CHAPTER

FIVE

Software Development Process

"If you can't describe what you are doing as a process, you don't know what you're doing." W Edwards Deming.

IGSTK is intended for use in the operating room. This type of mission-crttical software requires a robust software development process that ensures the quality of the software produced. A *r*obust software development process for IGSTK means well-defined, well-understood, and well-executed. It does not necessarily mean a heavyweight process definition that imposes large documentation products or rigid constraints that restrict the manner in which developers may innovate. However, the safety requirements of the mission critical domain do mandate that some controls be put in place. IGSTK does this by defining and adhering to a set of best practices for software development. This chapter presents these best practices and then describes the specific tools and techniques that are used to realize these practices.

5.1 IGSTK Best Practices

1. Recognize that people are the most important mechanism available for ensuring high-quality software. The IGSTK team consists of developers with considerable expertise in the application domain, supporting software, and tools. Their collective judgment outweighs process mandates.

2. Facilitate constant communication. To prevent distributed team members who are working on decoupled components from becoming too isolated, IGSTK members participate in a weekly teleconference and meet in person twice per year. IGSTK also employs a mailing list, instant messaging, and a wiki for online collaboration.

3. Produce iterative releases. IGSTKs development cycle includes twice-yearly external releases. We considered six months too long a horizon to manage development, so internal releases are broken down into approximately two-month iterations. At the end of an iteration, team members perform quality reviews and move code considered stable to the main repository.

4. Manage source code from a quality perspective. IGSTK defines different configuration management policies to satisfy different quality criteria. Codelines with separate policies, for example a main repository and a sandbox, lets developers collaborate on code that might not yet meet stringent requirements. Exploiting configuration management approaches early in a project helps document and track quality progress.

5. Focus on 100 percent code and path coverage at the component level. Unit tests ensure complete code coverage across all platforms. We are also developing customized visualization and validation tool machines to guarantee that correctness properties within all IGSTK state machines are verified with every nightly build. In addition, dynamic analysis tools prevent memory leaks and access violations.

6. Emphasize continuous builds and testing. IGSTK uses the open source DART tool to produce a nightly dashboard of build and unit test results across all supported platforms.

7. Support the development process with robust tools. In addition to Dart, IGSTK employs the CMake open source cross-platform build solution, KWStyle for source code style checking, and Doxygen, an open source documentation system. Best practices for coding and documentation posted on the wiki augment these tools.

8. Manage requirements iteratively in lockstep with code management. As requirements evolve and the code matures, adopting flexible yet defined processes for managing requirements becomes necessary.

9. Focus on meeting exactly the current set of requirements. Traceability is needed in safety-critical domains, particularly in surgical applications that need government approval, and implies heavy process structureslarge documents and invasive tools. IGSTK addresses this problem with continuous requirements review, lightweight tools, and codeline policies.

10. Evolve the development process. Through constant communication, IGSTK members recognize when to approach the complexities they face within the current process framework, when tweaks are required, or when to adopt entirely new practices.

IGSTK is *agile*. The approach followed by the team and recommended for component and application developers is to employ *lightweight* methods. Traditional approaches that introduce rigid process steps with volumes of documentation may actually lead to a decrease in quality and safety for projects such as IGSTK due to the distributed collaborative nature of open source software development. It is not beneficial to create strict process controls that will not can not, nor will not, be adhered to in such an environment. The resulting process would lead to inconsistent documentation and poor execution of the defined process, leading to false hope that by having such a strict process definition quality will be achieved.

IGSTK uses the best practices presented above as a means to achieve quality and safety by executing these practices throughout the software development process. The emphasis is on agile execution; if execution of a defined process is good, IGSTK executes these practices constantly. Tools, reviews, communication, builds, testing, release management are all performed

in a highly iterative continuous manner to ensure the quality and safety of the software produced.

5.2 Developer Practices

5.2.1 Code Conventions

Understandability of source code is a critical aspect in the maintenance of a software system. Its importance is magnified in open source projects such as IGSTK that rely on a large distributed development community to evolve the framework and construct applications. Defining enforceable coding standards and conventions is a powerful technique for ensuring the maintainability of an open source codebase. IGSTK employs a combination of tools, practices, and conventions to ensure understandability and maintainability of the source code.

IGSTK source code conventions are included in Appendix XXX. These include stylistic conventions, file organization, and best practices for developers (use of exceptions, macros, STL, etc.). All IGSTK component and application developers are strongly encouraged to review these conventions and adhere to them to the greatest extent possible. In this section we highlight some of the best practices and then discuss the use of the Doxygen and kwStyle tools.

The code conventions in Appendix XXX includes recommended best practices within the source code that all IGSTK component and application developers should follow. IGSTK discourages the use of generics (C++ templates), and encourages the use of the Standard Template Library (STL) but not at the API level. IGSTK relies on strong type checking to ensure API contracts, and the unnecessary use of templates may lead to type-related runtime errors.

IGSTK advocates the use ”smart pointers” to manage object references for objects with a significant memory footprint. Memory management can be an area of particularly hard to detect and correct defects. The use of smart pointers, while adding some overhead to application execution, reduces the opportunity for memory-related defects. IGSTK also advocates const correctness. As stated in the Appendix, ”A safe approach is to start considering everything as const and making classes and methods non-const only when a justification exists. Const verification is done by the compiler and prevents inappropriate and unsafe use of the classes and methods.” IGSTK has created macros that assist with using smart pointers and for enforcing setter/getter contracts.

The Doxygen documentation tool generates external documentation from commented source code. Doxygen automatically extracts comments delimited in a special way to generate the external documentation. The IGSTK style guide, Appendix XXX, section 10 defines Doxygen documentation conventions for classes and methods. These comments are only extracted from C++ header (.h) files. IGSTK developers are required to keep comments up to date with source code changes.

Coding stylistic conventions can often be the hardest standards for developers to consistently adhere to over a lengthy period of time. As code grows and evolves, enforcement of stylistic conventions via developer diligence and code reviews is tedious to maintain. The kwStyle tool

(http://public.kitware.com/KWStyle) performs static code checks on over 20 stylistic properties of C++ source code. IGSTK has codified stylistic rules in kwStyle and integrates kwStyle analysis into the nightly Dart dashboard. The kwStyle tool removes the time-consuming tedious process of checking for stylistic conformance and ensures code style does not degrade over time. kwStyle is open source freely available for download. You may also demo the tool via the web using the *C*heck my File option off the kwStyle homepage.

Source code is best understood and maintained when it looks as if it was written by a single developer. This is especially true when considering open source. Readers of the source code can more easily understand the intent of the code when they can apply a single mental model when internally ”parsing” it. Misinterpretation of source code intent may decrease code quality and application safety. Developers could add new features in the wrong place, or patch existing code in an unsafe manner, or invoke services in an improper manner. Tools such as Doxygen, kwStyle, and Dart can help developers adhere to conventions, but in the end developers must accept responsibility for creating readable, understandable, and maintainable source code. IGSTK core component source code reviews (see 5.2.2 below) always include detailed checks that these conventions are followed.

5.2.2 Code Reviews

Source code reviews are well-known as one of the best, if not the best, method to ensure quality software. An IGSTK code review is an informal review facilitated by the managed communication methods described below. IGSTK code reviews are integrated into the configuration management policies governing the source code repository and the iterative development cycle of IGSTK. Developers constructing applications using IGSTK are strongly encouraged to employ a code review process, leveraging other team members or the IGSTK community.

IGSTK code reviews do not have the formality of a software inspection (XXX see [Fagan, DeMarco, Weigers]), but are not so informal as to lack a record of defects found and fixed. IGSTK code reviews are performed by at least two reviewers, at least one of which should have deep knowledge of the functional domain of the introduced code. Reviewers use the IGSTK coding standards document (see XXX), the requirements repository, the Wiki, and tools such as kwStyle and DART to facilitate reviewing the code. Defects found by the reviewer are first posted to the Wiki to give an opportunity to the original developer to make fixes. Applying the principle of collective code ownership, the reviewer or other members of the development community may also perform fixes. Usually this process is fluid in the sense that the reviewers and the original developer communicate (via email, Wiki, or phone calls) to ensure a common understanding of the defect and the proper fix. Developers endeavor to fix defects as soon as possible in the current version of the source code, to prevent lingering defects that may propagate to other branches and releases of the software. In those cases where it is decided there is not an immediate solution, defects may be entered into the defect tracking repository. In this way, all defects found by reviewers are addressed by the team before the code is released.

As important as how the code review takes place is *when* the review takes place. All source code must complete a code review process before being included in an IGSTK release.

Code reviews are an important, arguably the most important, quality technique that can be applied to software development. Code reviews complement automated unit tests and code analysis tools by adding a dimension of expert evaluation to the source code. Further, code reviews reinforce the principle of collective code ownership and common source code understanding. IGSTK component developers are required to perform code reviews on every line of source code included in a framework release; IGSTK application developers are strongly encouraged to do the same.

5.2.3 Managed Communication

IGSTK was developed by a team of developers geographically distributed on a wide scale, and is intended to support the global community of interest in image-guided software for surgical applications.The IGSTK community makes use of websites, Wikis, and mailing lists to support the community. The main IGSTK website (http://www.igstk.org) has the latest news and information about IGSTK releases and other developments. The IGSTK Wiki (http://public.kitware.com/IGSTKWIKI/index.php) provides an online interactive forum for IGSTK developers. Modifications to the Wiki are restricted to authorized users.

The IGSTK community supports two mailing lists, one for developers and one for users. Participants on developers mailing list are core IGSTK developers. The users list is meant for support developers who intend to build applications on top of the IGSTK mailing list. If you choose to download IGSTK and build an application, we strongly recommend joining the users' mailing list to interact with the IGSTK community. You may join and view mailing list archives at http://public.kitware.com/mailman/listinfo/igstk-users.

One the foundational principles of open source development is *c*ommunity. The *o*pen in open source refers not just to the code but to the community. IGSTK intends to promote and support community involvement for the toolkit, and the principle vehicles for this are the communication tools described here. We encourage you to become a participant in the IGSTK community.

5.2.4 Configuration Management

Configuration Management (CM) is important for enforcing and maintaining the quality of software products. Most developers are familiar with using CM tools in projects large and small, but this use is often restricted to versioning files (or collections of files associated with a job) for the purposes of evolving and maintaining software. A common scenario is for a developer, when addressing a defect entered in the defect tracking repository, to *c*heckout the associated files, create a fix, locally test the fix, and check the files back in to the code repository. If any changes were made to the same files by other developers, the CM system will typically inform the developer and provide various pessimistic or optimistic strategies for resolving the issue. Another common scenario is for a developer to be working on a new feature for the next release, and to create new source files or modify existing ones. Again, if changes are required to existing files, these new versions are checked in to the code repository under the purview of the CM system.

IGSTK uses the popular and well-known Concurrent Version System (CVS) tool for version control of software products. The commands needed for obtaining IGSTK and supporting software packages such as ITK from CVS are described in Chapter 2. Developers are encouraged to use anonymous CVS access to download IGSTK. You may use the *e*xport cvs function to obtain a copy, or do a regular *c*vs checkout if you wish to maintain local repository information for the purposes of tracking histories, differences, and new versions of the software you have downloaded. Detailed online references for CVS may be found at CVS book and CVS wiki.

IGSTK recognizes CM as an important quality process, one that must be supported and integrated throughout the entire software lifecycle. IGSTK uses CVS to support defect fixes and the addition of new features as described by the above scenarios. However, IGSTK goes further by supporting best practices in CM such as minimizing branch creation to reduce product version proliferation, and establishing multiple *c*odelines and distinct *c*odeline policies for source code at different stages in the development process. A *b*ranch, in CM terms (and specifically in CVS) is a named (or *t*agged set of source code files that are then modified in a copy independent from the codeline from which the branch originated. Branches are a useful tool for supporting development on a codebase that is undergoing multiple types of changes but is under the same quality constraints. For example, the main codeline (or *t*runk is often reserved for new feature evolution, while a branch is created from a stable release of the code for the purposes of supporting defect fixes on the code while not disturbing new feature development. At a defined point in the lifecycle of the new release, defect fixes may be *m*erged back into the main codeline. This activity is usually controlled by a Change Control Board (CCB) comprised of representative of all stakeholders of the codeline. Another reason to branch might be to support custom features for a particular customer or project that are not targeted for the main product line. When a branch is no longer needed (because the release or custom project is no longer supported) it is deprecated. While branches are useful for these scenarios, branch proliferation can be problematic to manage. A single change anywhere in the codebase now has to be reviewed by more stakeholders to determine if that change applies to them. In effect, it creates multiple release versions of the software, resulting in additional complexity for requirements change management and testing processes. Therefore generally accepted axioms are to branch as late as possible and as seldom as possible to avoid this complexity. IGSTK, during its development, did not create branches, though it did employ separate codelines.

A *c*odeline is a repository of source code distinct from other repositories. The main codebase, or trunk, referred to above, is a codeline. A codeline has an associated set of rules that govern where, when, and what developers may commit changes to the codebase. The *w*here determines what branch of a codeline to commit a modification, this was described above. The *w*hen determines at what point in the lifecycle of that codebase the modification may be submitted. The *w*hat defines the criteria by which modifications are allowed, possibly including *w*ho may submit such modifications. The when, what, and who of codeline management is critical is applying different quality criteria to different source code files. For example, the main product codeline may define that only developers on the product development team may commit changes, and may only do so after a complete unit test and code review of the source code. An experimental codeline, where developers are working collaboratively in an evolutionary manner on a high-risk feature, may define lower quality standards such as informal walkthroughs and lesser stylistic checks in order to facilitate rapid development. However, note that if an exper-

imental feature were to be targeted for the main product line, the code could not be moved to that line without upgrading the quality checks to meet the quality policies defined for the main codeline. In this way, using separate codelines better enforces quality standards than simple branch-and-merge.

Since IGSTK included the development of several innovative features as well as an innovative architecture, it used a multiple codeline approach known as *s*andboxing. A sandbox is a separate codeline with lesser quality policies where developers working collaboratively could prototype high-risk code. For these mini-projects that were deemed successful, the code was then subjected to more rigorous quality policies so it could be moved into the main IGSTK codeline. The quality policies on the IGSTK main codeline include adherence to IGSTK style guidelines (see section XXX), unit tests present for all behaviors, complete code coverage (every source line executed, see section XXX), no dynamic analysis (memory management) defects, cross-platform verification, traceability of the modification back to requirements, and a complete code review according to IGSTK's defined code review practices (see section XXX). Sandboxing does have implications for building production versus experimental releases as discussed below in section XXX.

Going forward, IGSTK will support periodic releases, and support tagging and branching in CVS as appropriate to support maintenance of those releases. These policies will be posted on the IGSTK wiki. Future component developers for the toolkit will be expected to adhere to the quality criteria described for the main product line. Application developers are encouraged to follow IGSTK's CM patterns when developing and supporting applications that depend on IGSTK. Only through the application of consistent quality policies can quality and safety of software for surgical environments be achieved. For further reading on best practices for CM, we recommend [XXX] and [XXX].

Finally, we emphasize one related practice in IGSTK that is often not considered as a CM-type problem. Unlike many component-based systems being created today, IGSTK specifically avoids run-time configuration of framework behaviors via configuration files. While many component-based systems, such as web-based applications or embedded systems, use external configuration files to set run-time behavior and component communications, IGSTK avoids this in favor of compiling a single pre-configured binary version. In this way, clinicians do not have to worry about misconfigured software deployments in the operating environment. This is an example of another configuration management best practice, as it ensures that the behaviors deployed in a given environment have been enforced by the compiler and build process in general. In effect, instead of many possible runtime versions of the software in an operating environment, there is exactly one version deployed, and it is the version targeted for that environment. In the next section we elaborate on the IGSTK build process and how it helps enforce safety.

5.2.5 Build and Release Management Processes

1. IGSTK Release Cycle
2. use of CMake
3. Dependencies on ITK, VTK, etc. - did we ever build in compile-time safety against
4. Deploying IGSTK

```
1. emphasis on compile-time safety over run-time configuration options
2. How to read an IGSTK logfile
```

5.2.6 Continuous Testing using DART

IGSTK relies on extensive automated unit testing to ensure all lines of code are verified along some execution path. Automation is provided by CMake and DART integration. DART is a regression testing system that supports web-based report generation for a variety of test types. The web reports generated by DART creates an online *d*ashboard that the entire team uses on a daily basis to understand exactly what the quality level of the source code is at that instant in time. A sample IGSTK dashboard screenshot is shown in Figure XXX below.

```
Need a DART screenshot here
```

ITK developers should already be familiar with the DART dashboard concept; the philosophy of complete continuous regression testing in IGSTK was adapted from ITK. Readers familiar with Chapter 14 of the ITK Software Guide [XXX] will recognize the following descriptions of test types, slightly abridged from that chapter for IGSTK:

1. Compilation. All source and test code is compiled and linked. Any resulting errors and warnings are reported on the dashboard.

2. Regression. Some IGSTK tests require comparing test output against a valid baseline image. If the images match then the test passes. The comparison must be performed carefully since many graphics systems (e.g., OpenGL) produce slightly different results on different platforms. IGSTK also performs regression tests on Tracker-related operations.

3. Memory. Problems relating to memory such as leaks, uninitialized memory reads, and reads/ writes beyond allocated space can cause unexpected results and program crashes. IGSTK checks for run-time memory errors using Valgrind, a freely available open source debugging platform for Linux.

4. PrintSelf. IGSTK follows the ITK practice of having each class implement a PrintSelf method to print out all instance variables. The CMake-configured unit test driver checks to make sure that this is the case.

5. Unit. Each class in IGSTK must have a corresponding unit test where all class functionalities are exercised and quantitatively compared against expected results. These tests are typically written by the class developer and should endeavor to cover all lines of code including Set/Get methods and error handling.

6. Coverage. IGSTK unit tests should ensure that each and every line of code is executed at least once by the suite of available unit tests. This is commonly referred to as *n*ode coverage, and is the most important type of coverage for IGSTK as the state machine

architecture ensures that conditionals are kept to an absolute minimum, thereby reducing the need for edge and full path coverage. For safety purposes, the goal is 100 percent coverage for source code committed to the main codeline. In practice the IGSTK dashboard has usually been at 90 percent coverage or greater.

7. Style checking. The KWStyle tool described earlier in this chapter also provides a table-formatted display of stylistic violations.

These test types are augmented by a set of demo applications that exercise IGSTK functionality, and a state machine validation tool to ensure proper construction and execution of component state machines (see appendix XXX).

Each weekly teleconference of IGSTK developers includes a review of Dashboard status and action items to bring any quality parameters back inline if they are out of bounds. The focus is on continually and aggressively maintaining safety and quality, and the DART dashboard provides the best way to do this. DART is freely available; IGSTK application developers are encouraged to setup their own DART server and run cross-platform unit tests on a nightly basis. The IGSTK community is available to assist with this process.

Finally, IGSTK believes in defect tracking, and uses a customized implementation of the open source PHP BugTracker. The IGSTK team addresses defects as soon as they are found, but any defects that remain unresolved for longer than a short window of time are entered into the defect tracking repository. The defect is revisited at least on every internal release boundary.

5.3 Software Development Process Summary

IGSTK employs an agile philosophy for the development of safety-critical software. While the lightweight nature of agile methods might give some cause for concern, we believe that vigilance in the application of the process is the most important aspect for creating quality. A process is not a good process if it is not followed by the developers.