IGSTK: An Open Source Software Platform for Image-Guided Surgery

Kevin Gary[1], M. Brian Blake[2], Luis Ibanez[3], David Gobbi[4], Stephen Aylward[3], and Kevin Cleary[5]

[1] Division of Computing Studies, Arizona State University, Mesa, Arizona, 85212, USA
kgary@asu.edu
[2] Department of Computer Science, Georgetown University, Washington, DC, 20007, USA
blakeb@cs.georgetown.edu
[3] Kitware Inc., Clifton Park, NY, 12065, USA
{luis.ibanez,stephen.aylward}@kitware.com
[4] Atamai Inc., London, Ontario, N6B 2R4, Canada
dgobbi@atamai.com
[5] Imaging Science and Information Systems (ISIS) Center, Department of Radiology,
Georgetown University Medical Center, Washington, DC, 20007, USA
cleary@georgetown.edu

The Image-Guided Software Toolkit (IGSTK: pronounced IGstick) is an open source project supporting the development of reliable software for patient-critical image-guided surgical applications. In its broadest sense, image-guided surgery involves the use of medical images for instrument guidance and for clinical decision making during minimally invasive procedures. Specifically, image-guided surgery systems, also known as computer-aided surgery systems, have been developed to provide physicians with a virtual, real-time display of the anatomy located in the region of a surgical instrument [1]. The typical display is composed of a combination of anatomical images acquired before the surgical intervention and graphical representations of the surgical instruments; it also may include intra-operative image updates of the anatomy. Image-guided surgical procedures mean substantially less trauma for the patient than open surgical procedures. Commercial image-guided surgery systems are now available for the brain, the spine, and other applications.

To display the location of surgical instruments with respect to the patient's anatomy, the relative location of the instrument and the patient is measured using devices known as trackers. The display then can help guide the physician to the precise anatomical target and can also provide the physician with "x-ray vision," in that the physician can see what lies beneath a surgical instrument before moving the instrument. A typical image-guided surgery system for brain surgery is shown in Figure 1.

Figure 1. Image-guided system for brain surgery showing the optical tracking system at the top right and the display overlay on the far left. The patient is under the blue cover in the middle. (Photograph courtesy of Richard Bucholz, MD, St. Louis University).

To define the clinical and technical requirements for the image-guided surgery toolkit IGSTK, we analyzed several specific surgical procedures. One procedure we analyzed was radiofrequency ablation (RFA) of liver lesions, a minimally invasive cancer treatment typically performed by interventional radiologists. In this procedure, the lesion is first located using CT and/or ultrasound imaging. A RFA probe (needle) is then advanced through the skin and into the lesion. Once the probe is positioned, it is connected to a RF generator, and RF energy is sent through the probe and into the tumor. Temperatures of up to 100° C lasting 10-30 minutes are obtained to ensure cell death. Burns as large as 7 cm can be obtained and this size burn allows the treatment of a 5 cm spherical lesion with a 1 cm "negative margin" burn around the lesion. Because of the desired 1 cm safety margin around each lesion, the radiologist must position the RFA probe within 5 mm of its intended location.

Each surgical procedure we analyzed was then compared to determine both unique and common components. For example, in the RFA application, the need for a 3D ultrasound image composition component was identified. This component provides 3D tracking of a 2D ultrasound probe so that 2D ultrasound images can be organized into a 3D volume. This component addresses the challenging task of reconstructing a valid liver image despite liver movement due to respiration, cardiac motion, and probe pressure during the acquisition of a sequence of 2D ultrasound images. In another component, image-based registration is used to account for the internal movement of organs. This component involves tracking the ultrasound probe and patient in order to initialize the registration of the intra-operative 3D ultrasound data and the pre-operative computed tomography (CT) or magnetic resonance imaging (MRI) data. The image-based registration component then refines that initial registration to account for the intra-operative displacement of a designated organ. Using these components, a tracked needle is overlaid onto fused ultrasound/CT/MRI data and used intra-operatively to guide the procedure. In Figure 2, the image on the left illustrates the fusion of a liver lesion from CT data with segmented vessels and data from an intra-operative 3D ultrasound scan. The image on the right illustrates results from the registration metric and shows one slice of co-registered MRI and 3D ultrasound liver images.

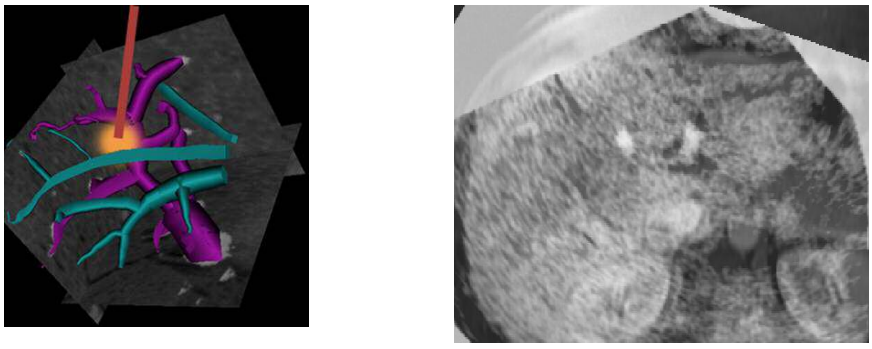

Figure 2. Fused 3D US and CT data with segmented vessels (green and purple), tumor (orange), and probe in red (left side);one slice of fused US and MRI liver data (right side).

Based on these component-level analyses, the structure of a typical image-guided surgery system was identified as a complex combination of three major technologies: 1) a control computer; 2) software for image processing, control, and the user interface; and 3) a tracker for localizing instruments and the patient in three-dimensional space. The following is a typical sequence of steps performed when using an image-guided surgery system:

1. A pre-operative CT or MRI scan is obtained. Fiducials (small markers) are usually placed on the anatomy prior to this scan for registration in Step 4.
2. The CT or MRI images are imported into the computer. A reference target is typically attached to the anatomy to compensate for any inadvertent motion of the camera or patient. This technique is known as dynamic referencing.

3. Registration is the next and most important step in image-guided surgery. It is done by identifying the fiducials in the image and the tracking space, and by computing a transformation matrix between these two coordinate systems.
4. The system can now track surgical instruments, including probes or pointers, and can display the anatomy beneath these instruments. A four-quadrant view (axial, sagittal, coronal, and 3D images) is typical, but many variations exist; for example, oblique reformatting of the images at any angle can provide better image guidance.

A Need for Safe, Open-Source Software

Of the above listed technologies, software is the most critical in developing an image-guided system. The software must integrate information from tracking systems, correlate this information with the relevant images, and display real-time updates of the instruments and patient anatomy. Moreover, the software is intended for use in life-critical applications, so it must be carefully designed and managed to ensure ease of use, robustness, and stability. Because of these stringent requirements, image-guided surgery systems have tightly coupled components, involve complex mathematics, and have high synchronization constraints.

Traditional software engineering techniques do not provide cost-effective, non-intrusive methods for validating such systems for clinical use. Open-source software has the potential to meet these needs; however, it remains largely unexplored because many academic research groups and small businesses do not have the infrastructure to carefully design and test robust software. These considerations led us to form a multidisciplinary team of software engineers and medical imaging scientists to develop the image-guided software toolkit, IGSTK. The toolkit contains the basic software components to create an image-guided system, including a component for controlling the tracker, as well as a display component for providing image overlay of the patient anatomy and the surgical instruments. In the remainder of the paper, we describe the software engineering principles that we applied as we developed the architecture and the processes that drive the IGSTK.

IGSTK Architecture

IGSTK supports safety-critical surgical applications in which software errors may lead to catastrophic results. IGSTK minimizes the risk of patient harm that may result from negligent misuse of the framework. This goal of *safety-by-design* is achieved by adherence to the following principles:

1. IGSTK uses a component-based layered architecture style. Every component has a well-defined set of features governed by a state machine. Strongly-typed interfaces provide enforceable interaction contracts between components.

2. The state of a component is explicit and always known, and all transitions are valid and meaningful. A component's state machine is encapsulated, meaning clients of the component may not manipulate state outside the contract specified by the component interface.

State Machines

A state machine is defined by a set of states, a set of inputs, and a set of directed transitions between states. Transitions in a state machine change the current state of the state machine in response to some stimulus, or input. A behavior may execute during a transition, on entry to a state, or on exit from a state. A full treatment of the formal semantics of state machines can be found in the literature [2, 3]. It is worth noting that the presence of formal semantics and a wide field of study on state machines influenced our decision to go with this trusted architectural pattern. Figure 3 shows the state machine implementation for a Spatial Object component (the IGSTK components will be described later in this section).

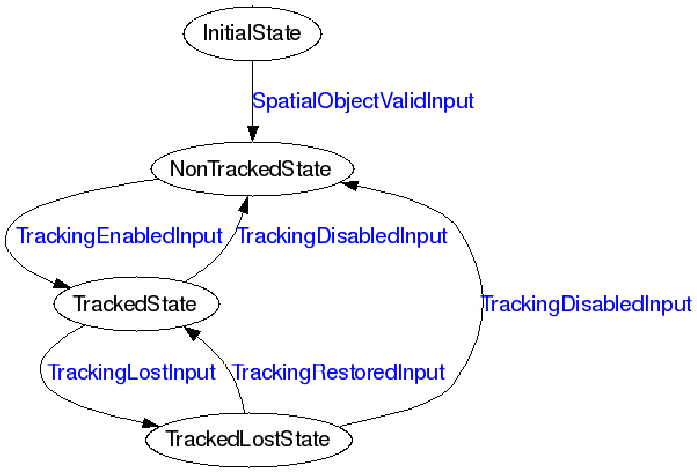

Figure 3. State machine for an IGSTK Spatial Object showing the four states in black and the transitions in blue.

Component-based architectures are naturally described in terms of state machines, yet few implementations make use of explicit state machines to represent component state. Such relaxed programming practices lead to components with under-defined and poorly understood states, and thus may cause the components to behave unpredictably, which is unacceptable in image-guided surgery. The need for reliability and robustness in IGSTK led us to select the state machine model in the very early stages of the project. State machines ensure that a component is always in a valid state which can guarantee repeatable and deterministic behavior.

State machines in IGSTK provide:

- *Safety and Reliability*: A state machine ensures that component behavior is deterministic and that all components are in a known and error-free state at any given moment.
- *Cleaner design*: Since developers must anticipate all possible inputs, states, and transitions, the state machine encourages and enforces a cleaner and more robust design, free of untested assumptions.
- *API simplicity:* A focused, clearly expressed application programming interface (API) is a must for supporting robustness and reliability. In the context of surgical guidance, we believe that *flexibility* and *abundance of features* are undesirable, because they create more opportunities for things to go wrong during a surgical intervention.
- *A consistent integration pattern*: The toolkit's value as it matures will undoubtedly be tied to the incorporation of additional functionality at the component level. This functionality will often take the form of reusable code from existing toolkits. State machines provide a consistent pattern for integrating this functionality while adhering to the safety-first principles necessitated by the application domain.
- *Quality Control*: State machines facilitate code coverage in the sense of lines of code tested, as well as path coverage on a per component basis. Using code that is not based on state machines may result in applications that exhibit unreliable behavior. At run time, they can easily enter into any number of untold states that were never explored by the developer, and can lead to error conditions that may or may not become visible to the users of the application.

The safety emphasis in IGSTK encourages explicit knowledge of component state and determines whether a given behavior may be executed while in that state. Hence, our application is a combination of traditional concepts in reliability (being in a known state) and the State Pattern [4] (for managing availability of behavior dynamically).

IGSTK Components

Component-based computing is a prevalent model in modern computing architectures. Component boundaries within IGSTK are governed by strict contract-based interfaces. Component realizations encapsulated behind these boundaries allow IGSTK to manage the complexity required in a safety-critical domain. We take advantage of components in the following ways:

- Interaction patterns between components are easily visualized.
- Rigid component boundaries with well-understood interaction patterns allow for rigorous testing, creating "safe zones" at the component level.
- The framework is extensible in a structured way. Specialized implementations of behaviors are mapped to concrete realizations of base interfaces. Together with the State Machine, this feature ensures that component behavior is, in a sense, bounded.
- Component implementations that are based on third-party code are integrated into IGSTK in a safe manner. For example, IGSTK extends facilities provided by the Insight Segmentation and Registration Toolkit ITK (http://www.itk.org) and the Visualization Toolkit VTK (http://www.vtk.org), but does so in a safe, predictable way.
- From a development process perspective, fewer team members need to be concerned about component interactions; they can instead focus their concerns on the capabilities of individual components.

IGSTK's component architecture does not come without costs. A principal concern is possible performance penalties incurred due to extra method invocations since we are wrapping third-party interfaces to achieve safety, as well as overhead associated with dynamic binding in a polymorphic language such as C++. A second concern is the complexity incurred in managing decoupled components. As with any object-oriented software system that relies on encapsulation, specialization, and loose coupling, abstraction layers obscure linear views of realized functionality, making it difficult for developers to unwind component interactions to troubleshoot issues during development.

Despite these concerns, we decided to leverage a component-based architecture in order to meet our high-priority safety requirements. We briefly describe the four key component types in this section and show an architecture diagram in Figure 4.

Tracker

A tracking device has the capability to provide position, orientation, and other relevant information pertaining to tracked surgical tools. The IGTSK Tracker component, which directly incorporates a state machine into code donated by Atamai Inc., provides an object-oriented representation of tracking devices, tracked tools, and all pertinent static and dynamic information associated with tracking.

The internals of the Tracker require a separate thread to communicate with tracking hardware at an acceptable frequency. The Tracker component manages this communication and updates internal state information at this frequency rate. Spatial Objects then pull information about their corresponding tools from the Tracker at the desired rendering rate.

Spatial Objects

Spatial Objects are central to maintaining the types and physical locations of objects in a surgical environment. They are used to render a surgical scene. This layer ties together rendering functionality in the Viewer and Representation layers with Tracker functionality. IGSTK Spatial Object classes encapsulate ITK Spatial Objects inside a restricted API subject to the control of a state machine. In this way, the Spatial Object class retains all the functionality without the risk associated with exposing all the flexibility of ITK classes.

Spatial Object Representation

It is convenient to separate the geometrical description of a surgical instrument from its graphical representation in a display. Spatial Objects are intended to provide a geometrical description, while Spatial Object Representations specify how those same objects should appear on the screen. Each Spatial Object Representation is responsible for displaying its associated Spatial Objects within a specific context, which is typically either a 3D view of the patient anatomy or a 2D tomographic or projective image through the patient anatomy.

Viewers

Viewers present renderings of surgical scenes to the clinician. Viewers are critical since they are the main source of information presented to the clinician. Viewers are built using VTK classes encapsulated into a restrictive API and are subject to the control of a state machine. Viewers limit the number of ways in which a user can interact with the scene. These interactions are defined under the assumption that a surgeon may have access to only a touch screen during the procedure as a keyboard or mouse are not convenient input devices in the surgical environment.

As shown in Figure 4, the end user interacts with Viewers presented in an application interface. Viewers aggregate Spatial Object Representations, which in turn derive their geometric information from the Spatial Objects. Spatial Objects query Trackers to determine current position and orientation. To synchronize scene generation and provide an acceptable rendering frequency, IGSTK uses a PulseGenerator object to generate "ticks," and a real-time clock mechanism to check timing intervals for validity of scene information. The current IGSTK implementation runs on common operating platforms such as Linux, MacOS, and Windows. Our continuing work aims to extend IGSTK for real-time operating system services.

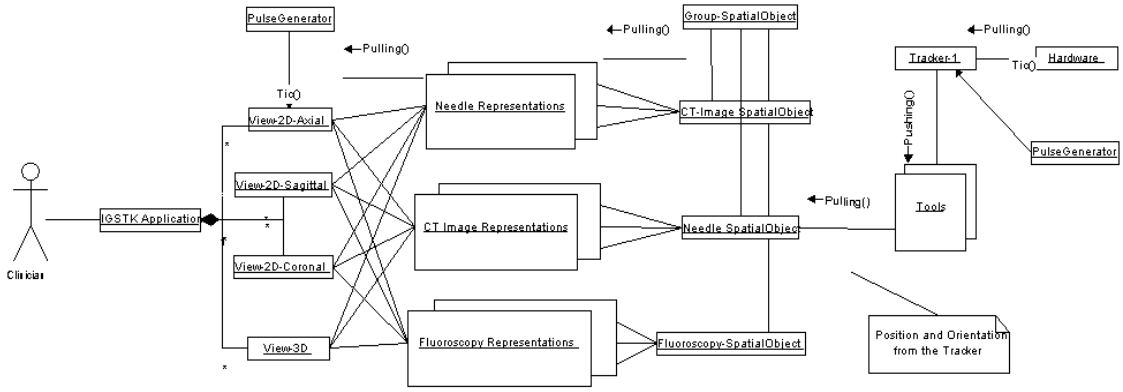

Figure 4. IGSTK Component Architecture.

Component Interaction and State Machines

Using Figure 4 as a reference, component interactions in IGSTK are such that components on the left of the figure request information or services from components on their right; components on the right move information to the left via events. A lightweight event model, an implementation of the Observer Pattern [4], ensures information passed right-to-left is moderated at each layer by a state machine. Error-prone if/else conditions on return values are

avoided. Instead, events are processed via a translation step to a state machine input. Again, in keeping with the principle of safety-by-design, the interaction in both directions happens under the purview of state machines. To understand why this feature is important, consider the rudimentary example diagrammed in Figure 5.

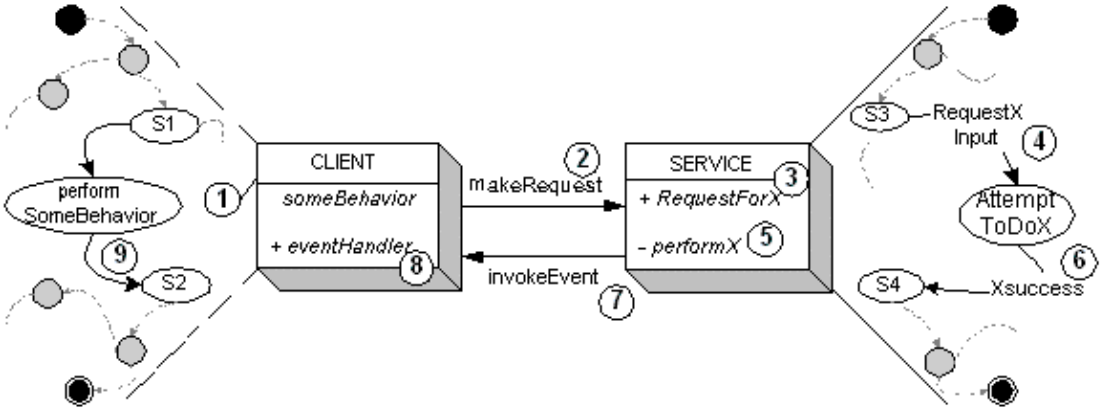

Figure 5. Example state machine managed interaction between components.

In this example, we term the component requesting service the CLIENT and the called component SERVICE. The relevant part of each component's private state machine is shown next to the component. The interaction sequence is indicated by numbered circles. Initially, CLIENT is performing some behavior while in some well-known state of its state machine (1). CLIENT then makes a request for service (2) via a public method invocation on SERVICE. SERVICE accepts the request (3), and translates it to an input to its state machine (4). Based on the transition, a private method invocation on component SERVICE may also occur (5), representing the execution of the desired computation. Upon successful completion of the computation (6), SERVICE generates an event (7) containing the requested information and dispatches the event to CLIENT (8). CLIENT translates the event into an input on its state machine (9), thereby avoiding error-prone conditional logic on return values.

While this sequence may at first seem awkward to the conventional programmer compared to simply calling a method and checking its return value, it has several salient features that robustly enforce safety. When a component makes a request, it first transitions to a state indicating it is busy processing. By not performing a computation as an action associated with a transition, the state machine avoids situations in which a component is "between" known states (IGSTK state machines are Moore machines) [2]. Second, IGSTK state machines use fully populated state transition tables, meaning no matter what input is given, a state machine will take a known and verified transition. Third, public request methods delegating to private behaviors mitigated by a component's state machine protect the execution of the behavior. In other words, the behavior only gets executed when it is certain that it is safe to do so. Finally, returning information via events that are translated to state machine inputs avoids error-prone conditional logic. The logic is embedded in a component's state machine, and can be visualized and verified explicitly.

IGSTK Software Process

Open source is an increasingly popular model for software development and delivery. Many open source projects rely on skilled development teams whose members are distributed throughout the world. Often, these teams use Agile methods to facilitate an evolutionary style of development. IGSTK deals with unique complexities deriving from the nature of the requirements, the makeup of the team, the dependence on pre-existing software packages, and the need for high quality standards for surgical applications. In this section, we discuss these complexities and suggest requirements techniques and quality management best practices that augment typical Agile methods to provide a robust process.

IGSTK Best Practices

IGSTK Quality Management processes focus on the correctness of the software. *Correctness* to us means that the software fulfills all of, and only, the requirements in an error-free manner, including all nonfunctional requirements, especially the safety requirements for the framework. This definition of correctness includes *completeness*, intentionally sounding like a traditional definition of Verification and Validation.

Given the distributed nature of the development team, the complexity of the algorithms implemented, and restricted access to subject matter experts, we decided early in the project that adopting a "heavy" software process was not an option. Instead, IGSTK supports correctness via a collection of Best Practices woven into an Agile process framework. Our "Top 10 Best Practices for Safety-Critical Development with Agile Methods" are:

Best Practice #1. Recognize that people are the most important mechanism available for ensuring high quality software. The IGSTK team is comprised of developers with a high degree of expertise with the application domain, supporting software, and tools. Their collective judgment is weighted over process mandates.

Best Practice #2. Facilitate constant communication. To prevent over-isolation of a distributed team working on decoupled components, IGSTK facilitates constant communication. IGSTK members participate in a weekly teleconference and meet in person twice per year. IGSTK employs a mailing list, instant messaging, and a Wiki for online collaboration.

Best Practice #3. Produce iterative releases. IGSTK's external release cycle includes twice-yearly releases. Internally, six months was considered too long a horizon to manage development, so releases are broken down into approximately two month "sprints" called iterations. At the end of an iteration, development stops, code and other quality reviews are performed, and code that is considered stable is moved to the main code repository.

Best Practice #4. Manage source code from a quality perspective. IGSTK applies different configuration management policies that satisfy different quality criteria. Separate codelines with separate policies allow developers to collaborate on code that may not yet meet stringent quality criteria. Exploiting configuration management approaches early in development helps document and track project quality progress.

Best Practice #5. Focus on 100% code and path coverage at the component level. Unit tests are required to ensure 100% code coverage across all platforms. We are also developing a customized visualization and validation tool for IGSTK state machines, to ensure that all paths within all state machines are executed at least once. Additionally, dynamic analysis tools ensure there are no memory leaks or access violations.

Best Practice #6. Emphasize continuous builds and testing. IGSTK uses the open source DART tool (http://public.kitware.com/Dart/HTML/Index.shtml) to produce a nightly dashboard of build and unit test results across all supported platforms.

Best Practice #7. Support the process with robust tools. Best Practice #6 describes the use of the DART tool for continuous testing. IGSTK also employs an open source cross-platform build solution called CMake (http://www.cmake.org/) and an open source documentation system called Doxygen (http://www.stack.nl/~dimitri/doxygen/). These tools are augmented with defined best practices for coding and documentation as posted on the Wiki.

Best Practice #8. Manage requirements iteratively in lockstep with code management. As requirements evolve and the code matures, it is necessary to adopt flexible yet defined processes for managing requirements. Requirements management is a complex process for a project such as IGSTK, and thus is discussed in detail in the next section.

Best Practice #9. Focus on meeting exactly the current set of requirements. Traceability implies heavy process structures – large documents and invasive tools. Traceability is needed in safety-critical domains, and particularly in surgical applications that need FDA approval. IGSTK addresses this problem with continuous requirements review (practice #6), lightweight tools (practice #7), and codeline policies (practice #4).

Best Practice #10. Evolve the process. Through constant communication, IGSTK members recognize when the complexities they face can be approached within the current process framework, when "tweaks" are required, or when entirely new practices should be adopted.

These Best Practices encompass the "lightweight" approach to robust software development in IGSTK. We do not equate "lightweight" with process "ignorance"; we readily recognize that a collection of good practices applied without structure will not lead to software quality. These Best Practices include process structures – namely people, managed requirements, continuous testing, and iterative development. Nor does "lightweight" mean "optional" process execution; IGSTK developers are not free to simply ignore Best Practices when convenient.

Lightweight Requirements Management Process

Significant expertise is needed to develop effective image-guided surgical applications. Development processes require a close connection between subject matter experts (SMEs) and software engineers. Waterfall and spiral development approaches tend to incorporate "top-down" processes that assume that *complete* requirements can be defined early in development. In this domain, we discovered that development processes must tightly integrate iterative involvement of SMEs. In our work, we introduce a customized extension to Agile development methods through the introduction of an Agile requirements management process.

Agile Requirements Management Process

The process for requirements management is significantly integrated with code management. The first step is to categorize new requirements as *architecture requirements*, *style guidelines requirements*, or *application requirements*. These classifications were derived from the authors' earlier work on Component-Based Product Line Analysis and Development (C-PLAD) [5]. Developers introduce new architecture and style requirements as components are developed. Application requirements are evolved through interaction with clinicians, as described below. The IGSTK project employs a collaborative process for reviewing, implementing, validating, and archiving these requirements, integrated with application development. This process is illustrated as a UML state diagram in Figure 6.

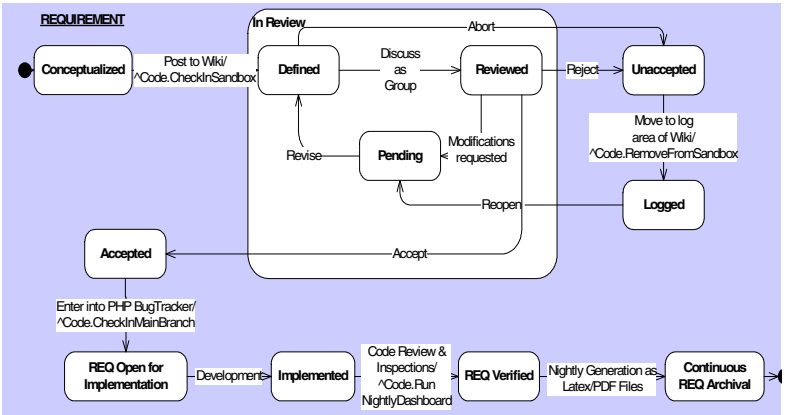

Figure 6. IGSTK Requirements Management Process as a UML state diagram.

As shown in Figure 6, once a developer identifies new potential requirements, s/he posts a description on the Wiki (Defined). At the same time, initial code that fulfills the requirements may be entered into a sandbox repository. The requirement then undergoes an iterative process in which team members review, discuss, and potentially modify the requirement. Based on the team's decision, requirements are rejected or accepted. Accepted requirements are entered into an online database and marked as "open." Once the supporting software is implemented and its functionality is confirmed, the requirement is marked as 'verified." As nightly builds takes place, all verified requirements are automatically archived.

Conceptualizing Requirements via Collaborative Modeling

The disparity of knowledge between software engineers and SMEs creates a barrier to conceptualizing requirements. These two groups speak in totally different terms. In IGSTK, the best way for software engineers to collaborate with SMEs is by documenting application flow using UML activity diagrams. Figure 7 is a UML diagram of the liver lesion application described at the beginning of this article. We created several such diagrams during the initial application requirements gathering phase to facilitate the identification of the required functionality for IGSTK components.

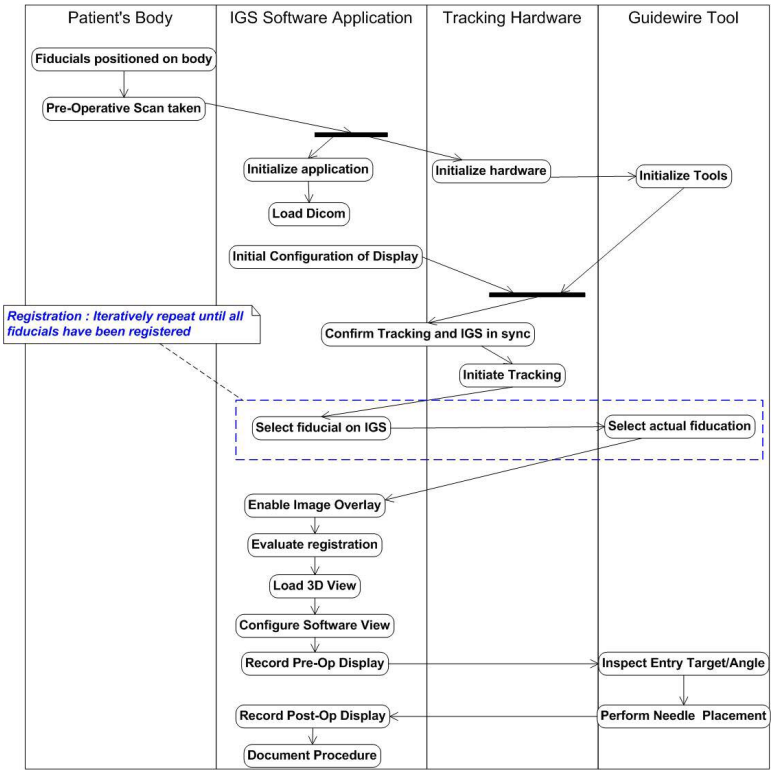

Figure 7. UML Activity Diagram Illustrating Liver Lesion Ablation Scenario.

Developing Requirements for Safety-Critical Software

Safety-critical applications demand that clinicians understand the functionality of each level of the underlying software that they use. Defining requirements at each level leverages the Agile development approach to creating robust software. In addition, activity diagrams provide a vehicle for visual communication between software engineers and clinicians. Requirements engineering approaches within IGSTK concentrate on this tight interaction

between medical specialists and software engineers and on the close traceability between requirements and software. This combination of requirements engineering approaches and techniques assures a major criterion for safety: that the software being created is well-defined, managed, and understood by the people who will ultimately use it.

Summary

Image-guided surgery applies leading-edge technology and clinical practices to provide better quality of life to patients who can benefit from minimally invasive procedures. Reliable software is a critical component of image-guided surgical applications. Costly expertise and technology infrastructure barriers hamper current research and commercialization efforts in this area. The image-guided surgical toolkit IGSTK applies the new open source development and delivery model to this problem. Agile and component-based software engineering principles reduce the costs and risks associated with adopting this new technology, resulting in a safe and reusable software infrastructure. While the project is still ongoing, the initial version of the toolkit has been completed, and further details can be found at http://www.igstk.org/.

Acknowledgements

This research is supported by the National Institute of Biomedical Imaging and Bioengineering (NIBIB) at the National Institutes of Health (NIH) under grant R42EB000374 and by U.S. Army grant W81XWH-04-1-0078. The content of this manuscript does not necessarily reflect the position or policy of the U.S. Government. The authors would like to thank the IGSTK project team including Rick Avila, Patrick Cheng, Andinet Enquobahrie, Julien Jommier, Hee-su Kim, Sohan Ranjan, and James Zhang. The authors would also like to thank the IGSTK advisory board for their advice throughout the project: Will Schroeder of Kitware; Ivo Wolf of the University of Heidelberg; Peter Kazanzides and Anton DeGuet from John Hopkins University; and Ingmar Bitter, Matt McAuliffee, and Terry Yoo from the NIH.

References

1. R.L. Galloway, "The Process and Development of Image-Guided Surgical Procedures," *Annual Reviews of Biomedical Engineering*, 2001, Volume 3, pp. 83-108.

2. J.E. Hopcroft, et al., Introduction to Automata Theory, Languages, and Computation, Addison Wesley (2nd ed.), 2000.

3. A.M.K. Cheng, Real-Time Systems: Scheduling, Analysis, and Verification, Wiley-Interscience 2002.

4. E.Gamma, et al., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

5. M.B. Blake, et al., " Use Case-Driven Component Specification: A Medical Applications Perspective to Product Line Development", *Proceedings of the ACM Symposium on Applied Computing*, Santa Fe, March 2005, pp. 1470-1477.

Authors

Kevin Gary is an Assistant Professor in the Division of Computing Studies at Arizona State University's Polytechnic and is an IEEE member. His research focus is software engineering, in particular software process. He earned his Ph.D. in Computer Science from Arizona State University in 1999.

M. Brian Blake is an Associate Professor in the Department of Computer Science at Georgetown University and is a Senior Member of the IEEE. His research focuses on service-oriented computing, component-based software

engineering, and workflow modeling. He earned his Ph.D. in Information and Software Engineering from George Mason University in 2000.

Luis Ibanez is a Senior Research Engineer with Kitware Inc. His main interest is the promotion of Open Access publications and open source software for medical image analysis. He earned is Ph.D. from Rennes University, France in 2000.

David Gobbi is the Vice President, CEO, and lead software designer of Atamai Inc., a company that develops image analysis, visualization, and control software for medical research applications. He earned his Ph.D. in Medical Biophysics from the University of Western Ontario, Canada, in 2003.

Stephen Aylward is Chief Medical Scientist at Kitware Inc. and is an Associate Member of IEEE. He is also the president of the Insight Software Consortium, a non-profit consortium that promotes open-source software for medical image analysis. His research includes model-based image registration and vascular network segmentation and characterization. He earned his Ph.D. in Computer Science from the University of North Carolina at Chapel Hill in 1997.

Kevin Cleary is an Associate Professor in the Department of Radiology's ISIS Center at Georgetown University Medical Center and is an IEEE member. His research focuses on image-guided surgery and medical robotics. He earned his Ph.D. in Mechanical Engineering from the University of Texas in 1990.

(end of document)