ISVis Adaptation Project

Fall 2008 Term Paper (Updated)

http://www.cc.gatech.edu/morale/tools/isvis/2009/

Andrew Bernard
College of Computing, Georgia Tech
agbernard@gmail.com

March 9, 2009

Abstract

The objective of this study is to examine some of the challenges that come with refactoring a 10+ year old legacy system. The research was conducted over the course of 15 weeks after which we were able to compare our experience to that of others that have documented similar endeavors. The lessons described in this paper can serve to add to the documentation of the task of refactoring a legacy system.

1 Introduction

There are a variety of reasons people have encountered prompting them to re-examine the structure of a software application. This re-structuring is often referred to as either "refactoring" or "reengineering" - but mistakenly, they are sometimes used interchangeably. According to Fowler in [8] "refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure." Reengineering has a much broader scope according to Chikofsky and Cross - in [6] they define reengineering as "the examination and alteration of a system to reconstitute it in a new form." While a main goal of refactoring is to maintain the behavior of the code being changed, it is very often the case that new functionality is added (or old functionality changed) during a reengineering project. Reengineering very often necessitates some sort of refactoring, but refactoring very rarely has the impact that a reengineering process would.

The main characteristic that is common to both processes is that the practitioner is trying to change the target application (or system) in some way. Generalizing the motives for each of these processes might lead us to the notion that we refactor to improve maintainability and we reengineer to add functionality. Taking these motives into consideration, we quickly come to realize that when dealing with a legacy application, both processes

become very necessary and both bring with them unique challenges. We will explore some of those challenges in Section 2 and try to see if we can use the experience of others to help us in our endeavor to adapt a ten year old program used to support the browsing and analysis of program execution scenarios. The program is called Interaction Scenario Visualizer (ISVis) and further explanation of its adaptation project is given in Section 3. Reflections on ISVis and the research conducted is given in Section 4 and we end with some ideas for future work.

2 Related Work

There is a plethora of work on how to handle legacy systems. [16] summarizes the efforts of an XP team from ThoughtWorks Technologies to introduce new features to a legacy system. It provides a great informal guide to working with a legacy system using an agile development process (although not all the pointers necessitate an agile development process). Since agile methods are relatively new and most legacy systems were developed under a non-agile development process, it is interesting to see those techniques applied to an older system to try and make it more maintainable.

In [17], Thomas argues that the older the code base and the more mission critical the application, the more difficult it is to maintain or enhance it. One primary difficulty is just understanding the code base which may not be well-designed or well-written. For this reason, the author promotes the idea that when working with legacy software, Discovery and Transformation should be the focus before considering Design and Development. Discovery combines the stories obtained from experienced developers and customers with knowledge gained by analyzing the code, associated documentation, and test cases. Transformations of the code should be done systematically in small pieces, testing each change before moving to the next. These two concepts are important parts of the plan for refactoring ISVis (which will be elaborated on below).

The discovery process is also a primary concern of the research described in [9], where the authors address the problems associated with identifying hidden concerns in legacy code. A hidden concern (HC) is any issue that has been grafted into existing code to extend it's functionality in some way that is different from the piece of code it is being added to. This decreases the code quality (by reducing it's coherence) because the code is now handling more than one specific issue. More often than not, an HC suffers from two problems that make them hard to track down: they are usually scattered throughout the project and tangled with other code. Consequently the primary focus as it relates to HC's is how to identify and extract the code related to a hidden concern - a task which the authors recognize as being non-trivial.

A possible approach to identifying hidden concerns could be feature-oriented refactoring, as described in [11]. This is the process of decomposing a program into features, thus recovering a feature based design and giving it an important form of extensibility. A characteristic of feature-oriented refactoring (FOR) that makes it challenging is that a

feature cannot always be implemented by a single module. To try and deal with this problem Batory et al. describe a methodology for expressing programs and features as mathematical equations. Engaging in this practice gives the practitioner a reliable way of predicting the effects of adding or removing features from the base program because the derived equations communicate properties that decompositions and their underlying base modules and derivatives must have. These properties are automatically checked by the tools the authors developed to aid in the FOR process. The authors developed an Eclipse plugin to be used with legacy Java applications. The plugin guides the user through a five-step process which defines the base modules of the program, optional modules, and then reconstitutes the program for the user.

Pertaining to ISVis, FOR would seem like a useful technique, but since the tool the authors developed only works for Java programs, the work involved in implementing their ideas by hand would probably not be worth the effort. Additionally, this method requires a substantial knowledge of the program being examined and that would make using it on ISVis difficult since ISVis suffers from the "embedded knowledge" symptom of an aged system.

Aging symptoms of legacy systems are described in [18] where Visaggio et al. try to advise developers on how to measure and handle the inevitable aging of a system. In addition to "embedded knowledge" — defined as knowledge about the system that can no longer be derived from existing documentation — the authors describe four other symptoms to watch for:

- Pollution - components that are no longer used by the users
- Poor Lexicon - variables and components that have names with no meaning
- Coupling - the flow of control/data between components is very tightly linked
- Layered Architectures - occurs when there are several different solutions spread out across a system's architecture.

ISVis exhibits all of the above symptoms to varying degrees, suffering most severely from embedded knowledge.

Because of these symptoms, we know ISVis definitely has to be refactored and we know from our definition above that after a refactoring, a program must be syntactically correct. There are ways to preserve a program's behavior which rely on the compiler catching mistakes, but there are some errors which could change the behavior of a program that a compiler would not be able to catch. A particular set of program properties have been found to be easily violated if explicit checks are not made before a program is refactored. Those properties are described in [15]. While ISVis is not quite at the stage of implementing a refactor (for reasons described in Section 4), the properties described by Opdyke should be considered when it is time to start changing the code.

When that time comes, there are a variety of ideas to draw upon for guidance on how

to proceed - or even on how not to proceed. In [7], Doblar and Newcombe argue for an increased focus on automating transformation processes based on the statistic that, on average, a well trained programmer can only transform about 160 LOC per day. To transform an entire system at that rate would be too costly. The innovative part of the process described by Doblar et al. is that they propose using a suite of artificial intelligence technology tools to automate an unexaggerated 99 percent of the transformation work. Most of what is left for manual transformation should just be what truly requires a human decision maker. They make the point that while high levels of automation are achievable for transformation tasks there will always be tasks that are manually intensive. Nevertheless, they argue that anything that can be automated should be.

Unfortunately, the paper did not go into how the AI performs it's tasks, but that is most likely because the research comes from a private company dealing with defense software. Also, the techniques described were applied to languages to convert them to C++ - it is unclear whether they would work if trying to perform a transformation to the same language. Regardless, these techniques would probably be overkill for reengineering ISVis.

Another idea could be to use a goal model - a graph structure representing stakeholder goals and their inter-dependencies. [21] describes a methodology for obtaining a goal model from legacy code, but it is a process that is currently unnecessary for the ISVis project since the main focus of the reengineering process has already been identified. However, the methods described by Yu et al. could potentially be useful in future refactoring efforts on ISVis as they would allow for the categorization of the code to help identify candidates for refactoring.

In [12], Heineman and Mehta describe a 3-step methodology for evolving a system into components based on the features the system implements. They applied their proposed process to a 14 year old product and they describe the lessons they learned from their work. Two of those lessons are applicable to the ISVis adaptation project: the authors conclude that features are good candidates for evolution into components if they "change often, are concentrated in fewer functions, or depend on or share global variables as a means of communication." While the "display feature" of ISVis might not necessarily change often, it was chosen as the primary candidate for refactoring because it is currently nonfunctional on modern machines and this holds it back from being used or updated. Secondly, the authors comment that "the true measure of a successful evolution methodology is in reduced future maintenance costs." From that perspective, the adaption of ISVis to have a plugin-style interface for its display code will certainly be a success since that accomplishment will drastically reduce the cost of maintaining that portion of the code.

[10] describes the use of a formula to calculate the cost effectiveness of a planned refactor. While it still remains the job of the developer to decide what to refactor, the proposed formula aims to help determine when that refactor should happen by calculating the return on investment (ROI) for the planned activity. If the ROI is greater than or equal to one, then the planned refactoring is deemed to be cost effective. It would be hard to determine the ROI for the planned refactoring of ISVis since there are currently no regression tests available, a key component to calculating the formula. It would be a rather

meaningless effort anyway since the planned refactor is a necessity to get the application running on modern machines — so we wouldn't not refactor it if our ROI was below one. Nevertheless, this could prove to be useful in the future, after ISVis is working properly, to determine how to maximize the effort spent on further modifications.

[3] describes a process for incrementally re-engineering a legacy application whose architecture has degraded over time. Specifically, the authors' goal was to develop a target architecture for a system and then re-engineer the system to the desired architecture using a defined series of steps. Reengineering ISVis will most likely produce the architecture erosion Coelho et al. talk about since there is no clearly defined documentation on the design of the system. The lack of architectural documentation will basically ensure that any changes made to the code will not be in line with the original author's thinking, thus producing the erosion.

[4] describes a process model for re-engineering a legacy system - particularly a system that is in use and cannot be shut down for an extended period of time. One of the main issues the paper aims to address is the traditional approach of reengineering a system all at once and consequently prohibiting the use of the system while it is being changed. To combat this, the authors propose a method which share features with the Chicken-Little Strategy [5] and the Butterfly Methodology [20] while alleviating some of their weaknesses. [4] was used as inspiration for the Refactoring Plan that was developed for the ISVis display (found in Section 3.3). While [4] was targeted for data-centric systems, the more general idea of iterating over a reengineering process will definitely be applied to the process for adapting ISVis.

3 ISVis Adaptation Project

The purpose of ISVis is to support the browsing and analysis of execution scenarios, derived from actual program executions - more details about the application can be found at [2]. It is useful during software engineering tasks requiring a behavioral understanding of programs, such as design recovery, architecture localization, design/implementation validation, and reengineering. The key features of ISVis are its use of visualization techniques to depict the large amounts of information available to a user, and the notion of recurring scenarios, or "interaction patterns", as abstractions which help bridge the gap between low-level event traces and high-level design models.

ISVis was originally developed over ten years ago in an older (pre-standard) version of C/C++ - the persistence portion was written using Rogue Wave, the user interface portion was written using the Motif toolkit and the platform for it to run on was Solaris. Since then, the program has become dated with the standardization of C++, upgrades to Motif and Rogue Wave, and the need for portability to different platforms. In Fall of 2006 a team of Georgia Tech students picked up ISVis as a semester project and started the task of adapting it to modern operating systems.

After some analysis, they decided to tackle the adaptation project in two stages. The first stage was to upgrade ISVis and get the program to run on the same platform it was originally written for. Upon that accomplishment, the second stage was to make ISVis portable to multiple platforms, including Windows. The team almost completed stage one before having to leave the project behind - documentation of their work can be found at [1]. They left it in a state where the code was up-to-date with respect to programming language standards, yet it would not run. Dependencies on external libraries, in particular Rogue Wave, had been completely eliminated and replaced by the more stable Standard Template Library (STL) classes (where appropriate). Because persistence of data was highly dependent on libraries not further available, persistence related capabilities were disabled. The system also had a known colormap issue.

It was later picked up in Spring 2008 by Angela Navarro who started off by solving the colormap problem. After fixing some bugs that were introduced during the upgrade to STL, she was able to get the program to run and accept input from the user. When she left the project, the program was also able to read a trace file and process the information in it, but she reports that the system did not produce correct results due to a custom iterator behaving unexpectedly.

This is where I picked up the project, but unfortunately I had to start a step behind Angela. Much of my time this semester has been spent trying to get ISVis to build and execute. I got it to build on Fedora 9 (the distribution on my laptop at the time) and Red Hat Enterprise Linux 4 (tampere.cc.gatech.edu) but in both cases the program would crash when trying to open the GUI. This is most likely because the code handling the display is not compatible with modern monitors or drivers.

My next attempt to get the GUI to show was to run it on an older OS with an older monitor. After updating the Makefile to allow for a smoother build process, I built it on gaia.cc.gatech.edu which runs SunOS 5.8. We did this primarily so we could use an old Sun monitor that we think should be able to display the ISVis interface. I was able to get it to execute successfully on gaia so the default screen displays. This was accomplished by setting the display class to a hard-coded X library constant which I discovered could be handled by the Sun monitor. A more desirable solution would be to have this be detected dynamically depending on the display being used. As it stands right now, some of the menu items are causing segmentation faults so that needs to be debugged. I did change the code, however, to handle trapping signals to the program should not crash when these faults occur. Along with dealing with these build issues, I was also able to start on a preliminary plan for refactoring the display code in ISVis. This work is described in the following sections.

3.1 Discovery

During my examination of the code, I produced the class diagram shown in Figure 1 showing the code related to the GUI display. `ViewManager` is the entry point to the system - when ISVis starts, it is the `main` function in `ViewManager` that gets called. `X_Application`

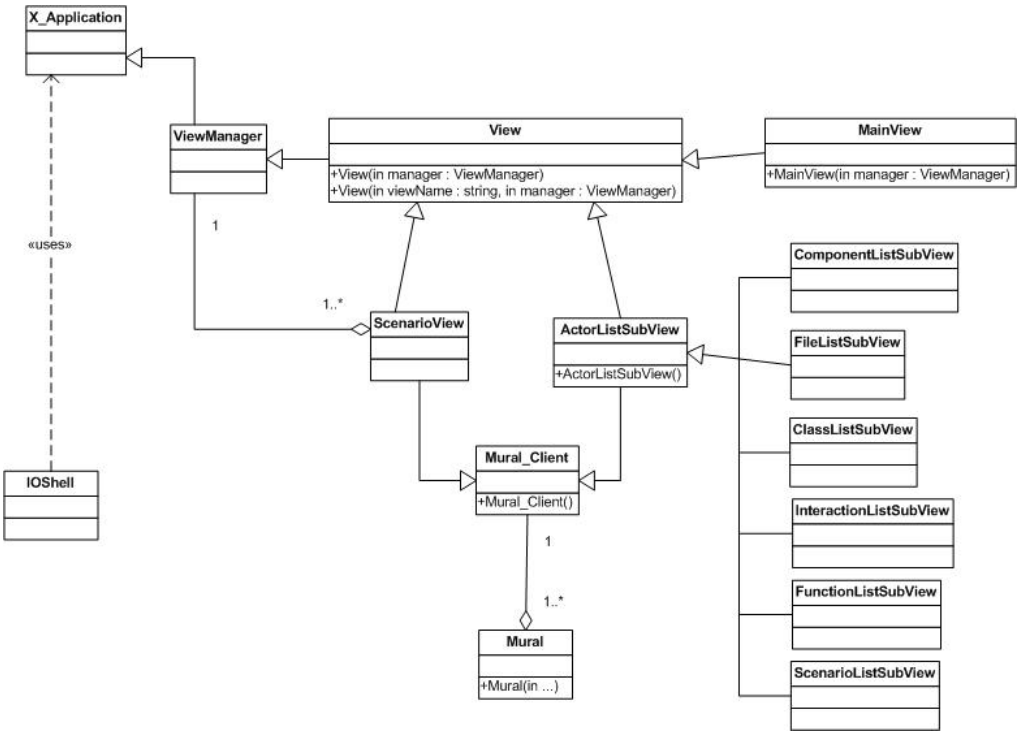

Figure 1: Class Diagram of display-related code

kind of acts like a utility class giving access to many of the X library calls that ISVis uses although it is unclear if there is a specific set of functionality it is meant to encapsulate. It is unclear because many of the other classes make calls to the X library directly so further analysis is needed to determine `X_Application`'s true purpose. `IOShell` is another utility class with methods provided for I/O functions. `MainView` seems to be responsible for instantiating, positioning, and drawing all the sub view classes. `View` is used as an abstract class, although it is not defined as one - meaning its type is used as a reference, while only it's children are ever instantiated. The rest of the classes shown relate directly to the GUI components they are named for.

This diagram, along with the statistics shown in A.1, was developed using a combination of the techniques described in [9] and [15]. In order to "flush out" a hidden concern [9] describes a process that combines text-based and type-based mining. The paper illustrates why doing them separately doesn't prove to be as effective as doing them together and then introduces the Aspect Mining Tool, developed by the authors. The techniques described in this paper would have proven to be very useful in refactoring ISVis since the display-related code is scattered in various places throughout the application and a quick way of finding it would have helped in determining what needed to be changed. Unfortunately that specific tool is only available for Java at the time of this writing.

Nevertheless, while it might not be as a ideal, text-based mining was done manually to search for references to the X library, pointing us to the classes that were related to

the display. Type-based mining was done using the compiler strategy described in [15]. The compiler strategy just involved commenting out include statements that referenced X libraries and building the code to find what data type references broke. The compiler strategy was needed to catch references to classes in the X library that did not have the typical X* naming convention.

3.2 Transformation Approach

One of the main purposes behind identifying which classes are related to the display is to facilitate its abstraction. Our eventual goal is to create an interface around the current code associated with the display. All the code interacting with the display would do so through interfaces thereby decoupling the display implementation from the program's functionality. Once this layer of abstraction is in place we will be able to "plug out" the current code used for the display and plugin something else - like code for displaying on Windows or Linux. This will make ISVis much more portable and much less brittle to future changes.

3.3 Refactoring Plan

To accomplish this transformation, the refactoring plan shown in Figure 2 was developed. At the start there are two parallel processes occurring. The left arc starts with an Analyze phase in which aging symptoms in ISVis are identified. From that identification we then choose a candidate to focus our refactoring efforts on. After making that choice we redesign the component in question and then implement the new design by refactoring.

While all of this is happening, the right arc shows two other steps that should also be taking place: building the code and capturing data of the current build to be used in regression tests after the code has changed. Once both arcs have been completed we can proceed with building the newly refactored code. The last box labeled Regression Test includes executing the code and gathering data that can be reliably compared to the data gathered from the pre-refactor execution. This begs the question: what is reliable? That cannot really be determined ahead of time as it is dependent on how fine or coarse the scope of the refactoring is - the candidate for refactoring can be anything from one method to an entire set of classes. However, as we mentioned above, we will most likely bite off very small chunks of code and iterate over this process many times to achieve our desired end result.

There are some potential issues with this plan. Firstly, the process of selecting a candidate for refactoring is going to be mostly subjective. Do we pick a candidate based on a part of the code we want to change, a part of the interface to test, or both? I imagine it will be a combination of those two options with a fair amount of trial and error taking place. Another issue will arise when it comes time to capture execution data for regression testing - we will discuss this more in Sections 4.1 and 5.

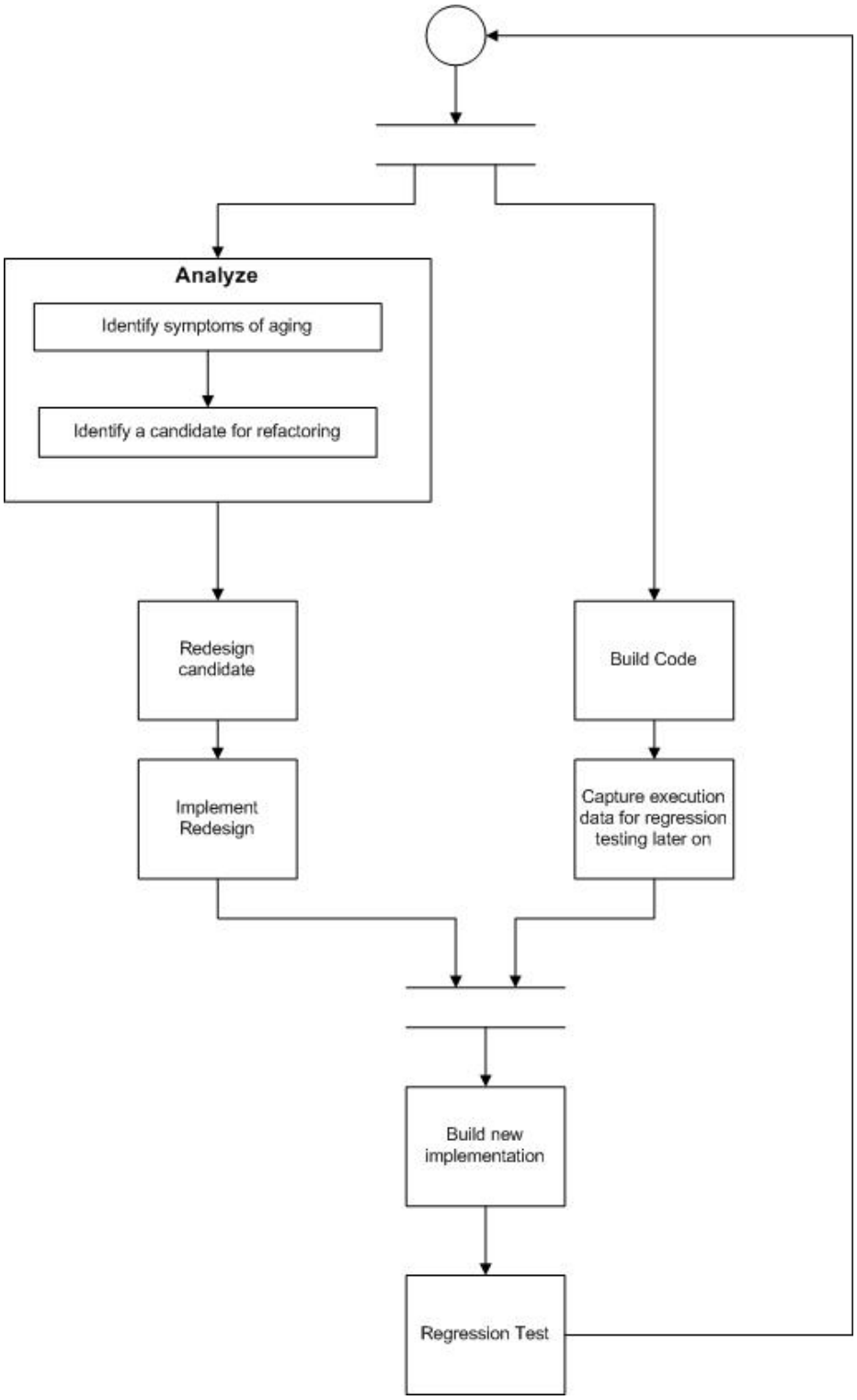

Figure 2: Refactoring Plan for ISVis

4 Reflections

4.1 Regression Testing a GUI

In section 2 we discussed a methodology for evolving a system into components - two key assumptions made in [12] are that (1) the system is coded in a language for which a code-profiling tool is available and (2) that the legacy system has regression test suites. ISVis satisfies (1) but not (2) and thus could not use the methodology described without first developing a reliable set of test cases. This raises an important issue: how do we reliably test a GUI based application? There is a technique described by Lee White in [19] where automated regression tests can be generated to test both static and dynamic event interactions in a GUI. Static interactions are those that are contained within one GUI screen. Dynamic interactions involve decisions happening on multiple screens. White describes three different solutions that could be used for generating the interaction test cases: (1) brute force method of enumerating the elements of each possible path, (2) randomly generating test cases, and (3) using Mutually Orthogonal Latin Squares. Of the three, Latin Squares could possibly be applied to ISVis, however an entirely different approach could be to use the techniques described by Memon et. al in [13].

This paper describes the use of planning, an AI technique, to automatically generate test cases for a GUI. The paper argues that instead of trying to design test cases for every possible state of a GUI, it is more effective and simpler to specify the goals a user might want to accomplish and have a tool that can automate the sequence of events that would need to occur to meet those goals. The test case generation system developed by the authors takes these goals as input in the form of initial and end states. Their system then generates sequences of actions to get between the two states, which serve as the test cases for the GUI.

While Memon and White propose two different techniques to test case generation, they both agree that automation of test cases for GUI's is a necessity to accommodate the changes that can occur during its development and maintenance. Which technique to use for ISVis still remains to be seen. There are even decisions that must be made after the test cases are in place concerning how to maintain them. Memon & Soffa write about a possible solution to that dilemma in [14] describing a technique to repair unusable test cases after the components of a GUI have changed.

When a GUI changes it is inevitable that some tests will become unusable because the chain of events that occur for interactions with the GUI will change. Normally, these unusable tests would have to be regenerated, but GUI test case generation is expensive. Memon & Soffa's solution is based on modeling the GUI's events and components and then comparing the original and modified versions of these models. From that comparison they are able to automatically detect unusable test cases and repair the ones that can be. ISVis isn't quite ready for these techniques since there are no test cases related to the GUI yet and the GUI is not in a rapid state of change, but it may be wise to consider such things when choosing a method to generate the test cases that will be needed. Further reading

on this topic is required to help develop a more reliable process in refactoring ISVis.

4.2 Aging

Knowing that ISVis exhibits the symptoms described in [18], steps should be taken to improve the general understandability of the code. After defining the symptoms to watch for, the paper went on to describe the lessons learned from reengineering a legacy system. The following lessons directly apply to our eventual refactoring of ISVis and should be kept in mind as we move forward with the reengineering process:

- Lesson 1 - "Before renewing an old software system, it is wise to clean it thoroughly of the pollution that has accumulated over the years." ISVis is polluted with dead code and ambiguous comments.
- Lesson 3 - "Extracting the knowledge incorporated in the programs is a long and costly activity." At some point, this is a process that will have to be undertaken with ISVis. The lack of code documentation is a major hurdle that anyone picking up the code has to overcome before they can make meaningful progress.
- Lesson 4 - "It is best to check the lexical quality of the programs frequently." I doubt if the class or variable names have ever been examined and refactored to improve code readability.

4.3 The Code

These lessons give ample reason why the build process and the general understandability of the code is lacking in clarity. The code is polluted with sections of dead code — this dead code is documented by the comments of previous viewers, yet the code is left there, I assume to provide clues to what the original author may have intended. Most of the comments present in the code are either commented out code or warnings of potential pitfalls. Extracting the meaning and purpose of each component of the system into well thought out comments and documentation would be a great undertaking, but also a great help to future viewers of the code.

4.4 ISVis Prerequisites

In its current state, ISVis has some pretty stiff requirements for working with it. C++ goes without saying since that is the language it is written in. Familiarity with Linux is a must, particularly

- the link-editor (ld) for dealing with build issues

- a command line debugger (gdb)
- make files

I spent a great deal of time learning how to use these tools for the first time and I'm sure it slowed me down from making more meaningful progress. The ReadMe that accompanied the code assumed all of this knowledge and was rather vague when it came to what needed to happen for the code to build.

5 Next Semester

The build process has been cleaned up and the GUI is displaying on solaris. The next step is to debug the faults that are occurring for certain menu items and try to get it work on Linux by configuring the display class to be detected dynamically (instead of being hard-coded as described in Section 3). After that is out of the way we can start implementing the plan described in Section 3.3. As mentioned above, an issue that needs further analysis is how to reliably test and regression test GUI's. This will be an important part of our refactoring process, but we will need more than just GUI tests. Ideally we would have a suite of tests that verified the functionality of the code separate from the GUI. If we do a good job with pulling out the display code, testing the GUI will just be a matter of making sure the data our functions calculate is being displayed properly. Checking the calculations themselves will happen in a different set of test cases.

A Appendix

A.1 ISVis Statistics

Library	Package	Class	ISVis Reference	Frequency
X11	Instrinsic	ArgList	xapp.C	1
			xapp.H	1
			Total	**2**
X11	X	Drawable	xapp.C	10
			xapp.H	10
			Total	**20**
X11	Instrinsic	Pixel	scenario_view.C	3
			xapp.C	6
			xapp.H	2
			Total	**11**
X11	Instrinsic	Widget	actor_list_view.H	2
			io_shell.C	3
			io_shell.H	4
			main_view.H	22
			mural.H	9
			scenario_view.C	43
			scenario_view.H	36
			view.H	5
			xapp.C	15
			xapp.H	14
			Total	**153**
X11	Instrinsic	WidgetClass	xapp.H	1
			Total	**1**
X11	X	Window	actor_list_view.H	1
			main_view.H	1
			mural.C	1
			mural.H	1
			scenario_view.H	1
			xapp.C	4
			xapp.H	3
			Total	**12**

X11	Instrinsic	XtAppMainLoop	xapp.H	1
			Total	**1**
X11	Instrinsic	XtCallbackProc	actor_list_view.C	1
			io_shell.C	1
			scenario_view.C	39
			xapp.H	1
			Total	**42**
X11	Instrinsic	XtPointer	actor_list_view.C	2
			io_shell.C	2
			io_shell.H	2
			main_view.H	38
			mural.H	6
			scenario.H	69
			view.H	5
			xapp.C	3
			xapp.H	4
			Total	**131**
X11	Instrinsic	XtWorkProc	scenario_view.C	1
			xapp.C	1
			xapp.H	1
			Total	**3**
X11	Instrinsic	XtWorkProcId	scenario_view.H	1
			xapp.C	2
			xapp.H	2

			Total	**5**
X11	Xlib	XImage	xapp.C	2
			xapp.H	3
			Total	**5**
X11	Intrinsic	XtAppContext	xapp.C	1
			xapp.H	2
			Total	**3**
X11	Xlib	Display	xapp.C	3
			xapp.H	2
			Total	**5**
X11	Xlib	GC	xapp.C	1
			xapp.H	1
			Total	**2**
X11	Xlib	Visual	xapp.C	1
			xapp.H	1
			Total	**2**
X11	X	Colormap	xapp.C	2
			xapp.H	1
			Total	**3**
X11	X	Cursor	xapp.C	1
			xapp.H	1
			Total	**2**
X11	Xlib	XColor	xapp.C	8
			xapp.H	1
			Total	**9**
X11	Xlib	XFontStruct	xapp.C	1
			xapp.H	1
			Total	**2**
X11	Xresource	XrmValue	scenario_view.C	1
			xapp.C	4
			xapp.H	1
			Total	**6**

Xm	CascadeB.h	xmCascadeButtonWidgetClass	scenario_view.C	6
			xapp.C	1
			Total	**7**
Xm	ToggleB.h	xmToggleButtonWidgetClass	scenario_view.C	2
			xapp.C	2
			Total	**4**
X11	Xlib	XEvent	actor_list_view.C	1
			actor_list_view.H	1
			io_shell.C	1
			main_view.H	1
			mural.H	3
			scenario_view.C	1
			scenario_view.H	1
			view.C	2
			view.H	2
			xapp.C	1
			Total	**14**
			Grand Total	**445**

A.2 Outstanding Issues

1. Segmentation faults not being caught
2. No reliable Makefile

A.3 Time Log

Note: the numbers of the summaries refer to the links on the website at the home of this document.

Week	Hours	Activity
8/25 - 8/31	0.5	Meeting
	2	Finding background reading for the semester.
	2.5	Reading and summarizing [1]
	2	Reading the documents on the website.
Total:	7	
9/1 - 9/7	1	Meeting
	1.5	Reading and summarizing [2]
	8	Reviewing the code
Total:	**10.5**	
9/8 - 9/14	1	Meeting
	2	Reading and summarizing [3]
	2	Met with Angela to get an overview of the code and get some questions answered.
	6.5	Trying to build the code - it is not directly compatible with my Fedora setup.
Total:	**11.5**	
9/15 - 9/21	1	Meeting
	1.5	Reading and summarizing [4]
	3	Gathering metrics on the display related classes.
	3.5	Created a class diagram of the sections of code related to the display.
	3	Still working out issues with the build
Total:	**12**	
9/22 - 9/28	1	Meeting
	3	Reading and summarizing [5]
	2	Reading/Learning some linux commands and tools (ld, gdb, Makefile)
	6	Got the code to build but it is producing a seg fault. I spoke with Angela about it and she thinks it has to do with it not being compatible with my laptop display.
Total:	**12**	
9/29 - 10/5	0	Meeting canceled.
	3.5	Reading and summarizing [6]
	8	Debugging/reviewing the code.
Total:	**11.5**	

10/6 - 10/12	0	Meeting canceled.
	4	Creating a refactoring plan.
	2	Reading and summarizing [7]
	6	Trying to build the code on tampere (instead of my laptop) so we can try to run it on a Sun monitor in the lab
Total:	**12**	
10/13 - 10/19	0	Meeting canceled - fall break.
	2	Reading and summarizing [8]
Total:	**2**	
10/20 - 10/26	0	Meeting canceled.
	1	Building on tampere.
Total:	**1**	
10/27 - 11/2	1	Meeting
	1.5	Reading and summarizing [9]
	3	Got the code built on tampere. A very important goal for the future is to produce a makefile that will work regardless of the operating system (linux/sun).
Total:	**5.5**	
11/3 - 11/9	1	Meeting
	2	Reading and summarizing [10]
	1	Getting familiar with the xhost command. We're going to try to use this to run the code from tampere but display it on gaia (which is what we log in to while using the sun machine)
Total:	**4**	
11/10 - 11/16	1	Meeting
	1.5	Reading and summarizing [11]
	3	Reorganizing the website file structure - this changed all the links so I had to go an correct them all.
	2	Translating time log to html - in so doing, I created a some javascript classes that can be used in the future to input time. View the source on this page to see.
	4	The xhost stuff didn't work (meaning I was still getting a seg fault when running it from tampere using gaia's display) so we'll have to try and get it built on gaia
Total:	**11.5**	

11/17 - 11/23	0	Meeting canceled.
	2	Reading and summarizing [12]
	6	Building the code on gaia - had to install boost in my local directory because I don't have write permissions anywhere else (that I know of). As an aside, the time entries throughout this log that reference building the code may seem unnecessarily long but that is due to the long build time of boost. I've had to rebuild boost on numerous occasions as I was trying to figure out why the (isvis) code was not building.
Total:	**8**	
11/24 - 11/30	1	Meeting
	2	Prepared an outline for the term paper.
	6	Finally got the code to build on gaia but it is still not running properly on the sun machine. It seems to be trapped in a loop somewhere (it prints out the same string over and over) and then it seg faults. Setting a breakpoint in gdb also causes a seg fault for some reason.
	3	Started working on term paper.
Total:	**12**	
12/1 - 12/10	1	Meeting
	16	Working on term paper.
Total:	**17**	

Total Time: 137.5

References

[1] Isvis adaptation project - 2006. http://www.cc.gatech.edu/morale/tools/isvis/2006/isvis.html.

[2] Isvis original home page. http://www.cc.gatech.edu/morale/tools/isvis/original/isvis.html.

[3] Marwan Abi-Antoun and Wesley Coelho. A case study in incremental architecture-based re-engineering of a legacy application. In *WICSA '05: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*, pages 159–168, Washington, DC, USA, 2005. IEEE Computer Society.

[4] Alessandro Bianchi, Danilo Caivano, Vittorio Marengo, and Giuseppe Visaggio. Iterative reengineering of legacy systems. *IEEE Transactions on Software Engineering*, 29(3):225–241, 2003.

[5] Michael L. Brodie and Michael Stonebraker. *Migrating legacy systems: gateways, interfaces & the incremental approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995.

[6] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Softw.*, 7(1):13–17, 1990.

[7] R. A. Doblar and P. Newcombe. Automated transformation of legacy systems. *CrossTalk*, December 2001.

[8] Marting Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[9] Jan Hannemann and Gregor Kiczales. Overcoming the prevalent decomposition in legacy code. *ICSE 2001 Workshop on Advanced Separation of Concerns*, May 2001.

[10] Rob Leitch and Eleni Stroulia. Understanding the economics of refactoring. In *5th International Workshop on Economics-Driven Software Engineering Research (EDSER-5): The Search for Value in Engineering Decisions*, pages 44–49, May 2003.

[11] Jia Liu, Don Batory, and Christian Lengauer. Feature oriented refactoring of legacy applications. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 112–121, New York, NY, USA, 2006. ACM.

[12] Alok Mehta and George T. Heineman. Evolving legacy system features into fine-grained components. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 417–427, New York, NY, USA, 2002. ACM.

[13] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Using a goal-driven approach to generate test cases for guis. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 257–266, New York, NY, USA, 1999. ACM.

[14] Atif M. Memon and Mary Lou Soffa. Regression testing of guis. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 118–127, New York, NY, USA, 2003. ACM.

[15] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1992.

[16] Andy Pols and Chris Stevenson. An agile approach to a legacy system. In *5th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2004)*, pages 123–129, 2004.

[17] Dave Thomas. Agile evolution towards the continuous improvement of legacy software. *Journal of Object Technology*.

[18] Giuseppe Visaggio. Ageing of a data-intensive legacy system: symptoms and remedies. *Journal of Software Maintenance*, 13(5):281–308, 2001.

[19] L.J. White. Regression testing of gui event interactions. *Software Maintenance 1996, Proceedings., International Conference on*, pages 350–358, Nov 1996.

[20] Bing Wu, Deirdre Lawless, Jesus Bisbal, Ray Richardson, Jane Grimson, Vincent Wade, and Donie O'Sullivan. The butterfly methodology: A gateway-free approach for migrating legacy information systems. In *ICECCS '97: Proceedings of the Third IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '97)*, page 200, Washington, DC, USA, 1997. IEEE Computer Society.

[21] Yijun Yu, Yiqiao Wang, John Mylopoulos, Sotirios Liaskos, Alexei Lapouchnian, and Julio. Reverse engineering goal models from legacy code. In *RE '05: Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE'05)*, pages 363–372, Washington, DC, USA, 2005. IEEE Computer Society.