SOFTWARE PROCESS: THE KEY TO DEVELOPING ROBUST, REUSABLE AND MAINTAINABLE OPEN-SOURCE SOFTWARE

William J. Schroeder, Luis Ibáñez, Kenneth M. Martin

Kitware, Inc.
http://www.kitware.com

ABSTRACT

The practice of image processing inherently requires software development. Creating this technology requires designing, implementing, debugging and testing software applications on a continual basis. Furthermore current software development is typically performed in a distributed environment involving many developers. While the use of open-source software may create collaborative communities that enhance overall technology exchange, it does nothing directly to manage change nor does it address the quality of the underlying software. This paper describes a software development process that has proven vital to the success of the widely used open-source toolkits ITK (itk.org) and VTK (vtk.org). This process facilitates cross-platform development, incorporates automatic documentation generation, integrates continuous testing, and posts the results of the process on publicly accessible web pages. The net result is that a responsive feedback loop is created between the developers in the community and automated processes to measure software quality. With this process software converges towards better software as long as the process is enforced. The tools described here are open-source and available for use in academic and commercial applications.

1. INTRODUCTION

The days when a single developer could author a software system are gone. Today software is larger and more complex than ever before. This complexity comes at a price: software requires teams to develop and demands long-term maintenance if it is to thrive and grow. In addition the complexity of software means that testing is vital to insuring the quality of the code. Documentation needs are greater since users require assistance to wade through the system and find the parts of the software that really matter to them. Managing change to the software base is critical to capture bug fixes and for revision control. Finally, the abundance of different computer configurations (hardware, operating systems, enabling software and compilers) means that development must address cross-platform issues. Thus new processes are necessary to replace the approaches of the past if we are to create the software technology of the future.

We have developed a software process that addresses these many requirements for several large open-source projects including the *Visualization Toolkit* (VTK) and the *Insight Segmentation and Registration Toolkit* (ITK). The size of these communities are large: VTK has over 2,000 subscribers on its users list and 92 on the developer's list, and the ITK project was developed by a group of eleven academic and commercial organizations with a total of 114 subscribers on the developer's list and over 500 on the user's list. Such distributed, collaborative development environments are typical of modern development efforts.

This paper is organized as follows. Section 2 characterizes the requirements for the software development process. Section 3 describes the tools supporting the software process. Section 4 discusses the process and offers suggestions for future work.

2. PROCESS OVERVIEW

The methodology described in this paper has developed over several years in support of large-scale open-source software projects. While it can be adopted for proprietary software, portions of the process require that all developers have access to the source code. This process does introduce overhead and may not be suitable for small projects, but even in this case the benefits are typically worth the extra effort.

The software process we use is based on principles of *agile programming* or *extreme programming*. The idea is that the standard development tasks:

- requirements generation,
- software design,
- managing source code versions and updates,
- configuring projects for specific platforms,
- compilation and linking,
- testing the code at run time,
- verifying the validity of output,
- documenting the code, and
- tracking and repairing bugs

are performed continuously rather than in the waterfall fashion that conventional development efforts typically use. Generally the process begins with a small kernel implementation that is incrementally evolved by simultaneous application of the steps listed previously. The key to the process is an automated testing facility that posts results to a central web page where all developers can monitor the efforts of the community. In our practice all developers have the right and encouragement to repair errors in other developer's code. In fact, we take pride in the fact that the code appears as if it were written by a single person.

3. PROCESS TOOLS

Figure 1 illustrates the interactions between the tools supporting the software process described here. First, multiple developers contribute code in a CVS repository. CVS keeps track of what, who, when and why source code was changed. The build process is controlled by the CMake cross-platform build tool. CMake is unique in that it does not replace native build files such as make and Windows workspaces; rather it generates these from platform-independent CMakeLists.txt files and then uses the native build tools to manage the compile/link process. DART coordinates the testing process. This extensive testing tool tracks compile and link errors, checks style, runs memory checking tools such as Valgrind (http://valgrind.kde.org) or Purify, and executes hundreds of tests that developers are expected to contribute as they check in source code. DART clients post testing results using an XML protocol which are then posted by the DART server on the project web pages. These web pages are referred to as the project *dashboard* since it summarizes the state of the project. The details of these tools and some additional documentation tools are described in the next subsections.

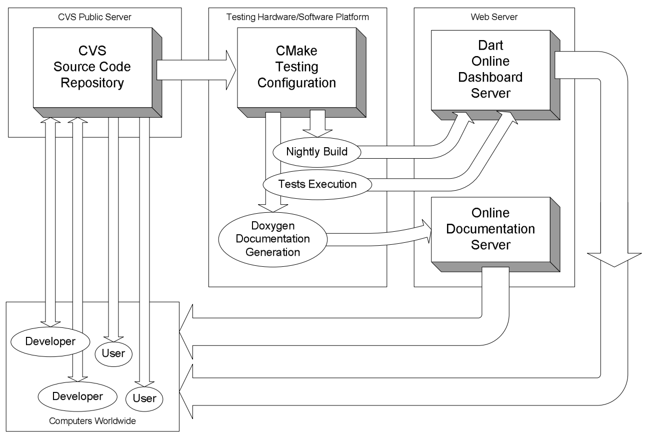

CVS Public Server
CVS Source Code Repository
Testing Hardware/Software Platform
CMake Testing Configuration
Web Server
Dart Online Dashboard Server
Nightly Build
Tests Execution
Doxygen Documentation Generation
Online Documentation Server
Developer
User
Developer
User
Computers Worldwide

Figure 1. Tools supporting the software process.

3.1. CVS

The Concurrent Versions System (CVS) is a tool designed for maintaining a central repository of source code in which multiple developers retrieve, modify and commit files and changes to files. CVS supports simultaneous edits of files and merges changes into the code base. Merge conflicts occur rarely and when they do CVS marks the conflicts that must be resolved by the developer community. CVS supports tagging and branching the repository so it is relatively easy to manage releases.

3.2. CMake

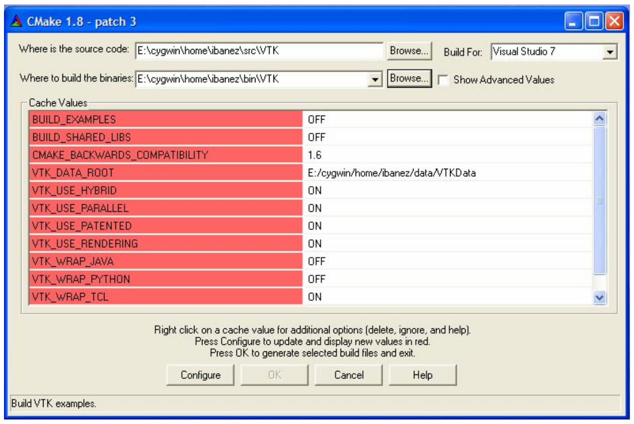
CMake is a cross-platform build system. The amount of effort to manage the build process is often underestimated by software developers. The task is particularly difficult when the software must function across different computing configurations. Configuring a project involves tasks such as:

- Finding the appropriate compiler to build the software. This includes testing whether a compiler supports particular language features.
- Selecting compiler flags in a way that is consistent across all systems.
- Specifying the directories where headers and libraries from other required software packages are located.
- Generating code; for example executing a wrapper generator tool such as SWIG (swig.org).
- Specifying the location(s) to produce object code, libraries, executables and install packages.

CMake simplifies this process by using platform-independent configuration files to generate the appropriate workspace(s) or makefile(s) for the target compiler. Developers can then use the native compilation tools with which they are familiar. Currently CMake supports most C++ compilers found today including Microsoft Visual Studio 6.0, .NET, .NET 2003, Borland, Linux, Unix (e.g., HP, Sun, SGI) and Mac OSX. The developers write simple ASCII, CMakeLists.txt files that are maintained in the same CVS repository as the source code. CMake can do everything that autoconf can do and more since it runs cross-platform without operating system emulation tools. It also integrates with the DART testing tool. That is, it runs as a DART client and can run and submit testing results to the DART dashboard.

When CMake runs it invokes a GUI consistent with the platform on which it is executing. Figure 2 shows the CMake GUI on Microsoft Windows. An equivalent interface is available for Unix based on the curses library. Other interfaces are possible and have been implemented with cross-platform GUI builders such as wxWindows (http://www.wxwindows.org). The CMake configuration process is iterative. Developers select configuration options and then execute the CMake configure process.

Each configuration step may expose new build options that the developer specifies, followed by repetition of the configuration step. Eventually the process converges and the developer selects the generate option to produce the native build files. CMake is an open-source project originally created as part of the ITK project. Further information about CMake is available at http://www.cmake.org.



CMake 1.8 - patch 3

Where is the source code: E:\cygwin\home\ibanez\src\VTK

Build For: Visual Studio 7

Where to build the binaries: E:\cygwin\home\ibanez\bin\VTK

Show Advanced Values

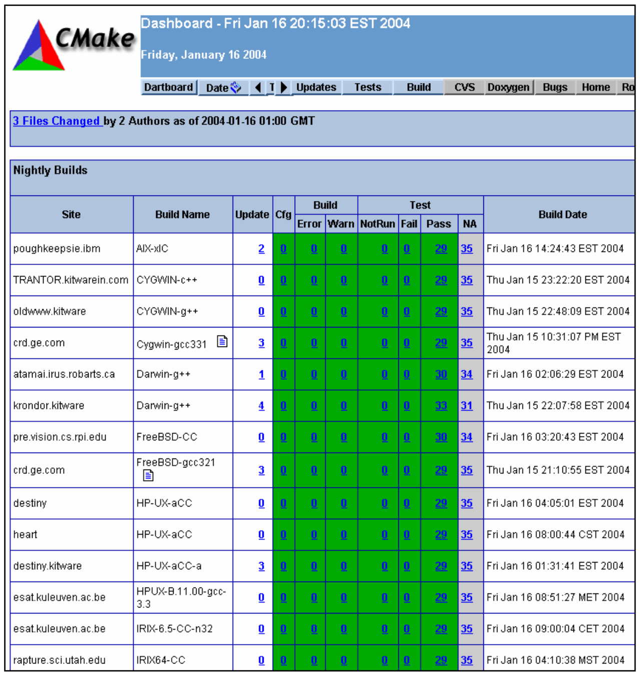
Cache Values

BUILD_EXAMPLES	OFF
BUILD_SHARED_LIBS	OFF
CMAKE_BACKWARDS_COMPATIBILITY	1.6
VTK_DATA_ROOT	E:/cygwin/home/ibanez/data/VTKData
VTK_USE_HYBRID	ON
VTK_USE_PARALLEL	ON
VTK_USE_PATENTED	ON
VTK_USE_RENDERING	ON
VTK_WRAP_JAVA	OFF
VTK_WRAP_PYTHON	OFF
VTK_WRAP_TCL	ON

Right click on a cache value for additional options (delete, ignore, and help).
Press Configure to update and display new values in red.
Press OK to generate selected build files and exit.

Build VTK examples.

Figure 2 - The CMake GUI on Windows.



Dashboard - Fri Jan 16 20:15:03 EST 2004

Friday, January 16 2004

3 Files Changed by 2 Authors as of 2004-01-16 01:00 GMT

Nightly Builds

Site	Build Name	Update	Cfg	Build		Test				Build Date
				Error	Warn	NotRun	Fail	Pass	NA	
poughkeepsie.ibm	AIX-xlC	2	0	0	0	0	0	29	35	Fri Jan 16 14:24:43 EST 2004
TRANTOR.kitwarein.com	CYGWIN-c++	0	0	0	0	0	0	29	35	Thu Jan 15 23:22:20 EST 2004
oldwww.kitware	CYGWIN-g++	0	0	0	0	0	0	29	35	Thu Jan 15 22:48:09 EST 2004
crd.ge.com	Cygwin-gcc331	3	0	0	0	0	0	29	35	Thu Jan 15 10:31:07 PM EST 2004
atamai.irus.robarts.ca	Darwin-g++	1	0	0	0	0	0	30	34	Fri Jan 16 02:06:29 EST 2004
krondor.kitware	Darwin-g++	4	0	0	0	0	0	33	31	Thu Jan 15 22:07:58 EST 2004
pre.vision.cs.rpi.edu	FreeBSD-CC	0	0	0	0	0	0	30	34	Fri Jan 16 03:20:43 EST 2004
crd.ge.com	FreeBSD-gcc321	3	0	0	0	0	0	29	35	Thu Jan 15 21:10:55 EST 2004
destiny	HP-UX-aCC	0	0	0	0	0	0	29	35	Fri Jan 16 04:05:01 EST 2004
heart	HP-UX-aCC	0	0	0	0	0	0	29	35	Fri Jan 16 08:00:44 CST 2004
destiny.kitware	HP-UX-aCC-a	3	0	0	0	0	0	29	35	Fri Jan 16 01:31:41 EST 2004
esat.kuleuven.ac.be	HPUX-B.11.00-gcc-3.3	0	0	0	0	0	0	29	35	Fri Jan 16 08:51:27 MET 2004
esat.kuleuven.ac.be	IRIX-6.5-CC-n32	0	0	0	0	0	0	29	35	Fri Jan 16 09:00:04 CET 2004
rapture.sci.utah.edu	IRIX64-CC	0	0	0	0	0	0	29	35	Fri Jan 16 04:10:38 MST 2004

Figure 3 - The DARTdashboard on a good day.

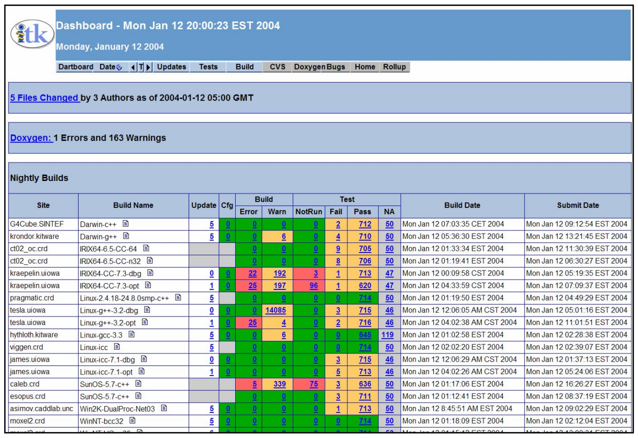
3.3. DART

DART is the focal point for the developer community. It is built around a client-server architecture. Clients, which may be distributed anywhere on the internet, are responsible for testing software and posting the results of the testing in an XML form to the DART server. In turn the server uses XSLT to control how the testing information is displayed. DART is capable of extensive nightly builds where the entire test suite is run, or continuous builds where very quick tests (in response to developer check-ins to CVS) are performed and posted immediately to the dashboard. (Developers may also post experimental build results when they wish to try something out without committing code to CVS.)

The DART testing process is extensive. It tracks build and link errors; checks code style; measures code coverage; reports changes to the code base since the last reporting period; runs memory tests; and executes the many tests associated with the project. All of this information is gathered and available on the dashboard. Figure 3 is an example dashboard on a “good” day (the dashboard is green). Each row in the figure represents a different DART client (i.e., operating system and compiler configuration). The columns report errors and warnings that may be selected to drill down into the associated information. For example, developers can select build warnings and see the actual warning produced from the compiler. In this way it is possible for a developer without access to a particular platform see the results from another developer who does have access to the platform, and correct any errors as necessary. Figure 4 shows a DART dashboard on a “bad” day. Notice that errors and warnings are highlighted with the appropriate colors. Thus it becomes readily apparent when a developer affects the dashboard in a negative way. Community peer pressure usually forces a rapid resolution to the problem. However repeated excesses can result in denial of CVS access, which is the ultimate punishment for a developer.

Developers are expected to create tests that exercise the source code. The tests are used to generate coverage results as well as to compare against valid output. Typically when a test is first created a valid image or other form of output is created and checked into a CVS testing repository. When DART runs the tests at a later date, it compares the test output to the valid output. A comparison is performed and if change occurs an error is flagged. On some systems thresholds are used to take into account differences due to graphics cards or other expected variations in output.

One of main benefits of DART is that it identifies errors as they occur. In the past we often waited long periods of time before performing tests prior to a release. While this would certainly uncover errors, it was extremely difficult to trace the origin of the problem because the causal relationship to the code change was lost. With DART, changes are immediately tested and if problems occur it is generally easy to trace to the source of the problem. In fact with continuous builds problems are identified almost immediately and usually resolved prior to the nightly testing cycle. Find more information about DART at http://public.kitware.com/dart.


Dashboard - Mon Jan 12 20:00:23 EST 2004

Monday, January 12 2004

5 Files Changed by 3 Authors as of 2004-01-12 05:00 GMT

Doxygen: 1 Errors and 163 Warnings

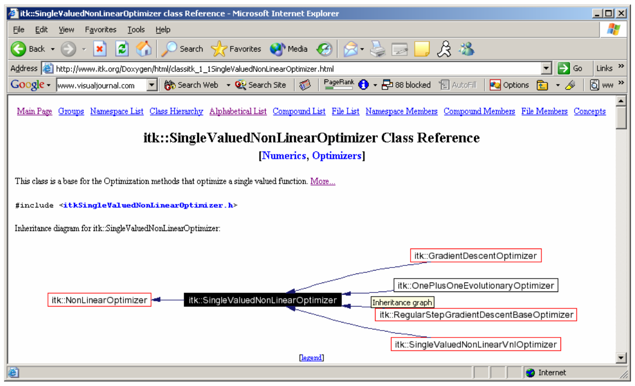
Nightly Builds

Site	Build Name	Update	Cfg	Build		Test				Build Date	Submit Date
				Error	Warn	NotRun	Fail	Pass	NA		
G4Cube.SINTEF	Darwin-c++	5	0	0	0	0	2	712	50	Mon Jan 12 07:03:35 CET 2004	Mon Jan 12 09:12:54 EST 2004
krondor.kitware	Darwin-g++	5	0	0	6	0	4	710	50	Mon Jan 12 05:36:30 EST 2004	Mon Jan 12 13:21:45 EST 2004
ct02_oc.crd	IRIX64-6.5-CC-64			0	0	0	9	705	50	Mon Jan 12 01:33:34 EST 2004	Mon Jan 12 11:30:39 EST 2004
ct02_oc.crd	IRIX64-6.5-CC-n32			0	0	0	8	706	50	Mon Jan 12 01:19:41 EST 2004	Mon Jan 12 06:30:27 EST 2004
kraepelin.uiowa	IRIX64-CC-7.3-dbg	0	0	22	192	3	1	713	47	Mon Jan 12 00:09:58 CST 2004	Mon Jan 12 05:19:35 EST 2004
kraepelin.uiowa	IRIX64-CC-7.3-opt	1	0	25	197	96	1	620	47	Mon Jan 12 04:33:59 CST 2004	Mon Jan 12 07:09:37 EST 2004
pragmatic.crd	Linux-2.4.18-24.8.0smp-c++	5		0	0	0	0	714	50	Mon Jan 12 01:19:50 EST 2004	Mon Jan 12 04:49:29 EST 2004
tesla.uiowa	Linux-g++-3.2-dbg	0	0	0	14085	0	3	715	46	Mon Jan 12 12:06:05 AM CST 2004	Mon Jan 12 05:01:16 EST 2004
tesla.uiowa	Linux-g++-3.2-opt	1	0	25	4	0	2	716	46	Mon Jan 12 04:02:38 AM CST 2004	Mon Jan 12 11:01:51 EST 2004
hythloth.kitware	Linux-gcc-3.3	5	0	0	6	0	0	645	119	Mon Jan 12 01:02:58 EST 2004	Mon Jan 12 02:28:38 EST 2004
viggen.crd	Linux-icc	5		0	0	0	0	714	50	Mon Jan 12 02:02:20 EST 2004	Mon Jan 12 02:39:07 EST 2004
james.uiowa	Linux-icc-7.1-dbg	0	0	0	0	0	3	715	46	Mon Jan 12 12:06:29 AM CST 2004	Mon Jan 12 01:37:13 EST 2004
james.uiowa	Linux-icc-7.1-opt	1	0	0	0	0	5	713	46	Mon Jan 12 04:02:26 AM CST 2004	Mon Jan 12 05:24:06 EST 2004
caleb.crd	SunOS-5.7-c++			5	339	75	3	636	50	Mon Jan 12 01:17:06 EST 2004	Mon Jan 12 16:26:27 EST 2004
esopus.crd	SunOS-5.7-c++			0	0	0	3	711	50	Mon Jan 12 01:12:41 EST 2004	Mon Jan 12 08:37:19 EST 2004
asimov.caddlab.unc	Win2K-DualProc-Net03	5	0	0	0	0	1	713	50	Mon Jan 12 8:45:51 AM EST 2004	Mon Jan 12 09:02:29 EST 2004
moxel2.crd	WinNT-bcc32	5	0	0	0	0	0	714	50	Mon Jan 12 01:18:09 EST 2004	Mon Jan 12 02:12:04 EST 2004

Figure 4 - The DART dashboard on a bad day.

3.4. Documentation Tools

Long experience as developers and users of software has proven the value of documentation. Currently we use a process that requires the developer to incorporate documentation directly into the source code. While we prefer Doxygen (http://www.stack.nl/~dimitri/doxygen) as our documentation generation system, other tools such as Doc++ and JavaDoc work equally well. Doxygen produces a code index, method descriptions, inheritance and collaboration diagrams (Figure 5). It can be configured to produce other information and can generate web pages, LaTeX documentation and a variety of other forms.


itk::SingleValuedNonLinearOptimizer class Reference - Microsoft Internet Explorer

Address http://www.itk.org/Doxygen/html/classitk_1_1SingleValuedNonLinearOptimizer.html

itk::SingleValuedNonLinearOptimizer Class Reference

[Numerics, Optimizers]

This class is a base for the Optimization methods that optimize a single valued function. More...

```
#include <itkSingleValuedNonLinearOptimizer.h>
```

Inheritance diagram for itk::SingleValuedNonLinearOptimizer:

itk::GradientDescentOptimizer
itk::OnePlusOneEvolutionaryOptimizer
itk::NonLinearOptimizer
itk::SingleValuedNonLinearOptimizer
Inheritance graph
itk::RegularStepGradientDescentBaseOptimizer
itk::SingleValuedNonLinearVnlOptimizer
[legend]
Internet

Figure 5. Doxygen generated UML diagrams.

Recently in the ITK project we used CMake, Perl, and LaTeX to automatically extract source code from examples and incorporate the formatted code directly into a book. The benefit of this approach is that we were sure that the code in the documentation was correct as long as the dashboard for the day was green.

3.5. Other Tools

The process utilizes many other tools as well. We use phpBugTracker (http://phpbt.sourceforge.net/) to keep track of bugs and feature requests. MailMan (http://www.list.org/) is used to manage a user's and developer's mailing list. CableSWIG (http://www.itk.org/-HTML/CableSwig.html) is an extension to the popular SWIG interpreted language wrapper generator. It uses GCC_XML (http://www.gccxml.org) to produce XML from arbitrarily complex C++ code, and then interfaces this XML into the SWIG internal parse structures in order to generate code.

3.5. Summary and Future Work

We have created a process for developing large-scale distributed, open-source software projects. We have found that the use of CMake for cross-platform development, DART for testing, and CVS for source code managements works well in real world applications. In particular, the use of the DART dashboard creates a feedback loop that results in high quality code.

While the process works well it does require discipline. In particular, developers must pay attention to the dashboard, and one or more enforcers must make sure that errors are corrected immediately. In our open-source communities we expect errors to be corrected in a day or less. In addition, developers must create tests as they check in source code. These white-box tests are designed to exercise the features of the software and insure that code is covered adequately (we aim for 80% coverage or higher). Generally community peer pressure is enough to enforce the process; but removing code from CVS and revoking CVS access are occasionally used.

In the future we will solidify the XML schema for the testing process. We are also working on simplifying the installation of the DART server. Another desirable feature is to keep testing results in a database. Queries can be made and statistical studies can be used to judge the quality of a particular piece of code. Additional hardware also benefits the process as well; currently the continuous testing matrix is inadequate to catch errors as early as they could be caught. We encourage the community to offer their computers as testing clients; the more coverage the better. Please join the VTK, ITK, or CMake communities if you would like to help.

REFERENCES

[1] L. Ibanez, W. Schroeder. *The ITK Software Guide*, Kitware, Inc., Clifton Park NY, ISBN 1-930934-10-6. 2003.

[2] W. Schroeder, K. Martin, W. Lorensen. *The Visualization Toolkit An Object-Oriented Approach to 3D Graphics 3rd Ed.*. Kitware, Inc. ISBN 1-930934-07-6, 2002.

[3] Kitware, Inc, *The VTK User's Guide*, Kitware Inc, Clifton Park NY, ISBN 1-930934-08-4, 2003.

[4] K. Martin, B. Hoffman, *Mastering CMake*, Kitware Inc, Clifton Park NY, ISBN 1-930934-09-2, 2003.

[5] K. F. Fogel, *Open Source Development with CVS*, Second Edition, The Coriolis Group, http://cvsbook.red-bean.com/, 1999.