C++ Coding Standards

101 Rules, Guidelines, and Best Practices

Herb Sutter
Andrei Alexandrescu

C++ In-Depth Series • Bjarne Stroustrup

C++ Coding Standards

The C++ In-Depth Series

Bjarne Stroustrup, Editor

"I have made this letter longer than usual, because I lack the time to make it short."
—Blaise Pascal

The advent of the ISO/ANSI C++ standard marked the beginning of a new era for C++ programmers. The standard offers many new facilities and opportunities, but how can a real-world programmer find the time to discover the key nuggets of wisdom within this mass of information? **The C++ In-Depth Series** minimizes learning time and confusion by giving programmers concise, focused guides to specific topics.

Each book in this series presents a single topic, at a technical level appropriate to that topic. The Series' practical approach is designed to lift professionals to their next level of programming skills. Written by experts in the field, these short, in-depth monographs can be read and referenced without the distraction of unrelated material. The books are cross-referenced within the Series, and also reference *The C++ Programming Language* by Bjarne Stroustrup.

As you develop your skills in C++, it becomes increasingly important to separate essential information from hype and glitz, and to find the in-depth content you need in order to grow. The C++ In-Depth Series provides the tools, concepts, techniques, and new approaches to C++ that will give you a critical edge.

Titles in the Series

Accelerated C++: Practical Programming by Example, Andrew Koenig and Barbara E. Moo

Applied C++: Practical Techniques for Building Better Software, Philip Romanik and Amy Muntz

The Boost Graph Library: User Guide and Reference Manual, Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine

C++ Coding Standards: 101 Rules, Guidelines, and Best Practices, Herb Sutter and Andrei Alexandrescu

C++ In-Depth Box Set, Bjarne Stroustrup, Andrei Alexandrescu, Andrew Koenig, Barbara E. Moo, Stanley B. Lippman, and Herb Sutter

C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns, Douglas C. Schmidt and Stephen D. Huston

C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks, Douglas C. Schmidt and Stephen D. Huston

C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond, David Abrahams and Aleksey Gurtovoy

Essential C++, Stanley B. Lippman

Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions, Herb Sutter

Exceptional C++ Style: 40 New Engineering Puzzles, Programming Problems, and Solutions, Herb Sutter

Modern C++ Design: Generic Programming and Design Patterns Applied, Andrei Alexandrescu

More Exceptional C++: 40 New Engineering Puzzles, Programming Problems, and Solutions, Herb Sutter

For more information, check out the series web site at www.awprofessional.com/series/indepth/

C++ Coding Standards

101 Rules, Guidelines, and Best Practices

Herb Sutter

Andrei Alexandrescu

Addison-Wesley

Boston

Publisher: John Wait
Editor in Chief: Don O'Hagan
Acquisitions Editor: Peter Gordon
Editorial Assistant: Kim Boedigheimer
Marketing Manager: Chanda Leary-Coutu
Cover Designer: Chuti Prasertsith
Managing Editor: John Fuller
Project Editor: Lara Wysong
Copy Editor: Kelli Brooks
Manufacturing Buyer: Carol Melville

Library of Congress Cataloging-in-Publication Data:

Sutter, Herb.
C++ coding standards : 101 rules, guidelines, and best practices / Herb Sutter, Andrei Alexandrescu.
p. cm.
Includes bibliographical references and index.
ISBN 0-321-11358-6 (pbk. : alk. paper)
C++ (Computer program language) I. Alexandrescu, Andrei. II. Title.

QA76.73.C153S85 2004
005.13'3—dc22

2004022605

ISBN 0-321-11358-6

Text printed in the United States on recycled paper at Courier Stoughton in Stoughton, Massachusetts.

9th Printing August 2010

For the millions of current C++ programmers.

This page intentionally left blank

Contents

Preface

Get into a rut early: Do the same process the same way. Accumulate idioms. ***Standardize.*** *The only difference(!) between Shakespeare and you was the size of his idiom list—not the size of his vocabulary.*

— Alan Perlis [emphasis ours]

The best thing about standards is that there are so many to choose from.

— Variously attributed

We want to provide this book as a basis for your team's coding standards for two principal reasons:

- *A coding standard should reflect the community's best tried-and-true experience:* It should contain proven idioms based on experience and solid understanding of the language. In particular, a coding standard should be based firmly on the extensive and rich software development literature, bringing together rules, guidelines, and best practices that would otherwise be left scattered throughout many sources.
- *Nature abhors a vacuum:* If you don't consciously set out reasonable rules, usually someone else will try to push their own set of pet rules instead. A coding standard made that way usually has all of the least desirable properties of a coding standard; for example, many such standards try to enforce a minimalistic C-style use of C++.

Many bad coding standards have been set by people who don't understand the language well, don't understand software development well, or try to legislate too much. A bad coding standard quickly loses credibility and at best even its valid guidelines are liable to be ignored by disenchanted programmers who dislike or disagree with its poorer guidelines. That's "at best"—at worst, a bad standard might actually be enforced.

How to Use This Book

Think. Do follow good guidelines conscientiously; but don't follow them blindly. In this book's Items, note the Exceptions clarifying the less common situations where the guidance may not apply. No set of guidelines, however good (and we think these ones are), should try to be a substitute for thinking.

Each development team is responsible for setting its own standards, and for setting them responsibly. That includes your team. If you are a team lead, involve your team members in setting the team's standards; people are more likely to follow standards they view as their own than they are to follow a bunch of rules they feel are being thrust upon them.

This book is designed to be used as a basis for, and to be included by reference in, your team's coding standards. It is not intended to be the Last Word in coding standards, because your team will have additional guidelines appropriate to your particular group or task, and you should feel free to add those to these Items. But we hope that this book will save you some of the work of (re)developing your own, by documenting and referencing widely-accepted and authoritative practices that apply nearly universally (with Exceptions as noted), and so help increase the quality and consistency of the coding standards you use.

Have your team read these guidelines with their rationales (i.e., the whole book, and selected Items' References to other books and papers as needed), and decide if there are any that your team simply can't live with (e.g., because of some situation unique to your project). Then commit to the rest. Once adopted, the team's coding standards should not be violated except after consulting with the whole team.

Finally, periodically review your guidelines as a team to include practical experience and feedback from real use.

Coding Standards and You

Good coding standards can offer many interrelated advantages:

- *Improved code quality:* Encouraging developers to do the right things in a consistent way directly works to improve software quality and maintainability.
- *Improved development speed:* Developers don't need to always make decisions starting from first principles.
- *Better teamwork:* They help reduce needless debates on inconsequential issues and make it easier for teammates to read and maintain each other's code.
- *Uniformity in the right dimension:* This frees developers to be creative in directions that matter.

Under stress and time pressure, people do what they've been trained to do. They fall back on habit. That's why ER units in hospitals employ experienced, trained personnel; even knowledgeable beginners would panic.

As software developers, we routinely face enormous pressure to deliver tomorrow's software yesterday. Under schedule pressure, we do what we are trained to do and are used to doing. Sloppy programmers who in normal times don't know good practices of software engineering (or aren't used to applying them) will write even sloppier and buggier code when pressure is on. Conversely, programmers who form good habits and practice them regularly will keep themselves organized and deliver quality code, fast.

The coding standards introduced by this book are a collection of guidelines for writing high-quality C++ code. They are the distilled conclusions of a rich collective experience of the C++ community. Much of this body of knowledge has only been available in bits and pieces spread throughout books, or as word-of-mouth wisdom. This book's intent is to collect that knowledge into a collection of rules that is terse, justified, and easy to understand and follow.

Of course, one can write bad code even with the best coding standards. The same is true of any language, process, or methodology. A good set of coding standards fosters good habits and discipline that transcend mere rules. That foundation, once acquired, opens the door to higher levels. There's no shortcut; you have to develop vocabulary and grammar before writing poetry. We just hope to make that easier.

We address this book to C++ programmers of all levels:

If you are an apprentice programmer, we hope you will find the rules and their rationale helpful in understanding what styles and idioms C++ supports most naturally. We provide a concise rationale and discussion for each rule and guideline to encourage you to rely on understanding, not just rote memorization.

For the intermediate or advanced programmer, we have worked hard to provide a detailed list of precise references for each rule. This way, you can do further research into the rule's roots in C++'s type system, grammar, and object model.

At any rate, it is very likely that you work in a team on a complex project. Here is where coding standards really pay off—you can use them to bring the team to a common level and provide a basis for code reviews.

About This Book

We have set out the following design goals for this book:

- *Short is better than long:* Huge coding standards tend to be ignored; short ones get read and used. Long Items tend to be skimmed; short ones get read and used.

- *Each Item must be noncontroversial:* This book exists to document widely agreed-upon standards, not to invent them. If a guideline is not appropriate in all cases, it will be presented that way (e.g., "Consider X…" instead of "Do X…") and we will note commonly accepted exceptions.
- *Each Item must be authoritative:* The guidelines in this book are backed up by references to existing published works. This book is intended to also provide an index into the C++ literature.
- *Each Item must need saying:* We chose not to define new guidelines for things that you'll do anyway, that are already enforced or detected by the compiler, or that are already covered under other Items.

  Example: "Don't return a pointer/reference to an automatic variable" is a good guideline, but we chose not to include it in this book because all of the compilers we tried already emit a warning for this, and so the issue is already covered under the broader Item 1, "Compile cleanly at high warning levels."

  Example: "Use an editor (or compiler, or debugger)" is a good guideline, but of course you'll use those tools anyway without being told; instead, we spend two of our first four Items on "Use an automated build system" and "Use a version control system."

  Example: "Don't abuse **goto**" is a great Item, but in our experience programmers universally know this, and it doesn't need saying any more.

Each Item is laid out as follows:

- *Item title:* The simplest meaningful sound bite we could come up with as a mnemonic for the rule.
- *Summary:* The most essential points, briefly stated.
- *Discussion:* An extended explanation of the guideline. This often includes brief rationale, but remember that the bulk of the rationale is intentionally left in the References.
- *Examples (if applicable):* Examples that demonstrate a rule or make it memorable.
- *Exceptions (if applicable):* Any (and usually rare) cases when a rule doesn't apply. But beware the trap of being too quick to think: "Oh, I'm special; this doesn't apply in my situation"—that rationalization is common, and commonly wrong.
- *References:* See these parts of the C++ literature for the full details and analysis.

In each section, we chose to nominate a "most valuable Item." Often, it's the first Item in a section, because we tried to put important Items up front in each part; but

other times an important Item couldn't be put up front, for flow or readability reasons, and we felt the need to call it out for special attention in this way.

Acknowledgments

Many thanks to series editor Bjarne Stroustrup, to editors Peter Gordon and Debbie Lafferty, and to Tyrrell Albaugh, Kim Boedigheimer, John Fuller, Bernard Gaffney, Curt Johnson, Chanda Leary-Coutu, Charles Leddy, Heather Mullane, Chuti Prasertsith, Lara Wysong, and the rest of the Addison-Wesley team for their assistance and persistence during this project. They are a real pleasure to work with.

Inspiration for some of the "sound bites" came from many sources, including the playful style of [Cline99], the classic **import this** of [Peters99], and the legendary and eminently quotable Alan Perlis.

We especially want to thank the people whose technical feedback has helped to make many parts of this book better than they would otherwise have been. Series editor Bjarne Stroustrup's incisive comments from concept all the way through to the final draft were heavily influential and led to many improvements. We want to give special thanks to Dave Abrahams, Marshall Cline, Kevlin Henney, Howard Hinnant, Jim Hyslop, Nicolai Josuttis, Jon Kalb, Max Khesin, Stan Lippman, Scott Meyers, and Daveed Vandevoorde for their active participation in review cycles and detailed comments on several drafts of this material. Other valuable comments and feedback were contributed by Chuck Allison, Samir Bajaj, Marc Barbour, Travis Brown, Neal Coombes, Damian Dechev, Steve Dewhurst, Peter Dimov, Attila Feher, Alan Griffiths, Michi Henning, James Kanze, Mat Marcus, Petru Marginean, Robert C. "Uncle Bob" Martin, Bartosz Milewski, Balog Pal, Jeff Peil, Peter Pirkelbauer, Vladimir Prus, Dan Saks, Luke Wagner, Matthew Wilson, and Leor Zolman.

As usual, the remaining errors, omissions, and shameless puns are ours, not theirs.

Herb Sutter
Andrei Alexandrescu

Seattle, September 2004

This page intentionally left blank

Organizational and Policy Issues

If builders built buildings the way programmers wrote programs,
then the first woodpecker that came along would destroy civilization.

—Gerald Weinberg

In the grand tradition of C and C++, we count the zero-based way. The prime directive, Item 0, covers what we feel is the most basic advice about coding standards.

The rest of this introductory section goes on to target a small number of carefully selected basic issues that are mostly not directly about the code itself, but on essential tools and techniques for writing solid code.

Our vote for the most valuable Item in this section goes to Item 0: Don't sweat the small stuff. (Or: Know what not to standardize.)

0. Don't sweat the small stuff. (Or: Know what not to standardize.)

Summary

Say only what needs saying: Don't enforce personal tastes or obsolete practices.

Discussion

Issues that are really just personal taste and don't affect correctness or readability don't belong in a coding standard. Any professional programmer can easily read and write code that is formatted a little differently than they're used to.

Do use consistent formatting within each source file or even each project, because it's jarring to jump around among several styles in the same piece of code. But don't try to enforce consistent formatting across multiple projects or across a company.

Here are several common issues where the important thing is not to set a rule but just to be consistent with the style already in use within the file you're maintaining:

- *Don't specify how much to indent, but do indent to show structure:* Use any number of spaces you like to indent, but be consistent within at least each file.
- *Don't enforce a specific line length, but do keep line lengths readable:* Use any length of line you like, but don't be excessive. Studies show that up to ten-word text widths are optimal for eye tracking.
- *Don't overlegislate naming, but do use a consistent naming convention:* There are only two must-dos: a) never use "underhanded names," ones that begin with an underscore or that contain a double underscore; and b) always use **ONLY_UPPERCASE_NAMES** for macros and never think about writing a macro that is a common word or abbreviation (including common template parameters, such as **T** and **U**; writing **#define T *anything*** is extremely disruptive). Otherwise, do use consistent and meaningful names and follow a file's or module's convention. (If you can't decide on your own naming convention, try this one: Name classes, functions, and **enum**s **LikeThis**; name variables **likeThis**; name private member variables **likeThis_**; and name macros **LIKE_THIS**.)
- *Don't prescribe commenting styles (except where tools extract certain styles into documentation), but do write useful comments:* Write code instead of comments where possible (e.g., see Item 16). Don't write comments that repeat the code; they get out of sync. Do write illuminating comments that explain approach and rationale.

Finally, don't try to enforce antiquated rules (see Examples 3 and 4) even if they once appeared in older coding standards.

Examples

Example 1: Brace placement. There is no readability difference among:

```
void using_k_and_r_style() {
  // ...
}

void putting_each_brace_on_its_own_line()
{
  // ...
}

void or_putting_each_brace_on_its_own_line_indented()
  {
  // ...
  }
```

Any professional programmer can easily read *and write* any of these styles without hardship. But do be consistent: Don't just place braces randomly or in a way that obscures scope nesting, and try to follow the style already in use in each file. In this book, our brace placement choices are motivated by maximizing readability within our editorial constraints.

Example 2: Spaces vs. tabs. Some teams legitimately choose to ban tabs (e.g., [BoostLRG]), on the grounds that tabs vary from editor to editor and, when misused, turn indenting into outdenting and nondenting. Other equally respectable teams legitimately allow tabs, adopting disciplines to avoid their potential drawbacks. Just be consistent: If you do allow tabs, ensure it is never at the cost of code clarity and readability as team members maintain each other's code (see Item 6). If you don't allow tabs, allow editors to convert spaces to tabs when reading in a source file so that users can work with tabs while in the editor, but ensure they convert the tabs back to spaces when writing the file back out.

Example 3: Hungarian notation. Notations that incorporate type information in variable names have mixed utility in type-unsafe languages (notably C), are possible but have no benefits (only drawbacks) in object-oriented languages, and are impossible in generic programming. Therefore, no C++ coding standard should require Hungarian notation, though a C++ coding standard might legitimately choose to ban it.

Example 4: Single entry, single exit ("SESE"). Historically, some coding standards have required that each function have exactly one exit, meaning one **return** statement. Such a requirement is obsolete in languages that support exceptions and destructors, where functions typically have numerous implicit exits. Instead, follow standards like Item 5 that directly promote simpler and shorter functions that are inherently easier to understand and to make error-safe.

References

[BoostLRG] • [Brooks95] §12 • [Constantine95] §29 • [Keffer95] p. 1 • [Kernighan99] §1.1, §1.3, §1.6-7 • [Lakos96] §1.4.1, §2.7 • [McConnell93] §9, §19 • [Stroustrup94] §4.2-3 • [Stroustrup00] §4.9.3, §6.4, §7.8, §C.1 • [Sutter00] §6, §20 • [SuttHysl01]

1. Compile cleanly at high warning levels.

Summary

Take warnings to heart: Use your compiler's highest warning level. Require clean (warning-free) builds. Understand all warnings. Eliminate warnings by changing your code, not by reducing the warning level.

Discussion

Your compiler is your friend. If it issues a warning for a certain construct, often there's a potential problem in your code.

Successful builds should be silent (warning-free). If they aren't, you'll quickly get into the habit of skimming the output, and you *will* miss real problems. (See Item 2.)

To get rid of a warning: a) understand it; and then b) rephrase your code to eliminate the warning and make it clearer to both humans and compilers that the code does what you intended.

Do this even when the program seemed to run correctly in the first place. Do this even when you are positive that the warning is benign. Even benign warnings can obscure later warnings pointing to real dangers.

Examples

Example 1: A third-party header file. A library header file that you cannot change could contain a construct that causes (probably benign) warnings. Then wrap the file with your own version that **#include**s the original header and selectively turns off the noisy warnings for that scope only, and then **#include** your wrapper throughout the rest of your project. Example (note that the warning control syntax will vary from compiler to compiler):

```
// File: myproj/my_lambda.h -- wraps Boost's lambda.hpp
//   Always include this file; don't use lambda.hpp directly.
//   NOTE: Our build now automatically checks "grep lambda.hpp <srcfile>".
// Boost.Lambda produces noisy compiler warnings that we know are innocuous.
// When they fix it we'll remove the pragmas below, but this header will still exist.
//
#pragma warning(push)       // disable for this header only
  #pragma warning(disable:4512)
  #pragma warning(disable:4180)
  #include <boost/lambda/lambda.hpp>
#pragma warning(pop)        // restore original warning level
```

Example 2: "Unused function parameter." Check to make sure you really didn't mean to use the function parameter (e.g., it might be a placeholder for future expansion, or a required part of a standardized signature that your code has no use for). If it's not needed, simply delete the name of a function parameter:

```
// ... inside a user-defined allocator that has no use for the hint ...

// warning: "unused parameter 'localityHint'"
pointer allocate( size_type numObjects, const void *localityHint = 0 ) {
  return static_cast<pointer>( mallocShared( numObjects * sizeof(T) ) );
}

// new version: eliminates warning
pointer allocate( size_type numObjects, const void * /* localityHint */ = 0 ) {
  return static_cast<pointer>( mallocShared( numObjects * sizeof(T) )   );
}
```

Example 3: "Variable defined but never used." Check to make sure you really didn't mean to reference the variable. (An RAII stack-based object often causes this warning spuriously; see Item 13.) If it's not needed, often you can silence the compiler by inserting an evaluation of the variable itself as an expression (this evaluation won't impact run-time speed):

```
// warning: "variable 'lock' is defined but never used"
void Fun() {
  Lock lock;

  // ...

}

// new version: probably eliminates warning
void Fun() {
  Lock lock;
  lock;

  // ...

}
```

Example 4: "Variable may be used without being initialized." Initialize the variable (see Item 19).

*Example 5: "Missing **return**."* Sometimes the compiler asks for a **return** statement even though your control flow can never reach the end of the function (e.g., infinite loop, **throw** statements, other **return**s). This can be a good thing, because sometimes you only *think* that control can't run off the end. For example, **switch** statements that

do not have a **default** are not resilient to change and should have a **default** case that does **assert(false)** (see also Items 68 and 90):

```
// warning: missing "return"
int Fun( Color c ) {
  switch( c ) {
  case Red:    return 2;
  case Green:  return 0;
  case Blue:
  case Black:  return 1;
  }
}

// new version: eliminates warning
int Fun( Color c ) {
  switch( c ) {
  case Red:    return 2;
  case Green:  return 0;
  case Blue:
  case Black:  return 1;
  default:     assert( !"should never get here!" );   // !"string" evaluates to false
               return -1;
  }
}
```

Example 6: "Signed/unsigned mismatch." It is usually not necessary to compare or assign integers with different signedness. Change the types of the variables being compared so that the types agree. In the worst case, insert an explicit cast. (The compiler inserts that cast for you anyway, and warns you about doing it, so you're better off putting it out in the open.)

Exceptions

Sometimes, a compiler may emit a tedious or even spurious warning (i.e., one that is mere noise) but offer no way to turn it off, and it might be infeasible or unproductive busywork to rephrase the code to silence the warning. In these rare cases, as a team decision, avoid tediously working around a warning that is merely tedious: Disable that specific warning only, disable it as locally as possible, and write a clear comment documenting why it was necessary.

References

[Meyers97] §48 • [Stroustrup94] §2.6.2

2. Use an automated build system.

Summary

Push the (singular) button: Use a fully automatic ("one-action") build system that builds the whole project without user intervention.

Discussion

A one-action build process is essential. It must produce a dependable and repeatable translation of your source files into a deliverable package. There is a broad range of automated build tools available, and no excuse not to use one. Pick one. Use it.

We've seen organizations that neglect the "one-action" requirement. Some consider that a few mouse clicks here and there, running some utilities to register COM/CORBA servers, and copying some files by hand constitute a reasonable build process. But you don't have time and energy to waste on something a machine can do faster and better. You need a one-action build that is automated and dependable.

Successful builds should be silent, warning-free (see Item 1). The ideal build produces no noise and only one log message: "Build succeeded."

Have two build modes: Incremental and full. An incremental build rebuilds only what has changed since the last incremental or full build. Corollary: The second of two successive incremental builds should not write any output files; if it does, you probably have a dependency cycle (see Item 22), or your build system performs unnecessary operations (e.g., writes spurious temporary files just to discard them).

A project can have different forms of full build. Consider parameterizing your build by a number of essential features; likely candidates are target architecture, debug vs. release, and breadth (essential files vs. all files vs. full installer). One build setting can create the product's essential executables and libraries, another might also create ancillary files, and a full-fledged build might create an installer that comprises all your files, third-party redistributables, and installation code.

As projects grow over time, so does the cost of not having an automated build. If you don't use one from the start, you will waste time and resources. Worse still, by the time the need for an automated build becomes overwhelming, you will be under more pressure than at the start of the project.

Large projects might have a "build master" whose job is to care for the build system.

References

[Brooks95] §13, §19 • [Dewhurst03] §1 • [GnuMake] • [Stroustrup00] §9.1

3. Use a version control system.

Summary

The palest of ink is better than the best memory (Chinese proverb): Use a version control system (VCS). Never keep files checked out for long periods. Check in frequently after your updated unit tests pass. Ensure that checked-in code does not break the build.

Discussion

Nearly all nontrivial projects need more than one developer and/or take more than a week of work. On such projects, you *will* need to compare historical versions of the same file to determine when (and/or by whom) changes were introduced. You *will* need to control and manage source changes.

When there are multiple developers, those developers will make changes in parallel, possibly to different parts of the same file at the same time. You need tools to automate checkout/versioning of file and, in some cases, merging of concurrent edits. A VCS automates and controls checkouts, versioning, and merging. A VCS will do it faster and more correctly than you could do it by hand. And you don't have time to fiddle with administrivia—you have software to write.

Even a single developer has "oops!" and "huh?" moments, and needs to figure out when and why a bug or change was introduced. So will you. A VCS automatically tracks the history of each file and lets you "turn the clock back." The question isn't whether you will want to consult the history, but when.

Don't break the build. The code in the VCS must always build successfully.

The broad range of VCS offerings leaves no excuse not to use one. The least expensive and most popular is **cvs** (see References). It is a flexible tool, featuring TCP/IP access, optional enhanced security (by using the secure shell **ssh** protocol as a backend), excellent administration through scripting, and even a graphical interface. Many other VCS products either treat **cvs** as a standard to emulate, or build new functionality on top of it.

Exceptions

A project with one programmer that takes about a week from start to finish probably can live without a VCS.

References

[BetterSCM] • *[Brooks95] §11, §13* • *[CVS]*