An Architectural Investigation of the Open Source Image-Guided Surgery Toolkit (IGSTK) Project

John F. Heidenreich
Arizona State University, East
John.Heidenreich@asu.edu

Abstract— Software evaluation plays a critical role in the realization of a successful software project. While this role is typically performed by the organization building and distributing the software, an open source development paradigm, one where project artifacts are made public, makes it possible to evaluate project data that otherwise might not be available. This project will evaluate the open source Image Guided Surgery Toolkit (IGSTK) project in order to demonstrate how public data can aid in the evaluation of the architectural and evolutionary aspects of an open source project and how these two aspects are related. First, a short review of historical public data in the form of CVS logs will take place in order to help understand its significance in terms of the evolution of the architecture of the system. This exercise represents an evaluation of evolutionary aspects of the system. Second, an architectural review will be performed using a quasi-form of the Architectural Tradeoff Analysis Method (ATAM) developed by the Software Engineering Institute (SEI) at Carnegie Mellon. This second architectural evaluation will draw from documentation made public by the project contributors

I. Introduction

This project will use the IGSTK project in order to perform a software evaluation. This study relies on publicly available software artifacts that can be valuable tools in evaluating software projects. The availability of these tools is facilitated by an open source development model, where project artifacts are made public. CVS log data, as will be shown, can help to demonstrate how the software project has grown and evolved over time. While the growth and evolution of a system is not typically associated with software architecture, it is a critical component in understanding how the architecture was implemented. It can address the process by which the architecture was realized [2]. The majority of the work of this project represents an architectural review of the IGSTK project. This architectural evaluation will use the ATAM model developed by SEI, drawing from documentation made public by the project contributors.

II. Problem Statement

Software architecture underscores the importance of modeling and planning high level design in order to realize the organization of a software system that effectively achieves the quality requirements necessary for it to be deemed successful. With this in mind it becomes important for software project contributors as they participate in the software process to understand and conceptualize the architecture. In addition the ability to effectively evaluate a software project in terms of its growth as well as its architecture can perform a critical role in its acceptance by the software community [1]. Open source software artifacts, which are publicly available, offer tremendous value in terms of their ability to evaluate how the software was written [1].

III. Methodology & Results

A. IGSTK Historical Analysis Data Retrieval Methods

An evolutionary analysis helps to answer key question as to a software project. Was it written by many or a few? Were releases rolled out more or, less frequently? How steady was the growth of the code? All of the project evolution data retrieved for the IGSTK project indicates a growth pattern that follows an agile approach.

For this analysis several data extraction and visualization tools are used. It can be helpful to use source control management tools such as CVS and SVN, as empirical data gathering tools for open source software evaluation. Table I lists a few of the tools that were used to mine data in this study.

TABLE I
CVS Analysis TOols

CVSGrab	CVSGrab is both a data mining and data visualization tool that allows a researcher to quickly make assessments about the nature of a project that would have otherwise taken a researcher a great deal of time meeting with project participants. It visualizes the evolution of software projects, showing data all the way down to the file level. It allows for the correlation of data based on activity and contributors [3].
StatCVS	StatCVS provides basic usage reports. Data is displayed in easy to read graphs. The CVS data is summed up and displayed in a format that is more cognitively understandable,

	through visual representation [4].
CVSGraph	This tool is a graphical representation of all the branches and revisions in a CVS repository [5].

B. Historical Analysis Results

The tools that were used to analyze the IGSTK project consistently resulted in some key observation. As shown in figure 1. The figure below shows that the IGSTK project has resulted in a fairly steady rate of continuous development. The number of lines of source code has grown at a sub-linear rate spanning across frequent public releases and sandbox releases. This pattern demonstrates that even though development is ongoing, it is being performed in an incremental fashion that allows for frequent testing.

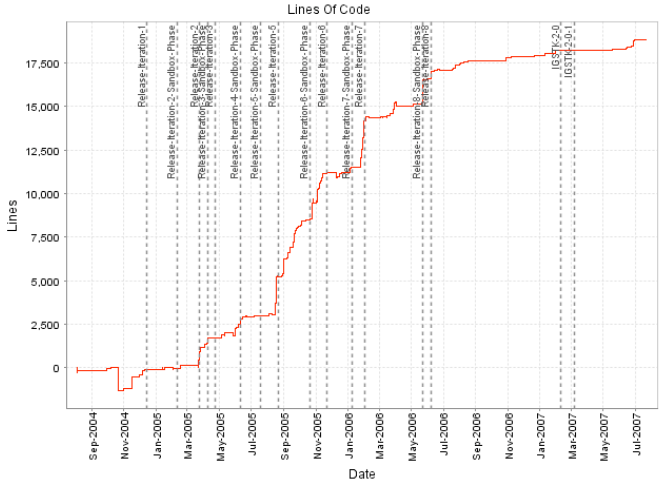


Fig 1. Lines of Code over time and across releases – StatCVS-XML

The frequent number of commits represented by thin lines in figure 2 represents an active group of developers who are working at an active rate. The frequent rate of released working versions, with at least six iterations in a given year can be characterized as rapid development. It is common for software projects to release only one or two version releases each year. This demonstrates that there is a steady rate of development release iterations being made public.

Tag	Files	Lines of Code
Release-Iteration-1	38	-89
Release-Iteration-2-Sandbox-Phase	41	-34
Release-Iteration-2	72	906
Release-Iteration-3-Sandbox-Phase	85	1707
Release-Iteration-3	85	1698
Release-Iteration-4-Sandbox-Phase	82	2765
Release-Iteration-5-Sandbox-Phase	83	2989
Release-Iteration-5	127	5238
Release-Iteration-6-Sandbox-Phase	142	8521
Release-Iteration-6	217	11195
Release-Iteration-7-Sandbox-Phase	223	11495
Release-Iteration-7	272	14156
Release-Iteration-8-Sandbox-Phase	280	16528
Release-Iteration-8	348	16987
IGSTK-2-0	360	18246
IGSTK-2-0-1	360	18250

Fig. 3. New Source files over version releases – CVS-Graph

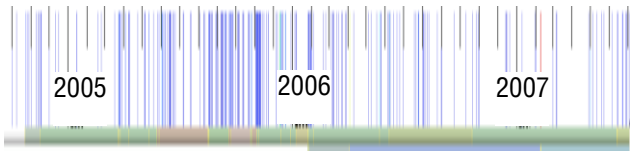


Fig 2. Blue lines represent commits over, time indicating an active developer base. – CVSGrab

Figure 3 demonstrates that a substantial number of new source files and new lines of code were not added until later in the life of the project. This is a sign of the willingness of developers to incorporate changes late into the development process. Previous releases indicate a working version.

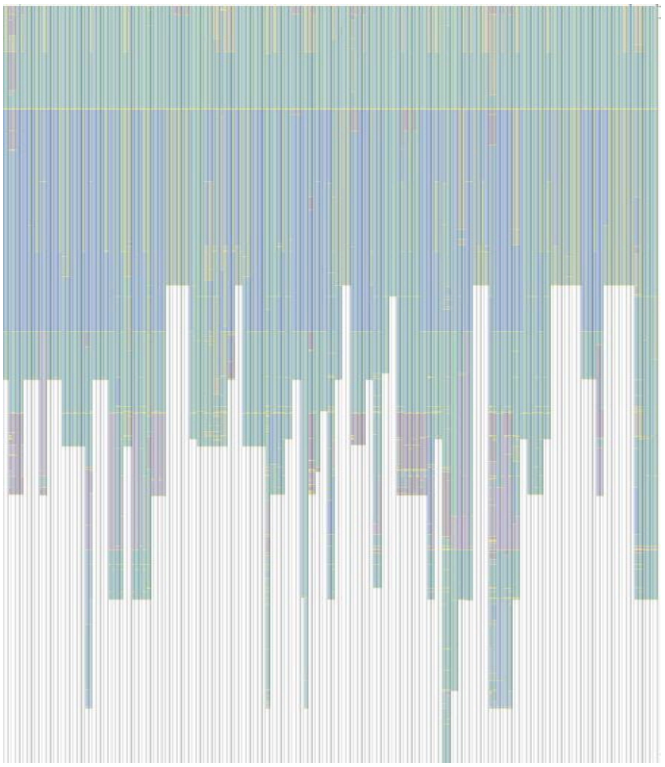


Fig 4. Files modified by developers – CVSGrab. Bars represent code files over time. The changing colors represent changing ownership of files by different developers. Such a breadth of color across virtually all files represents an understandable code base that is understood by a variety of project developers who have maintained a consistent level of self organization over time.

It can be gathered that the rate of developer participation tends to spread throughout the life of the project indicating that the number of committed developers remains fairly constant. It is common with open source projects for the majority of the work to shift to a smaller group of developers rather than a consistent distribution between participants over time. As shown in Figure 4, coding responsibilities seem to shift from one developer to another quite readily. Different colors are associated with distinct contributors. This would suggest a degree of simplicity or understandability that would allow a straightforward transfer of development responsibilities from one file to the next. Likely, this is facilitated through appropriate documentation or adherence to common coding

styles. The frequency of file ownership changes also suggests that developers are being assigned to different areas of work with a constant frequency. Such a high change in developer responsibility seems to be intentional as it appears that other developers check out files that had been previously been checked out by other developers. This suggests a degree of strong self organization.

Upon review of the release patterns it appears that each version release of the IGSTK module follows a release of the Sandbox module which is released with the intention of testing the newly released code that has not been subjected to higher policies of quality. This method of sandboxing demonstrates a high level of attention to good design methods and a serious commitment to technical quality.

These observations all seem to represent an agile approach to software design. This approach is marked by examples of rapid, continuous development over the life of the project, frequently released versions of the software, an active self organized team of developers with a demonstrated pattern of planned organization. The visualization suggests a design that facilitates understandability as developers moved from control of files to others readily. It also suggests that IGSTK was designed with an emphasis on testability. The visualization suggested good attention to solid design principles and technical strength through rigid testing processes. All of these are signature elements of agile methods in software construction [6]. Through this analysis it is evident that testability and understandability are two architectural qualities that this agile design methodology supports. Testability also leads to supporting another architectural quality, safety.

Can this type of observed agile behavior be used to support the IGSTK architecture? If the architecture requires strong support for safety and robustness, will an agile approach support these quality goals? One of the major faulty criticisms of the agile software development model is that it represents an undisciplined approach to software construction [7]. This criticism embodies the resistance from the school of thought that places more value on a plan-driven method such as a waterfall model, for software construction. Proponents of agile methods in turn argue that a focus on talented individuals working with customers to achieve goals is preferred over the weighed down process oriented development associated with plan driven methods [6].

It is interesting to note that several industry leaders, expressing concerns over the appropriateness of agile methods in mission critical projects, have suggested that a risk analysis model should be used to decide whether to use a plan-driven method or agile methods [7]. The IGSTK project demonstrates the extent to which care has been taken to consistently issue sandbox releases. This process appears to represent a significant portion of the development process. Sandboxing allows for a platform to test new or prototyped code. In the case of IGSTK it appears that strict adherence through the implementation of sandboxing as evidenced in the SCM visualization, as a means to validate quality as well as extensive guidelines for testing requirements and best practices, could be used to ensure the required level of quality that is needed for safety critical applications [7]. The open source historical artifact analysis has uncovered a pattern of agile testing centric design process. For the next portion of the project a software architecture validation tool will be used to evaluate the IGSTK architecture, paying particular attention to any effect that an agile software process may have had on the architectural decisions.

C. Architectural Tradeoff Analysis Method (ATAM)

Phase 1 - Present the Modified ATAM Approach

This evaluation model was chosen to explore elements of the IGSTK Architecture in order to clarify the quality attribute requirements of the architecture through mapping of architectural decisions. It is also meant to provide a set of risks, sensitivity points and tradeoffs. The ATAM affords a concise presentation of the architecture [10]. The ATAM developed by the SEI at Carnegie Mellon, is made up of nine steps. Steps 7 and 8 are repeat attempts of steps 5 and 6 with the only difference being that they are performed at varying levels of the organization. For this implementation of the ATAM all information is extracted from the same set of documentation and not from individuals at different levels of the organization. For this reason, these two steps will not be performed leaving us with only 7 steps in total. The first of these steps encompasses an explanation of the ATAM and what it attempts to accomplish. The steps continue as follows [10]:

Step 1 - Present ATAM
Step 2 - Present Business Drivers
Step 3 - Present Architecture
Step 4 - Identify Architectural Approaches
Step 5 - Generate Quality Attribute Utility Tree
Step 6 - Analyze Architectural Approaches
Step 7 - Present Results

While a typical ATAM process would rely on extensive interactions with the project architects, in this case the process relies on the existing documentation of the IGSTK project. While a typical ATAM exercise draws from the collective knowledge of an organization, this study will draw primarily from the following documents: "IGSTK: An Open Source Software Platform for Image-Guided Surgery", submitted to IEEE Computer December 2005 [11], the recently released IGSTK The Book for release 2.0, authored by Kevin Cleary and the Insight Software Consortium [8], an exhaustive IGSTK wiki site [12] as well as a legacy version of the wiki site[13]. This evaluation will generate the following: a set of discovered and prioritized scenarios, questions for better understanding and evaluating the architecture, a “utility” tree, a description and prioritization of the driving architectural requirements, architectural approaches and styles used in the IGSTK, the risks and non-risks, points of sensitivity points and tradeoffs. The main goal of the ATAM will be to determine to

what extent the quality requirements are satisfied by the different architectural approaches.

Phase 2 - Present the Business Drivers

There are several important functional requirements of the IGSTK system. The main goal of IGSTK is to create an open source framework for creating and validating reusable, robust image-guided software specifically for surgical applications. The IGSTK framework sets out to accomplish the following [8]:

- The ability to read and display medical images including CT and MRI in DICOM format.
- An interface to common tracking.
- A graphical user interface and visualization capability including a four-quadrant view (axial, sagittal, coronal, and 3D) as well as a multi-slice axial view (from 1 by 1 to many by many such as 10 by 10).
- Registration: point based registration and a means for selecting these points.
- Robust common internal software services for logging, exception-handling and problem resolution.

As an open source community there are certain organizational constraints. Those participating on IGSTK are separated by spatial boundaries. Communication is mostly restricted to email, message boards, teleconferences or other types of non face-to-face forms of communication. The fact that IGSTK is a framework built for others to create image guided surgery applications, the project contributors are faced with technical constraints associated with creating a product that is flexible enough for other systems to adopt yet rigid enough to maintain levels of quality and safety. Another significant technical constraint is that of domain complexity. Building an image guided surgery application framework requires significant collaboration between developers and industry experts.

The concept for creating IGSTK arose partly from a group of researchers at Georgetown University who observed that providing an open source framework layer specifically for creating image guided surgery, on top of other existing open source frameworks could be of great value to those who work in the area of image guided surgery. By creating a way to greatly simplify the implementation of image guided surgery tools through the use of standard components for reading DICOM images, image display, segmentation, registration, and an AURORA interface, biomedical researchers and clinicians stand to benefit a great deal from the IGSTK architecture [11].

The major architectural drivers that shape this architecture are closely linked to the fact that these tools will be used in the operating room where the well being of others are placed at risk. As such, safety is of the utmost concern. Reliability and fault tolerance are also major quality attribute goals.

Phase 3 – Present the Architecture

The architecture presentation begins with a run-through of some of the technical platform constraints associated with the architecture. IGSTK is built to compile on the following operating system compilers; Microsoft Windows Visual C++ 7.0, Microsoft Windows Visual C++ 7.1, Microsoft Windows Visual C++ 8.0, Linux GCC 3.4, Linux GCC 4.1, and Mac OSX GCC 4.0 [8].

IGSTK is a layered framework and as such is built on other open source toolkits. In this layered architecture the operating system is the foundation layer with a set of others above. The Insight Toolkit (ITK) offers image analysis functionalities and infrastructure classes. The Visualiation Toolkit (VTK) provides image and geometrical model displays. It also manages substantial user interaction. The Fast Light Toolkit FLTK or QT can be used for the GUI. The IGSTK is built on top of these other layers. Image-Guided Surgery (IGS) applications are built on top of IGSTK. Figure 5 offers a representation of the layered architecture

Fig 5. IGSTK Layered Architecture [8]

IGS code can only interface with IGSTK classes not with the underlying ITK and VTK classes. The reason for this level of abstraction is to force application developers to pass through the safeguards that IGSTK enforces. IGSTK classes restrict the ability of IGS developers to use a class in a way that is not in keeping with the limited use cases enforced by the IGSTK class.

Figure 6 represents the main components of the IGSTK system. It gives a concise summary of the main pieces that make up the toolkit. The Display category deals with classes that will be rendering and displaying the surgical scene. It is close to the GUI. Next are two categories that deal with issues surrounding geometrical aspects of scene objects and visual representations of those objects. Tracking is a category that has to do with mapping trackers that track the geometric position and orientation of the surgical tools to the 3D space. The Reader category represents all the classes that bring data into a visual representation. The Calibration category is used for computing the common points between the image and the tracker to capture an accurate representation of the surgical scene. The Services category provides encapsulated services to IGSTK components. Timing contains the classes that manage the real-time coordination of everything that the surgeon sees in the room and on the display. It is necessary that this represents a consistent, real-time view of the surgical operation.

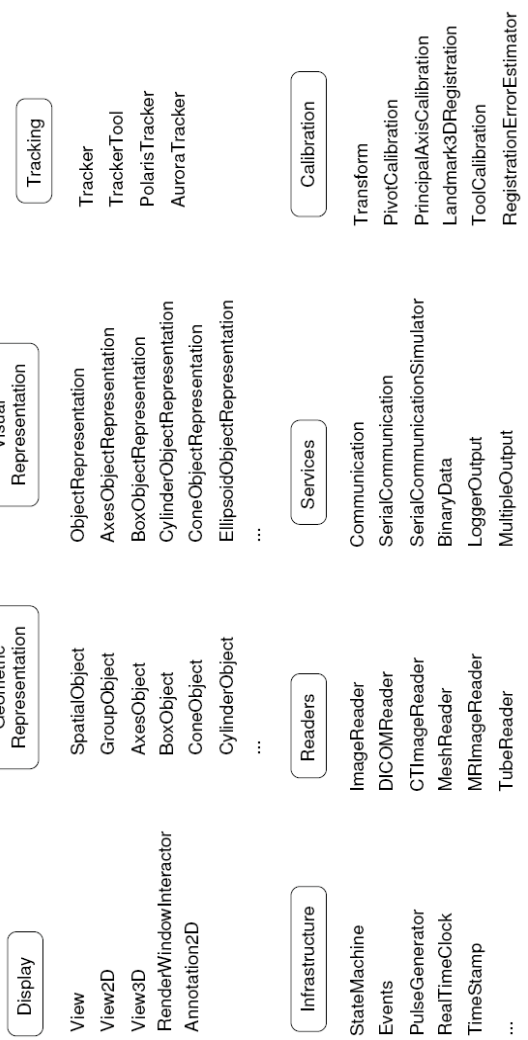


Fig 6. Components of the IGSTK [8]

No master class is needed for controlling all the activities at the top level of the application. This is due to the timing architecture which provides self contained behavior of each component of the toolkit. Below is a state machine diagram of the igstk::PulseGenerator class. The pulse generator is constantly alternating between the PulsingState and the WaitingEventReturnState, when it is actually on. Inserting items into the state machine queue is what is accomplished through alternating to the appropriate state.

The decision to use a state machine based architecture strikes at the heart of the main architectural requirement to protect the patient. The need for a lightweight robust fault tolerant system is another architectural requirement, which arises from the overall safety risk which overshadows the project. These requirements are satisfied through a state machine implementation as well as a layered architecture consisting of medium size components communicating via an event observer pattern implementation. The logic of these components is controlled by a state machine in order to restrict misuse.

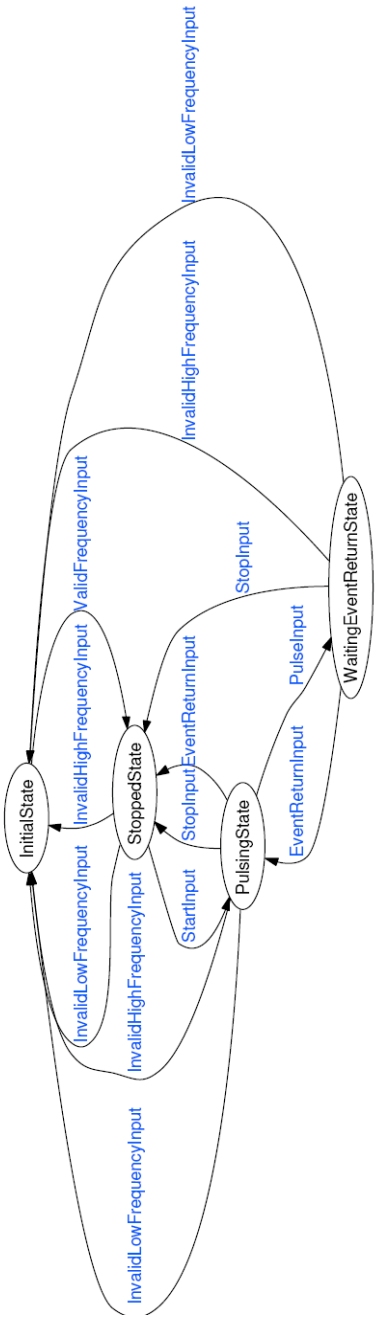


Fig 8. State Machine Diagram of the PulseGenerator Class [8]

Phase 4 - Identify Architectural Approaches

This short step lists the already identified architectural styles and approaches that were identified in the previous section where the key elements of the architecture were presented. Safety has been identified as the first and foremost concern. This is addressed through the state machine implementation. Communication between components occurs through an event observer pattern. This has the result of preserving decoupling and encapsulation as well promoting reuse of code. The IGSTK Framework is a layered architecture built on top of a

set of other components in order to abstract functionality into a top layer.

Phase 5 - Generate Quality Attribute Utility Tree

In this step the main goal will be to capture quality attribute scenarios. This will be done through the creation of a quality attribute utility tree. This approach relies on a top-down approach which starts with "utility", serving as the single most important quality goal. Quality goals are the further refined to more specific goals such as performance, security, or interoperability. The final leaf nodes are specific instances of quality scenarios. Each of these specific characterized attribute scenarios is weighted in two areas with a respective rating of high (H), medium (M), and low (L). The first rating indicates the importance of the item in achieving the overall success of the system. The second rating indicates the relative difficulty in achieving the specific quality for which it was written.

TABLE I I
QUALITY ATTRIBUTE UTILITY TREE

	Quality Attributes	Quality Sub-Factors	Attribute Characterizations
Utility	Safety/Robustness	Framework Misuse	H, H Prevent framework misuse and ensure IGS applications access to a basic unified layer
			H, H - IGSTK classes will not throw exception in order to curb misuse
		Visual/Instrumentation Failure	H, H – Ensure that the surgical view is up to date.
		Component Failure	H, M – Provide logging when lower level component failure occurs and provide failure meausre to user.
	Testability	Error Detection	H, M – Provide logging when lower level component failure occurs and provide failure meausre to user.
			H, H - Create a set of predictable deterministic behaviors with a high level of code coverage 90%
		Code Incorrectness Detect-ability	H, M - Create a system of testing that uses sandboxing to test and prototype all release
	Usability	Latency	M, M - Response Time for visualization is reduced to the smallest possible delay

Phase 6 - Analyze Architectural Approaches

This is the presented outline for capturing architectural approaches for the scenarios pulled from the utility tree. The main purpose of this phase is to document the relationship between the architectural decisions that have been made and the quality attribute requirements. Architectural approaches have been discussed in phase 4, however it is yet to be explained how these approaches help to realize the quality requirements. For this purpose a set of architectural approach descriptions is created using the scenarios generated as attribute characterizations from the utility tree to help understand these approaches more in depth, helping to identify risks, sensitivity points and tradeoffs. The IGSTK architectural approach descriptions that were generated as a part of this step can be found in appendix A. Table III is a template for an architectural approach description.

TABLE I I I
ARCHITECTURAL ATTRIBUTE DESCRIPTION

Scenario: <a scenario from the utility tree of from scenario brainstorming>			
Attribute: *<performance, security, availability, etc.>*			
Environment: *<relevant assumptions about the environment in which the system resides >*			
Stimulus: <a precise statement of the quality attribute stimulus (e.g., failure, threat, modification, …) embodied by the scenario>			
Response: *<a precise statement of the quality attribute response (e.g., response time, measure of difficulty of modification)>*			
Architectural Decisions:	Risk	Sensitivity	Tradeoff
<list of architectural decisions affecting quality attribute response>	*<risk #>*	*<sens. point #>*	*<tradeoff #>*
Reasoning: *<rationale for why the list of architectural decisions contribute to meeting the quality attribute response requirement>*			
Architectural Diagram: *<potential diagram or diagrams of architectural views>*			

Figure 2.4 Architectural Approach Documentation Template [10]

The main point of generating these architectural attribute descriptions is to generate as much information as possible with regard to the approach in question. The goal is to reason that the approach being evaluated will fulfill the requirements of the system. From the description a list of sensitivity points, risks, and tradeoffs are created. Each item on these lists will be associated with achieving the scenario that satisfies the sub-quality requirements from the utility tree.

Summary of Approaches

A set of risks, sensitivity points and tradeoffs associated with the architectural approaches were uncovered with the help of the architectural approach description generation activity. These activities can be found in the appendix. The resulting lists of tradeoffs, risks, and sensitivity points are also found in the appendix, however in this portion of the paper, these risks, sensitivity points and tradeoffs are summarized and explored for each of the three main architectural approaches; state machine implementation, encapsulation of the layered architecture and communication via the event observer pattern.

State Machine

The implementation of a state machine forces all events to pass through the IGSTK model for handling states, transitions and actions. The developer sacrifices flexibility over lower layer api calls for the convenience and safety that the IGSTK has to offer in the form of a state machine which manages all behavior of the system. By implementing a state machine architecture all behavior passes through the state machine and as such the state machine implementation is aware of nearly all possible behavior. The requirement to account for all possible behaviors in the state machine makes it substantially easier for the creators of IGSTK to provide at least 90% code coverage, as all decisions are accounted for. This heightened code coverage results in a limited set of functionality in comparison to the combined size of the underlying third party APIs. 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. The tradeoff between increased features and required safety levels is necessary. Despite the limited size of objects in IGSTK a sensitivity point can be found in the requirement to account for all potential outcomes. This sensitivity is negatively correlated the size of the object and with the number of methods it extends.

Layered Architecture

The layered nature of the IGSTK implements encapsulation of toolkit functionalities to prevent developers from directly manipulating objects. This is another example of tradeoffs 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. One potential risk results from implementation of a layered 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. The state machine carries the onus for handling lower level errors and this way safety is maintained, however potential problems with lower level encapsulated objects could still result in undesirable behavior.

Observer Pattern

A critical component of the IGSTK architecture is the timing mechanism. One of the most important tasks of the IGSTK is to manage surgical views. Keeping a real time view of the physical view is essential to patient safety and is a potential sensitivity point. A series of steady pulses are used to manage the rate of visual and physical representations over time. Expired views are not displayed and visual indicators are displayed. In order to ensure safety, surgical views may not be available at critical times. There is an implicit tradeoff between usability and performance. The amount of buffered data is set with an expiration data and measured against the pseudo real-time clock (RTC) implementation in order to generate a user notification as to whether the view can be trusted. If this management fails serious damage could result from mismanaged physical and logical views. The IGS application developer can set the rate of operations per pulse intervals resulting in a more contiguously updated view, while transforms may expire if they are not able to process at the desired rate. Logging data is buffered in order to improve performance. Logging data is in risk of not being written quick enough to disk in case of immediate component failure. Errors can be reported in the form of events. The state machine acts as an observer when an event is transduced as an input into the state machine that results in writing to the log file and notifying the user interface. This is a tradeoff with respect to the flexibility of an IGS developers ability to handle an error generated from a lower level component and to provide his/her own mechanism for handling the error.

Phase 7 – Presenting the Results

A concise presentation of the architecture has been offered. The project goals and motivations were discussed. A discussion was presented focusing around the quality requirements of IGSTK in particular and the importance of safety as a quality requirement. The utility tree allowed for the mapping of architectural decisions to quality requirements and the generation of specific scenarios. A set of tradeoffs, sensitivity points, and risks were developed through an architectural approach description generation activity, the results of which are available in the appendix. Specific lists of tradeoffs, sensitivity points, and risks associated with each description generation were generated and are also available in the appendix.

IV. Conclusions

This paper presented a method for analyzing the architecture that provided a way to map architectural approaches to quality requirements. This has been accomplished through performing the Architectural Tradeoff Analysis Method developed by SEI. A search for risks, sensitivities and tradeoffs, was apart of this analysis. In addition to this architectural evaluation another evaluation was performed that relied on the availability of publicly available software artifacts to mine for historical data

that suggests a software development process for the open source IGSTK. It was determined that the data represents the implementation of testing-centric agile methods approach to software design. It was concluded that this agile test-centric design process reinforced both the understandability and testability quality attributes.

This study provides fertile ground for the exploration of the role between software methodology and software architecture exploration in open source systems.

APPENDIX

A. Architectural Approach Descriptions

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

Attribute: Safety/Robustness

Environment: Normal / Strained Use

Stimulus: Developer Misuse

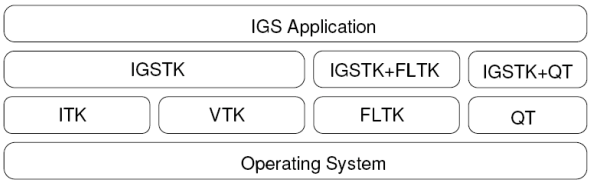
Response: Classes that inherit lower level component behavior are restricted to basic functionality.

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:



Layered Architecture[8]

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

Attribute: Safety/Robustness

Environment: Normal Use / Strained Use

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

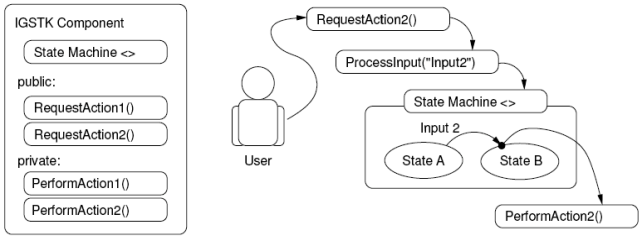
Response: IGSTK classes will not throw exception in order to curb misuse

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.
- 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 [8]

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

Attribute: Safety/Robustness

Environment: Normal use

Stimulus: System may experience stress conditions

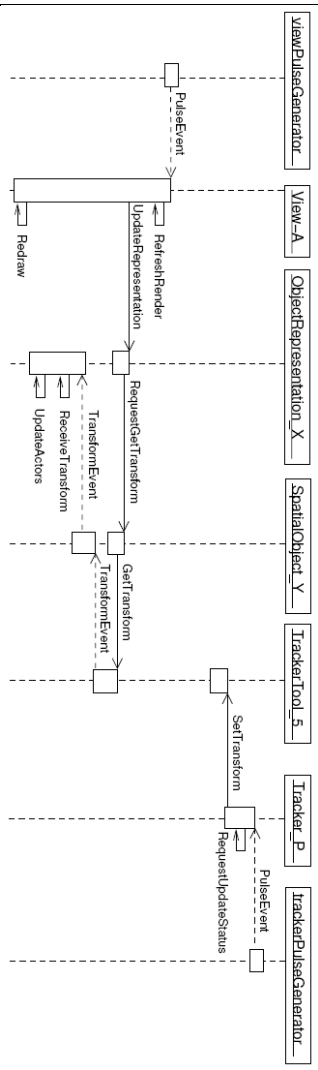
Response: Synchronicity is maintained through an event observer pattern. Through a series of pulses a tracker class will query the actual hardware tracker device and will get from it information about the position of the tracked instruments in the operating room.

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:



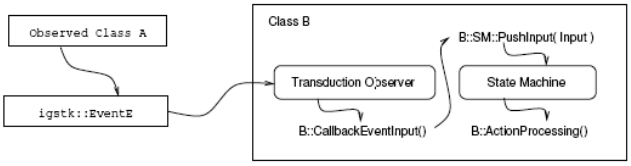
UML Sequence Diagram of the IGSTK Timing Collaboration [8]

Scenario: S5 Provide logging when lower level component failure occurs, and provide failure message to user.			
Attribute: Testability			
Environment: System is performing tracking functions			
Stimulus: lower layer component failure			
Response: Logging is buffered and written to file for later review			
Architectural decisions	Risk	Sensitivity	Tradeoff
Buffered logging	R2		
Events			T5

Reasoning:

- When logging is buffered, buffered data is in risk of not being written quick enough in case of immediate component failure
- Errors can be reported in the form of events. The state machine acts as an observer when an event is transduced as an input into the state machine that results in writing to the log file and notifying the user interface. This is a tradeoff with respect to the flexibility of an IGS developers ability to handle an error generated from a lower level component and to provide his/her own mechanism for handling the error.

Architectural Diagram:



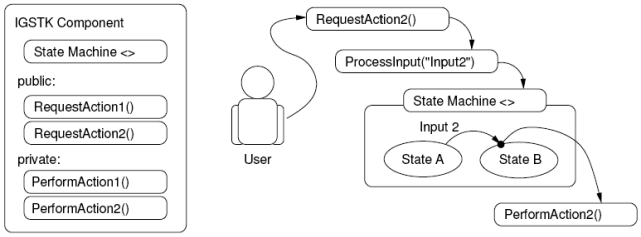
Event Observer Transduction Diagram [8]

Scenario: S6 Create a set of predictable deterministic behaviors with a high level of code coverage 90%			
Attribute: Testability			
Environment: The system operating in all possible environments (stressed, regular use, heavy use)			
Stimulus: An intentional planned testing environment where all possible behaviors are tested under every known environment			
Response: Use a state machine based architecture to implement a high level of code coverage			
Architectural decisions	Risk	Sensitivity	Tradeoff
Sate machine architecture		S2	T4

Reasoning:

- By implementing a state machine architecture a tradeoff between IGS developer flexibility/freedom and safety/reliability/predictability.
- A sensitivity of the system may manifest itself through the difficulty in creating a state machine that consists of every single set of possible outcomes. This sensitivity may be directly correlated to the size of the objects , as smaller sized objects would result in a reduced set of behaviors

Architectural Diagram:



Separation Between Public and Private Interface of IGSTK Components [8]

Scenario: S7 Create a system of testing that uses sandboxing to test and prototype all release			
Attribute: Testability			
Environment: An intentional planned testing environment where all possible behaviors are tested under every known environment			
Stimulus: A nightly process attempting to build the changes committed to the project.			
Response: Sandbox implementation for multiple codelines			
Architectural	Risk	Sensitivity	Tradeoff

decisions			
Sandboxing			T6

Reasoning:
- While sandboxing may tend to be considered a development process approach as opposed to an actual architectural approach, it may also be considered architectural in that it is a replica of elements of a system with a reduced level of overall quality. This approach exchanges quality as a tradeoff for testability. In the nightly test environment code is more testable as it is not a threat to any production level.

Scenario: S8 Response Time for visualization is reduced to the smallest possible delay

Attribute: Usability

Environment: Stressed/Normal Use

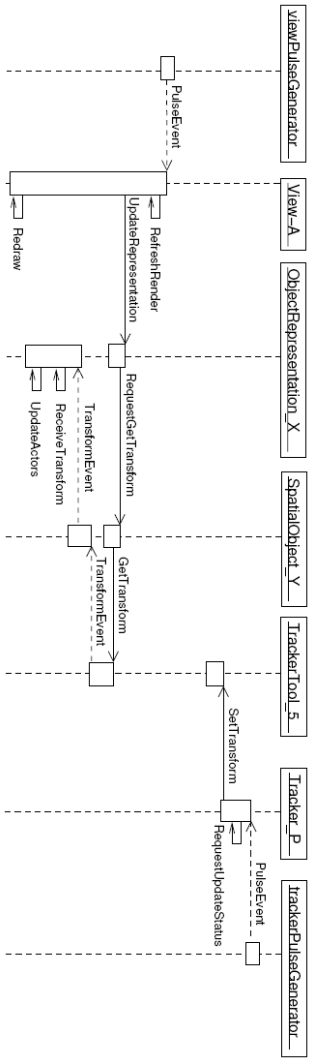
Stimulus: Tracking functionality

Response: Creating a timing mechanism to manage the real time nature of surgical views

Architectural decisions	Risk	Sensitivity	Tradeoff
Event/Observer Pattern		S1	
Timing Architectural Design	R3		T8, T7

Reasoning:
- The IGS application developer can set the rate of operations per pulse intervals resulting in a more contiguously updated view, while transforms mayexpire if they are not able to process at the desired rate.
- In order to ensure safety surgical views may not be available at critical times.
- There is an implicit tradeoff between usability and performance. The amount of buffered data is set with an expiration data and measured against the pulse tracker mechanism in order to generate a user notification as to whether the view can be trusted.

Architectural Diagram:



UML Sequence Diagram of the IGSTK Timing Collaboration [8]

V. APPENDIX B

	Risks
R1	Layered architecture relying on other toolkits/APIs not maintained by developers
R2	When logging is buffered, buffered data is in risk of not being written quick enough in case of immediate component failure
R3	In order to ensure safety surgical views may not be available at critical times.

	Sensitivities
S1	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
S2	Difficulty in creating a state machine that consists of every single set of possible outcomes. This sensitivity may be directly correlated to the size of the objects

	Tradeoffs
T1	Encapsulation of toolkit functionalities, trading flexibility for safety.
T2	Medium sized objects resulting in a tradeoff between reduced functionality for robust applications.
T3	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.
T4	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.
T5	Forcing error handling of lower layer components to be handled by the observer state machine, using error events as inputs into the state machine
T6	This approach exchanges quality as a tradeoff for testability. In the nightly test environment code is more testable as it is not a threat to any production level.
T7	There is an implicit tradeoff between usability and performance. The amount of buffered data is set with an expiration data and measured against the pulse tracker mechanism in order to generate a user notification as to whether the view can be trusted.
T8	The IGS application developer can set the rate of operations per pulse intervals resulting in a more contiguously updated view, while transforms may expire if they are not able to process at the desired rate.

REFERENCES

[1] G. Polancic, R.V. Horvat, T. Rozman, "Comparative assessment of open source software using easy accessible data," International Conference on Information Technology Interfaces, 2004, Vol. 1, pp. 673 - 678

[2] I. Alsmadi, K. Magel, "Software Maintenance, 2006. ICSM '06. 22nd IEEE International Conference on ", 2006, Sept, pp. 276-278

[3] CVSGrab Available: http://www.win.tue.nl/~lvoinea/VCN.html

[4] StatCVS-XML Available: http://statcvs-xml.berlios.de/

[5] CVSGraph Available: http://www.akhphd.au.dk/~bertho/cvsgraph/

[6] V. Subramaniam, Andy Hunt, Practices of an Agile Developer, Raleigh North Carolina, The Pragmatic Bookshelf, 2006

[7] B. Boehm, R. Turner, Balancing Agility and Discipline: A Guide for the Perplexed, Boston, MA, Addison-Wesley, 2004

[8] K. Cleary, The Insight Software Consortium, IGSTK: The Book For release 2.0, Gaithersburg, Maryland, Signature Book Printing, 2007

[9] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, Second Edition, Addison Wesley, 2004

[10] R. Kazman, M. Klein, P. Clements, ATAM: Method for Architecture Evaluation CMU/SEI-2000-TR-004, ADA382629). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. Available: http://www.sei.cmu.edu/publications/documents/00.reports/00tr004.html

[11] K. Gary, M. B. Blake, L. Ibanez, D. Gobbi, S. Aylward, K. Cleary." IGSTK: An Open Source Software Platform for Image-Guided Surgery." Submitted to IEEE Computer

[12] IGSTK wiki papge, Available: http://public.kitware.com/IGSTKWIKI

[13] IGSTK Legacy wiki papge, Available: http://public.kitware.com/IGSTKWIKI/index.php/Previous_wiki_page