
foo@@/main/mg/3, derived object version, , shared
$ ct des -fmt "%n, %m, %[DO_ref_count]p, %[DO_kind]p\n" foo@@/main/mg/1
foo@@/main/mg/1, derived object version, , unshared
$ ct lsdo foo
cleartool: Error: Not a derived object: "foo"
```

These versioned derived objects are thus there only for explicit manipulation. An idea for making them more useful would be (some kind of request for enhancement) to make it possible to promote them to a status related to the real derived objects, that is, to support *lazy identification*: allowing them to work as a bound between sets of derived objects built on different sites.

Summary

We showed that MultiSite is an integral part of ClearCase, and that its design has deep roots into requirements set by the sophistication of build management. It is thus essential to take it into consideration while designing the development process.

6
Primary Metadata

ClearCase offers auxiliary objects to help managing files: **metadata**. The different metadata types are not equally important. One way to trace a boundary is to elect as *primary* the ones appearing in the version extended view paths: **labels** and **branches**.

We do here a first pass at handling these objects, leaving deeper and less critical aspects for later.

- Singling out labels and branches
- Types and instances
- The relative roles of labels and branches
- Labels: floating and fixed
- Baselines and incremental labels
- The delivery process

Metadata in the version extended view

From the version extended perspective, we notice at once that labels and branches share a common namespace: it is not easy a priori to tell them apart. The distinction between the two has to be built on convention. It is customary to follow the example set by ClearCase itself for predefined types: uppercase for labels such as `LATEST` and lowercase for branches such as `main`.

To illustrate this, let us compare the element extended version path and its version tree:

```
$ ct lsvtree
.@@/main
.@@/main/14 (REL1)
.@@/main/38
.@@/main/br1
```

```
.@@/main/br1/6 (L1)
.@@/main/br1/t1
.@@/main/br1/t1/3 (PUB3.5)
.@@/main/br1/t1/4
.@@/main/br1/8
.@@/main/45
.@@/main/br2
.@@/main/br2/2
.@@/main/46
```

The `lsvtree` command without the `-all` option shows only *important* versions: that is, labeled (`/main/14`,`/main/br1/6`, and `/main/br1/t1/3`), last on their branch (`/main/br1/8`, `/main/br1/t1/4`, `/main/46`, and `/main/br2/2`), and versions of which new branches were spawned (`/main/38`, `/main/br1/6`, and `/main/45`).

The element's extended view shows a "projection" of the version tree: on the top level, it shows the first-level branches (`main`) and all the shortcuts, i.e. the labels that give access to a particular version of this element.

```
$ ll .@@
total 8
drwxrwxrwx 2 ann jgroup  223 Feb  4 2008 REL1
drwxrwxrwx 2 ann jgroup  932 Feb 14 2008 L1
drwxrwxrwx 4 ann jgroup    0 Sep  3 2007 main
drwxrwxrwx 2 ann jgroup 1159 Aug 25 2008 PUB3.5
```

The second-level branches (`br1`, `br2`) can be found as sub-directories of the main directory:

```
$ ll .@@/main
total 90
drwxrwxrwx 2 jgroup 2000    0 Sep  3 2007 0
drwxrwxrwx 2 jgroup 2000   52 Sep  3 2007 1
...
drwxrwxrwx 2 jgroup 2000    0 May 18 2009 br2
drwxrwxrwx 3 jgroup 2000    0 Sep 13 2007 br1
drwxrwxrwx 2 jgroup 2000  223 Feb  4 2008 REL1
drwxrwxrwx 3 jgroup 2000 1034 Sep 16 2008 LATEST
```

The other subdirectories are explicit versions: 0, 1, ... 46; and the labeled versions, on *this* branch: `REL1`, `LATEST`.

Note that for directory elements (as in our example above), all the entries (both labels and branches) are directories, whereas for file elements, *label* entries are always files as they refer to a particular version of this *file* element. *Branch* entries are always directories as they contain at least the following versions for each branch: `0`, `LATEST`:

```
$ ll foo.html@@
total 9
dr-xr-xr-x 3 ann 2000     0 Feb 14 2008 main
-r-xr-xr-x 1 ann 2000 10927 Sep  9 2008 L1
$ ll foo.html@@/main
total 5
-r-xr-xr-x 1 ann 2000 0 Feb 14 2008 0
dr-xr-xr-x 3 ann 2000 0 Feb 14 2008 br1
-r-xr-xr-x 1 ann 2000 0 Feb 14 2008 LATEST
```

Types and instances

ClearCase consistently follows a philosophy (born in *static typing*) of distinguishing between declaration/definition on the one hand and use on the other. We shall therefore systematically meet **types** and **instances** as we just did. Note that this is proper to ClearCase, and sometimes confusing to users with other backgrounds. Indeed, verification is always based on the consistency of representations, and therefore requires some degree of duplication. Splitting tasks in two steps, often separated in time and space, allows such verifications. We shall refer to this philosophy later, trying to apply it ourselves where support for it is not built in (see *Use of locking* section).

It is somewhat awkward to consistently talk of **label** and **branch** types, so that we shall, in non-ambiguous contexts, follow the common practice of using the terms *labels* and *branches*. For example, we shall speak of *applying a label*, when more rigorously, we *create* several *instances* of a single *label type*.

Functions will come in pairs: first make **type** (`mklbtype` and `mkbrtype`, for the kinds of metadata we consider here) for the declaration, and then make **instance** (`mklabel` and `mkbranch`) for the actual use.

Both label and branch types are abstract, element-independent concepts. On the contrary, instances are concrete and bound to a particular ClearCase element (either a file or a directory). A label is *applied to* it, a branch is *created on* it.

```
$ ct mklbtype -c "a concise comment, only if useful" MYLABEL
$ ct ls foo
foo@@/main/4                          Rule: /main/LATEST [-mkbranch br1]
$ ct mklabel MYLABEL foo
$ ct des -fmt "%n %Nl\n" foo@@/MYLABEL
foo@@/main/4 MYLABEL
```

We created a label type `MYLABEL` and applied a label of this type to a version `/main/4` of the element `foo`. Then we showed how to access the labeled version directly by specifying the label:

```
$ ct mkbrtype -nc br1
$ ct catcs
element * CHECKEDOUT
element * .../br1/LATEST
mkbranch br1
element * /main/LATEST
$ ct co -nc foo
Created branch "br1" from "foo" version "/main/4".
Checked out "foo" from version "/main/br1/0".
```

We first created a branch type `br1`, and then created the branch by setting a `mkbranch br1` config spec rule and checking out the element, which resulted in the `br1` branch creation of the `foo` element. The `br1` branch was spawned off the view-selected version `/main/4`. Branches can also be created explicitly by the `ct mkbranch` command (although the implicit branching described above is more common):

```
$ ct mkbranch -nc br1 bar@@/main/br/2
Created branch "br1" from "bar" version "@@/main/br/2".
Checked out "bar.txt" from version "/main/br/br1/0".
```

One more observation one can make in the last example is that the ClearCase branches *cascade*. For example, if we branch off, using type `br1` from a version `/main/2` of the element, this creates the `/main/br1` branch of it; then continuing to branch off using now `br2` from version `/main/br1/1`, we get branch `/main/br1/br2`, and so on. This is why, in the config spec, it is convenient to specify the branch type as a wildcard, such as `.../br2/LATEST`, rather than with its full name as `/main/br2/LATEST`, as the latter may not select all the desired versions (as there can be both elements with branches `/main/br2`, and `/main/br1/br2`, and so on).

Note that one cannot cascade branches indefinitely because of system limitations (of 1024 bytes for a full version extended pathname). Besides, it is not very handy either. One may be interested to take a look at the MGi extension of the ClearCase Wrapper, providing support for `BranchOff: root` rule in the config spec: it forces new branches to be created from the root (usually `/main`) rather than following the cascading mode. It maintains the genealogy with **Merge arrows** (see *Chapter 7, Merging*).

Each vob must have one and only one definition for each branch or label type, which it may inherit from another vob if using global types.

The following exclusion principle applies, by default, to branch and label instances: every element can have only one version carrying a particular label, and only one branch of any given type in its version tree (see the version tree of a current directory in the preceding example).

This may be bypassed at label type creation by using the `-pbranch` option, which allows to use the same types on different branches of an element; we cannot recommend this in general, as this leads to the possibility of ambiguous rules in config specs.

Labels or branches?

There is a great deal of symmetry between the two sets of functions (pertaining to labels and branches), and actually between the concepts. Both may be used in config specs, with very similar effect at first sight. The strategies built upon them are *dual*, to use a metaphor familiar to the electric engineer thinking of currents and tensions when considering a circuitry schema. However, as in the electrical case, this duality eventually reaches some boundaries, and we'll see which. It must be stressed that such analyses are very specific to ClearCase, and bear no validity at the abstract level of conceptual CM; they don't apply to other tools.

We already mentioned in the last chapter the main difference between labels and branches: labels can be aliased whereas branches cannot, or in other words, a given version may bear as many labels as one likes but it sits only on one single branch.

This is the technical aspect. On the functional side, one might think that branches embody *intentions* (the future), whereas labels represent *states* (the past). Now, this division is not always respected in practice, especially in config specs. This is particularly true in the context of UCM, where both labels and branches are treated as implementation details, and buried under the higher-level concepts of *activities*, *streams*, and *projects*. We shall ignore them here, not because we believe the concepts themselves would not be sound (with a reserve concerning the "project" one), but because their implementation in UCM is built upon an unfortunate foundation. This will become obvious in the following, although we won't mention it anymore until *Chapter 13*, *The Recent Years' Development*. As mentioned in *Chapter 2*, *Presentation of ClearCase*, UCM is largely covered in existing literature, and we focus here on issues ignored therein.

What needs to be stressed is that metadata can and should be used to implement communications between the members of the development team, in an objective way, hence supported by the tools. Interpretation remains necessary, but is pushed forward, or upwards if one wants to retain the "high-level" metaphor. This may and should be enforced by conventions. We shall review here, *bottom-up*, the constraints that drive the making of sensible conventions.

Parallel development

Derived objects are shared transparently under ClearCase. This happens through build avoidance, winkin, and DO identification. However, nothing of this is spread via replication through MultiSite! The alternative while using MultiSite is thus to be down to the rudimentary level of version control, or to do something to raise it back to the full ClearCase power.

Applying labels serves the latter purpose: config records are not replicated, but labels applied using them are. The sharing of derived objects is not transparent, but it is reasonably easy to reproduce them on another site, or to compare the bill of materials of objects produced locally with this of objects produced remotely.

Also, every developer needs a workspace under her own control, both protected from interferences from others and easy to update with the latest relevant changes. We saw in Chapter 5 that MultiSite considerations direct to work in branches, which the developer should create herself on a need basis. We will defer to the next chapter, on merging, the details on how to update them, but will address in the following section, the questions of delivering her work thus releasing the protection acquired while creating the branches, and also specifying clearly to her collaborators, the changes from which they might be willing to update their own environment.

Config specs

The place of choice to express semantics in such a way that the tools will obey them is the config specs. This fights the common practice of *generating* them, using once again, *higher-level* considerations.

One concern will be to avoid the pitfall (already mentioned in Chapter 2) of growing the complexity of config specs to a point when their generation might meet a demand from users.

To be convenient, config specs should be both simple (and concise) and stable.

We already mentioned (in Chapter 2) the first requirement: few generic rules so that the user can have a clear idea of which of them applies to any given element. This ought to be governed by simple considerations, ideally either of two: her own changes on one hand, or the common baseline applying to her situation on the other—maybe in addition, some intermediate level changes, shared with close collaborators, on top of the aforementioned baseline.

The second requirement comes from the fact that config specs are not themselves versioned. Since they are not managed, they shouldn't change. Optimally again, the user shouldn't have to modify her config spec in the following two most common scenarios (otherwise, this would be redundant and distracting):

- As the common baseline (again, specific to her situation) would be updated
- As she would herself deliver her work

Getting back to the example of the *useful* config spec from the Chapter 2:

```
element * CHECKEDOUT
element * .../branch/LATEST
mkbranch branch
element * LABEL
element * /main/0
```

Here, the fourth line (`element * LABEL`) represents the common baseline, that is, the set of versions labeled with `LABEL`. In case this baseline gets updated (e.g. the label `LABEL` is moved to different versions after a bug fix), the changes will be reflected instantly in the user's view, and the new versions will be selected.

The user's own changes are done in the branch named `branch`, and selected here with the second line (`element * .../branch/LATEST`) specifying that the user wants to select her own changes first. The third line (`mkbranch branch`) is about driving checkouts to branch off the versions carrying `LABEL` labels, or from `/main/0` on elements where there are no such labels.

The user can designate (they are in fact already public, even if not yet delivered) her changes by applying her own user-specific label, say `ANN_BUGFIX2.1`, to the full set of versions she wants to bind together. This label can be used for verification or integration purposes either by herself or by others. Note that this does not have to affect the user's config spec.

Of course, she would want to switch to a different config spec if her situation would change, e.g. to subscribe to a different baseline. In this case it would be enough to change the fourth line in the above config spec to `element * NEWLABEL`, where `NEWLABEL` refers to a different set of versions.

Also, while debugging a difficult issue, she might want extra stability, and thus to be protected even from normal updates. She might do this by switching to a special config spec, which could be the following:

```
element * CHECKEDOUT
element * .../branch/LATEST
mkbranch branch
element * MYLABEL
element * /main/0
```

Here `MYLABEL` is the user's own baseline label, which she is sure will not be modified by anyone except herself.

A more complex config spec for a similar purpose (e.g. debugging one's own changes, perhaps with the intention of discarding the temporary changes put in place for the time of debugging) could look like this:

```
element * CHECKEDOUT
element * .../tempbranch/LATEST
mkbranch tempbranch
element * ANN_BUGFIX2.1
element * MYLABEL
element * /main/0
```

Here on top of foundation baseline `MYLABEL`, the own published changes are selected (labeled with `ANN_BUGFIX2.1` label) and branched-off to a different own branch `tempbranch`.

Floating and fixed labels

The issue of stability and volatility is thus essential. Managing changes is managing their propagation, and keeping stable the environment used to manage it.

A simple convention concerning labels conveniently addresses this need: label application may use the `-replace` flag, in which case the labels may be moved from existing versions to new ones. Labels using this feature should clearly be distinguished from others: they are customarily called **floating**, whereas the others, the stable ones, are referred to as **fixed**. More generally, fixed labels should thus offer a guarantee of stability. Floating labels too offer one essential element of stability: their name can be seen as a handle to up-to-date, and therefore evolving, reality.

Floating labels make it possible for config specs to be stable. Consider the following config spec:

```
element * CHECKEDOUT
element * .../mybranch/LATEST
mkbranch mybranch
element * TOOLS
```

Here, `TOOLS` is a floating label. It is actually applied so that it points to the current version of the particular tool(s) in use. For example, one may have several versions of gcc compiler stored in ClearCase: version 4.4.1 labeled with `GCC_02` label and version 4.1.2, labeled as `GCC_01` (both of these are fixed labels). When the gcc version 4.1.2 was selected for the development purposes, the label `TOOLS` was applied exactly to the same element versions as those carrying `GCC_01` labels. When the newer, 4.4.1 gcc compiler version needs to be taken into use, the floating label `TOOLS` will be moved so that it applies exactly to the same element versions as those labeled with `GCC_02`. The developers' config spec will not change.

It is often a good idea to conventionally bind a floating label to a family of fixed ones. This is most naturally achieved by sharing a common prefix. In the example above, we might have a `GCC` floating label, designating the current recommended version of the Gnu compiler as well as a `TOOLS_1.27` fixed label, contributing to keep track of the position of `TOOLS` at a given point in time, across a consistent set of tools.

Note also that we choose to keep part of the name (the running number, `_01` and `_02` in the example) of the label type free from any predefined meaning (such as `_4.1.2`). The idea is that it is difficult to foresee the future, and that later one might have to import a new, different release of the same version of gcc (for a different platform, or itself using another tool, or whatever). It is much better to put detailed information in the comments of the type than in its name: updating this information if the need arises will be easier than changing the name.

It is important to note that the floating label based config specs we have been promoting here cannot be used to reproduce a precise event (for example a certain build or test run): they are not meant for this purpose. As we saw in *Chapter 3*, *Build Auditing and Avoidance*, an event should be recorded in a derived object hierarchy. The top of this hierarchy may be used to apply labels, and we are naturally speaking here of fixed labels (at least at first: in any case, we intend to easily reconstruct the full baselines which the config records constituted). These labels will be replicated to the other sites, and thus make the event reproducible across MultiSite boundaries.

Here are some hints on how to apply fixed labels.

Suppose, we would like to publish all the element versions that are needed in order to produce a certain result, specified as a build target (this may be a fix, enhancement, whole component or system, etc).

To apply a fixed label:

- Check that there are no issues with the config record (see again Chapter 3):

```
$ ct catcr -check -union build.tag
```

- Determine the vobs involved (`build.tag` is the top derived object produced by the build, refer to Chapter 3):

```
$ VOBS=$(ct catcr -flat -type d -s build.tag | grep \\.@@ | \
 perl -pe 's:/\.@@.*$::' | sort -u)
```

- Create fixed label type in every vob:

```
$ FIXED="TOOLS1.1"
$ for v in $VOBS; \
 do ct mklbtype -vob $v -c "$COMMENT" $FIXED; done
```

  Note that one may prefer to use global types, even without admin vobs—create a global type in the vob where the derived object resides, use `cptype` to create local copies in the other vobs, and explicitly create the `GlobalDefinition` hyperlinks

- Apply the label using the `build.tag` config record:

```
$ ct mklabel -con build.tag $FIXED
```

- Then in order to "select" a certain version of a tool, which bears a fixed label, we would apply a floating label on top of this fixed label as follows:

```
$ FIXED="TOOLS1.1"
$ FLOATING="TOOLS"
$ cleartool find . \
-version "lbtype($FIXED) && ! lbtype($FLOATING)" -print | \
xargs cleartool mklabel -rep $FLOATING
```

- We also need to remove a floating label from versions that do not bear the fixed label (this would be needed when changing the selected tool version):

```
$ cleartool find . \
 -element "lbtype_sub($FLOATING) && ! lbtype_sub($FIXED)" \
 -print | xargs cleartool rmlabel $FLOATING
```

Baselines and incremental labels

A fixed label can designate a **baseline** when it is **full**, that is, applied to all the elements included in a particular software configuration. The label can also be **incremental** (a.k.a. partial or sparse), i.e. applied only to the elements that have changed comparing to a certain baseline. Note that *baseline* is a keyword in UCM, and we are not speaking of it here. A baseline is a software configuration used as a reference.

Let's state for clarity that, for us, a baseline is full by definition. We take the association: "incremental baseline" as an oxymoron. A baseline may be represented by a full label, or by an aggregation of incremental ones. It is anyway a full baseline (and this is a pleonasm).

For the purpose of selecting a baseline, a floating label may be applied on top of either a full fixed label, or an aggregation of incremental ones.

For example, one may have a baseline denoted by the fixed label `REL_1.00`, which has been applied to all the versions included into the software configuration of the release 1 of the software product. Then, a floating label `REL` would be applied to all the versions already having the label `REL_1.00`, designating the currently selected release.

In case of a bug fix, the changed code base may be labeled as `REL_1.01`, including only the versions changed for this particular fix (incremental label).

Proceeding further with bug fixing activities, another incremental label, `REL_1.02`, can be created, and so on.

In order to aggregate the incremental labels, one could use either of two options. The first is to set the following config spec:

```
element * REL_1.02
element * REL_1.01
element * REL_1.00
```

The other option is to move the label `REL`, first to all the versions labeled as `REL_1.01`, and then further, to all the versions labeled as `REL_1.02`.

Then, the integration view config spec would contain a single line:

```
element * REL
```

The alternative to using incremental labels would be to make a full fixed label recursively applied to the new code base. This would create a new baseline, `REL_1.03`, including all the fixes on top of the original release 1.

Recursively applying labels is however an expensive, time-consuming, operation. Moving a floating over a subset of changed versions out of a large configuration is often far lighter: fewer database write operations are involved. Such a baseline, resulting from an original application of floating labels over a full configuration, followed with moving some of the labels over successive change sets, will be equivalent to one obtained by applying fixed labels over the last full configuration (neglecting the case of elements removed from the configuration at some stage): as the baseline defined in our example by the `REL` floating label, which would be equivalent to the baseline defined by the `REL_1.03` fixed label.

This notion of baseline equivalence may be exploited to achieve a synthesis of the two valuable properties: the relative cheapness of application of the floating labels, and the possibility to use fixed labels as a record. One needs to consider a list of sparse (incremental) labels, applied at every stage to the change sets. It is easy to build a config spec with rules using the incremental labels as fall-back of each other (as the config spec containing `REL_1.02`, `REL_1.01` and `REL_1.00` rules in our example above). Such a config spec would be equivalent to the one line config spec based on the floating label alone (`REL`). Starting from the type which is on the top of the list at any given date in the past, one might reproduce any of the successive baselines the floating labels have embodied over time.

Support for this strategy is implemented in our CPAN module, ***ClearCase::Wrapper::MGi***, where the issue of removing elements from the baseline is addressed.

One might object that this is a typical example of config spec generation, and we must admit it is, but this use is exceptional, and only meant to allow as a rule to use a stable and simple config spec with the guarantee that reproducibility is not jeopardized by the gain in efficiency and convenience.

One limitation of this scheme is that this generation of equivalent config specs works only for one single floating type (which may be global, and thus span across multiple vobs): there is no simple way to interleave the rule stacks that would result from emulating several types. We do not see this limitation as constraining in practice: it only forces to consolidate one's floating labels into a single one, representing the whole baseline.

Branches and branch types

It is sometimes difficult for users to understand that branches may actually be themselves objects, distinct from the branch types. A way to convince one of this fact is to use the `chtype` and `rename` operations, and to notice how they apply respectively to branch instances and to branch types.

Using `chtype` (or `rename`) may be an option to modify the way in which a given config spec will select a version of an element: this will happen if either the initial or the new type (name) is matched in a rule:

```
$ ct ls foo bar zoo
foo@@ [no version selected]
bar@@/main/m/3                  Rule: .../m/LATEST
zoo@@ [no version selected]

$ ct catcs
element * CHECKEDOUT
element * .../m/LATEST

$ ct lsvtree foo
foo@@/main/m1
foo@@/main/m1/0
foo@@/main/m1/1

$ ct lsvtree zoo
zoo@@/main/m1
zoo@@/main/m1/0
zoo@@/main/m1/1
zoo@@/main/m1/2

$ ct chtype m foo@@/m1
Changed type of branch "foo@@/m1" to "m".

$ ct ls foo bar zoo
foo@@/main/m/1                  Rule: .../m/LATEST
bar@@/main/m/3                  Rule: .../m/LATEST
zoo@@ [no version selected]
```

As one can see in this preceding example, it is only the *branch* `m1` of the `foo` element, which has been renamed to `m`, not the `m1` branch type itself; and the other element's (`zoo`) branch `m1` remained unaffected.

On the contrary, when we use the `rename` command, the change affects all the branches:

```
$ ct ls foo bar zoo
foo@@ [no version selected]
bar@@/main/m/3                     Rule: .../m/LATEST
zoo@@ [no version selected]
$ ct rename brtype:m1 m
cleartool: Error: Name "m" already exists.
cleartool: Error: Unable to rename branch type from "m1" to "m".
```

```
$ ct chtype m1 bar@@/m
Changed type of branch "bar@@/m" to "m1".

$ ct rmtype -f -rmall brtype:m
Removed branch type "m".

$ ct rename brtype:m1 m
Renamed branch type "m1" to "m".

$ ct ls foo bar zoo
foo@@/main/m/1                    Rule: .../m/LATEST
bar@@/main/m/3                    Rule: .../m/LATEST
zoo@@/main/m/2                    Rule: .../m/LATEST
```

Note that we needed to remove the existing `m` type first (to make the rename succeed), which removed all the existing branches of this type. This is why we preserved the branch `m` of the bar element by *chtyping* it to `m1`.

We can now see that all the elements having branches of type `m1` were affected.

Let's note a couple of peculiarities pertaining to branches:

- Locking branch types affects the branches, so that the versions cannot be labeled anymore. In our opinion, this pretty much defeats the purpose of locking branch types at all
- The `multitool chmaster` command, when applied to a branch type, will also affect existing branches of this type. This is convenient but has one well-founded yet non-intuitive restriction: only if their mastership has not been changed explicitly (this applies also to `main` branches of elements created with the `-master` option)
- One can remove an element's branch being any one of the branch creator, the element owner or the vob owner, or root. The branch creator can remove only his own branch, provided it has no subbranches created by other users, and if no version on this branch carries a locked label. The element owner can remove any branch, unless some version on the branch is protected by a locked label:

```
$ ct des -fmt "%[owner]p\n" foo
sam

$ ct lsvtree foo
foo@@/main
foo@@/main/0
foo@@/main/aa
foo@@/main/aa/0
foo@@/main/aa/bb
foo@@/main/aa/bb/0
```

```
$ ct des foo@@/main/aa
branch "foo@@/main/aa"
  created 2010-08-01T08:59:59+02:00 by Name=mary

$ ct des foo@@/main/aa/bb
branch "foo@@/main/aa/bb"
  created 2010-08-01T09:03:38+02:00 by Name=joe

$ sudo -u joe cleartool mklabel J foo@@/main/aa/bb/0
Created label "J" on "foo" version "/main/aa/bb/0".
$ sudo -u joe cleartool lock lbtype:J
Locked label type "J".

$ sudo -u joe cleartool rmbranch -f foo@@/main/aa/bb
cleartool: Error: Lock on label type "J" prevents operation ######
                                                    "remove version".
cleartool: Error: Only VOB owner or privileged user may remove ###
              a version labeled with an instance of a locked type.
cleartool: Error: Unable to remove branch "foo@@/main/aa/bb".

$  sudo -u sam cleartool rmbranch -f foo@@/main/aa/bb
cleartool: Error: Lock on label type "J" prevents operation ######
                                                    "remove version".
cleartool: Error: Only VOB owner or privileged user may remove ###
              a version labeled with an instance of a locked type.

$ sudo -u joe cleartool unlock lbtype:J
Unlocked label type "J".

$ sudo -u sam cleartool rmbranch -f foo@@/main/aa/bb
Removed branch "foo@@/main/aa/bb".
```

Delivery

We believe to follow the common use in distinguishing *delivery* from *release*, by considering that the latter involves an additional *packaging* step, implying further extraction and installation. A similar procedure may be necessary if some level of testing in mandated to take place in an environment where the SCM system is not available.

The developer works in her branches, checks in and out her code as she needs to produce save-points, builds and tests her results. At some point, she has something to offer to others. Her delivery will result in updating a public baseline. It is essential that:

- What the user publishes matches exactly what she just tested
- The procedure is trivial and cheap, which guarantees she will use it often and with small increments
- The procedure may apply in cascade to various degrees of integration
- Different developers do not block each other and do not waste time in synchronization
- Concurrent publications do not obliterate each other (losing with one contribution what the previous brought in)
- Derived objects built prior to the delivery are not obsoleted by it. On the contrary, they are offered for sharing, thus avoiding the race condition that otherwise takes place for the creation of the first DOs matching the new delivery. Such derived objects also remain valid for config record comparison, in case a problem is found and one needs to assess why it wasn't detected by previous testing
- The procedure is reversible, so that in case of error, the delivery can be withdrawn (with no loss of information) and analyzed by the developer, in preparation for a new fixed delivery.

These requirements plead for an in-place delivery, with no modification of data: only a change of status. This is typically what labeling offers.

Note how the model of delivering by merging to integration branches fails to achieve these requirements. Merges produce new versions, which invalidates the existing derived objects. The process may attempt to compensate for this loss by requiring a build to take place as part of the delivery, but this hugely raises its cost (for example in time) and introduces new intermediate points of failure. Reversibility at this point may only be achieved if the build uses checked out versions, by unchecking them out. But this loses the information needed to analyse the cause of the failure. In any case, the identity of the delivered changes cannot be guaranteed by the system since distinct versions were produced. Once the initial strategic error is made, there are only bad solutions: fixing any one aspect unescapably creates a new problem.
One problem we must acknowledge is that delivering by labeling doesn't solve the problem of offering derived objects as part of the baseline update over MultiSite. This is an issue for which we do not have an elegant and simple solution. The best we can offer it the recommendation to stick to local labels in usual config specs. Thus follow a remote delivery by first producing a build using it, and then updating the local baseline to use this build.

Ironically, in spite of the similarities between the label and branch concepts mentioned earlier, people tend for historical reasons to think of integration as merging everything to a single branch. Using this branch type in the config spec is even considered safe and clear. Label-based config specs on the contrary are seldom used for integration purposes and often neglected. The obvious duality between labels and branches is often lost: users are surprised when reminded that labels are sufficient to select versions, independently from their position in the version tree. Select a baseline with integration labels, and you don't need integration branches: it is irrelevant where the labels are found in the version trees.

One of the above requirements, though, requires special attention: ensuring continuity with other developers' contributions. This implies that the delivery labels are not moved blindly from the versions to which they are currently applied, but only from direct ancestors of the new versions. In practice, this requires what UCM terms a *rebase* operation, and we shall name, in order to avoid confusion with the `rebase` *command* (which we cannot use outside of UCM), a **home merge**. We shall deal with this in our next chapter.

Archiving

As mentioned earlier, the config spec of the developer authoring the delivery shouldn't have to be modified. Again, by delivery here we mean **publishing in-place** *by labeling*. For example, the user has labeled her deliveries with fixed `ANN_BUGFIX_1` and `ANN_BUGFIX_2` labels, and also applied a floating "integration" label named `ANN_BUGFIX`.

Until the delivery, the new versions were selected either with branch-based rules in the developer's config spec (`mybranch` in the next example), or with previous delivery labels (`REL`). There might also be pre-delivery labels. This would be the case if team work was integrated in steps. For example, the developer could be using the following config spec (building on one already described in the beginning of this chapter):

```
element * CHECKEDOUT
element * .../mybranch/LATEST
mkbranch mybranch
element * ANN_BUGFIX
element * REL
element * /main/0
```

The use of a personal label is possible, on top of the team pre-delivery one, and even useful to keep track of correspondences between different elements, and to communicate between different views the user may have.

It is at any rate essential that the developer starts using the delivery herself, and doesn't stay caught selecting her own development versions. With the in-place delivery, there is no difference at first sight, but there might soon arise some, as other developers start delivering further improvements.

Our solution is to *archive*, as a part of the delivery, one's branches and floating labels, referenced in the user's config spec, and to do this via renaming.

The branch type `mybranch` can be archived when `ANN_BUGFIX` label application is complete and needs to be tested properly:

```
$ ct rename brtype:mybranch mybranch-001
$ ct mkbrtype -nc mybranch
```

The label type `ANN_BUGFIX` can be archived when user's published changes have been accepted and the floating label REL has been moved over it:

```
$ ct rename lbtype:ANN_BUGFIX ANN_BUGFIX-001
$ ct lock -obs lbtype:ANN_BUGFIX-001
$ ct mklbtype -nc ANN_BUGFIX
```

This way the user's config spec need not change after the delivery: the `mybranch` and `ANN_BUGFIX` (and possible pre-delivery) types are again free for development purposes. Contrast this to keeping the names unchanged and modifying the config spec instead. In this latter and common case:

- All the config specs that use the types need to be updated synchronously
- There is no support for backup and history of these changes

This is something that may be scripted (and again, is already supported by our CPAN module ***ClearCase::Wrapper::MGi***).

The binding between the branch type and the label type may be interpreted as the creation of a higher-level concept, not far from *activity* or *stream*: the stream completes at the delivery.

Rollback

The possibility to rollback one's changes is offered by the fact that the delivery was exclusively a labeling (considering the eventual home merge as part of pre-delivery development, and indeed, this one might legitimately have been followed with build and test recording). As such, it is trivially reversible.

Of course, the situation gets slightly more complex as soon as further deliveries have happened on top of the one, which its author wishes to roll back. We'll see in the next chapter (because reconstructing consistent versions will indeed involve some merging) that removing an intermediate step gets facilitated by the fact that it maps to a distinct set of branches.

Use of locking

Locking is often thought as a final operation, meant to conclude changes, making further modifications impossible.

We see this as highly paradoxical: locking is a management operation and management is needed to allow change. Avoiding changes is an act of control, and thus, quite the opposite.

So, in fact, the more one needs to change things, the more one wants to lock the label types. If there was no need for any change, no locking would be needed.

After all the labels have been applied as needed, it is advisable to lock both fixed and floating label types.

Label types should be locked in order to prevent accidental label removal, moving, and application to wrong elements. It is important here to avoid confusing safety with security: these tools are proper to fight error, not fraud. Misusing them to counter mythical attackers easily compromises safety by artificially growing complexity, making fixes cumbersome, and alienating developers of their responsibilities.

When the floating label needs to be moved, its type must be unlocked first, the labels applied, and the type locked again.

Locking to manage change is a way to be faithful to the *static typing* philosophy stated at the beginning of this chapter: unlocking a floating label before moving it parallels creating the type before applying it the first time, and therefore restores the first step required to allow for consistency checking.

In addition, locking provides an event recorded in the history of the label type, and thus with a time stamp and information about the user and the context. This event marks the end of the label application.

To lock a label type, one can use the plain `cleartool lock` command:

```
$ ct lock lbtype:L1
Locked label type "L1".

$ ct lstype -kind lbtype | grep L1
--07-06T21:15 ann label type "L1" (locked)

$ ct des -fmt "%[locked]p\n" lbtype:L1
locked
```

One can also use the `cleartool lock -obsolete` command, to lock it and mark as *obsolete*:

```
$ ct lock -obs lbtype:L2
Locked label type "L2".
```

Obsolete label types are not listed by default by the `cleartool lstype` command, unless `-obsolete` flag is specified, which makes `lstype` queries slightly more efficient and less verbose:

```
$ ct lstype -kind lbtype | grep L2
$ ct lstype -obs -kind lbtype | grep L2
--07-07T22:34 ann label type "L2" (obsolete)

$ ct des -fmt "%[locked]p\n" lbtype:L2
obsolete
```

To obsolete the already locked label, one can specify the `-rep` and `-obs` flags to the lock command:

```
$ ct lock -rep -obs lbtype:L1
Locked label type "L1".

$ ct des -fmt "%[locked]p\n" lbtype:L1
obsolete
```

To unlock the label type (also obsolete one), use the `unlock` command:

```
$ ct unlock lbtype:L1
Unlocked label type "L1".

$ ct des -fmt "%[locked]p\n" lbtype:L1
unlocked
```

Locking an object is possible only for its owner (or to an administrator). This may easily become a nuisance, and may thus requires scripting (the use of `sudo`, `suid` scripts, `ssh`, or `runas` on Windows), for example, to authorize group members to lock/unlock each others' types.

Obsoleting types, under ClearCase, has yet another distinct and loosely related use: obsolete locks are the only ones replicated, as the effects of locking are otherwise redundant with these of mastership.

Let's note that one cannot change the mastership of locked objects, which is often a good thing as such: it is better to leave these objects locked (and thus mastered in their original replica), than to unlock them, `chmaster` them, to lock them back in the new replica (losing part of the original history for no clear benefit). We may further note that may be surprisingly the mastership of objects locked obsolete may be changed.

Types as handles for information

A last note on types is that they often constitute convenient objects to which to attach information to be shared between files.

This is trivially true of comments, but also of attributes and hyperlinks.

We'll give examples of such uses in the *Chapter 9*, *Secondary Metadata*.

Summary—wrapping up of recommended conventions

We announced that our review of constraints would give rise to recommendations of conventions, and it did: we made a good deal of suggestions that go against some traditional *best practices*. Let's mention that we do *not* believe these recommendations would only be suitable for specific teams, environments, or levels of expertise. On the contrary, they are designed to be scalable, generic, and consistent, which doesn't mean they would constitute a silver bullet or cannot be further enhanced. Here is a small summary, and it covers the essential aspects we examined in this chapter:

- Checkout exclusively in (private) branches, of one single and stable type
- Use branch rules in your config spec of only this type
- Apply labels in pairs: a full floating label and an incremental fixed one
- Use only the floating label in your development config specs
- Deliver exclusively by applying labels thus in place, and archive your development branches away
- This way, keep your config spec stable
- Prior to delivering, *rebase* where the baseline has moved after you branched off it
- Lock your labels and unlock the floating ones before moving them

7
Merging

As much as one may attempt to avoid conflicts, it is also necessary to solve them when they happen. This concern is the object of the present chapter.

Merging has become a pet peeve for many, to the point of being overused, and for purposes other than conflict resolution.

We'll review both practical aspects, and will guide about things to do and things to avoid.

Faithful to our focus, we shall ignore any UCM aspects and the graphical tools, with their additional and accidental complexities.

Somehow, this chapter might naturally feel of lesser importance than the preceding ones. This is of course by choice: we focused first on the essential, pushing forward the remaining explanations. Sharing our vision of the essential is part of our goal: to convince our reader to look at ClearCase *differently*! Our views are meant to be non-obvious at the start: they directly oppose the main stream thinking for which merging plays a central role. We believe, as we already tried to show, that this is unfortunate. In fact, a major part of our experience with merging comes from assisting users with concrete and complex problems that are seldom traced back to their real causes: processes and practices. Refusal to examine the cause of recurrent problems leads to treating the symptoms instead, especially with an extensive use of GUIs (naively seen as rescue buoys), or with scripts to force *copy merges* (see later); both of these have vicious circle effects and only make things worse.

We'll work out here the causes of some common misunderstandings. We believe it is fair to acknowledge a widely spread confusion: users don't feel confident with merging, and often resort to delegating it, hoping for (or requesting) miracles.

Although we strongly advise to avoid useless merges, merging is necessary to resolve conflicts and must thus be mastered.

It remains to say that the chapter is crossed by multiple themes that make the presentation of the material hard to order: they overlap, and will thus pop up several times at different points in the exposition. Let's mention the main "themes" and let you identify them in the sections:

- The tools (the practical aspect): `merge` and `findmerge`, but also `ln`, and anecdotal as it is, `rmmerge`
- The use and misuse of merging: conflict resolution versus mere version duplication
- The cumbersome undoing or rolling back
- The *Merge arrow*, or the tracing of contributions
- The nitty-gritty, that is, the type manager part, and the syntactic issues of data chunking

Patching and merging

One remembers the historical importance of *patch* in the birth of version control, and thus of *source code* management—the prehistory of SCM. This aspect of computing *deltas* (or *diffs*) and applying them on top of an existing text file is still at the heart of merging.

The differences, however marginal, lie on the one hand in using the identity of the element to determine a *base contributor* from the common version tree, and on the other hand in extending merging to directories and semi-binary files (lifting the strictest encoding limitations). The former is not a strict requirement (except for directories, maintained in the database and not as external *containers*), but covering the vast majority of the cases, builds upon the maintenance of *Merge arrows* to structure the version trees. In addition, bulk merging, using `findmerge`, only applies to versions of the same elements.

Patching text files

Reminiscent of the text file orientation of traditional languages, merging is offered by (several) *element type managers*, centered around the one for text files. While closer attention to element types will be postponed to *Chapter 9*, *Secondary Metadata*, we have to mention this delegation of functions to specific tools.

Let's first note that it is fully possible with files to use tools external to ClearCase to perform this function. In fact, this is exactly what we do ourselves, at least for verification, using the `ediff-buffers` function under *GNU emacs*, in any case of real merge involving more than three lines or more than one location in the file.

One has then to take care of the *Merge arrows*: see lower.

Looking more closely at the way the type managers are configured, that is, at the contents of the `lib/mgrs` directory under the installation root on every ClearCase client, one notices that on UNIX, the tools performing the various functions (such as `merge` for textual, and `xmerge` for graphical merges) are specified via symbolic links, whereas on Windows they are specified in a `map` file and linked there, either via paths or references into the Windows registry or the merge tools.

Either way, this allows sharing of some tools between various *managers* in a way which is in fact orthogonal to their inheritance hierarchy, and thus allows overriding it. What we can see is that the *text_file_delta* tools serve as basis for several of the other element types, and that in the case of the two functions we are concerned with in the current context, these resolve to: `cleardiff` and `xcleardiff` in UNIX and `cleardiff.exe` and `cleardiffmgr.exe` in Windows.

```
$ ll /opt/rational/clearcase/linux_x86/lib/mgrs/binary_delta
lrwxrwxrwx 1 root root 22 Nov 13 2008 compare -> ../../../bin/cleardiff
lrwxrwxrwx 1 root root 22 Nov 13 2008 merge -> ../../../bin/cleardiff
lrwxrwxrwx 1 root root 23 Nov 13 2008 xcompare -> ../../../bin/xcleardiff
lrwxrwxrwx 1 root root 23 Nov 13 2008 xmerge -> ../../../bin/xcleardiff

$ ll /opt/rational/clearcase/linux_x86/lib/mgrs/text_file_delta
lrwxrwxrwx 1 root root 22 Nov 13 2008 compare -> ../../../bin/cleardiff
lrwxrwxrwx 1 root root 22 Nov 13 2008 merge -> ../../../bin/cleardiff
lrwxrwxrwx 1 root root 23 Nov 13 2008 xcompare -> ../../../bin/xcleardiff
lrwxrwxrwx 1 root root 23 Nov 13 2008 xmerge -> ../../../bin/xcleardiff

$ cat /cygdrive/c/Program\ Files/Rational/ClearCase/lib/mgrs/map
text_file_delta  compare    ..\..\bin\cleardiff.exe
text_file_delta  xcompare  ..\..\bin\cleardiffmrg.exe
text_file_delta  merge        ..\..\bin\cleardiff.exe
text_file_delta  xmerge      ..\..\bin\cleardiffmrg.exe

binary_delta  compare     ..\..\bin\cleardiff.exe
binary_delta  xcompare   ..\..\bin\cleardiffmrg.exe
binary_delta  merge         ..\..\bin\cleardiff.exe
binary_delta  xmerge       ..\..\bin\cleardiffmrg.exe
```

This gives us a hint as to how we might define an element type to use a custom merge tool. We'll leave this to those among our readers whom we couldn't convince that this is only a sidetrack.

Once we are already this far down (or up: matter of taste) technicalities, let's conjecture about the meaning of *binary_delta*, and of merging, in this context. Merging and *diffing* implies *chunking* the data into comparable items, which seems simple in the case of lines of text, the only problem there being deciding the exact syntax of separators: end-of-line codes. However, HTML and XML standards made it obvious that this stood only on convention: in fact, few of the traditional software languages made it mandatory for software texts to break on *lines*! Statements and definitions broke more often on semicolons, forms on parentheses... Yet, until the advent of the world-wide-web, systematically removing any whitespace was practiced only for code obfuscation purposes.

Not so anymore when the data had to be downloaded under the constraints of limited bandwidth. Some other ways to identify comparable patterns, of size compatible with this of reasonable buffers, had to be devised. A further step has since been made with the different variants of Unicode, bringing back issues of ordering of multibytes characters. This has now made its way into specialized element types in version 7.1 of ClearCase.

Managing contributions

Back now to the issue of managing the contributions to a given version, that is, to identifying the contributing versions. During the merge, diffs (or deltas) have to be computed between every contributor and a common reference: the **base contributor**—typically the closest common ancestor, possibly one of the contributors.

The `merge` tool will use the version tree to compute a suitable version for the role of *base*, aiming at producing as small diffs as possible. For that purpose, beyond direct ancestry, special kinds of hyperlinks (between versions) will be used: instances of the *Merge* type. They will be created at the end of every merge, for use at the start of any later one. They may also be created and removed explicitly.

Let's work out an example, as simple as possible. We are trying here to describe a mechanism, not to recommend a process.

We'll use a `foo` text file element, make changes in an `mg` branch (selected by the config spec: every access of the main branch thus has to be explicit), and merge them to the `main` one.

The initial data is a single line.

```
$ echo 111 > foo
$ ct mkelem -nc -ci foo
Created element "foo" (type "text_file").
Created branch "mg" from "foo" version "/main/0".
Checked in "foo" version "/main/mg/1".
$ ct co -nc -bra /main foo
cleartool: Warning: Version checked out ("/main/0") is different from ###
                    version selected by view before checkout ("/main/mg/1").
Checked out "foo" from version "/main/0".
$ ct merge -to foo -ver /main/mg/1
Trivial merge: "foo" is same as base "/vob/test/foo/merge/foo@@/main/0".
Copying "/vob/test/foo/merge/foo@@/main/mg/1" to output file.
Moved contributor "foo" to "foo.contrib".
Output of merge is in "foo".
Recorded merge of "foo".
$ ct ci -nc foo
cleartool: Warning: Version checked in is not selected by view.
```

```
Checked in "foo" version "/main/1".
$ ct lsvtree -merge foo
foo@@/main
foo@@/main/0
foo@@/main/mg
foo@@/main/mg/1
 -> /main/1
foo@@/main/1
```

As we use the `-merge` option, `lsvtree` shows the *Merge* arrows.

We got here a first case of **trivial merge**: a merge is said trivial if the base contributor is the same as the merge target (the *to* contributor)—here, it is the version which was checked out, since no change was made to it yet. Trivial merges are automatic: the user is not prompted for a decision (unless he explicitly requires it with the `-qall` flag).

Let's now assume that for some reason, we want to roll back this change, that is, to *undo* the merge. The most natural option is to use a **subtractive merge** (not quite a merge at all in fact, but a related use of the same tool, `ct merge`, with `-del` option).

```
$ ct co -nc -bra /main foo
cleartool: Warning: Version checked out ("/main/1") is different from ###
                  version selected by view before checkout ("/main/mg/1").
Checked out "foo" from version "/main/1".
$ ct merge -to foo -del -ver /main/1
Trivial merge: "foo" is same as base "/vob/test/foo/merge/foo@@/main/1".
Copying "/vob/test/foo/merge/foo@@/main/0" to output file.
Moved contributor "foo" to "foo.contrib.1".
Output of merge is in "foo".
$ ct ci -nc foo
cleartool: Warning: Version checked in is not selected by view.
Checked in "foo" version "/main/2".
$ ct lsvtree -merge foo
foo@@/main
foo@@/main/0
foo@@/main/mg
foo@@/main/mg/1
 -> /main/1
foo@@/main/2
```

As we can see, this didn't result in a user noticeable change in the topology of the version tree, except for the creation of a new version: `/main/2`. Note that there are no Merge arrows for subtractive merges.

The base contributor was again selected as the closest common ancestor, which was also the contributor being deleted, as well as the *to contributor*; hence once again a *trivial* merge.

We also note the saving of view-private copies of the contributors, which we consider a minor nuisance.

Next, we'll make a further change to the version on the branch (maybe fixing the problem which resulted in the previous rollback):

```
$ ct ls foo
foo@@/main/mg/1    Rule: .../mg/LATEST
$ ct co -nc foo
Checked out "foo" from version "/main/mg/1".
$ echo 222 >> foo
$ ct ci -nc foo
Checked in "foo" version "/main/mg/2".
```

And we'll attempt to roll out (well, to merge) again. This should be understood as a common practice (note that we do not say a good one), yet its result is not *intuitive* (users typically notice at this stage that they weren't extremely clear about their own expectations):

```
$ ct co -nc -bra /main foo
cleartool: Warning: Version checked out ("/main/2") is different from ###
                  version selected by view before checkout ("/main/mg/2").
Checked out "foo" from version "/main/2".
$ ct merge -to foo -ver /main/mg/2
*******************************
<<< file 1: /vob/test/foo/merge/foo@@/main/mg/1
>>> file 2: /vob/test/foo/merge/foo@@/main/mg/2
>>> file 3: foo
*******************************
-----[deleted 1 file 1]------|------[after 0 file 3]-------
111                          |-
                            -|
*** Automatic: Applying DELETION from file 3 [deleting base line 1]
============
============
----[after 1 file 1]---------|-------[inserted 2 file 2]----
                            -| 222
                             |-
Do you want the INSERTION made in file 2? [yes]
Applying INSERT from file 2 [line 2]
============
============
Moved contributor "foo" to "foo.contrib.2".
Output of merge is in "foo".
Recorded merge of "foo".
$ cat foo
222
```

For a start, this wasn't a trivial merge anymore: ClearCase had to prompt the user. We must understand that this kind of dialog is suboptimal:

- It stops the transaction (maybe multiple times)
- It is error-prone, and thus stressful to the user: no more predictable for her than for the tool
- The transcript will often get lost, which means that the merge result can only be inspected by others on the basis of less and different data than was presented to the user

This means that a procedure based on this kind of merging cannot be atomic and may introduce states that affect others.

Then, we were prompted for the *wrong* thing; the question concerned the last addition (line `222`, the *fix*), instead of the previous deletion. Why does this feel wrong? Because it differs from the *normal* behavior: what happened the first time, before the rollback. The common user does not expect this. What is different for the user is the deletion. The above sequence is, however, the correct behavior from the point of view of ClearCase, and we'll have to understand why. The result is anyway that the deletion was confirmed, against the intention; the line added to version `/main/mg/1` (`111`) might have had an undesirable side effect, which motivated the rollback, but it is still part of the fixed version `/main/mg/2` and the user expects it to be found in the version resulting from the merge.

Let's clean up, that is, abort this attempt by unchecking out its results and try again, taking this time the responsibility of naming ourselves the base contributor, picking first the new version in the branch, which results in skipping the history of the previous deletion.

```
$ ct unco -rm foo
Checkout cancelled for "foo".
$ ct co -nc -bra /main foo
cleartool: Warning: Version checked out ("/main/2") is different from ###
                 version selected by view before checkout ("/main/mg/2").
Checked out "foo" from version "/main/2".
$ ct merge -to foo -base foo@@/main/mg/2 -ver /main/mg/2
Trivial merge: "/vob/test/foo/merge/foo@@/main/mg/2" is same as base ####
                                                        "foo@@/main/mg/2".
Copying "foo" to output file.
Moved contributor "foo" to "foo.contrib.3".
Output of merge is in "foo".
$ cat foo
$
```

We are satisfied of reaching our goal of getting a trivial (therefore automatic) merge, but should be slightly surprised as this behavior doesn't match well with the ***documentation*** as we had read it; the *base* and *to* contributors are now different!

However, no visible change took place (the resulting file is still not correct, as it is empty now)! Let's pick the *to contributor* as *base*, thus reconciling the documentation of trivial merges:

```
$ ct merge -to foo -base foo@@/main/2 -ver /main/mg/2
Trivial merge: "foo" is same as base "foo@@/main/2".
Copying "/vob/test/foo/merge/foo@@/main/mg/2" to output file.
Moved contributor "foo" to "foo.contrib.4".
Output of merge is in "foo".
$ cat foo
111
222
```

The data is now the expected one. Let's thus check in and go forward.
Note, however, that in the last two merge examples, when we specified explicitly the base contributor ourselves, we did not get the `Recorded merge of "foo"` message, and hence, no Merge arrow was created automatically:

```
$ ct lsvtree -merge foo
foo@@/main
foo@@/main/0
foo@@/main/mg
foo@@/main/mg/1
 -> /main/1
foo@@/main/mg/2
foo@@/main/2
foo@@/main/CHECKEDOUT view "marc"
```

This time, we'll make a change to the checked out version, prior to the merge (in parallel to a change to the version to be merged):

```
$ ct ci -nc foo
cleartool: Warning: Version checked in is not selected by view.
Checked in "foo" version "/main/3".
$ ct co -nc foo
Checked out "foo" from version "/main/mg/2".
$ echo 333 >> foo
$ ct ci -nc foo
Checked in "foo" version "/main/mg/3".
$ ct co -nc -bra /main foo
cleartool: Warning: Version checked out ("/main/3") is different from ###
                    version selected by view before checkout ("/main/mg/3").
Checked out "foo" from version "/main/3".
$ echo 444 >> foo
$ cat foo
```

```
111
222
444
$ cat foo@@/main/mg/3
111
222
333
$ ct merge -to foo -ver /main/mg/3
*******************************
<<< file 1: /vob/test/foo/merge/foo@@/main/mg/1
>>> file 2: /vob/test/foo/merge/foo@@/main/mg/3
>>> file 3: foo
*******************************
-------[after 1 file 1]--------|-----[inserted 2-3 file 2]----
                               -| 222
                                | 333
                                |-
-------[after 1 file 1]--------|-----[inserted 2-3 file 3]----
                               -| 222
                                | 444
                                |-
Do you want the INSERTION made in file 2? [yes]
Applying INSERT from file 2 [lines 2-3]
Do you want the INSERTION made in file 3? [no] yes
Applying INSERT from file 3 [lines 2-3]
============
============
Moved contributor "foo" to "foo.contrib.5".
Output of merge is in "foo".
Recorded merge of "foo".
$ cat foo
111
222
333
222
444
$ ct lsvtree -merge foo
foo@@/main
foo@@/main/0
foo@@/main/mg
foo@@/main/mg/1
 -> /main/1
foo@@/main/mg/3
 -> /main/CHECKEDOUT
foo@@/main/3
foo@@/main/CHECKEDOUT view "marc"
```

The result is that the base and to contributors are again different, hence the merge non-trivial.

Furthermore, the options we were prompted didn't once again leave us the choice of selecting anything satisfying (note though the automatic creation of the Merge arrow in the `lsvtree` command output above).

We'll record the first remark as advice: do not modify the checkedout version if you wish the merge to be trivial. Let's try and investigate the cause for the surprising prompts; thus clean up and retry without any change to the checked out version:

```
$ ct unco -rm foo
Checkout cancelled for "foo".
$ ct co -nc -bra /main foo
cleartool: Warning: Version checked out ("/main/3") is different from ###
                 version selected by view before checkout ("/main/mg/3").
Checked out "foo" from version "/main/3".
$ ct merge -to foo -ver /main/mg/3
*******************************
<<< file 1: /vob/test/foo/merge/foo@@/main/mg/1
>>> file 2: /vob/test/foo/merge/foo@@/main/mg/3
>>> file 3: foo
*******************************
-------[after 1 file 1]-------|-----[inserted 2-3 file 2]----
                              -| 222
                               | 333
                               |-
-------[after 1 file 1]-------|-----[inserted 2 file 3]------
                              -| 222
                               |-
Do you want the INSERTION made in file 2? [yes]
Applying INSERT from file 2 [lines 2-3]
Do you want the INSERTION made in file 3? [no]
============
============
Moved contributor "foo" to "foo.contrib.6".
Output of merge is in "foo".
Recorded merge of "foo".
$ cat foo
111
222
333
```

We now obtain the intended result in the data, but after being prompted: the merge was not trivial, and that is because the base contributor was selected to be the first version in the branch.

This is only possible because no *Merge arrow* was actually created from `/main/mg/2` to `/main/3` as a part of the step in which we forced the base contributor (contrary to our expectation, reinforced by the fact that, as we'll see later, such arrows *are* created in a similar context by `findmerge`)!

To verify this theory (that the cause was the absence of the arrow), let's clean up again, and create ourselves the expected and missing hyperlink and check that its presence suffices to drive the expected behavior:

```
$ ct unco -rm foo
Checkout cancelled for "foo".
$ ct lsvtree -merge foo
foo@@/main
foo@@/main/0
foo@@/main/mg
foo@@/main/mg/1
 -> /main/1
foo@@/main/mg/3
foo@@/main/3
$ ct merge -ndat -to foo@@/main/3 -ver /main/mg/2
Recorded merge of "foo".
$ ct lsvtree -merge foo
foo@@/main
foo@@/main/0
foo@@/main/mg
foo@@/main/mg/1
 -> /main/1
foo@@/main/mg/2
 -> /main/3
foo@@/main/mg/3
foo@@/main/3
$ ct co -nc -bra /main foo
cleartool: Warning: Version checked out ("/main/3") is different from ###
                 version selected by view before checkout ("/main/mg/3").
Checked out "foo" from version "/main/3".
$ ct merge -to foo -ver /main/mg/3
Trivial merge: "foo" is same as base ####################################
                                    "/vob/test/foo/merge/foo@@/main/mg/2".
Copying "/vob/test/foo/merge/foo@@/main/mg/3" to output file.
Moved contributor "foo" to "foo.contrib.7".
Output of merge is in "foo".
Recorded merge of "foo".
$ cat foo
111
222
333
$ ct lsvtree -merge foo
foo@@/main
foo@@/main/0
```

```
foo@@/main/mg
foo@@/main/mg/1
 -> /main/1
foo@@/main/mg/2
 -> /main/3
foo@@/main/mg/3
 -> /main/CHECKEDOUT
foo@@/main/3
foo@@/main/CHECKEDOUT view "marc"
$ ct ci -nc foo
cleartool: Warning: Version checked in is not selected by view.
Checked in "foo" version "/main/4".
```

We created the missing *Merge arrow* with the merge tool, using the `-ndata` flag. We could as well have used `mkhlink`, explicitly with the `Merge` type:

```
$ ct mkhlink Merge foo@@/main/mg/2 foo@@/main/3
Created hyperlink "Merge@232@/vob/test".
```

In the same way, to remove a *Merge arrow*, maybe in order to remove an element (as the presence of hyperlinks suffices to restrict the use of `rmelem` to the vob owner or the administrator), one may use either `rmmerge`, or `rmhlink` with a hyperlink name obtained with `describe -l`.

Note that `rmver` is also protected in a similar way from removing interesting versions, for example, the versions bearing hyperlinks. The way to bypass the protection is different, though—it is to use an `-xhl` flag (`-xla` for labels):

```
$ ct rmver foo@@/main/mg/3
cleartool: Error: Removal of "interesting" versions must be explicitly ##
                                                                  enabled.
Not removing these "interesting" versions of "foo":
 /main/mg/3 (has: hyperlinks)
cleartool: Error: No versions of "foo" to remove.
$ ct rmver -xhl -f foo@@/main/mg/3
Removing these versions of "foo":
 /main/mg/3 (has: hyperlinks)
Removed versions of "foo".
```

And by the way, `rmbranch` is protected in a different way: please refer to the section *Branches and branch types* from *Chapter 6*, *Primary Metadata*.

Despite the fact that we found ways to resolve the various contradictions and obtain the properties we wanted—triviality (automatic handling), expected outcome and tracing of the contributions—the complexity of the situations met may still be felt daunting. To recapitulate what is in our experience the most surprising to users: the merge behavior is driven in part by the presence or absence of Merge arrows, which depends on the prior history of merging.

In particular, some user communities will avoid subtractive merges and prefer destructive rollbacks, that is, removal of the versions resulting from the merges. We believe this is counter-productive: no less error-prone and removing the evidence on the basis of which the problems met may be analyzed.

In other words, radical as it may be, such a strategy is not radical enough; the problem is not subtractive merging, it is publishing by merging, instead of publishing in-place by moving labels.

Merging directories

The issue of merging directories is actually quite different from that of merging files.

Let's note that, under ClearCase, directories are not stored in containers (standard file objects found in vob *pools*) unlike file elements, so that their merging cannot be delegated to external tools, acting upon inodes accessed over NFS. It has to use ClearCase functions at the low level, to create and remove hard links.

In fact, with large directories, it is often more convenient and more manageable to use `rm` and `ln` explicitly than to use `merge`, and correct the results before checking in:

```
$ ct co -nc -bra /main .
cleartool: Warning: Version checked out ("/main/6") is different from ###
                  version selected by view before checkout ("/main/mg/3").
Checked out "." from version "/main/6".
$ ct ln .@@/main/mg/3/foo .
Link created: "./foo".
$ ct ln .@@/main/mg/3/foo1 ./zoo
Link created: "./zoo".
$ ct rm bar
Removed "bar".
$ ct ci -nc .
cleartool: Warning: Version checked in is not selected by view.
Checked in "." version "/main/7".
```

This is especially true when trying to restore an entry removed in a previous version, or to ensure that parallel works will not result in the creation of *evil twins*, by selectively sharing some work before its delivery.

We have already dealt with the issues of recovering files by using `ln`, hard links and evil twins in *Chapter 4*, *Version Control*.

Let's now demonstrate how merging directories may produce hard links, and let's make it more obvious by making these hard links of the same element under different names.

In the example we start from, the file name exists only on an `mg` branch of the directory.

We'll first merge this "back" to the main branch, then rename the entry in the branch:

```
$ ct lsvtree .
.@@/main
.@@/main/0
.@@/main/mg
.@@/main/mg/1
$ ct co -nc -bra /main .
cleartool: Warning: Version checked out ("/main/0") is different from ###
                  version selected by view before checkout ("/main/mg/1").
Checked out "." from version "/main/0".
$ ct merge -to . -ver /main/mg/1
*******************************
<<< directory 1: /vob/test/foo/merge@@/main/0
>>> directory 2: /vob/test/foo/merge@@/main/mg/1
>>> directory 3: .
*******************************
-------[ directory 1 ]---------|-----[ added directory 2 ]----
                              -| foo --08-01T12:04 marc
*** Automatic: Applying ADDITION from directory 2
Recorded merge of ".".
$ ct ci -nc .
cleartool: Warning: Version checked in is not selected by view.
Checked in "." version "/main/1".
$ ct co -nc .
Checked out "." from version "/main/mg/1".
$ ct mv foo bar
Moved "foo" to "bar".
$ ct ci -nc .
Checked in "." version "/main/mg/2".
```

Now, we'll create a second branch of the directory (of a new `fff` type), and rename the entry there as well. We do this in order to confuse `merge` so that it doesn't remove the original name.

```
$ ct mkbrtype -nc fff
Created branch type "fff".
$ ct mkbranch -nc fff .@@/main/1
Created branch "fff" from "." version "/main/0".
Checked out "." from version "/main/fff/0".
$ ct mv foo zoo
Moved "foo" to "zoo".
$ ct ci -nc .
cleartool: Warning: Version checked in is not selected by view.
Checked in "." version "/main/fff/1".
```

Last, we merge the two branches at the same time into the `main` one, which we previously checked out.

The base contributor is selected to be `/main/0`, which results in keeping the original name `foo`, as well as adding the other two.

```
$ ct co -nc -bra /main .
cleartool: Warning: Version checked out ("/main/1") is different from ###
                    version selected by view before checkout ("/main/mg/3").
Checked out "." from version "/main/1".
$ ct merge -to . .@@/main/mg/2 .@@/main/fff/1
*******************************
<<< directory 1: /vob/test/foo/merge@@/main/0
>>> directory 2: .@@/main/mg/2
>>> directory 3: .@@/main/fff/1
>>> directory 4: .
*******************************
-------[ directory 1 ]-------|-----[ added directory 2 ]----
                             -| bar  --08-01T12:04 marc
*** Automatic: Applying ADDITION from directory 2
------[ directory 1 ]--------|-----[ added directory 4 ]----
                             -| foo  --08-01T12:04 marc
*** Automatic: Applying ADDITION from directory 4
-----[ directory 1 ]---------|-----[ added directory 3 ]-----
                             -| zoo  --08-01T12:04 marc
*** Automatic: Applying ADDITION from directory 3
Recorded merge of ".".
$ ls -la .
total 1
drwxrwxr-x 2 marc jgroup 69 Aug 1 20:45 .
drwxrwxr-x 4 marc jgroup 0 Jul 31 21:41 ..
dr-xr-xr-x 1 marc jgroup 0 Aug 1 12:04 bar
dr-xr-xr-x 1 marc jgroup 0 Aug 1 12:04 foo
dr-xr-xr-x 1 marc jgroup 0 Aug 1 12:04 zoo
```

We check that the three names are indeed hard links of the same element, by printing out their common *object id*:

```
$ ct des -fmt "%n %On\n" bar@@ foo@@ zoo@@
bar@@ 7cc55281.9d5d11df.926d.00:01:84:2b:ec:ee
foo@@ 7cc55281.9d5d11df.926d.00:01:84:2b:ec:ee
zoo@@ 7cc55281.9d5d11df.926d.00:01:84:2b:ec:ee
$ ct ci -nc .
cleartool: Warning: Version checked in is not selected by view.
Checked in "." version "/main/2".
$ ct lsvtree -merge .
.@@/main
.@@/main/0
.@@/main/mg
.@@/main/mg/1
```

```
 -> /main/1
.@@/main/mg/2
 -> /main/2
.@@/main/1
.@@/main/fff
.@@/main/fff/1
 -> /main/2
.@@/main/2
```

We showed here how merging directories is in fact handling hard links. We showed that the merge tool doesn't offer any significant additional safety against producing duplicates of the same element: *caveat emptor*. Let's examine this situation, and the reasons there might be to manage it.

It is the second time in this book we meet this case: we already mentioned it as a technique to reach certain goals in *Chapter 2*, *Presentation of ClearCase*, in the paragraph on config specs. We admitted that these goals were exceptional, but reckoned they might be legitimate, and we stated that offering a solution was better than forcing the users to all too simple and more dangerous workarounds.

Hard links open the door to selecting different versions of the same elements in the same view (via scope rules). This fights *the* basic strategy of SCM for identifying resources in two distinct steps: as a set and as a certain item in the set. Achieving this with hard links is a means to ensure the case remains manageable and will be reminded (for example at the time of applying labels: the application will be attempted once for every name, and while the first attempt will succeed, any latter will fail—unless using a `-replace` flag to silence the error).

But there is a case in which this may result in random looking errors, and it is when the hard linked elements are directories. Two preliminary remarks:

- The IBM documentation warns against creating hard links of directories in other contexts than directory merges; doing so is a red flag for the support.
- The procedure exclusively using merge that we showed above and which resulted in multiple names for the same element, did not depend on the element being a file; it works exactly the same with directories.

Now, the problem. As mentioned earlier, unlike files, directories are not mapped to cleartext containers, that is, to file objects outside the database. With files, the view merely identifies or produces a container, which it returns to the application, the interaction between the two being thus an atomic transaction. Not so with directories: the view, and thus the mvfs kernel module below it, services in turn all the requests as the application generates them. The problem is that concurrent applications selecting different versions of the same directory may interfere over time with each other. The effects are not easily predictable, but may at least be incorrect version selections or stale nfs handles.

Note that there has been a longstanding prevention in UNIX against hard linking directories. This was rooted, we believe, in recursion due to cycles while navigating directory trees, which led in the worst cases to file systems corruption. It is still easy to experiment such issues (i.e. endless recursion, not file system corruption) using the *find* tool down from the `/view` root, even if this may be a case of mount point recursion. The fact is directory hard links are pervasive in ClearCase MVFS (in the view extended space).

Rebase or home merge

After presenting multiple examples of *trivial merges* produced with the `merge` command, we must apologize to our reader for what could be seen as confusing and contradictory: we pushed rather far the extent of the acknowledgement that we would *describe a mechanism, not recommend a process*.

Let's make our point explicit: *any merges resulting in mere duplication of data are to be avoided whenever possible*. Note how this precisely condemns trivial and copy merges!

SCM attempts to factor commonalities and make *real* differences stand out. On the contrary, using both of two identical versions introduces *spurious* differences into the dependency tree of dependent artifacts, and results in invalidating derived objects for artificial reasons. This harmful noise makes it harder to detect meaningful differences.

There remains one legitimate, necessary use of merging. The issue is to resolve *real* conflicts between versions that typically arise in the context of parallel development when different users concurrently modify the same resource: at some point, one user notices that the baseline from which she started has now moved forward to integrate somebody else's contributions. This *purpose* is well covered by the concept of *rebase*. We feel however that speaking rather of **home merge** is *process agnostic*, hence more general; the idea is to modify one's working version with changes coming from somewhere else.

In any case, irrespective of how you name them, *rebase* operations can be carried out with the `merge` tool, as well as, for bulk merges (see below), with `findmerge`.

Let's dive back, for one more time, into the context of the (ill-advised as it is) tradition of publishing by merging *back* to *main* or *integration* branches. In this context, one attempts to achieve *trivial* merges for the *delivery* as a means to reduce the inherent instability of the process, which is to make the symptoms bearable instead of curing their causes.

The best way to ensure that a merge will be *trivial* is to have performed it already, to empty the remaining merge from its real contents. This is achieved by performing a preemptive merge in the reverse direction from the one actually intended. The object is to reach a situation satisfying the condition for trivial merges: the coincidence of the base and the *to contributors*. The notion of merging direction being inherently confusing, this first preemptive merge is termed *rebase*, and the next "merge" operation is consistently assimilated to its purpose: *delivery*.

Rebasing alone is not totally foolproof, as it takes a non-negligible time (especially if it concerns many files); there is always a window of opportunity for changes to take place in the integration branches between the "rebase" and the "delivery". This may be prevented with locking, and/or by wrapping up the whole procedure into a "higher level" and heavier operation (what UCM does).

A yet more desperate alternative to trivial merges is to force the delivery to use *copy* merges, that is, to ignore any unlikely but possible changes that might have happened since the last rebase. There are various ways to achieve this, the simplest involving an operating system copy instead of merge (and possibly using `merge -ndata` in addition, only to create the *Merge arrow*). The copy merge will also protect against surprises resulting from former subtractive merges.

A last note concerning automatic, hence *trivial* merges: fine-tuning the merge tool to resolve the differences correctly and perform the actual merges without, or with as little user intervention as possible, is in itself a useful task, as it enforces reproducibility and reduces the possibility for user errors. We saw how managing the base contributor could, in some cases, be a tool to affect this aspect of things.

Complex branching patterns

Until now we have unfortunately only scratched the surface of the real complexity met when trying with such technologies, to maintain cascading branching patterns, with several levels of integration and different release projects running in parallel. If these levels and projects have been assigned specific branch types, one will need to merge the same changes to several places in turn (including thus the possible rebasing).This alone would work as a red flag in the absence of all too common process anesthesia.

Just consider the task of identifying the base contributors in the contexts of the various patterns!

Rather than delving into such intricacies, we feel it is time to remind that they are purely artificial—the consequences of an adverse choice to stick to a tradition inherited from the early systems that had no support for branches—and to try to get back at delivery time to this historical simplicity.

Let us thus send you back to the advice spelled in *Chapter 5*, under the title: *Avoid depending on mastership*, and publish in-place, exclusively using labels.

Following this advice, one may share the same versions in the different contexts in which it makes sense (by applying different labels), and merging is left to the only cases from where it cannot be expelled: for resolving contradictions, where it need not anymore be called *rebasing*.

Rollback of in-place delivery

We asserted that in-place delivery would be reversible. Now is the time to show it in practice, on an example. First the trivial case: a new version of the file `foo` was edited in an `mg` branch, spawn from the baseline represented by the floating label type `AAA`, equivalent at this point to the fixed `AAA_2.37`. The situation prior to the delivery was thus seen with the `lsgenealogy` command from our ***ClearCase::Wrapper::MGi*** wrapper, invoked with a depth of two levels of ancestry:

```
$ ct lsgen -d 2 foo
foo@@/main/mg/5 (MG, MG_1.25)
 foo@@/main/mg/1
   foo@@/main/ts-012/7 (AAA, AAA_2.37, TS-005, TS_3.01)
```

After the delivery, including *archiving* (see Chapter 6) of the branch (`mg` -> `mg-003`) and label (`MG` -> `MG-013`) types, the situation changed to the following (only labels have been applied, and branch types renamed *away* in order to allow existing config specs to retain their semantics in presence of a modified environment):

```
$ ct lsgen -d 2 foo
foo@@/main/mg-003/5 (AAA, AAA_2.38, MG-013, MG_1.25)
 foo@@/main/mg-003/1
   foo@@/main/ts-012/7 (AAA_2.37, TS-005, TS_3.01)
```

At this point, the rollback would be, as announced earlier, trivial, and would yield:

```
$ ct lsgen -d 2 foo
foo@@/main/mg-003/5 (AAA_2.38, MG-013, MG_1.25)
 foo@@/main/mg-003/1
   foo@@/main/ts-012/7 (AAA, AAA_2.39, AAA_2.37, TS-005, TS_3.01)
```

Let's not forget that this triviality contrasts with the weight of the procedures one might consider to use instead in the context of delivery by merging, and which would be in no way equivalent!

But let's consider now that instead of this rollback, there takes place a new phase of development (in the `ts` branch) before the flaw leading to the need to rollback is discovered:

```
$ ct lsgen -d 3 foo
foo@@/main/ts/1 (TS, TS_3.03)
 foo@@/main/mg-003/5 (AAA, AAA_2.38, MG-013, MG_1.25)
   foo@@/main/mg-003/1
     foo@@/main/ts-012/7 (AAA_2.37, TS-005, TS_3.01)
```

Let this get delivered too:

```
$ ct lsgen -d 3 foo
foo@@/main/ts-013/1 (AAA, AAA_2.39, TS-006, TS_3.03)
 foo@@/main/mg-003/5 (AAA_2.38, MG-013, MG_1.25)
   foo@@/main/mg-003/1
     foo@@/main/ts-012/7 (AAA_2.37, TS-005, TS_3.01)
```

And now, let's consider the challenge of rolling the previous change back!
The task decomposes in two steps:

- Rolling back to the stage before the faulty delivery
- Reapplying the changes that took place later, "on top of it"

We have already considered the first step; what happened later doesn't affect the pre-delivery stage version (labeled `AAA_2.37`).

The question is thus to merge the next changes (from the version carrying the label `AAA_2.39`), and to subtract from them the intermediate ones (`AAA_2.38`), but this time creating a unique new version and not a duplicate of an old one (`AAA_2.37`). The first merge is a normal ClearCase one; however, the subtractive merge is not. ClearCase subtractive merges can only affect those changes that took place in the same branch. We'll have to resort to the UNIX diff and patch tools that we mentioned in Chapter 2. We'll use them with a `-c` flag to generate and use context information, allowing to apply the negative differences to a different version of the file than the one on the basis of which they were computed.

```
$ ct co -nc foo@@/AAA_2.37
Created branch "mg" from "foo" version "/main/0".
Checked out "foo" from version "/main/mg/0".
$ ct merge -to foo -ver AAA
Trivial merge: "foo" is same as base ##################################
                                     "/vob/test/merge/foo@@/main/ts-012/7".
Copying "/vob/test/merge/foo@@/main/ts-013/1" to output file.
Moved contributor "foo" to "foo.contrib".
Output of merge is in "foo".
Recorded merge of "foo".
$ diff -c foo@@/AAA_2.38 foo@@/AAA_2.37 > foo.patch
```

```
$ patch -c foo <foo.patch
 Looks like a normal diff.
done
$ ct ci -nc foo
Checked in "foo" version "/main/mg/1".
$ ct mklabel MG foo
Created label "MG_1.26" on "foo" version "/main/mg/1".
Created label "MG" on "foo" version "/main/mg/1".
$ ct mklbtype -nc -inc AAA
Created label type "AAA_2.40".
$ ct mklabel -over MG AAA
Created label "AAA_2.40" on "./foo" version "/main/mg/1".
Moved label "AAA" on "./foo" from version ##############################
                                          "/main/ts-013/1" to "/main/mg/1".
$ ct lsgen -d 4 foo
foo@@/main/mg/1 (AAA, AAA_2.40, MG, MG_1.26)
[siblings: foo@@/main/mg-001/1]
foo@@/main/ts-012/7 (AAA_2.37, TS-005, TS_3.01)
foo@@/main/ts-013/1 (AAA_2.39, TS-006, TS_3.02)
 foo@@/main/mg-003/5 (AAA_2.38, MG-013, MG_1.25)
  foo@@/main/mg-003/1
    [alternative path: foo@@/main/ts-012/7]
```

Several points require comments. First, we used our wrapper for checking out, with explicit (in the config spec) support for implicitly branching off `/main/0`. This is what explains that the successor to `foo@@/AAA_2.37`, a.k.a `foo@@/main/ts-012/7`, is checked out from `foo@@/main/mg/0` instead of from `foo@@/main/ts-12/mg/0`. You'll remember there our strategy to avoid cascading forever.

Next, we show gradually deeper views of the genealogy, as we add new generations. This shows two parents (or contributors) to the same `foo@@/main/mg/1`, and presents them at the same level of indentation. We also notice two annotations to the genealogy output: information that version `/main/mg/1` has at least one *sibling*, not shown in this genealogy, but spawn from the same parent; and a reminder that version `/main/ts-012/7` is reachable by two distinct paths, and thus need not be repeated.

There is one important detail we ought not to hide: when we take the `AAA_2.37` label as the version to rollback, we make a *human*, interactive decision, not an automatic one. In this case, the choice is trivial, but if we were dealing with many files, nothing would guarantee the previous label would be the same for all. This is, however, only a missing functionality of our wrapper; in the next version, it will compute the correct version to rollback to, on an element basis.

This will allow the rollback procedure (the "first part", as mentioned earlier, and which we did not replay as an intermediate step) to be fully automatic.

The second part, reapplying the later changes, will however remain potentially interactive: it is a *real* merge, resolving differences, and producing a unique new version.

This example, which ought still to be concluded with an archiving, showed the clear superiority of our delivery model over the traditional one. If one still needs to convince oneself, one may compare this earlier case to the functionally simpler scenario we reviewed in the *Managing contributions* section. Do we need to list the advantages?

- **Atomicity**: No intermediate state, such as checking out
- **Reversibility**: The rollback itself is a toggle, rolling forward again would be instantaneous
- **Simplicity**: No version added
- **Fairness**, in a MultiSite context (using local labels): Delivery and rollback are decoupled from branch mastership concerns (consistent state of remote labels cannot be trusted anyway)
- **Speed** and hence **robustness**: No window of opportunity for errors

The further advantage we attempted to exploit is the practical availability of all the contributor versions for further fixing or development.

One may be tempted, and we'll not decide here, to modify the topology of the graph drawn by the Merge arrows (that is, to rewrite history) so that the version `AAA_2.38` which we rolled back would not be seen anymore as part of the genealogy of `AAA_2.40`. As explained previously, such a change could avoid later surprises with using the merge tool. We would of course retain both `2.37` and `2.39` as direct contributors, but replace the two Merge arrows, from `2.37` to `2.38` and from `2.38` to `2.39`, with a single and direct one, from `2.37` to `2.39`. We leave it to our reader to decide whether such bold historical revisionism would increase or decrease confusion.

Bulk merges

We've been analyzing merging in the simple case of one single element. Obviously, the more usual case concerns a whole change set, so we have to transpose what we came up with so far in the context of another tool: `findmerge`. As the name suggests, it combines the functions of `find` and of `merge`, performing in addition the necessary checking out in the middle. This applies to all the cases of merging we met, especially to the real merges, a.k.a rebases, with the possible exception of the *copy* merges (back to this below).

Because `findmerge` is expected to perform `checkout` operations, the *to contributors* are implicitly the ones selected by the view. The function has a lot of options, and we'll only cover the most useful ones. The most idiomatic (simplest) is probably to design the other set of contributors (*from*) with a label, using the `-fversion` option. One may also use another view and `-ftag`, but this slightly raises the risk of creating evil twins.

```
$ ct findmerge . -nc -fve FOO -merge
```

This will produce `.contrib` as well as a `findmerge.log.<date-time>` private files, and leave the modified elements checked out.

In case of non-trivial merges, it will prompt the user, which as we already stated, but even more so in the context of `findmerge`, which is often inconvenient.

There are two ways to improve on this:

- One is to add the `-gmerge` flag after `-merge`, which will start a graphical merge: clearly not to our taste, but maybe yours? Note though that such an option can be more harmful than useful (refer to the section named *Evil twins* later in the chapter).
- The other is to add the `-abort` flag after it: this will go on handling all the trivial cases, leaving you with only the interactive ones.

The `findmerge.log` file then proves to be of value: it contains an executable shell script to replay the remaining commands (the part already completed being commented away). You'll have to edit it (for example, remove the `-abort` flags), but at the very least, it gives you the list of files to process. Here is an example of the `findmerge.log` file:

```
$ cat findmerge.log.2010-08-06T14:28:12+01:00
#cleartool findmerge .@@/main/22 -d -fver /main/t1/6 -log /dev/ #########
                                    null -fbtag myview -merge -nc -abort
# Skipping "./bbb.txt".
# Skipping "./ccc.txt".
# Skipping "./mak".
# The following element is invisible in the "base" view: #################
                             /view/myview/vob/test/.@@/foo/t1/4/Makefile.
# Findmerge will select base contributor. The following is not a valid ##
           common ancestor: /view/myview/vob/test/.@@/main/t1/4/Makefile.
#cleartool findmerge ./tc/Makefile@@/main/0 -fver /m/t1/5 -log /dev/ ####
                                    null -fbtag myview -merge -nc -abort
# The following element is invisible in the "base" view: #################
                            /view/myview/vob/test/.@@/main/t1/4/pathconv.
# Findmerge will select base contributor. The following is not a valid ##
          common ancestor: /view/myview/vob/testi/.@@/main/t1/4/pathconv.
```

```
#cleartool findmerge ./tc/pathconv@@/main/0 -fver /main/t1/4 -log /######
                                  dev/null -fbtag myview -merge -nc -abort
# Skipping "./tmp/a.txt".
```

Another useful option is `-fbtag`: specifying the current view will force trivial merge by explicitly setting the *base* contributors to match the *to* ones.

Note that this doesn't guarantee copy merges, in the case of a history of subtractive ones.

To guarantee this, one might use the `-exec` option. One problem with this option (common to its use with the `find` command) is that it will spawn a new invocation of the specified command for every match. For this reason, it is not a good idea to use it to invoke **cleartool** commands as most of the load (and thus of the time) will be spent in initializing and finalizing the cleartool processes.

There also is a `-co` flag, which comes handy. We use it here in conjunction with `-exec` (despite our strict understanding of the man page) to achieve the *copy* merge (we have to deal with directories first, and to cope with file names possibly containing spaces):

```
$ ct findmerge . -nc -type d -fve FOO -fbtag marc -merge
$ ct findmerge . -nc -type f -fve FOO -fbtag marc -co \
 -exec 'cp "$CLEARCASE_FXPN" "$CLEARCASE_PN"'
Needs Merge "./f2" [(automatic) to /main/mg/1 from /main/1 ###############
                                                  (base also /main/mg/1)]
Checked out "./f2" from version "/main/mg/1".
Needs Merge "./f1" [(automatic) to /main/mg/3 from /main/5 ###############
                                                  (base also /main/mg/3)]
Checked out "./f1" from version "/main/mg/3".
Log has been written to "findmerge.log.2010-08-02T20:32:48+01".
```

`findmerge` command may also be used to test the effect of command without actually running them. This is obtained with the `-print` option, which may be usefully augmented with `-whynot`.

Another possible use of the `-print` option is to produce lists of versions to be merged, for arbitrary post-processing. In this case, the `-short` addition is probably useful, to restrict the output exclusively to lists of contributors (*to* first).

There may however still be a few corner cases, involving directory merges, in which the output obtained with such dry runs could diverge from the result of actually effectuating the merges.

Evil twins

We already mentioned evil twins in Chapter 4, so why mention them again when speaking of merging?

The answer is that one cannot (easily) merge evil twins. Merging, contrarily to patching, concerns versions of the same element, and evil twins are precisely different elements.

The fact is that evil twins are for this reason often detected while merging, and more specifically, while merging whole directory trees. It comes as a surprise to many users that `findmerge` will fail to find suitable contributors, a situation they may show you with a screen dump but cannot explain.

It must be said that it is our experience that evil twins are more frequently produced by Windows GUI users than by command line UNIX ones. The reason is that the GUI puts in the hands of a naive user a "power" she doesn't master, offering an illusion of help, but on the contrary contributing to adding to the complexity. As we already mentioned, interactive decisions based on perception are not traceable.

How then to merge evil twins? The best tool is once again ***synctree!*** Importing directory trees from one view to another, using the powerful `-vreuse` and related options to avoid adding to an already regrettable chaos.

Summary—wrapping up

We hope that we could give some evidence of the complexity of merging. We believe that this complexity is typically underestimated, and that designing development processes based on branch patterns that imply a lot of merging is asking for trouble: it spreads this complexity to realms in which it doesn't belong, making the resulting complexity artificial.

We believe that rolling back changes gives a convincing example: it is incomparably simpler and more reliable to move back a label than to perform a subtractive merge. The apparent problem with the former is that the version belonging to the baseline, the one bearing the label, is after the rollback not the latest on the branch. Why should *this* be a problem? We can only find bad and artificial reasons: "Doctor, when I hit my head on the wall, it hurts".

If one decides to identify the baseline with labels, one doesn't care anymore on what branch the baseline version of any element *is* sitting: merging "back" to integration branches cannot be justified.

We met here a new set of reasons to stick to the advice we gave first in Chapter 5, and which we came back to already in Chapter 6—to publish in-place.

8
Tools Maintenance

Tools are an essential concern in build management. If we consider the mass of information that software development has to deal with, and structure it as sets of software configurations, that is, (using the formidable tool ClearCase offers us) as dependency trees, then we find tools at both boundaries: as the *roots* of the dependencies, with sources; at their *leaves*, with the customer deliverables (as part of the run-time environment).

The opacity of tools, as well as that of sources, is of course only a matter of viewpoint: the boundaries are only surfaces of separation between realms of distinct coherence. Some other organization is responsible for developing and delivering them, but typically under a different SCM.

This chapter will be structured in a traditional way, answering two questions about maintaining tools under ClearCase, that is maintaining them in vobs:

- Why?
- How?

In the first part, we'll offer an analysis, which is too often skipped, that is, the questions are skipped even after the answers have popped up. We'll thus, as earlier, question some traditional answers.

The second part we choose to confront through an example, convinced as we are that *the Devil is in the details*. The review will show the practical difficulty of anticipating changes beyond one's control and the compromises this leads to.

Why?

There are many reasons for managing third-party tools under ClearCase:

- Dependency control during builds: we want to ensure the reproducibility of our build. As the build depends on third-party tools (libraries, compiler, and so on), we naturally want to keep track of those dependencies.
- Safety with updates: we may easily switch back and forth between successive versions.
- Referential transparency: we must be able to switch between versions of tools without having to change our software or makefiles.
- Flexibility: in most cases, simultaneous use of different versions must be possible.
- Localization/Fixing/Enhancing: we can configure, or even correct or modify products of which we have the sources. We could report or publish such modifications, in the hope that they are being integrated into the next official versions.
- Tracking of indirect dependencies: third-party components may themselves use chains of other components, which we may use in different contexts. We want to be able to detect such dependencies and possible conflicts between them.
- Replication via MultiSite: we can share the installed products (and if need be, maintain different local configurations).

Dependency control

We want to conform our results to our intentions, and thus to exclude any factors of instability. We want to be able to compare our results: our own successive ones, as well as ours with others'. We need to ensure the consistency of our builds in that all parts of them use the same or compatible tools. And finally we want to be able to analyse the results in order to identify and fix the cause of problems, which we need to narrow down and reproduce in isolation.

If our build depends on a tool *X*, and *X* is not under ClearCase, then clearmake offers only limited support:

- It records a dependency on the build script, with the values of makefile variables expanded
- It records the time stamps of *declared* dependencies

This seems to open the way to two strategies: explicit declaration and hardcoding version information in path names. The former is not really an option on a larger scale: there are just too many files to name. It is important however to investigate it, as it may pop up as a recourse in case of failures with the other.

We'll thus try to explore both and show their shortcomings, so as to justify the extra effort needed to import and maintain the tools in vobs.

Safety with updates

If the product *X* is not managed in ClearCase:

- Derived objects that depend on different versions of the product *X*, will not be validated properly; unconditional rebuild would be required
- Having upgraded *X*, one would need to keep apart an installation of the previous version, for example, in order to reproduce a bug reported by a customer in the configuration in which it was used

Explicitly declare tools as dependencies?

According to the documentation (*Tracking non-MVFS files in **ClearCase IBM Rational ClearCase Guide to Building Software***), clearmake tracks the checksum of explicit dependencies not in a vob.
We found this description insufficient and misleading, as we'll show now.

Let's create a tool named `makefoo`, producing a derived object, `foo`. Let's place it in the `/tmp` directory (hence *not* under ClearCase), and record its time stamp (force it to an arbitrary value) and its size:

```
$ cat /tmp/makefoo
#!/usr/bin/bash

echo zoo > foo
$ touch 08141200 /tmp/makefoo
$ ll /tmp/makefoo
-rwxrwxr-x 1 marc  jgroup 32 Aug 14 12:00 /tmp/makefoo
```

Let's now use our tool in a makefile, in a vob, being careful to declare it as a dependency of our `foo` target:

```
$ cat Makefile
foo: /tmp/makefoo
        /tmp/makefoo
$ clearmake
/tmp/makefoo
```

```
$ ct catcr foo
Derived object: /vob/test/nmvfs/foo@@--08-14T12:07.23206
...
Initial working directory was /vob/test/nmvfs
----------------------------
MVFS objects:
----------------------------
/vob/test/nmvfs/foo@@--08-14T12:07.23206
----------------------------
non-MVFS objects:
----------------------------
/tmp/makefoo <2010-08-14T12:00:00+01>
----------------------------
Build Script:
----------------------------
 /tmp/makefoo
----------------------------
$ clearmake
`foo' is up to date.
```

We checked that a second build gets avoided as it should.

Let's now modify our tool (we replace `zoo` with `bar`), but keeping its size and time stamp unchanged. Of course, its checksum changes:

```
$ perl -pi -e 's%zoo%bar%' /tmp/makefoo
$ touch 08141200 /tmp/makefoo
$ ll /tmp/makefoo
-rwxrwxr-x 1 marc jgroup 32 Aug 14 12:00 /tmp/makefoo
$ clearmake
`foo' is up to date.
```

Let's face it: we are disappointed. From the above referenced documentation, clearmake should have detected the checksum change, and rebuilt. Let's investigate further.

We then force the rebuild (by removing our first product) and check that the tool produces a different result.

```
$ cat foo
zoo
$ rm foo
$ clearmake
 /tmp/makefoo

$ cat foo
bar
```

Our next attempt is now to do a similar modification, only this time, not preserving the size of the tool. We check that the rebuild happens:

```
$ perl -pi -e 's%bar%bar1%' /tmp/makefoo
$ touch 08141200 /tmp/makefoo
$ ll /tmp/makefoo
-rwxrwxr-x 1 marc jgroup 33 Aug 14 12:00 /tmp/makefoo
$ clearmake
 /tmp/makefoo

$ cat foo
bar1
```

Last, we modify only the time stamp—first, to a value newer than the recorded one for the tool, but older than this of the derived object:

```
$ touch 08141210 /tmp/makefoo
$ clearmake
'foo' is up to date.
```

No rebuild! Secondly we set it to a date newer than this of the derived object, which would trigger the behavior of a "standard" make tool (using the `-T` flag would show it, but would obviously defeat our purpose):

```
$ ll foo
-rw-rw-r-- 1 marc jgroup 5 Aug 14 12:13 foo
$ touch /tmp/makefoo
$ clearmake
'foo' is up to date.
$ ll /tmp/makefoo
-rwxrwxr-x 1 marc jgroup 33 Aug 14 12:39 /tmp/makefoo
```

This is downright embarrassing... The explanation for all these behaviors comes from reading the improbable small print of a ***technote (#1386111):*** *both* the time stamp and the checksum must change for clearmake to rebuild! Alone, neither of them is sufficient (but as it happens, the size seems to be...)

We are of course nit-picking (unintentionally!): seldom will the time stamp remain the same while the contents changes. But this possibility, added to the discontinuity with the standard make behavior (ignoring a time stamp newer than this of the source), is enough to cast a doubt: we cannot depend on this functionality.

Lacking confidence in this behavior of the build system, one traditionally resorts to `make clean`, or to building *unconditionally*.

This is not only sub-optimal in terms of build performance, but introduces discontinuities in the development process, and thus trade-offs to upgrading or just tuning one's tools.

Furthermore, it doesn't encourage supporting fine-tuning one's build system, and therefore feeds the vicious circle leading to further lack of confidence.

ClearCase has better to offer!

Let's just move the tool in a vob, as a view private object. We may drop declaring it:

```
$ cat Makefile
foo:
 /vob/test/nmvfs/makefoo
$ ll makefoo
-rwxrwxr-x 1 marc jgroup 33 Aug 14 15:24 makefoo
$ clearmake
 /vob/test/nmvfs/makefoo

$ clearmake
`foo' is up to date.
$ touch makefoo
$ clearmake
 /vob/test/nmvfs/makefoo

$ ct catcr foo | grep 2010
Reference Time 2010-08-14T15:34:12+01, this audit started ################
                                                    2010-08-14T15:34:12+01
/vob/test/nmvfs/makefoo <2010-08-14T15:26:10+01>
$ touch 08141530 /tmp/makefoo
$ clearmake
 /vob/test/nmvfs/makefoo
```

ClearCase takes responsibility for auditing the identity of the tool, using the time stamp for a view private object (and the version for a checked in element).

We are anyway left now with only one alternative to eliminate: hardcoding the version in the tool's pathname. We must reckon that, in the above scenario, this would have worked just fine.

Referential transparency

The situation just described gets worse if one considers having to switch back and forth between several versions of the tools.

Without ClearCase, the traditional solution is to rely upon naming, that is, to hardcode the version of the tools into the paths used to access them.

Of course, one needs then to produce one's derived objects under names or paths matching this hardcoding.

Such a solution may unfortunately be considered in simple cases, before the combinatorial explosion of tool variants gets acknowledged. At that time, one has already invested in generic complexity of the build system (using macros, and/or generating the build scripts).

ClearCase offers a far more elegant solution by decoupling the versioning dimension from the common namespace. Allowing the versions of tools to be selected via the config spec makes it possible for static files (build and other scripts, documentation) to be reused among different software configurations, thus reducing dramatically the global number of artifacts.

Referential transparency—the property for the same name to map to different instances of a resource in different contexts—decreases the artificial complexity and helps making essential differences emerge.

Switching versions of tools, that is, switching software configurations, becomes an incremental task, concerning only a few files. The threshold for experimenting or tuning the environment gets lowered, and the overall quality improves.

With naming-based solutions, the smallest change is an all-or-nothing challenge. This results in postponing fixes, and bundling changes together, which in turn produces massive changes and long term instability. It results in matching investments to expectations, instead of making choices based on real experiments.

Flexibility

It is quite typical that different components of a large system, different applications, do not evolve synchronously with respect to their use of tools. One application may meet a limitation or a bug, which makes it critical for it to move onto the next release of the tool. Feeling the pain, the team is motivated to invest the effort to validate it and to adapt their own software to the API changes. Other applications will, on the contrary, be fighting with different problems and satisfied with the functions offered by the current version.

Adopting a new version, or a new tool in replacement for an existing one, is not a trivial enterprise, and typically takes time and requires making changes. It is highly beneficial that this may happen in-place, without requiring one to clone the existing development environment. Once the evaluation has happened, assuming the decision was made to move onto the new tool, one ought to be able to use the results of the effort without having to pay the fee a second time.

These remarks fight a common bad practice to consider the SCM repository as a certified archive, in other words, as a place to store only *official* artifacts, ones that have gone through some kind of formal approval procedure. As we already stated, management is most useful before the artifacts have been validated. Vobs are the natural place to store the candidates of such a validation, and the validation, as any kind of release, should happen *in-place*.

Tool fixes

The boundary between tuning a tool version and making a new one is itself neither clear-cut nor stable: tools typically have to be configured, and one may even have to maintain workarounds for defects, before a vendor may provide fixes.

Tuning typically depends on usage patterns, and therefore happens naturally late in the development, and in small increments, with trial and error. Clearly, this should defeat a tool maintenance strategy based on naming: with such a strategy, in order to maintain the ability to rollback one's changes or to compare the results with previous ones, every change should result in duplicating a large portion of the tool hierarchy as well as the set of products. This may still be considered in simple cases, but is utterly non-scalable in the presence of combinations of tools and/or platforms.

Indirect dependencies

Third-party tools are themselves built from third (or is it fourth?) party tools. It is commonplace to find the same dependencies crop via different paths, and this is the cause of many consistency concerns: upgrading one tool may force to upgrade several others in order to build up a coherent virtual baseline for their combined dependencies. This is a non-trivial task, in which few vendors will assist, and for which none of them will take responsibility. In fact, this is an excellent argument in favor of the Open Source model.

Managing the tools under ClearCase certainly makes it easier to build up such baselines, and evolve them. You may help ClearCase to help you, by sharing the tools that would be embedded, instead of dumping in various places many instances of the same. This will help detecting problems, which is often a prerequisite for fixing them.

Now, the notion of dependency used in this context need not (and often cannot) be as strict as the SCM notion of dependency (based on identity). There comes the weaker concept of *compatibility* (especially of *backwards* compatibility). One will often have to trade for a common set of tools, adjusted from a compromise between otherwise incompatible tool requirements.

MultiSite replication

Tools maintained in vobs may obviously be replicated, and thus shared between different sites. This leverages the installation work, which may be used to distribute the effort, or... to distribute the pain. It is advisable to opt for a collaborative model, and let sites feeling first the need for a new tool or an upgrade do the investment of installing and validating the new item, rather than for a centralized model, in which such upgrades have to be ordered. The obvious danger of not doing so is that it will not prevent the other sites from installing and using the new release: they will only go underground, and it will be harder to converge back. The more you control, the less you manage!

Our recommendation is here again to remember to depend in config specs on *local* labels, that is, to use remote ones (applied at another site) only as the basis for moving one's own. The rationale is the intrinsic asynchrony of replication, leading to the fact that there is never any guarantee of consistency in the local image of a remote baseline: at every synchronization not only the present, but the *past* may change (we looked at this in *Chapter 5*, *MultiSite*, under the section named *Labels*).

How?

Installing a tool in a vob takes place in two phases:

- The installation proper, to a local ClearCase client
- The import from there into the vob

The import phase can often use `clearfsimport`, from the ClearCase distribution, but in many cases goes much easier with ***synctree***.

The installation phase depends mostly on the way the tool is packaged and distributed. It is however often necessary to consider the import phase first, because it may influence the parameters used to configure the installation (the paths for a start).

We'll start with an example of a tool, the one we announced already in *Chapter* 1, *Using the Command Line*: perl. We'll try to generalize from it.

Example—perl installation under a ClearCase vob, with multi-platform support

Get a perl distribution for the first platform, for example, Linux. But here comes the first choice: binary or source distribution?

Perl will embed the paths to its modules in an `@INC` variable. It will also record the compiler used to build it, so as to use the same for building binaries that might be part of modules to be installed on it (if they so require).

These aspects guide us to build perl ourselves, from a source distribution. However, in the particular case of Windows, it is possible to use a binary distribution and to configure it to use a vob path (associated to a drive letter).

Building perl itself is straightforward, if your environment meets the requirements that are clearly defined in per platform readme files. You may of course want to use the compiler and other tools from a vob, and thus to import them first. You may get into chicken and egg problems, and to solve these, you would have to use binary distributions first.

Before we install it, let's anticipate about what the maintenance will be. Let's thus start by the end, by importing or updating individual modules on top of this vob installation of perl.

Importing CPAN modules

Download the Perl module from CPAN:

```
$ wget http://search.cpan.org/CPAN/authors/id/D/DS/DSB/###################
                                                    ClearCase-Argv-1.49.tar.gz
$ tar xzf ClearCase-Argv-1.49.tar.gz
$ cd ClearCase-Argv-1.49
```

First install locally, but using perl from the vob (which has been imported there already, see below):

```
$ which perl
/vob/tools/perl/bin/perl
$ perl Makefile.PL PREFIX=/home/marc/tmp
$ make
cp Argv.pm blib/lib/ClearCase/Argv.pm
Manifying blib/man3/ClearCase::Argv.3
$ make install
Installing /home/marc/tmp/lib/site_perl/5.10.1/ClearCase/Argv.pm
Installing /home/marc/tmp/man/man3/ClearCase::Argv.3
Writing /home/marc/tmp/lib/site_perl/5.10.1/i686-linux/auto/##############
                                                  ClearCase/Argv/.packlist
```

```
Appending installation info to /home/marc/tmp/lib/perl5/5.10.1/ #########
                                                    i686-linux/perllocal.pod
```

Import from there:

```
$ ct setview v1
$ ct setcs cs/linux
$ ct catcs
element * CHECKEDOUT
element * .../imp/LATEST
mkbranch imp
element * /main/LATEST

$ sb=/home/marc/tmp; db=/vob/tools/perl
$ synctree -sb $sb -db $db -/ipc=1 -reuse -vreuse -ci -yes -label LINUX \
 lib/site_perl/5.10.1/ClearCase/Argv.pm man/man3/ClearCase::Argv.3
```

This method is easy, apart from the issue of updating the `perllocal.pod` and the `.packlist` files.

New information is appended to `perllocal.pod` for every module installation. The problem is that the paths recorded there must be manually fixed after importing to the vob. The file is found in the platform-dependent tree (which is a reason to share this tree between the platforms).

```
$ perl -pi -e 's#/home/marc/tmp#/vob/tools/perl#' \
 /home/marc/tmp/lib/perl5/5.10.1/i686-linux/perllocal.pod
$ ct co -nc $db/lib/perl5/5.10.1/i686-linux/perllocal.pod
$ cat $sb/lib/perl5/5.10.1/i686-linux/perllocal.pod \
 >>$db/lib/perl5/5.10.1/i686-linux/perllocal.pod
$ ct ci -nc $db/lib/perl5/5.10.1/i686-linux/perllocal.pod
$ ct mklabel LINUX $db/lib/perl5/5.10.1/i686-linux/perllocal.pod
```

The `.packlist` files contain the list of files belonging to a package. They are therefore more stable than `perllocal.pod`. They are also found in platform-specific trees.

```
$ cat \
 ~/tmp/lib/site_perl/5.10.1/i686-linux/auto/ClearCase/Argv/.packlist
/home/marc/tmp/lib/site_perl/5.10.1/ClearCase/Argv.pm
/home/marc/tmp/man/man3/ClearCase::Argv.3

$ perl -pi -e "s#/home/marc/tmp#/vob/tools/perl#" \
 ~/tmp/lib/site_perl/5.10.1/i686-linux/auto/ClearCase/Argv/.packlist

$ synctree -sb $sb -db $db -/ipc=1 -reuse -vreuse -ci -yes -label LINUX \
 lib/site_perl/5.10.1/i686-linux/auto/ClearCase/Argv/.packlist
```

Even if it is the easiest method, one often has to import different parts separately: scripts, perl modules, auto-split directories, man pages... For some of them, one may let the clean up to the `-rm` option of synctree (to remove some of the elements if they are not present in the current version of the module), for others it's impossible: it depends whether the destination directories are shared with other modules or not.

The vast majority of CPAN modules are perl only, hence, platform independent. They can thus be imported once for all the platforms. We must only avoid introducing accidental factors of dispersion. Two come to mind at this stage:

- Avoid platform-specific branches
- Avoid platform-specific directory names

For the platform-dependent modules, one could use:

- Separate branches, say, `imp1`
- Platform specific labels—`SOLARIS`

We have to admit that we didn't configure perl to avoid the platform-specific names for a few directories. We could have done it by first disabling the platform-specific lib paths in the perl installation and replacing them with a common path `arch` such as `lib/site_perl/5.10.1/arch` (by using the `-Darchname=arch` option for `Configure` script, see below). And then we could even specify such common paths as `INSTALLSITELIB` and `INSTALLPRIVLIB` parameters when building our module, in an attempt to make our imports easier and thus less error-prone:

```
$ perl Makefile.PL PREFIX=~/tmp \
 INSTALLSITELIB=~/tmp/lib/site_perl/5.10.1/arch \
 INSTALLPRIVLIB=~/tmp/lib/5.10.1/arch
```

Instead, we did share the platform-specific directory names, by renaming them as configured, so that in our experience, the `perllocal.pod` and `.packlist` files mentioned above are shared at the element level but unfortunately not (at least for `perllocal.pod`) at the version level:

```
$ ct setview v2
$ ct setcs cs/solaris
$ ct catcs
element * CHECKEDOUT
element * .../imp1/LATEST
mkbranch imp1
element * LINUX
element * /main/LATEST

$ ct co -nc /vob/tools/perl/lib/site_perl/5.10.1
$ ct mv /vob/tools/perl/lib/site_perl/5.10.1/i686-linux \
 /vob/tools/perl/lib/site_perl/5.10.1/sun4-solaris-thread-multi
$ ct ci -nc /vob/tools/perl/lib/site_perl/5.10.1
$ ct mklabel -rep SOLARIS /vob/tools/perl/lib/site_perl/5.10.1
```

We also renamed the `/vob/tools/perl/lib/perl5/5.10.1/i686-linux` in the same way and then we proceed with importing the Solaris version of the module to the vob, in the same way as described earlier for Linux—in `imp1` branch and we label it with `SOLARIS` label. But this discussion anticipates on the *Upgrading the distribution* section that will come later in the chapter.

Installing the Perl distribution

Get the Perl distribution (the source code) for Linux (version 5.8.8 is chosen on purpose: it anticipates upon the section on upgrading):

```
$ wget http://www.cpan.org/src/perl-5.8.8.tar.gz
$ tar xzf perl-5.8.8.tar.gz
$ ct setview v1
$ ct startview v2
$ ct mkbrtype -nc imp@/vob/tools
$ ct setcs -tag v1 cs/linux
$ ct setcs -tag v2 cs/linux
$ ct catcs -tag v2
element * CHECKEDOUT
element * .../imp/LATEST
mkbranch imp
element * /main/LATEST
$ ./Configure -des -d -Dprefix=/vob/tools/perl
$ make install
$ synctree -sb /vob/tools/perl -db /view/v2/vob/tools/perl \
 -/ipc=1 -reuse -vreuse -ci -yes -label LINUX
```

After the successful import, remove the `v1` view-private directory `/vob/tools/perl` and its content, which is currently eclipsing the imported versions:

```
$ ct des /vob/tools/perl
View private directory "/vob/tools/perl"
$ rm -rf /vob/tools/perl
$ ct des -fmt "%n %l\n" /vob/tools/perl
/vob/tools/perl@@/main/t/1 (LINUX)
```

Repeat for other platforms, for example, Solaris, Windows. At import time, one needs to modify the config spec to take the previous baseline, identified with the `LINUX` labels (as we have just showed above in the *Importing CPAN modules* section).

For those, you'll need to use different label types (for example, `SOLARIS`, `WINDOWS`), and to add one option (already mentioned for modules) to the `synctree` command: `-rm`. This is to remove files (obviously, it was redundant in the very first import) that would not exist in the new source base.

An additional flag is often useful: `-lbmods`, to apply labels only to the versions modified. We choose here to apply one full label per platform though, and thus not to use this flag.

At least with the `Configure` line presented above (the one we used ourselves), we obtain some sub-trees as follows:

```
lib/site_perl/5.8.8/sun4-solaris-thread-multi
lib/site_perl/5.8.8/i686-linux
lib/5.8.8/sun4-solaris-thread-multi
lib/5.8.8/i686-linux
```

One option would be to create hard links for lib directories:

```
$ cd lib/site_perl/5.8.8
$ ls
i686-linux
$ ct co -nc .
$ ct ln i686-linux sun4-solaris-thread-multi
$ ct ci -nc .
```

We suggest rather renaming the platform-specific directories for every platform. It makes it easier to compare the different installations, and even to share some files there (see earlier section named *Importing CPAN modules*). To do so, one needs to prepare the *destination base*, that is, the vob copy, prior to running *synctree*.

Some `Configure` options to consider (see earlier):

- `-Dinc_version_list=none`: This disables an option offered by perl to keep historical versions of some scripts in release-specific directory trees
- `-Darchname=arch`: This standardizes the directory names above (but `thread-multi` is still appended), that is, the platform-specific paths, `sun4-solaris-thread-multi` and `i686-linux`, are replaced by the common one, `arch`:

  ```
  $ ./Configure -des -d -Dprefix=/vob/tools/perl -Darchname=arch
  $ make install
  $ ls /vob/tools/perl/lib/site_perl/5.8.8
  arch
  $ ls lib/site_perl/5.8.8
  arch
  ```

Upgrading the distribution

The situation is slightly different when it comes to upgrading this installation, say from 5.8.8 (which is already under ClearCase) to 5.10.1.

We suggest that prior to upgrading the first platform (for example, Linux), the Perl version-specific directories are renamed manually just as we did for the platform-specific ones:

```
$ ct co -nc /vob/tools/perl/lib/site_perl
$ ct mv /vob/tools/perl/lib/site_perl/5.8.8 \
 /vob/tools/perl/lib/site_perl/5.10.1
$ ct ci -nc /vob/tools/perl/lib/site_perl
$ ct des -fmt "%n %c" lbtype:LINUX1
LINUX1 Perl 5.10.1 for Linux
$ ct mklabel LINUX1 /vob/tools/perl/lib/site_perl/5.10.1
```

Then one can proceed with installation of the new (5.10.1) version and importing it to the vob. Note that one needs to be careful using the `-rm` flag: it is appropriate for the distribution part, but not under the `site_perl` directories, where it would result in removing any CPAN modules that have been installed in the meanwhile.

At the point of importing the second platform, one faces however a dilemma: what baseline should one use?

- The previous baseline for the same platform (which was for 5.8.8)?
- The baseline for the new version, but for the platform just imported?

This is where the `-reuse` and `-vreuse` options of `synctree` make the difference. There is more on this in the section on *Branching and labeling*.

Installation

Time now to try to generalize... or as we'll soon notice, to fail to do so, facing the specificity of each tool. We can only draw our reader's attention to a few issues that need to be considered.

The perl model: extract tar, `Configure` (or `configure`, for example, for GNU tools), and `make install`, is of course not the only one.

The choice of installing binaries or sources may be met with a packaging tool other than tar: for example, with rpm. In the case of Linux, one typically installs first a binary distribution, and it is only when wanting to debug and fix a problem, and possibly to contribute the fix back, that one installs a source distribution, and builds from there. In any case, there are strong reasons *not* to attempt to build the tool under clearmake: it is unlikely that the build system of the tool will meet the requirements that would make builds avoidable under winkin and thus manageable via derived objects. Changes one would make to it could not be contributed back, and would thus constitute a *fork*. Not sharing the results as derived objects, one would have to *stage* (that is, fall back to the model of the binary distribution).

In short, one has to have a good reason to manage the *sources* of the third-party tool under ClearCase and we do not see fit to make it a rule.

The installation typically uses a packaging tool specific to the package format, typically platform dependent (`pkgadd` on Solaris, `rpm` on Linux, `swinstall` on HP-UX, and so on), and the import is done from the installation tree. There is typically an option (such as `-R` for `pkgadd`) to relocate this tree to a path under a vob, but whether this is possible or not depends also on the product itself: some products are hardcoded to be used from certain paths. It may still be possible to import them in a vob, and to use them via symbolic links (which will only work under a view context).

The install tool may require *root* privileges, and the product may depend on certain access rights.

Note that a package targeted to multiple platforms cannot be assumed to install identically on every one of them, so that one could install it and import it only once.

Let's take the example of a Solaris package, intended for both *sparc* and *i386* platforms. The `pkgmap` file found when extracting it drives the installation with `pkgadd`. It contains lines that describe file objects to be installed, one per line. It is however possible and customary to use macros in the paths that are set in the `checkinstall` script, for instance based on the value returned by `uname -p` (precisely, `sparc` or `i386`). In the simplest case, this mechanism will drive the creation of a symbolic link pointing to binaries of the suitable architecture.

Products may themselves contain symbolic links (internal or external), which may have to be adjusted to work under the vob. Again, synctree has an option to assist with this: `-rellinks`.

Using a tool from the vob, one must remember to adjust some environment variables used by various utilities (`MANPATH` to access manual pages, `LD_LIBRARY_PATH` or platform equivalent to access shared libraries, `JAVAHOME`, and so on).

As in the perl case, the first installation is usually simpler than any subsequent ones. It is however the place to make wise decisions, concerning the paths. Fortunately, ClearCase will make it possible to reconsider those decisions, to learn from one's mistakes, and to tune the directory layout to favor **stability**, even if one failed initially to do so.

Subsequent installations of a product may take the existing configuration into consideration. Patch installations will even require an existing installation. Typically, the vob image of the product is not directly suitable: it cannot be overwritten, even with root privileges. One needs therefore to be able to produce a writable copy of the vob image, suitable for the installer, and using the same configuration as for the original install. One option is to keep the original copy after importing to a vob from it. Another, perhaps a better option, is to use a dynamic view with `-none` rules in its config spec, and a view private copy of the vob contents. For copying the vob contents to the view, in case `cp -R` is not suitable on the current platform, one may use the old idiom:

```
$ tar cf - . | ( cd /view/copy$(pwd) && tar xf - )
```

Import

Now that we do have a local installation (after either doing it for the first time or applying a patch, or even in some cases just extracting an archive), our next goal is to import it as such into the vob.

Minor checks prior to importing

We already mentioned, in the earlier section, about the possible issue of symlinks, which may have to be adjusted. Apart from those, special protections may have to require attention. First, the groups used must be declared to the vob. The most exotic protection schemes (such as those used in the vob storages for the contents of the `.identity` directory) may not be supported. Neither can devices, sockets, pipes, and such objects.

It is a good idea to check (and fix) the access rights in the local installation (protections may be corrected in the vob, but the resulting events may be filtered from the synchronization to other replicas, depending on permissions preserving settings): executable files should have the executable bits set (this might not be the case if the image was extracted from a Windows archive); directories should be executable as well, and usually group-writable; file objects should be world readable. Contradictions between these rules and "security" concerns should be handled case by case, examining possible consequences.

Branching and labeling

Traditionally, separate branches types would be recommended:

- For releasing tools
- Local configuration and changes
- Binaries built out of sources

This is not necessary if using synctree with the `-vreuse` and `-label` options. In this case, the goal of selecting versions exclusively with labels, and being agnostic to the actual location in the version tree, may be restored.

The script will reuse existing versions, even if they are not selected: if it finds a suitable one in the version tree, it will apply the new label there instead of importing a duplicate.

Each tool release can be labeled by a fixed label (which may be partial), for example `GCC_2`.

A distinct common *floating* label, for example `TOOLS`, can be used for publishing the whole tools set: it will be applied over a chosen fixed label for each selected tool.

Issues during the import

One issue not automatically handled by the import tools (neither clearfsimport nor synctree) is that of incompatible requirements on the type of some elements between the version already in the vob and the one being imported. This could typically be met with HTML pages (the old version had short lines and could be imported with an old, text based, element type, whereas the new one has lines above 1024 characters and requires a new "binary" element type), but also, for example, with GNU info files.

These problems are difficult to predict and will typically be detected only as import errors, reported by the tool but always easy to miss from verbose transcripts. The elements will be left checked out. The fix requires manual intervention, usually with `chtype` and the appropriate new type. Synctree will attempt to cope with transitions between symlinks and files or directories. Cases in which the same name is assigned between two releases to a file and a directory will require manual removal (`rmname`) of the offending object.

Operating system

ClearCase is not available before late runlevel 2 (3 on HP-UX). This means that everything needed before that cannot be maintained (exclusively) in vobs. Of course, it is possible to switch, once ClearCase is available, and for resources it doesn't use itself, to copies from the vobs (some boot sequences use a distinct local file system in single user mode, not available later) via symbolic links. A view context is of course required to access them. One needs a pair of scripts used at boot and shutdown time to set and reset those links, renaming away and back the local versions of the resources.

This makes maintenance tasks heavier (for example, installing patches) and more disruptive. The set of resources used from the vobs is also not guaranteed to be stable.

Versioning under ClearCase also makes the whole system dependent on license availability. Even users who would only *use* the tools, for any other need than software development, would need a ClearCase license once the tools are stored in a vob. Additional cost may understandably become a concern.

There are other limitations to what may be versioned. For instance, some resources are inherently unique, thus can only be found in one version, for example, sockets, but also license and other databases. One may also think of hashes of tools, for commands such as `locate` or `man`.

Finally, there may be a performance penalty, an impact on storage requirements, on access control, on maintenance tasks, and so on. All these possible issues must be studied and compared with the benefits described before. The best tradeoff will heavily depend on the environment.

Shared libraries

Shared libraries are a special case: some of them will be used by tools needed to set up the environment for running ClearCase. In fact, cleartool itself uses `libC.so` (Linux/Solaris extension). Since they are shared, it is enough that one of their clients needs to be available outside of a ClearCase context, so that the shared libraries must be used from standard storage.

An interesting aspect of shared libraries is their versioning. The dynamic loader implements a crude version control system, based on names recorded at link time in client applications. In order to allow applications to link with the latest (that is current) version of the shared libraries, without relying on the developers having to be aware of it, a stable name is maintained as a symbolic link to the version-decorated name of the actual file:

```
libfoo.so -> libfoo.so.3
```

As it happens, it is often older versions of shared libraries that are in use in operating system utilities: older at least than the versions that are offered for linking to the developers. This means that one may often maintain the recent versions under ClearCase, and leave older versions in the system directories. Or rather, that there is no conflict in installing the latest patches in the vobs, without upgrading the hosts themselves.

One digression while dealing with shared library versions is that whether in the standard scheme it is the symbolic link which bears the stable name, and the actual file which bears the version decorated one, this is actually not required either at link time or at run time. It may thus be more convenient in the vobs to maintain the name pairs in the other way around, at least when developing shared libraries: producing new versions of the shared libraries under a stable name, and creating, as needed version decorated symbolic links to them. The name recorded in client applications must only be the version decorated one:

```
$ ld bar.o -G -o libbar.so -h libbar.so.1
$ ln -s libbar.so libbar.so.1
$ gcc -o foo foo.o -L. -lbar
$ export LD_LIBRARY_PATH=/vob/test/libso
$ ldd foo1 | grep libbar
 libbar.so.1 => /vob/test/libso/libbar.so.1
$ ls -l libbar.so.1
lrwxrwxrwx 1 marc jgroup 10 Aug 21 19:10 libbar.so.1 -> libbar.so
```

It may be worth stopping for a while on this issue of shared libraries, and considering it in a wider perspective: what this means is that the run-time environment is already able (in a limited way) to cope with multiple versions of the same resources, that is, it doesn't completely fulfill the basic assumption allowing one to consider it as one (consistent) software configuration: the exclusion principle, *at most one version of any resource*. Or in other terms, there is not one monolithic and consistent software configuration, but an overlapping collection thereof. Let's note that this is only a tendency taken to a much wider scope with for example, the Eclipse repository. We shall mention this again in *Chapter 12, Challenges*.

A different strategy to maintain shared libraries is to deliberately break the overlapping between the development and the run-time environment, thus abandoning the idea of *sharing* them: maintain a copy of the shared libraries in the vob, instructing the linker to write the standard path (instead of the one actually used at link time, pointing within the vob) into the executables. This means that the result of builds may not be usable in the run-time environment, which one may consider as an extension of a concept of cross-compilation.

Licenses

Some tools have host-based licenses, so that sharing them in vobs will not work. It is sometimes possible to store the license keys for every host in a standard path, and to create a symbolic link to it from the vob.

The tool will be usable only on host for which the link to the license key will not be dangling.

MultiSite and binary elements

The previous discussion about licenses applies to more license schemes (not only host-locked) in presence of MultiSite, but the treatment remains similar.

MultiSite brings some other concerns, which would be common with staging (if practiced), and relate to storing and shipping of large binaries.

The mere size of binaries, together with their "opacity", fight continuity: every version takes a large storage, events concerning them are large and cannot be split (or only under certain conditions), resulting in large sync packets that take a long time to ship, require large buffers, and result in frequent errors.

The only common option is compression, but this is a two-edged sword, as time is needed to compress and decompress and it often requires yet larger buffers (and results in duplication at the container level (refer to *Chapter 9*, *Secondary Metadata*, about the storage of binary types).

In short, ClearCase can version binaries, but doesn't do miracles: tradeoffs come near. In particular, there is often very little value in storing packages or zipped archives in vobs.

Labels, config specs, and multiple platforms

We already mentioned using (site) local labels in the context of MultiSite, and understand now that handling binaries makes it even more necessary for tools than for source code. Let's add that (remote) labels crossing vob boundaries (either *global*, or just sharing the same name and thus the same rules in config specs) are even more in danger of facing transient inconsistencies.

We also saw that tools are often to be dealt in large baselines of consistent (or at least compatible) releases. It is often convenient to name these baselines with compound label types, with the effect that these types suffer from combined volatility: they change as soon as any tool changes, which is sometimes not predictable (it depends on defects and fixes by other organisations).

This is a typical case for *floating* labels (for developers' convenience, and minimizing errors). It is however important to retain underneath fixed labels for the individual tools.

Both the floating and the fixed labels should be maintained for every platform. As seen previously (*Referential Transparency*) one ought to share as much as possible between the various platforms, under the same paths, and avoid duplicates and *evil twins*. Tools are at the bottom of the development chain, and thus in a strategic position: errors made at that level will easily multiply and result in combinatorial explosions, missed opportunities for sharing results, and duplicated work.

Special cases: Java 1.4.2_05 on Linux

Exceptionally, it is possible that some tools cannot be used under ClearCase.

We know only the case of Java 1.4.2_05 and above versions on Linux. In November 2004, Sun made a change to the Java launcher, exclusively on the Linux platform (although the same thing could have been made, for example, on Solaris), and which broke among other things (for example the use of `LD_LIBRARY_PATH`), launching java from a vob. The change required that the `lib` directory containing the shared libraries used by the binaries would be co-located to the directory tree returned from the value of `/proc/self/exe`. This was an arbitrary assumption, defeated by the fact that in the context of ClearCase, the path returned would point inside the cleartext pool.

Working around this was allegedly possible by implementing an interposer library to fake a failure of the `readlink` system call, thus falling back to another mechanism. This could affect other tools and contexts.

In practice, the change made it impossible to use Java from a vob on Linux.

This situation prevailed for at least a few years. We are unclear about the ***current status***. Maybe the IBM JRE, bundled with every Rational product, could offer a solution.

Naming issues: acquisitions, splits, mergers

Our last remark will be to warn against hardcoding in the installation paths names beyond one's control, and which are not strictly necessary.

Mentions to companies in particular are now-a-days particularly volatile.

Let's remember, for example, that ClearCase was not always a product of IBM: one might have had to rename occurrences of *Rational*, after *Pure*, after *Atria*.

Summary

We hope the long list of concerns we reviewed in our second part did not overshadow the bright perspectives opened in the first one!

Tool maintenance is a good example of the *no free lunch* reality prevailing in software development. The issues have to be addressed, and the sooner they are, the better: these are investments that pay back! Once the base environment is in place, upgrades will soon feel like routine. The comfort of being able to switch back and forth with only config spec changes between different releases of a tool will feel invaluable: was a bug fixed? did the performance change? Is this new tool compatible with the rest of the portfolio or where to trace the next baseline? These are questions that are much better answered, and with incomparably more confidence, by actual testing than by reviewing release notes!

In the next three chapters, we shall continue to fiddle with administration issues, never losing the user's perspective: what's behind the scenes, and what should there be, so that my needs are fulfilled in the best possible way? What can I, and what should I, do to make it work even better?

9
Secondary Metadata

We have seen two kinds of ClearCase objects:

- *mvfs* objects (elements and derived objects—let's say *files*)
- Metadata (label and branch types and instances)

ClearCase database are clearly structured (indexed) towards designing the first ones as the *first class citizens*.

This can be felt with the way in which queries about the former are faster than (non-trivial) queries concerning the latter.

We may notice that in recent releases of ClearCase, the `find` command has been extended to support searching for arbitrary objects (not only files), and thus also for metadata and types. So far in our experience, the efficiency for the latter has been the same as this of `lstype`, and the expressive power of the supported queries is still restricted. But who knows? This may change.

It is time now to turn our investigations onto some details of the primary metadata met already (labels and branches: see Chapter 6), and some less essential, yet useful kinds of metadata, which can be attached to both of the former kinds, as well as to most vob objects (but not to derived objects).

We'll find consistency of design, and the advantages of having type objects, suitable for maintaining information shared by many instances.

This chapter will thus cover:

- Secondary metadata types: attributes and hyperlinks, but also comments and triggers
- ClearCase scrubbing mechanisms concerning metadata
- User-defined types and custom type managers
- Native type managers, especially `text_file`.

Triggers

Let's first deal with triggers. They are not really metadata, but since there is a `trtype`, may be dealt with as such in some contexts.

Triggers are very popular. You'll find a lot of them from numerous web sites, and this in itself would be enough of a justification for us to leave them alone.

We'd like however to argument *against* using triggers—almost at all. Triggers are one of those slippery slopes on which newbie administrators jump joyfully, and which end up spoiling the joy of ClearCase for users, thus defeating any reasonable purpose. It is all too easy to write (bad) triggers, and all too difficult to get rid of them. The last point is based on experience: whereas it is easy to list the existing triggers (`ct lstype -obs -kind trtype,"-obs"` not to miss the ones which would have been removed), there is no way to know who depends on them, and in particular what scripts might break if you remove them. In practice, triggers are seldom removed: only when a bug can be found to have a bad impact. And seldom are they re-examined to check whether they still fit their original purpose (more on this in the further sections).

Triggers are vob local: in a trigger environment, things happen differently in different vobs.

They are not replicated (thank God!).

If they don't break on Windows, they'll break in snapshot views, or in the remote client, or via Eclipse, or with clearfsimport, or using cygwin, and the list is not exhaustive.

In order to write "portable" triggers, one needs a version of Perl that would satisfy some annoying and artificial constraints—be available and consistent in all contexts, be independent from the user view—and one starts using the one bundled with the ClearCase installation, which is limited and volatile (may change at the next upgrade), and... not meant for that! This clearly competes against maintaining Perl in a vob (see *Chapter 8*, *Tools Maintenance*).

In short, triggers are surprising. SCM is about making development manageable by users, making it easy for them to understand what happens and why. Sharing their experience and saving their efforts. Triggers are deceitful as a promise of a free lunch: they are hard to get right, hard to debug, hard to get coherent across the many contexts in which users may find themselves, and hard to keep right in the context of other triggers (in what order are they called?). And triggers are also unfair: users cannot practice creating ones, cannot usually skip them, or fix them.

There are two kinds of administrators; the skillful ones don't write triggers. Existing triggers have unfortunately been written by the others.

Maybe a few examples might help illustrating this. Let's pick them from a popular page in the IBM web site: ***The ten best triggers***.

NO_RMELEM

This is number one. Two scripts for UNIX and Windows, using the version of Perl bundled with ClearCase, respectively *Perl* and *ccperl*, and the same inline code: unconditional `exit 1`. The first comment is of course to question the need for this trigger. As we saw earlier, `rmelem` is a well-protected command. As a normal user, you won't be allowed to remove an element you do not own, which has any version not created by you, or which bears labels or hyperlinks. So, no real benefit.

What are its costs? This is clear: it prevents a user who notices he just created a wrong element (typical scenario: an evil twin of an existing element either in a different branch or a different directory) from removing it immediately before somebody else has any chance of accessing it by mistake.

The good point is that it is easy to work around:

```
$ PATH=~/bin:$PATH
$ cat <<eot > ~/bin/Perl
echo Skipped \$*
eot
$ chmod +x ~/bin/Perl
$ ct rmelem -f aaa
Skipped exit 1
Removed element "aaa".
```

Please note that we are not being facetious here when we write that this is a good point. Let us also trust the author that this is intentional! He could have put the full path to Perl (which would fail `rmelem` in a different way on Cygwin though). If he did not do it, it was precisely in order to allow knowledgeable users in a real need to work around the trigger. There remain several questions:

- Why wasn't this documented? Our belief is that it is because this is "security by obscurity". Triggers belong to the realm of naive security.

- How can experts such as this script author make such objective mistakes as to recommend such a trigger? Again, we can conjecture that he doesn't know. Two possible scenarios might explain why he wouldn't know:
    - He has been using this trigger for so long that he is now blind to the real behavior of ClearCase (`rmelem` was not always as secure as it is now): remember that in presence of triggers, you are not dealing with ClearCase anymore...
    - He only ever uses a vob owner account, and has thus never experienced the normal user situation. This would be a harsh criticism, if we wouldn't know some adverse environments in which tools such as sudo (which allows one to change uid just for running the few commands requiring super user rights) are banned "for security reasons".

CHECK_COMMENT

Again coming in platform-specific guises and using pathless bundled Perl versions, but with a more sophisticated behavior than previously, requiring a script with a full path.

As previously, let's first consider the intention: gently blame the author of a checkin for not feeding a comment. Let's go to the costs. There are several:

- Prompting the user interactively for every checkin without a comment, starting a GUI (unless on UNIX the `DISPLAY` variable is unset, in which case the interactive prompting is textual in the shell). This is a small penalty, unless one is performing a mass-checkin. In this case, the best is to abort (killing the process) and start again with a dummy comment; hopefully not at the end of a session which would perform a cleanup for a long task.
- Pushing people to write those comments. This is probably our worst critique: forcing users to write comments is counter-productive! It only ends up in ruining any value the comments might have, leading to the fact that nobody will ever bother to read them!

We may now share some comments on the implementation.
First, the installation was meant to be performed on Windows only; if one does it on UNIX, the use of double quotes leads the shell to interpret the double backslash (`-execwin "ccperl \\mw-ddiebolt\triggers\check_comment.bat"`) as a single one, explicitly escaped, which results in a wrong path and a failure of the trigger on Windows (fortunately allowing the checkin):

```
Can't open perl script "\filer\views\marc\check_comment": No such file ##
                                                                or directory
cleartool: Warning: Trigger script for "CHECK_COMMENT" returned failed ##
                                                                 exit status
```

Next, the use of two distinct files (a Perl script disguised as a Windows batch file, and a small wrapper with a `.pl` extension and invoking the former from UNIX, which is not shown on the IBM web page) is a common practice in the ClearCase distribution. It is completely unnecessary once one invokes Perl explicitly! Only one common and simpler file is thus needed.

Finally, the script is now shared from a filer, automounted as `/net/titeuf` on UNIX and known to Windows as `\\mw-ddiebolt`. In any case, this makes the trigger depend on network connectivity, which is a dangerous dependency for a command as useful as `checkin`. Fortunately again, failing the trigger does not fail the checkin. Summary on this trigger? No real benefit: 100% cost. With an implementation which was never reviewed critically!

REMOVE_EMPTY_BRANCH

Let's look at the third case in this list, which is again gradually more interesting (it uses environment variables).

First as previously, the intention. We may surprise you, but it is indeed often a good idea to remove empty branches! These would keep matching in the config spec after the user had forgotten about them, preventing her from being informed of new deliveries.

Now, straight to the implementation. This is only a Windows trigger. Do not set it in an inter-operating environment as it will prevent UNIX users from unchecking out anything!

This is the comment as with the previous trigger about the useless batch preamble. The interesting aspect in this code is the `. . .`comments, and by this we mean the commented `printf` commands. What they tell us is that triggers are hard to debug, even if it is possible to to set the `CLEARCASE_TRACE_TRIGGERS` environment variable, or to invoke ccperl with the `-d` flag starting the debugger.

Let's uncomment those print statements, and on the contrary, comment the actual command away. Now, let's create a subbranch, checkout, and uncheckout:

```
$ ct lsvtree .@@/main/mg
.@@/main/mg
.@@/main/mg/0
.@@/main/mg/aa
.@@/main/mg/aa/0
$ ct co -nc .
Checked out "." from version "/main/mg/0".

$ ct unco -rm .

Trigger is fired \main\mg\0...

\main\mgXXXXXXXXXX Count : 10

Only version 0

\main\mg\0 Count : 0

ToDo : cleartool rmbranch -force \\view\marc\test\.@@\main\mg
cleartool: Warning: This uncheckout left only version zero on branch ####
                                                                  "\main\mg".
Checkout cancelled for ".".
```

There would have gone the `aa` sub-branch. We wondered what the code was actually counting: the number of `X` it had itself inserted and removed twice, but only to the version extended name of the argument version, not to anything below it in a possible version tree…

It is not the first such trigger we meet which is actually public, old, and dangerous.

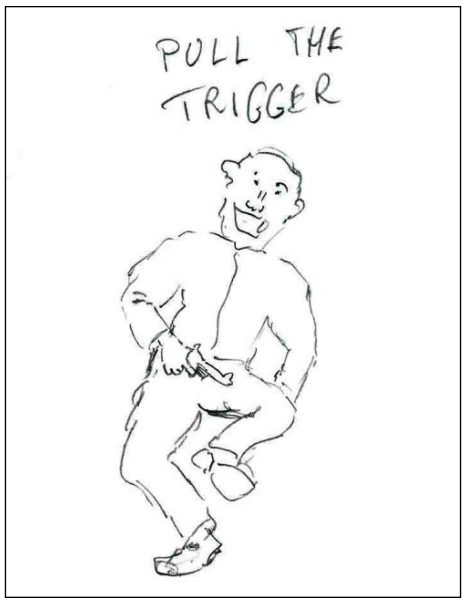
The last word on this case is that removing the branches should also be done in conjunction to `rmver`, and that doing it with a trigger is actually using the wrong weapon: this would force you to install your same trigger in all vobs. The same idea, with better code, should be implemented as a **cleartool wrapper**, only once.

In afterthought, these triggers (well, the first two) seem to us highly *political*. They are milder than many triggers in actual use. Triggers is such a touchy issue that IBM experts cannot openly write what we write here (and this is why we have to write it!): avoid them!

This page was first published in 2004, and last edited in 2006. It was labeled as *Introductory*, which it indeed is. It had been accessed more than 20000 times when we read it, and had been rated four out of five rated by 116 people.

Its title is undoubtedly misleading. It is a pleasant introduction to triggers, showing scenarios in a pedagogic order, certainly not a list of best examples. But it does show a lot.

One last point on the evil of triggers which didn't come up from the three examples reviewed, but would from another popular trigger (redundant in its purpose *with* `NO_RMELEM`): a trigger which changes the owner of new element to the vob owner. Besides sharing with `NO_RMELEM` a misguided intention, it displays a performance impact: cleartool will be invoked on any new element. This is enough to defeat any attempt at optimizing mass creation.



Comments

Next kind of exceptional metadata: comments. Comments are the lowest common denominator. The bottom of the bin. They are untyped. You cannot really search them explicitly (for example using the `ct find` command)... Well, here is what you can do:

```
$ ct lstype -kind lbtype -fmt '%n: %Nc\n' | grep foo
```

This searches for label types with a comment matching `foo`. The comment is displayed using the `-fmt` option, with the `%c` pattern, modified with `N`.

Only, this will not work well on multiline comments; the `N` modifier is only there for the last newline, allowing us to ensure that we produce at least one line per type, even if there is no comment. In case the match is on the second line, the type name will be filtered away.

OK, one can do better. Here is how we can handle the multiline comments properly, thanks to the ***ClearCase::Argv*** Perl module:

```
$ perl -MClearCase::Argv -e \
'$c=new ClearCase::Argv({ipc=>1,autochomp=>1});
for my$t($c->lstype([qw(-kind lbtype -s)])->qx){
  $t=~s/ \(locked\)$//;
  print "$t\n"
    if grep /foo/, $c->des([qw(-fmt %c)],"lbtype:$t")->qx}'
```

This is acceptable? Good.

`ipc=>1` ensures there is a single background cleartool invocation for all the types. `autochomp=>1` strips ending newlines from the output of cleartool commands. The `des` function produces a list, which is tested by the `grep` one. As an empty list is a valid list anyway, we may skip the `N` modifier in the format string.

Well, our point was anyway that the situation gets much better if we needn't parse multiline output and tell things apart that may or may not relate with each other. So, comments are useful only if used sparingly, and as a last, informal resort.

One corollary is that if you have nothing special to say, the most efficient way to do it is by keeping silent, that is, avoid polluting the comment space with *forced* comments (OK: triggers was the previous paragraph).

Scrubbers

Don't be impatient: we're getting closer.

Our next digression on the way to metadata is on the issue of scrubbers. You noticed the plural? Yes, there are two, typically scheduled as tasks: `scrubber`, and `vob_scrubber`.

We shall be concerned here with the latter, so let's talk of the former first.

The *scrubber* (already mentioned in the *Removing derived objects* section of *Chapter 3, Build Auditing, and Avoidance*) takes care of maintaining the pools within reasonable disk space limits. Actually only two of the three pools: the *cleartext* and the *do* pools. Pools are directory trees in the vob storage outside the database that contain the actual data. The third is the source pool which stores versions of file elements. We'll review the most interesting case of source containers—the case of text files—in the end of this chapter. We'll keep on with pools in the next chapter. Please note now

that we started this chapter by mentioning two kinds of ClearCase objects: files and metadata. Files are actually produced, and thus scrubbed—not only derived objects, but versions as well (the *cleartext containers*). See more on the scrubber in *Chapter 10, Administrative Concerns*.

The vob scrubber deals with database items: events and oplogs. We'll be back to oplogs in *Chapter 11, MultiSite Administration*.

Note how events are classified in the way the vob scrubber deals with them (not uniformly).

For example, `mkattr` events are kept (by default) only for seven days, and the last one for an attribute type for 30 days, as reads from the `/var/adm/rational/clearcase/config/vob_scrubber_params` file:

```
    event mkattr  -keep_all 7 -keep_last 30
```

You can always check the log for the last runs, at least using perl:

```
$ ct des -fmt "%[replica_host]p\n" \
 replica:$(ct des -fmt «%[replica_name]p\n» vob:.)
beyond.lookingglass.uk
$ ct lsvob /vob/foo
* /vob/foo /vobstg/foo.vbs public (replicated)
$ ct getlog -host beyond.lookingglass.uk -full vob_scrubber | \
 perl -n00e 'print «$p$_» if $p and m%/foo%; $p=/^Number/?$_:q()'
Number of  Events      Events      Events      Kind of event
objects    before      deleted     after       by meta-type
----------  ----------  ----------  ---------- -----------
     1        162         79          83  versioned object base
     3        209         58         151  replica
     3          5          0           5  pool
     1          1          0           1  replica type
    13         13          0          13  element type
    76         78          0          78  branch type
  2545       7648       1138       6510  label type
    27         27          0          27  attribute type
    27         27          0          27  hyperlink type
  3376       3376          0       3376  directory element
  3623       3623          0       3623  file element
 18142      18147          3      18144  branch
 13959      88599      26403      62196  directory version
 17873      97310      27923      69387  version
     4          4          0          4  symbolic link
 95625          0          0          0  derived object
  3199        628        178        450  hyperlink
158497     219857      55782     164075  total in VOB
Event scrubbing done.
Oplog scrubbing done.
Finished VOB «/vobstg/foo.vbs» at 2010-08-29T07:14:10+01.
```

The `vob_scrubber` log is not formatted, which means that it is not trivial to know what exactly to wait for: in our example we are only interested in `/vob/foo`. Even if it is rotated relatively often, the log is still pretty large. We read it in paragraph mode (the `-n00` flag), and print when finding a suitable match, both the previous and the current paragraphs. This implies that we must remember the previous paragraph, and print it in case it was of the right kind. This is roughly what the small inline script does.

This processing allows us to understand the presence or absence of certain events in the `lshistory` output. In this case, we could verify that no `mkattr` events had been scrubbed for the last period. We knew the attribute was set recently to a label type (which was confirmed by the presence of the last `mkattr` events), but the question was to know whether it had possibly had another (incorrect) value previously. Now, we could show that no such event had been scrubbed.

If the events you'd like to inspect have been scrubbed, there is still one recourse, at least in replicated vobs: `dumpoplog`. This is a useful yet under-documented command, which produces, slowly, a very verbose output. You need again to be prepared to post-process it, in a flexible way. Here is one example, run within a vob, with a view set:

```
$ mt dumpoplog -l -vreplica wonderland -name -since 7-Aug | \
  perl -n00e 'print "$1 $2\n" if /^op= mklabel.*^op_time= ([\
w:-]+).*^obj_oid= .*?\(.*?: (.*?)\).*^lbtype_oid= .*?\(FOO\)/sm'
2010-08-09T10:57:55Z /vob/foo/.@@/main/1/demo.txt/main/1
2010-08-09T10:57:55Z /vob/foo/.@@/main/1
2010-08-09T10:59:01Z /vob/foo/bar@@/main/1
...
```

We use again the paragraph mode, and now we parse the result in multiline mode (`m` modifier) where the "`.`" wildcard may match newline characters, allowing nevertheless the beginning and end markers, respectively "`^`" and "`$`", to match at line boundaries (`s` modifier). In the regular expression, we successively match word constituents (`\w`) extended to colons and dashes (in order to match time stamps), and sequences of arbitrary characters in non-greedy mode ("`.*?`").

The preceding example shows which versions have been labeled with `FOO` labels, and this possibly after the `mklabel` events have been scrubbed.

Now, oplogs may be scrubbed too. It is even both recommended and recommendable to scrub them. This just doesn't happen by default, and thus requires editing the host global param file `vob_scrubber_params` mentioned earlier (probably the best way), or a per vob setting, a file with the same name placed in the vob storage area. The setting is to keep oplogs for a certain time, and the recommendation is to keep at least two months of oplogs to be able to reproduce sync packets that would have been lost before being imported on remote sites.

Attributes

Attributes are the preferred alternative to comments. They are preferred because their **typing** allows to avoid depending on parsing informal text. This typing involves two levels:

- A low-level typing determining the format of the values and a set of operators of the query language applying to them
- A declarative level, reserving a name which formally identifies the type

Attributes may be attached to different vob objects: file or type objects.

The existence of options restricting their application to elements, per version or per branch, may bias their use toward the first kind (file, rather than type objects). Attributes are also often used to store status information—values that may vary. These are uses for which the `-shared` option (in a MultiSite context) would be as questionable as for labels or branches.

We tend to apply attributes mostly to objects of the second kind, almost only label types; and with stable values, so that the `-shared` option is safe and justified. Attributes allow to attach information at the right level of stability in a structure of inter-related types, thus offering significant flexibility. The same attributes may be used on different sites, in conjunction to locally mastered and applied labels:

```
$ ct mkattype -nc -shared AAA
Created attribute type "AAA".

$ ct mkattr AAA '"aaaa"' lbtype:ZOO
Created attribute "AAA" on "ZOO"
```

The requirement to enclose string values within double-quotes may surprise, and leads at times to errors. It doesn't concern the other value types.

We'll take an example of applying (unshared) attributes to versions, and it is from the ***ClearCase::Wrapper::MGi*** Perl module. We use attributes there to record the fact that a label of a certain type hierarchy was removed, and from which exact version on. The background is that for every label family, we maintain a chain of incremental fixed label types, recording the successive positions of a common floating label type. Suppose our floating type is named `FOO`, then the incremental types, applied at every stage to the concerned versions only (therefore offering better performance and manageability), are named `FOO_1.00`, `FOO_1.01`, ..., `FOO_2.34`, and so on.

First, we create a family label type, which is identified to a floating label (FOO). Its first increment, a fixed label FOO_1.00, is created automatically (note that ct is aliased as usual to the standard cleartool, and ctx is aliased here to the cleartool wrapper, actually from ***ClearCase::Wrapper***):

```
$ alias ctx ct
alias ctx='/usr/bin/cleartool.plx'
alias ct='/opt/rational/clearcase/bin/cleartool'
```

One may also set the expanded_aliases shell option, or prefer to define functions instead of aliases:

```
$ function ctx {
  /usr/bin/cleartool.plx $@
}

$ ctx mklbtype -nc -fam FOO
Created label type "FOO_1.00".
Created label type "FOO".
Created attribute type "RmFOO".
```

We can create the successive increments with the mklbtype -inc command:

```
$ ctx mklbtype -nc -inc FOO
Created label type "FOO_1.01".
```

We use a two-level numbering scheme to guarantee that the types may be sorted naturally, and to leave some space for growing the size of the namespace if need-be (for example, jumping to FOO_2.000 after FOO_1.99). This strategy seems to be enough to allow using only the floating label to name the baseline in development config specs, and to produce afterwards a list of config spec rules equivalent to any situation in the past, for example:

```
element * FOO_1.23
element * FOO_1.22
…
element * FOO_1.00
```

However, this holds only as long as the FOO label was not removed from a version at any point. To support this case, we set an attribute when we remove a label (as well as to previous versions bearing a label of the family). In our example, the attribute type would be RmFOO (it must be specific to the label type), and the value records the increment at which FOO was removed. The version will of course still bear the fixed label corresponding to the increment at which FOO was applied, so that it is correctly considered part of any equivalent config spec corresponding to an increment in the range until this at which FOO was removed.

```
$ ct des -fmt "%Nl\n" foo.txt
FOO FOO_1.14
$ ct des -ahl -all lbtype:FOO
FOO
 Hyperlinks:
   EqInc -> lbtype:FOO_1.23@/vob/foo
$ ctx rmlabel FOO foo.txt
Removed label "FOO" from "foo.txt" version "/main/mg-013/18".
Created attribute "RmFOO" on "foo.txt@@/main/mg-013/18".
$ ct des -fmt "%Nl\n" foo.txt
FOO_1.14
$ ct des -aattr -all foo.txt
foo.txt@@/main/mg-013/18
 Attributes:
    RmFOO = 1.23
```

Every config spec rule produced (in the list above) must thus be corrected as follows (here for increment `1.15`):

```
element * "{lbtype(FOO_1.15)&&!attr_sub(RmFOO,<=,1.23)}"
```

The main problem with attributes attached to types, is that queries are inherently inefficient, and that their performance is tied to the number of types, and thus decreases as the size of the system grows, which is clearly an undesirable property. The same is of course true of queries for file objects (using the `find` command), but these ones are better supported by the indexing strategy of the ClearCase databases.

To face this problem, one needs to revert the strategy from *query* to *navigation*, and to rely for this upon a new structure, based on hyperlinks.

Hyperlinks

We already know of two dimensions, or hyperspaces, in which to structure software configurations: the most trivial, and least interesting one is directory trees; the most interesting one is audited dependencies. Hyperlinks provide a third layer (or rather, dimension). What makes directories less interesting is that they record the contingent history of the development, and thus lose a real value of relevance: they tell you about your background (which has its value, for sure), not about your actual life situation. The audited dependencies are real stuff; furthermore, they are collected for free. Sometimes, they lack flexibility: there is a lot of interesting stuff beyond plain reality.

Hyperlinks have this flexibility. For a start, they may relate other objects than elements: label types for instance.

We already mentioned in the previous chapters the hyperlink mechanisms ClearCase uses for merging (`Merge` hyperlinks in *Chapter 7*, *Merging*) and global typing (`GlobalDefinition` hyperlinks in *Chapter 5*, *MultiSite Concerns*).

We use them quite extensively to build label type families, with a floating label type representing the current baseline (and applied to all the elements present in a configuration), and fixed types recording the successive increments of the development history, and only used indirectly to recreate an equivalent config spec pointing to a past configuration.

We described this earlier in the *Attributes* paragraph: the successive increments are linked together with hyperlinks of `PrevInc` type (defined in ***ClearCase::Wrapper::MGi***). The top of the linked list, quite a volatile information, is accessible via the floating label type (a well-known and stable name, even if the set of versions it is applied to changes frequently) with an `EqInc` hyperlink. Both kinds of hyperlinks are maintained by the `mklbtype` function of our cleartool wrapper.

The following transcript illustrates the relationships explained above between the floating label `FOO` and its increments (the fixed labels of the `FOO` family, such as `FOO_1.22, FOO_1.23` and `FOO_1.24`).

The family type floating label `FOO` is linked to the current increment `FOO_1.23`:

```
$ ct des -ahl -all lbtype:FOO
FOO
 Hyperlinks:
   EqInc -> lbtype:FOO_1.23@/vob/foo
```

This is how the successive incremental label types are linked together:

```
$ ct des -ahl -all lbtype:FOO_1.23
FOO_1.23
 Hyperlinks:
    PrevInc -> lbtype:FOO_1.22@/vob/foo
    EqInc <- lbtype:FOO@/vob/foo
```

Now we add the next increment:

```
$ ct mklbtype -nc -inc FOO
Created label type "FOO_1.24".
```

And see how the label hierarchy is re-arranged:

```
$ ct des -ahl -all lbtype:FOO lbtype:FOO_1.24
FOO
 Hyperlinks:
   EqInc -> lbtype:FOO_1.24@/vob/foo
FOO_1.24
 Hyperlinks:
    PrevInc -> lbtype:FOO_1.23@/vob/foo
    EqInc <- lbtype:FOO@/vob/foo
```

This is the point for a syntactical note: there is a difference in support for describing hyperlinks and attributes. Or rather, the syntax we used in the above example may also be used for attributes:

```
$ ct des -aattr -all lbtype:ZOO
ZOO
 Attributes:
   AAA = "aaaa"
```

...but there is another handy syntax for attributes, which has no equivalent for hyperlinks:

```
$ ct des -fmt '%[AAA]NSa\n' lbtype:ZOO
"aaaa"
```

This prints the value (`aaaa`) of the `AAA` attribute. As one can see, this syntax is flexible and allows to tailor one's needs so as not to require post-processing (except to get rid of the double-quotes).

We also use hyperlinks across vob boundaries, and there mostly ones of the predefined `GlobalDefinition` type, even if we do not set an admin vob for this purpose (see Chapter 5).

The main problem with the concept of admin vob is that there is only one `AdminVOB` hyperlink type (which is used to link vob objects between one another), and its effect is global. This therefore constrains other sites in an excessively inflexible way, which ends up defeating collaboration. Client vobs cannot really be used without their admin vob, and importing vobs from different sites, which wouldn't have been submitted to a centralized control from the beginning, invariably results in conflicting requirements to use incompatible admin vob hierarchies.

Fortunately, `GlobalDefinition` hyperlinks may be created explicitly, by our wrapper or other scripts, and existing instances are obeyed to functions such as `lock`, `rename`, and so on without having to submit to the constraints of admin vobs.

A final note on hyperlinks is that one may use textual hyperlinks as an alternative to attributes. These hyperlinks do not reference any other object but instead a new text value. This may be useful if one needs to create an object to hold the value, for example, in order to be able to transfer its mastership, or if one wants to create several instances of the same type "attached" to the same object.

Type managers and element types

ClearCase comes with a good few native types, but also allows the users to define their own. At least, this was the intention, and since the very beginning—one sees there that ClearCase was a product of the flamboyant times of object orientation, and carried ideas experimented in *Apollo Domain* typed file system.

The list of native types found depends on the vob feature level, and it has increased over time. Notable additions had to be made to support file formats that departed from the implicit assumption that text should break on new lines (so as to be generically printable, without the need for a parser, aware of the specific syntax). This concerned the HTML and XML files produced by some popular tools, mostly on Windows. More recent additions concern the variants of UTF, to support Unicode in different contexts.

The magic files

One may use element types *explicitly*, with an `-elt` option to `mkelem`, or with the `chtype` command, but the stereotypical way is to use them *implicitly*, via the **magic file** mechanism, in the old UNIX tradition.

A `default.magic` file is provided by the ClearCase installation, with rules to take into use all the existing native types... and more. The syntax of magic files is documented in the `cc.magic` man page.

Here is a sample extract from it:

```
$ cat /opt/rational/clearcase/config/magic/default.magic
# Check stat type
directory : -stat d ;
block_device : -stat b ;
char_device : -stat c ;
socket : -stat s ;
fifo : -stat f ;

# Match by name without examining data
program compressed_file : -name "*.[eE][xX][eE]" | -name "*.bin" ;
object_module compressed_file : -name "*.[oO][bB][jJ]" ;
shlib library compressed_file : -name "*.[dD][lL][lL]" ;
```

```
zip_archive archive file : -name "*.[zZ][iI][pP]" ;
tar_archive archive compressed_file : -name "*.tar" ;
…
# assumed to be text
java_source source text_file : -name "*.[jJ][aA][vV][aA]" ;
...
# catch-all, if nothing else matches
compressed_file : -name "*" ;
```

It is made of rules, with a first match logic similar to this of config specs. Each rule consists in two parts: an ordered list of types and a set of selection criteria. When a selection criterion matches, the types are tried in order, and the first one found in the current vob is used; if none of them is found, an error is spit. If no selection criterion matches, an error is also spit, but `default.magic` has a catch-all rule to preempt this case and use then `compressed_file`.

Let us note while we are here that `default.magic` anticipates the creation of some user defined types, such as `java_source`. We shall fulfill this precise anticipation in the next section.

ClearCase design does not intend the users to modify `default.magic`. On the contrary, it suggests that they would create new magic files in other directories, and take them into use via a `MAGIC_PATH` environment variable (a local `.magic` file in every user's home directory will also be searched by default, and this one might of course be a symbolic link to a site-wise shared copy). It is wise to restrict oneself to provide in local `.magic` files only specific rules and criteria to override some critical element types, and to keep the standard path to `default.magic`—with its catch-all rule—last in the `MAGIC_PATH`. This ensures that one doesn't shut oneself off from the benefits of upgrades from further ClearCase releases.

Remember that the catch-all rule will preempt any further rule if found too early. This might happen if the built-in path was not the last in `MAGIC_PATH`, or if magic files with names later than `default.magic` in sorting order would be placed there: they could never be reached. Even if any directory in `MAGIC_PATH` may contain an arbitrary number of magic files, it may be wise to keep those to a bare minimum, for fear that they might compete against each other, with the arbitration of the alphanumeric order of their respective names, which the `MAGIC_PATH` mechanism is precisely meant to avoid.

What may be recognized here is yet one special case of the fundamental challenge of SCM: how to manage sharing, how to propagate changes made locally, so that all benefit from them without surprises and unanticipated conflicts? ClearCase unfortunately falls short of offering a satisfying answer in the context of element types.

User defined types

All the types the user might define do not present the same challenges.

Type without a new manager

We mentioned the existence of a potential `java_source` type in the `default.magic` file. This is an insightful provision, as we'll see in *Chapter 12*, *Challenges*. Leaving until then the rationale for such a change, let's consider implementing this type now.

This is a good example of a trivial yet useful—nay: mandatory— type definition. The time stamp of java source files should be this of the last file saving: it ought to be preserved at check in, which is not the default behavior. The command is simple:

```
$ ct mkeltype -c 'Preserve time of java sources at checkin' \
 -super text_file -ptime java_source
```

Note that there is no mention of a `-shared` flag: element types are shared by default (they may be used at other replicas). There is a restriction related to this: they cannot be changed afterwards in replicated vobs (using the `-replace` flag of the `mkeltype` command will yield an error):

```
$ ct mkeltype -rep -super binary_delta_file java_source
cleartool: Error: Can't redefine element types when VOB is replicated.
cleartool: Error: Unable to replace definition of element type ##########
                                                          "java_source".
```

It may be a good idea to make the type `-global` (unless it is only a cosmetic issue: you decide). In any case, this type will have to be created (or maybe copied) into every vob in which it is needed.

However, it will be replicated properly, without any additional configuration, and will be directly usable in other replicas, both explicitly and implicitly.

New type manager

How about creating a more ambitious type—a type which would be associated with a new type manager?

We'll explore here only the most simple case, hence avoid writing code to implement some type manager methods in ways specific to a new type (which is covered in an IBM ***white paper*** by Laurent Maréchal). Examples of useful managers could be: managing tar (and other container) files, by actually extracting them and maintaining the resulting directory trees (with a program similar to ***synctree***); or managers providing compare and merge capabilities for file types requiring an ad-hoc graphical editor.

One manager we might want to implement would manage source files in such a way that new branches spawning from the root of the version tree would not result in a full copy of the base version in the source container.

Type managers are a collection of methods. In UNIX, these methods are grouped in directories under the `/opt/rational/clearcase/lib/mgrs` directory. Most of the entries there are actually symbolic links, with the help of which the actual code is shared among multiple types (including implementation of type inheritance). In Windows, all the methods are listed instead in a single `map` file (in `C:\Program Files\Rational\ClearCase\lib\mgrs`).

The most trivial implementation we are alluding to would be to create a (completely useless) symbolic link to an existing type manager, let's say `text_file_delta`. Let our symlink be named `foo`, and let's explore the implications of creating a new element type using it:

```
$ ct mkeltype -nc -super text_file -man foo foo_file
Created element type "foo_file".
$ ct co -nc .
$ ct mkelem -nc -elt foo_file foo
Created element "foo" (type "foo_file").
Created branch "mg" from "bar" version "/main/0".
Checked out "foo" from version "/main/mg/0".
$ ct ci -nc .
```

So far, so good (note that for this to work, you actually need to be using a local view). But what about editing this file in another view, hosted on another host? You'll get an error until you install the type manager on the second view server. On Windows, you'll have to do it using the map file. In any case, remember to take a backup of your changes, in order to be able to reapply them after every ClearCase upgrade; it is hard to predict which ones will overwrite the map file, or tamper the `mgrs` directory.

The same issue awaits you with MultiSite: a prior type manager installation will be needed on all vob servers to allow the export and import of sync packets containing events involving the custom type manager. ClearCase doesn't assist you with this, beyond giving a clear error message, which is unfortunately easy to miss, and to lose while rotating the synchronization logs.

Native types

ClearCase native element type managers differ mostly in the way they use source and cleartext containers: the former are used for storing the data, and the latter for presenting it (this of the selected version only) to the user or her tools. Do the two need to differ at all? Can multiple versions share a common source container? And what are the implications in terms of performance and storage?

Binary types

There are two basic native binary types: `file` and `compressed_file`. There is also one special type: `binary_delta_file`.

The `file` type uses the `whole_copy` type manager. Elements of this type actually have no *cleartext containers*: every version is stored integrally in a new source container.

Note from the following output that both container paths are identical, and from the source pool (`s/sdft`):

```
$ ct mkelem -nc -elt file file
$ mkfile 1k file
$ ct ci -nc file
$ ct dump file | grep cont=
source cont="/vob/foo.vbs/s/sdft/32/3e/####################################
                                    2-3774301fbe9311df913100156004455c-6g"
clrtxt cont="/vob/foo.vbs/s/sdft/32/3e/####################################
                                    2-3774301fbe9311df913100156004455c-6g"
$ ct co -nc file
$ ct ci -nc file
$ ct dump file | grep cont=
source cont="/vob/foo.vbs/s/sdft/32/3e/####################################
                                    2-52e43027be9311df913100156004455c-qq"
clrtxt cont="/vob/foo.vbs/s/sdft/32/3e/####################################
                                    2-52e43027be9311df913100156004455c-qq"
```

The `compressed_file` type uses `z_whole_copy`. The source containers are compressed, and thus uncompressed cleartext containers are generated on access:

```
$ ct mkelem -nc -elt compressed_file cfile
$ mkfile 1k cfile
$ ct ci -nc cfile
$ wc -l cfile
     0 cfile
$ ct dump cfile | grep cont=
source cont="/vob/foo.vbs/s/sdft/10/42/####################################
                                    3-1ec43037be9411df913100156004455c-1m"
clrtxt cont="/vob/foo.vbs/c/cdft/30/15/1ec43037be9411df913100156004455c"
$ ls -l /vob/foo.vbs/s/sdft/10/42/3-1ec43037be9411df913100156004455c-1m \
 /vob/foo.vbs/c/cdft/30/15/1ec43037be9411df913100156004455c
-r--r--r-- 1 vobadm adm 1024 Sep 12 18:30 ################################
               /vob/foo.vbs/c/cdft/30/15/1ec43037be9411df913100156004455c
-r--r--r-- 1 vobadm adm 36 Sep 12 18:28 #################################
         /vob/foo.vbs/s/sdft/10/42/3-1ec43037be9411df913100156004455c-1m
$ ct co -nc cfile
$ ct ci -nc cfile
$ ct dump cfile | grep cont=
```

```
source cont="/vob/foo.vbs/s/sdft/10/42/####################################
                                        3-8c643047be9411df913100156004455c-9n"
clrtxt cont="/vob/foo.vbs/c/cdft/20/2/8c643047be9411df913100156004455c"
```

One interesting conclusion is that compressing files will actually first increase the space consumption. Space will be saved after enough versions will have been created, assuming only few of them remain accessed (and the others' cleartext containers get scrubbed). The break-even will depend on the compression factor.

The last binary type (`binary_delta_file`) is actually not a binary type, but a text one, only lacking a predictable record delimiter. It is the supertype used by the various HTML and XML types mentioned earlier.

Text type

By far the most interesting element type, for purposes of version control, is `text-file`.

A basic understanding of the format of the source containers is an investment which may pay back in some cases of catastrophe recovery.

There is one single source container for all the branches and all the versions.

Creating a new version actually involves creating a new updated source container and removing the old one. The information in the container is however stable.

The container starts with a header, beginning with `^S` and ending in `^E`. A 2-digit id is assigned to branches `^B` and versions `^V` bottom-up (that is `/main` == `0 0` last).

Here is an example:

```
^S db1 4
^V 72bda4c9.fbe111dc.8c74.00:01:84:16:9f:47 4 1 47eb692e
^V 6c2da4c5.fbe111dc.8c74.00:01:84:16:9f:47 4 0 47eb692e
^B 6c2da4c1.fbe111dc.8c74.00:01:84:16:9f:47 4 2 1
^V 7c7075ce.fb1911dc.8b8a.00:15:60:04:45:5c 3 1 47ea1a4a
...
^B e71bad18.f20c11dc.8ef5.00:01:84:16:9f:47 0 0 0
^E e71bad14.f20c11dc.8ef5.00:01:84:16:9f:47
```

The first digit of the id identifies the branch. For branches, the second digit identifies the parent branch, for versions, the version id.

```
0 0 /main
1 0 /main/mg-001
2 1 /main/mg-001/mg-002
4 2 /main/mg-001/mg-002/mg50
3 1 /main/mg-001/mg
```

The last field on the line is a time stamp (*time2date*) for versions.

The header is followed with annotations of two kinds: insertions `^I` or deletions `^D`, with the id and the number of lines:

```
^I 1 1 613
package ClearCase::SyncTree;

^D 4 1 1
^D 3 1 1
^D 2 1 1
^D 1 7 1
^D 1 4 1
^D 1 2 1
$VERSION = '0.47';
^I 1 2 1
$VERSION = '0.48';
^I 1 4 1
...
```

Here is a possible recovery scenario: an administrator misunderstood a user request and removed a critical version (`rmver` or `rmbranch`).

A previous version of the source container is found in a filer snapshot. It is possible to identify the container by looking at the time stamps and grepping some older data.

The challenge: produce from there a cleartext container corresponding to the lost version(s), and check it back in again.

By comparing with the current source container, it is possible to identify the oids of the missing version and of the branch object.

```
^V 2594991b.490d11dd.9673.00:15:60:04:45:5c 17 2 486dfe83
...
^B bd588b72.237145da.8ecf.6a:2a:92:7a:ac:e1 17 15 1
```

In practice, one could interpret `17` as the branch, and thus `17 2` as version `2` on it, checking that this one had been removed.

So, from this container, one can run the `construct_version` method from the command line, and produce a `/tmp/clrtxt.out` file using the version oid found in the source container header:

```
$  srccon=/vob/foo.vbs/.snapshot/vobs_snapshot.1/s/sdft/1e/d/0-a14e41314e
7111dd925a00156004455c-qp
$ cd /opt/rational/clearcase/sun5/lib/mgrs/text_file_delta
$ ./construct_version $srccon /tmp/clrtxt.out \
 2594991b.490d11dd.9673.00:15:60:04:45:5c
```

Summary

This completes our review of metadata. Contrast the power and the beauty of element types, to the superficiality of triggers. Know to use attributes and hyperlinks, and avoid abusing comments.

The next chapter will continue with low-level concerns, close to the ClearCase implementation.

10
Administrative Concerns

Administration is too important to be left to administrators.

Some insight is necessary for end users as well, even if only to investigate and narrow down problems, and then make useful requests to administrators.

One may consider two approaches to **administration**:

- **Top-down proactive**, concerned with resource planning, installation, and monitoring
- **Bottom-up** reactive, driven by investigation and problem solving, and virtually involving anybody

The latter is often more challenging, as under error conditions the system does not exhibit the same level of coherence as under normal ones. We'll mention a few such cases. What is important is that there is always a balance of the two.

We have actually already dealt with quite a lot of administrative concerns, especially in the previous chapter:

- Vob scrubbing (but so far we have left pools scrubbing)
- Structure of source containers for text files
- Construction of cleartext containers
- Tradeoffs in compression (existence of cleartext or not)
- Use of `ct dump`

We shall push to the next chapter issues relating to MultiSite.

Finally, we must note that administration is a well-documented part of ClearCase, partly because of the ***Administrator's manual***, but also because of the profusion of technotes available from the ***IBM website***. The reason for such a profusion of information is clear: administration confronts the multitude of the interfaces with a huge variety of systems, as well as a variety of the situations. It is not reasonable for us to attempt to compete against this; we'll as usual attempt to avoid repeating things available already, but try to focus on useful insights left from existing literature and documentation.

What does this suggest to us as a driver for this chapter? Maybe:

- First a review of a few top-down topics:
    - License and registry monitoring
    - Multiple regions and their synchronization
    - Client monitoring
    - The location broker
    - Adding logs and scheduler jobs
    - Storage and backup concerns
    - Vob and pools size
    - Authentication
    - Importing data to the vobs
    - Duplicating and registering vobs
    - Configuring Apache to access vobs
- To leave space for a couple of bottom-up case analyses, without any attempt at exhaustiveness:
    - Albd account and its password
    - Changing types
    - Dbids and the underlying database
    - Protecting vobs
    - Cleaning lost+found

Top-down

The goal is not to squeeze away any top-down concerns: some basic understanding is needed to set up the scene and as a context to filling in some holes as we promised.

License and registry

As discussed in our presentation (Chapter 2), ClearCase servers are split both into functions and hosts. The former may be mapped onto the latter, so that a standalone installation is possible, where all functions, including the role of client, are played by one single host; and on the contrary, an installation in which every function is held by a different host, and some functions sometimes by several, is possible. The functions have an order, which is reflected in the order in which one must upgrade the servers—with the exception of the license server, which is somehow nevertheless the top of the hierarchy.

With relatively recent versions of ClearCase, there have been two options for the license server. Or rather, an additional option of FLEXlm licenses has been added to the traditional ClearCase offering. The latter one being stable, and mostly satisfactory, has been kept. The main rationale for FLEXlm is that this is the implementation of choice for other IBM/Rational products, starting with ClearQuest. We won't go into ClearQuest (although we'll mention it again in *Chapter 13*, *The Recent Years' Development*), because its integration with ClearCase has not been very convincing. While both request tracking and change management are reasonable concerns, electing ticket life cycle as the main driver of development has not proven to be compelling. It had a large share of the dramatic development of UCM and led in our experience to counterproductive results. Another possible rationale for FLEXlm is that it is a widely used platform, relatively feature rich, and flexible. It does offer an answer to areas in which the standard ClearCase licensing required ad-hoc solutions: redundancy and high-availability in various scenarios (network or host failure) and catastrophe recovery, and monitoring tools. The advantages are however not compelling. The solutions to offer similar benefits with the old framework were neither complex nor expensive. They relied on the acquisition of a second set of license keys, a commercial/political negotiation, disconnected from our technical point of view. We cannot comment much on the FLEXlm offering, because it goes beyond our experience, limited to early evaluations that were not at the time fully convincing.

One piece of advice worth mentioning is to make large license pools (within world "regions", which is the only restriction traditionally set by ClearCase to limit the use of the same licenses around the clock, successively in different time zones), to benefit from a maximal flexibility. Licenses are granted for a period tunable, but typically of 30 minutes, and their acquisition is not significantly impacted by latency.

The license server is at best a stable and dedicated host, seldom to be powered down for maintenance. Its ClearCase version need not be upgraded often, not even for all major releases, as traditional licensing uses only the **albd** protocol. FLEXlm ClearCase licenses do not even require that.

License monitoring (non FLEXlm) relies upon parsing the output of the *clearlicense* tool. One limitation of this method is that it reaches to only the *local* license server (the one specified in the `config/license_host` file on the current host, lacking a `-host` option such as in the `ct getlog` or `hostinfo` commands). But this is easily worked around on a dedicated low-end PC (with no users), by changing the contents of the `config/license_host` file before running the command in order to collect the data from multiple license servers. Another strategy to monitor license usage could be to add the string `-audit`, to the `license.db` file (on the license server, in the `/var/adm/rational/clearcase` directory), and to use `getlog albd`. This is however not fully reliable.

MultiSite licenses are distinct (for historical shortsighted commercial reasons). A user needs one if she uses a replicated vob.

The **registry** collectively designates a few flat files in the `rgy` directory under `/var/adm/rational/clearcase` on the registry server (which may be backed up to a second host, allowing for a `rgy_switch` procedure for maintenance or failure recovery purposes). There are two main pairs of files: for vobs (`vob_object` and `vob_tag`) and views (`view_object` and `view_tag`); each pair consists of an *object* and a *tag* registry. They are meant to be updated with the `register`/`unregister` (for object registries), and `mktag`/`rmtag` (for tag ones) commands. The `rgy_check` command allows to detect inconsistencies (such as duplicate or missing tags or the presence of tags for missing objects, among other things).

One cannot in general recommend to edit these files, but it may be useful to read them, in order, for example, to compare entries produced by different commands. This is often faster than parsing the output of `ct lsvob` or `lsview`, for example, to find entries sharing a certain storage or erroneously set to use the file server (where ClearCase is not installed) as vob or view server. Note that it is not necessary to edit the registry files to access objects created with non-intended names such as a `-foo` tag for a view. Cleartool commands understand where necessary the `"--"` last flag convention:

```
$ ct lsview -foo 2>&1 | head -2
cleartool: Error: Unrecognized option "-foo"
Usage: lsview [-short | -long] [-host hostname [-quick]]
$ ct lsview -- -foo
* -foo                  /cc/views/-foo.vws
$ ct rmview -tag -foo
$
```

Of the other files in the registry, let's mention a few:

- `regions`: Accessible via the commands `ct mkregion`, `rmregion`, and `lsregion`. The region to which a ClearCase client belongs (it belongs to one, and only one at a time) is set in the `config/rgy_region.conf` file, in the ClearCase data area. Vobs and views are *registered* once per registry, but *tagged* per region. Regions are usually used for Windows/UNIX interoperability, so that two regions are defined, and vobs are tagged in both (with tags obeying the respective operating system constraints). More exotic setups can be met, with only part of the vobs tagged in certain regions. Views may also be tagged to multiple regions—a practice subject to controversies.
- `vob_tag.sec`: It is the encrypted registry password required to create public vobs. This is produced by the *rgy_passwd* utility, requiring *root* privileges, and vob owner accounts have no special access to it.
- `storage_path`: Accessible via `ct mkstgloc`, `rmstgloc`, and `lsstgloc`. Storage locations are useful and handy for creating vobs and views, by simplifying the options to the `ct mkvob` and `mkview` commands. This is especially true when using the `-stg -auto` combination. One way for the administrator to affect it in presence of several storage locations is for her to use the `-ngpath` flag (no global path) for `mkstgloc` command to disqualify the locations not intended for "automatic" usage.
- `site_config`: Accessible via `ct setsite` and `lssite`. This holds some default values concerning ClearCase behavior, as well as performance tuning (view cache size, accessible on a per view basis with `ct setcache` and `getcache`).

Synchronization between regions

Synchronizing views tags and vobs tags, and between the different ClearCase regions (for example, UNIX and Windows) is just a matter of:

- Having the view and vob storage locations available in both environments
- Having the view and vob servers accessible in both environments
- Creating the view and vob tags with `cleartool mktag` command in the target region

For example, consider the following two regions:

```
$ ct lsreg
unix_reg
win_reg
```

If we have a view created in the `unix_reg` region as:

```
$ ct lsview -l joe_view
Tag: joe_view
 Global path: /filer01/viewstg/joe_view.vws
 Server host: viewserver.domain.com
 Region: unix_reg
 Active: YES
View on host: viewserver.domain.com
View server access path: /filer01/viewstg/joe_view.vws
View owner: joe
```

We can synchronize it with the Windows region by using the following command:

```
W:\>cleartool mktag -view -tag joe_view -reg win_reg \
 -hos viewserver.domain.com \
 -gpa \\filer01\viewstg\joe_view.vws \\filer01\viewstg\joe_view.vws
```

And similarly for the vob:

```
$ ct lsvob -l /vob/foo
Tag: /vob/testi2
 Global path: /filer01/vobstg/foo.vbs
 Server host: vobserver.domain.com
 Access: public
 Mount options:
 Region: unix_reg
Vob on host: vobserver.domain.com
Vob server access path: /filer01/vobstg/foo.vbs

W:\>cleartool mktag -vob -tag \foo -reg win_reg \
 -hos vobserver.domain.com
 -gpa \\filer01\vobstg\foo.vbs \\filer01\vobstg\foo.vbs
```

One persistent problem in interoperable contexts concerns the creation, from Windows, of views with storage on a filer, using a UNIX view server. This would be required for views intended to be accessed from both UNIX and Windows. The error is as follows:

```
cleartool: Error: Failed to record hostname <UNIX view server>
```

This is because the client is responsible for setting the view storage directory permissions and the view ACL. Unfortunately, these are Windows-specific constructs. The views must be created on UNIX and tagged to Windows later.

Monitoring client activity

ClearCase keeps track of the clients of the registry, and of the license database, in `client_list.db`, in the data area (`/var/adm/rational/clearcase`). This file is accessible remotely with the `ct lsclients` utility.

Note that the license database is different from the registry, and here is how one can distinguish between the two (concerning the client access):

```
$ ct lsclients -host beyond.lookingglass.uk -l -type registry
Client: beside.wonderland.uk
 Product: ClearCase 7.0.1.3
 Operating system: Linux 2.6.9-78.0.22.EL #1 Fri Apr 24 12:35:12 EDT 2009
 Hardware type: i686
 Registry host: beyond.lookingglass.uk
 Registry region: alice
 License host: license.wonderland.uk
 Last registry access: 2010-09-05T20:01:23+03:00

$ ct lsclients -host beyond.lookingglass.uk -type license -l
Host "beyond.lookingglass.uk" has no license clients.
```

So, the `beyond.lookingglass.uk` acts as the registry server for the client host `beside.wonderland.uk`, but not as the license server.

The license server configured for this client is `license.wonderland.uk`:

```
$ ct lsclients -host license.wonderland.uk -type license -l
Client: beside.wonderland.uk
 Product: ClearCase 7.0.1.3
 Operating system: Linux 2.6.9-78.0.22.EL #1 Fri Apr 24 12:35:12 EDT 2009
 Hardware type: i686
 Registry host: beyond.lookingglass.uk
 Registry region: alice
 License host: license.wonderland.uk
 Last registry access: 2010-09-05T20:01:23+03:00
```

The file format is line oriented, but the output of the utility, with the `-long` option, is not, and therefore often requires in practice some (Perl) post-processing to be useful. One typical reason for such a need comes from the use of DHCP addresses: client hosts are identified in the client database with their IP addresses, and if these are granted by DHCP servers, they will (typically) be leased for a limited time. This suffices to make the record unreliable, the same hosts being possibly recorded multiple times. The database gets flushed only at restart time (we do not know of any other way).

In the example below, the uniquely sorted list of clients is about 2.5 times shorter than the original one:

```
$ ct lsclients -host licsrv -type license | wc -l
 813
$ ct lsclients -host licsrv -type license | cut -d: -f1 | sort -u | wc -l
 331
```

Using the long output, we may build a hash, and store the recorded access time. When meeting the same name again, we update the hash record only if the new time is more recent than the one stored.

```
$ ct lsclients -l -host licsrv -type license | perl -n00e \
 'chomp; ($n,$a) = (/Client: ([^\n]+).*?(?:access: (.*)|$)/s);
 if (!$h{$n}{a} or $h{$n}{a} lt $a) { $h{$n}{a} = $a?$a:q() }
 END{ for(sort {$h{$a}{a} cmp $h{$b}{a}} keys %h)
  { print "$_: $h{$_}{a}\n" } }' | tail -3
mars: 2010-09-25T19:15:00+01
venus: 2010-09-25T19:15:02+01
jupiter: 2010-09-25T19:15:16+01
```

We are again (refer to *Chapter 9*, *Secondary Metadata*) using the paragraph-oriented input mode, stripping the newline, and matching the items with the `s` modifier (so that the "`.`" wildcard may match newlines). We take advantage of the fact that the standard format of time stamps makes them string-wise comparable.

Note that some records do not have a last access time (for reasons we can only try to guess). In order for them to nevertheless match the pattern with which we extract the fields we are interested in, we enclose the second part in a "`(?: ... )`" bracket pair (group, but do not record), as an alternative to an end of line, and to make the previous indiscriminate pattern *non-greedy*: "`.*?`"; We then store an empty value `q()` instead of the undefined one; we use an empty value so that it compares "older" than any recorded access time. Our output format is line oriented, so as to be greppable (or "tailable" as in the example above). Producing it in an `END` block allows us to sort it, here by the last access time.

Other fields may of course be recorded and printed (see the earlier example) such as `Product`, `Operating system`, `Hardware type`, and so on.

Location broker

Clients can be accessed remotely via the **albd service**, listening to port 371, for example, with the following utilities: `/opt/rational/clearcase/etc/utils/albd_list` (`c:\Program Files\Rational\ClearCase\etc\utils\albd_list.exe` on Windows), `ct getlog`, and so on.

```
$ albd_list beside.wonderland.uk
albd_server addr = 100.175.111.249, port= 371
PID 16530:
 syncmgr_server, tcp socket 33101: version 1; BUSY
  Storage path syncmgr_server
Albd_list complete
```

This "pings" the ClearCase client and lists the currently running albd processes (if any). On a ClearCase server, the output may be quite long.

```
$ ct getlog -host Win12345 albd
============================================================
Log Name: albd Hostname: Win12345 Date: 2010-09-24T12:36:32+03:00
Selection: Last 10 lines of log displayed
------------------------------------------------------------
2010-09-24T04:43:51+03:00 albd_server(24860): Job 6 ######################
                                "Daily VOB Space" (166208) Completed: OK.
2010-09-24T04:43:50+03:00 albd_server(24860): Job 6 ######################
                                     "Daily VOB Space"  (166208) Started.
2010-09-24T04:43:49+03:00 albd_server(24860): Job 5 ######################
                                   "Daily View Space" (169308) Completed: OK.
2010-09-24T04:30:12+03:00 albd_server(24860): Job 5 ######################
                                     "Daily View Space" (169308) Started.
2010-09-24T04:30:12+03:00 albd_server(24860): Job 3 ######################
                                "Daily Registry Backup" (169276) Completed: OK.
2010-09-24T04:30:12+03:00 albd_server(24860): Job 3 ######################
                                  "Daily Registry Backup" (169276) Started.
2010-09-24T04:30:12+03:00 albd_server(24860): Job 2 ######################
                                "Daily VOB Snapshots" (167520) Completed: OK.
2010-09-24T04:30:07+03:00 albd_server(24860): Job 2 ######################
                                     "Daily VOB Snapshots" (167520) Started.
2010-09-24T04:30:07+03:00 albd_server(24860): Job 1 ######################
                           "Daily VOB Pool Scrubbing" (166296) Completed: OK.
2010-09-24T04:30:04+03:00 albd_server(24860): Job 1 ######################
                             "Daily VOB Pool Scrubbing" (166296) Started.
============================================================
```

We'll look a bit deeper into logs and the scheduler in the further sections.
The **albd port** (actually, both tcp and upd ports) is the only well-known port ClearCase uses, and the only one on which a failure to connect will actually tell of a problem. One can also test it, for example, with `telnet client_host 371`, even if this admin idiom offers no advantage over using `albd_list`: as the albd protocol is not text based, the telnet session will display nothing and have to be interrupted. Sessionless communication resulting in small amounts of data may be carried on this port. For anything beyond that, the albd server will negotiate a connection backwards to the requester, using a high port, and continue on this port. Only for shipping purposes is there a way to restrict this high port within a range (more on this in next chapter). Understanding this is essential for whoever considers configuring a **firewall**: opening port(s) 371 is not enough; all the high ports must be opened, and the negotiation is initiated by the remote end. Let's make clear that using firewalls between ClearCase hosts can thus only really be considered in a few cases:

- For shipping
- For license acquisition, and even then, it will impact the ability to run *clearlicense*
- For web access, including via a remote client (and then the communication is limited and channeled via the http, https, ssl, ...ports)

Surprisingly to old ClearCase users, the presence of a firewall between a client and its registry will not stop ClearCase from working (as it used to do, on timeouts, in previous releases of ClearCase). It will however affect it, and impact even *other* users from other clients: delays between database lock acquisition and release will significantly increase, due to the latency resulting from firewall processing, adding up to the round-trip times.

Firewalls may add specific error cases such as by dynamically blocking ports as a measure against *Denial of Service* attacks, if a single command results in *excessive* output. This example is a typical default firewall configuration problem:

```
$ cleartool lsview
albd_rgy_get_entry call failed: RPC: Timed out
cleartool: Error: Trouble contacting registry on host "regsrv": #########
             timed out trying to communicate with ClearCase remote server
```

Remote monitoring infrastructure

We saw that the ClearCase infrastructure is typically distributed among multiple hosts. Problem investigation thus requires access to distributed information, first and foremost logs of various kinds, but also the state of the background tasks.

This is well supported by the ClearCase architecture, through the `ct getlog` and `schedule` commands. This architecture is well documented, and actually open to extension, and administrators would be well inspired in using this option, instead of resorting to the UNIX tools *crontabs* and *syslog*.

There is an exception to the consistency with which ClearCase uses its own architecture, and it is with shipping server installation, but we'll treat this in the next chapter.

We have already treated the output of both functions. There could be one deficiency to `getlog`, which is worth noting: it is not always obvious which log to inspect (between `error`, `albd`, `vobrpc`, `view`, and so on), and it is not trivial to make up one's mind. The best is often to log in to the remote host, and to tail a list of the logs sorted by their timestamps. Of course, this defeats the advantage we just praised.

An alternative is to ask for the last line of every log, and to sort them by their timestamps:

```
$ ct getlog -last 1 -all -host beyond 2>/dev/null | \
 perl -nle 'if(/^Log Name: (\w+)/) { $n=$1; next }
 $h{$n}=$_ if /^\d+/;
 END {print "$_: $h{$_}" for sort {$h{$a} gt $h{$b}} keys %h}'
view: 2010-09-24T11:01:26+01 view_server(12404): Db closed
mvfs: 2010-09-24T19:34:47+01 global(0): fs: Error: view=joe ##############
         vob=/vob/test - Lock on VOB database prevents write transactions
albd: 2010-09-26T04:33:00+01 albd_server(744): ############################
                           Job 6 "Daily VOB Space" (26669) Completed: OK.
```

This gives a clue of what may be worth reading next: we retained the log name, and the last line only if there was one, in a hash indexed by the name.

In an end block, we print both together on one line, sorting the logs by the contents of the records, taking advantage from the fact that these start with the timestamps, and as we noted earlier already, the timestamps are string-wise comparable. Note however that this only works for the line formatted class of logs, as it assumes a timestamp. This unfortunately excludes the `error` log, for instance.

Scheduler

We'll turn now to the question of adding one's own custom tasks and logs.

Every *job* in the scheduler actually runs a *task*. All the tasks are numbered and listed in a flat file: `/var/adm/rational/clearcase/scheduler/tasks/task_registry`. The tasks themselves should also be located either in the same directory `/var/adm/rational/clearcase/scheduler/tasks` (usually the customized ones) or in `/opt/rational/clearcase/config/scheduler/tasks` (usually the standard ones). Our first move (after writing the script we intend to use as a scheduler job) is thus to add one task description to the end of file.

The example we take here is one on which we'll come back in the next chapter, for monitoring the status of the MultiSite replication:

```
Task.Begin
Task.Id: 103
Task.Name: "Moving received repmon logs"
Task.Pathname: "repmon_mv.sh"
Task.End
```

Here `repmon_mv.sh` is the name of our script. It moves replica monitoring logs received from other hosts to a certain location: `/tmp` dir in our example. As we placed it into the `/var/adm/rational/clearcase/scheduler/tasks` directory, we do not need to specify its full path.

We need next to create a new job in the scheduler, using this task. We can use the interactive command (which will start the editor specified in the environment, by default `vi`):

```
$ cleartool sched -edit
```

And with it, we add something like the following:

```
Job.Begin
Job.Name: "Daily Repmon logs moving"
Job.Description.Begin:
Move all received replica monitoring packets to /tmp
Job.Description.End:
Job.Schedule.Daily.Frequency: 1
Job.Schedule.FirstStartTime: 00:00:00
Job.Schedule.StartTimeRestartFrequency: 04:00:00
Job.DeleteWhenCompleted: FALSE
Job.Task: 103
Job.Args:
Job.NotifyInfo.OnEvents: JobEndFail
Job.NotifyInfo.Using: email
Job.NotifyInfo.Recipients: root
Job.End
```

Administrators may want to keep a backup of the schedule and possibly to version their changes.

This is not conveniently supported by `ct schedule` because the output combines commands and outputs.

Let's focus on one single job (the one we just added), instead of on the whole schedule (`ct sched -get -sched`):

```
$ ct sched -get -job "Daily Repmon logs moving"
Job.Begin
 Job.Id: 20
 Job.Name: "Daily Repmon logs moving"
 Job.Description.Begin:
Move all received replica monitoring packets to /tmp
 Job.Description.End:
 Job.Schedule.Daily.Frequency: 1
 Job.Schedule.StartDate: 2010-09-26
 Job.Schedule.FirstStartTime: 00:00:00
 Job.Schedule.StartTimeRestartFrequency: 04:00:00
 Job.DeleteWhenCompleted: FALSE
 Job.Task: 103
 # Job.Task: "Moving received repmon logs"
 Job.Args:
 Job.NotifyInfo.OnEvents: JobEndFail
 Job.NotifyInfo.Using: email
 Job.NotifyInfo.Recipients: root
 Job.Created: 2010-09-26T14:07:35+01 by anis/root@beyond
 Job.LastModified: 2010-09-26T14:07:35+01 by anis/root@beyond
 Job.NextRunTime: 2010-09-26T20:00:00+01
 Job.LastCompletionInfo.ProcessId: 2894
 Job.LastCompletionInfo.Started: 2010-09-26T16:00:00+01
 Job.LastCompletionInfo.Ended: 2010-09-26T16:00:02+01
 Job.LastCompletionInfo.ExitStatus: 0x0
Job.End
```

Actually, even commands get reinterpreted: job ids are produced automatically and task names are kept only as comments.

But the most obvious is that timing information is added and updated, as well as completion info, possibly including multiline excerpts of the standard error:

```
Job.LastCompletionInfo.Messages.Begin:
mv: cannot stat `/usr/atria/shipping/ms_ship/incoming/*.repmon*': #######
                                                    No such file or directory
Job.LastCompletionInfo.Messages.End:
```

This results in the fact that some processing of this output is needed if one intends to use it as input, for example, to restore a previous state. Some more work for Perl.

Now, we want to access the result of our customized task (the replica monitoring logs in the `/tmp` directory) via the `cleartool getlog` command, so we add a new log.

The ClearCase logs are also defined in a flat file, in the *read-only* hierarchy this time: `/opt/rational/clearcase/config/services/log_classes.db`.

This is where we have to add one line to gain access to the received and moved replica monitoring logs (still our same example) via *cleartool getlog*.

This is not documented in ClearCase:

```
-class=repmon;-form=imsg_file;-sources=/tmp/*.repmon;- ###############
                    description=ClearCase VOB replica monitoring log;
```

The modified `log_classes.db` file does need to be maintained manually through the ClearCase upgrades installations (which may overwrite it).

One may replace it with a symlink to under the `/var/adm` hierarchy (which will not get overwritten by upgrades), and keep the backup there.

There are two main log formats: *ascii* and *imsg_file* (plus *mvfs_file*, where the timestamps are encoded, requiring the `time2date` utility in the `etc/utils` directory, to decode them).

Logs in the `log` directory get rotated, so that the *sources* typically mention two files: the current and the old one.

We chose not to use this feature for our log.

We may now check that our log is known to the `ct getlog -inquire` command, and fetch it as any other:

```
$ ct getlog repmon
```

All the `-host`, `-last`, `-since`, and `-around getlog` options are available.

```
$ ct getlog -host beyond -last 100 repmon
$ ct getlog -host beyond -since today repmon
```

Storage and backup

Storage administration was greatly simplified with the certification of **NetApp filers**, followed by some other vendors.

Filers may be configured for use as **Network Attached Storage** (**NAS**), that is, file servers, serving through NFS and/or CIFS, or **Storage Area Network** (**SAN**). The latter potentially offers greater performance, at a lower level and with less flexibility. We admit having seen them used effectively with ClearCase. Nevertheless, it is the NAS configuration (the filer proper), which seems most attractive to us, and with which we have the best experience. We'll restrict our comments to this case.

Filers were first used to store the pools (which were then known as *remote*) via symbolic links. This was temporary until they could be certified, so that whole vobs, including the database, were stored on the filer. Let us drop this outdated setup as well.

The filer relieves the vob server from some noticeable load, part of which is that of running the NFS and CIFS (*samba*) servers.

It also makes it trivial to switch a vob from one server to another (of the same architecture); the data may stay untouched. This is especially valuable in upgrade scenarios.

The most important advantage of filer storage, from an end user perspective, is the **snapshot** functionality and their use for backup.

Filers do not overwrite the data they store; new versions of files are stored in newly allocated disk clusters. This makes it possible to freeze the state of the storage at any given time by storing an image of the directory objects (and keeping the clusters referenced from there from being reverted to the free pool).

This function is termed taking a *snapshot*. From the perspective of a database, the only concern is to ensure that the database caches are flushed before this happens, so that the disk image is consistent.

This function is performed by a side-effect of *locking* the vobs: once the vob is locked, it may be snapped, which takes a few seconds, and then it may be unlocked. The backup (for example, to tape or to any suitable archive) may thus take place on the snapshot (and not on live data).

Restoring from there is easier and incomparably faster than from tape. We saw in the last chapter how it may offer opportunities to restore individual versions.
This implies that the snapshots may be mounted to user accessible mount points.

In recovery scenarios, one typically needs to synchronize the views with the recovered vob.

We'll deal later in this chapter with issues of vob protection, including the storage area, pools, and containers.

Vob size

The `ct space -vob` command allows to understand what composes the size of a given vob:

```
$ ct space -vob /vob/foo
Use(Mb) %Use Directory
 13.7 0% VOB database /vobstg/foo.vbs/db
  0.1 0% administration data /vobstg/foo.vbs/admin
  0.3 0% cleartext pool /vobstg/foo.vbs/c/cdft
  0.0 0% derived object pool /vobstg/foo.vbs/d/ddft
146.9 0% source pool /vobstg/foo.vbs/s/sdft
------------------------------------------------------
161.1 0% Subtotal
1078199.0 92% Filesystem fstore:/vol/vol01/vobstg (capacity 1177804.8 Mb)

Total usage 2010-09-26T06:47:19+01 for vob "/vob/foo" is 161.1 Mb
```

Note that without the `-update` option, the `ct space -vob` command works on the last results obtained from the *Daily VOB Space* built-in job, and will thus not be affected by any changes on the spot.

One may of course force a run of the job (as with any job):

```
$ ct sched -run "Daily VOB Space"
```

But this runs the job for all local vobs, and may thus take a long time.

The cleartext and the DO pools contain only transient data, which may be reproduced. Cleartext containers only cache versions of elements accessed recently. We saw in the previous chapter in the case of text files how the file manager constructed them. Another typical case is that of expanding versions stored in compressed format. The size of these two pools is affected by the scrubber, which is typically run by the predefined scheduler job: *Daily VOB Pool Scrubbing*.

```
$ ct sched -get -job "Daily VOB Pool Scrubbing" | \
 egrep 'Schedule.*Freq|#.*Task'
    Job.Schedule.Daily.Frequency: 1
    # Job.Task: "VOB Pool Scrubber"
$ ct sched -get -tasks | perl -n00e 'print if /: 3$/ms'
Task.Begin
    Task.Id: 3
    Task.Name: "VOB Pool Scrubber"
    Task.Pathname: "scrubber.sh"
Task.End
```

This task uses a script (by default: `/opt/rational/clearcase/config/scheduler/tasks/scrubber.sh`) to drive the `/opt/rational/clearcase/etc/scrubber` tool. The default options examine all the pools in all the vobs. The result is logged in a way accessible to `getlog`, in unformatted mode:

```
$ ct getlog -host alice -full scrubber | \
 perl -ne 'print if m%^Started.*/foo.*Feb 12%...m%^Fin.*/foo.*Feb 12%'
Started VOB alice:/vobstg/foo.vbs at Sat Feb 12 12:26:42 2011
Start scrub VOB alice:/vobstg/foo.vbs Pool ddft
Stats for VOB alice:/vobstg/foo.vbs Pool ddft:

Get cntr tm   0.060150
Setup tm      0.015319
Scrub tm      0.000035
Total tm      0.075504
Start size 1  Deleted   0  Limit size        0
Start files       2  Deleted          0  Subdir dels       0
Statistics for scrub of DO Pool ddft:
DO's              0  Scrubs            0  Strands           0
Lost refs         0  No DO's           2
Start scrub VOB alice:/vobstg/foo.vbs Pool cdft
Stats for VOB alice:/vobstg/foo.vbs Pool cdft:

Get cntr tm   2.527032
Setup tm      0.000019
Scrub tm      0.000417
Total tm      2.527468
Start size 1514  Deleted   1  Limit size        0
Start files     182  Deleted          1  Subdir dels       8
Finished VOB alice:/vobstg/atcctest.vbs at Sat Feb 12 12:26:45 2011
```

This time we used a range line-oriented matching as we had a clear way to specify the start and end of the area of interest. The output shows various timings and deletion statistics for both pools, DO and cleartext. In this run, only one cleartext container was deleted.

Let's note that the scrubber may be run manually, with parameters involving size, age, and heuristics to preserve useful data, along with options allowing us to select the vobs and pools.

If the source pool grows abnormally, one must suspect the process of importing or creating new versions and look for redundant data: either evil twins or identical versions.

The cure is then to remove the versions, branches, or elements created by mistake (respectively with `rmver`, `rmbranch`, and `rmelem`).

However, this will not affect (at least not decrease) the size of the database. The only thing which may affect it is `vob_scrubber`.
In particular, scrubbing the oplogs may be the solution. We touched the topic in the last chapter.

The tool to give an account of the space consumption within the database is *countdb*. Details on the size of the various objects is found in a technote: *About the ClearCase database utility countdb* (***#1126456***).

Authentication

ClearCase has an identity management mechanism, which is called *credmap*. It is switched off by default and is rarely used in practice, as it introduces one more level of authentication, most often unnecessary in the case of a UNIX vob server with NetApp storage for vobs and views (and the appropriate NetApp access rights) and CIFS for Windows users mapping. This is why IBM mostly ***recommends*** it in case one uses a Windows vob server and needs to introduce a verification mechanism for its UNIX users.

In some rare cases, though, one might want to enable credmap authentication on the UNIX vob server as well, for the purpose of enforcing that Windows users belong to a certain Windows network domain.

This can be done by creating a `/var/adm/rational/clearcase/config/credmap.conf` file, putting the desired Windows domain name there:

```
$ cat /var/adm/rational/clearcase/config/credmap.conf
AllowedDomain=RATIONAL
```

Or in ClearCase 7.0 and up, there is an option to allow Windows users to belong to any network domain (but no local users would be allowed then):

```
$ cat /var/adm/rational/clearcase/config/credmap.conf
AllowedDomain=[-]
```

Importing files to ClearCase

Now we will take a look at a few issues related to importing data to ClearCase or relocating the data between vobs using both standard tools (*clearfsimport*, *clearimport*/*clearexport*, *ct relocate*) and *synctree*.

A priori, these tools are very different:

- ct relocate is targeted at *moving* elements, with all their history, from one vob to another
- clearimport is targeted at *copying* elements with their history into a vob
- clearfsimport and synctree are useful for *copying* only versions, from a source to a destination that may be within the same vob

All these tools are applicable only for base ClearCase vobs. First, we will also take a brief look at UCM to see how data is supposed to be imported and moved there.

Even UCM has to use Perl

One interesting observation concerning UCM (more of it later in Chapter 13) is that it is rather difficult to get there (for example, import the initial set of elements), and rather difficult to change it (relocate something from one vob to another). Like doing arithmetic using Latin numbers: it was just not designed for that! And the general recommendation from IBM is ...not to do it. It is also not possible to convert a UCM component vob to a base ClearCase vob.

The already mentioned *synctree* tool (there is more on it a bit later in this chapter) can also be used to fetch data from a UCM vob via a UCM view and import it to a base ClearCase vob.

The rescue scenario would be the following: choose a UCM baseline (for example, `BL1`), create a development stream based on it along with a view (for example, `BL1_view`), set a base ClearCase view (`view1` in the example that will follow), and then perform the data import via the view-extended path of the UCM view from the UCM vob (`/vob/ucmvob`) into the base ClearCase vob (`/vob/tools`):

```
$ ct setview view1
$ synctree -sb /view/BL1_view/vob/ucmvob/foo -db /vob/tools/foo -ci -yes
```

One can, of course, fetch several baselines in the same way from the UCM vob, by creating a new development stream along with a view for each baseline. There, the `-reuse` and `-vreuse` options (as well as `-rm`) become also useful.

A similar need (the ability to move the data around a bit even within a UCM environment) was also recognized by IBM, and in order to be able to relocate some data from a UCM vob to another something that originally was completely out of the question), one can (without any guarantee of course) find some aid from a custom Perl script `mkelem_cpver.pl`, provided in the ClearCase installation.

Relocating

When it comes to relocating (part of) the vob to a different vob, it is recommendable to always use the `cleartool relocate` command instead of `clearexport/clearimport` pair. The latter has a hardcoded behavior, which is worth knowing in advance; it is able to handle the whole version trees of the *file* elements (including all its branches) but has a restriction concerning these of the *directory* elements:

```
$ ct setview myview
$ cd /vob/foo
$ clearexport_ccase -r -o /tmp/foo_export .
$ clearimport -d /vob/newfoo /tmp/foo_export
...
$ ct lsvtree /vob/newfoo
/vob/newfoo@@/main
/vob/newfoo@@/main/0
/vob/newfoo@@/main/1 (FOO, FOO_3.00, RREL_0.80, ZOO, ...)

$ ct ls /vob/newfoo
/vob/newfoo/mdir@@/main/1 Rule: /main/LATEST [-mkbranch br1]
/vob/newfoo/bdir@@/main/1 Rule: /main/LATEST [-mkbranch br1]
/vob/newfoo/zdir@@/main/1 Rule: /main/LATEST [-mkbranch br1]

$ ct ls /vob/foo
/vob/newfoo/mdir@@/main/45 Rule: /main/LATEST [-mkbranch br1]
/vob/newfoo/bdir@@/main/4  Rule: /main/LATEST [-mkbranch br1]
/vob/newfoo/zdir@@/main/2  Rule: /main/LATEST [-mkbranch br1]
```

As can be seen on this example, `clearexport` exports only the current version of the directory element selected by the view, and `clearimport` always imports it as the `/main/1` version in the new vob. Tuning the views config specs does not help to fetch the rest of the directory elements version trees.

However, the file elements' version trees are fully imported:

```
$ ct lsvtree /vob/newfoo/java/1/makefile
/vob/newfoo/java/1/makefile@@/main
/vob/newfoo/java/1/makefile@@/main/8 (LBL_1.03, LBL_1.02, LBL_1.00)
/vob/newfoo/java/1/makefile@@/main/jb
/vob/newfoo/java/1/makefile@@/main/jb/2

$ ct lsvtree /vob/foo/java/1/makefile
/vob/foo/java/1/makefile@@/main
/vob/foo/java/1/makefile@@/main/8 (LBL_1.03, LBL_1.02, LBL_1.00)
/vob/foo/java/1/makefile@@/main/jb
/vob/foo/java/1/makefile@@/main/jb/2
```

`cleartool relocate` lifts these restrictions and is able to relocate both files and directories elements along with all their branches and versions; however, it is not designed to relocate a whole vob but only a part of it (which has to be specified as a pathname in the vob):

```
$ cleartool relocate -f -update /vob/foo/makedir /vob/newfoo
```

The `makedir` directory element and all its entries will be relocated correctly to the `/vob/newfoo` vob.

The closest to relocating the whole vob is to copy/move it under another one as a subdirectory:

```
$ ct relocate -update /vob/foo /vob/newfoo
$ ll /vob/newfoo
total 4
drwxrwxr-x 6 vobadm cc 294 May 6 13:05 foo
drwxr-xr-x 2 joe jgroup 0 May 6 13:03 lost+found
```

So, to relocate the whole vob, one would need to use other tools (see *Copying a vob* section below).

Note that here we used relocate in an *update* mode, which amounts to a *copy*, even if it was originally intended as a transient phase in a move (`ct relocate` without `-update` option). In the update mode, the original data is kept untouched.

Note that the oid of the elements is preserved:

```
foo> ct des -fmt "%On\n" /vob/foo/makedir@@ /vob/newfoo/makedir@@ | \
 sort -u
8a3fd1d6.14a511df.8283.00:01:37:45:23:13
```

Relocate still depends on the current config spec for creating the version of the root directory of the import. In its default mode (move), it is destructive for the information at the original site. Even if it builds symbolic links (and one hyperlink) to trace the relocation, it does break the config records of any derived objects depending on any relocated version.
Note that it may create in the target vob events older than the vob itself.

It looks as if a tool such as *synctree* (see next section) for importing files to ClearCase would desperately be needed for vob relocation purposes.

Importing with synctree

We already mentioned synctree at large in *Chapter 8*, *Tools Maintenance*.

Importing large amounts of files from external directories, for example, third-party tools, may lead to problems mentioned earlier such as importing the same files multiple times.

This may happen when importing several releases out of order in succession, using the same branch.

Let's consider one file, `foo`, which changes between release 1 and release 2; let's suppose we import in succession releases 1, then 2, and then 1.1.
This will yield three versions of `foo`, 1 and 3 being identical.

A worst case yet is when `foo` is deleted in release 2 (with the `-rmname` option either in *clearfsimport* or *synctree*): in that case, importing 1.1 over 2 with `clearfsimport` will yield an evil twin, whereas the `-reuse` option of the *synctree* helps to avoid that. In fact, the `-rm` option is only safe once one knows one can rely on a `-reuse` one to avoid creating evil twins later.

Another scenario is this of releases for different platforms, imported to platform-specific branches: platform independent files (for example, header files) will be needlessly imported into every branch.

These scenarios are taken into consideration by special options of synctree:

```
$ synctree -sb source -db destination -reuse -vreuse -label FOO -/ipc=1
[...]
```

- `-reuse` will avoid the evil twin creation, by resurrecting a previously deleted element from the version tree (note that it requires `-label` or `-lb_mods` to resurrect existing versions without creating duplicates).
- `-vreuse`, in conjunction with `-label`, will compare a file not matching the selected version, to versions of the same size in the version tree. Finding a suitable version, it will skip importing it anew and will label it instead.

- `-/ipc=1` will use *the ipc mode* of the underlying ***ClearCase::Argv*** module, thereby invoking one single instance of cleartool, and ensuring that the extra calls needed to support the above functionality do not incur a prohibitive cost. It also makes the `-cr` option reasonable. This one aims at preserving the config records of derived objects, but as it uses the `-from` option of `checkin`, it requires a distinct invocation for every element, which makes it prohibitive without the ipc mode. This functionality is unique to synctree, with no equivalent in clearfsimport.

Synctree also fixes some clearfsimport hiccups, such as infinite cycles in case the data you want to import is already in place as view-private:

```
$ ct ls -d /vob/test/dir1
/vob/test/dir1
$ clearfsimport -r /vob/test/dir1 /vob/test/dir1
Creating directory "/vob/test/dir1".
Private version of "/vob/test/dir1" saved in "/vob/test/dir1.keep".
Creating directory "/vob/test/dir1".
Created branch "br1" from "/vob/test/dir1" version "/main/0".
Validating directory "/vob/test/dir1".
clearfsimport: Warning: Using existing checkout of directory ############
                                                        "/vob/test/dir1".
Creating directory "/vob/test/dir1/dir1".
Created branch "br1" from "/vob/test/dir1/dir1" version "/main/0".
Creating directory "/vob/test/dir1/dir1/dir1".
Created branch "br1" from "/vob/test/dir1/dir1/dir1" version "/main/0".
```

In synctree, the same is worked around:

```
$ synctree -sb /vob/test/dir1 -db /vob/test/dir1
/usr/bin/synctree: Error: 6 view-private files exist under ###############
                                                  /view/v1/vob/test/dir1:
```

Synctree also provides a number of useful features such as:

- Branch from root (non-cascading) branch support
- Support for regular expressions (`-Narrow [!]<re>` option) to limit files to be imported to those matching `/re/`
- Hash mapping of source set of files to the destination via `-map` option
- Possibility (`-rellinks` option) for turning absolute symlinks within source base into relative ones

ClearCase::Wrapper

For certain fine-tunings related to importing files to ClearCase, one would find a great help in ***ClearCase::Wrapper***, which provides such useful functionality as, for example, recursive `mkelem` and `checkin`.

Copying a vob

We'll consider here moving a whole vob, for instance migrating it to a new server, and why it is not suitable for duplication. Vob duplication is requested in case the company's development environment has been split or in similar situations.

The sad news is that there is no shortcut from exporting and re-importing the relevant data.

Moving vob storage

One can re-register the vob by moving its storage to a new directory and making a new vob tag:

```
$ ct lock vob:/vob/foo
$ ct umount /vob/foo
$ ct unreg -vob /vobstg/foo.vbs
$ ct rmtag -vob /vob/foo
$ pkill -f foo.vbs

$ cd /vobstg/foo.vbs
$ find . -depth | cpio -pdmu /vobstg/bar.vbs

$ ct reg -vob /vobstg/bar.vbs
$ ct mktag -vob -tag /vob/bar -public -host vobserver \
 -gpath /vobstg/bar.vbs /vobstg/bar.vbs
$ ct mount /vob/bar
$ ct unlock vob:/vob/bar
```

Note however, that this method is not suitable to duplicate the vob, in other words, to keep both `foo` and `bar` vobs independently. The problem is that the vob oids of the two copies are identical, which constitutes a mine field: this consanguinity of the two copies will survive their divergence and lead to corruption even years later at yet unknown sites, if a packet originating from the first hierarchy reaches a replica of the second. Let's note that using MultiSite to produce a new replica does nothing to solve the problem. Forcibly modifying the oid in one copy hits "internal error" problems when creating hyperlinks.

Copying vob by replication

The easiest way to copy a vob to a different host is often to replicate it there.

```
[host1]$ multitool mkreplica -exp -work /tmp/foo -nc -fship \
 host2:rep2@/vob/foo
```

If the actual replication is not desired for some reason, one can consider reciprocal replica removal at both sites:

```
[host2]$ ct lsrep -fmt "%n %[replica_host]p\n" -invob /vob/foo
rep1 host1
rep2 host2
[host2]$ mt chmaster -all -obsolete_replica rep1@/vob/foo rep2@/vob/foo
Chmaster -obsolete_replica will transfer mastership of all objects
 currently mastered by rep1@/vob/foo to rep2@/vob/foo. Do not proceed
with this
operation unless the VOB for rep1@/vob/foo has been deleted. Do you want
to proceed? [no] yes
Changed mastership of all objects
You must now complete the removal of rep1@/vob/foo by entering
the following command at the master site for rep1@/vob/foo:
multitool rmreplica rep1@/vob/foo
[host2]$ multitool rmreplica rep1@/vob/foo
The last remaining replica has been deleted; ############################
                                            disabling replication in VOB.
Deleted replica "rep1".
[host2]$ ct lsrep -fmt "%n %[replica_host]p\n" -invob /vob/foo
rep2 host2


[host1]$ ct lsrep -fmt "%n %[replica_host]p\n" -invob /vob/foo
rep1 host1
rep2 host2
[host1]$ mt chmaster -all -obsolete_replica rep2@/vob/foo rep1@/vob/foo
Chmaster -obsolete_replica will transfer mastership of all objects
currently mastered by rep2@/vob/foo to rep1@/vob/foo. ####################
                                                    Do not proceed with this
operation unless the VOB for rep2@/vob/foo has been deleted. #############
                                          Do you want to proceed? [no] yes
Changed mastership of all objects
```

```
You must now complete the removal of rep2@/vob/foo by entering
the following command at the master site for rep2@/vob/foo:
multitool rmreplica rep2@/vob/foo
[host1]$ multitool rmreplica rep2@/vob/foo
The last remaining replica has been deleted; ############################
                                                 disabling replication in VOB.
Deleted replica "rep2".
[host1]$ ct lsrep -fmt "%n %[replica_host]p\n" -invob /vob/foo
rep1 host1
```

Re-registering a replica

This happens when a vob has to be re-registered with a new vob server host name (for example, in case the vob server domain name has changed), or when the vob has been migrated to a new vob server:

```
$ ct reg -vob -host newvobsrv.domain.com -rep \
 -hpath /vobstg/foo.vbs /vobstg/foo.vbs
```

The hostname must also be updated explicitly for the vob's replica (even if the vob is not actually replicated!):

```
$ ct lsvob -l /vob/foo | grep host
 Server host: newvobsrv
Vob on host: newvobsrv

$ ct lsreplica -fmt "%n %[replica_host]p\n" -invob /vob/foo
rep1 oldvobsrv
```

As it turns out that even if the vob is registered with a new vob server, its replica stays configured with the old one.

Here is how to fix it:

```
$ multitool chreplica -host newvobsrv rep1@/vob/foo
Updated replica information for "rep1@/vob/foo".
```

Views cleanup

Having views on a dedicated view server is a big relief for the administrator, as all the "abandoned" views (and other) problems can be handled in place. Troubleshooting the problems with local (end-user workstation located) views is trickier.

ClearCase administrators are used to the following scenario: a local view has been removed incorrectly or is not accessible as the workstation has crashed (has been replaced, and so on). Typically there would be some checkouts in such an abandoned local view.

In order to clean it up, one needs to find out the view uuid, and perform the `ct rmview -uuid` command on some host belonging to the appropriate region (the same one as where the view was registered).

To find out the view tag in question and its uuid, the administrator would sometimes need to check it using the `ct des -l vob:` command:

```
$ ct des -l vob:/vob/foo
versioned object base "/vob/foo"
...
 VOB holds objects from the following views:
 Win123456:c:\cc\winview1.vws ##############################################
                            [uuid 03f6851c.a49f11db.8a12.00:16:35:7f:04:48]
```

The view in question should be unregistered and its tag removed (so that for example, a view with the same tag could be re-created):

```
$ ct unreg -view -uuid 03f6851c.a49f11db.8a12.00:16:35:7f:04:48
$ ct rmtag -view winview1
```

The last step (and the longer) is to clean up the vob references to this view (for all the vobs in the following example):

```
$ ct rmview -uuid 03f6851c.a49f11db.8a12.00:16:35:7f:04:48 -all
```

Note that this operation leaves the view inconsistent, which is why it is better to unregister first: it is not supposed to be usable anymore.

ClearCase and Apache integration

ClearCase integration with a web server (Apache) could be beneficial for both parties:

- **ClearCase**: Providing public and easy (read-only) access to ClearCase vobs via http
- **Apache**: Maintaining the web server pages under ClearCase to be able to manage them better

This can be configured on any ClearCase client host, having Apache installed there as well.

The main idea is to use a dedicated view, let's call it `webview`, with a config spec preventing checkout.

The following is an example config spec:

```
$ ct catcs -tag webview
element * TOOLS -nocheckout
element * .../br1/LATEST -nocheckout
element * .../m/LATEST -nocheckout
element /vob/foo/... .../z/LATEST -nocheckout
element * /main/LATEST -nocheckout
```

Then for the Apache side configuration, set the following in `/etc/httpd/conf/httpd.conf`:

```
Alias /vob/ "/view/webview/vob/"

<Directory "/view/webview/vob">

Options Indexes FollowSymlinks MultiViews
AllowOverride None
Order allow,deny
Allow from all
EnableSendFile off
```

Note the `EnableSendFile off` setting turning Apache optimization off: using the *sendfile* system call, which is efficient but not supported by the MVFS filesystem. The symptom resulting of missing this step is that every file just looks empty. Note that this is necessary only if one wants to use a dynamic view (we do).

You may also set `DocumentRoot` to some location in a ClearCase vob:

```
DocumentRoot "/view/webview/vob/web"

<Directory "/view/webview/vob/web">
```

That's basically all concerning the configuration. Now, the `webview` should be first started, followed by restarting of the web server:

```
$ ct startview webview
$ apachectl -k start
```

Now you can read your ClearCase vobs via http (where `ccserver` is the name of the host where you did this integration):

```
http://ccserver/vob/foo
```

Versions that are not selected by the current `webview` view configuration can be accessible via the ClearCase version extended path (as usual) even in the URL path, for example:

```
http://ccserver/vob/foo/bar@@/main/br1/1
```

This provides the most lightweight type of access to a ClearCase vob: instant and easy (no client installation or setup, or any bulky data download), but of course, it is read-only and non-configurable. This could be ideal for accessing some stable documentation or codebase stored in a vob for reference purposes.

It would be recommendable to add a new UNIX service for managing the `webview` and ensuring it starts up automatically in case of the host reboot:

```
$ cat /etc/init.d/webview

#! /bin/bash
#
# webview Start/Stop the ClearCase webview view
# Source function library.
. /etc/init.d/functions

RETVAL=0

start() {
echo -n $"Starting the webview view: "
/opt/rational/clearcase/bin/cleartool startview webview
RETVAL=$?
echo
[ $RETVAL -eq 0 ]
return $RETVAL
}

stop() {
echo -n $"Stopping the webview view: "
/opt/rational/clearcase/bin/cleartool endview webview
RETVAL=$?
echo
[ $RETVAL -eq 0 ]
return $RETVAL
}

case "$1" in
start)
start
;;
stop)
stop
;;
```

```
*)
echo $"Usage: $0 {start|stop}"
exit 1
esac

exit $?

$ chkconfig --add webview
```

Installation tricks

Sometimes installing ClearCase on a non-supported platform (such as Fedora) is just a matter of one configuration file fine-tuning.

A default installation attempt would produce an error: `platform is not supported`. The applicable workaround is just a matter of creating `/etc/redhat-release` file, such as:

```
$ cat /etc/redhat-release
Red Hat Enterprise Linux AS release 4
```

After this the installation goes fine and ClearCase is fully functional (but it won't be supported by IBM).

Bottom-up

Here we will analyze a few error cases encountered by the end users, and will discuss possible ways of resolving them.

ALBD account problems

Working with a Windows ClearCase client, one might encounter the following problem: the Albd service does not start up due to the ALBD account login failure (note that this is a Windows ClearCase client-specific problem; no such configuration is applicable for the UNIX client). ClearCase becomes then unusable: no views can be created or started, etc. (Let's note though that some ClearCase operations could still be possible, provided that originally ALBD account was set up correctly, and got misconfigured later: for example, dynamic views would not be usable, but certain operations with snapshot views, including the view update and even checkouts and checkins could succeed).

Sometimes one would see errors of this kind when working in a snapshot view with the misconfigured (wrong username/password) ALBD account:

```
Info 10 May, 2009 10:46:31 view_server 1884 Using view ###############
                                    c:\joe\view.stg, on host: Win123456
Warning 10 May, 2008 10:07:12 view_server 2740 albd_contact call #####
                                       failed: RPC: Unable to receive;
                              errno = [WINSOCK] Connection reset by peer
Error 10 May, 2009 10:07:12 view_server 4336 view_server.exe(4336): ##
                Error: Unable to contact albd_server on host 'Win123456'
Error 10 May, 2009 09:11:57 view_server 2740 view_server.exe(2740): ##
    Error: Unable to get cleartext for vob: 1204d946.d7b011db.8730.00:
                                              16:35:7f:04:52 ver 0x112ea.
Error 10 May, 2009 09:11:57 view_server 2740 view_server.exe(2740): ##
      Error: Unable to construct cleartext for object "0x112EA" in VOB
"vobserver.domain.com:/vobstg/foo.vbs": error detected by ClearCase ##
                                                              subsystem
Error 10 May, 2009 09:11:57 view_server 2740 view_server.exe(2740): ##
       Error: Type manager "text_file_delta" failed construct_version
                                                            operation.
Warning 05 May, 2009 09:11:57 view_server 2740 text_file_delta: Error:
    Unable to open file "\\vobserver.domain.com\vobstg\foo.vbs\s/sdft\
        2a/20/0-ade01e85026c4e8da772460af72cbc72-rb": Invalid argument
```

Usually the ALBD password is not available for the end users in clear text form because of security reasons. The ALBD password is stored in the `sitedefs.dat` file in the ClearCase Windows release area, in encrypted format. But it does not help if one needs to reset the ALBD password manually (in **Services | Atria Location Broker | Logon**).

The way to do it is either to reinstall the ClearCase client from the available Windows release area (which is really overkill, at least from the end user's point of view), or to use the ***ALBD account reset utility*** provided by IBM (not officially supported though).

Using this utility would require that one has access to the Windows ClearCase release area, and would specify its location as a parameter. Then the encrypted password is read from the `sitedefs.dat` and is updated in the appropriate way in the Windows Registry. This is quite a light way of working around the ALBD problem, and is usually much appreciated by the end users.

Changing the type manager

Sometimes it turns out that certain files do not "qualify" for their default type manager:

```
$ ct ci -nc /vob/tools/jdk/THIRDPARTYLICENSEREADME.txt
text_file_delta: Error: "/vob/tools/jdk/THIRDPARTYLICENSEREADME.txt" ####
           is not a 'text file': it contains a line exceeding 8000 bytes.
Use a different type manager (such as compressed file).
```

Indeed, the text file type manager cannot handle such long lines. One must obey and change the element type in order to use a different type manager:

```
$ ct chtype file /vob/tools/jdk/THIRDPARTYLICENSEREADME.txt
Change version manager and reconstruct all versions for ##################
                   "/vob/tools/jdk/THIRDPARTYLICENSEREADME.txt"? [no] yes
Changed type of element "/vob/tools/jdk/THIRDPARTYLICENSEREADME.txt" ####
                                                              to "file".
```

After that the checkin succeeds:

```
$ ct ci -nc /vob/tools/jdk/THIRDPARTYLICENSEREADME.txt
Checked in "/vob/tools/jdk/THIRDPARTYLICENSEREADME.txt" version #########
                                                    "/main/mybranch/1".
```

dbid and the Raima database

This case starts with a build error:

```
    fetch cleartext view=joe vob=/vob/work dbid=0x101d - ##################
                                                    No permission match
```

Similar mentions of `dbid` may be found in other situations, for example, in system logs. They refer to the underlying database, used for both views and vobs. The first question is: what object is this about?

A first way, which may work, uses cleartool from the shell.

In case of any problems, one may use perl and the ***ClearCase::Argv*** module:

```
$ ct setview joe
$ cd /vob/work
$ ct dump $(ct find -a -ver '!created_since(now)' -print) | \
 egrep '^oid=.*dbid=.*\(0x101d\)'
bash: /usr/atria/bin/cleartool: Arg list too long
$ perl -MClearCase::Argv -e \
 '$c = new ClearCase::Argv({autochomp=>1,ipc=>1,stderr=>0});
 for($c->find([qw(-a -ver !created_since(now) -print)])->qx) {
 @v = grep/^oid=.*dbid=.*\(0x101d\)/, $c->dump($_)->qx;
 print "@v\n" if @v }'
oid=15009808.3bcd11de.965e.00:30:6e:4b:55:70 dbid=4125 (0x101d)
```

The `dbid` field is not directly accessible: it is only the identifier for the underlying Raima database.

There is one such record for every version of every element (and in fact for other objects as well, but we are looking for a cleartext container, thus for an element). In order to get all the existing versions in the vob, we need a pass-all query. *Not created since now* plays this role.

There are two problems with this:

- It will typically generate a huge output (and take long)
- Part of the versions (for example, the ones currently checked out) will not be accessible

To tackle these, one can use the following:

- ***ClearCase::Argv***: Perl deals better than a shell with lots of output, and the module presents it one item at a time to cleartool.
- The *ipc* mode: This uses one single cleartool invocation, thus optimizing performance.
- `stderr=>0`: Ignore the errors... The other option would be to run the `dump` commands in dedicated views, but this would be extremely slow. It might be considered in last resort.

The value printed by `ct dump` as `dbid` is given in decimal, with the value in parentheses being until 2003.06 the hexadecimal transcription.

This can be checked for a file element on an example. First, with 2003.06:

```
$ ct dump foo.txt | egrep '^oid'
oid=28003dd9.226a11dd.9bab.00:01:84:1e:63:a9 dbid=147591 (0x24087)
$ printf "%d\n" 0x24087 147591
```

Then with 7.0.1 (same version):

```
$ ct dump foo.txt | egrep '^oid='
oid=28003dd9.226a11dd.9bab.00:01:84:1e:63:a9 dbid=147591 (0xfe8b71e8)
```

With 7.0.1, the hexadecimal value is shared between all the versions, branches, and the element itself:

```
$ ct lsvtree -s apps | tail +2 | sed -e 's/\(.*\)/dump \1/' | \
 cleartool | egrep '^oid=.*dbid='
oid=520147ad.912211dc.8834.00:01:83:0a:15:19 dbid=145669 (0xfe8371e8)
oid=520147b1.912211dc.8834.00:01:83:0a:15:19 dbid=145670 (0xfe8371e8)
oid=520147b5.912211dc.8834.00:01:83:0a:15:19 dbid=145671 (0xfe8371e8)
oid=529147b9.912211dc.8834.00:01:83:0a:15:19 dbid=145672 (0xfe8371e8)
oid=533147c1.912211dc.8834.00:01:83:0a:15:19 dbid=145674 (0xfe8371e8)
oid=5310001f.a40411dc.8f90.00:01:83:0a:15:19 dbid=148344 (0xfe8371e8)
oid=cba8673c.74cb4bd3.a5ff.a8:81:b3:4c:f7:1a dbid=211770 (0xfe8371e8)
$ ct dump apps@@ | egrep '^oid=.*dbid='
oid=520147a5.912211dc.8834.00:01:83:0a:15:19 dbid=145667 (0xfe8371e8)
```

The hexadecimal value is common to all the elements in the vob!
We leave the topic of *dbids* on these findings, without a satisfactory solution for v7.0.

Protecting vobs: protectvob, vob_sidwalk, fix_prot

Here our target is to restrict the access to `/vob/secret` to one single group (sec); thus we first reprotect this vob from the previous group (`grp`) to the new one. To re-protect the vob (change owner, group, add/remove additional access groups), the `cleartool protectvob` command can be used:

```
$ ct protectvob -chown secadm -chgrp sec /filer01/vobstg/secret.vbs
This command affects the protection on your versioned object base.
While this command is running, access to the VOB will be limited.
If you have remote pools, you will have to run this command remotely.
```

In case of a vob having remote pools (for example, located on a NetApp storage, although this kind of setup is now outdated as seen earlier), the remote pools have to be re-protected separately.

For this purpose the *chown_pool* utility (from `/opt/rational/clearcase/etc`) can be useful:

```
$ chown_pool secadm.sec /filer01/vobstg/secret.vbs/s/sdft
$ chown_pool secadm.sec /filer01/vobstg/secret.vbs/d/ddft
$ chown_pool secadm.sec /filer01/vobstg/scret.vbs/c/cdft
```

The `protectvob` command requires *root* access:

```
$ cd /vob/secret
$ newgrp sec
$ id uid=31415(secadm) gid=117(sec)
```

```
$ ct find -a -ele '!created_since(now)' -print | \
 perl -MClearCase::Argv -ne \
 'BEGIN{$ct=ClearCase::Argv->new({autochomp=>1,ipc=>1})}
 chomp;$ct->protect([qw(-chgrp sec)],$_)->system
  if $ct->des([qw(-fmt %[group]p)],$_)->qx eq q(grp)'
```

Not bad, especially as it could be done by the vob owner (not root). However, this affects only elements.

Then on the vob server, check what was left:

```
~> /opt/rational/clearcase/etc/utils/vob_siddump /vob/secret /tmp/sec.map
VOB Tag: /vob/secret (vobhost:/vobstg/secret.vbs)
Meta-type "directory element" ... 208 object(s)
Meta-type "directory version" ... 746 object(s)
Meta-type "tree element" ... 0 object(s)
Meta-type "element type" ... 13 object(s)
Meta-type "file element" ... 2622 object(s)
Meta-type "derived object" ... 0 object(s)
Meta-type "derived object version" ... 0 object(s)
Meta-type "version" ... 5334 object(s)
Meta-type "symbolic link" ... 0 object(s)
Meta-type "hyperlink" ... 0 object(s)
Meta-type "branch" ... 2830 object(s)
Meta-type "pool" ... 3 object(s)
Meta-type "branch type" ... 1 object(s)
Meta-type "attribute type" ... 9 object(s)
Meta-type "hyperlink type" ... 9 object(s)
Meta-type "trigger type" ... 0 object(s)
Meta-type "replica type" ... 1 object(s)
Meta-type "label type" ... 40 object(s)
Meta-type "replica" ... 2 object(s)
Meta-type "activity type" ... 0 object(s)
Meta-type "activity" ... 0 object(s)
Meta-type "state type" ... 0 object(s)
Meta-type "state" ... 0 object(s)
Meta-type "role" ... 0 object(s)
Meta-type "user" ... 0 object(s)
Meta-type "baseline" ... 0 object(s)
Meta-type "domain" ... 0 object(s)
Total number of objects found: 11818

Successfully processed VOB "/vob/secret".
~> ll /tmp/sec.map
-rw-r--r-- 1 secadm sec 521 Aug 5 17:05 /tmp/sec.map
~> cat /tmp/sec.map
anis/secadm,USER,UNIX:UID-31415,IGNORE,,,43
anis/joe,USER,UNIX:UID-79521,IGNORE,,,209
anis/bill,USER,UNIX:UID-58228,IGNORE,,,61
anis/jane,USER,UNIX:UID-80911,IGNORE,,,2490
```

```
anis/jill,USER,UNIX:UID-79707,IGNORE,,,35
anis/jack,USER,UNIX:UID-7080,IGNORE,,,62
anis/sam,USER,UNIX:UID-79706,IGNORE,,,4
anis/joan,USER,UNIX:UID-87669,IGNORE,,,9
anis/mark,USER,UNIX:UID-18543,IGNORE,,,1
anis/sec,GROUP,UNIX:GID-117,IGNORE,,,2830
anis/grp,GROUP,UNIX:GID-100,IGNORE,,,84
~> echo "anis/grp,GROUP,UNIX:GID-100,anis/sec,GROUP,UNIX:GID-117" \
 > /tmp/newmap
~> /opt/rational/clearcase/etc/utils/vob_sidwalk -execute \
 -map /tmp/newmap /vob/secret /tmp/map2
vob_sidwalk: Error: No permission to perform operation "modify vob".
vob_sidwalk: Error: Must be one of: root
vob_sidwalk: Error: Insufficient permission to fix protection: ##########
                                  error detected by ClearCase subsystem
```

Despite what the man page says (which was actually a ***documentation bug***), you have to be root to run `vob_sidwalk`, at least with the `-execute` flag!

We had protected so far (using `find/protect`) all the elements. There remained 84 objects. What about types?

```
$ for k in attype brtype eltype hltype lbtype trtype; \
 do ct lstype -fmt "protect -chgrp epc %Km:%n\n" -kind $k; done | \
  cleartool
```

This solved the problems for all the non-locked types.
A next use of `vob_sidwalk` showed a rest of 29 objects, 17 of which were locked types. Of those, seven could be files in the vob storage, and the remainder was unknown.

The `vob_sidwalk` command (with the same map) was run as root, and the result showed (`sam` lost?):

```
~> cat /tmp/epc.map3
anis/secadm,USER,UNIX:UID-31415,IGNORE,,,43
anis/joe,USER,UNIX:UID-79521,IGNORE,,,209
anis/bill,USER,UNIX:UID-58228,IGNORE,,,61
anis/jane,USER,UNIX:UID-80911,IGNORE,,,2490
anis/jill,USER,UNIX:UID-79707,IGNORE,,,35
anis/jack,USER,UNIX:UID-7080,IGNORE,,,62
Account Unknown,USER,UNIX:UID-79706,IGNORE,,,4
anis/joan,USER,UNIX:UID-87669,IGNORE,,,9
anis/mark,USER,UNIX:UID-18543,IGNORE,,,1

anis/sec,GROUP,UNIX:GID-117,IGNORE,,,2914
```

But the `tmp` source containers (left over from failing to create a new version: these are the new containers, which should have been renamed after the previous one was removed) were not protected, nor any of the files in the vob storage directory:

```
~> ll /vobstg/secret.vbs/s/sdft/13/10/tmp_8436.2
-rw-rw-rw- 1 secadm grp 18493016 Dec 15 2009 ############################
                                /vobstg/secret.vbs/s/sdft/13/10/tmp_8436.2
~> find /vobstg/secret.vbs -group grp | wc -l
7
~> find /vobstg/secret.vbs -group eei-atusers
/vobstg/secret.vbs/.hostname
/vobstg/secret.vbs/s/sdft/pool_id
/vobstg/secret.vbs/s/sdft/13/10/tmp_8436.2
/vobstg/secret.vbs/s/sdft/13/10/tmp_2240.1
/vobstg/secret.vbs/s/sdft/0/3d/tmp_10020.2
/vobstg/secret.vbs/d/ddft/pool_id
/vobstg/secret.vbs/c/cdft/pool_id
```

The next attempt is to use `fix_prot` (as *root*):

```
secret.vbs# fix_prot -force -r -root -chown secadm \
 -chgrp sec /vobstg/secret.vbs
CAUTION! This program reprotects every file and directory in a
storage directory tree and should be used only when the protection
has been damaged (e.g., through the process of copying the tree or by
direct manipulation through a tool like the File Manager/Explorer).
Protecting "/vobstg/secret.vbs/.identity"...
Protecting "/vobstg/secret.vbs/s/sdft/0/0"...
...
```

Note that the `-root` option tells the tool that the argument is the root of a vob storage. Since we do not use a `-chmod` option, we do not need to run the command in two phases.

This does reprotect all the remaining files, but also affects the `.identity` directory (in the vob storage), in a way inconsistent with the output of the `ct des` command:

```
$ ct des vob:/vob/secret
versioned object base "/vobs/swdi/tools"
...
 VOB ownership:
  owner anis/secadm
  group anis/sec
 Additional groups:
  group anis/root
$ ls -l /vobstg/secret.vbs/.identity
total 0
-r----s--- 1 secadm sec 0 Aug 13 19:08 gid
-r-S------ 1 secadm sec 0 Aug 13 19:08 uid
```

One must restore (or remove, depending on the need, but to achieve consistency) the additional group:

```
$ ct protectvob -add root /vob/secret
$ ls -l /vobstg/secret.vbs/.identity
total 0
-r----s--- 1 secadm sec  0 Aug 13 19:08 gid
-r----s--- 1 secadm root 0 Aug 16 16:31 group.0
-r-S------ 1 secadm sec  0 Aug 13 19:08 uid
```

Alternatively, the changes to the `.identity` directory can be done manually (just using `touch`, `chown` and `chmod` command), and this also works out fine.

As a conclusion of this section, let's mention the technote about the ***summary of the protection commands and utilities***.

Cleaning lost+found

Let's first remind what the role of the `lost+found` directory in every vob is:

```
$ ct rmelem -f d1
cleartool: Warning: Object "bar" no longer referenced.
cleartool: Warning: Moving object to vob lost+found directory as ########
                                    "bar.1f89b0fe6e5511df8b600001842becee".
Removed element "d1".
```

A directory was removed: this resulted in the objects it contained losing their last reference in the file system.

The objects were kept and made accessible via the `lost+found` directory. In order to avoid name clashes with other objects, their name was made unique, by appending their oid.

Note that some other scenarios might lead to the same result as destroying a directory element, for example, unchecking a directory right after creating an element inside it.

Preserving file objects in `lost+found` makes it possible to restore them to any directory found suitable—either an existing one or created for this purpose. The directory must of course be checked out preliminarily, and checked in afterwards.

```
$ ct mv /vob/foo/lost+found/bar.1f89b0fe6e5511df8b600001842becee bar
```

Can we detect an event such as the one produced above?

```
$ ct lshis -minor -since 21:42 -all
--09-26T21:47 marc  destroy element in versioned object base "/vob/foo"
 "Destroyed element "/vob/foo/d1"."
--09-26T21:47 marc  remove name from directory version ##################
                                                    "/vob/foo@@/main/mg/3"
 "Uncataloged directory element "d1"."
```

The `rmelem` events are kept forever (by default) in `vob_scrubber_params`.
The `rmname` ones are scrubbed at the latest after 14 days.

In order to be able to restore valuable data from `lost+found`, one needs to detect it. This implies that `lost+found` is not cluttered with lots of needless files, and thus sets a rationale for cleaning it up: keep a simple background for cases in which complexity will emerge anyway.

The main tool for the `lost+found` cleanup is `rmelem`; however, the procedure grows complex if it is not practiced regularly.

Do not attempt to remove the contents of `lost+found` recursively! The fact that a directory is not accessible from anywhere anymore doesn't mean that its contents are not: you might remove something valuable and still reference it from elsewhere in you vob.

Also, you may remove only what you master locally.

Some commands for the vob owner:

First, `uncheckout` the checked out versions. You have to do it using the view in which the objects are checked out, assuming this view is *dynamic*, still healthy, and reachable (both host and storage).

However, it is possible that the view cannot access the version anymore, or the `lost+found` directory itself.

Therefore, use the *oid* and the vob tag:

```
$ ct lsvtree * 2>/dev/null | perl -MClearCase::Argv -ne \
 'BEGIN{$ct=new ClearCase::Argv({autochomp=>1,ipc=>1,stderr=>0});
   $tag=$ct->des([qw(-s)],"vob:.")->qx}
 if(m%^(.*)\@\@/.*/CHECKEDOUT view "(.*)"$%){
 $o=$ct->des([qw(-fmt %On)], $1)->qx;
 $ct->argv(qw(setview -exec), "cleartool unco -rm oid:$o\@$tag",
  $2)->system}'
```

This will as such not affect names starting with a dot. Treat them with care, as ".*" would match the ".." parent directory!

It will also skip checked out versions bearing a label. You need to remove the label first anyway, before using `rmelem`.

Then, use (non recursive!):

```
$ ct rmelem -f *
```

Process there as well names starting with a dot specifically (like,".a*")
You may repeat either step as many times as necessary (as long as there is something to remove).

Summary

Strict role assignment leads to incompetence. Administrators are not experts, and will not grow such if they only deal with routine administration and never look out of their box. All the same, users will break things if they don't attempt to understand what they do, and do not feel responsible for the environment (somebody else's job!) They will break fragile things: robustness is a quality one acquires by evolutionary mechanisms. Optimizing the processes will only take place if the various actors understand each other enough to criticize what others do and suggest meaningful alternatives; if all respect each other enough to listen to such critique and suggestions.

What this chapter reviewed is a long list of practical spots where users often face their administrators. A basic knowledge, but moreover the confidence that the information is there to be found if needed, are necessary to make this encounter a fruitful collaboration and not a frustrating confrontation. As we showed, this doesn't go without work, and is therefore subject to tradeoffs. This works both ways: users might want to avoid spending too much time and effort on tasks which, they feel fall beyond their attributions, but on the other hand, they have a different perspective, and thus different priorities than administrators.

11
MultiSite Administration

Following up on the previous chapter, understanding MultiSite administration cannot hurt. On the contrary, it helps lifting unreasonable expectations: people with an experience of centralized or disconnected systems may assume synchronous behavior (that the data is *in sync*) without really needing it or without considering the implied cost (in terms of waiting). **Asynchrony** in ClearCase MultiSite actually protects the user from depending upon remote (thus both slow and often out-of-reach) resources and events. A good understanding of the difference between latency and bandwidth is a valuable asset in a world where the latter is both well advertised and increasing regularly, and the former is often overlooked and largely incompressible.

Two Open Source version control tools—Mercurial and git—have recently made distributed systems popular. We shall remember that ClearCase was a precursor in this domain as well, even if it may surprise most of its users—mainly those of UCM or of the various web interfaces and Eclipse plugins, who have been driven to see ClearCase as a centralized system. MultiSite comes with an off-the-shelf configuration that favors fairness between all replicas but which is dramatically not scalable. We shall show how to ensure **scalability** without jeopardizing **fairness**: the very value that distinguishes between centralized and distributed systems.

This chapter will be structured along three parts:

- A brief top-down view, with considerations on replica properties and connectivity
- A presentation of two simple but effective enhancement proposals
- A bottom-up review of some troubleshooting cases

Setting up the scenery

MultiSite is implemented with the **multitool** command and the **shipping_server** program. It is somehow unfortunate that multitool is distinct from **cleartool**, although they share a number of operations, which has been growing over time. There also came from the beginning a couple of Perl scripts—`sync_export_list` and `sync_receive`—using a common library `MScommon.pl`. The two scripts are intended for use from the scheduler (`sync_receive` is also suitable for use as receipt handler: see below).

A minor note:

The scripts come, in fact, each as a pair—the real Perl script being found in a `.bat` file with a clever preamble suitable for interpretation by Perl as a dummy array declaration, and by the Windows command as an instruction to pass the file to the bundled (under Windows) `ccperl`. The other files without extension— `sync_export_list` and `sync_receive`—are UNIX shell scripts passing the former to the bundled (under UNIX) Perl.

t some point in time, there came a new family of scripts, together with a `syncmgr_server` program. This new family possibly had some advantages, but it was backwards incompatible and required a synchronous switch on all the sites. This condition proved unacceptable in all the contexts we have met (because of subcontractors sites for instance), so that we'll simply ignore the sync manager and build our configuration on top of the original scripts, to which we made backwards compatible enhancements.

Every vob may be **replicated** to as many sites as needed. The set of replicas builds up a **vob family**—a *replica* on every site. Both kinds of entities have their typed object in the database. In every vob database, there is one vob object and as many replica objects as there are physical replicas, imported on the various sites. Every vob thus maintains a representation of itself and of its siblings.

Events produced on one site are journaled locally into *oplogs*. These oplogs are collected on a schedule, and exported as sync packets for the other replicas. They are then shipped, possibly routed, and finally imported on their destination host. Every replica keeps an *epoch* number of the last oplog it has exported to every other replica, and of the last one it has imported from them. This builds up an *epoch table*: every replica maintains its own. What is important to understand is that at no point in time is there any guarantee that these tables match, and thus that the replicas are *in sync*. On the contrary, it is a safe assumption that there are always some packets on their way, which explains small discrepancies. Obviously, these discrepancies should not grow: the epoch numbers, on the contrary, should grow on every site.

```
$ pwd
/vob/a
$ mt lsrep -s
s1
s2
$ ct des -fmt "%[replica_name]p\n" vob:.
s1
$ ct des -fmt "%[replica_host]p\n" replica:s1
beyond.lookingglass.uk
$ mt lsepoch s1
oid:d5249fcb.011611db.90b6.00:01:83:10:fe:84=32783 (s1)
oid:6ba49d09.011611db.8aa1.00:01:83:10:fe:84=78    (s2)
```

What the table shows here is, on site `s1`, its own *row*, made of lines with the epoch number for every registered replica. For itself this gives the best authorized information: all the oplogs that were effectively created here (`32783`). For the other replicas, this gives the best knowledge we have locally, that is, the number of the last oplog imported from there (`78`).

```
$ mt lsepoch s2
oid:d5249fcb.011611db.90b6.00:01:83:10:fe:84=32781  (s1)
oid:6ba49d09.011611db.8aa1.00:01:83:10:fe:84=78     (s2)
```

If we now look at the row for another replica (`s2`), the values we see are:

- On the line for our own replica, the oplog number (`32781`) we, as the site hosting the s1 replica, last exported to s2, whether or not it could be successfully imported there (even the shipment is unsure).
- On the lines for any other replica, again the best knowledge we have locally: the last oplog known to have been exported from there—`78`. This information may have been received indirectly, that is, from a third replica.

This mechanism shows that the replication is an all or nothing issue on a per vob basis: one cannot avoid sending certain data, confidential for example. Oplogs must be imported in order (serialized): if an oplog is missing, later oplogs cannot be imported.

Permissions preserving

One apparent exception to the previous rule is protection events, which get filtered based on a property of replicas (a creation option that may be changed afterwards). Originally, the choice was only between preserving and non-preserving (and this encompassed both permissions and identities), but later a separate permissions-preserving option was added. The latter is the obvious correct option in most cases (in which NIS or active directory are not shared between sites): it is a good idea to transfer the events that set the execution right to a program or the write permission for group to a directory! For protections events to get transmitted, both the exporting and importing replicas must be preserving of the related flavor.

However, note that this filtering doesn't affect the epoch numbers.

Connectivity

Some multitool commands have `-actual` options—`lsepoch` and `chepoch`. These commands are exceptionally synchronous. They require direct connectivity to the remote hosts and do not usually work through firewalls.

Another command that attempts to bypass the standard replication model is `reqmaster`.

These commands obviously break the fairness between replicas. They may be handy interactively: don't use them in scripts or build a process upon using them!

Note in any case that mastership changes are normal events (whichever command, `reqmaster` or `chmaster`, was used to create them), and they are therefore shipped and imported only after any previous oplog has been successfully imported. While a mastership is in transit, nobody holds it.

This alone shows that transferring mastership should be avoided for any other purpose than administrative (for example, sending the mastership of the replica object after it was successfully imported so that it would be self-mastered, and before deleting a replica).

The rules of thumb for sound multisite development processes are simple (see *Chapter 5*, *MultiSite*), yet seldom followed:

- No mastership transfer
- No remote commands

Especially, one should avoid merging back branches to integration or other upper branches, which obviously may only be mastered on one site.

Configuration

The main problem of the initial MultiSite setup is one of combinatorial explosion. The simplest is to describe it with an example scenario.

Let's take a vob:`/vob/a`, with replicas `s1` and `s2` (we'll designate the replicas and the sites with the same names).

Suppose we create from site `s1` a new replica `s3`, to be shipped and imported to site `s3`. Now, `s2` will start creating and shipping to their destinations, sync packets for both `s1` and `s3`.

Suppose now we create from site `s3` a replica `s4`. On the same schedule as previously, `s2` will now create and ship packets for `s1`, `s3`, and `s4`. And so forth. And every replica will receive packets from every other. All the three phases: export, shipping, import are thus impacted, and for all the existing replicas!

This is clearly suboptimal: there is a lot of redundancy in the contents of the packets. In fact, if we do produce packets to all replicas, they should contain the same data—the only difference coming from the fact that they are produced at slightly different times.

The same is of course true of received packets; they will mostly contain the same oplogs, which will thus get imported once and skipped while importing the following packets— again some useless load.

Finally, the network will suffer artificial congestion, the packets typically taking the same routes for a large part of their paths.

The naive solution is to limit on every site the list of synchronized replicas, and thus to produce packets only for those. This has undesirable side effects:

- It requires manual maintenance of the lists and coordination between the sites
- It stops updating certain parts of the epoch tables, leading to the fact that all replicas are not equal anymore: in case of need, one might not be able to use certain replicas in order to re-create one, which one way or another became corrupted

These first-level issues may be fixed in ways which further break the fairness of the original model, but let's examine our alternatives.

Export

First, let's note that the `sync_export_list` script does not take advantage of the existing ability of `mt syncreplica -export` to create a single packet for multiple replicas. This is the first enhancement we make, and we make it to this script: instead of creating one packet for every replica in the argument list (including the `-all` case), create a common one. This will be reasonable if the replicas are all about at the same epoch level: otherwise, the packet will contain useless oplogs for the replicas nearly up-to-date, as driven by the needs of the most outdated one.

We'll return to exporting oplogs to multiple replicas in the further section, while considering the *hub* function.

In any case we keep the original script and rename our customized version as `sync_export_list_hub`. This leaves the original version for manual use (and backup), and protects our version from being overwritten during an upgrade.

Shipping/routing

The next question is this of shipping: the default behavior of the shipping server is to dispatch the packet to its destinations. We do not change this, but only rely on the `ROUTE` settings in the `shipping.conf` file: if several destinations share a route, the packets will be sent only once and dispatched at the point where the routes split. We consider now setting up a *hub* (hence the suffix appended to the script name). Let's call our hub host *proxima*. The `shipping.conf` files on all the vob servers other than the hub would then contain a single default route:

```
ROUTE proxima -default
```

Shipping servers are needed to cross firewalls, at least if one intends to restrict the range of open ports. The setting in `shipping.conf` allowing us to define a port range must be matched with one in the `albd_server` environment, in the `clearcase` startup script:

```
$ grep 'M.*PORT' /var/adm/rational/clearcase/config/shipping.conf
CLEARCASE_MIN_PORT    49152
CLEARCASE_MAX_PORT    65535

$ grep 'M.*PORT' /opt/rational/clearcase/etc/clearcase
CLEARCASE_MIN_PORT=49152
CLEARCASE_MAX_PORT=65535
export CLEARCASE_MIN_PORT
export CLEARCASE_MAX_PORT
```

This affects all ClearCase processes on the host.

It is obvious that for crossing firewalls, as well as on long-distance connections, the bandwidth cannot be wasted in shipping redundant data.

The names of the packets produced by default by `mt syncreplica -export`, invoked by the `sync_export_list` script, contain the name of the source replicas but not the vob tags. The rationale is that the same vobs may be tagged differently in different sites. Many administrators work around the resulting inconvenience by duplicating the vob tag in the replica names. We note that this practice suffers from the effect described in the previous argument (the vob tags may be different). Next that it creates a pressure to rename the replicas if one re-tags the vobs; finally that it requires some conventions to deal with the case of structured tags (the path separator not being a legal character in replica names), and that longer names have a price in terms of clarity of the outputs. For all these reasons, we cannot recommend this practice and prefer to trade for using a common site name for all replicas.
The vob uuid is anyway found in the packets and easily read from there with the `mt lspacket` command:

```
$ mt lspacket sync_* | grep family | awk '{print $5}' | sort -u
c6052140.08e511d5.b124.00:01:80:e6:b1:a0

$ ct lsvob -fam c6052140.08e511d5.b124.00:01:80:e6:b1:a0
* /vob/foo /vobstg/foo.vbs public (replicated)
```

Where the command is not available (nor the registry to map it to the local tag), i.e., on shipping servers on the way, the packet bears a transient name. We should note that this transient name is not normally exposed, as it is not recorded in the shipping logs, where it is replaced by the original shipping packet name. It is only in the exceptional cases, when the shipping fails, that one can see the shipping packets with names such as `sh_d_...` in the outgoing bay:

```
[shiphost]$ pwd
/opt/rational/clearcase/shipping/ms_ship
[shiphost]$ ll
-r--r--r-- 1 root root 372 Oct 10 14:16 sh_d_99185
-rw-rw-rw- 1 root root 597 Oct 10 14:16 ##################################
                              sh_o_sync_s1_2010-10-10T14.16.09+03.00_9517

[shiphost]$ scp sh_d_99185 vobsrv:/tmp
[vobsrv]$ mt lspacket /tmp/sh_d_99185
Packet is: ######################################################################
         /var/adm/rational/clearcase/shipping/ms_ship/outgoing/sh_d_99185
Packet type: Update
Packet fragment: 1 of 1
VOB family identifier is: eccf73e6.e44e11dc.9acd.00:16:35:7f:04:52
Comment supplied at packet creation is:
Packet intended for the following targets:
 s1 [ local to this network ] tag: /vob/foo
```

And from the shipping order, we can see both the original and the transient packet names:

```
$ cat sh_o_sync_s1_2010-10-10T14.16.09+03.00_9517
# ClearCase MultiSite Shipping Order
# Version 1.0
%IDENTIFIER d65843f5.d4a311df.894b.00:12:3f:93:d3:e9
%CREATION-DATE 1286738169 2010-10-10T14:16:09+03:00
%EXPIRATION-DATE 1287947770 2010-10-24T14:16:10+03:00
%ORIGINAL-DATA-PATH "/opt/rational/clearcase/shipping/ms_ship/outgoing/ #
                                  sync_s1_2010-10-10T14.16.09+03.00_9517"
%LOCAL-DATA-PATH "/opt/rational/clearcase/shipping/ms_ship/outgoing/ ####
                                                          sh_d_99185"
%DESTINATIONS
 s1
%ARRIVAL-ROUTE
 s2 1286738169 2010-10-10T14:16:09+03:00
%DELIVERIES
%FAILED-ATTEMPTS
 R 1277634571 2010-06-27T13:29:31+03:00 s2 s1
%NOTIFICATION-ADDRESSES
%COMMENT
```

Now, we must admit that when grepping remote shipping logs, one has nothing more than the packet names at one's disposal. One can consider implementing a change to `sync_export_list_hub` to use the `-out` option to add a vob identification to the packet name, and use `mkorder` to ship the packet. Taking into account the argument of the possible disparity of vob tags, one might use instead of the tag a special `packet_name` attribute: it would be attached to the vob object and thus replicated, which would guarantee a common value on all sites. This would combine the advantages of the various solutions: save the replica name from having to depend on the vob tag, and publish a shared and stable name for the packets.

Import

Let's now consider the hub. What behavior would be optimal there? What should it do on receiving packets?

Our goal is to reduce the number of packets received by the destinations. The hub will receive packets from multiple origins. How can it combine them so that the number of packets received at any final destination does not grow linearly with the number of replicas? The solution is simple: the hub must itself host a replica, import the packets, refrain from forwarding them to their other destinations (again, altering the default behavior, which is to forward packets that are not for the local replica), and re-export packets containing the compound oplogs on its own schedule.

This implements a version of an *import/export hub*, as opposed to this of a *forwarding hub*, already available in the default `shipping_server`.

More precisely, our changes affect two steps: first, when the script is used as a **proactive receipt handler** (see below), we set a flag `skip_ship` (here is an excerpt from the diffs):

```
+ my $skip_ship = 0;
 if ($CFG::MScfg_proactive_receipt_handler && $CFG::receipt_handler) {
 $cmd = qq/$CFG::MultiTool syncreplica -import -receive -invob $tag/;
+ if ($actual_shiporder and ($sclass ne 'express')) {
+  dbgprint "Do not ship this: $actual_shiporder,\nif the import #####
                                                        is successful. ".
+ "It would be redundant with the sync packet produced later by ######
                                                              the hub\n";
+  $skip_ship = 1;
+ }
... [ other branch not altered ]
```

Then, on import success, and with the flag set, we delete the packet and shipping order to prevent the forwarding (with this second excerpt, we nearly showed all the changes we made!):

```
+ if ($skip_ship) {
+  dbgprint "About to delete $pkt and $actual_shiporder\n";
+  if (open SHORDER, "<$actual_shiporder") {
+   my ($line, $pkt) = ();
+   foreach $line (<SHORDER>) {
+     next if $line =~ /^\s*\#.*/; # skip comment lines
+      if ($line =~ /^\%LOCAL-DATA-PATH \"(.*)\"/) {
+       $pkt = $1;
+      last;
+   }
+  }
+  close SHORDER;
+  if ($pkt) {
+   unlink $pkt, $actual_shiporder;
+   dbgprint "Deleted $pkt and $actual_shiporder\n";
+   $actual_shiporder = 0;
... [ only error reports in the other branches ]
```

Note here that the packets first arrive in the *outgoing* bay, and are only hard linked to the *incoming* one: this is to support the case when the packets would have several destinations and would thus have to be forwarded after being imported. This sets a requirement that the two bays are co-located on the same file system. Note also how the handling of compressed packets gets simplified now (since v7.0), as the `syncreplica` command supports directly the `-compress` switch! Previously, this had to be handled by the `sync_export_list` script.

We must, however, guard against some cases. Before the hub replica itself has been imported, packets must be forwarded to their destination. The same must happen if the local import fails: an error on the hub should not block the replication.

Then, when exporting changes for all the other replicas, we shall often meet the case when the changes came in fact exclusively from one among them (at least for the current export, if the schedule is tight enough). Clearly, there is no point in re-exporting to this replica its own changes!

This is handled in our `sync_export_list_hub` script by the following excerpt:

```
my %remepo = GetReplicaEpochs($vob, $sib);
my $skip = 1;
my $key;
foreach $key (keys %remepo) {
  if ($remepo{$key} < $locepo{$key}) {
    $skip = 0;
    last;
  }
}
next if $skip;
```

We do not attempt anything fancier than skipping a destination if there is nothing new for its replica.

Next, we must be prepared to distribute the hub function among several hosts: one single server holding a replica for all known vobs would soon become a bottleneck. This may however require a dedicated shipping server performing intelligent dispatching on the hub site. The problem here is that the `ROUTE` settings on the various sites are shared between all vobs, without a syntax to discriminate between them. On the other hand, one cannot mandate that the same vobs would be co-located on the same vob servers on all sites. It is thus best if one host may be designated on the hub site to act as the next hop for all routes using the hub. This host will need to refrain from dispatching the packets, before their import has been attempted on the host holding the replica within the hub site.

Finally, we must consider redundancy: one must be able to ship packets via different routes, at least in certain cases. If this can be restricted to certain destinations, the strategy depicted so far is fully sufficient: one only needs to add the route settings to the `shipping.conf`. Supporting this on a per vob basis however requires further adjustments to the scripts, or maybe the use of yet another dedicated shipping server, this time to dispatch packets before shipping them to remote sites.

Receipt handler

The export phase must be scheduled.

The import phase, on the contrary, is better handled as soon as the packet arrives: there is no advantage to wait. This is by the way true as well on a shipping server for the purpose of purely dispatching or forwarding packets. This doesn't preclude the use of a scheduled job to process such packets that, for one reason or another, would have been dropped by the receipt handler.

The receipt handler is invoked by `shipping_server`. One may invoke different handlers per storage class, but it is not obvious how to make use of this feature at its best. We shall assume here a `-default` handler. In `shipping.conf`:

```
RECEIPT-HANDLER -default /opt/rational/clearcase/config/scheduler/ ###
                                                    tasks/sync_receive_hub
```

We also add a `MSimport_export.conf` file under `/var/adm/rational/clearcase/config`, and there we have:

```
proactive_receipt_handler = 1
```

This allows the receipt handler to process other packets than the one just received, if they are for the same vob. This lowers the probability of dropping packets because of missing oplogs.

Shipping server

There is a special ClearCase shipping server installation on UNIX. This one is lighter, and doesn't, for instance, require an access to licenses. It comes therefore neither with *cl*eartool nor multitool. However, the ClearCase *scheduler* functionality is enabled on the shipping server, and indeed, it may be very useful to schedule the following tasks:

- `sync_export_list -poll` (*Daily MultiSite Shipping Poll* job) to poll all the packets in the outgoing bay and shipping them to the next hop
- `sync_receive` (*Daily MultiSite Receive* job), the functionality of which is reduced on a shipping server to moving received packets from the incoming to the outgoing bay

Note that the use of a receipt handler is also possible, and even encouraged. We even propose a trivial receipt handler adapted to the needs of a shipping server. This will forward packets until the maximum number of ports in the firewall range is exhausted. It is thus meant to coexist with a scheduler job, which will process the packets dropped under bursts of activity.

Setting up the scheduler on a shipping server

As we already mentioned, the shipping server installation does not include cleartool. Hence, the scheduler cannot be set up locally using the `cleartool sched` command as described in *Chapter 10, Administrative Concerns*. One can however read the shipping server schedule from a full ClearCase client, using the following command:

```
$ ct sched -host shipping_server_hostname -get -sched
```

One won't be able to change it though, until the scheduler ACLs are set up adequately on the shipping server. The default ACL settings on any ClearCase host are:

```
$ ct sched -get -acl
# Scheduler ACL:
Everyone: Read
```

This grants write permissions only to the local `root` account, and must thus be modified locally first:

```
$ ct sched -edit -acl
```

One can set them as desired, for example, to:

```
# Scheduler ACL:
Everyone: Read
User:<unknown>/joe Full
```

One can then copy the `/var/adm/rational/clearcase/scheduler/db` file from the local host to the shipping server, and with it goes the ACL just set.
After that the shipping server scheduler can be edited remotely (as user `joe`):

```
$ ct sched -edit -host shipping_server_hostname
```

Monitoring

Because `mt lsepoch -actual` won't work through firewalls, one can enable remote hosts replica monitoring with the help of the ***repoch*** script.

Client side (remote host)

The `repoch` script running on the remote host creates a replica monitoring report containing the current epoch number of the replicas hosted on that site and sends the report to the other site by creating MultiSite shipping order for it.

The setup on the client site includes the following steps:

1. Copy `repoch` to `/var/adm/rational/clearcase/scheduler/tasks` directory (make sure it has execution permission for everyone)
2. Edit `/var/adm/rational/clearcase/scheduler/tasks/task_registry` and add the following to the end of it:

```
Task.Begin
Task.Id: 102
Task.Name: "Replica monitoring"
Task.Pathname: repoch
Task.End
```

3. Add a new job to the cleartool scheduler (`ct sched -edit`):

```
Job.Begin
Job.Name: "Daily Replica Monitoring Poll"
Job.Description.Begin:
Send replica monitoring log to the next host.
Job.Description.End:
Job.Schedule.Daily.Frequency: 1
Job.Schedule.FirstStartTime: 00:00:00
Job.Schedule.StartTimeRestartFrequency: 04:00:00
Job.DeleteWhenCompleted: FALSE
Job.Task: 102
Job.Args: --dest destination_host_name
Job.NotifyInfo.OnEvents: JobEndFail
Job.NotifyInfo.Using: email
Job.NotifyInfo.Recipients: root
Job.End
```

Server side (local host)

The local host receives replicas monitoring reports from the remote host using the MultiSite shipping infrastructure and makes them available to the standard `getlog` mechanism. The setup on the server site includes the following steps. We took this in Chapter 10 as an example of *scheduler* configuration, and shall not reproduce the details here.

1. Create a simple shell script (like below) for the purpose of periodically moving the received `repoch` logs from the incoming bay:

```
#!/bin/bash
mv /usr/atria/shipping/ms_ship/incoming/*.repoch* /tmp
```

2. Name it `repoch_mv.sh` and place it in the `/var/adm/rational/clearcase/scheduler/tasks` directory.
3. Add a new scheduler *task* using the `repoch_mv.sh` script.

4. Add the corresponding job invoking the task periodically.
5. Add a new *repoch* entry to the log database named `/opt/rational/clearcase/config/services/log_classes.db`, allowing access to the collected logs with `ct getlog repoch`.

Troubleshooting

Here we will give a few examples of troubleshooting sessions, with the hope of demonstrating routines and tools, which apply to other cases as well.

Missing oplogs

A site may fail to import some packets because they depend on previous ones not found in any available packets. The cure is to generate again the missing oplogs, and to ship and import the packets. For this, one must use the `chepoch` command to set the epochs back in time to the situation required to generate the oplogs, and use `syncreplica` to recreate the missing packets. The appropriate values match the actual situation on the destination site, thus for the destination replica, the epoch number before the missing oplog. Note however that one may also have to change the oplogs on more than one line of the row: if one forgets some, the import will fail and tell which. The information about the epoch values at the various export times is actually also available in the history of the replica object (for the remote replica): see the section titled *History of exports*. This is what the `recoverpacket` command uses.

Just resetting the oplogs may however result in a huge amount of data right away, with no guarantee that it will be directly importable.
We recommend taking a more careful approach:

- Exclude the destination from the scheduled synchronization; this involves using a list of replicas as job argument, whether or not one does it by default.
- Reset the epoch tentatively.
- Send one packet (with multitool syncreplica -export `-max 50k -lim 1`) and record the new epoch value.
- Set the epochs back, so that the avalanche doesn't start, even if the scheduler is reset to its normal state, for example, to synchronize other replicas of the same vob.
- Try to import the packet created and shipped, and record the error if any.
- If the import succeeds, then the epochs will grow, and you may set them to the new value. Note, however, that existing packets in the incoming bay on the destination will become importable at some stage, so there's no need to send all the packets again if only few are needed.

- If the import failed, fix the condition and try again.

Let's illustrate the described approach with an example:

```
$ ct getlog -last 100 sync_import
multitool: Error: Sync. packet /opt/rational/clearcase/shipping/ms_ship/
  incoming/sync_s2_2010-08-04T10.44.12+05.30_23515 was not applied to VOB
  /vobstg/foo.vbs
 - packet depends on changes not yet received
Packet requires changes up to 1369900; VOB has only 1369884 from ########
                                                              replica: s2
...
multitool: Error: Sync. packet /opt/rational/clearcase/shipping/ms_ship/
  incoming/sync_s2_2010-08-04T11.16.19+05.30_1256 was not applied to VOB
  /vobstg/foo.vbs
 - packet depends on changes not yet received
Packet requires changes up to 1482751; VOB has only 1369884 from ########
                                                              replica: s2
```

On the `s1` site, the epoch table looks as follows:

```
$ mt lsepoch -invob /vob/foo
For VOB replica "/vob/foo":
Oplog IDs for row "s1" (@ v1.uk):
oid:3b56807b.489a11de.9517.00:21:5e:40:81:8c=3493 (s1)
oid:76d45938.f99a11dd.b047.00:02:c4:65:3f:4c=1369884 (s2)
Oplog IDs for row "s2" (@ v2.uk):
oid:3b56807b.489a11de.9517.00:21:5e:40:81:8c=3493 (s1)
oid:76d45938.f99a11dd.b047.00:02:c4:65:3f:4c=1369884 (s2)
```

So, it looks like just a tiny missing packet (the missing epoch numbers from `1369884` to `1369900`) has resulted in a huge number of non-imported sync packet in the incoming bay, with epoch numbers up to `1482751` (and actually above that).

Now consider the `s2` site:

```
$ mt lsepoch -invob /vob/foo
For VOB replica "/vob/foo":
Oplog IDs for row "s1" (@ v1.uk):
oid:3b56807b.489a11de.9517.00:21:5e:40:81:8c=3493 (s1)
oid:76d45938.f99a11dd.b047.00:02:c4:65:3f:4c=1489641  (s2)
Oplog IDs for row "s2" (@ v2.uk):
oid:3b56807b.489a11de.9517.00:21:5e:40:81:8c=3493 (s1)
oid:76d45938.f99a11dd.b047.00:02:c4:65:3f:4c=1489641 (s2)
```

Here, the `s2` replica recorded that the sync packets with epoch numbers up to `1489641` have been exported for the `s1` replica and it presumes they were applied there.

The quick and dirty solution would be to reset the `1489641` epoch number to `1369884` and leave it like that until the replicas get synchronized:

```
$ mt chepoch s2@/vob/foo s1=1369884
$ mt lsepoch -invob /vob/foo
For VOB replica "/vob/foo":
Oplog IDs for row "s1" (@ v1.uk):
oid:3b56807b.489a11de.9517.00:21:5e:40:81:8c=3493 (s1)
oid:76d45938.f99a11dd.b047.00:02:c4:65:3f:4c=1369884 (s2)
Oplog IDs for row "s2" (@ v2.uk):
oid:3b56807b.489a11de.9517.00:21:5e:40:81:8c=3493 (s1)
oid:76d45938.f99a11dd.b047.00:02:c4:65:3f:4c=1489641 (s2)
```

But that would result in the "avalanche" export of all the epoch numbers from `1369884` to `1489641` into sync packets in the `s2` replica, and avalanche shipping them all (once again) to `s1` host. This produces a huge traffic of unnecessarily duplicated sync packets that have already been delivered to the `v1` host.

So, a smarter solution would be to temporarily disable `sync_export` (or do it well between its scheduled executions), then reset the epoch number to `1369884` as specified above, generate a small sync packet, and check whether the new increased epoch number is equal to or more than the desired `1369900` number (the point where importing the first packet from the stuck pile can start):

```
$ mt chepoch s2@/vob/foo s1=1369884
$ mt sync -export -max 50k -lim 1 -nc -fship s1@/vob/foo
$ mt lsepoch s1@/vob/foo
For VOB replica "/vob/foo":
Oplog IDs for row "s1" (@ v1.uk):
oid:3b56807b.489a11de.9517.00:21:5e:40:81:8c=3493 (s1)
oid:76d45938.f99a11dd.b047.00:02:c4:65:3f:4c=1369902 (s2)
```

This looks fine now, so we can set the original epoch number `1489641` back to avoid generating any more sync packets, as the small packet we exported should be enough for importing all the sync packets that have accumulated in the incoming bay on the `v1` host:

```
$ mt chepoch s2@/vob/foo s1=1489641
```

History of exports

There is a good deal of information in the history of the replica objects: in the local replica, the history of imports, and in the remote replicas, the history of exports to the respective destinations. This is an important tool to troubleshoot synchronization problem: when did you last successfully import a packet from a given replica? What were the epochs prior to the import? The epochs after the import, you get from the next import event, or if this was the last one, from the current epochs.

The only problem is the verbosity of the records. They contain the epoch for all the replicas, including the deleted ones (which safely serves the needs of the `recoverpacket` command; see earlier in *Missing oplogs*). Here is our last export to the `andromeda` replica:

```
$ ct lshis -last replica:andromeda
--10-05T18:25 root export sync from replica "wonderland" to replica #####
                                                              "andromeda"
 "Exported synchronization information for replica "andromeda".
 Row at export was: centauri=10758 andromeda=67 wonderland=1377 #########
                                                    centauri.deleted=3"
```

It is typically more convenient to restrict the output to one or a few relevant replicas:

```
$ ct lshis -last 8 -fmt "%d %Nc\n" replica:andromeda | \
  perl -n0777e \
    'while (/^([\dT:+-]+) .*?wonderland=(\d+)[^\n]*$/gms){
      print "$1: $2\n"}'
2010-10-05T18:20:01+05:30: 1377
2010-10-05T14:50:02+05:30: 1374
2010-10-05T11:50:24+05:30: 1371
2010-10-04T18:40:01+05:30: 1370
2010-10-04T17:10:09+05:30: 1368
2010-10-01T19:30:14+05:30: 1365
2010-10-01T19:00:13+05:30: 1360
2010-10-01T18:50:03+05:30: 1354
```

This is again a job for Perl post-processing. This time, we use the "slurp" mode, the paragraph mode with a non-existing character (octal `0777`) as separator. We treat the multiline output as scalar, as a large string. We then repeatedly retrieve the interesting records, from which we every time extract and print, the time stamp and the starting epoch value concerning our own replica. Note the use of the `g` modifier (*Global matching*) to retrieve all the occurrences of matching the regular expression, updating the position to the beginning of the next line at every step.

Consequences of replicas being out of sync

The window of opportunity for creating *evil twin types* (label, attribute, and so on) grows. When the vobs get eventually synched, the names of the remote types clash at import with the ones created locally. They get renamed.

Now, the funny detail is that the format for the renaming has changed at some point (presumably between 7.0.1.1 and 7.0.1.4).

It used to be `<replica_name>:<type_name>` and now changed to `<replica_name>_<type_name>`.

Here is a scenario. We are using a script that creates when needed in the current vob a shared BldDir attribute type and applies it. It doesn't matter which site masters the type (which one needed it first), but it matters that both sites use the same if they use any. We used this script at our local replica, wonderland, before we noticed that recent packets from the andromeda replica had not been imported. When the synchronization is restored, we notice that a BldDir attribute type had been created at andromeda, the name of which now clashed with the type we created at wonderland. During the import, the remote type was automatically renamed to andromeda_BldDir. Note that at andromeda, the situation is mirrored; the local type is named BldDir and the remote one as wonderland_BldDir:

```
[wonderland]$ ct lstype -fmt "%n %[type_mastership]p %[master]p\n" \
 -kind attype | grep BldDir
BldDir shared wonderland@/vob/foo
andromeda_BldDir shared andromeda@/vob/foo

[andromeda]$ ct lstype -fmt "%n %[type_mastership]p %[master]p\n" \
 -kind attype | grep BldDir
wonderland_BldDir shared wonderland@/vob/foo
BldDir shared andromeda@/vob/foo
```

How do we rename the type back at the wonderland replica? One must first remove the offending local type BldDir:

```
[wonderland]$ ct rmtype attype:BldDir
Removed attribute type "BldDir".
```

But even after that one cannot rename wonderland_BldDir back locally because it is mastered by the remote wonderland.

The solution is to request that the andromeda site generate two rename events:

```
[andromeda]$ ct rename attype:BldDir Foo
[andromeda]$ ct rename attype:Foo BldDir
```

On importing these events at the wonderland site, the remote type andromeda_BldDir will get back its original name—BldDir. In the meantime, you may of course create attributes using the renamed name; they will eventually be of type BldDir, upon synchronization between the replicas:

```
$ ct mkattr andromeda_BldDir '"foo/bar"' lbtype:FOO
Created attribute "andromeda_BldDir" on "FOO".
```

Export failures

Here is just an example of a very simple `sync_export` error: `Protect Container failed`. Let's explore how it is presented, how to dig up the essential information, and how to fix it.

```
~> ct getlog -last 34 sync_export
============================================================
Log Name: sync_export Hostname: beyond  Date: 2010-10-06T16:42:14+05:30
Selection: Last 34 lines of log displayed
------------------------------------------------------------
Target replica(s) up to date. No export stream generated.
Generating synchronization packet /stg/shipping/ms_ship/outgoing/ #######
                          sync_wonderland_2010-10-06T15.01.18+05.30_17101
multitool: Error: Vob server operation "Protect Container" failed.
Additional information may be available in the vob_log on host "beyond"
multitool: Error: Unable to change permissions of "c/cdft/18/42/ ########
            5640ca84a17511df80d300018503cbba": No such file or directory.
multitool: Error: Vob server operation "Protect Container" failed.
Additional information may be available in the vob_log on host "beyond"
multitool: Error: Unable to change permissions of "c/cdft/18/42/ ########
            5640ca84a17511df80d300018503cbba": No such file or directory.
multitool: Warning: ../vob_export.cxx:222: operation ####################
                    'vob_ver_get_data' failed: No such file or directory.
multitool: Error: Could not get statistics of the version data file #####
                                                        for this operation.
2153121:
op= checkin_do
replica_oid= 0d33a902.6d6f11df.98d6.00:30:6e:5d:e0:86
oplog_id= 766
op_time= 2010-08-07T04:54:16Z create_time= 2010-08-07T04:54:16Z
version_oid= 5640ca84.a17511df.80d3.00:01:85:03:cb:ba
event comment= ""
cr_info= 0xc7f00
data size= 148 data= 0xb5d08
------------
ver_oid= 5640ca84.a17511df.80d3.00:01:85:03:cb:ba
ver_num= 2
ver_fstat= ino: 0; type: 1; mode: 00
 usid: DONTCARE
 gsid: DONTCARE
 nlink: 0; size: 14154
 atime: Thu Jan 1 05:30:00 1970
 mtime: Sat Aug 7 10:24:16 2010
 ctime: Sat Aug 7 10:24:16 2010
ckout_ver_oid= 5640ca84.a17511df.80d3.00:01:85:03:cb:ba
nsdir_elem_oid= 00000000.00000000.0000.00:00:00:00:00:00
name_p= ""
multitool: Error: Removing incomplete packet /stg/shipping/ms_ship/ #####
                 outgoing/sync_wonderland_2010-10-06T15.01.18+05.30_17101
```

```
ERROR: command './bin/multitool syncreplica -export -maxsize 50m ########
       -fship -limit 1 replica:andromeda@/vob/foo >&2' encountered error.
~> ct getlog -aro 15:00 vob
============================================================
Log Name: vob Hostname: beyond Date: 2010-10-06T16:20:25+05:30
Selection: Lines between 2010-10-06T14:50:00+05:30 and 2010-10-
06T15:10:00+05:30 displayed
------------------------------------------------------------
2010-10-06T15:01:35+05:30 vob_server(1766): Error: unable to access file
 c/cdft/18/42/5640ca84a17511df80d300018503cbba: No such file or directory
2010-10-06T15:01:35+05:30 vob_server(1766): Error: Unable to chmod
 container /vobstg/foo.vbs/c/cdft/18/42/5640ca84a17511df80d300018503cbba:
 No such file or directory
...
```

We take the oid shown in the log, which is also the base name of the container, just formatted differently. This allows us to find the exact version corresponding to this oid, by executing a `describe` command in a view context, from a directory in the vob:

```
$ ct des -s oid:5640ca84.a17511df.80d3.00:01:85:03:cb:ba
/vob/foo/bar/example@@/main/mg/2
$ file /vob/foo/bar/example@@/main/mg/2
/vob/foo/bar/example@@/main/mg/2: commands text
```

Back to the vob storage:

```
$ ls -l /vobstg/foo.vbs/c/cdft/18/42/5640ca84a17511df80d300018503cbba
-r-xr-xr-x 1 vobown jgroup  14154 Oct 6 16:24 /vobstg/foo.vbs/c/cdft/ ###
                                    18/42/5640ca84a17511df80d300018503cbba
```

Now the once missing cleartext container is found where expected. It is of course the `file` command that forced the generation of the container, and this alone was enough to make the export succeed! The reason for the error was that exporting the event required protecting the cleartext container, but this one had been scrubbed, and there was no instruction to recreate it. Such a recreation happens automatically, but requires a view context, as in our example above.

Incompatibility between ClearCase releases

It may happen that in a ClearCase MultiSite setup, different sites run different ClearCase versions; even such versions that support different feature levels (for example, 7.0 with FL 5 and 2003.06.00 with FL 4). When a new vob is created on a site with FL 5, and replicated to a site with a lower feature level, the replica package is not importable at the destination.

To support MultiSite inter-operability between such sites, the feature level of new vob must explicitly be set low at vob creation time, by using the `-flev target_site_fl` option of `ct mkvob`.

```
[joe@v1 ~]$ ct hostinfo -l | grep Product
 Product: ClearCase 7.0.1.6
[joe@v1 ~]$ ct hostinfo -host v2 -l | grep Product
 Product: ClearCase 2003.06.10+
```

So, the site `s1` has a default feature level of 5, and the site `s2` supports only FL 4. Here is how to create a new vob on site `s1`, with the intention to replicate it to site `s2` (to a vob server `v2`):

```
$ ct mkvob -tag /vob/foo -flev 4 -nc -stgloc -auto
$ ct des -fmt "%[flevel]p\n" vob:/vob/foo
4
```

We can now create a replica for `s2` site, as usual:

```
$ mt mkrep -exp -work /tmp/foo -nc -fship v2:s2@/vob/foo
```

MultiSite shipping problems—a tricky case

Here is a known problem with ClearCase MultiSite shipping, easy enough to produce accidentally, but tricky to debug.

This will be the occasion to show the use of `mkorder` and of some under-documented debugging features of ClearCase: special environment variables, and the debug option of `shipping_server`. All of these apply for sure to a large range of ClearCase MultiSite administration cases.

The MultiSite configuration in this case is the following: two sites with a vob server and a shipping server on each side of a firewall. A range of ports is open in the firewall between the two shipping servers, so that they may contact each other, but not the other site's vob server through the firewall.

Let's call the vob servers `v1` and `v2`, and the shipping servers `sh1` and `sh2`.

The problem symptoms: shipping goes successfully in one direction (for example, `v1-> sh1 -> sh2 -> v2`), but fails in the other with the following messages:

```
$ cleartool getlog -since yesterday -host sh2 shipping
07/06/2009 02:56:12 PM shipping_server(23352): 4691): Error: unable to ##
              deliver/forward order /usr/atria/shipping/ms_ship/outgoing/
              sh_o_s2.ddd_1_46
07/06/2009 02:56:12 PM shipping_server(23352): 4691): Error:
A shipping_server RPC failed. Additional information may be ##############
                    available in the albd_log on the destination machine.
```

```
07/06/2009 02:56:12 PM shipping_server(23352): 4691): Error: ############
                  shp_forward_request_V1: RPC: Unable to receive; errno =
                  Connection reset by peer
07/06/2009 02:54:51 PM shipping_server(23352): 4691): Error: ############
                                Unable to contact albd_server on host sh1
07/06/2009 02:54:31 PM shipping_server(23352): 4691): Error: ############
                                     connect failed: Connection timed out
```

We check that `sh2` can contact the `albd_server` on `sh1`:

```
[sh2]$ albd_list sh1
albd_server addr = 155.111.22.33, port= 371 PID 10962:
shipping_server, tcp socket 49158: version 1; BUSY PID 29560:
shipping_server, tcp socket 49164: version 1; BUSY PID 30105:
Albd_list complete
```

So, let's create a test packet and ship it with `mkorder` from the shipping server `sh2` to the vob server `v1` (advantage: it will not get imported):

```
$ hostname
sh2
$ touch /tmp/foo
$ mkorder -data /tmp/foo -fship v1
Shipping order "/tmp/sh_o_foo" generated.
Attempting to forward/deliver generated packets...
---- NOTE: consult log file ##############################################
        (/var/adm/rational/clearcase/log/shipping_server_log) for errors.
mkorder(19770): Error: Store-and-forward server ##########################
       "/opt/rational/clearcase/etc/shipping_server" failed with status 1
mkorder(19770): Warning: Unable to send packet /tmp/foo ##################
                                                (see store-and-forward log)
```

We check that the error is logged:

```
$ cleartool getlog -host sh2 shipping
[...] shp_forward_request_V1: RPC: Unable to receive; errno = ############
                                                      Connection reset by peer
```

We verify that shipping from `sh1` to `v2` works:

```
[sh1]$ mkorder -data /tmp/foo -fship v2
Shipping order "/tmp/sh_o_foo" generated.
Attempting to forward/deliver generated packets...
-- Forwarded/delivered packet /tmp/foo
$ tail -1 /var/adm/atria/log/shipping_server_log
07/06/09 16:25:37 shipping_server(6515): Ok: Forwarded order #############
                                    /tmp/sh_o_foo (data "foo") to host "v2"
```

Let's now start `shipping_server` with options `-server -d`, and set the ClearCase environment variables—`TRACE_SUBSYS` and `TRACE_VERBOSITY`—to monitor the shipping session. Note that `TRACE_SUBSYS` may be used with any subsystem to restrict the verbosity to the one of current interest. This is the server side, at `sh1` host in our case, where we start the shipping server process in the verbose debug mode:

```
$ export TRACE_SUBSYS=shp_subsys
$ export TRACE_VERBOSITY=4
$ /usr/atria/etc/shipping_server "" -server -d
>>> (shipping_server(4956)): Configuration
>>> (shipping_server(4956)): -------------
>>> (shipping_server(4956)): max data size 2097151
>>> (shipping_server(4956)): minimum port setting 49152
>>> (shipping_server(4956)): maximum port setting 49171
>>> (shipping_server(4956)): notify_prog = /usr/atria/bin/notify
>>> (shipping_server(4956)): storage classes
>>> (shipping_server(4956)): -default ---> /usr/atria/shipping/ms_ship
>>> (shipping_server(4956)): -default ---> /usr/atria/shipping/ms_rtn
>>> (shipping_server(4956)): routes[...]
>>> (shipping_server(4956)): sh2: -default
>>> (shipping_server(4956)): administrators (1)*** listening on #########
   port 49153 ***And we record the port number it is listening on: 49153.
```

Now, from our client side, on `sh2`, we can open a shipping session, specifying the exact port used on the server side (`49153`):

```
$ mkorder -copy -ship -data /tmp/foo v1
Shipping order "/opt/rational/clearcase/shipping/ms_ship/outgoing/ ######
                                                    sh_o_foo" generated.
$ export TRACE_VERBOSITY=4
$ export TRACE_SUBSYS="*"
$ shipping_server /opt/rational/clearcase/shipping/ms_ship/outgoing/ ####
                                                  sh_o_foo -d sh1:49153
shipping_server(692): Error: Operation "connect" failed: No such file ###
                                                        or directory.
$ cat /opt/rational/clearcase/shipping/ms_ship/outgoing/sh_o_foo
# ClearCase MultiSite Shipping Order
# Version 1.0
%IDENTIFIER c5dfcedc.263247dd.a40e.f3:83:5e:db:d9:3b
%CREATION-DATE 1082014994 06-Jul-09.10:43:14
%EXPIRATION-DATE 1083224595 06-Jul-09.10:43:15
%ORIGINAL-DATA-PATH /opt/rational/clearcase/shipping/ms_ship/outgoing/ ##
                                                                  sh_o_foo
%LOCAL-DATA-PATH /opt/rational/clearcase/shipping/ms_ship/outgoing/ #####
                                                                  sh_o_foo
%DESTINATIONS v1
%ARRIVAL-ROUTE%DELIVERIES
%FAILED-ATTEMPTS
```

```
R 1082015248 06-Jul-09.10:47:28 sh1 v1
%NOTIFICATION-ADDRESSES
%COMMENT
```

On the server side one receives the following output:

```
>>> (shipping_server(4956)): socket addr is 0.0.0.0
shipping_server(4956): Error: timeout waiting for transfer RPC request
```

The investigation showed that the final destination matters—shipping only to `sh1` worked:

```
[sh2]$ mkorder -data /tmp/foo -fship sh1
Shipping order "/tmp/sh_o_foo" generated.
Attempting to forward/deliver generated packets...
-- Forwarded/delivered packet /tmp/foo
$ tail -1 /var/adm/atria/log/shipping_server_log
07/06/09 16:25:37 shipping_server(6515): Ok: Forwarded order ############
                                  /tmp/sh_o_foo (data "foo") to host "sh1"
```

But mentioning `v1` as the destination caused a failure, as we demonstrated earlier.

In this case, the cause of the problem was a misconfiguration of the DNS, mentioning a decommissioned server:

- The shipping server always attempts to resolve the destination, and this implies by default to issue a DNS request
- In the case that the DNS server doesn't reply, the shipping session times out
- The quick work-around was to add the destination to `/etc/hosts` (as `/etc/nswitch.conf` instructed to use *files* before *DNS* to resolve host names)

The actual solution was of course to fix the DNS servers in `/etc/resolv.conf`.

The tricky problem here is that for shipping purposes and routing via a shipping server, resolving the actual destination (the vob server behind a firewall) is, of course, completely useless: the host name is likely to be non resolvable, its IP address non reachable. The difference in the shipping server behavior is dramatically different though depending on whether it does receive a negative answer from a configured DNS server (such as "`destination host is not resolvable`") within a certain time limit, or it keeps waiting for the pending DNS answer (in case the DNS itself is misconfigured) and the shipping session terminates then after a timeout.

Summary

Once again, we were able to show some scenarios in which users could solve their problems by themselves without the need for administrative rights or special skills, just by paying careful attention to the error messages. In the last case of course, the administrator was eventually required, but the decisive step was to use the `mkorder` tool to analyze the shipment into simpler steps that could be shown to succeed. Even the MultiSite architecture, with its routing tables and shipping servers, is not as complex as it might seem, and information may be brought closer to the users as we shown with the remote epoch monitoring.

It is our experience that end users are willing to understand whether a delay in synchronization is normal, or the symptom of a problem to report or to investigate.

12
Challenges

The unique features of clearmake we have been describing so far, and to which we grant so much value, fall short in at least two extremely significant contexts: *Java* and *MultiSite*. Our goal in this chapter will be to analyze the causes of the problems, to examine the existing solutions, and to propose directions for extensions. In the end, we'll also briefly consider a few other challenges, opened by recent evolutions in software development.

Java

ClearCase was mostly (at least originally) written in C++, and its build model is well suited (with some historical adjustments to cope with templates in two generations of compilers) to development using this language. Java, although already old at the time of its wide success, broke a few significant assumptions of the model.

Problems with the Java build process

The traditional build model is stateless, and therefore easily reproducible: running the same build command in the context of static sources (*leaves* of the dependency tree, seen upwards from the products) produces the same results, but doesn't alter the context. This is not the case anymore with javac. The reason is trivial: javac integrates into the compiler a build tool function. The *compiler* reads the Java source as a build script and uses the information to build a list of dependencies, which it verifies first, using traditional time stamp comparison between the sources and class files produced, and rebuilding missing or out-dated class files. It doesn't, however, perform a thorough recursive analysis, nor attempt to validate jars, for instance.

This behavior is highly problematic from a clearmake point of view, as it results in the list of derived objects produced with a given rule (siblings of the target) being variable from one invocation to the next, and conversely, in a given derived object potentially being produced by several different rules. Both of these effects result in incorrect dependency analysis, and in spurious invalidation of previous build results.

Let's note that since javac performs time stamp comparisons, the default behavior of cleartool to set the timestamp at checkin time is inadequate for Java sources, and results in needlessly invalidating classes produced before checkin: set up a special element type manager defaulting to the `-ptime` (preserve time) checkin option.

The second traditional assumption broken by Java is a practical one: the language has been designed to optimize compilation speed, which means that build time stops being a primary issue. This is of course obtained by using a single target, the Java virtual machine, and at the expense of run-time performance; but history has already clearly validated this choice, in the context of favorable progresses in hardware.

This is obviously not a problem in itself, but it had two clear consequences:

- The winkin behavior of clearmake cannot be *sold* to users anymore on the argument of its saving build time (by side-effect). As we know, the argument of the accuracy of management appeals only to users having experienced its importance, and *reward following investment* is a known recipe for failure in evolution.
- It encourages carelessness among developers: throw away (clean) and start again from scratch.

Of course, the gain in speed is mostly felt in small configurations, and at the beginning of projects: this strategy doesn't scale, as the total build time still depends on the overall size of the component, instead of on this of the increment (the number of modified files). It is however often late to change one's strategy when the slowness becomes noticeable.

.JAVAC support in clearmake

Support for Java was added relatively late to clearmake (with version 2003.06.00), in terms of a `.JAVAC` special target (and a `javaclasses` makefile macro). The idea (to which your authors contributed) was to use the **build audit** to produce a `.dep` file for every class, which would be considered by clearmake in the next invocation, thus giving it a chance to preempt the *javac* dependency analysis. Of course, the dependency tree would only be as good as this of the previous compile phase, but it would get refined at every step, eventually converging towards one which would satisfy even the demanding `catcr -union -check`.

Special care was needed to handle:

- Inner classes (producing several class files per java source, some of them being prefixed with the name of the enclosing class, with a dollar sign as separator—not a friendly choice for UNIX shells).
- Cycles, that is, circular references among a set of classes: a situation which clearmake could only process by considering all the set as a common target with multiple siblings.

This solution should be very satisfactory, from the point of view of ensuring correctness (consistency of the versions used), sharing of objects produced, and thus managing by differences. It should offer scalability of performance, and therefore present a breakeven point after which it would compete favorably with from scratch building strategies.

One might add that a makefile-based system is likely to integrate with systems building components written in other languages (such as C/C++), as well as with performing other tasks than compiling Java code.

Let us demonstrate how the dependency analysis and derived objects reuse are working using the `.JAVAC` target in the makefiles, testing exactly the aspects mentioned above—inner classes and cycles.

In our small example, `Main.java` implements the main class, `FStack.java` implements another independent class, which the `Main` class is using. Finally, the `FStack` class also contains an inner class `Enumerator`, which results after the compilation in a file of name `FStack$Enumerator.class`:

```
# Main.java
public class Main {
   public static void main(String args[]) {
      FStack s = new FStack(2);
      s.push("foo");
      s.push("bar");
   }
};

# FStack.java
public class FStack {
   Object array[];
   int top = 0;
   FStack(int fixedSizeLimit) {
      array = new Object[fixedSizeLimit];
   }
   public void push(Object item) {
      array[top++] = item;
   }
```

```
      public boolean isEmpty() {
         return top == 0;
      }
      public class Enumerator implements java.util.Enumeration {
         int count = top;
         public boolean hasMoreElements() {
            return count > 0;
         }
         public Object nextElement() {
            return array[--count];
         }
      }
      public java.util.Enumeration elements() {
         return new Enumerator();
      }
   }
```

We create a tiny `Makefile` making use of the `.JAVAC` target. Note that we do not have to describe any dependencies manually; we just mention the main target `Main.class`, leaving the rest to the javac and the ClearCase Java build auditing:

```
   # Makefile
   .JAVAC:
   .SUFFIXES: .java .class
   .java.class:
           rm -f $@
           $(JAVAC) $(JFLAGS) $<
   all: /vob/jbuild/Main.class
```

The first run of the clearmake does not look very spectacular: it just executes the javac compiler, submitting the `Main.java` source to it, and all the three class files (`FStack.class`, `FStack$Enumerator.class`, and `Main.class`) get generated. The same would have been produced if we used the "default" Makefile (the same, but without the `.JAVAC` target):

```
$ clearmake -f Makefile
  rm -f /vob/jbuild/Main.class
  /usr/bin/javac  /vob/jbuild/Main.java
```

Note though that one thing looks different from the default Makefile execution: our `".JAVAC" Makefile` produces the following dependency (`.dep`) files:

```
$ ll *.dep
-rw-r--r--  1 joe jgroup 654 Oct 19 14:45 FStack.class.dep
-rw-r--r--  1 joe jgroup 514 Oct 19 14:45 Main.class.dep
```

But their contents are somewhat puzzling at the moment:

```
$ cat FStack.class.dep
<!-- FStack.class.dep generated by clearmake, DO NOT EDIT. -->
<version value=1 />
<!-- (A build of this target has not been directly audited.) -->
<mytarget name=/vob/jbuild/FStack.class conservative=true />
<mysource path=/vob/jbuild/FStack.java />
<!-- Target /vob/jbuild/FStack.class depends upon the following #########
                                                              classes: -->
<target name=/vob/jbuild/Main.class path=/vob/jbuild/Main.class />
<cotarget name=/vob/jbuild/FStack.class path=/vob/jbuild/ ###############
                                    FStack$Enumerator.class inner=true />

$ cat Main.class.dep
<!-- Main.class.dep generated by clearmake, DO NOT EDIT. -->
<version value=1 />
<!-- (A build of this target has been directly audited.) -->
<mytarget name=/vob/jbuild/Main.class conservative=false />
<mysource path=/vob/jbuild/Main.java />
<!-- Target /vob/jbuild/Main.class depends upon the following ############
                                                              classes: -->
<target name=/vob/jbuild/FStack.class path=/vob/jbuild/ #################
                                       FStack.class precotarget=false />
```

So, it looks as if the `FStack` class was depending on the `Main` class, and the other way around as well. But that's what one can only figure out after a single javac execution—The `Main` class was produced and, in order to compile it, two more classes were needed: `FStack` and `FStack$Enumerator`.

But we can do better. Let's try the second subsequent clearmake execution, without any real changes (for our purpose: in a real work scenario, a new build would of course be motivated by a need to test some changes). It does not yield `all is up to date`, as one would expect when using the default Makefile, but instead it does something interesting:

```
$ clearmake -f Makefile
  rm -f /vob/jbuild/FStack.class
  /usr/bin/javac  /vob/jbuild/FStack.java

  rm -f /vob/jbuild/Main.class
  /usr/bin/javac  /vob/jbuild/Main.java
```

Note that it does not even execute the default script, but rather some other one (`/usr/bin/javac /vob/jbuild/FStack.java`). Where did it come from? Actually from the `FStack.class.dep` dependency file mentioned above. And what about the dependency files themselves?-They have somewhat changed:

```
$ cat FStack.class.dep
<!-- FStack.class.dep generated by clearmake, DO NOT EDIT. -->
<version value=1 />
<!-- (A build of this target has been directly audited.) -->
<mytarget name=/vob/jbuild/FStack.class conservative=false />
<mysource path=/vob/jbuild/FStack.java />
<!-- Target /vob/jbuild/FStack.class depends upon the following #########
                                                              classes: -->
<cotarget name=/vob/jbuild/FStack.class path=/vob/jbuild/ ###############
                                    FStack$Enumerator.class inner=true />

$ cat Main.class.dep
<!-- Main.class.dep generated by clearmake, DO NOT EDIT. -->
<version value=1 />
<!-- (A build of this target has been directly audited.) -->
<mytarget name=/vob/jbuild/Main.class conservative=false />
<mysource path=/vob/jbuild/Main.java />
<!-- Target /vob/jbuild/Main.class depends upon the following ############
                                                               classes: -->
<target name=/vob/jbuild/FStack.class path=/vob/jbuild/ #################
                                       FStack.class precotarget=false />
```

And now this looks right! The `FStack` class depends on `FStack$Enumerator`, but it does not depend on the `Main` class, and this is noted in the modified `FStack.class.dep`. The `Main` class, on the other hand, does depend on `FStack`, and that is stated correctly in `Main.class.dep`.

Now, if we try to run clearmake once again, it yields `'all' is up to date`:

```
$ clearmake -f Makefile
'all' is up to date.
```

But this time it means that all the dependencies have been analyzed and recorded in the `dep` files.

Let's try to rebuild. Removing either `FStack.class` or `FStack$Enumerator.class` causes its rebuild, which triggers in turn a rebuild of the dependent `Main.class`:

```
$ rm FStack$Enumerator.class
$ clearmake -f Makefile
  rm -f /vob/jbuild/FStack.class
  /usr/bin/javac  /vob/jbuild/FStack.java
  rm -f /vob/jbuild/Main.class
  /usr/bin/javac  /vob/jbuild/Main.java
```

Note that with the default makefile (which does not have any dependencies recorded manually), such a removal would not invalidate the build and `'all' is up to date` would have been yielded.

And even an unconditional rebuild goes differently than the first time clearmake execution. The main target gets built last, and before it, the other independent targets get executed:

```
$ clearmake -f Makefile -u
  rm -f /vob/jbuild/FStack.class
  /usr/bin/javac  /vob/jbuild/FStack.java

  rm -f /vob/jbuild/Main.class
  /usr/bin/javac  /vob/jbuild/Main.java

$ rm Main.class
$ clearmake -f Makefile
  rm -f /vob/jbuild/Main.class
  /usr/bin/javac  /vob/jbuild/Main.java
```

And what about the dependency files? Have they changed? Not this time. As long as we do not change the code, the dependencies remain the same and the `dep` files are not affected by the rebuild.

Using a second view, we can now demonstrate the winkin:

```
$ ct setview view2
$ clearmake -f Makefile
Wink in derived object "Main.class.dep"
Wink in derived object "FStack.class.dep"
Wink in derived object "FStack$Enumerator.class"
Wink in derived object "FStack.class"
Wink in derived object "Main.class"
```

Let's now introduce an artificial circular dependency between the `FStack` and `Main` classes, for example by adding the following line to the `FStack` class constructor:

```
Main m = new Main();
```

We see that after a couple of clearmake executions, the new dependencies get resolved and recorded in `FStack.class.dep`:

```
$ clearmake -f Makefile
  rm -f /vob/jbuild/FStack.class
  /usr/bin/javac  /vob/jbuild/FStack.java

  rm -f /vob/jbuild/Main.class
  /usr/bin/javac  /vob/jbuild/Main.java

$ clearmake -f Makefile
  rm -f /vob/jbuild/FStack.class
  /usr/bin/javac  /vob/jbuild/FStack.java
  rm -f /vob/jbuild/Main.class
  /usr/bin/javac  /vob/jbuild/Main.java

$ clearmake -f Makefile
'all' is up to date.

$ cat FStack.class.dep
<!-- FStack.class.dep generated by clearmake, DO NOT EDIT. -->
<version value=1 />
<!-- (A build of this target has been directly audited.) -->
<mytarget name=/vob/jbuild/FStack.class conservative=false />
<mysource path=/vob/jbuild/FStack.java />
<!-- Target /vob/jbuild/FStack.class depends upon the following #########
                                                              classes: -->
<cotarget name=/vob/jbuild/FStack.class path=/vob/jbuild/ ################
                                    FStack$Enumerator.class inner=true />
<target name=/vob/jbuild/Main.class path=/vob/jbuild/ ####################
                                           Main.class precotarget=true />
<cotarget name=/vob/jbuild/Main.class path=/vob/jbuild/ ##################
                                                 Main.class inner=false />
```

Now, `FStack` depends on `Main`, and `Main` depends on `FStack`, and invalidating either of them would cause a rebuild of the other:

```
$ rm FStack.class
$ clearmake -f Makefile
  rm -f /vob/jbuild/FStack.class
  /usr/bin/javac  /vob/jbuild/FStack.java
  rm -f /vob/jbuild/Main.class
  /usr/bin/javac  /vob/jbuild/Main.java

$ clearmake -f Makefile
'all' is up to date.
$ rm Main.class
$ clearmake -f Makefile
  rm -f /vob/jbuild/FStack.class
  /usr/bin/javac  /vob/jbuild/FStack.java
```

```
rm -f /vob/jbuild/Main.class
/usr/bin/javac  /vob/jbuild/Main.java
```

This scheme supports some level of sophistication such as maintaining one single makefile for several source directories, and exporting the classes to a build hierarchy. One may use for this purpose the `javaclasses` macro, including while using a distinct deployment directory:

```
%.class : %.java
    rm - f $@
    $(JAVAC) $(JFLAGS) -d $(BUILD) $<

COM = org/wonderland
SROOT = $(SRC)/$(COM)
CROOT = $(BUILD)/$(COM)
CLASSES = $(javaclasses $(CROOT)/dir1, $(SROOT)/dir1) \
    $(javaclasses $(CROOT)/dir2, $(SROOT)/dir2) \
    $(javaclasses $(CROOT)/dir3, $(SROOT)/dir3)

all: $(CLASSES)
```

Note however that this macro is only mechanically computed from the contents of the source directories and cannot reliably be used to name all the classes collected in a *jar*: it would miss the inner classes.

Also, it may in fact catch files that would actually not be needed. It is thus not obvious whether using it is actually better than relying as in our first case, on naming one (or some) main targets and relying upon the dependency analysis to find the others!

The use of a deployment directory (used with the `-d` option of javac) makes it easy to create a jar from what was actually produced.

This `.JAVAC` support seems quite usable, even if we lack a real size experience. All the infancy problems that were reported at some stage have been satisfactorily addressed.

Unfortunately, this solution has not been widely used in practice due to the following reasons:

- It came too late.
- It performs comparably poorly in small configurations, because it invokes javac once per target; javac starts a Java virtual machine and the cost of initializing and finalizing it overshadows the build cost proper: it is much more efficient to invoke javac once for a large set of files.
- It departs from the main stream practices, and appeals to competences not typically shared by Java developers (makefiles).

These are however to some extent only speculations, based upon assumptions more than on measurements. The performance penalty alluded to above may actually be overstated...

The third bullet leads us naturally to the next sections: both *ant* and *maven* come with off-the-shelf support for producing the other kinds of deliverables (*jar*, *war*, *ear*... and so on) common to web development, as well as for deploying them to the various application servers (*WebSphere*, *tomcat*,... and so on) and the various repositories available, under *Eclipse* or *OSGi*. Nothing inherently impossible to do with makefiles, only no Open Source project or consortium maintains and offers such a support.

Ant and XML

The *main stream* tool for building Java code is *ant*, although it is getting challenged by *maven*. Neither is actually deeply concerned with the issues of consistency of the build products: they typically delegate the dependency traversal, at the local level, as well as the compilation to javac tasks in the Java virtual machine, and rely for the rest on performing a *clean*. Their goal is to supplement javac in generating sources from *idl*, handling multiple components and their inter-dependencies, packaging, deploying, and so on.

The efficiency of ant comes from the reason we described earlier: ant is itself written in Java—invoking it results in only one initialization of the java virtual machine. In theory, one could intercept the dependency analysis between build jobs. There is even a mechanism which may be used for this purpose: *listeners*.

After promising a *clearant* for a few years, ClearCase came with an `ant_ccase` man page describing how to audit Ant builds, using `CCAudit.jar`. No winkin is supported so far:

```
$ ant -listener CCAudits ...
```

This is of course a way to produce config records that might be analyzed with `catcr -union -check`. Will the evidence produced be sufficient to convince users to switch back from their *ant* build system to a clearmake / `.JAVAC` one? We recommend our readers to test their builds: we are used to raising surprise among users! The recipe is straightforward and non-invasive:

```
$ export ###################################################################
       CLASSPATH=/opt/rational/clearcase/java/lib/CCAudits.jar:$CLASSPATH
$ export ###################################################################
    LD_LIBRARY_PATH=/opt/rational/clearcase/linux_x86/shlib:$LIBRARY_PATH
$ ant -listener CCAudits -buildfile build.xml
```

In most of the cases, the cause of the surprise is incomplete cleaning (for example, of generated Java sources). Not auditing one's builds moves some responsibility back from the tool to the developers, which is error-prone. Remember that barring auditing, all the dependency analysis must be specific to the tool chain.

But we should note that the dependencies produced are not (at all) fine grained: basically all of the derived objects share one common (bulky) config record, as with the first run the `.JAVAC` clearmake build mentioned above. But unfortunately, in Ant build auditing there is no way to improve it.

We can again only speculate about the reasons why *clearant* didn't make it as promised (i.e. fully supporting *winkin*). One might conjecture that it wasn't so easy to beat the results achieved with clearmake and the `.JAVAC` mechanism.

In other words, the existing support is actually better than commonly thought, and originally anticipated.

Audited Objects

The resolution of our problems may come from a recent Open Source product: ***Audited Objects***. This offers build auditing without ClearCase, that is, using user level system call interception and an SQL database.

The huge interest there is that one may hope to transfer some functionality (competence, concerns) from ClearCase, to e.g. *git*.

MultiSite

We have already seen a few of the most obvious limitations of *ClearCase MultiSite*: since it doesn't replicate config records, i.e. the information about the successful production of derived objects, using well identified resources, MultiSite doesn't really allow to distribute a development environment. While crossing site boundaries, one loses the most interesting part of the information. The interactions between developers on the same site are much richer than the ones across the MultiSite boundary. This may feel acceptable in the context of a top-down process of work division and assignment of tasks, but it would be a limitation for a bottom-up process of managing peer contributions, which is closer to this of Open Source development.

We showed how to work around some of these limitations by using labels, but this obviously requires user attention, and manual action.

We also mentioned the problems of clashes between the names of types created in different replicas, within the windows of opportunity left open by the delays in synchronization. Resolving these problems in a satisfactory way would require structuring the namespace of types, or in fact allow to *version* types (group the types in sets, considering them as representatives thereof).

Some recent products bring interesting perspectives on these issues. Here are a few.

Maven, and Buckminster

We already mentioned *maven*. The most interesting conceptual novelty in this product, in spite of its reputation for efficient template processing, may be even more strongly demonstrated in *Buckminster*. Both systems identify certain build products in a way independent from the site in which they were originally produced—they allow one to alternatively download binaries *or* produce them locally: what matters is the result.

Of course these systems use crude naming conventions as the primary means of identification (which checksums in addition), and ClearCase is far more robust and flexible in this respect.

Mercurial and git

Distributed management has known a recent favor with *Mercurial* and *git* (and some other less widely used: *svk*, *Gnu arch* to name a few). These offer an alternative to the centralized model of *subversion* (and CVS) and to a paradox this model leads to—it resorts to *committers* applying patches, that is, to a process completely outside subversion itself!

All these products support a *pull* (*get*) instead of the *push* model of ClearCase MultiSite. Once again, this means that information about interesting versions to pull has to be conveyed by means external to the SCM system: users must get convinced to make their pulling efforts by other means than evidence supported by the system.

These considerations should remind us that ClearCase was a forerunner in this category as well (*distributed* management), and that it is still a technological leader for those who want to use it.

However, here as well as for the continuity of management by auditing, some obvious limitations would need to be lifted for it to support tens of thousands of replicas. All replicas cannot be flatly exposed: some structure is badly needed.

In this respect, one must admit that both Mercurial and git are far superior to ClearCase: they easily support hundreds of sites, which may or may not be shared among communities of users. ClearCase developers came too hastily with the short-sighted concept of snapshot views, when laptops flooded their customer base. They missed the opportunity to extend MultiSite towards facing the challenge.

Perspectives in Software Engineering

Eclipse and OSGI

The *eclipse* repository stores multiple versions of the same jars, and offers some management of dependencies between user applications. This allows to propagate changes gradually, instead of upgrading all the applications depending on a common resource synchronously; it even allows to leave some applications as such, if they do not benefit from the changes.

This displays a radical novelty: the customer environment, under which one delivers, is not *homogeneous* and *consistent* anymore—delivery happens *under* SCM!

To be honest, this had always existed in UNIX, with shared libraries (see *Chapter 8, Tools Maintenance*). Using symbolic links, one could decouple the current (latest) version of a shared library, for linking purposes, from the ones actually bound to existing applications, which could thus use different versions and did not need to be updated every time a new version of the shared library was published.

Now, the support in eclipse clearly shows an increase in such uses.

One challenge may be to simulate this under ClearCase in the same view, for example, with hard links and pattern-based config spec rules (which we mentioned already, while speaking of *config specs* in *Chapter 2, Presentation of ClearCase*).
But one could take a wider perspective and consider that different views offer a much more flexible and robust support, and that the need for an SCM system is in fact much wider than might have been anticipated. One might want to see the eclipse repository as a first case of *delivering under SCM*, into an SCM system itself deployed in the customer environment. What is missing there is a *standard* or rather an open specification, to protect customers from being coupled to one single vendor.

The only existing attempt to standardize versioning interfaces is *WebDAV*, the extension of the HTTP protocol, geared towards authoring, that is, towards the other end of the configuration space (or dependency tree).

Virtual machines

ClearCase virtual file system, with its dynamic views, was a foregoer, but has its own limitations: some resources cannot be versioned (such as sockets, or any interfaces on which one and only one protocol is being used). This problem was first met with versioning shared libraries used at boot time.

An interesting trend is the use of virtual machines to split the resources of one host, protecting the users of a slice from what happens in another. Or other way around: cloud computing, federating multiple physical hosts to offer a virtual environment.

ClearCase has already certified certain virtual machines as platforms, but isn't the real challenge the opposite? Isn't there a need to run multiple virtual machines under some kind of software configuration management?

The challenge here would thus be to extend dynamic views to the status of full-blown virtual machines.

Conclusion

This chapter ended up posing more questions than it offered answers.

What it should convince us of is that there is still a need to develop ClearCase, and not to drop the experience specifically gained with this wonderful tool, being far from the main streams of tradition. That one's investment in ClearCase is not lost, even if it doesn't always seem directly usable in the contexts of its competitors.

13
The Recent Years' Development

This chapter is devoted to a cursory review of topics we did not deal with (at least in any depth) in this book, although they have been the focus of recent developments in ClearCase. We deliberately chose to spend time and effort on aspects left from existing literature, but also to focus on a consistent subset of ClearCase functionality. By this, we do not mean that the most recent developments in ClearCase would have been inconsistent: only that during recent years, ClearCase development has departed significantly, to our dismay, from the original project.

Let's consider this aspect in the (critical) light of the expertise gained so far, and review the following items:

- Triggers
- Snapshot views
- Express builds
- UCM
- Web access, the remote clients, and the Eclipse plugin
- CM API

Historical perspective

Although ClearCase was designed at Apollo, and originally developed at Atria, then PureAtria, its commercial success, especially on the Microsoft Windows platform, took place at Rational, and then IBM.

Most of what we have covered in this book concerns ClearCase as it was designed in the first phase of its development (referring to the above split).

The latter phase, besides taking care of upgrades and ports to new platforms, notably Linux platforms, has mostly addressed a different customer base than the former. The aspects of ClearCase developed more recently were those of an opaque product, aimed at consumers judging it as a black box, instead of as an integral part of their own conceptual toolbox.

The first ClearCase had been promoted as a tool set, not mandating a process. In 1997, the new owners from Rational thought that there was, on the contrary, a need for an out-of-the-box process. This perception matched the prospect of conquering the Windows market, whereas the original success of ClearCase had been gained almost exclusively on UNIX. This was a major cultural change. The user of UNIX command line is in charge of the process, whereas the user of a Windows GUI lets this one drive: choices have then to be made automatically, according to a pre-canned process, followed passively, without passion or any feeling of involvement. Mishaps will be reported, at best with a screen dump, to a black hole service desk, waiting for a magical "solution" in return; and this will happen only if the "problem" survives a reboot.

There may also have been a perception that concurrent products such as *Continuus* (which became *Synergy* and is now an IBM product) were gaining an edge with their radically different strategy, based on process enforcement. Last, Rational also wanted to integrate a full suite of products, starting with ones for bug tracking tickets (there were several, which eventually converged into *ClearQuest*), but also turning around visual modeling, the original area of expertise of the company (after Ada systems for the US Department of Defense).

Finally, there was a customer demand for *higher-level* concepts. This demand, borne in a *business* overtaking of the enterprise, after a long and controversial period of technological creativity, could dangerously well be met with the offering from the competition: components, activities, and tasks, driven from the bug tracking ticket perspective.

Triggers

Triggers were the first mechanism that could be used for process enforcement. They were indeed part of early implementations of ClearCase. We briefly handled them in *Chapter 9*, *Secondary metadata*.

Due to their popularity, triggers kept being expanded: they could be attached to new UCM or MultiSite events. They started being invoked in some remote client operations. They were not powerful enough, though, to implement the intended out-of-the-box process. This came as a disproof of the original claim that ClearCase provided the tools, and that the customers could arbitrarily design their process and use the tools to support it.

More or less following the same path as the UCM developers, we believe that triggers are not the proper technology to implement tailored support for one's own processes. We found wrappers as a more powerful choice, less surprising from the end user point of view, easier to debug and to override or skip in case of unexpected problems. We regret wrappers are not advertised to the same level (or rather, to a higher one) as triggers in the ClearCase documentation. In wrappers, there lies in our opinion, the best potential for Open Source contributions to ClearCase, at least for the time being.

Triggers remain an interesting tool for some prospective fixes (often temporary), especially limited to the context of a single vob. Then it is always a good idea to get the temporary fix (by the trigger) replaced by a more robust solution (for example, a wrapper). At this point, the temporary trigger can be removed. Let's note that the best way to disable a trigger is usually to `lock` it `-obsolete`. Add the comment first, and lock next, as the `-obsolete` option ignores any simultaneous comment. This allows us to document (in the comment field) the precise reason for the experimented inadequacy so that the same "fix" is not attempted again.

Snapshot views

We mentioned snapshot views very briefly in *Chapter 10*, *Administrative Concerns* and *Chapter 12*, *Challenges*. Snapshot views were introduced with the need to support work in a temporarily disconnected environment (typically on a laptop). They were not a great conceptual creation, as they merely implemented the traditional model of *sandboxes*, common to the older generation version control systems, still present under the name of *working copies*, in, for example, *subversion*.

Snapshot views still require occasional connectivity, to `load` and `update` them. They are registered in the ClearCase registry, and their state is published, as much as it concerns checkouts (reserved or not). Because ClearCase doesn't support a concept of private branches (such as in *Mercurial*), working in disconnected mode may lead to conflict situations (*hijacking*) to be resolved eventually, when the connectivity is restored.

In a disconnected snapshot view, one may edit elements that one had previously checked out, *hijack* some more files, and build using one's private files. Problems with proper ClearCase operations start with the lack of licenses, and even earlier with the need for authentication from network servers:

```
$ ct ls foo
cleartool: Error: Unable to contact albd_server on host 'ccserver'
cleartool: Error: Cannot contact license server host "ccserver"
cleartool: Error: You do not have a license to run ClearCase.

$ ct catcs
cleartool: Error: Unable to contact albd_server on host 'ccserver'
cleartool: Error: Cannot contact license server host "ccserver"
cleartool: Error: You do not have a license to run ClearCase.
```

Of course, one may wonder whether these operations make sense in the context of a disconnected snapshot view, in which the versions have already been selected and the choice may not be revised until next time the connectivity is restored. A snapshot is after all no more than a dead copy.

The hijacking mechanism is actually the accepted way to work around this state of things in the disconnected mode, with the promise to restore consistency later when connected. Suppose one is working in disconnected mode; then one can only change the permissions (remove the read-only flag in Windows) on a downloaded version in the snapshot view. ClearCase commands are of course still unavailable:

```
$ ct co -nc foo
cleartool: Error: Unable to contact albd_server on host 'ccserver'
cleartool: Error: Cannot contact license server host "ccserver"
cleartool: Error: You do not have a license to run ClearCase.

$ chmod 755 foo
$ echo foo >foo

$ ct ci -nc foo
cleartool: Error: Unable to contact albd_server on host 'ccserver'
cleartool: Error: Cannot contact license server host "ccserver"
cleartool: Error: You do not have a license to run ClearCase.
```

When connection is again established, one can spot the hijacked version:

```
$ ct update .
Processing dir "bar\www".
Processing dir "bar\www\cc".
.......
End dir "bar\www\cc".
.
Keeping hijacked object "bar\www\foo" - base "\main\4".
...........
```

```
End dir "bar\www".
.
Done loading "\bar\www" (28 objects, copied 0 KB).

$ ct ls foo
foo@@\main\4 [hijacked] Rule: \main\LATEST [-mkbranch b1]
```

The discrepancy may be resolved either by checking in or by removing the offending private file.

First, the later option (clean up the hijacking file):

```
$ rm foo
$ ct update foo
Loading "bar\www\foo" (1654 bytes).
.
Done loading "\bar\www\foo"

$ ct ls foo
foo@@\main\4 Rule: \main\LATEST [-mkbranch b1]
```

Alternatively, the former option, check it in (checking out first to a separate branch to avoid a possible conflict with a checkout by another user):

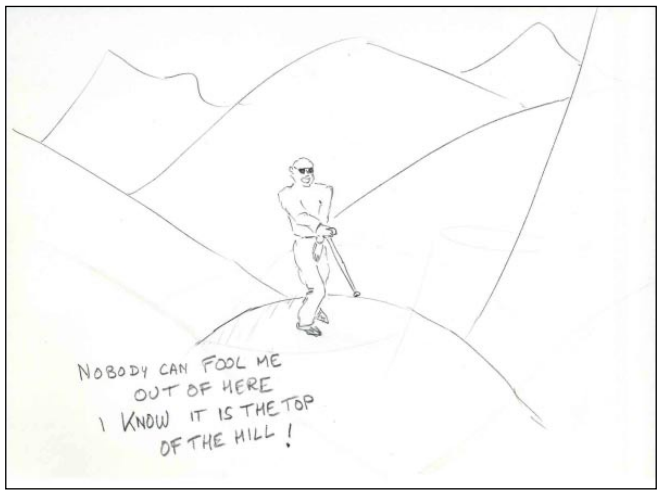
```
$ ct co -nc foo
Created branch "b1" from "foo" version "\main\4".
Checked out "foo" from version "\main\b1\0".
"foo" has been hijacked.
Do you want to use it as the checked out file?
(If not, it will be renamed.) [yes]

$ ct ls foo
foo@@\main\b1\CHECKEDOUT from \main\b1\0 Rule: CHECKEDOUT

$ ct ci -nc foo
```

Snapshot views present a real management challenge: the snapshot view directory is not recorded in the view storage, and may be virtually anywhere on the user laptop (or whatever accessible storage). This makes it impossible to service by administrators (backup, upgrade, and so on) even when the laptop is connected. This is especially true in the case of hijacked versions.

Building in snapshot views, clearmake behavior falls back to the traditional make model, using only local resources from standard file systems and producing view-private artifacts.

On a low level, this is more efficient than building in dynamic views in an MVFS file system: process communications are greatly simplified (only local), and neither auditing nor shopping for derived objects, with its sophisticated evaluation and decision process, takes place.

This appealed to many users, especially in cases when these didn't, for one reason or another, use or benefit from *clearmake winkin*. From our point of view, such users are effectively trapped, with no easy way out of their hole.



Express builds

The low-level performance gain obtained in snapshot views can be emulated in dynamic views as well, thus retaining the comfort of synchronous updates.
A first way is to use the `-v` flag of clearmake, restricting configuration lookup to the current view and thus disabling winkin. The derived objects produced are still recorded, with an incurred cost.

The next step is thus to create views with a special `-nsh` flag, instructing them to produce non-shareable derived objects. This way, clearmake is effectively downgraded to a standard make tool.

UCM

UCM was an attempt to raise the level of support, in other words to support "higher-level" concepts directly: components, as a way to structure sets of files; activities and tasks, as a way to attach some meaning to change sets; streams, as a way to relate and transcend labels and branches.

Unfortunately, it didn't build upon the existing sophisticated features of ClearCase (dynamic views, clearmake winkin, and derived object management), and resulted in lowering the build support.

It also forced users to make decisions upfront, and made it from hard to nearly impossible to revise them later.

It finally capitalized on a branching strategy, which was state-of-the-art at the time, but proved counter-productive in retrospect: one dedicated integration stream (and integration branch type), where all the changes should be *merged* to (*delivery*) from a number of development streams (branches). The latter would be kept in sync (again, by merging) with the integration stream (*rebase*). Streams did bring a welcome improvement over plain old labels, with a way to support *incremental* labeling, a functionality that we had unsuccessfully requested in base ClearCase and finally had to match with *label type families* in our ***ClearCase::Wrapper::MGi*** wrapper.

UCM claims to implement SCM *best practices*, and to raise the level of abstraction, so that the end user would not need to deal with low-level SCM tool concepts such as individual elements, and could access a representation of the whole software architecture.

Let's take a closer look at what these best practices are and how they are implemented.

Boot-strapping the simplest UCM project, be it only for one single user at the start, already involves quite a few steps. One must define and create at least the following minimal set of UCM artifacts: a project vob, a component within this project vob, an initial baseline for the component, a UCM project itself, an integration stream for the UCM project, an integration view on the integration stream, and an activity. Usually at least one development stream with a *separate* development view is created as well.

After creating and configuring all of the above, and getting all this finally to work, one can assume that the basic "out-of-box process" skeleton is in place. Now, changing one's configuration back and forth should be a piece of cake, so to speak. It turns out not to be quite so. One can happily make a code change in the development stream, deliver it to the integration stream, and create a baseline there. This one can also be *recommended* in the integration stream, and the development stream can be rebased with it. Usually at this point in the UCM literature the reader is promised an ocean of future possibilities such as compare baselines, merge baselines, create a child stream from the baseline you want, rebase the child stream with the recommended baseline, or even find the changes between baselines and undo/modify/redo them. But simple things first. Let us not be distracted from paying attention to one obvious thing: how easy it is to roll back, that is, what happens if you change your mind and would like to get back to the previous (initial in our case) baseline? Let's see: you seem to be able to recommend the initial baseline back in

the integration stream instead of the newly created one. So far so good! But when you *synchronize* your integration view with the stream (which is surprisingly needed even for dynamic views!), your view still selects the latest changes you made, which are not part of the initial baseline. It's time to check the integration view config spec (even this is discouraged in UCM: the only changes allowed to config spec are vob load rules, which don't make any sense for dynamic views of course):

```
$ ct setview intview
$ ct setcs -stream # Synchronize with the stream
$ ct catcs
ucm
identity UCM.Stream oid:a6cffe7b.6980496a.aaab.11:56:4d:2c:a4: ##########
                    b5@vobuuid:5af58e42.640210d5.b8a4.00:c0:61:20:6a:53 1

# ONLY EDIT THIS CONFIG SPEC IN THE INDICATED "CUSTOM" AREAS
#
# This config spec was automatically generated by the UCM stream
# "PROJ_Integration" at 2010-10-15T10:19:42+01:00.
#
# Select checked out versions
element * CHECKEDOUT

# Component selection rules...
element "[2ba58ea2640201c6a8a300d048306e57=/vob/bar]/comp1/..." #########
                                                  .../PROJ_Integration/LATEST
element "[2ba58ea2640201c6a8a300d048306e57=/vob/bar]/comp1/..." #########
                                          INITIAL -mkbranch PROJ_Integration
element "[2ba58ea2640201c6a8a300d048306e57=/vob/bar]/comp1/..." #########
                                          /main/0 -mkbranch PROJ_Integration

end ucm

#UCMCustomElemBegin - DO NOT REMOVE - ADD CUSTOM ELEMENT RULES ##########
                                                          AFTER THIS LINE
#UCMCustomElemEnd - DO NOT REMOVE - END CUSTOM ELEMENT RULES

# Non-included component backstop rule: no checkouts
element * /main/0 -ucm -nocheckout

#UCMCustomLoadBegin - DO NOT REMOVE - ADD CUSTOM LOAD RULES #############
                                                          AFTER THIS LINE
```

Our project's name is `PROJ`. The initial baseline of the component `comp1` was called `INITIAL` (and was based on the set of versions labeled with the label type `INITIAL`). The new baseline we just created has the name `BASELINE1`, which is actually a ClearCase label type `BASELINE1` attached to the newly created set of elements. But ...it doesn't appear in the config spec of the integration stream! And actually no matter how many baselines one creates or recommends in the integration stream,

none of them will ever be mentioned explicitly in the integration stream config spec: the integration stream is always `. . ./PROJ_Integration/LATEST`; that is, no matter how you configure your project, you will always see only the latest versions in the integration stream and this cannot be configured differently! Hardly a best practice, is it? By recommending the baseline one only records the current label, which is a good idea in itself, but the record does not go any further, as far as the integration view is concerned! This also means that if one chooses to work in a "minimal" UCM project having just one integration stream, and no development streams, there is no way to manage one's configuration at all.

Now it should be a right moment to recall that IBM Rational recommends not to modify or even look at UCM config specs ("because you don't need to")—we may now understand better why...

So, the UCM integration stream *always* points to the latest versions. If you want even to take a look at a previous baseline, not to mention working on it, you need to do it in a separate child (development) stream.

So, how about the development stream? First we rebase it:

```
$ ct setview devview

$ ct rebase -baseline BASELINE1
   Advancing to baseline "BASELINE1" of component "comp1"
   Updating rebase view's config spec...
   Creating integration activity...
   Setting integration activity...
   Merging files...
   Checked out "/vob/bar/comp1/foo" from version ##########################
                                                "/main/PROJ_development/4".
   Attached activity:
    activity:rebase.PROJ_development.20101112.115451@/vob/ ###############
            pvob  "rebase    PROJ_development on 11/12/2011 11:54:51 AM."
   Needs Merge "/vob/bar/comp1/foo" [to /main/PROJ_development/ #########
CHECKEDOUT from   /main/PROJ_Integration/3 base /main/PROJ_development/2]
   ******************************
   <<< file 1: /vob/bar/comp1/foo@@/main/PROJ_development/2
   >>> file 2: /vob/bar/comp1/foo@@/main/PROJ_Integration/3
   >>> file 3: /vob/bar/comp1/foo

…
$ ct rebase -commit
```

The new baseline is visible in its config spec, so this is already something to start with:

```
$ ct setview devview
$ ct setcs -stream
$ ct catcs
ucm
identity UCM.Stream oid:595d3b4a.bb0943f1.23d4.15:ac:01:e8:73: ##########
                    52@vobuuid:5af58e42.640210d5.b8a4.00:c0:61:20:6a:53 3

# ONLY EDIT THIS CONFIG SPEC IN THE INDICATED "CUSTOM" AREAS
#
# This config spec was automatically generated by the UCM stream
# "PROJ_development" at 2010-10-15T13:57:05+01:00.
#
# Select checked out versions
element * CHECKEDOUT

# Component selection rules...

element "[2ba58ea2640201c6a8a300d048306e57=/vob/bar]/comp1/..." #########
                                                    .../PROJ_development/LATEST
element "[2ba58ea2640201c6a8a300d048306e57=/vob/bar]/comp1/..." #########
                                          BASELINE1 -mkbranch PROJ_development
element "[2ba58ea2640201c6a8a300d048306e57=/vob/bar]/comp1/..." #########
                                           /main/0 -mkbranch PROJ_development

end ucm

#UCMCustomElemBegin - DO NOT REMOVE - ADD CUSTOM ELEMENT RULES ##########
                                                          AFTER THIS LINE
#UCMCustomElemEnd - DO NOT REMOVE - END CUSTOM ELEMENT RULES

# Non-included component backstop rule: no checkouts
element * /main/0 -ucm -nocheckout

#UCMCustomLoadBegin - DO NOT REMOVE - ADD CUSTOM LOAD RULES #############
                                                          AFTER THIS LINE
```

Okay, this looks alright. But then can we change it back to the initial baseline? The answer is NO. See what happens if we try.
We recommend back the initial baseline for the integration stream of our PROJ project. And then trying the rebase:

```
$ ct pwv
Working directory view: devview
Set view: devview

$ ct rebase -recommended
```

```
cleartool: Error: Can't revert to earlier baseline "INITIAL" ############
                  of component "comp1" because the stream has made changes
                  based on the current baseline.
cleartool: Error: Unable to rebase stream "PROJ_development".
```

Why should this be impossible? This is because there is a hidden trick in the above mentioned development stream config spec: what UCM actually does at rebase (and similarly at the deliver as well) is physically merging the versions carrying the label `BASELINE1` to *all* the versions already having a `PROJ_development` branch. That means the `element ... BASELINE1` rule in the config spec (`element "[2ba58ea2640201c6a8a300d048306e57=/vob/bar]/comp1/..." BASELINE1 -mkbranch PROJ_development`) comes *too late*: only in second. It only applies to versions not having a `PROJ_development` branch yet, that is, that have never been checked out in the views on this development stream. That is why reverting to a previous baseline `INITIAL` is simply not possible, as it is not possible to "un-merge" the latest versions on the `PROJ_development` branch from the ones carrying `BASELINE1` label. Without that harmful merge, the revert would be just a matter of replacing the `BASELINE1` with `INITIAL` in the development view config spec.

The only possibility to make use of the former baseline is to create a *new* development stream (`PROJ_dev1`) from the integration one using the needed baseline (`INITIAL`) as its foundation (that is to branch off a new branch from the integration one using the `INITIAL` label).

Likewise, the UCM components must be read-only in a project in order to be able to change their initial baseline later on. Otherwise, in case of any changes (even the dummy ones) made to a component, rebase becomes forbidden. Furthermore, removing a component from a UCM project stream would typically not be possible either:

We must admit such practices are quite commonplace indeed, but can they still qualify as *best*?

It already seems that UCM makes its motto from: "no way back", and we'll be back on this.

But the UCM delivery by merge model is even more questionable. This is especially felt in large projects, involving many developers.

Any element version change in UCM has to be reported to an *activity*. Generally speaking, having activities as a means of tracking one's own (or others') changes is not a bad idea in itself. Developers can find it quite useful for their own bookkeeping purposes.

Performing a delivery to the integration stream takes place in the scope of whole activities: their *change sets* are being delivered (merged) to the integration stream. This delivering by merge occasionally brings surprises, as we explained in *Chapter 7*, *Merging*: presumably fixed bugs may re-appear either in the integration or in the rebased development streams. Non-trivial merges hidden among massive amounts of trivial ones may yield, on either the delivery or the rebase way, what appear as random results. Bulky merges from multiple development streams may interfere with each other, and a bug fix delivered from one development stream may unintentionally get overwritten so that the original bug is restored.

Preventing non-trivial merges becomes the goal of policies such as forcing a rebase prior to delivery. This is however a two-edged sword, as it affects the data being delivered that ought to be re-tested. This in turn extends the overall delivery time, thus increasing the risk of collisions, which will force rebases, *ad nauseam*.

It is sometimes recommended to lock the shared integration stream while building in it to validate the result of one's delivery before *completing* it; this to ensure a tight serialization of deliveries, prevent activity changes, and avoid to find numerous elements checked out as an effect of other pending deliveries. Both alternatives of either locking or leaving it open to changes have significant drawbacks.

Do we need to remind our readers that these problems affect radically less *in-place deliveries* (refer to *Chapter 6*, *Primary Metadata*), based on labeling, which are faster, thus offer less occasions for collisions, are reversible, hence offer the option to solve the problems out of the way of competitors, and do not modify the data therefore do not require a new testing phase as part of the delivery process proper? The scalability problems we describe here, growing faster than the size of the system by any measurement, are typical of UCM, not a fatality of parallel development in ClearCase! One should remember not to mix the actual parallel development concept and UCM or UCM-like ways of using base ClearCase.

The intention behind UCM activities was interesting, but unfortunately ClearQuest integration largely impacted their usefulness. ClearQuest imposes a number of additional restrictions to the already tight UCM environment. Typically, the ClearQuest-UCM integration is used to impose a number of policies requiring that all the activities are created beforehand in ClearQuest by the designated roles (such as *project manager*) and preferably assigned to the developers: in the best case a developer could try herself to select one out of a set of activities created in advance. Additional restrictions can apply to activities delivery, status change, and so on. This

is clearly an expensive (and very heavy) overkill. It tends to distract developers, who end up finding workarounds and developing their code in their own "sandboxes", outside of UCM and ClearQuest altogether.

Let's now take a brief look at a few typical UCM problems or rather use cases.

Even apart from merge-related issues, and whether with ClearQuest integration or without it, delivering a dedicated subset of one's activities can be quite challenging. Although one can reassign the modified versions from the change set of one activity to another, this is often not enough to eliminate some *dependencies* (for example, one of the activities contains a change to a version of the vob root directory) between the activities, so one is often forced to make a bulky delivery including *all* the activities:

```
$ ct deliver -act act1
cleartool: Error: Activity "act2" must be added to activity list.
cleartool: Error: Version "/vob/foo@@/main/PROJ_development/3" ##########
      from the activity "act2" is missing from the required version list.
cleartool: Error: The list of activities specified is incomplete.
cleartool: Error: Unable to deliver selected activities.
cleartool: Error: Unable to deliver stream "PROJ_development".
```

For delivering a set of activities, as nearly for everything in UCM, the recommendation is to use the (slow...) GUI. When the project grows, the usability of GUI operations, such as delivery, suffers, and one starts to meet all kinds of strange errors, like shown in the next screenshot:

The command line delivery command may help in this situation:

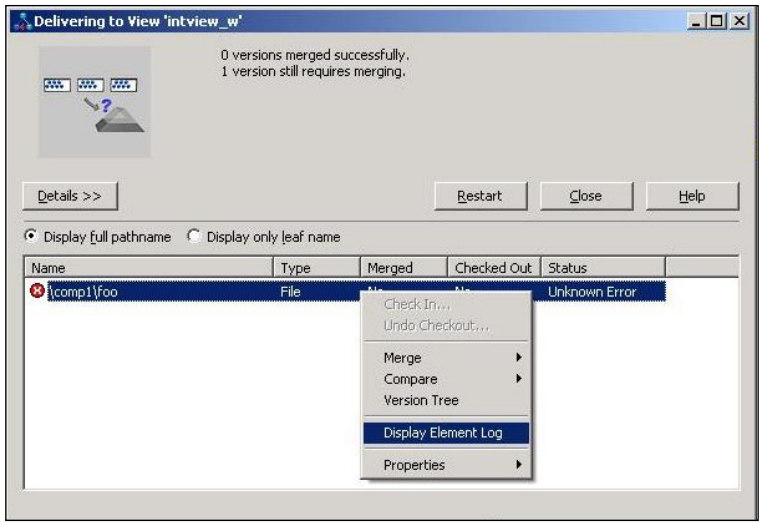
```
$ ct deliver
Changes to be DELIVERED to default target stream in project "PROJ":
 FROM: stream "PROJ_development"
 TO: stream "PROJ_Integration"
Using target view: "intview".
Activities included in this operation:
  activity:act1@/vob/bar joe "act1"
```

Every UCM stream requires a separate view per user involved. Several UCM projects co-exist usually in parallel (for example, a main release project, a maintenance project, and so on), and every project can have many streams (some of them for the purpose of branching off a former baseline as explained above), so the number of views grows very soon. The view names tend to be very long (for example, `userid_ucmproject_int`, `userid_ucmproject_dev`) to allow distinguishing between one another's views and even between one's own views, which also becomes a source of confusion and errors. Especially in the absence of clear transcripts and logs, due to the encouraged use of GUIs, even simple situations can become puzzling for not very experienced users.

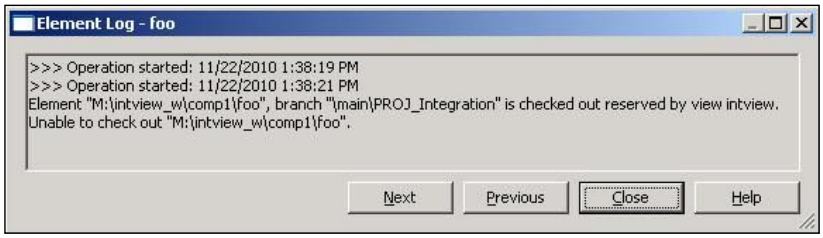
Let's give an example scenario. A user tries to *deliver* (merge) her *activities* from her *development stream* using her *development view* to the project's integration stream. She gets an error message saying that the element `foo`, which she attempts to merge, is in a checked-out state. Actually, even getting this information is not obvious; as with the *ClearCase Project Explorer* GUI, it is hidden behind the red icon next to the element name in the *Merge Tool*, and the *element log* (actually the GUI pop-ups below are Windows UCM client-specific, as the similar UNIX interface seems to be somewhat less messy):

There are not many options except for clicking **OK** on this not too descriptive pop-up, which results in still one more similar one:

After this a third pop-up window follows, with at least some details this time, but one must know to right-click on the item having the red icon on the left and choose **Display Element Log** to get any actual information about the problem:



And finally one is rewarded with the actual problem description (note that log reading abilities are required though):



And if using Eclipse with the *ClearCase plugin*, the message she gets asks her to do a "resource restoration" ("resource" is the Eclipse term for ClearCase *element* and "restoration" presumably stands for *checkin*). But trying to "restore resource", by *updating* her development view, doesn't help, as she cannot see any problem with her `foo` element and it does not seem to be checked out. The user might conclude that there are some synchronization problems between the repository and her view. The actual problem is that the element is being checked out in another view, for example, in one of the integration views (hers or somebody else's). As the integration stream always uses the `.../Integration_branch/LATEST` model, as we explained above, such an element left in a checked-out state can prevent users from making deliveries. This is just an illustration of a very simple problem. And such minor issues turn out to be very tricky and time-consuming to solve because of the artificial and unmanageable complexity of the UCM environment.

The use of integration branches, instead of main ones, suggests a MultiSite setup: the advantage is to allow the coexistence of several integration streams, whereas there can only be one single *main* branch at the root of any element. Integration streams are bound to projects, so that in order to have two, one needs to create two different projects, based on the same components, enabling inter-project delivery policies: each project performs inter-project delivery of the other project's baseline to its own integration stream. This helps avoid the unmanageable *posted* deliveries (the recommended practice under MultiSite), one other major woe of UCM. Another option could be to have a project with a dummy integration stream and two child (development) streams, each mastered at its own site, and to perform inter-stream deliveries from each other's baseline to one's own stream.

As we already mentioned in Chapter 9, the standard ClearCase *clearimport*, *clearexport*, and *cleartool relocate* utilities do not work with UCM vobs (*clearfsimport* does work).

A last note, well known as it is: one can easily convert one's vobs to UCM, but there is no way back. There is no documented way to convert a UCM vob to base ClearCase.

If the vob has originally been converted to a UCM component from a base ClearCase vob, one can make it a base ClearCase vob back, by explicitly removing the hyperlink to the Admin UCM vob:

```
$ ct des -l vob:/vob/comp1
versioned object base "/vob/comp1"
...
 Hyperlinks:
  AdminVOB@738@/vob/comp1 -> vob:/vob/pvob

$ cleartool rmhlink AdminVOB@738@/vob/comp1
cleartool: Warning: An AdminVOB hyperlink to a UCM PVOB is being removed.
This can cause serious problems with UCM.
If desired, this hyperlink may be replaced using the command:
 cleartool mkhlink AdminVOB vob:/vob/comp1 vob:/vob/pvob
Removed hyperlink "AdminVOB@738@/vob/comp1".
```

Note that the UCM branches and versions are preserved in the vob (and can be accessed, for example, with version-extended path names), so a manual conversion is possible:

```
$ ct lstype -kind brtype
--11-22T12:30 joe branch type "main"
 "Predefined branch type used to represent the main branch of elements."
--11-22T12:53 joe branch type "PROJ_development"
--11-22T12:39 joe branch type "PROJ_Integration"
```

```
$ ll .@@/main/PROJ_Integration/1/foo/main/PROJ_Integration/ ##############
                                                          PROJ_development
total 6
-r--r--r-- 1 joe jgroup 0 Nov 22 12:58 0
-r--r--r-- 1 joe jgroup 4 Nov 22 12:59 1
-r--r--r-- 1 joe jgroup 4 Nov 22 12:59 LATEST

$ ll .@@/main/PROJ_Integration/1/foo/main/PROJ_Integration/ ##############
                                                        PROJ_development/1
-r--r--r-- 1 joe jgroup 4 Nov 22 12:59 .@@/main/PROJ_Integration/1/ #####
                              foo/main/PROJ_Integration/PROJ_development/1

$ cat .@@/main/PROJ_Integration/1/foo/main/PROJ_Integration/ #############
                                                        PROJ_development/1
foo
```

We tried here to show with concrete examples how UCM submits developers to additional constraints, hence increases the overall complexity of the development environment, and reduces developers to a passive role. We are deeply aware of the fact that we shall have failed to convince aficionados (and there are many). These might retort either with further escalation (creating yet another stream as an example) or blaming decisions already taken (the processes for ClearQuest integration already in place). What we face here is a case of non-*falsifiability*. The net result of this failure to achieve undisputable conclusions will in any case be that developers will leave the field to experts: the tool is not *their* tool, it is imposed on them and felt as an external requirement they have to satisfy.

UCM has systematically been presented as "enhanced" ClearCase. This is a myth, which may only hold superficially. UCM not only ignores the main assets of ClearCase but it undermines them. The clearest case is maybe how a delivery by merge forces a full rebuild by failing to promote the derived objects produced in the development branches. This could be addressed in either of two ways: by performing the rebuild as part of the delivery before the baseline is updated, or by letting everybody compete to build. The former would make the delivery longer yet, and is thus not chosen in practice. The latter results in race conditions and thus the production of equivalent derived objects, which pollute the DO pool and jeopardize winkin at large. This is again only an example. Other examples could be the impossibility to exploit labels produced with `mklabel -config` or the general inability to handle the config specs of UCM views. The bottom line is clear: UCM is deeply incompatible with an effective use of base ClearCase, as the one we have tried to present throughout this book.

Web access and remote clients

Internet drives and federates the best news software brought to us during the past 10 years. Hence it's a must to provide web interfaces, that is, to allow accessing ClearCase from the new universal tool: the browser. This does of course bring some real benefits: making ClearCase accessible from mobile phones (although the smarter ones provide a decent terminal interface), tunneling access through one single port (ouch), and allowing to support a centralized model, including over WAN connections.

The two latter arguments are actually good news only from an administrator's point of view, especially with the mythical, but so easy to sell to management, 'security' concern in mind, but maybe also with the similar *cost saving* one (that is, from *absolute* points of view, which do not have to take anything else into consideration, such as securing what? Or what does it cost to save?)

Anyway, there's already been several waves of web interfaces to ClearCase, and more are expected. This is actually one of the most active development areas. ClearCase comes with a *Rational Web Platform* (the ClearCase Web server, under `/var/adm/rational/clearcase/ccweb` directory), which allows for creating *web views*: a special kind of snapshot view with its storage located on the dedicated web server, and the view root directory on the local client workstation.

The web server must run a *ClearCase client*, and be connected to the license, registry, and vob servers.

The user must just have an account on the web server: no ClearCase client installation or connection to the vob/registry server is needed.

The original interface (so called ClearCase Web Interface), now obsolete, worked in a standard browser.

A *Rational ClearCase Remote Client* (CCRC) is also available, accessing the web server, but offering more sophisticated functionality and running locally on the user workstation. A first version was a *native* client, now replaced by one based on Eclipse technology. It may use web views.

The ClearCase Web interface stopped being supported before it was taken away. Where one can still access it, one may expect to get all sorts of Java exceptions, such as `Error: "java.lang.IllegalStateException: Timer already cancelled`, maybe related to the version of Java used on the browser. This interface always was extremely limited, even comparing to CCRC. CCRC has been the major focus of attention lately. In the latest releases even some initial command line interface has been implemented (it had always been just a GUI).

CM API

We didn't mention the CM API, which came out with v7.1, as we have no experience of using it.

It is not the first attempt at offering an API to extend ClearCase. The very first one was a C API. Then came the ***ClearCase::CtCmd*** Perl package, distributed on CPAN and intended to be linked with the ClearCase shared libraries; next the *Rational ClearCase Automation Library (CAL)*, exclusively on Windows.

Let's note that neither CtCmd nor CAL were compatible with *Perl/ccperl*, the Perl bundled with ClearCase installation respectively on UNIX/Windows (actually, the version of ccperl distributed with v7.x has the `Win32::OLE` required to access *CAL*, as has *Common/ratlperl*).

ClearQuest came with its own cqperl, different from ccperl. Both tools now use ratlperl, with cqperl and ccperl being kept only for backwards compatibility.
None of these attempts at offering an API to support customer extensions and customization were wholly satisfactory.

The CM API is a Java API, and it is common to ClearCase and ClearQuest.
We understand it is bound to the CCRC (thus still limited in functionality), and intended to supersede the use of cleartool (frightening to us as this may sound!). We do not feel compelled (yet?) to jump on this bandwagon.

We further skip products farther away yet from the ClearCase "umbrella"—Build Forge (the relatively recently acquired *Continuous Integration* offering of IBM) and Rational Team Concert. To us, both products show the disaffection of ClearCase concepts, and replace noticeable parts of its architecture with new tools. For instance, Build Forge does address some of the missing functionalities in clearmake support, for Java and web development. It does it in ways incompatible with the specific solutions in clearmake.

Summary

In this chapter, we didn't so much give our readers assistance to make use of the features of ClearCase we dealt with, but rather, as it happens, reasons to avoid them. The bulk of the new functionalities are centered around the UCM extension, which we consider ...catastrophic.

Working with UCM doesn't mean UCM working for you: it is you who are working for UCM.

Our advice is bluntly simple: do not use UCM.

ClearCase was, and 20 years after its conception, still is a revolutionary product. Being revolutionary, it took the risk of showing a different path: putting SCM upside down and making information emerge from relationships audited at build time, instead of flow down from intentions expressed at design time. In its attempt to conquer the wide hence profitable market of PC-based software development, Rational, and then IBM, refocused the ClearCase product away from its avant-garde concerns back into the traditional mainstream. It is not that the problems that ClearCase was meant to tackle would have been solved, quite on the contrary: the mainstream chose to ignore them to bury them under more fluffy activity, bureaucracy, and colorful reports. It is a fallacy to pretend that this shift could happen without penalizing the idiosyncratic ClearCase. For 10 years, we have been waiting for a resolution of the conflicts. It is time now to admit that the chasm can only get wider.

Our conclusion could seem to be a very negative view of ClearCase if it wasn't balanced with all the useful and promising functionality we have found in ClearCase, and which we have described at length in this book. In this context, our advice is only the most effective one we may deliver.

ClearCase Future

We tried hard in this book to be practical, to **show the code** as the Open Source precept mandates. We wish our reader will indulge us now the right to draw some more abstract, more theoretical conclusions, but bear with us until the end: we'll drive back to practical proposals.

There are good reasons to feel concerned with the future of ClearCase.
As we saw, there are reasons as well to believe that ClearCase has still much to contribute to SCM, which itself has by far not got the attention it deserves in today's software development. So, in short, what place may ClearCase have in the future; does it even have any?

This question is a natural, and a difficult one, for people having already invested in ClearCase, or thinking of acquiring competences in SCM.

It is commonplace to read extremely negative comments on ClearCase, especially in Open Source contexts.

We'll consider ClearCase from three points of view—of IBM, of software engineering at large, and of the Open Source. Then we'll draw our conclusions, and offer an agenda to our readers.

ClearCase and IBM/Rational

With UCM, Rational and IBM drove ClearCase into a dead end.

Delivering *back* to an integration stream has only disadvantages, compared to delivering in-place by moving (or applying) delivery labels, in terms of (not coming back to developing each of these points) the following:

- Performance
- Safety/robustness
- Reversibility
- Support for collaboration over MultiSite
- Promotion of derived objects

This *idea* was a non-idea, based on an accident of history: version control systems first had one single stream with successive versions, before they were used for collaboration, and had to support parallel development. Version control grew from a personal backup instead of from a communications tool.

Merging back came naturally as a way to piggyback the new functionality as an extension to existing systems. It should be obvious that it brings *back* all the problems that branching off helped to solve.

Deliver by merging back is not the only reactionary idea in UCM. Another idea is to solve problems with *specific* instead of *generic* solutions (a Copernican revolution in reverse): hence the concepts of *project* and of *component*, in addition to that of *element*, whereas the concept of **derived object** offered an elegant and generic way to bring structure into software configurations.

A last reproach we'll do to UCM concerns its focus on *control* instead of on *management*. Or taking the word Brian White uses in his presentation of UCM, *Software Configuration Management Strategies and Rational ClearCase—on enforcement*, which he defines as follows (2.1.3, p 21, in the first edition): *enforcement is a proactive control that disallows changes unless certain conditions are met.* The two approaches are opposite: *do (only) what you control*, instead of *manage what you do*. Control is invasive, and ultimately counterproductive: people will keep doing what one intended to prevent, but they'll do it underground. At the end of the day, the more you control, the less you manage. Control cannot of course be completely excluded, but it should be clear that one ought to control only what one cannot manage.

We explicitly recommended to stay away from a major part of the ClearCase functionality, but it must be noted that IBM recommends in practice to abandon an equally large part of it, the one we value: from clearmake to dynamic views and MultiSite. This was obvious in the subset selected for ClearCase LT. It is fully consistent with all recent developments.

IBM recommendations to users are implemented in the integration to Build Forge, in the CCRC, or the CM API. The new Rational Team Concert, which has been actively promoted recently, is totally disconnected from ClearCase. Finally, it is striking how little IBM seems to use ClearCase internally.

ClearCase and the software crisis

Forty years ago there was proclaimed a *software crisis*. It was the theme of a historical NATO Software Engineering Conference in 1968, at which Doug McIlroy presented a paper, *Mass produced Software Components*, in which he advocated a solution based on two concepts:

- Software components
- Software factory

Several factors have since then radically obscured the situation, so that the concern lost its acuity. The most significant is of course the explosion of the Internet, which on one hand brought in a mass of fresh developers and on the other made the software market extremely attractive to venture capitalists (with a concern for "professionalism"—see below). This helped greatly with the implementation of the concept of software factories. Graphical user interfaces (emphasizing intuition against reason) and tools developed for the mass market were imposed on developers as well, which turned them into specialized workers in the most Taylorist tradition.

The call for software components was also heard; it gave rise to Object Orientation, and deeply influenced the development of SCM. Both of them are concerned with the identification of items, with the reduction of duplication in the representation of information, diagnosed as a cause of degradation of the signal per noise ratio, and with the management of relations between the identified items. Both of them hold *separation of concerns* as a basic principle.

It wasn't understood at once that the concept of software factories could run against that of components, due to the nature of software. One traded the competence of hackers against the professionalism of subcontractors, and failed to admit the difference between production and reproduction.

We would argue that this led to a dramatic loss of productivity and of quality, only hidden under the explosion in the amount of software, and that the software crisis problem was never solved. Software development has been submitted to arbitrary pseudo-business requirements, which only increased bureaucracy.

The state of the art in terms of parallel development regressed: once well-established knowledge was lost, and is only being discovered anew, as the failure of naive technologies (for example, *continuous integration*) becomes apparent.

We believe that we need to face the facts and hope we will soon again be in a position to state as Edsger Dijkstra during the 1968 conference:

> *The general admission of the existence of the software failure in this group of responsible people is the most refreshing experience I have had in a number of years, because the admission of shortcomings is the primary condition for improvement.*

This is where ClearCase has a role to play.

ClearCase and the Open Source

ClearCase suffered from its being proprietary, and from the commercial strategies of its successive owners.

These vagaries indeed impacted its users, to the point of raising the question: is it sensible for a commercial company to be captive from the vendor of its SCM tool? Doesn't Open Source, or rather Free Software, offer a natural way out of this dilemma?

The question is interesting, but given the emerging situation we described in the previous section, such a rational thinking is eclipsed by more mundane and short-sighted concerns (saving at all costs): since there exist gratuitous alternatives, who will still pay for SCM products?

Both *Free Software* and *Open Source* are, if not trademarks, at least clearly identified keywords. The historical distinction between them is however only marginally relevant from an SCM point of view. We base our distinction not so much on licensing terms, but on the practical manageability of changes, which is not granted by simply publishing the sources: Java and Subversion are examples among others, of products, the source code of which is publicly available, yet are tightly and effectively controlled. Far from us to consider that the focus of Open Source (as the name says it) on intellectual property issues was not valid (and the existence of patents on certain aspects of ClearCase is there to remind us of it) but there are other ways to *close* software than restricting access to the sources. In these two examples: respectively the complexity of the build environment, and the control

of the integration tree. Let's however not focus exceedingly on speculations about the intentions of the vendor/owner: what closes the product is the lack of manageability from the potential contributor's point of view, unable to test the integration of her changes, which at best she may hand to a dedicated *committer*.

We develop below (as a wrap-up of a theme already covered in this book) on the inadequacy of focusing SCM on sources.

The Open Source has, at least as much as commercial software, been impacted by the resurgence of the software crisis. SCM concerns were completely obliterated by the frenzy of the first years, and left to historical tools, tied to the original version control paradigm. The recent years have seen the birth of a plethora of candidate successors, unfortunately mostly targeted at offering GUIs and hiding the disturbing complexity of builds behind template processing, thus turning it into complete chaos (from the point of view of an end user who would like to analyze the results beyond the surface presented by the GUI: What was kept, what is new? If something was shared, whom is it shared with? And so on).

The burning SCM challenge, in the context of free software, is to allow everyone, and not only a mythical administrator, to manage the contributions of others, which constitute the main source of complexity. Managing in this context means in practice *ignore safely*: how to efficiently spot in the mass of contributions the ones having enough value with respect to one's own, for one to invest time to review them in detail and to ensure convergence with them. The easy part there is the traditional merge-build-test; and the hard one is communicating the result to others in a way which saves their time: we are back here to the techniques shown in our *Teaser*!

One thing free software has achieved (beyond freeing maintenance from the ties to an original vendor) is to supersede the concept of *standards*. The most convincing example is obviously the victory of TCP-IP over the OSI 7-layer model of telecommunications. The case of the GNU tools, and first of all of *gcc*, the GNU C compiler, which was instrumental in making Linux possible, is almost as compelling. We have been using for our tools, our experiments, and our proof-of-concept developments, the architecture based on CPAN ***ClearCase::Argv*** and ***ClearCase::Wrapper***. In the advent in which we'd have to leave the home of ClearCase, the most promising platform towards which to target our porting efforts seems today to be *git*, maybe with *Audited Objects* (mentioned in *Chapter 12, Challenges*) as the key build management tool.

ClearCase is dead, long live ClearCase!

We highlighted functions of ClearCase that we consider essential, although they are commonly overlooked.

In fact, precisely *they* build up the revolutionary aspect of ClearCase which, it has to be said, escaped even its authors.

The crux is the **winkin** functionality of the **clearmake** build tool. It was designed for performance, without anticipation that it might in fact provide a way to uniquely **identify**, under the control of the tool, the **derived objects**.

There is no point in identifying transient and private artifacts, but winkin promotes derived objects to a shared status, so that their lifetime expands and their identification suddenly makes sense: it allows users to compare results produced in similar yet slightly different contexts, by developers possibly not aware of one another.

Identification is the most fundamental function of SCM, and traditionally, it could only be based on the source files (the only —relatively— stable artifacts). What the whole SCM community overlooked was the possibility offered by the clearmake mechanism to base identification on the real valuable assets (the deliverables, mostly the executable ones), and thus to put SCM back on its feet: upside-down!

The reversal of perspective is complete if one accepts the idea that the derived object in ClearCase is a better prototypical match for the abstract concept of *configuration item* than the source file. ClearCase uses a different scheme to identify derived objects and "source files" (versioned elements), but which scheme is in use doesn't matter once it takes the responsibility for the identification: the idea of basing the version identification onto a numbered position on a branch is sometimes simply inadequate. This becomes apparent (for a practical example) in a corner case of the use of the ***synctree*** tool: using the `-vreuse` option (in conjunction with the `-label` one); refer to the *Import* section of *Chapter 8*, *Tools Maintenance*. This option attempts to avoid creating a new version, by *shopping* for an existing and suitable one, to which it merely applies a label of the provided type. Here, the exact branch and version number are irrelevant: all the version tree is considered as a pool. One ought to recognize here the logic of winkin, normally used for derived objects, but now in the context of importing an external "source file" hierarchy.

The legacy of ClearCase

Software development involves a chain of processes from editing *sources* to producing *deliverables* that can be distributed, installed, and eventually run. Traditional management systems are based on *version control*. ClearCase on the contrary, is based instead on *build management*, which is largely ignored by the competition, as well as by the UCM extension. In the traditional view, build management is only the *production* of software artifacts, which nobody attempts to manage directly, only relying on a *cookbook* approach. ClearCase, and precisely clearmake, makes it possible instead to focus on *reproducing* as much as possible *configurations* contributed by others, so as to make differences emerge in the context of a clearly supported (identified) stability.

This is in short what needs to be retained of ClearCase, into what we could term as *ClearCase HT* to escape the misleading *base ClearCase*. In a conservative way, which might be proven wrong, this compound functionality encompasses the build tool, **clearmake**, **dynamic views** (in order to support referential transparency in the build scripts and documents), and **MultiSite** (in order to guarantee that all synchronous interactions audited in the build transaction only involve a local server).

Some errors to avoid, or the limits of ClearCase

User errors are always grounded somewhere. It should always be easier, cheaper, and safer for the user to do *the right thing*, than to find a workaround for it. Often, the most stupid processes start with a good idea, or a reasonable concern; it is extrapolating (scaling) it that leads to the problems. So, it is worth trying to understand the motivation behind the requests, instead of just (or in addition to) granting them.

The implementation of the software factory brought in weak professionals. A culture of *impatience with irresolution* (quoting the TV producer David Milch, himself quoted by Dan Meyer in a ***TED talk***), which encouraged *intuition*, implicitly blaming people for not understanding at once, and discouraging them from spending time to reason. A culture welcoming *magic* and, confusing responsibility with guilt, favoring dumping part of one's job on others, over exerting one's own freedom. There are certainly no silver bullets to cure this situation, but the resolution should be to educate the users out of this hole, not to help administrators to keep them in it.

Why many users despise ClearCase config specs? Because config specs are needlessly complex (too feature rich syntax), because they are not versioned, so that the changes are hard to revert.

Why do people pack information into names? Because they don't know how to get it from elsewhere, for example, from comments, or better from named attributes. The `-fmt` option is very powerful, but not obvious to find. Besides, comments may be multi-line, which is poorly supported (you feel it like a punishment when the comment contained 500 lines of useless text). This certainly applies to many names within ClearCase, from types to replicas, and thus appeals for different enhancements.

Users want to send changes by mail. Why is it so? We would tend to reply: impatience, with a reference to the quote, earlier in this chapter, but we cannot stop there for several reasons. First, *impatience* is not necessarily a bad thing, at least according to Perl philosophy, which makes it a *virtue*. The difference is subtle between *solving* problems, and *getting rid* of problems. We'll try now to focus on the next point, the point which might be blamed on ClearCase. Before we do so, however, we have to clear up several issues which make the problems harder to identify and to solve, yet, we shall argue, do not *cause* them.

These issues are ones we dealt with at length in this book. Let's mention them briefly once again.

As we explained already in *Chapter 11*, *MultiSite Administration*, the off-the-shelf setup of ClearCase MultiSite does not scale. One-to-all synchronization very soon causes traffic jams among the sync packets, making the actual problems (missing packets) difficult to locate and long to solve.

In addition, development practices based on the centralized integration branch concept, aggravated with extensive needs to move masterships between the sites, ruin the whole idea of distributed development (as exemplified by the infamous *posted deliveries* of UCM). We showed in Chapter 11 how to use smart MultiSite hubs, and truly distributed processes to decouple and decentralize development.

At the bottom, we however identify a conflict between two levels of synchronization: the user wants to (synchronously) solve her problem at hand, across several sites, whereas MultiSite replication tends to impose some other synchronization (or serialization, which is the same thing: strict ordering of events) at a vob level: exporting and shipping changes implies successfully processing all the previous changes pertaining to the same vob. This results in the fact that merely letting the user run an export job via the scheduler (which certainly is an option) would be unlikely to satisfy her (or the system administrator, because it would raise the overall load).

The problem here, in our opinion, is a missed opportunity to *manage* the changes, instead of once again, merely *controlling* them. This time, *management* takes the name of *laziness* (another Perl virtue, now opposed to ClearCase *eagerness* to synchronize events beyond the ones directly interesting the user). We do not want to design here a solution to this problem, only to sketch a direction for a development which might be beyond what may be implemented on top of ClearCase: database requirements are maybe too close to the surface. As one cannot prevent the user from sending changes by mail, the system ought to support delivering them with equal or superior convenience in a way not detrimental to others. Laziness in this context might mean offering a way to accept temporary inconsistencies, with the promise of resolving them *later* (for example, when the actual sync packets are finally imported).

Laziness would by the way also apply to the case of re-conciliating *equivalent derived objects* (see the *Removing derived objects* section from *Chapter 3*, *Build Auditing and Avoidance*).

Being proprietary, ClearCase has closed itself from other tools. There is a lack of conversion tools to other systems, and especially of tools helping to keep in sync with non-ClearCase repositories (needed, for example, with partners not necessarily using ClearCase), a lack of a standard replication protocol. ClearCase is paying now for its isolation.

Let's work together

We showed in the book that there is a great deal of power to develop ClearCase with or without support from the vendor, by using Perl and an architecture based on CPAN modules and culminating in wrappers. This allows the community of users, first to survive the end of the development we witness every day, but also to experiment and design the next generation Software Configuration Management, which should be based on the breakthrough ClearCase once brought in. This is not only an opportunity for ClearCase experts, but a key to bridge the current gap in software development at large. A challenge to which we invite our readers, and which we wish could involve IBM.

Appendix

Chapter 1

http://neuroeconomics-summerschool.stanford.edu/pdf/KAHNEMAN1.pdf (p 15)
http://search.cpan.org/perldoc?ClearCase::CtCmd (p 19)
http://search.cpan.org/perldoc?C learCase::Wrapper::MGi (p 22)
http:// search.cpan.org/perldoc?ClearCase::Wrapper (p 22)

Chapter 2

http://www.vestasys.org (p 30)
http://sourceforge.net/projects/audited-objects (p 30)
http://www.ibm.com/support/docview.wss?uid=swg21147041 (p 39)

Chapter 3

http://miller.emu.id.au/pmiller/books/rmch (p 53)
http://publib.boulder.ibm.com/infocenter/cchelp/v7r0m0/topic/com.ibm.rational.clearcase.hlp.doc/cc_main/toc_hlpovw_build_sw.htm (p 54)

Chapter 4

http://search.cpan.org/perldoc?ClearCase::Wrapper (p 90, 104)
http://search.cpan.org/perldoc?ClearCase::Wrapper::DSB (p 90)

Chapter 6

http://search.cpan.org/perldoc?ClearCase::Wrapper::MGi (p 132, 138)

Chapter 7

https://publib.boulder.ibm.com/infocenter/cchelp/v7r0m0/index.jsp?topic=/com.ibm.rational.clearcase.tutorial.doc/a_trivial_nontrivial.htm (p 151)
http://search.cpan.org/perldoc?ClearCase::Wrapper::MGi (p 162)
http://search.cpan.org/perldoc?ClearCase::SyncTree (p 168)

Chapter 8

http://publib.boulder.ibm.com/infocenter/cchelp/v7r0m0/index.jsp?topic=/com.ibm.rational.clearcase.books.cc_build_unix.doc/clearcase_build_concepts.htm#wq15 (p 171)
https://www.ibm.com/support/docview.wss?uid=swg21386111 (p 173)
http://search.cpan.org/perldoc?ClearCase::SyncTree (p 177)
http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6189256 (p 190)

Chapter 9

http://www.ibm.com/developerworks/rational/library/4311.html (p 195)
http://search.cpan.org/perldoc?ClearCase::Argv (p 200)
http://search.cpan.org/perldoc?ClearCase::Wrapper::MGi (pp 203, 206)
http://search.cpan.org/perldoc?ClearCase::Wrapper (p 204)
http://www.ibm.com/support/docview.wss?uid=swg27009697&aid=1 (p 210)
http://search.cpan.org/perldoc?synctree (p 210)

Chapter 10

http://publib.boulder.ibm.com/infocenter/cchelp/v7r0m0/index.jsp?topic=/com.ibm.rational.clearcase.books.cc_admin.doc/cc_admin.htm (p 218)
http://www-947.ibm.com/support/entry/portal/Overview/Software/Rational/Rational_ClearCase (p 218)
http://www.ibm.com/support/docview.wss?uid=swg21126456 (p 234)
http://www.ibm.com/support/docview.wss?uid=swg21285812 (p 234)
http://search.cpan.org/perldoc?ClearCase::Argv (pp 239, 248, 249)
http://search.cpan.org/perldoc?ClearCase::Wrapper (p 240)

http://www.ibm.com/developerworks/rational/library/07/0703_nellis/index.html (p 247)
http://www.ibm.com/support/docview.wss?uid=swg1PK53029 (p 252)
http://www.ibm.com/support/docview.wss?uid=swg21211784 (p 254)

Chapter 11

http://code.google.com/p/clearcase-cpan/source/browse/branches/mg/msite/sync_receive_hub.bat (p 265)
http://code.google.com/p/clearcase-cpan/source/browse/branches/mg/msite/repoch (p 268)

Chapter 12

http://audited-objects.sourceforge.net/ (p 293)

Chapter 13

http://search.cpan.org/perldoc?ClearCase::Wrapper::MGi (p 304)
http://search.cpan.org/perldoc?ClearCase::CtCmd (p 315)

Conclusion

http://search.cpan.org/perldoc?ClearCase::Argv (p 321)
http://search.cpan.org/perldoc?ClearCase::Wrapper (p 321)
http://www.ted.com/talks/dan_meyer_math_curriculum_makeover.html (p 323)

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

Z

Thank you for buying

IBM Rational ClearCase 7.0: Master the Tools That Monitor, Analyze, and Manage Software Configurations

About Packt Publishing

Packt, pronounced 'packed', published its first book "Mastering phpMyAdmin for Effective MySQL Management" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Enterprise

In 2010, Packt launched two new brands, Packt Enterprise and Packt Open Source, in order to continue its focus on specialization. This book is part of the Packt Enterprise brand, home to books published on enterprise software – software created by major vendors, including (but not limited to) IBM, Microsoft and Oracle, often for use in other corporations. Its titles will offer information relevant to a range of users of this software, including administrators, developers, architects, and end users.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

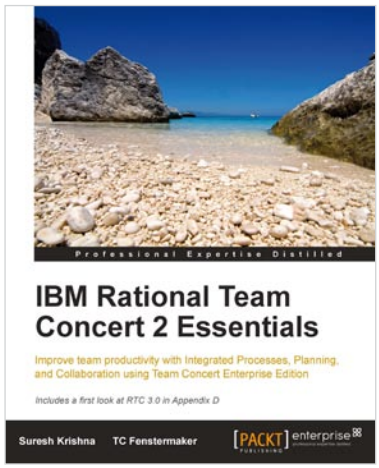
[PACKT] enterprise
professional expertise distilled
PUBLISHING

Application Development for IBM WebSphere Process Server 7 and Enterprise Service Bus 7

ISBN: 978-1-847198-28-0 Paperback: 548 pages

Build SOA-based flexible, economical, and efficient applications

1. Develop SOA applications using the WebSphere Process Server (WPS) and WebSphere Enterprise Service Bus (WESB)
2. Analyze business requirements and rationalize your thoughts to see if an SOA approach is appropriate for your project
3. Quickly build an SOA-based Order Management application by using some fundamental concepts and functions of WPS and WESB



IBM Rational Team Concert 2 Essentials

ISBN: 978-1-84968-160-5 Paperback: 308 pages

Improve team productivity with Integrated Processes, Planning, and Collaboration using Team Concert Enterprise Edition

1. Understand the core features and techniques of Rational Team Concert and Jazz platform through a real-world Book Manager Application
2. Expand your knowledge of software development challenges and find out how Rational Team Concert solves your tech, team, and collaboration worries
3. Complete overview and understanding of the Jazz Platform, Rational Team Concert, and other products built on the Jazz Platform