SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2011; **41**:945–962
Published online 25 April 2011 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.1075

Agile methods for open source safety-critical software

Kevin Gary[1,*,†], Andinet Enquobahrie[2], Luis Ibanez[2], Patrick Cheng[3], Ziv Yaniv[3], Kevin Cleary[4], Shylaja Kokoori[1], Benjamin Muffih[1] and John Heidenreich[1]

[1]*Department of Engineering, Arizona State University, Mesa, AZ 85212, U.S.A.*
[2]*Kitware Inc., Clifton Park, NY 12065, U.S.A.*
[3]*Imaging Science and Information Systems (ISIS) Center, Department of Radiology, Georgetown University Medical Center, Washington, DC 20007, U.S.A.*
[4]*The Sheikh Zayed Institute for Pediatric Surgical Innovation, Children's National Medical Center, Washington, DC 20010, U.S.A.*

SUMMARY

The introduction of software technology in a life-dependent environment requires the development team to execute a process that ensures a high level of software reliability and correctness. Despite their popularity, agile methods are generally assumed to be inappropriate as a process family in these environments due to their lack of emphasis on documentation, traceability, and other formal techniques. Agile methods, notably Scrum, favor *empirical process control*, or small constant adjustments in a tight feedback loop. This paper challenges the assumption that agile methods are inappropriate for safety-critical software development. Agile methods are flexible enough to encourage *the right amount* of ceremony; therefore if safety-critical systems require greater emphasis on activities, such as formal specification and requirements management, then an agile process will include these as necessary activities. Furthermore, agile methods focus more on continuous process management and code-level quality than classic software engineering process models. We present our experiences on the image-guided surgical toolkit (IGSTK) project as a backdrop. IGSTK is an open source software project employing agile practices since 2004. We started with the assumption that a lighter process is better, focused on evolving code, and only adding process elements as the need arose. IGSTK has been adopted by teaching hospitals and research labs, and used for clinical trials. Agile methods have matured since the academic community suggested almost a decade ago that they were not suitable for safety-critical systems; we present our experiences as a case study for renewing the discussion.

Received 10 September 2009; Revised 15 December 2010; Accepted 31 January 2011

KEY WORDS: agile methods; software process; open source; safety

1. INTRODUCTION

It seems to be a universally accepted maxim that agile development methods are not suitable for safety-critical domains. Agile methods, as the argument goes, do not encourage formal, document-centric activities needed to satisfy robust process requirements, such as documented design, requirements management, and other forms of traceability. Although the *Agile Manifesto* [1] was introduced almost a decade ago, only in the past couple of years has the maturity of agile methods become apparent through industry adoption, availability of reference materials, certification processes for individuals, increased scholarly activity, and convergence on a focused set of agile process models. This recent evidence suggests that now is a good time to step back and reconsider the broader implications of agile methods for safety-critical software.

*Correspondence to: Kevin Gary, Department of Engineering, Arizona State University, Mesa, AZ 85212, U.S.A.
†E-mail: kgary@asu.edu

Open source is an increasingly popular development and distribution model for software. Open source teams often employ agile methods, as the focus is on concurrent development and fast production (sprints) over gated production (milestones). However, unlike some of the core principles of agile methods, many open source projects rely on dedicated and highly skilled volunteers (or part-time supporters) in distributed development teams with no singularly available customer.

The image-guided surgical toolkit (IGSTK) is an open source project that relies on the collaboration of a skilled distributed development team to construct a cross-platform application framework in a safety-critical domain. IGSTK provides features for image-guided surgery (IGS), including DICOM image import, image display, registration, and segmentation. It provides advanced functionality such as device tracking support and scene graph manipulation, and non-functional features such as portability across a variety of operating systems. The goal of the project is to facilitate developing research and commercial applications faster by providing a reusable application framework. Researchers and entrepreneurs developing surgical applications can quickly build functionality by using core IGSTK components or modifying them to fit their needs. Framework usage reduces the need to be concerned about the intricate details in developing a safety-critical surgical application [2]. IGSTK is a rare intersection of agile and open source processes in a safety-critical domain.

Extreme care is needed in the design and development of safety-critical applications, because the occurrence of an error could result in loss of life [3]. The software engineering research community suggested shortly after agile methods first started becoming popular that they were not suited for building safety-critical systems (particularly Boehm [4], which we discuss later in this paper). The reasons range from a perceived lack of documentation, formal specification, to detailed planning. Agile methods have matured past the hype phase into a maturely defined, understood, and executed family of process models. We argue that agile methods can contribute to safety-critical software development, particularly in the areas of process management and implementation quality. Specifically, Scrum process management, eXtreme Programming (XP), and open source development principles can enhance traditional safety activities. The IGSTK team has enhanced the process with a tailored set of best practices augmenting common agile and open source methods for the express purpose of delivering safety-critical software. In this paper we make an argument for using these methods based on our experience with IGSTK since 2004.

2. BACKGROUND

Software safety is a necessary quality attribute in certain classes of systems because of the impact it has on life or property. Software safety deals with minimizing threats or risks to the system and mitigating loss in the event of failures or adverse events. Leveson [5] defines risk as 'a function of the probability of a hazardous state occurring in a system, the probability of the hazardous state leading to a mishap, and the perceived severity of the worst potential mishap that could result from the hazard'. Consequently, while engineering for safety in safety-critical systems, high levels of assurance are needed (not solely based on testing) that will determine whether a system can and should be used [5, 6]. Leveson also argues that safety should not only prevent malicious actions from happening in general, but should also be concerned with inadvertent actions happening.

Are agile methods appropriate for safety-critical systems? In [4] Boehm performs a comparative study of agile methods vs plan-driven or traditional methods in developing software development methods and asserts that it is important to know which method is applicable to what type of project. Boehm suggests that life critical systems need stable requirements and that an agile approach might not be suitable for such applications. Further, Boehm suggests that more thorough planning will reduce risk in developing such systems.

We question the underlying assumptions of Boehm's argument. First, there is an implication that agile methods do insufficient planning and that more thorough planning eliminates risk. We disagree; agile methods, particularly Scrum, do plan while focusing on empirical process control [7] as a constant oversight and management mechanism. Scrum plans by managing the product

DOI: 10.1002/spe

backlog, which may include activities related to risk reduction, HA (hazard analysis), FTA (fault-tree analysis), FMEA (failure mode effects analysis), and formal specification (where warranted). Further, Scrum advocates pervasive process management through sprints, daily standups, and visibility (report dashboards and related tools). Second, Scrum recognizes that knowledge is incomplete, change happens, and provides a framework for dealing with it, instead of *ad hoc* workarounds outside the boundaries of a plan-oriented process model. Finally, Boehm's discussion does not consider the impact of the expertise on the team. The omission implies that team members are interchangeable parts, whereas Cockburn [8] argues for their inclusion in the process. Formal specifications, models, tools, and processes reduce risk in the application domain, but software, perhaps more than any engineered discipline, is the output of a human-oriented process. The expertise of the team, the communication patterns it utilizes, and the human capacity to reason with incomplete knowledge are important factors in determining that output (the software). The IGSTK team consists of experts specialized in the image-guided surgical domain and from other fields, such as software engineering and robotics. The best practices presented later in this paper reinforce the role of the experts in this process.

In an eWorkshop report [9], 18 agile experts from around the world evaluate Boehm's statement regarding life critical systems and agile methods. They contend that when performance requirements and plans about level of testing for the project are made at an early stage in development, agile methods are ideal for safety-critical systems because in an agile approach the customer is available throughout the development process to obtain and clarify requirements. In [10] Gelowitz *et al.* acknowledge this claim after performing a step-by-step comparison of XP process against a traditional waterfall method considering the concept, requirements, design, implementation, test, and maintenance phases theoretically. They say XP performs all the phases better, except it does not create elaborate design document during the design phase. However, they also say that not creating elaborate design documents provides flexibility with respect to accommodating changes.

Several industry experience reports suggest that adopting an agile approach has improved the efficiency and reliability of their project. Spence [11] discusses how an organization had to adopt an agile methodology due to the shortcomings of traditional plan-based methods in handling frequently changing requirements. The author claims that the team reviewed agile methodology in detail and is confident that it can be used to develop safety-critical software. Similar to IGSTK, their approach in becoming agile was incremental, trying and adapting agile principles on a constant basis. Van Schooenderwoert [12] says their team benefited by adapting XP in an embedded project. The author claims that XP is ideal when requirements are not concrete at the beginning of the project, asserting that agile principles work best for volatile or ambiguous requirements. In [13] van Schooenderwoert explains the importance of agile testing in embedded projects. The project team performed unit testing on the software and used mock object simulation to test the hardware on the embedded project. The author asserts that the project displayed very few bugs at any given point of time because of this testing approach. Another industry report by Manhart *et al.* [14] describes the development of embedded software for Daimler Chrysler including safety-oriented functions like Automatic Breaking System (ABS). The authors point out how the team, fundamentally oriented towards plan-driven development in order to mitigate risks, had to adopt agile practices to manage changes to their requirements efficiently at a later stage of development. Grenning [15] and Bowers *et al*. [16] describe how they have successfully used agile principles on their large mission critical projects. They claim that adopting agile principles, such as pair programming, iterative development, refactoring, and automated testing, has helped to improve the quality of the code and productivity of the team.

Agile methods and open source are suitable for safety-critical systems because these methods are synergistic with safety principles, not orthogonal to them. Agile methods bring strong practices in the area of process management and software construction, while having a philosophy that allows for traditional safety-oriented practices *to the extent they are warranted*. Understood in the proper way, agile methods reinforce Boehm's argument in [4] that thorough project management is required for safety-critical software development. Agile methods can minimize risk by reducing the probability of loss through their best practices. Yet they allow for traditional analysis methods

DOI: 10.1002/spe

that help determine the magnitude of the loss, thereby allowing the team to determine the risk exposure.

The following section presents an overview of IGSTK as a point of reference in our position, and describes some of the unique challenges it presents.

3. AN OVERVIEW OF IGSTK

IGS presents interesting design challenges for software and system implementers. A typical IGS environment is shown below.

Figure 1 depicts several challenges. Physical devices, such as robotic arms and tracking devices, need to be integrated in a controlled environment. The software must be usable by specialized users, such as surgeons and surgical assistants. The software system must have fail–safe mechanisms, provide correctness and soft real-time performance, and ensure safety. IGSTK must exhibit these traits, and also address additional architectural qualities, including portability (hardware and operating systems), reusability, maintainability, and openness.

IGSTK is an open source framework for creating surgical applications. IGSTK is distributed under a BSD-like license that allows for dual-use between academic research labs and commercial entities. IGS involves the use of preoperative medical images to provide image overlay and instrument guidance during procedures. The toolkit contains the basic software components to construct an image-guided system, including a tracker and a four-quadrant view incorporating image overlay, as shown in Figure 2.

The remainder of this section describes the challenges that IGSTK presents from software process and architecture perspectives.

3.1. Software process challenges

IGSTK development presents interesting challenges from a process perspective. The first challenge derives from the nature of the framework-level requirements, which is difficult to completely understand before applications are constructed upon it. Waterfall-style methodologies [17] that attempt to define requirements completely before development begins are not considered suitable, as the range of possible behaviors to be supported on the framework is necessarily incomplete. Similarly, use-case analysis modeling [18] is selectively applied, as one cannot assume requirements derived from a set of applications today represent a complete set of requirements for the future. Given the complexities of the requirements process and application domain, we discuss IGSTK's approach in Section 4.

The second challenge is the makeup of the team, comprising academic and commercial collaborators in a distributed setting. All of the team members have other demands on their time. These factors create challenges for setting project deliverables and expectations over medium- and

Figure 1. Components of an image-guided surgical environment.

DOI: 10.1002/spe

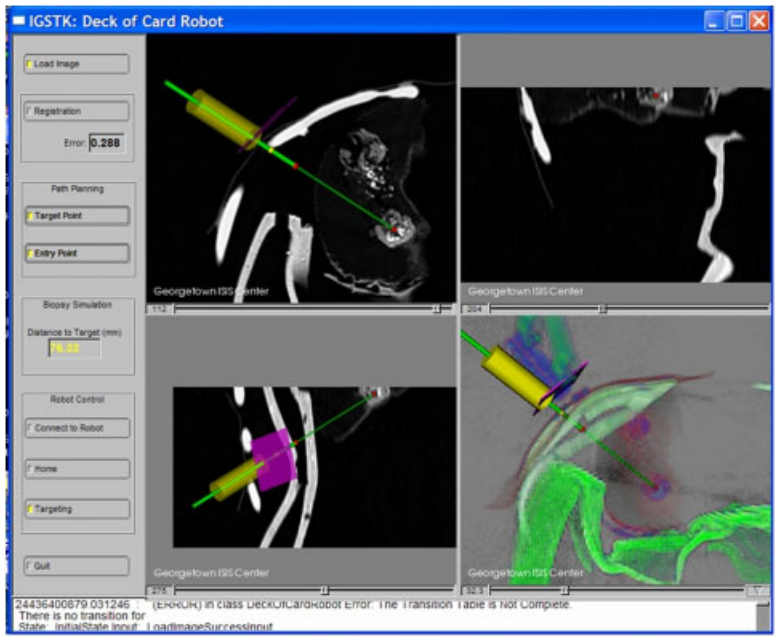


Figure 2. 4-up GUI for a robotic needle driver application.

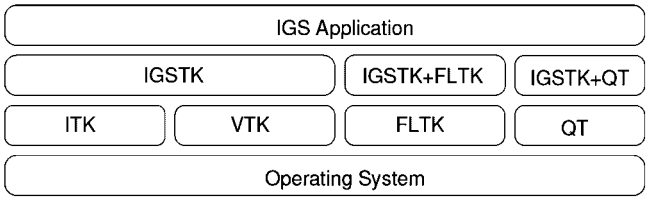


Figure 3. IGSTK software dependencies.

long-term horizons. Fortunately, most developers are deeply familiar with the domain and have significant exposure to agile methods. Their expertise combined with the shorter horizons (Scrum sprints) for delivering software mitigates this issue.

Another challenge, the high quality standards of the application domain, suggests that both agile and traditional practices should be used. Safety-critical software should undergo code review [5, 6], which IGSTK does at the end of each sprint. The agile practice of comprehensive unit testing and full code coverage ensures that each component is extensively tested [19]. Additionally, the open source community exercises the released code at an early stage, finding defects and usability issues to evolve a stable codebase. Finally, as discussed in section 2, there are no constraints in the agile process against performing various safety analysis techniques. However, the IGSTK has not performed these safety techniques as they are more appropriately applied to specific surgical applications, not at the framework level.

3.2. Architecture challenges

Figure 3 shows IGSTK's dependencies and its role in support of an IGS application. The top layer corresponds to IGS applications built on top of IGSTK. IGSTK also interfaces with third-party tools like ITK and VTK to provide image-related functionality. FLTK performs user interface-related tasks; other tools like Qt could be used for this purpose as well. The bottom layer, the operating system, forms the base of this entire architecture. As shown in Figure 3, the IGS applications interact with the lower layers only using IGSTK APIs.

DOI: 10.1002/spe

A significant challenge is providing a safe platform when there are many third-party dependencies. The IGSTK layer is a *safety region*, wrapping underlying library functionality, and decorating them with safety attributes. The state machine is a significant abstraction in this approach, and is described in detail in Section 4.3.

The distributed nature of the core development team is not only a process challenge but also an architecture challenge. The lack of good communication practices and strong open source evolutionary principles could lead to a fragmented architecture. The IGSTK strong communication patterns, coupled with a strong central design pattern (the state machine), resulted in a stable and consistent architecture.

Another challenge is maintaining proper safe configurations of the software when components may be assembled based on varying application requirements. IGSTK does this in two main ways. First, run-time configurability of the software is kept at a minimum. The only configuration possible is scoped internally to tracking components. Component *connectors* are determined at compile-time and verified through strong typing mechanisms enforced by the toolkit; no configuration files exist to wire components at run-time. Second, each compiled configuration is verified through a continuous integration and build process.

4. AGILE METHODS AND IGSTK

Lutz [20] proposes six key areas to consider when engineering for safety. These include:

1. *Hazard analysis*: identification and analysis of hazards in terms of their severity of effects and likelihood of occurrence.
2. *Safety requirements specification/analysis*: specify requirements in formal notation, allowing formal analysis to investigate whether certain properties are perceived.
3. *Designing for safety*: focus on consequences to avoid in the general system context.
4. *Testing*: tests should demonstrate that the software responds appropriately to some anticipated environment.
5. *Certification and standards*: this involves assessing it against certain criteria.
6. *Resources*: utilizing resources and books for good software safety engineering [21].

IGSTK focuses on 2, 3, 4, and 5 from an agile perspective. Classic safety analysis techniques (#1), such as HA, FTA, and FMEA, are not the issue in our research. These techniques have been shown to be useful for software safety, and there is nothing about an agile process that suggests a project should omit or de-emphasize them. Resources (#6) are not in conflict with agile methods; this best practice is software life cycle model agnostic. To reiterate, the overriding principle is the 'right amount of ceremony', therefore these techniques should be applied to the extent they are required, no more no less.

Formal methods (#2) enhance the safety and reliability of safety-critical systems [22, 23] by providing models and model transformations that can be analyzed for completeness and correctness. Models are constructed in a language with well-defined semantics; then, a correctness-preserving transformation turns the model into an executing program, supported by an environment that guarantees certain runtime properties. If the model, through formal analysis and/or simulation, is accepted as correct, then the executing code is accepted as correct. However, the IGSTK team has adopted an agile approach in developing the framework, synergistic with the practices of open source tool development but orthogonal to traditional approaches [2]. Agile software development practices emphasize working software as the most important measure for progress. This means developers usually implement the software 'by hand', with no formal design specifications or modeling tools, or the use of automatic code generators. Thus, developers have to constantly communicate, re-design, re-code, and re-test software throughout the entire life cycle of the project. Agilists argue that this approach results in better low-level code quality and adherence to user/customer intent, and that formal modeling may result in constrained requirements that do not meet customer needs.

DOI: 10.1002/spe

The IGSTK team believes that formal models are useful, but does not want to change from an agile process model to leverage their benefits. The model-driven community espouses forward, waterfall-like flow to guide the development safe software. Agilists prefer to work in short fluid iterations ('sprints'). IGSTK attempts to leverage the best of both worlds by using an agile process while incorporating formal checks integrated into that process via a validation toolset as described in Section 4.5.

IGSTK is designed for safety (#3). From the very beginnings of the project the team adopted an architectural style based on layered components governed by state machines. This safety-by-design principle is pervasive to IGSTK, and is presented in Section 4.3.

IGSTK employs extensive testing (#4), particularly at the component level. A continuous testing and integration dashboard reports on automated unit tests run around the globe on a variety of platforms. Application-specific testing, other than the sample applications included in the source tree, does not apply, as IGSTK is a framework, and not a specific application. This agile testing methodology, presented in detail in Section 4.4, adheres to the recommendation of Leveson and Turner [19] to test extensively at the component and system levels.

Software engineered for safety often must meet liability laws and standards set forth by government agencies or licensing bureaus (#5). Unfortunately, the current models do not guarantee that builders of safety-critical systems meet the requirements of the regulatory institutions [5]. Software safety should ensure that the system executes properly without unacceptable risk. The risk that is considered acceptable usually involves other factors (economical, political, and moral) defined outside the realm of software engineering. IGSTK, as an open source framework for IGS, must deal with FDA approval, IRB review, and open source licensing issues. IRB review for surgical applications within a hospital setting is based on the surgical procedure, and not on a specific technology. IGSTK has elicited requirements from surgeons for example surgical applications, constructed activity models for these procedures, and received IRB approval for clinical trials at a major research university hospital [24]. FDA approval is challenging as software is treated as a medical device [19, 25], therefore design and traceability are important. In response, IGSTK has adopted a lightweight but integrated requirements management process described in Section 4.2, and the safety-by-design approach presented in Section 4.3. Finally, as an open source framework, IGSTK must deal with unique licensing issues, both in terms of what it relies upon and who uses it. IGSTK relies on other open source platforms (see Section 3) and must be disseminated as open source. This restricts which third-party software may be integrated, which is sometimes an issue for certain algorithms or commercially available tracker devices.

In the remainder of this section we describe how agile methods have been applied, and in some cases augmented, to meet IGSTK's safety-critical needs.

4.1. Best practices

Early on the team recognized the need to establish a collaborative set of principles, or agile culture, on the project. These were expressed as a set of best practices [2] that take precedence over dogmatic adherence to scripted processes. An abridged version is given below.

Best Practice #1. Recognize that people are the most important mechanism available for ensuring high quality software [8]. The IGSTK team comprises developers with a high degree of training and experience with the application domain, supporting software, and tools. Their collective judgment is weighted over any high-level process mandate.

Best Practice #2. Promote constant communication. This is difficult in open source projects with distributed and part-time teams. IGSTK members constantly communicate through weekly teleconferences, biyearly meetings, mailing lists, and an active Wiki.

Best Practice #3. Produce iterative releases. IGSTK's external releases coincide with IGSTK yearly user group meetings. Internally, releases are broken down into approximately two month 'sprints'. At the end of a sprint, the team can stop, assess and review progress, and determine what code is considered stable enough to move to the main code repository.

DOI: 10.1002/spe

Best Practice #4. Manage source code carefully. Require 100% mainline code coverage. Use sandboxes for evolving code with different check-in policies to allow developers to share experimental code without sacrificing quality policies in the mainline.

Best Practice #5. Augment the validation process with reverse engineering tools that support complex white-box testing. This best practice is a nod to the specialty of the domain.

Best Practice #6. Emphasize continuous builds and testing. IGSTK uses the open source CDash tool to produce a nightly dashboard of builds and unit tests across all platforms. Developers are required to ensure that code coverage stays as close as possible to 100%, that their source code builds on all supported platforms, and that all unit tests pass.

Best Practice #7. Support the process with open tools. IGSTK uses CDash, a test dashboard, CMake, a cross-platform build solution, and Doxygen, a documentation system.

Best Practice #8. Emphasize requirements management in lockstep with code management. As requirements evolve and code matures, it is necessary to adopt flexible processes for managing requirements and source code. The organization and tracking of requirements is a complex process for a project such as IGSTK, and thus is detailed in the following section.

Best Practice #9. Focus on meeting exactly the current set of requirements. This is one of the most important benefits of an agile approach. The team focuses on only the current backlog, and not burdening itself on designs not realized in the current code.

Best Practice #10. Allow the process to evolve. Through constant communication, IGSTK members recognize when the complexities they face can be addressed within the current process, when 'tweaks' are required, or when new practices should be adopted.

These best practices are not new to agile and open source practitioners. However, in a safety-critical domain, following these practices alone is insufficient. In the spirit of doing the *right amount of ceremony*, the IGSTK team augments these practices. To illustrate, we describe how the IGSTK team performs lightweight requirements management (#8), safety-by-design (#9), continuous integration and testing (#6), and architecture validation (#5).

4.2. Requirements management

Requirements management is described in best practice #8, traceability of requirements at development time. Developers introduce new requirements into the product backlog. The team then selects the subset of the product backlog suitable for implementation in the current sprint. The team employs a collaborative process for reviewing, implementing, validating, and archiving these requirements, and it is integrated with application development.

This process is illustrated as a UML state diagram in Figure 4. When a developer identifies a new potential requirement (*Initialized*), s/he will post a description (*Defined*) on the Wiki. At the same time, the initial code that fulfills the requirement is entered into the sandbox. The requirement undergoes an iterative review process where team members review, discuss, and potentially modify the requirement. Based on the team's decision, requirements can be *Rejected*/*Aborted* or *Accepted*. *Rejected* requirements are archived on the Wiki (*Logged*) so that they can be reopened later if necessary. *Accepted* requirements are included in the product backlog. Once the supporting software is implemented and its functionality confirmed, the requirement is marked as *Verified*. As nightly builds take place, all *Verified* requirements are automatically extracted into Latex and PDF files and archived.

This synchronization between requirements management and code management gives a more evolutionary process feel to backlog management.

4.3. Safety-by-design

IGSTK's layered component-based architecture is shown in Figure 5. Each component has a strongly typed interface that accepts request events and returns event responses. Events are translated into inputs to an internal state machine that determines whether the request can be satisfied in the component's current state.

DOI: 10.1002/spe

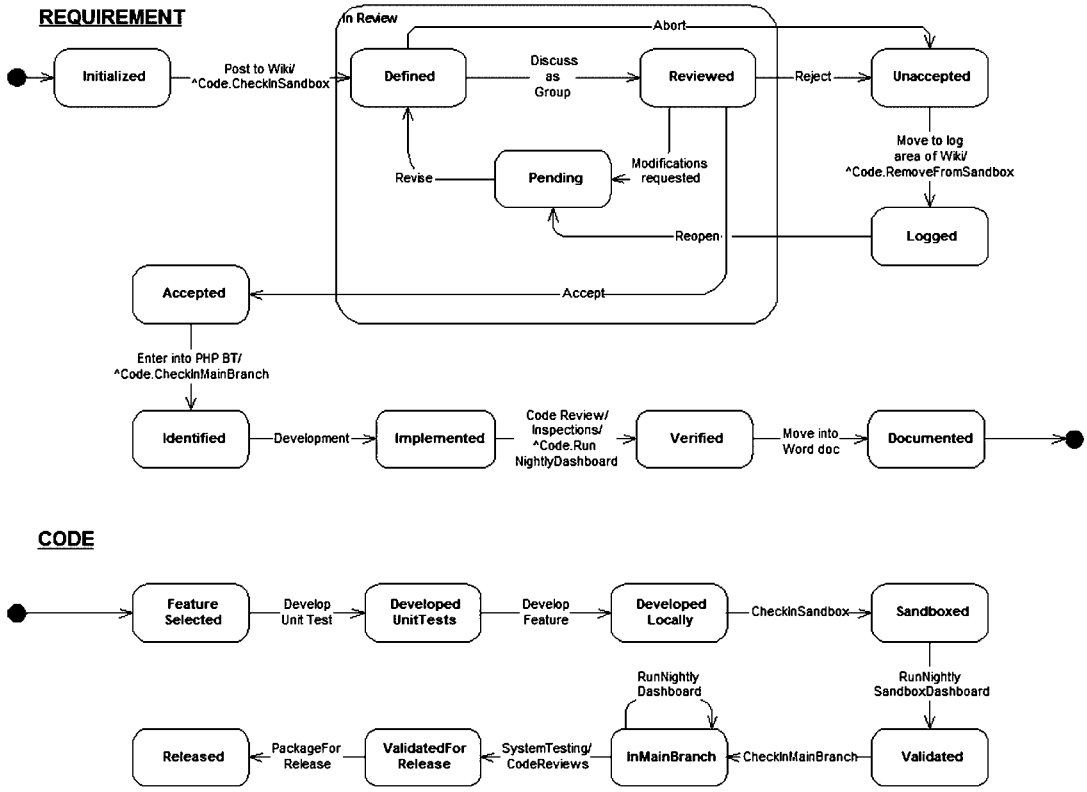


Figure 4. Requirements Management process in IGSTK.

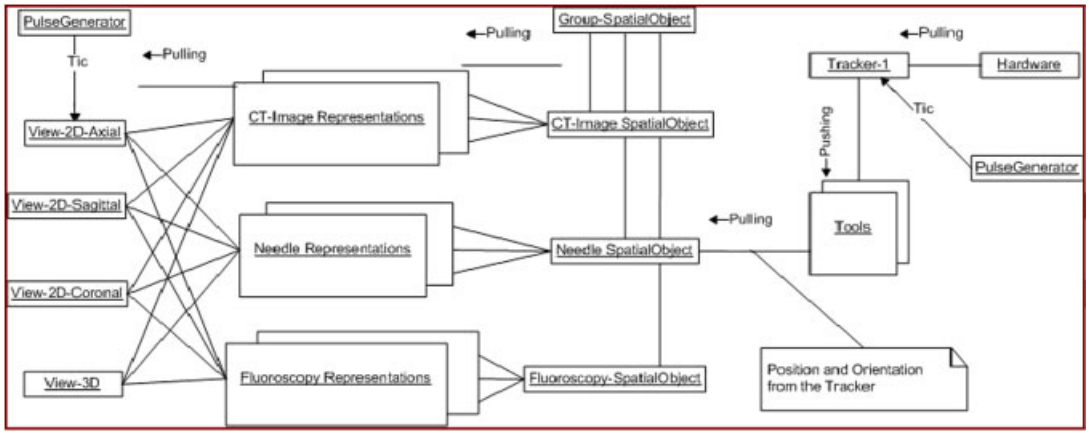


Figure 5. IGSTK components and connectors.

State machines are the principal mechanism for the safety-by-design approach. State machines govern each component instance, restricting the allowable behaviors that a component exhibits at a given point in time based on the state of the component. Strong encapsulation of state machines inside components means that they cannot be manipulated by the outside world other than through interaction with the specified interfaces. State machines have fully populated transition tables to ensure that any possible input has a defined response. State machines provide a reliable medium for high levels of assurance needed that will determine whether a system can and should be used in a safety-critical environment [26]. Safety-critical component-based systems like those used in the aeronautical, medical, and defense industries are often engineered using state machines in order

DOI: 10.1002/spe

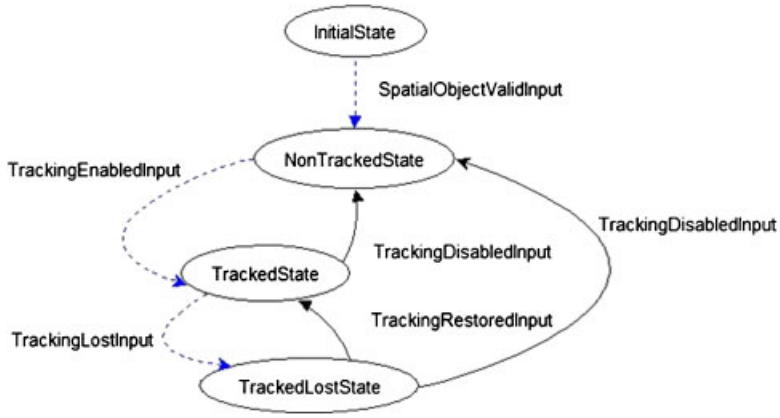


Figure 6. State machine for the Spatial Object component type.

to meet the safety regulations and standards set forth by liability laws and government agencies or licensing bureaus [5]. An example state machine is shown in Figure 6.

The inputs define valid transitions between states. Inputs are generated by transduction from requests (events) to the component. The state machine determines if a component is in a state where it can process that request (input). State machines were an early architectural decision that has persisted over the lifetime of IGSTK, resulting in a very stable architecture. The absence of architectural shift and rework is evidence of the platform's safety.

The safety-by-design approach in IGSTK supports Leveson's notion of *intrinsic safety* [5], wherein no component can be in an unexpected state. The U.S. Food and Drug Administration (FDA) as a necessary attribute for certified medical devices (software is classified as a medical device) require design safety [25]. IGSTK documents the critical elements of component design, including its state machine catalog, on the Wiki, and conducts manual inspections and automated validation of these design implementations.

4.4. Continuous integration and testing

Testing is essential to ensure software quality. It involves generating test cases, executing the application against the test cases, and comparing the results against expected results. Testing helps ensure that the software meets the functional and non-functional requirements specified for the project. Automating the test process is beneficial as it can generate and run a large number of test scripts and provide results in a faster and reliable manner.

IGSTK relies heavily on the agile practice of continuous and extensive unit testing with automated tool support. IGSTK requires 100% code coverage from its unit tests. Automation is achieved through the CTest tool, which posts results to a CDash dashboard (Figure 7). Dashboards are powerful tools for agile methods as they provide transparency into the quality of the software. IGSTK automates tests from computers at sites around the world.

An important contributing aspect of open source methods to the agile perspective is the leveraging of the community to evolve the software to a stable point. An open source community creates a ready population of early adopters that identify defects near to the time they are injected, and exercises a framework like IGSTK in ways that the development team often cannot anticipate. Figure 8 shows a bar 'stack' chart tracking the identification and resolution of the major and minor defects in IGSTK since 2006 on a quarterly basis.

Comparing the left and right bars in each pairing, particularly the lower part of each stack, shows that the vast majority of defects are resolved within the same quarter. The community-driven open source process helps to identify the defects, and the highly iterative agile sprints ensure defect resolution before the next release. The chart also shows the impact of dedicated stabilization sprints (the two spikes). The reduction in the total and major defects from quarter 11 on coincides with

DOI: 10.1002/spe

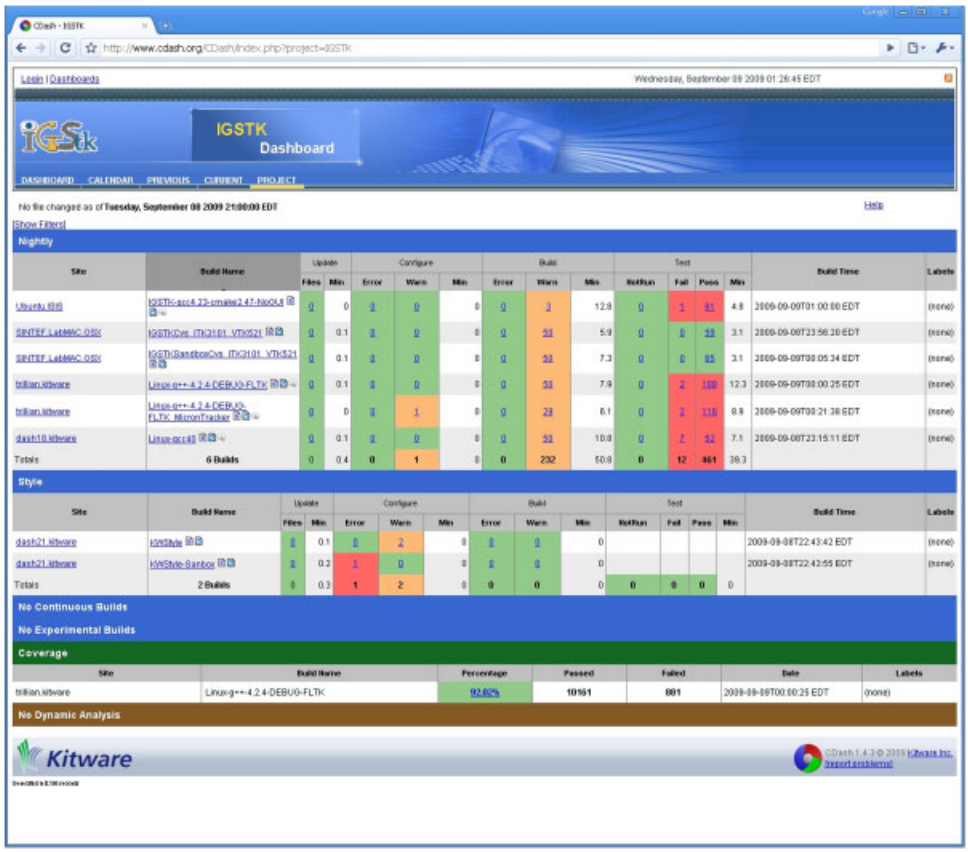


Figure 7. CDash dashboard displaying IGSTK nightly build and test results.

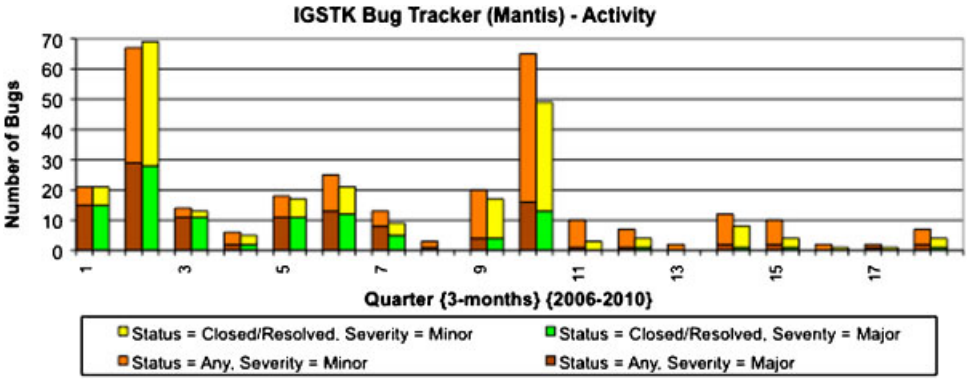


Figure 8. Quarterly defect tracker activity for IGSTK since 2006.

the release of IGSTK 2.0, supporting the hypothesis that early participation by the community leads to later stabilization of the software.

4.5. Agile architecture validation in IGSTK

Advances in model-checking tools should be leveraged in IGSTK, but there are no formal models created in the design process. To address this problem, the team constructed reverse engineering tools that extract models from the code, validates them through a simulation-based testing methodology, and posts the results to the dashboard. In this way the design is validated with the code as a

DOI: 10.1002/spe

part of the agile continuous testing process. The validation toolset is described elsewhere [24, 27], here we discuss the key features with respect to an agile process.

The center of IGSTK's validation tool suite is a simulator that accepts descriptions of state machines in SCXML [28] reverse engineered from the source code, and event sequences defined by a variety of sources, and 'runs' the events as a simulation. Observers of a simulation may provide a number of useful functions, such as constraint checking, execution visualization (animation), playback and record, and so on. The key is in determining an appropriate set of events to give to the simulator. The validation suite currently supports two kinds of event generators, *coverage* and *record-and-playback*. Record-and-playback allows developers to trace an execution of an IGSTK application and capture state machine inputs from a generated logfile. The inputs can then be replayed through a visual animation application or through automated testing components.

Coverage measures the percentage of requirements satisfied when a test criterion is applied [23, 29, 30]. Traditional code coverage mechanisms include statement coverage, branch coverage, and path coverage. Translated into state machines, the types are state, transition, and path coverage. State coverage involves checking that each state in the state machine of an IGSTK component is *reachable* from the initial state [31]. Transition coverage verifies that each pairwise transition between states occurs correctly as a result of processing a set of inputs. Path coverage is more involved, as (i) there exist expected path sequences for (functionally) correct component executions, and other sequences for unexpected and potentially catastrophic sequences and (ii) the state machine is an explicit representation of object state, which is conceptually long-lasting and repetitive—how many iterations over a path should a test conduct? Our coverage algorithms align with Watson and McCabe's work on cyclomatic complexity and basis paths [32]. Specifically, since exhaustive path coverage of state machines is not feasible, we employ structural- and domain-related heuristics (cf. [24]) to eliminate superfluous paths, reducing the total number of paths generated. This helps in prioritizing the paths of interest, providing a sophisticated testing mechanism for IGSTK.

We present a brief example to give a better idea about these heuristics. Consider the state machine for a spatial object shown in Figure 6. The dashed arcs in the figure represent a minimal set of inputs to process to achieve state coverage, yet this sequence has no reflection on the object's expected use. A structural (domain-independent) path heuristic might suggest that the next node in a recurring choice state (such as *TrackedState*) be the least recently visited. Thus if in the previous iteration of the path the *TrackingDisabledInput* was processed, then this time the *TrackingLostInput* is processed. Note that this decision is based on the structure and history of the path, and not on any information about what the state machine itself represents. This is different from a domain heuristic, which is implemented based on expectations of component usage. For example, a domain heuristic might cause the *TrackingLostInput* to be processed 3% of the time based on empirical evidence that the specific tracking device used in a clinical environment has a data loss rate of 3 frames per every 100. This may then be combined with a projected number of times around the loops present in this state machine based on the estimated lifetime of the spatial object during the given clinical procedure.

The need for continuous architecture validation motivated integration with the CDash dashboard. This integration requires formalized test statements, capturing results of test executions in the simulator, and posting results to the dashboard. Formalized test statements are constructed using a freely available business rules engine, Drools[‡]. Test statements specified constraints on the global state of the system, expressed as the union of states at a given time in each component state machine throughout the system. In this way the global state of the system can be validated. Rules are evaluated after the simulator processes each state machine input from a test input stream (generated from a replay or coverage algorithm), and results indicate whether test conditions pass or fail. These results are transmitted to IGSTK's dashboard. The significance of this process is that it shows (i) how activities grounded in formal methods can be incorporated on an agile project with the proper tool support and (ii) that such methods can be applied after implementation as a

[‡]http://jboss.org/drools.

DOI: 10.1002/spe

focused part of validation, instead of as an unbounded (heavy) task in the beginning of the process cycle—again, just the right amount of ceremony.

Before creating our own tool we reviewed the existing tools, such as SPIN [33], UPAAL [34], LTSA [35], and commercial offerings RoseRT and Rhapsody. These tools were either too burdensome, solved the wrong problem, or were not license compatible with IGSTK. IGSTK follows an agile approach. The IGSTK team does designs—on its Wiki. These designs are peer reviewed and a prototype implementation presented. IGSTK constructs models—state machine models to be exact—directly in the source code. The validation suite of tools checks the fidelity of the model's representation and execution semantics.

5. IGSTK AND SAFETY

Are agile methods appropriate for safety-critical systems? We claim yes, or at least that agile practices can contribute to a software process that results in safer software. This is a case study, but we are encouraged by our experiences since 2004. IGSTK is in use in 27 hospitals and research centers worldwide. IGSTK has been used in clinical trials for transthoracic lung biopsy approved by the IRB of a major university research hospital (for details see [36]), and received a determination of non-significant risk from the FDA. Finally, IGSTK is a principle component of the LUTi platform (http://www.luti.com.ar), a commercial image-guided surgical navigation product used in ten neurosurgery cases in Argentina.

Assessing the safety of IGSTK is difficult because it is a framework and not a specific application. One can evaluate a particular application such as the one described above as the surgical procedure has exact requirements and domain-specific constraints. The application software and how it uses IGSTK could be evaluated with traditional safety techniques, but to our knowledge no IGSTK user has conducted such an analysis. Instead we evaluate the architecture using a modified Architecture Tradeoff Analysis Method (ATAM [37]). This is appropriate as IGSTK is effectively an off-the-shelf architecture for surgical applications.

The modified process examined IGSTK documentation from all sources (Wiki, code, mailing lists, book, papers, etc.) to extract the key quality attributes and characterizations, and construct a quality utility tree, as shown in Table I. Attribute characterizations include two ratings, one for the importance of the item and the second the relative difficulty in achieving that quality attribute level. The table essentially describes risks and scenarios in a way that is compatible with Boehm's [4] notion of risk and loss.

We define architectural approach descriptions to map how the IGSTK architecture addresses the scenarios. Scenario descriptions for three of the four safety scenarios are included in the appendix

Table I. Quality Attribute Utility Tree for IGSTK.

Attributes	Quality sub-factors	Attribute characterizations
Safety	Framework misuse	**(S1) H, H**—Prevent framework misuse and ensure IGS applications access to a basic unified layer
		(S2) H, H—IGSTK classes won't throw exceptions in order to curb misuse
	Visual/instrumentation failure	**(S3) H, H**—Ensure that the surgical view is up to date
	Component failure	**(S4) H, M**—Provide logging when component failure occurs and provide failure message to user
Testability	Error detection	**H, M**—Provide logging when lower level component failure occurs and provide failure measure to user
		H, H—Create a set of predictable deterministic behaviors with a high level of code coverage 90%
	Code incorrectness detectability	**H, M**—Create a system of testing that uses sandboxing to test and prototype all release
Usability	Latency	**M,M**—Response Time for visualization reduced to smallest possible delay

DOI: 10.1002/spe

of this paper. The fourth (S4) is not included as logging occurs automatically in all scenarios in IGSTK, and the failure messages provided to end users are actually application-level requirements based on the surgical procedure.

The modified assessment process adds some rigor to the evaluation of how IGSTK addresses quality attributes, and in particular safety. But it is not a quantitative or formal analysis of failure risks and loss. A more traditional safety analysis using techniques, such as FTA or FMEA, should be performed on specific system instances that include IGSTK.

6. DISCUSSION

Leveson and Turner [19] recommend guidelines for safety-critical systems in their review of the Therac-25 medical system, which the authors note failed due to coding, and were not requirements errors. IGSTK follows these recommendations through agile and open source practices.

1. *Documentation should not be an afterthought*: documentation is created as the code is being developed through (a) a Wiki that is constantly updated, (b) automated online documentation created from the source via Doxygen, and (c) a freely available book which is created from the source tree via the same build process as the source code.
2. *Software quality assurance practices and standards should be established*: IGSTK has established practices and standards for quality. The practices have been described throughout this paper, from traditional techniques like code reviews to agile techniques such as pervasive unit testing.
3. *Designs should be kept simple*: IGSTK's design is simple; in fact simplicity is an overriding agile principle. Component designs share a common architecture pattern (the state machine), and design principles are enforced in code through strongly typed interfaces and specific macros. Designs are documented and discussed on the Wiki.
4. *Ways to get information about errors—for example, software audit trails—should be designed into the software from the beginning*: IGSTK's logging facility records each request and response in the system, providing a full audit trail. The lightweight requirements management process ensures lockstep requirement and code changes.
5. *The software should be subjected to extensive testing and formal analysis at the module and software level; system testing alone is not adequate*: IGSTK is rigorously tested at the unit level on a continuous basis. Formal analysis is incorporated into the agile process through a validation suite that reports results directly to the dashboard.

The IGSTK team has adopted an agile methodology and tailored it to their needs due to the nature of the project. IGSTK's agile approach is a combination of Scrum management practices combined with XP coding practices with the support of an open source community. IGSTK team members include experts from different fields, such as software engineering, computer graphics, imaging, and robotics. Software development is based on a set of best practices iteratively applied with continuous automated unit testing and 100% code coverage to ensure software quality. This 'best practices' approach is augmented by the right amount of 'heavier' practices taken from traditional approaches to safety-critical systems. But even these practices, as shown with requirements management, continuous integration and testing, and architecture validation, are integrated into the process in such a way as to reinforce, and not obstruct, the agile culture. While an agile purist might object that these activities are not agile, we believe the overriding principle of 'just the right amount of ceremony' justifies the selected application of heavier practices in areas critical to ensuring system safety.

We acknowledge that IGSTK's agile approach is neither as rigorous nor as complete as it could be for a safety-critical domain. A single case study, no matter how deep the experience, makes 'the right amount of ceremony' seem like a platitude, but we believe that many traditional process models (or instances of those models by software development organizations) forget the obvious and instead conduct a 'checklist execution' of the process. The tale of IGSTK's agile evolution, we

DOI: 10.1002/spe

think, offers lessons and hope for applying agile methods to safety-critical domains. A more agile process, augmented with key process elements and faithfully followed by the people that execute it, is a way to achieve safety. Agile is not the absence of process, and lightweight does not mean 'skip'. Agile means the right amount, adapting to change. Our position is supported only by our experience, and we acknowledge that the debate is too broad and nuanced for us to start to outline in this paper. The community assumes that agile methods cannot make a contribution in safety-critical domains, and that document-centric process models are the only option. The debate should be reopened.

APPENDIX: ARCHITECTURAL APPROACH DESCRIPTIONS

Scenario: S1 Prevent framework misuse and ensure IGS applications access to a basic unified layer

Attribute: Safety

Environment: Normal / Strained Use

Stimulus: Developer Misuse

Response: Classes that inherit lower level component behavior are restricted to basic functionality.

Architectural Decisions	Risk	Sensitivity	Tradeoff
Encapsulation of toolkit functionalities			T1
Layered architecture	R1		
Medium sized objects			T2

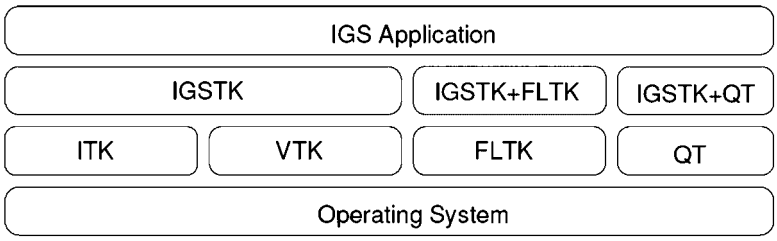
Reasoning:

Encapsulation of toolkit functionalities, preventing developers from directly manipulating objects without passing first safeguards. Here there is a choice between flexibility and managing safety. By using safe encapsulation to restrict functionality the IGSTK can better manage lower level APIs forcing all API calls through safe checks managed via tactics such as a state machine implementation.

By implementing an architecture that depends on other toolkits/APIs not maintained by IGSTK developers, the reliability of the IGSTK is limited to the APIs that it depends on. While safety may be managed from the IGSTK layer, reliability can only be maintained to the extent to which it can be tested.

Medium sized objects resulting in reduced functionality again sacrifice flexibility in order to achieve safety. A limited set of function calls allow IGSTK to manage complexity that could threaten safety.

Architectural Diagram:



Scenario: **S2 Reduce the IGS Application's ability to miss potentially harmful errors**.

Attribute: Safety

Environment: Normal Use / Strained Use

Stimulus: An error is thrown by any element of the IGSTK or underlying layer of components.

Response: IGSTK classes will not throw exception in order to curb misuse

Architectural decisions	Risk	Sensitivity	Tradeoff
Limited use case.			T3
State Machine			T4

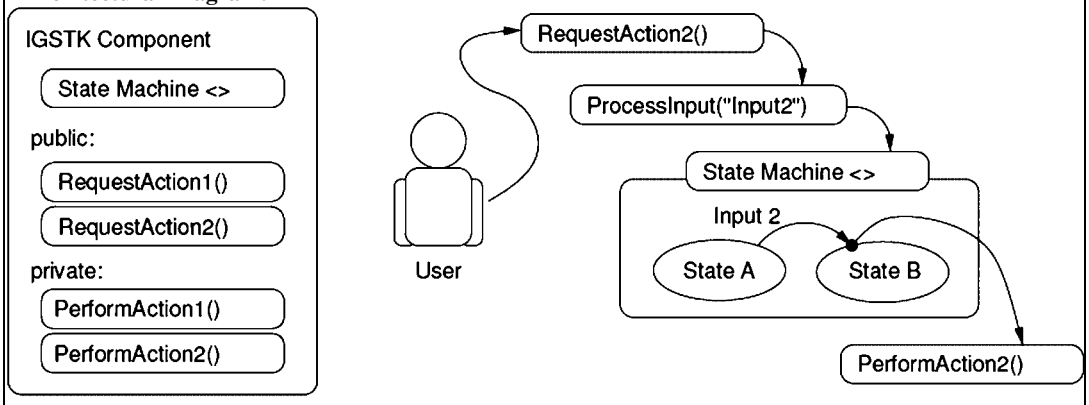
Reasoning:

By limiting the number of choices (functions) available to IGS developers the IGSTK developers can ensure a limited number of use cases that can cover nearly the entire set of possible scenarios for which to test resulting in a high level of code coverage.

DOI: 10.1002/spe

The implementation of a state machine forces all events to traverse through the IGSTK model for handling states, transitions and actions. The developer sacrifices flexibility for the convenience and safety that the IGSTK has to offer.

Architectural Diagram:



Separation Between Public and Private Interface of IGSTK Components

Scenario: **S3 Ensure that the surgical view is up to date**

Attribute: Safety

Environment: Normal use

Stimulus: System may experience stress conditions

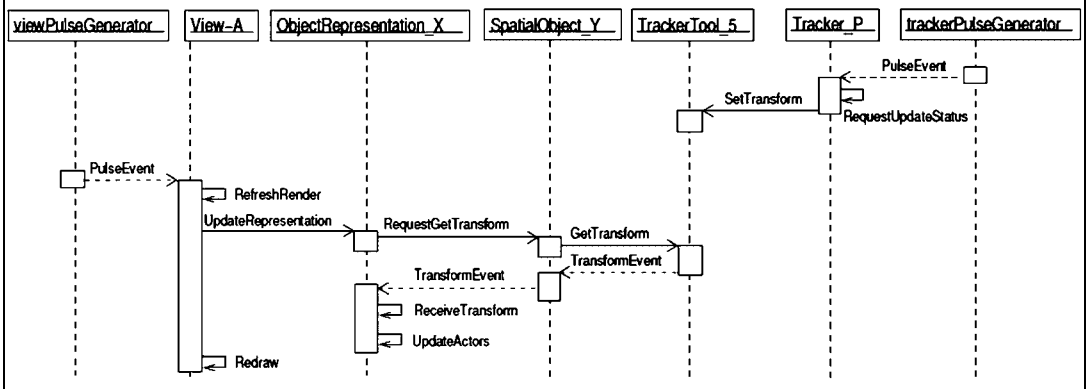
Response: Synchronicity is maintained through an event observer pattern. Through a series of pulses a tracker class will query the actual hardware tracker device and will get from it information about the position of the tracked instruments in the operating room.

Architectural decisions	**Risk**	**Sensitivity**	**Tradeoff**
Event observer pattern to facilitate updated surgical views		S1	

Reasoning:

- Event observer pattern used to track and manage visual and physical representations over time through a series of steady pulses. Expired views are not displayed and visual indicators are displayed. If this management fails serious damage could result from a mismanaged physical and logical views

Architectural



Diagram

UML Sequence Diagram of the IGSTK Timing Collaboration

ACKNOWLEDGEMENTS

This work was funded by NIBIB/NIH grant R01 EB007195. This paper does not necessarily reflect the position or policy of the U.S. Government.

DOI: 10.1002/spe

REFERENCES

1. Beck K, Beedle M, van Bennekum A, Coburn A, Cunningham W, Fowler M, Grenning J, Highsmith J, Hunt A, Jefferies R, Kern, Schwaber K, Sutherland J, Thomas D. Manifesto for Agile Software Development, November 2001. Available at: http://agilemanifesto.org,last [10 December 2010].
2. Gary K, Ibanez L, Aylward S, Gobbi D, Blake MB, Cleary K. IGSTK: An open source software toolkit for image-guided surgery. *IEEE Computer* 2006; **39**(4):46–53.
3. Bowen JP, Stavridou V. Safety-critical systems, formal methods and standards. *IEE/BCS Software Engineering* 1993; **8**(4):189–209.
4. Boehm B. Get ready for agile methods, with care. *IEEE Computer* 2002; **35**(1):64–69.
5. Leveson NG. Software safety: Why, what, and how. *ACM Computing Surveys* (*CSUR*) 1986; **18**(2):125–163.
6. Parnas D, van Schouwen AJ, Kwan SP. Evaluation of safety-critical software. *Communications of the ACM* 1990; **33**(6):636–648.
7. Schwaber, K, Beedle M. *Agile Software Development with Scrum* (1st edn). Prentice Hall PTR: Upper Saddle River, NJ, 2001.
8. Cockburn A. Characterizing people as non-linear, first-order components in software development. *Fourth International Multi-Conference on Systems*, *Cybernetics and Informatics*, Orlando, FL, 2000.
9. Lindvall M, Basili V, Boehm B, Costa P, Dangle K, Shull F, Tesoriero R, Williams L, Zelkowitz M. Empirical findings in agile methods. *Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods—Xp/Agile Universe 2002* (*Lecture Notes in Computer Science*, vol. 2418), Wells D, Williams LA (eds.). Springer: London, 2002; 197–207.
10. Gelowitz C, Sloman I, Benedicenti L, Paranjape R. *Real-Time Extreme Programming* (*Lecture Notes in Computer Science*, vol. 2675). Springer: Berlin, 2003.
11. Spence JW. There has to be a better way! [software development]. *Proceedings of the Agile Development Conference, ADC*. IEEE Computer Society: Washington, DC, 2005; 272–278.
12. Van Schooenderwoert N. Embedded extreme programming: An experience report. *Embedded Systems Conference*, Boston, 2004.
13. Van Schoownderwoert N, Morsicato R. Taming the embedded tiger agile test techniques for embedded software. *Agile Development Conference*, Salt Lake City, 2004.
14. Manhart P, Schneider K. Breaking the ice for agile development of embedded software: An industry experience report. *Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society: Washington, DC, 2004; 378–386.
15. Grenning J. Launching extreme programming at a process-intensive company. *IEEE Software* 2001; **18**(6):27–33.
16. Bowers J, May J, Melander E, Baarman M, Ayoob A. Tailoring XP for large system mission critical software development. *Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods—Xp/Agile Universe 2002* (*Lecture Notes in Computer Science*, vol. 2418), Wells D, Williams LA (eds.). Springer: London, 2002; 100–111.
17. Royce WW. Managing the development of large software systems: Concepts and techniques. *Proceedings of IEEE WestCon*, Los Angeles, 1970.
18. Kruchten P. *The Rational Unified Process—An Introduction*. Addison-Wesley: Reading, MA, 2000.
19. Leveson NG, Turner CS. An investigation of the Therac-25 accidents. *IEEE Computer* 1993; **26**(7):18–41.
20. Lutz RR. Software engineering for safety: A roadmap. *Proceedings of the International Conference on the Future of Software Engineering*, Limerick, Ireland, 2000; 213–226.
21. Storey N. *Safety-Critical Computer Systems*. Addison-Wesley Longman: Harlow, England, 1996.
22. Bowen JP. Ethics of safety-critical systems. *Communications of the ACM* 2000; **43**(4):91–97.
23. Broy M, Jonsson B, Katoen JP, Leucker M, Pretschner A. *Model-Based Testing of Reactive Systems* (*Lecture Notes in Computer Science*, vol. 3472). Springer: Berlin, 2005.
24. Cleary K, Cheng P, Enquobahrie A, Yaniv Z. *IGSTK*: *The Book* (2nd edn). The ISIS Center, Georgetown University, 2009.
25. U.S. Food and Drug Administration. *Code of Federal Regulations*, Title 21, Chapter 1, Subchapter H, Part 820 Medical Device Quality System Regulation, 1996.
26. Raheja D. *Assurance Technologies*: *Principles and Practices*. McGraw-Hill: New York, 1991.
27. Gary K, Kokoori S, David B, Otoom M, Cleary K. Architecture validation in open source software. *Proceedings of the Third Workshop on the Role of Software Architecture for Testing and Analysis* (*ROSATEA'07*), Boston, MA, 2007.
28. W3C. State Chart XML (SCXML): State Machine Notation for Control Abstraction. State Chart XML (SCXML): State Machine Notation for Control Abstraction 1.0, 2005. Available at: http://www.w3.org/TR/2005/WD-scxml-20050705 [19 January 2011].
29. Harris IG. Fault models and test generation for hardware–software covalidation. *IEEE Design and Test of Computers* 2003; **20**(4):40–47.
30. Offutt J, Liu S, Abdurazik A, Ammann P. Generating test data from state-based specifications. *The Journal of Software Testing*, *Verification and Reliability* 2003; **13**(1):25–53.
31. Hune TS. Analyzing Real-Time Systems: Theory and Tools. *PhD Dissertation*, Basic Research in Computer Science, University of Aarhus, Denmark, 2001.

DOI: 10.1002/spe

32. Watson AH, McCabe TJ. *Structured Testing*: *a Testing Methodology Using the Cyclomatic Complexity Metric*. *NIST Special Publication 500-235*, 1996.
33. Holzmann G. *The SPIN Model Checker*. Addison-Wesley: Boston, MA, 2003.
34. Bengtsson J, Larsen K, Larsson F, Pettersson P, Yi W. UPPAAL—A tool suite for automatic verification of real-time systems. *Proceedings of Hybrid Systems III*. Springer: Berlin, 1996.
35. Magee J, Kramer J. *Concurrency*: *State Models and Java Programs* (2nd edn). Wiley: Hoboken, 2006.
36. Yaniv Z, Cheng P, Wilson E, Popa T, Lindisch D, Campos-Nanez E, Abeledo H, Watson V, Cleary K. Needle-based interventions with the image-guided surgery toolkit (IGSTK): From phantoms to clinical trials. *IEEE Transactions on Biomedical Engineering* 2010; **57**(4):922–933
37. Kazman R, Klein M, Clements P. ATAM: Method for architecture evaluation. *Technical Report CMU/SEI-2000-TR-004*, (ADA382629), Software Engineering Institute, Carnegie Mellon University, 2000.

DOI: 10.1002/spe