THE APPLICATION OF TECHNICAL DEBT MITIGATION TECHNIQUES

TO A MULTIDISCIPLINARY SOFTWARE PROJECT

by

Rachael Lee Luhr

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

April 2015

DEDICATION

This thesis is dedicated to my parents, Mary and Donald Luhr. It was through their guidance and unwavering support that I choose to study Computer Science and to continue into graduate school. From practicing long division in the pick-up truck before elementary school to programming a robot for my sixth grade science fair project, they were always encouraging me to pursue my interests and follow my dreams. I hope I can continue to inspire others in the same way.

ACKNOWLEDGEMENTS

TABLE OF CONTENTS

TABLE OF CONTENTS – CONTINUED

LIST OF TABLES

LIST OF FIGURES

viii

LIST OF EQUATIONS

NOMENCLATURE

DB – Database
DBMS – Database Management System
CAHN – Complex Adaptive Hierarchical Network
CERG – Computational Ecology Research Group
IDE – Integrated Development Environment
LOC – Lines of Code
LRES – Land Resources and Environmental Sciences
MAS – Multiagent System
MV – Modularity Violation
NEO – Network Exchange Objects
PT – Particle Tracking
SE – Software Engineering
SEL – Software Engineering Laboratory
SQL – Structured Query Language
TD – Technical Debt
UML – Unified Modeling Language

ABSTRACT

The research described by this thesis uses contributions made to the technical debt community to create a high quality multidisciplinary software project under collaboration between computer scientists and hydrologists. Specifically, additions to the body of knowledge regarding technical debt and modularity violations are described. Technical debt is a metaphor borrowed from the financial domain used to describe the sacrifices that developers make in order to get software released on time. We looked at the uncertainty associated with technical debt measurements and expanded on well-known equations by investigating how errors propagate. We also looked at how modularity violations affect the overall architectural quality of a large-scale industrial software system. Modularity violations occur when modular pieces of code that are not meant to change together, do change together.

The second portion of the thesis applies the research learned from modularity violations and from the uncertainty investigations in technical debt measurements to a specific problem in hydrology to create a more accurate, modularized, and extensible particle tracking algorithm. We used SonarQube's technical debt software to further investigate technical debt measurements. We then visualized the modeling output from the particle tracking algorithm using high-tech digital theater software that was extended to accurately represent natural science visualizations. Finally, we describe the design necessary to seed the application of multiagent system theories and technologies to improve 3D hydrologic modeling.

INTRODUCTION

Motivation

Software projects are often the work of collaborations –field and industry domain experts approach computer scientists to solve problems. The need for the work summed up in this thesis was made apparent by hydrologists from the Department of Land Resources and Environmental Sciences at Montana State University. Hydrologists often collect large amounts of data from field work; which requires significant amounts of pre-processing and organization in order to accurately represent said knowledge. A well architected system forms a foundational framework that can be used to build extensibility through new approaches and explicability of results through behaviorally deterministic algorithms. This foundation is a critical requirement when interpreting large amounts of data. Simulation and modeling practices are widely used by hydrologists as an approach to quickly validate results and contrast hypotheses that require the evaluation of large amounts of data. By architecting high quality extensible architectures that use systematic approaches to carry out experiments we have an opportunity to help. The work herein uses computer science (and specifically, software engineering) theories and technologies to fill the needs of those hydrologists.

Summary of Approach

To address the hydrologists' problem of understanding their data, this work follows a two-prong approach. The first step is to enhance and improve the idea of

particle tracking by designing and implementing an efficient and accurate particle tracker algorithm that is closely linked with the currently used modeling framework. Particle tracking refers to the modeling of artifacts that can be aggregated to form a currency. A currency is a collection of artifacts that flow through a system according to some step function (typically time). The particles that flow through the system can be abstracted to be any token of interest; however, in this case study, it is limited to water particles. The latter are simple agents that traverse a graph (representative of their setting of use) and are commanded externally. A particle that is externally commanded can be thought of as a simple agent, and a particle that can make its own decisions can be thought of as smart. Whilst the current work focuses on simple agents, the algorithm is also on track to use multiagent systems theory to create more intelligent particles. Intelligent particles are able to make informed decisions and process information instead of relying on external programs.

The second step in the approach is to visualize the data received from the particle tracking algorithm. Visualization is a very powerful and useful tool in many areas, not just hydrology. However, when hydrologists can see how water is moving through a watershed, it provides insight to what hydrologic and/or non-hydrologic processes are taking place in that watershed. This is an invaluable tool when understanding such complex data.

While developing an implementation of the algorithm, it was essential to work under the guiding principles that drive high quality in software engineering. Two relevant areas that affect the quality of architectures, and that form a significant contribution to

this work, were studied. Technical debt and modularity violations are considered to be important characteristics of high quality systems. Both characteristics formed a precursor to beginning this project and were fundamental to its success. Research was conducted to advance the science by identifying ways to minimize the impacts of technical debt [1] and modularity violations [2] in the context of developing the particle tracking algorithm enhancements and its complementary visualization techniques [3].

Summary of Contributions

- Increasing knowledge in the uncertainty involved in technical debt measurement.
- Observing how modularity violations affect the architectural quality of large-scale industrial software.
- Developing a more accurate, modularized, and extensible particle tracking algorithm.
- Visualizing modeling output using high-tech visualization software that has been extended to accurately represent natural science visualizations.
- Designing and laying the framework to start applying multiagent system theories and technologies to improve 3D hydrologic modeling.

Organization

The rest of this thesis is organized as follows: Chapter 2 discusses related work involving technical debt and modularity violations in software projects. Chapter 3 provides background information on the Network Exchange Objects modeling

framework and why it is useful in the field of hydrology. Chapter 4 describes the approach used when developing the particle tracker program and how it operates with the modeling framework. The work done with the visualization of models is presented in Chapter 5. Chapter 6 outlines the current and future research to be done in this area. Finally, Chapter 7 provides the conclusions of this work.

RELATED WORK

Uncertainty in Technical Debt

The understanding of the principles behind technical debt was critical in shaping this research. In order for developers to create valuable software while adhering to the best software engineering principles, technical debt must be understood and be at the forefront of design. If technical debt is accrued, they must also understand how much technical debt exists and how to eliminate it. In prior work [1], we discuss the uncertainty involved when measuring technical debt. The following two sections provide an abridged description of technical debt and why it is so difficult to measure.

What is Technical Debt?

Coined in 1992 by Ward Cunningham [4], the term ―technical debt" is a metaphor to describe the sacrifices that developers make in order to get software released on time. This metaphor is borrowed from the financial industry where sometimes, one needs to take on debt in order to advance. Like financial debt, when software needs to be refactored or redesigned, it costs extra time and money to make these fixes. However, sometimes incurring technical debt (like financial debt) is necessary. Martin Fowler [5] has shown that there are four main types of technical debt that can occur. The quadrant shown in Figure 1 illustrates these types of debt.

Reckless | Prudent

"We don't have time
for design"

"We must ship now
and deal with
consequences"

Deliberate

Inadvertent

"What's Layering?"

"Now we know how we
should have done it"
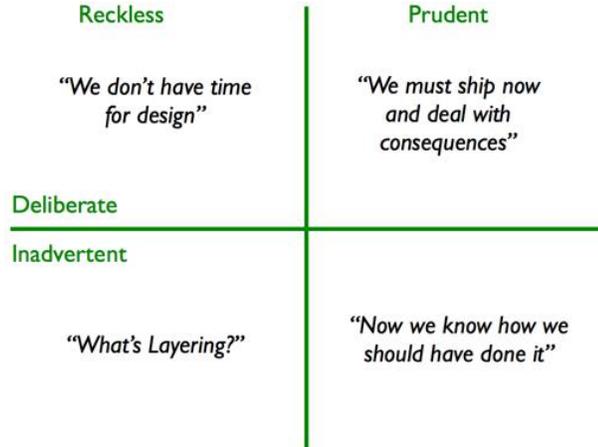
Figure 1. Technical Debt Quadrant [5].

It is especially difficult to measure and subsequently remove inadvertent technical

debt. This occurs when the developers are making poor design decisions and are not even

aware they are doing so. In cases when developers know they are acquiring technical

debt, they are doing it deliberately. As the quadrant shows, this can be done recklessly or

prudently. The ―best" kind of technical debt occurs deliberately and prudently. It must be

stated that technical debt is not always incorrect and is sometimes even necessary.

Developers must ensure that if they are taking on technical debt, it needs to be done in a

way that can be managed and understood. That way, like Seaman et al. mention in [6],

technical debt data can then be used in important decision making strategies such as when

(or if) a company needs to refactor its code.

How Do We Measure TD?

As stated in the previous section, technical debt is very difficult to measure.

Curtis et al. [7] state that ―there is no exact measure of Technical Debt, since its

calculation must be based only on the structural flaws that the organization intends to

fix." However, many organizations may not be aware of the technical debt in their software or may not want to fix the technical debt they are aware of.

In [8], Ferenc et al. describe three types of software quality assessment models. These models are useful because they can provide developers with quantitative information to assess the quality of their software. The three types of models are: Software Process Quality Models, Software Product Quality Models, and Hybrid Software Quality Models. The difference between the first two types of models is that the first attempts to measure the *process* of software and the second attempts to measure the *product* (that is, the software system itself). These types of models measure the quality of the product by combining different source code metrics.

Following this, there have been several different proposed ways to measure technical debt. Griffith et al. [9] performed a study to examine the relationship between several different methods of measuring technical debt and an external software product quality model. Several of these technical debt estimation methods will be discussed in the following paragraphs.

In 2012, Letouzey and Ilkiewicz introduced the Software Quality Assessment based on Life-cycle Expectations (SQALE) method [10]. SQALE is based on four main concepts: the quality model, the analysis model, the four characteristics (maintainability, changeability, reliability, and testability), and the indicators. Table 1 describes the characteristics and how to solve the problems related to these characteristics. The authors give a remediation function that describes approximately how long it would take to fix

these problems. However, developers are free to change these functions based on their

particular skills.

Table 1. SQALE method of measuring technical debt [10].

Characteristic	Requirement	Remediation microcycle	Remediation function
Maintainability	There is no commented-out block of instruction	Remove (no impact on compiled code)	2 minutes per occurrence
	Code indentation shall follow a consistent rule	Fix with help of the integrated development environment (IDE) feature	2 minutes per file, regardless of the number of violations
Changeability	There is no cyclic dependency between packages	Refactor with IDE and write tests	1 hour per file dependency to cut
Reliability	Exception handling shall not catch null pointer exception	Rewrite code and associated test	40 minutes per occurrence
	Code shall override both equals and hash code	Write code and associated test	1 hour per occurrence
	There is no comparison between floating points	Rewrite code and associated test	40 minutes per occurrence
	No iteration variable are modified in the body of a loop	Rewrite code and associated test	40 minutes per occurrence
	All files have unit testing with at least 70% code coverage	Write additional test	20 minutes per uncovered line to achieve 70%
Testability	There is no method with a cyclomatic complexity over 12	Refactor with IDE and write tests	1 hour per occurrence if measure is < 24; 2 hours if > 24
	There are no cloned parts of 100 tokens or more	Refactor with IDE and write tests	20 minutes per occurrence

In 2011, the CAST Research Labs (CRL) published a report that highlighted the

fact that companies are not budgeting for the millions of dollars' worth of technical debt

that exists in their software products [11]. This report used the following equation to

estimate the principal of technical debt in dollars, where principal is defined as the cost of

fixing problems remaining in the code after it has been released.

Estimated Technical Debt Principal =
$((((\sum \text{high severity violations}) \times \% \text{ to be fixed}) \times \text{avg hours to fix}) \times \$ \text{ per hour})) +$
$((((\sum \text{med severity violations}) \times \% \text{ to be fixed}) \times \text{avg hours to fix}) \times \$ \text{ per hour})) +$
$((((\sum \text{low severity violations}) \times \% \text{ to be fixed}) \times \text{avg hours to fix}) \times \$ \text{ per hour})) +$
Equation 1. CAST Equation for Estimating Technical Debt [11].

The authors of this equation purposefully leave the parameters adjustable so that each customer or company can customize the equation to fit their specific needs. The problem with this approach is that it leaves room for error if companies are unsure what values to use for the parameters.

Uncertainty in Calculations

Due to human factors, uncertainty does exist in measurements of technical debt. As the methods in the previous sections stated, there is no hard rule or set standard for how to calculate technical debt. In [1], we use techniques from physical sciences (like those discussed by Taylor [12]) and apply them to software engineering technical debt measurements. We use Equation 2 to calculate the measure of technical debt principal by using the developers best estimate of technical debt principal and adding (or subtracting) a margin of error or uncertainty.

$$measured\ value\ of\ TD_{principal} = (TD_{principal})_{best} \pm \varphi_{TD}$$
Equation 2. Technical Debt equation including error [1]

Comparing Measures. Unlike the physical sciences, where we can use multiple physical tools and compare them to calibrate a measurement, tools that measure technical debt cannot be compared because measurement of technical debt is still in its infancy and we have not yet agreed on a common metric –all approaches use different equations. Using calculations that are unadjusted for error are uninteresting. By providing a measure of uncertainty along with the technical debt measurement, allows scientists to begin moving towards an understanding of the significance that certain factors have on the

response variable (i.e. technical debt); which would allow us to more accurately compare two (or more) measurements.

Propagation of Error. This error is very important to measure because it can propagate through technical debt measurements. In [13], Nugroho et al. propose calculations to measure aspects of technical debt. In [1], Izurieta et al. used these equations to show this propagation of error. They discuss the propagation of uncertainty in sums, differences, products, and quotients of measured quantities.

Multivariate Uncertainty. Taylor [12] uses quadrature in formulas that need to deal with multivariate equations. This method is appropriate to use when the measurements come from Normal or Gaussian distributions and are independent. While this is a good place to begin investigating, this type of calculation needs more validation within the field of software engineering.

Until we have agreed upon standards or tools to accurately measure technical debt, it is important to report the corresponding error in technical debt measurements. Additional work in this area is critical to further the understanding of technical debt and how it affects software projects.

<div align="center">Modularity Violations</div>

Technical debt can appear in code in many different ways. In [14], Izurieta et al. evaluate four different types of debt and approaches to examine and mitigate this debt. This work is also extended in [15] to include more recent studies and findings. The four

indicators of technical debt discussed include design patterns and grime buildup [16] [17], code smells [18], ASA (automatic static analysis) issues [19] [20], and modularity violations [21]. All four of these areas deserve further attention so that we can gain more insight into technical debt. Their findings suggest that there exist significant gaps in technology, and that each technique is designed to measure different aspects of technical debt. At an architectural level, modularity violations are more relevant to the work necessary to extend this thesis. In this section we focus on modularity violations and their importance in the technical debt landscape.

<u>What are Modularity Violations?</u>

Baldwin and Clark [22] define a module as —a unit whose structural elements are powerfully connected among themselves and relatively weakly connected to elements in other units." Modularity violations occur when two modules that are not expected to change together do change together. These violations are very important to recognize because understanding their consequences can lead to better design decisions and/or highlight the need for refactoring, thus reducing technical debt. However, early detection of modularity violations can be very difficult because their influence in the code is not always immediately apparent.

In [21], a tool called CLIO was designed and tested to detect modularity violations. We conducted a replication case study [2] using this tool to test its efficacy at correctly detecting modularity violations and to increase the knowledge base surrounding modularity violations and their effect on technical debt.

Importance of Replication in SE

Experimental replication studies are important in a field such as empirical software engineering [23] [24] because they help build consensus around emerging theories. Observing and studying software projects often, if not always, also involve the study of humans. Human behavior can be extremely unpredictable and difficult to replicate. This makes conducting a controlled experiment very challenging. By performing replication case studies, we can increase the confidence of ideas.

The replication of case studies, such as the one performed by Reimanis et al. [2], are not as common as replications of experiments. However, they are just as important. Case studies take place in the real world (*in-vivo*), and observations are made in the context of their domains. This tends to be typical of software engineering studies because controlled experimentation is cost prohibitive. The observation of historical data (a longitudinal approach) is also common in empirical software engineering studies, and occurs when we observe phenomena over various versions of software. This type of information is invaluable when attempting to understand how projects actually evolve. In our replication study we borrowed terminology from existing information on experimental replication studies [25].

Effects of Modularity Violations in Software

We conducted a replication of a study by Schwanke et al. in [26]. Both projects were industrial software products. Our code base was a commercial software system developed by a local bioinformatics company – Golden Helix [27]. They gave us access to their code base because they were interested in learning about potential modularity

violations in their designs. They specifically wanted us to point out potential deficiencies in their code organizational structure.

We had five major treatment differences from the original study. Table 2 summarizes these differences. The first major difference is the programming language that the projects were written in. The baseline project was written in Java and our project was written in C++.

Table 2. Summary of different treatments between case studies [2]

Factor	Baseline Project	Our Project
Programming Language	Java	C++
# of Developers	Up to 20	Up to 11
Project Lifetime	2 years	4 years
# of Source Files	900	3903
KSLOC	300	1300

The difference of treatments in this factor brought about interesting complications in the study because Java projects are structured differently than C++ projects. Because of this, we had to slightly modify our definition of a module. In this study, we defined a module as a directory. This choice was based on Parnas et al.'s definition [28]. The terms module and directory are used interchangeably. We also had to group C/C++ source files and header source files together. This is because developers expect source files and their related header files to change together. Our study was only concerned with unexpected changes across modules.

The other differences between the baseline project and our project include the number of developers, project lifetime, number of source files, and kilo-source lines of

code (KSLOC). Our project had fewer developers, but a longer lifetime and is larger in terms of source files and LOC.

In order to gather data on this project, we followed the work of Schwanke et al. [26] and looked at seven different metrics; which were gathered for all file pairs across all seven versions of Golden Helix's software. These metrics included: file size, fan-in, fan-out, change frequency, ticket frequency, bug change frequency, and pair change frequency. The definitions of these metrics can be found in [2]. We used CLIO to gather measurements for these metrics by looking at the source code and the version history of the project. Following the baseline study, Kendall's *tau-b* rank correlation measure was used [29]. Table 3 shows the *tau-b* value calculated for each metric pair in releases 7 and 7.5 of the software.

Table 3. *Tau-b* values for metric pairs [2]

R7+R7.5	Fan-in	Fan-out	File size	Changes	Tickets	Bugs
Fan-in	1	0.257	0.301	0.331	0.328	0.464
Fan-out	0.257	1	0.441	0.417	0.416	0.637
File size	0.301	0.441	1	0.293	0.273	0.510
Changes	0.331	0.417	0.293	1	0.972	0.858
Tickets	0.328	0.416	0.273	0.972	1	0.857
Bugs	0.463	0.637	0.510	0.858	0.857	1

Highlighted cells with a value of 0.6 or greater indicate strong correlation [30]. Most of the highlighted results were expected. Changes, bugs, and tickets have significant correlation values. An unexpected result was the correlation between bugs and fan-out. This number shows that as the fan-out of a file pair increases, the number of bugs associated with that pair also increases. These results are consistent with the baseline case

study [26], adding to the knowledge of how modularity violations occur and how well CLIO detects them.

Developers at Golden Helix were unsurprised by the findings of our study. Many of the files that were causing modularity violations were known to be heavily dependent on other files, and some modularity violations were intentional. However, it was reassuring for the baseline developers to receive this information because it meant that there were few modularity violations occurring that they were not expecting. The latter exemplify prudent and deliberate technical debt.

BACKGROUND

Network Exchange Objects

Network Exchange Objects (NEO) is a software framework designed to facilitate development of complex natural system models [31] where models are represented as graphs that can carry and communicate data represented as flows of currencies. NEO began a re-engineering phase of development at Montana State University in 2009 as a joint venture between the Computer Science Department and the Department of Land Resources and Environmental Sciences (LRES). NEO can be used as a general-purpose modeling tool applicable to many domains [32].

What is NEO?

NEO was designed to study systems that can be described as ―complex adaptive hierarchical networks" (CAHNs). CAHNs are complex systems through which information is stored and routed and can be represented in a graphical form. CAHNs are constructed of cells that are linked by edges and are structured hierarchically. Cells represent system components. These cells can be defined as any physical component of a system. This can range from discrete sections of a riverbed (e.g., a model of a watershed) to an abstract representation of cars (e.g., a vehicular communication network). Any conceptual structural component the modeler is interested in can be represented as a cell.

The edges in a CAHN represent the interaction between cells. This is where the information about the behavior of exchanges between cells is stored. For example, in a watershed model, the interaction between cells might include the flow of water, exchange

of heat, or flow of sediment. In a vehicular communication model, the modeler might be

interested in the signals being exchanged between cars. The behavior (described as

calculations) for how these exchanges occur, is located in the edges of the graph.

There are two faces associated with each edge. Edges can contain behavior that is

synchronous; that is, the behavior of a face on one side of the edge is reflected on its

counterpart on the opposite face (e.g., the signals between two communicating cars), or

asynchronous, where each face has its own unique behavior (e.g., the flow of sediment in

a riverbed). This is illustrated in Figure 2. Having ―to‖ and ―from‖ sides of edges helps

distinguish the flow of information.
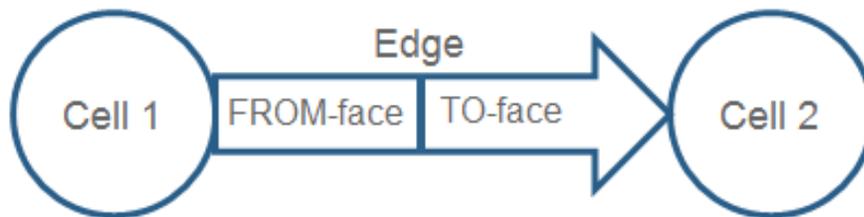
Figure 2. How cells are connected in a NEO model [33].

Within this graphical representation of a system model, currencies flow from the

―from‖ side of an edge to the ―to‖ side. A currency represents anything within the model

that is being exchanged between components of the modeled system (e.g., radio signals,

water, sediment, energy, economic capital, nutrients, or any other resource that is of

interest to the system). These currencies are then manipulated as they flow (or flux)

between cells and edges in the defined matrix network, representing the effect entities

have on the flow. For example, if Figure 2 was describing a watershed model where the

cells are discrete patches of the river and the behavior located in the edges describes the flow of water (i.e., the currency) between patches, then the water would be flowing downriver from Cell 1 to Cell 2.

There are several vocabulary words that are used in conjunction with NEO [33]. The following is an abridged selection of key definitions.

- Holon – Cells, edges, and/or faces that correspond to real-world components of the modeled system.

- StateVal – A variable within a holon that can be altered.

- Dynam – (Auto/Manual/Init) A static or dynamic method (i.e., an algorithm) in which a stateVal is manipulated. This is where the currency behavior is defined.

- Currency Package – A package written in the Java programming language that defines the behavior of a currency through a set of dynams.

- Model – A combination of the NEO framework and the necessary currency packages which are to represent the desired system.

How Does NEO Work?

The core algorithm of NEO uses a Trade-Store-Update cycle approach. This process determines the order of operations for the dynam calculations. In the Trade phase, currencies that are fluxed between cells are calculated within each edge and their values are saved to the currency stateVals within that edge. In the Store phase, dynams that are located in cells update their currency stateVals based on the values that were calculated in the Trade phase. Finally, in the Update phase, all automatically refreshing values update their stateVals.

To develop and execute a NEO model, the following software requirements are necessary. First, a development environment of the Java programming must be installed on the machine. Second, an Integrated Development Environment (IDE) must be used. IDEs provide a platform that allows the organization of files that correspond to NEO concept implementations. Whilst not essential, it provides a significant aid when managing models. Third, a database must exist in order to store information about the model inputs and outputs.

<u>Hydrology</u>

One natural science field that lends itself well to the NEO modeling framework is hydrology - specifically, the study of movement, distribution, and quality of water. Water flowing in a river exemplifies NEO functionality: the river is the structure that makes up the network and the water is the currency that fluxes through the system.

<u>Watershed Modeling</u>

The need for a system like NEO was made apparent by hydrologists from the LRES Department at Montana State University. Simulation modeling allows scientists to test many hypotheses in a relatively cheap environment, providing significant insights into specific factors that can then be further explored and validated through real world field studies, which are more expensive. Therefore, there are great advantages in hydrological modeling. Much of the research in this area is done to improve scientist's ability to predict or forecast the effects of land-use and climate change on the water balance, streamflow variability, and water quality. Hydrologists are particularly interested

in the flow of water through a watershed. According to Dingman [34], a watershed is

defined as the ―area that appears on the basis of topography to contribute all the water

that passes through a given cross section of a stream." An example of watershed

delineation is shown in Figure 3.
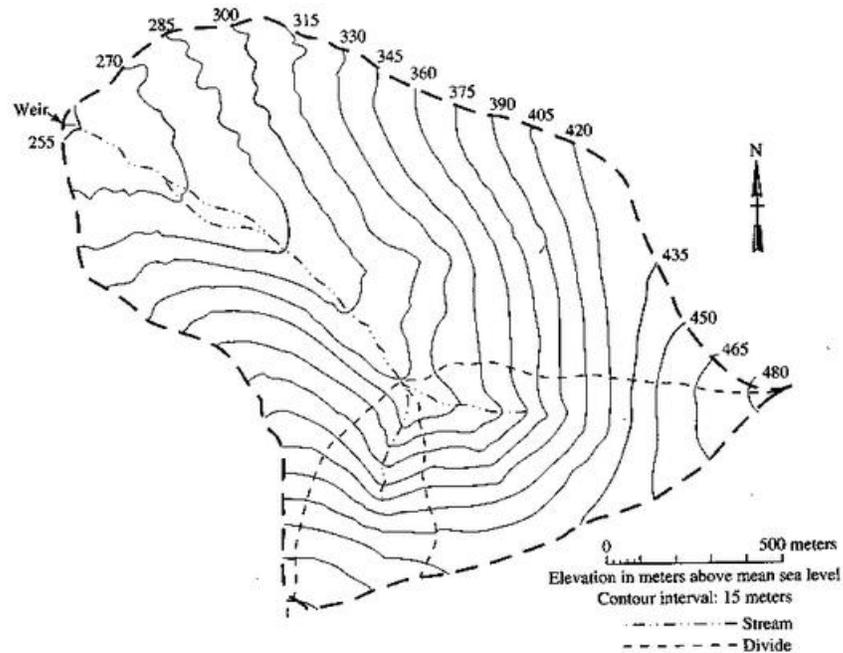
Figure 3. Watershed delineation on a topographic map [34].

The floodplain of interest (i.e., experimental unit) of this study is the Nyack

Floodplain located in Western Montana near Flathead Lake. The Nyack Floodplain

includes the watershed of the middle fork of the Flathead River. This area has been the

focus of considerable research in the state of Montana and the surrounding areas. This

research spans many topics including migratory patterns of bull trout [35], how the river

affects grizzly bear diets [36], and the structure of the river itself [37]. In 2012, Helton et

al. studied the temperature and dissolved oxygen dynamics of this system [38]. After the

implementation of NEO, this work was extended and provided the basis for the data used

in this thesis.

Helton et al. [39] were especially interested in the residence time of water within

the Nyack floodplain. Rivers are composed of surface and subsurface flow. Water that

becomes part of the subsurface (also known as ground-water) generally has longer

residence time. The topography of the landscape creates an immense influence on the

movement of water [40]. This leads to the idea of breaking up the watershed into smaller

blocks where water acts similarly within those blocks. Figure 4 shows how the river

breaks down into these separate components.
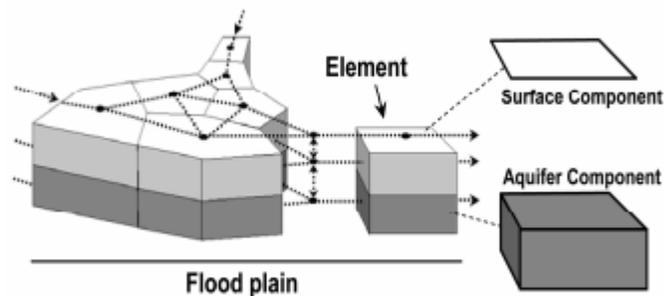
Figure 4. Inter-connected components of a river [41]. Figure Copyright © 2004
John Wiley & Sons, Ltd. Reproduced by permission.

In order to model this watershed in NEO, the river was broken down into discrete

patches. The ―cells" represent patches and are depicted in Figure 4 as nodes drawn in the

center of a corresponding polygon. The ―edges" are drawn as straight lines connecting the

cells. There are three layers in this watershed, as is indicated in Figure 4 (the surface

layer, the hyporheic layer, and the aquifer layer). This allows for representation of

horizontal water flow, horizontal and vertical subsurface flows, and the vertical

exchanges between subsurface and surface waters.

APPROACH

Particle Tracker Program Design and Development

To visualize how water flows through a floodplain, hydrologists use a method known as particle tracking. Particles can be thought of as independent agents that flow through the floodplain and behave according to domain specific equations. Agents can be configured to report on information according to predefined criteria. In the case of particles they are configured to output their position at various time steps. This output can then be used to gain insight on how water may actually be moving. The design of the tracking algorithm uses current and modern object oriented techniques that encompass high quality characteristics (i.e., low technical debt, use of design patterns, and use of object oriented design principles). The implementation is written in the Java programming language.

Technical Requirements

There are three technologies necessary to run the particle tracking software: the Java 8[1] developer's kit and runtime environment, an Integrated Development Environment (IDE), and a database. The development of our project used Eclipse 4.2.1 Juno[2] and PostgreSQL 9.2[3] as the Database Management System (DBMS) with full support for the Structured Query Language (SQL).

---

[1] https://java.com/en/download/
[2] http://www.eclipse.org/downloads/
[3] http://www.postgresql.org/download/

Output from a NEO model run is stored in a database. The particle tracking

program uses the information stored in the database tables to make informed decisions on

how to move the particles. The database contains two tables. One includes the

information to form the matrix (or the structure) of the river system. A portion of this

table is shown in Table 4. It consists of four columns that must be named: id, from_id,

to_id, and length. _id' is a unique integer that corresponds to the id of the link connecting

two nodes. _from_id' is an integer that corresponds to the id of the originating node that a

link is coming from. _to_id' is an integer that corresponds to the id of the destination

node that a link is going to. _length' is a value of type double that corresponds to how

long the link is (in meters). Links are directional because direction can be critical in most

flux networks. For example, we do not want water flowing –up" river.

Table 4. The first five rows of the matrix database table.

	id bigint	from_id bigint	to_id bigint	length double precision
1	1001	264	218	1045.9825
2	1002	262	264	472.115
3	1003	113	264	783.251
4	1004	113	218	740.3944
5	1005	262	208	323.2793

The information in this matrix table forms four layers of the floodplain. These

layers are surface water, shallow groundwater, deep groundwater, and soil. This data was

pulled into the R statistical software[4] and plots were created for each layer. These images

can be easily compared to the visualization to ensure that the data was consistent across

all programs (NEO, the particle tracker, and the visualization software). Figure 5 shows

---

[4] http://www.r-project.org/

the R plots for the soil links, deep groundwater links, shallow groundwater links, and surface water links.
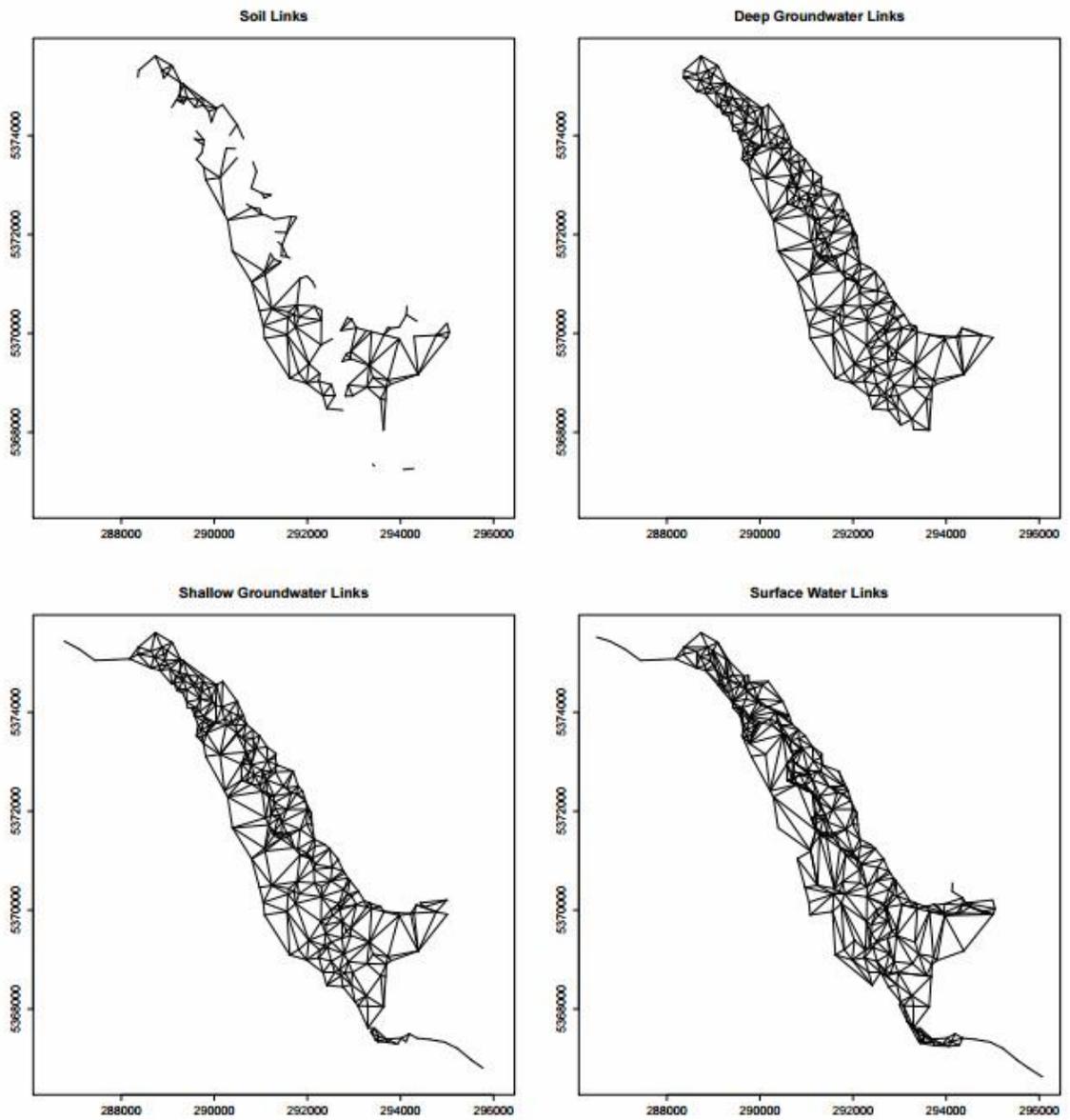
Figure 5. All four layers of links as plotted by R. (Provided by G. Poole)

The second database table, partially shown in Table 5, provides the results from the NEO model run. This table also consists of four columns: secs, uid, water, velocity. ‗secs' is the time (in seconds) associated with that particular line of output. This instantiation of the model run output every 43,200 seconds, or 12 hours. ‗uid' is the link id. This value corresponds to the link id from the matrix information table. The combination of ‗secs' and ‗uid' is what makes a row in the table unique. ‗Water' is the flux value of water flowing through that link at that particular time. ‗velocity' refers to how quickly that water is moving down the link. The equations that were used to represent the physical principles behind the water flow are described by Walton et al. [42] and Poole et al. [41].

Table 5. Five rows of the model results database table.

	secs double precis	uid bigint	water double precision	velocity double precision
28	1166400	1001	0.006161031	0.000351112
29	1209600	1001	0.005761998	0.000330817
30	1252800	1001	0.005640002	0.000325993
31	1296000	1001	0.00628345	0.000365929
32	1339200	1001	0.006754953	0.00039713

System Design

As mentioned in the previous section, the NEO output tables must be located in a database and, along with a specification file, provide the input for the particle tracker program. The program also outputs to a database table located in the same DBMS as the NEO output tables. Figure 6 shows the UML component diagram for the particle tracker program.
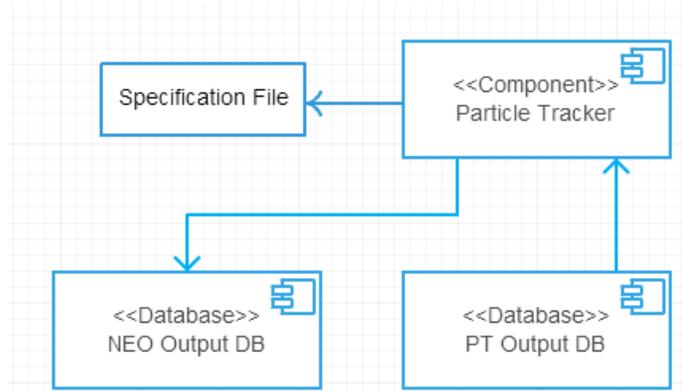
Figure 6. UML Component Diagram for the Particle Tracker program

The specification input file is where the user specifies how many ―particles" they would like released, where they would like them released, and how often they want the particles to report on their current location. Figure 7 displays this specification file (in this case called RiverInfo.txt). The first line, DataTimeInterval, is time interval that coincides with the NEO output data. The ReleaseInfomation contains the cell in the matrix (referred to in this context as a node) where the user wants the particles released, how many particles the user wants released, and the release time for those particles, respectively. The third line in the file specifies when to start reporting particle tracker information. The user may not always want to start reporting at the beginning of the model, so this allows for flexibility. The ReportInterval on line 4 describes how often to report output from the particle tracker. The accuracy of a simulation improves as this report interval decreases. Finally, the last line, Destination, describes cells (or nodes) in the matrix where the user wants to see the particles will exit the simulation (in this case, as the furthest downriver point of the floodplain.
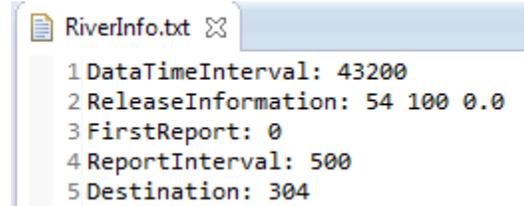
Figure 7. Specification file for the Particle Tracker program

Once the environment and the specification file are correctly specified, users may run the particle tracker program by selecting the Main.java class in Eclipse (or IDE of choice) and pressing the run button.

The Particle Tracker Algorithm

The movement of particles is based on available links, velocity, and flux. Particles start by moving along the links connected to the specified nodes. To move, particles query the database for the velocity along that link at that particular time. The particle then calculates how far it should move down the link based on how much time has passed multiplied by the velocity.

A particle knows it has reached the end of a link when its movement has exceeded the link's length. At arrival to a node, the particle queries the database to see which links are available for selection by observing which links contain the current node as the ―from node‖. Once there is a list of known available links, the particle must choose which one to select to continue its trajectory.

To more accurately represent a real system, a level of randomness is inserted into the choices that particles make when they reach a specific node. The program queries the database to find the flux values for all the available links at the current time. It then uses a

weighted formula to cumulatively sort these links from smallest flux value to largest (on a scale from 0.0 to 1.0). This is compared to a random number generated by the program (also between 0.0 and 1.0). The particle chooses the first link that has a higher weighted flux value than the random number. See Figure 8 for the portion of the code that handles this logic.

```java
Random r = new Random();
double rand = r.nextDouble();
double linkToTake = 0;
for (int j = 0; j < weightedFluxAndLinks.length; j++){
    if (Double.compare(rand, weightedFluxAndLinks[j][1]) < 0){
        linkToTake = weightedFluxAndLinks[j][0];
        break;
    }
}
```

Figure 8. Java code for the randomization portion of the algorithm

At the time step stated on line 4 of the specification file, the particle reports its particle id number, the id of the link it's currently traversing, the current time, and its position on the link. This information is written to a database table in order to provide easy access to the output once the particle tracker is finished. Table 6 shows the first six rows of this output table. You can see that the particle reports every 500 seconds (as specified). When the particle reaches the end of link 1741, it chooses link 1743 as its new link to follow its trajectory.

Table 6. Six rows of the particle tracker program output table

	particleid integer	currentlinkid integer	currenttime double precision	currentlocation double precision
1	0	1741	0	0
2	0	1741	500	64.3622555
3	0	1741	1000	128.724511
4	0	1741	1500	193.0867665
5	0	1743	2000	70.6338047294931
6	0	1743	2500	155.889020229493

Validation

The Nyack watershed model's matrix table contains a total of 3054 lines and its model results table contains a total of 369,534 lines. To validate that the particle tracker algorithm was working correctly, we created a smaller scale model that contained all the possible conditions present in the larger model. In collaboration with hydrology domain experts, the following represent the conditions that required testing:

1. Particles had to choose links correctly based on weighted flux with some element of randomness.

2. If the velocity reaches 0 in the middle of the link, the particle would stop moving, but would keep reporting its position.

3. If a particle comes to a junction where none of the available outgoing links have a flux value greater than 0, the particle will wait at the junction until the flux becomes greater than 0.

4. If a particle comes to a junction and there is a link with positive velocity but the link is the wrong direction, the particle should never choose to go down this link.

The graph constructed to test these conditions is shown in Figure 9. The node ids and link ids are shown on the graph. A model results database table was constructed to serve as an oracle that would have similar output to a NEO model run. This table also had the above conditions built into it. The particle tracker was run with those two tables as input and the specification file tailored to fit this model. The particles were released at node 1, reported every 100 seconds, and expected to arrive at nodes 6, 7, 8, or 9. After the particle tracker finished executing, the output was validated by hand to ensure that the four conditions were met and particles were behaving as expected. See Appendix D for input tables and output from this example.
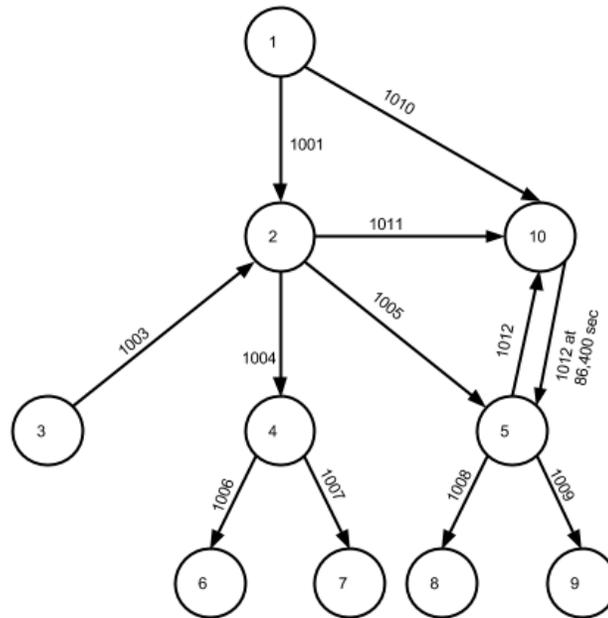
Figure 9. Simplified graph to test particle tracking conditions

Using SonarQube to Measure TD

The particle tracker program was developed with the goal to minimize technical debt. As mentioned in Chapter 2, there are many approaches to measuring technical debt within a software system. To measure the technical debt in the particle tracker system, the SonarQube platform was used [43]. SonarQube evaluates technical debt by examining various potential disharmonies in a system, including: duplications, coding standards, lack of coverage, potential bugs, complexity, documentation, and design. It calculates several scores: a Technical Debt score (measured in man days – how many 8 hour developer days it would take to fix all the issues), the SQALE rating [10] (from A-E, A being the highest), and the Technical Debt Ratio (how much technical debt the project has versus how large it is).

In order to get results from SonarQube, the user must download and run the SonarQube Server (which allows users to view their results online) and the SonarQube Runner (which launches the program and allows for projects to be analyzed). The user must also provide a sonar-properties file. The properties file used is shown in Figure 10. Once the SonarQube Server is running and the SonarQube Runner has been launched and ran against the project with the sonar-properties file, the user can see the output by navigating to http://localhost:9000 in their browser.

```
sonar-project.properties
 1   # Required metadata
 2   sonar.projectKey=particle-tracker
 3   sonar.projectName=Particle Tracker project analyzed with SonarQube
 4   sonar.projectVersion=1.0
 5
 6   # Comma-separated paths to directories with sources (required)
 7   sonar.sources=rachaels
 8
 9   # Language
10   sonar.language=java
11
12   # Encoding of the source files
13   sonar.sourceEncoding=UTF-8
```

Figure 10. Sonar-properties.properties file

The particle tracker program consists of 8 files (or classes), 45 functions, and a total of 802 lines of code (LOC). Figure 11 shows the amount of duplications and the complexity score as calculated by SonarQube.
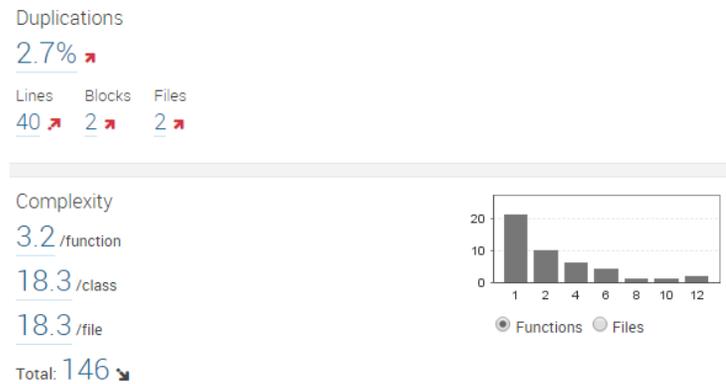
Figure 11. Duplications and Complexity Score calculated by SonarQube [43]

Figure 12 displays the SQALE Rating, Technical Debt Ratio, and the amount of Technical Debt of the program. SonarQube also allows the user to click on the ―Issues" measure to navigate to a dashboard where users can observe what the issues are and

where they are located in the files. This allows for much quicker fixes and provides a

visual tool for the user to see if they are making identical mistakes.
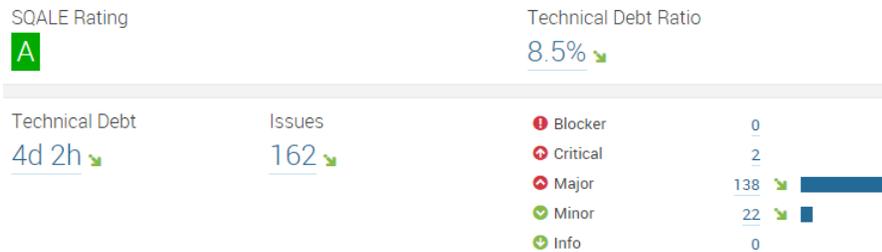
Figure 12. Different technical debt scores calculated by SonarQube [43]

On the SonarQube website, there is also a formula for calculating the cost of

remediating observed technical debt. The default value is parameterized at $500 per

developer day. Using this value, the cost to reduce technical debt in the particle tracker

program would be $2125. However, this value is configurable in the SonarQube

platform. If we chose a different amount for this project, for example, the average pay of

an intern in the Bozeman area ($160/day), the reported cost would be $680.

VISUALIZATION

Motivation

One of the main problems associated with the understanding of complex scientific data is mentally visualizing what processes are occurring. Given this difficulty, many scientists rely on models and visualizations to aid their understanding. The need for visualization is one of the main reasons why the work in this thesis began. One of the best (and most intuitive) ways to portray the hydrological model to a larger audience is with visualization.

Platform

The platform used as a vehicle for these visualizations is Digistar 4 [44], one of the most advanced and successful digital planetarium platforms. It was developed by Evans and Sutherland Company, based in Salt Lake City, UT. Digistar 4 was chosen because it was easily accessible and has a powerful graphics engine that provided the necessary requirements to accurately visualize the particle tracking model. It was also intuitive and user friendly. Figure 13 is a screenshot of the Digistar 4 dashboard.
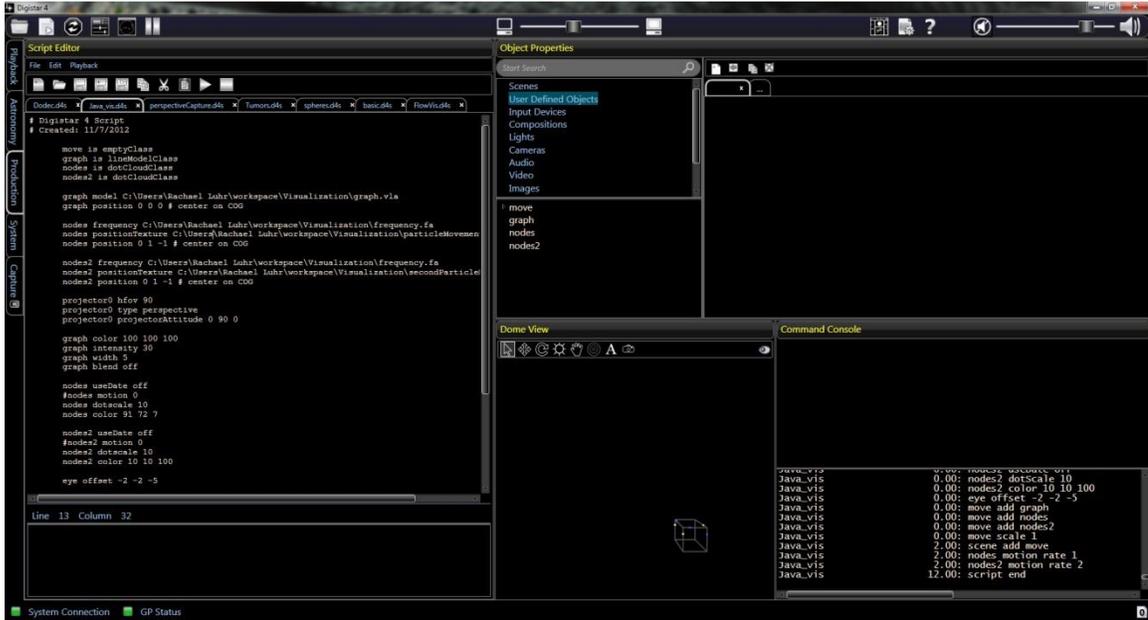
Figure 13. Screenshot of the Digistar 4 dashboard

Outreach

Another large reason of why Digistar 4 was chosen for this project is the potential

it had for outreach. The Taylor Planetarium is part of the Museum of the Rockies, which

in turn is a department of Montana State University. The Museum of the Rockies is the

most visited indoor attraction in Montana, with an annual attendance of 100,000 visitors.

The Taylor Planetarium seats 60,000 of those visitors each year. The Taylor Planetarium

is the only planetarium building in Montana, making it a premier travel destination. The

museum acts as an outreach arm for the land grant university, and it is tasked to educate

all ages, from adults to school children. The planetarium had been recently upgraded

from Digistar 2 to Digistar 5 (the newest version of the software at the time of

publication). The upgrade allowed for visualizations developed on the Digistar 4 platform

to be run in Taylor Planetarium. This spurred the idea of creating an MSU Minute show

to run in the planetarium before a normal planetarium show. The MSU Minute was an idea from the planetarium director to highlight research occurring at MSU[5]. The actual show lasted 2-4 minutes and was shown in front of various audiences for several months. Other departments at MSU had already taken advantage of this opportunity to display their research.

Starting in September 2013, an MSU Minute describing the initial work from this thesis was displayed in front of a planetarium show for approximately six months [45]. This allowed the work done by CERG (Computational Ecology Research Group) and SEL (Software Engineering Laboratory) to be viewed by thousands of visitors to the museum. Outreach in the sciences is important because it allows for a way to portray complex information in a relatable and understandable way to an audience unfamiliar with many of the complicated details.

<u>Approach</u>

To import the particle tracking model into Digistar 4, the information in the NEO database and the output from the particle tracker program had to first be converted into the correct Digistar file formats. These are proprietary formats and not easily readable. A Java program was created to serve as a filter that converts readable data to proprietary information needed for Digistar. The files required by Digistar 4 include: a graph file, which tells Digistar how to set up the model in 3D space; a frequency file, which contains how many particles are fluxing through the system and the duration (measured in time

---

[5] : https://vimeo.com/78744894

steps) that each particle exists within the model; and a position file, which has all the particle location information in 3D space during each moment in time.

Once the model information was in the correct file formats, Digistar 4's scripting language was used to create a mock planetarium show. This allowed the model to be shown in dome view (or visualization view). The scripting language and Digistar's advanced interface provided the necessary functionality to tweak parameters in the model in order to calibrate it to the right position and the correct size. The user can also control the flow rate of the particles, their size, and their color. It is also possible to navigate through the model in real time and to zoom in and out on selected areas. It can be looked at from above, below, or inside. This feature provided many useful options when creating the MSU Minute.

Figure 14 is a snapshot of what the Nyack floodplain visualization looks like in Digistar 4. There is an image in the background of the Nyack overlaid with the structure of the river. The particles are directly following along the edges of the matrix imitating the way that the water would flow.
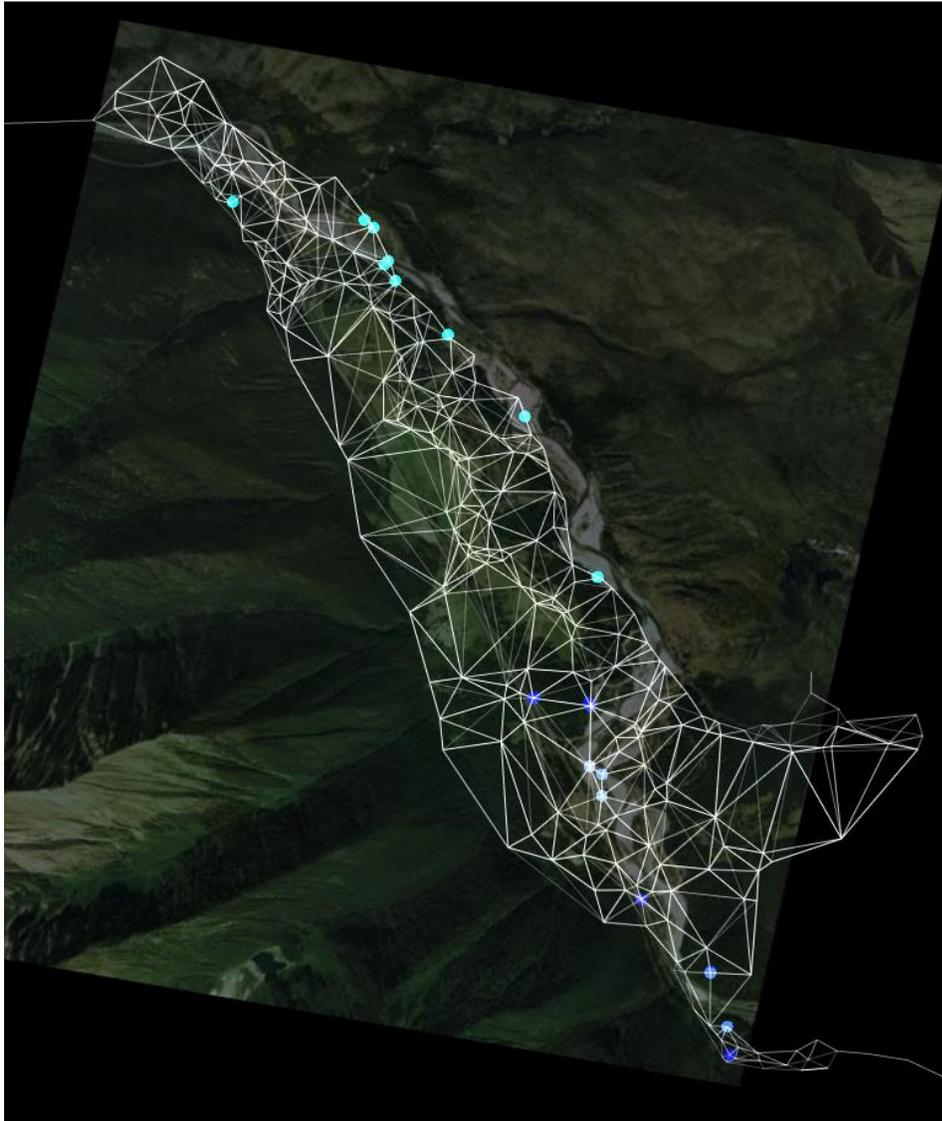
Figure 14. Screenshot of the Nyack Floodplain visualization

Multiple Currencies

While not necessary for the Nyack visualization, many models may require that more than one currency flux through the system at one time. For example, if the modeler also wanted to see the heat exchange within the river, then they would need to add a second currency (heat) to the model. The work for adding this functionality was shown in [3]. The ability to visualize models with multiple currencies is essential for many reasons.

It improves the interpretation of the visual data. It also allows the modeler to focus on

one of the currencies or both at the same time. This is achieved by setting the opacity of

the unwanted currency to zero. Furthermore, because the currencies are input into

Digistar 4 as separate files, the user can adjust the color and flow rate to be different for

each currency. Figure 15 shows an example of a heat currency (red) and a water currency

(blue) flowing through a simple cube-shaped matrix. The currency particles are different

sizes and are moving at different speeds.
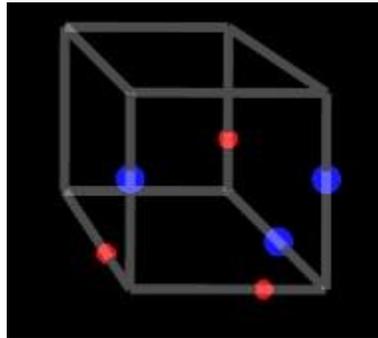
Figure 15. Visualization of heat and water currencies fluxing through a cube

CURRENT RESEARCH

Multiagent Systems

As is, the particle tracking program serves as a valuable commodity to
hydrologists, and while care has been taken to minimize the amount of technical debt
inherent in the system, additional work is required. Specifically, and to increase the
modularity of the system, the particle tracking algorithm needs to be fully integrated into
the current NEO architecture. After a series of design discussions with hydrologists, we
observed that particles behave as separate ―agents‖ that make independent decisions. This
prompted the study of multiagent systems.

Background

Whilst research on multiagent systems (MAS) began in the early 1980's, it only
really began to organize itself in the mid 1990's. Since then, with the advent of the
Internet and advances in computer science, multiagent system work has grown into the
large field that it is today. A MAS is characterized as a system where there is no global
system control, data is decentralized, computation is asynchronous, and each agent has
incomplete information or capabilities for solving the problem, and thus, has a limited
viewpoint.

An ―agent‖ is a computer system that is capable of independent action on behalf
of its user or owner [46]. In order for agents to be considered intelligent, they have to
possess two important abilities: they need to be capable of autonomous actions and they
need to be capable of interaction with other agents. These interactions include

cooperating, coordinating, and negotiating. To facilitate MAS technology, we require

mechanisms that allow agents to synchronize and coordinate their activities at run-time.

Weiss et al. [47] defined four different classes of agents:

1. Logic-based agents: in which the decision about what action to perform is
   made via logical deduction.

2. Reactive agents: in which decision-making is implemented in some form
   of direct mapping from situation to action.

3. Belief-Desire-Intention agents: in which decision making depends on the
   manipulation of data structures of the agent.

4. Layered architectures: in which decision-making is realized via various
   software layers, each of which is more or less explicitly reasoning about
   the environment at different levels of abstraction.

Research indicates that the type of agent architecture that would best fit this work

is the layered architecture, which is currently the most popular class of agent

architectures available. Figure 16 illustrates how agents can be organized and how they

interact with each other and their environment.

KEY

- - - - - organisational relationship

←——→ interaction

○ agent

Environment

sphere of influence
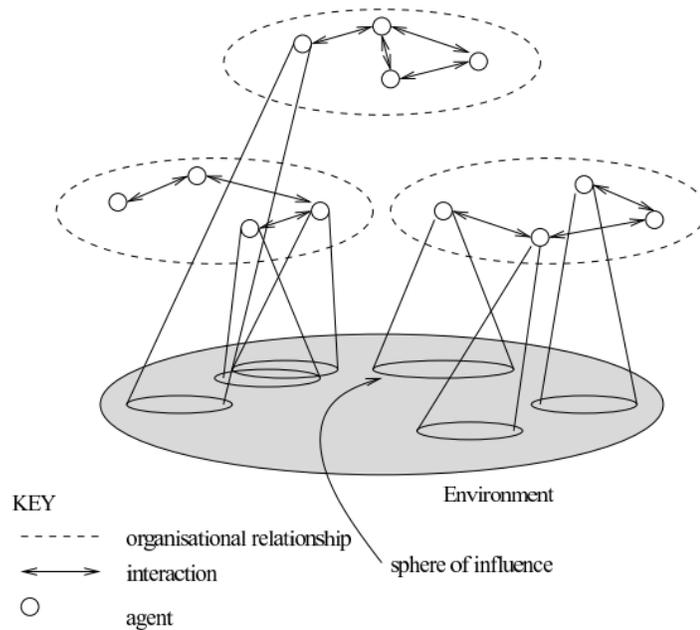
Figure 16. An example of how agents can be organized and interact [46]

One important aspect of a MAS is the ability for agents to communicate with each other. Because agents are autonomous, they can neither force other agents to perform actions, nor affect the properties, and thus the internal state of other agents. Agents are able to perform communicative actions in an attempt to influence other agents appropriately. These agents must also agree upon an ontology and a protocol for sharing information. Negotiation and bargaining are two important types of communication. Negotiation is a form of interaction in which a group of agents with conflicting interests try to come to a mutually acceptable agreement over some outcome. Bargaining is often solved by argumentation among agents [48]. Argumentation can be seen as a reasoning process consisting of the following four steps: Constructing arguments from available information, determining the different conflicts among the arguments, evaluating the

acceptability of the different arguments, and concluding, or defining, the justified

conclusions [49].

Below is a list of situations when agent based solutions are appropriate [50] and

why they are highly relevant additions to this project:

1. When the environment is open, or at least highly dynamic, uncertain, or
   complex – NEO models are dynamic and complex. You often have a large
   mesh network (i.e., a graph) with many currencies fluxing that need to
   updated, tracked, and reported on.

2. When agents are a natural metaphor – A hydrological model lends itself to
   agent metaphors.

3. Distribution of data, control, and expertise – In NEO models, each cell
   and/or edge holds different information. This makes the data distributed.

In Hydrology

The idea of using multiagent systems in the field of hydrology is not new [51].

The vernacular used in the field refers to them as agent-based models. These types of

models work well in natural sciences because they represent complex systems that can be

broken down into individual components and allow for communication between these

components.

Today, there is no standard in how to create these agent-based models [52]. Many

authors believe that creating this standard is an important step to wide-spread use of

agent-based models in the natural sciences. Volker et al. released an article in 2006 [53]

describing a standard protocol for describing agent-based models. The protocol was

widely used (having over 1000 citations) and in 2010, they published a review and update to the protocol [54]. Agent-based models have been widely used in the field of hydrology ([55], [56]) and highly relevant to the continuing refinement of the particle tracker and other currency algorithms.

An Integrated NEO Design

Research in multiagent systems revealed that the particle tracker program was already very close to representing intelligent agents. As particles move through a matrix, they observe their surroundings and make informed decisions about routes based on information that they collect. Thus, our investigations suggest that the particle tracker would be more useful to modelers when fully integrated with the NEO framework.

Functional Requirements

After speaking to NEO modeling experts, they outlined several functional requirements that the enhanced and fully integrated particle tracker program would need to adhere to. In this context, particles will now be referred to as ―agents". The functional requirements are:

1. Deploy agents: An agent manager will be able to either deploy agents at the beginning of a model or during model run time.

2. Track and move: Each agent will track its location in relation to nodes and/or links at run time as the agent moves through the network.

3. Sense and record conditions as an agent moves through a model: An agent will check its velocity and flux values in the current edge and will decide where to move based on that information.

4. Alter external values within the model: An agent will have the ability to update a counter in a node to track how many agents have passed through that node.

5. Retain information: A central agent manager will store the information that each agent collects.

6. Report: An agent will be able to either report during a simulation or a-posteriori.

7. Exit simulation: An agent manager within the model will handle the departure of agents.

8. Visualize simulation: The output collected by agents will be visualized in a visualization system.

System Design

Figure 17 depicts a UML class diagram of a possible design of the particle tracker. It employs intelligent agents, and uses NEO terminology to seamlessly integrate all components under the existing NEO architecture. The design encompasses all the functional requirements and was validated by NEO experts and hydrologists. After integration, agents become part of NEO, and will be configurable at run time to allow for more functionality and understandability of NEO models.
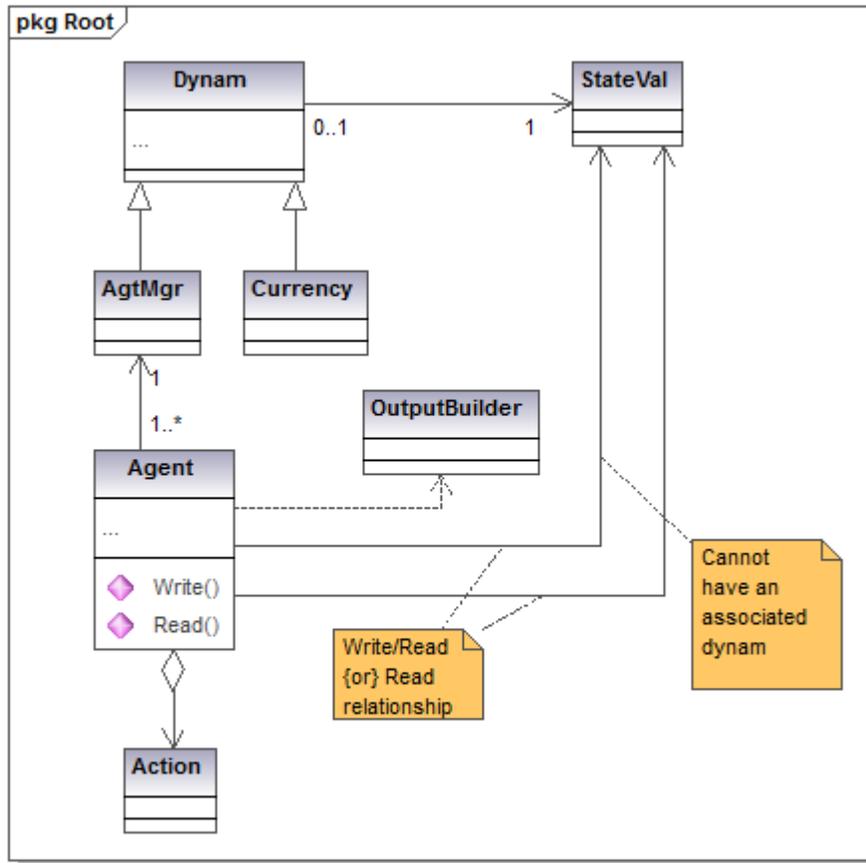
Figure 17. UML Class Diagram for integration with NEO

CONCLUSIONS AND FUTURE WORK

The work presented in this thesis contributes to the technical debt and modularity violation knowledge base –both topics of significant attention in the software engineering community. We used this knowledge and applied it to a multidisciplinary software project. The project was verified by using SonarQube's technical debt measurement tool and by domain experts in both software engineering and hydrology.

An accurate and extensible particle tracking algorithm was developed. The particle tracking program was run using hydrological data from the Nyack floodplain in northwestern Montana. The data was generated by NEO, a simulation software framework designed to facilitate development of complex natural system models. Further, the output from the particle tracking program was visualized using digital theater software –Digistar 4. The visualization of the data provided a unique and valuable alternative to understanding how the movement of the particles behaved and allowed modelers a gather greater understanding of their models.

The work presented in Chapter 6 introduced currently ongoing work to transform the particle tracking algorithm to exhibit additional properties inherent in multiagent systems that track intelligent agents. Although the design for integrating the particle tracking program with intelligent agents with NEO is completed, it has not been implemented. Another essential feature that would need to be added is the ability for agents to communicate with each other. In addition to functional enhancements, the particle tracking algorithm should be tested on models outside the field of hydrology to ensure that it is applicable to other domains.

REFERENCES CITED

[1] C. Izurieta, I. Griffith, D. Reimanis and R. Luhr, "On the Uncertainty of Technical Debt Measurements," in *International Conference on Information Science and Applications*, 2013.

[2] D. Reimanis, C. Izurieta, R. Luhr, L. Xiao, Y. Cai and G. Rudy, "A replication case study to measure the architectural quality of a commercial system," in *8th International Symposium on Empirical Software Engineering and Measurement*, 2014.

[3] R. Luhr, D. Reimanis, R. Cross, C. Izurieta, G. C. Poole and A. Helton, "Natural Science Visualization Using Digital Theater Software: Adapting Existing Planetarium Software to Model Ecological Systems," in *2013 International Conference on Information Science and Applications*, 2013.

[4] W. Cunningham, "The WyCash Portfolio Management System," *ACM SIGPLAN OOPS Messenger,* vol. 4, no. 2, pp. 29-30, 1992.

[5] M. Fowler, 14 October 2009. [Online]. Available: http://www.martinfowler.com/bliki/TechnicalDebtQuadrant.html. [Accessed 10 February 2015].

[6] C. Seaman, Y. Guo, C. Izurieta, Y. Cai, N. Zazworka, F. Shull and A. Vetrò, "Using technical debt data in decision making: Potential decision approaches," in *Third International Workshop on Managing Technical Debt*, 2012.

[7] B. Curtis, J. Sappidi and A. Szynkarski, "Estimating the Principal of an Application's Technical Debt," *IEEE software,* vol. 6, pp. 34-42, 2012.

[8] R. Ferenc, P. Hegedus and T. Gyimothy, "Software Product Quality Models," in *Evolving Software Systems*, Berlin, Springer-Verlag, 2013, pp. 65-100.

[9] I. Griffith, D. Reimanis, C. Izurieta, Z. Codabux, A. Deo and B. Williams, "The Correspondence between Software Quality Models and Technical Debt Estimation Approaches," in *Sixth International Workshop on Managing Technical Debt*, 2014.

[10] J.-L. Letouzey and M. Ilkiewicz, "Managing technical debt with the SQALE method," *IEEE software,* vol. 6, pp. 44-51, 2012.

[11] "CAST Report on Application Software Health," CAST, 2011.

[12] J. R. Taylor, An Introduction to Error Analysis, Univeristy Science Books, 1996.

[13] A. Nugroho, J. Visser and T. Kuipers, "An empirical model of technical debt and interest," in *2nd Workshop on Managing Technical Debt*, 2011.

[14] C. Izurieta, A. Vetrò, N. Zazworka, Y. Cai, C. Seaman and F. Shull, "Organizing the technical debt landscape," in *Third International Workshop on Managing Technical Debt*, 2012.

[15] N. Zazworka, C. Izurieta, S. Wong, Y. Cai, C. Seaman and F. Shull, "Comparing four approaches for technical debt identification," *Software Quality Journal,* vol. 22, no. 3, pp. 403-426, 2014.

[16] C. Izurieta and J. M. Bieman, "How software designs decay: A pilot study of pattern evolution," in *First International Symposium on Empirical Software Engineering and Measurement*, 2007.

[17] C. Izurieta and J. M. Bieman, "A multiple case study of design pattern decay, grime, and rot in evolving software systems," *Software Quality Journal,* vol. 21, no. 2, pp. 289-323, 2013.

[18] M. Fowler, Refactoring, imporving the design of existing code, Pearson Education India, 2002.

[19] A. Vetro, M. Torchiano and M. Morisio, "Assessing the precision of findbugs by mining java projects developed at a university," in *7th IEEE Working Conference on Mining Software Repositories*, 2010.

[20] A. Vetro, M. Morisio and M. Torchiano, "An empirical validation of FindBugs issues related to defects," in *15th Annual Conference on Evaluation & Assessment in Software Engineering*, 2011.

[21] S. Wong, Y. Cai, M. Kim and M. Dalton, "Detecting software modularity violations," in *33rd International Conference on Software Engineering*, 2011.

[22] C. Y. Baldwin and K. B. Clark, Design rules: The power of modularity (Vol. 1), MIT Press, 2000.

[23] F. J. Shull, J. C. Carver, S. Vegas and N. Juristo, "The role of replications in empirical software engineering," *Empirical Software Engineering,* vol. 13, no. 2, pp. 211-218, 2008.

[24] A. Brooks, M. Roper, M. Wood, J. Daly and J. Miller, "Replication's role in software engineering," in *Guide to advanced empirical software engineering*, Springer

London, 2008, pp. 365-379.

[25] N. Juristo and A. M. Moreno, Basics of software engineering experimentation, Springer Publishing Company, Inc, 2010.

[26] R. Schwanke, L. Xiao and Y. Cai, "Measuring architecture quality by structure plus history analysis," in *35th International Conference on Software Engineering*, 2013.

[27] "Golden Helix," [Online]. Available: http://www.goldenhelix.com/. [Accessed 26 2 2015].

[28] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM,* vol. 15, no. 12, pp. 1053-1058, 1972.

[29] M. G. Kendall, "A new measure of rank correlation," *Biometrika,* pp. 81-93, 1938.

[30] R. Ott and M. Longnecker, An introduction to statistical methods and data analysis, Cengage Learning, 2008.

[31] C. Izurieta, G. Poole, R. A. Payn, I. Griffith, R. Nix, A. Helton, E. Bernhardt and A. J. Burgin, "Development and Application of a Simulation Environment (NEO) for Integrating Empirical and Computational Investigations of System-Level Complexity," in *2012 International Conference on Information Science and Applications*, 2012.

[32] R. A. Payn, A. M. Helton, G. C. Poole, C. Izurieta, A. J. Burgin and E. S. Bernhardt, "A generalized optimization model of microbially driven aquatic biogeochemistry based on thermodynamic, kinetic, and stoichiometric ecological theory," *Ecological Modelling,* vol. 294, pp. 1-18, 2014.

[33] R. Nix, "Beginners Manual to the NEO modeling Framework," 11 October 2012. [Online]. Available: https://www.assembla.com/wiki/show/neo/Manuals. [Accessed 28 2 2015].

[34] S. L. Dingman, "Basic Hydrologic Concepts," in *Physical Hydrology*, Waveland Press, Inc., 2008, pp. 7-35.

[35] J. J. Fraley and B. B. Shepard, "Life history, ecology and population status of migratory bull trout (Salvelinus confluentus) in the Flathead Lake and River system," *Northwest Science,* vol. 63, no. 4, 1989.

[36] B. N. McLellan and F. W. Hovey, "The diet of grizzly bears in the Flathead River drainage of southeastern British Columbia.," *Canadian Journal of Zoology,* vol. 73, no. 4, pp. 704-712, 1995.

[37] J. A. Stanford and J. V. Ward, "An ecosystem perspective of alluvial rivers: connectivity and the hyporheic corridor," *Journal of the North American Benthological Society,* pp. 48-60, 1993.

[38] A. M. Helton, G. C. Poole, R. A. Payn, C. Izurieta and J. A. Stanford, "Scaling flow path processes to fluvial landscapes: An integrated field and model assessment of temperature and dissolved oxygen dynamics in a river-floodplain-aquifer system," *Journal of Geophysical Research: Biogeosciences,* vol. 117, no. G4, pp. 2005-2012, 2012.

[39] A. M. Helton, G. C. Poole, R. A. Payn, C. Izurieta and J. A. Stanford, ""Relative influences of the river channel, floodplain surface, and alluvial aquifer on simulated hydrologic residence time in a montane river floodplain," *Geomorphology,* vol. 205, pp. 17-26, 2014.

[40] G. M. Hornberger, J. P. Raffensperger, P. L. Wiberg and K. N. Eshleman, "Catchment Hydrology: The Hillslope-Stream Continuum," in *Elements of Physical Hydrology*, JHU Press, 1998, pp. 199-222.

[41] G. C. Poole, J. A. Stanford, S. W. Running, C. A. Frissell, W. W. Woessne and K. B. Ellis, "A patch hierarchy approach to modeling surface and subsurface hydrology in complex flood-plain environments," *Earth Surface Processes and Landforms,* vol. 29, no. 10, pp. 1259-1274, 2004.

[42] R. Walton, R. S. Chapman and J. E. Davis, "Development and application of the wetlands dynamic water budget model," *Wetlands,* vol. 16, no. 3, pp. 347-357, 1996.

[43] "SonarQube 4.5.4," SonarSource S.A., 26 2 2015. [Online]. Available: http://www.sonarqube.org/downloads/. [Accessed 10 3 2015].

[44] "Digistar," Evans & Sutherland, [Online]. Available: http://www.es.com/Products/Digistar.html. [Accessed 10 3 2015].

[45] "Taylor Planetarium Montana Moment: Watershed visualization," Vimeo, 9 2014. [Online]. Available: https://vimeo.com/78744894. [Accessed 10 3 2015].

[46] M. Wooldridge, "Introduction," in *An Introduction to MultiAgent Systems*, John Wiley & Sons, 2009, pp. 3-12.

[47] G. Weiss and M. Wooldridge, "Intelligent Agents," in *Multiagent Systems*, MIT Press, 2013, pp. 3-50.

[48] G. Weiss, S. Fatima and I. Rahwan, "Negotiation and Bargaining," in *Multiagent Systems*, MIT Press, 2013, pp. 143-176.

[49] G. Weiss and I. Rahwan, "Argumentation among Agents," in *Mutiagent Systems*, MIT Press, 2013, pp. 177-210.

[50] M. Wooldridge, "Methodologies," in *An Introduction to MultiAgent Systems*, John Wiley & Sons, 2009, pp. 183-193.

[51] D. L. DeAngelis and V. Grimm, "Individual-based models in ecology after four decades," *F1000prime reports,* vol. 6, no. 39, 2 June 2014.

[52] C. M. Macal and M. J. North, "Tutorial on agent-based modelling and simulation," *Journal of simulation,* vol. 4, no. 3, pp. 151-162, 2010.

[53] V. Grimm et al., "A standard protocol for describing individual-based and agent-based models," *Ecological modelling,* vol. 198, no. 1, pp. 115-126, 2006.

[54] V. Grimm, U. Berger, D. L. DeAngelis, J. G. Polhill, J. Giske and S. F. Railsback, "The ODD protocol: a review and first update," *Ecological modelling,* vol. 221, no. 23, pp. 2760-2768, 2010.

[55] M. Bithell and J. Brasington, "Coupling agent-based models of subsistence farming with individual-based forest models and dynamic models of water distribution," *Environmental Modelling & Software,* vol. 24, no. 2, pp. 173-190, 2009.

[56] S. Reaney, "The use of agent based modelling techniques in hydrology: determining the spatial and temporal origin of channel flow in semi-arid catchments," *Earth Surface Processes and Landforms,* vol. 33, no. 2, pp. 317-327, 2007.

APPENDICES

APPENDIX A

ON THE UNCERTAINTY OF TECHNICAL

DEBT MEASUREMENTS

APPENDIX A

ON THE UNCERTAINTY OF TECHNICAL DEBT MEASURMENTS

Contribution of Authors and Co-Authors

Manuscript in Appendix A

Author: Clemente Izurieta

Contributions: Conceived the need for the work. Showed the propagation of error through technical debt equations. Wrote first and final drafts of the manuscript.

Co-Author: Isaac Griffith

Contributions: Provided editorial feedback and review of the final manuscript.

Co-Author: Derek Reimanis

Contributions: Provided editorial feedback and review of the final manuscript.

Co-Author: Rachael Luhr

Contributions: Provided editorial feedback and review of the final manuscript. Prepared and presented the research at ICISA 2013.

Manuscript Information Page

On the Uncertainty of Technical Debt Measurements

Clemente Izurieta, Isaac Griffith, Derek Reimanis, Rachael Luhr

Software Engineering Laboratory
Department of Computer Science, Montana State University
clemente.izurieta@cs.montana.edu, {isaac.griffith, derek.reimanis, rachael.luhr}@msu.montana.edu

Abstract—**Measurements are subject to random and systematic errors, yet almost no study in software engineering makes significant efforts in reporting these errors. Whilst established statistical techniques are well suited for the analysis of random error, such techniques are not valid in the presence of systematic errors. We propose a departure from de-facto methods of reporting results of technical debt measurements for more rigorous techniques drawn from established methods in the physical sciences. This line of inquiry focuses on technical debt calculations; however it can be generalized to quantitative software engineering studies. We pose research questions and seek answers to the identification of systematic errors in metric-based tools, as well as the reporting of such errors when subjected to propagation. Exploratory investigations reveal that the techniques suggested allow for the comparison of uncertainties that come from differing sources. We suggest the study of error propagation of technical debt is a worthwhile subject for further research and techniques seeded from the physical sciences present viable options that can be used in software engineering reporting.**

Keywords- software quality and maintenance; error analysis; technical debt

INTRODUCTION

This line of investigation proposes an alternative to the way we analyze and report values that are subject to uncertainty[1]. Typically, given multiple calculations of a metric (i.e., technical debt), we use well-known statistical methods to analyze and compare them, with the standard deviation as a good approximation for uncertainty. This is allowed provided the uncertainty values are *random*; however not all uncertainties are random. Technical debt calculations may suffer from *systematic* errors that occur as a result of poor calibration of the tools used to measure the debt. For example, a tool that performs static analysis for detecting code smells may consistently report violations where there are none. Regardless of the number of times we repeat an experiment or increase the sample sizes, if the errors[6] are systematic, then we cannot use the standard deviation as an approximation for uncertainty. Unfortunately, detection of such systematic errors is extremely difficult. In the physical sciences it is common for laboratories to designate an instrument as having systematic uncertainty expressed as a percentage. Software engineering laboratories produce tools that are typically developed as prototypes without all the quality assurance expected of commercial products, yet these tools are used in empirical investigations. Due diligence suggests that we should also allow for systematic uncertainty of the various dependent and independent variables that our tools measure. Thus, measurements will have a random component and a systematic component of uncertainty. Even in the unlikely scenario where all uncertainties are random, using propagation techniques for all variables participating in calculations (dependent and independent) is a complementary methodology to statistical methods. We initially focus on technical debt measurements, as a significant amount of systematic errors can potentially exist. A number of tools are also available that are complex enough in how they perform their calculations and could potentially benefit from this line of inquiry. Examples of these tools include CLIO [7] for checking modularity violations, RBML compliance checker [8] that automatically calculates a distance between a realization of a design pattern and the intended design, and CodeVizard [9] that looks for code smells (as introduced by Fowler [10]). Our goal is to generate early feedback and suggest possible benefits from this research. We pose the following questions:

---

[1] The words error and uncertainty are synonymous and are used interchangeably throughout the manuscript.

Q1: How do we investigate systematic errors when computing technical debt parameters?
Q2: Are the suggested uncertainty propagation techniques enough?
Q3: What source code testing strategies can we use to identify and detect potential sources of systematic errors?

TECHNICAL DEBT

By definition, there is uncertainty in the measurement of technical debt. The technical debt metaphor describes a situation in which developers accept quality compromises in the current release to meet a deadline (e.g. delivering a release on time) [1]. Curtis, Sappidi and Szynkarski [2] state that ―there is no exact measure of Technical Debt, since its calculation must be based only on the structural flaws that the organization intends to fix," and not all organizations fix, are aware of, nor quantify technical debt with similar tools, techniques, algorithms or precision. Additionally, small changes to parameters can admit large fluctuations of technical debt calculations; thus revealing the sensitivity of final estimates. Few (if any) studies in software engineering report on the analysis of error, as well as its propagation during scientific experimentation. The ability to keep uncertainty at a minimum is critical in any scientific field. Technical debt studies reveal that no measurements made are free of uncertainty. This is evident in the calculations performed on technical debt as reported by CAST [3], Letouzey and Ilkiewicz [4] in the SQALE methodology, and by the Software Improvement Group (SIG) software quality assessment method [12] based on ISO/IEC 9126 [13]. Thus, in order to increase rigor in software engineering measurements, it is important to adopt a *modus operandi* that allows for representation of uncertainties and teaches how those uncertainties are propagated through calculations made in ratio scales. This is especially important when the source of errors is not random. We describe the early stages of experimenting with techniques used in the physical sciences [5] and applying them to software engineering measurements. Specifically, we focus on the measurement of technical debt. We expect that by using these techniques, scientists will be able to:

i. Compare two or more measured values (from potentially different sources) against one another,

ii. Propagate errors (random or systematic) through all allowed operators defined by a ratio scale, and

iii. Identify ways to provide meaningful error calculations in expressions that involve multiple variables– each variable subject to its own uncertainty.

METHODS

Definitions

We provide abridged definitions for the technical debt variables [6] that are under early investigation to quantify the uncertainty of their measurement.

1) Technical Debt Principal

Refers to the effort required to complete a task that is left undone. A task is a representation of a technical debt item that runs a risk of causing future problems if left undone.

2) Technical Debt Interest Amount

Refers to an estimate of the amount of extra work that will be needed to maintain the software if a technical debt item is not repaid. Interest incurs a continuing cost to its associated item.

3) Technical Debt Interest Probability

Refers to the probability that the technical debt, if not repaid, will make other work more expensive over a given period of time or a release. Probability is time sensitive. For example, a debt item may have a higher probability of being addressed in the next major revision of a product than the current version.

4) Violation

Refers to violations of agreed upon solutions to design or coding practices. CAST software [3] uses an ordinal scale to classify violations into high, medium, and low.

5) Uncertainty and Error

Random errors or uncertainty in measurement theory are abundant and refer to the delta that exists between the expected value of a measure and its actual measurement. Errors can overestimate or underestimate the expected value of a measurement. For example, in a technical debt context, the estimation of costs associated with technical debt items are subject to error.

Uncertainty of a Measurement

We use operations and notation proposed by Taylor [5] to state technical debt metrics as follows:

$$measured\ value\ of\ TD_{principal} = (TD_{principal})_{best} \pm \partial_{TD}$$

and represents an experimenter's best estimate of technical debt (*TD*) principal with a margin of error or uncertainty of ∂_{TD}. The estimate lies between $(TD_{principal})_{best} + \partial_{TD}$ and $(TD_{principal})_{best} - \partial_{TD}$. We use the uncertainty term ∂_{TD} as an aggregation of both random and systematic errors. The calculations presented herein can be applied to both.

UNCERTAINTY CALCULATIONS

Comparing Measures

Since results that report on a single measure are uninteresting, scientists typically compare two or more measurements against each other to show relationships between values. Technical debt literature is relatively new, and unlike in other scientific fields, *accepted values* (i.e., values accepted by a scientific community) of technical debt do not exist (e.g., the ideal level of quality of a design pattern). Suppose two organizations A and B report their $TD_{principal}$ as follows:

$$(A\_TD_{principal})_{best} \pm \partial_{A\_TD}$$

$$(B\_TD_{principal})_{best} \pm \partial_{B\_TD}$$

then the estimate for the highest probable value of the difference is computed as follows:

$$(A\_TD_{principal})_{best} - (B\_TD_{principal})_{best} + (\partial_{A\_TD} + \partial_{B\_TD})$$

and the estimate for the lowest probable value of the difference is computed as follows:

$$(A\_TD_{principal})_{best} - (B\_TD_{principal})_{best} - (\partial_{A\_TD} + \partial_{B\_TD})$$

Assuming that both organizations report the uncertainty associated with their respective measurements using the same number of significant figures; then it is important that the discrepancy in uncertainty is also reported using the same number of significant figures. If one organization reports its uncertainty measurements with significantly more precision than another, then we must use the coarser uncertainty measurements to report results.

Propagation of Error

When computing the value of technical debt, we must also account for how the uncertainties of technical debt interest and probability (i.e., the independent variables) propagate. Even when we calculate the value of technical debt principal alone we must account for uncertainties in the average hours needed to fix violations (high, medium and low), the dollar cost per hour, and the average number of hours required to fix a violation. We use Taylor's [5] rules to estimate the propagation of technical debt uncertainty. To exemplify the methodology we use the calculations proposed by Nugroho et al. [11].

Sums and Differences

The propagation of uncertainty in the sums or differences of measured quantities is similar to the example from section IV.A. Thus, if several quantities $x_1 \ldots x_n$ with corresponding uncertainties $\partial x_1 \ldots \partial x_n$ are measured with uncertainty, then the overall uncertainty of their additions, differences, or combination of both operations is given by:

$$Uncertainty = \partial x_1 + \ldots + \partial x_n$$

Products and Quotients

The propagation of uncertainty in products or quotients of measured quantities is best given using *fractional uncertainty* notation. Recall from III.B the calculation of technical debt as:

$$measured\ value\ of\ TD_{principal}$$
$$= (TD_{principal})_{best} \pm \partial_{TD}$$

then, the fractional uncertainty in the $TD_{principal}$ is defined as:

$$fractional\ uncertainty\ TD_{principal}$$
$$= \frac{\partial_{TD}}{\left|(TD_{principal})_{best}\right|}$$

Nugroho et al. [11] calculate technical debt principal by estimating the Repair Effort (*RE*) necessary to increase the quality of the software to an ideal level. *RE* is then calculated as follows:

$$RE = RF \times RV$$

where *RF* is a Rework Fraction and *RV* is the Rebuild Value. Multiplying the System Size (*SS*) against a Technology Factor (*TF*) carries out the *RV* calculation for a system. *TF* is calculated as the number of man-months per source code statement, and *SS* can be calculated either by using lines of code (LOC) or function points. There is a large possibility of having uncertainty in all these variable calculations; for example, using function points carries a significantly higher degree of uncertainty than using LOC. Thus, if *RF* is given by:

$$measured\ value\ RF = RF_{best} \pm \partial_{RF}$$

and
$$measured\ value\ RV = RV_{best}\ \pm\ \partial_{RV}$$

then if the uncertainty for the measured value of *RE* is given by ∂_{RE}, it follows that:

$$\frac{\partial_{RE}}{|RE_{best}|} = \frac{\partial_{RF}}{|RF_{best}|} + \frac{\partial_{RV}}{|RV_{best}|}$$

In general, if several quantities $x_1 .. x_n$ with corresponding uncertainties $\partial x_1 .. \partial x_n$ are measured, then the overall uncertainty of their products, quotients, or combination of both operations is given by:

$$\frac{Uncertainty}{|(measure)_{best}|} = \frac{\partial x_1}{|x_{1best}|} + .. + \frac{\partial x_n}{|x_{nbest}|}$$

Power Uncertainty
Nugroho et al. [11] calculate the technical debt interest amount as the difference between the *ideal* level of Maintenance Effort (*ME*) needed for a software module and the current level of maintenance effort that may include extra costs as a result of additional quality issues. The *ME* is given by the formula:

$$ME = \frac{MF \times RV}{QF}$$

where $QF = 2^{((qualityLevel-3)/2)}$, and quality levels range from 1 to 5, thus producing *QF* values of 0.5, 07, 1.0, 1.4, and 2.0 respectively. *MF* is a Maintenance Fraction representing the number of lines that undergo change in a year, and *RV* is the Rebuild Value (originally described in section IV.B.2); which can also be calculated as:

$$RV = SS \times (1+r)^t \times TF$$

thus; for time *t* and growth rate *r*, the rebuild value (*RV*) of a system increases over time if left unattended. If the rate *r* changes as time increases, then this formula must account for the range of that movement, or the uncertainty of the measurement. To demonstrate how to account for this uncertainty we only focus on the $(1+r)^t$ factor of the *RV* calculation. The multiplication by *SS*, and *TF*, subject to uncertainty is carried out using the propagation techniques for products and quotients described in section IV.B.2.

If *r* is measured with uncertainty ∂r, then the uncertainty of the calculation of the factor

$(1+r)^t$ is given by $t \times \frac{\partial r}{r}$ for a fixed value of *t*. The overall fractional uncertainty of the *RV* calculation is then given as:

$$\frac{\partial_{RV}}{|RV_{best}|} = \frac{\partial_{SS}}{|SS_{best}|} + t \times \frac{\partial r}{r} + \frac{\partial_{TF}}{|TF_{best}|}$$

Technical Debt Interest Probability
Since the probability of interest varies with different time frames, a time element must be attached to the probability. Additionally, the value assigned to interest probability tends to be classified in an ordinal scale based on historical data. There are no definitive technical debt interest probability calculations; however if ratio scale arithmetic is used, then propagation of error can be accounted for with the proposed techniques. If however probabilities are table based and ordinal, further exploration is necessary.

Multivariate Uncertainty
Taylor [5] describes formulas that use quadrature where appropriate, but these techniques need validation in the technical debt domain. Quadrature allows us to discount the negligible effect of some unlikely error propagation possibilities, thus providing more realistic error ranges when calculating the error of a multivariate expression. For example, in section IV.B.1 we described the calculation of $TD_{principal}$, where the expression is only made up of sums and differences. We showed that the total uncertainty equaled the sum of the uncertainties of each component in the expression. However; this formula clearly overestimates ∂_{TD} because this can only occur when we underestimate the values of each component in the expression by the full amount. Similarly we would underestimate ∂_{TD} when we overestimate the value of each constituent component in the expression by the full amount. As the number of components in the expression grows it is more unlikely that all participating components either overestimate or underestimate their values at exactly the same time. If the expression shown in section IV.B.1 only had two components (i.e., two variables) that represent the technical debt measurements of organizations A and B; given by $A\_TD_{principal}$ and $B\_TD_{principal}$, then there is only a 50% chance of underestimation or overestimation. The error is then given by $\sqrt{\partial_{A\_TD}{}^2 + \partial_{B\_TD}{}^2}$, which is always less than $\partial_{A\_TD} + \partial_{B\_TD}$. Quadrature is an

applicable calculation when measurements come from Normal or Gaussian distributions and the measurements are independent. Can we then be justified in using quadrature in software engineering technical debt calculations that are subject to systematic errors? Or do we need other techniques to find better approximations of errors?

CONCLUSIONS

Unlike the physical sciences where we can, for example, use multiple clocks or meters to identify an artifact that produces systematic errors (e.g., a clock that is 5 seconds fast, or a meter whose readings are consistently low), technical debt tools can not be compared against other tools because they all compute equations differently, thus making detection of systematic errors extremely difficult. Until accepted values for certain calculations exist, or known reliable tools that we can calibrate against exist, we must seriously consider reporting results with their corresponding uncertainty, and we must also be aware of how these uncertainties propagate. This research seeks answers and potentially encouraging lines of inquiry to help answer the three questions posed in the introduction. To answer **Q1**, we argue that in the absence of established accepted technical debt values (principal, interest and probability), we can use reported uncertainties associated with these values as a means to investigate the existence of systematic errors by comparing the overlap of the uncertainties. This can help identify calibration issues when comparing outputs from multiple tools. In section IV.B we described the various ways in which the propagation of errors can occur. We suspect that additional propagation techniques are required in order to minimize errors and help answer **Q2**. For example, Taylor [5] describes formulas that use quadrature where appropriate, but these techniques need validation in the technical debt domain. Quadrature allows us to discount the negligible effect of some error propagation factors, thus providing more realistic error ranges. Answers to **Q3** remain elusive, however the propagation of errors from experimentation could be automated, and final estimates of debt could be compared against *expected values* that have organizational context and are stored in testing golden files. Seaman et al. [6] state that ―in any approach to decision making about technical debt, some human intervention is required to provide information that cannot be reliably measured,‖ thus understanding how the propagation of uncertainty occurs is a critical factor if we want to continue to improve the decision making process of which items to refactor.

REFERENCES

[1] W. Cunningham, "The WyCash Portfolio Management System," in *Addendum to the proceedings on Object-oriented programming systems, languages, and applications*, 1992, pp. 29-30.

[2] Curtis, B.; Sappidi, J.; Szynkarski, A., "Estimating the Principal of an Application's Technical Debt," *Software, IEEE* , vol.29, no.6, pp.34-42, Nov.-Dec. 2012, doi: 10.1109/MS.2012.156

[3] ——, *CAST Report on Application Software Health*, tech. report, CAST Software, 2011.

[4] Letouzey, Jean-Louis; Ilkiewicz, Michel, "Managing Technical Debt with the SQALE Method," *Software, IEEE*, vol.29, no.6, pp.44-51, Nov-Dec. 2012.

[5] Taylor, J. R. 1997. *An Introduction to Error Analysis*. University Science Books, 2nd. Edition, ISBN-13: 978-0935702750.

[6] Seaman, C.; Yuepu Guo; Izurieta, C.; Yuanfang Cai; Zazworka, N.; Shull, F.; Vetro, A., "Using technical debt data in decision making: Potential decision approaches," *Managing Technical Debt (MTD), 2012 Third International Workshop on* , vol., no., pp.45-48, 5-5 June 2012. doi: 10.1109/MTD.2012.6225999

[7] S. Wong, Y. Cai, M. Kim, and M. Dalton, ―Detecting software modularity violations," in *Proc. 33rd International Conference on Software Engineering*, May 2011, pp. 411–420.

[8] S. Strasser, C. Frederickson, K. Fenger, C. Izurieta, ―An Automated Software Tool for Validating Design Patterns," in *Proc. 24th International Conference on Computer Applications in Industry and Engineering*, November 2011, Honolulu, HI.

[9] N. Zazworka, ―CodeVizard: a tool to aid the analysis of software evolution," in *Proc. Of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement,* September 2010, Bolzano-Bozen, Italy.

[10] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, 1st ed. Addison-Wesley Professional, Jul. 1999.

[11] Nugroho, A.; Visser, J.; Kuipers, T., ―An empirical model of technical debt and interest," In *Proceedings of the 2nd Workshop on Managing Technical Debt* (MTD '11). ACM, New York, NY, USA, 1-8. doi:10.1145/1985362.1985364 http://doi.acm.org/10.1145/1985362.1985364

[12] Heitlager, I; Kuipers, T.; Visser, J., ―A practical model for measuring maintainability." In *6th Internation Conference on the Quality of Information and Communications Technology*, 2007. QUATIC 2007. Pages 30-39.

[13] International Organization for Standardization. ISO/IEC 9126-1: Software Engineering – product quality – part 1: Quality model, 2001

<u>APPENDIX B</u>

NATRUAL SCIENCE VISUALIZATION USING

DIGITAL THEATER SOFTWARE

APPENDIX B

NATURAL SCIENCE VISUALIZATION USING
DIGITAL THEATER SOFTWARE

Contribution of Authors and Co-Authors

Manuscript in Appendix B

Author: Rachael Luhr

Contributions: Wrote first and final drafts of the manuscript. Developed visualization and added functionality for multiple currencies. Constructed MSU Minute.

Co-Author: Derek Reimanis

Contributions: Wrote portion of first draft. Helped develop initial visualization. Provided feedback on early drafts of the manuscript and reviewed the final manuscript.

Co-Author: Renee Cross

Contributions: Wrote portion of first draft. Artistically and technically aided in the construction of the MSU Minute. Provided feedback on early drafts of the manuscript and reviewed the final manuscript.

Co-Author: Clemente Izurieta

Contributions: Provided feedback on early drafts of the manuscript. Provided editorial feedback and review of the final manuscript.

Co-Author: Geoffrey C. Poole

Contributions: Served as hydrology domain expert. Provided editorial feedback and review of the final manuscript.

Co-Author: Ashley Helton

Contributions: Gathered data that was used to make visualization.

Manuscript Information Page

Natural Science Visualization Using Digital Theater Software

Adapting existing planetarium software to model ecological systems

Rachael Luhr[1,3], Derek Reimanis[1], Renee Cross[1,3], Clemente Izurieta[1,3], Geoffrey C. Poole[2,3], Ashley Helton[4]

[1]Software Engineering Laboratory, Department of Computer Science, Montana State University
[2] Land Resources and Environmental Sciences Department, Montana State University
[3] Montana Institute on Ecosystems
[4] Department of Biology, Duke University

{rachael.luhr, derek.reimanis, renee.thibeault}@msu.montana.edu, {clemente.izurieta, gpoole}@montana.edu, amh72@duke.edu

Abstract—**Data in the natural sciences can often be dense and difficult to understand. The process of visualizing this data helps to alleviate these issues. In this paper, we demonstrate how using digital theater software built for planetaria can be an appropriate medium in which to facilitate these visualizations; not only because of the software's complex and comprehensive graphics engine, but also as a means to convey the information this data is representing to the general public in way that is understandable, accessible and enjoyable. We exemplify the use of this technology through a case study where the simulation of water molecules in the Nyack Floodplain on the Middle Fork Flathead River, are visualized with dome technology.**

Keywords— smart media, 3D planetarium content, digital theatre, flux networks

INTRODUCTION

Digistar 4 is digital theater software produced by Evans & Sutherland [1] for use in planetariums. The platform has an easy-to-use interface to complement its complex graphics and physics engine. This allows the software to be used by non-experts to create planetarium shows that educate and entertain the public. This type of visualization tool could be applied to other areas of science for the purposes of new discoveries, education and community outreach. In the past few years, there has been collaboration between the Computer Science Department and the Department of Land Resources and Environmental Science at Montana State University in an effort to visualize complex ecological data. This representation eases the otherwise non-intuitive process of understanding the spatiotemporal dynamics commonly found in ecological models. This effort was successful and helped seed the idea that Digistar 4 can be used as a tool to help to comprehend complex data-intensive phenomena that are difficult to conceptualize otherwise.

This paper is organized as follows: Section II outlines the background information regarding the modeling framework and previous work in visualization, Section III explains the methodology used to develop and display the visualizations of the models, as well as the case study in the Nyack Floodplain used to exemplify this work. Section IV details the outreach opportunities available when using this technology, and Section V describes future uses of this technology.

BACKGROUND

Visualization of Scientific Models

Natural, social and engineering phenomena frequently exhibit strong spatial and/or temporal trends and interactions. Understanding these causal relationships between components and their evolutionary trajectories is the underpinning of a large number of domain sciences. As environmental data sets become larger and denser, effective exploration and pattern analysis becomes a critical bottleneck in analytical reasoning that can hinder decision making [2].

Data visualization is a dynamic discipline in order to quickly react to new developments in graphics hardware, virtual environments or network technology, to new computer graphics algorithms, and last but not least to the ever growing size of scientific datasets" [3]. Scientific data is difficult to understand for those who are not familiar with the specific field and research being done. The more complex models built by simulation software are even more difficult to understand. These models are often represented by long lists of information or matrices of numbers, which make this type of data difficult to comprehend. Visualizing these numbers helps to share the knowledge and discovery of scientific research to those outside the specific

field. This can be useful for education, community outreach, grant proposals and research funding.

Network Exchange Objects

Network Exchange Objects (NEO) is a simulation framework under development at Montana State University [4]. NEO facilitates the development of simulation models that describe the behavior of complex systems – specifically the flux and storage of multiple interactive "currencies" through systems represented as networks. A currency is anything within the model that is being exchanged between components or the modeled system (e.g., energy, economic capital, genes, carbon, nutrients, or any other resource of interest, depending on the system). Currencies are manipulated as they flow between nodes and edges in the network, representing the effect entities have on the flow.

NEO is designed to study systems that can be described as "complex adaptive hierarchical networks" (CAHNs). A CAHN is implemented with a graph G and uses a combination of network theory, complex systems theory, hierarchy theory, and interdependency idioms to characterize the structure and behavior of a model built atop this network (G) of interconnected nodes and edges. The implementation of these four characteristics of natural systems in NEO allows scientists to investigate (1) patterns that emerge from network connections, (2) interactions among system components, specifically the interactions of flow through nodes/edges, (3) hierarchical structure as it affects interaction, and (4) the effect on the model caused by currency interdependency [4].

METHODOLOGY AND CASE STUDY

The focus of our work involves leveraging the powerful features of the Digistar 4 framework to create working visualizations of ecological models. We designed a system to significantly reduce the effort required to handle the transfer of information resulting from simulations that ran in the NEO simulation environment to the Digistar 4 system. The interface to facilitate transfer of information was written using the Java language. Its functionality allows for the parsing of text or comma delineated files to extract the necessary information necessary to exercise the Digistar4 visualization engine.

The visualized models represent the movement of currencies in flux networks. That is, the models exhibit complex behavior through multiple data of different types flowing and interacting through the system. The following case study describes the successful visualization of water molecules that represent a single

currency in the Nyack Floodplain on the Middle Fork Flathead River. The river channel, floodplain surface and aquifer (structure) and the movement (behavior) of water (currency) were modeled using NEO.

To understand how different river compartments affect downstream transport times through the Nyack study site, Helton et al. [8] simulated water flow within and among the channel, subsurface, and floodplain surface hydrologic system. A three dimensional model (shown in Figure 1) was constructed from the site (shown in Figure 2).
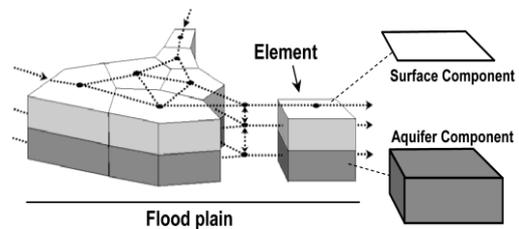
Fig 1. Inter-connected components of the river [7]. Figure Copyright © 2004 John Wiley & Sons, Ltd. Reproduced by permission.

The floodplain is divided into spatially discrete patches, represented by nodes connected in a three dimensional network. Edges represent the connections to neighbor patches. The model represents horizontal water flow; horizontal and vertical subsurface flows; and vertical exchanges between subsurface and surface waters. Equations used to represent physical principles behind the water flow are described by Walton et al. [6] and Poole et al. [7].

The output of the model was then visualized with the Digistar 4 planetarium engine. Figure 2 shows the end result; an aerial image of the Middle Fork Flathead River, taken from Google Earth [5], that is superimposed behind the modeled floodplain structure of the Middle Fork Flathead River.
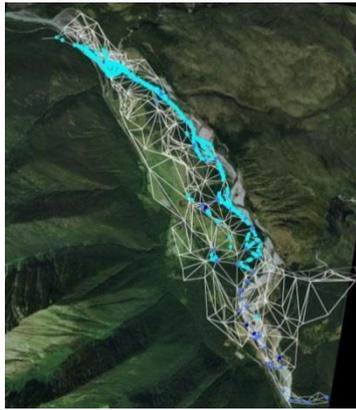
Fig 2. Single frame from an animated visualization of surface- and groundwater flow through the Nyack Floodplain of the Middle Fork Flathead River, Montana, USA. Light blue dots shows surface water molecules; darker blues represent groundwater molecules with increasing depth.

Although being able to visualize models with only one moving currency is instrumental in understanding the data, the outputs produced from complex simulations contain many currencies that not only interact with each other, but may also operate at different time steps. We developed the functionality to handle the visualization of multiple currency flows. Multiple currencies are necessary when modeling behavioral aspects of dynamic systems such as those found in the natural sciences. Multiple currency support allows scientists the ability to visualize the complex interacting behaviors inherent in complex adaptive hierarchical networks (CAHNs) and enhance the accuracy of the model.

Using the Digistar 4 system framework, we can also adjust many aspects of the currencies, such as color, opacity, and flow rate. For example, to visualize the data from a river system that includes two currencies, i.e., water flow and heat exchange, we enhanced the Digistar 4 with the functionality to turn the color of the water currency blue and the color of the heat currency red. This helps improve the interpretation of the visual data. Also, if a model contains multiple currencies but only one of those currencies needs to be analyzed, we can set the opacity of the unneeded currency to zero. This maintains the integrity of the model while giving us control over what we wish to visualize. Further, we also added the ability to slow or speed the rate at which currencies flow through the system.

Different currencies will likely move at different motion rates, so having the ability to set different motion rates is also an invaluable tool in visualizing this data. Because different currencies are input into Digistar 4 as separate data files, we are easily able to manipulate

individual attributes of the currencies. This work is still in progress, but has been initially realized in a simple visualization of a cube (Figure 3). The currencies represented in this cube are different colors and sizes, and are moving at different rates of speed.
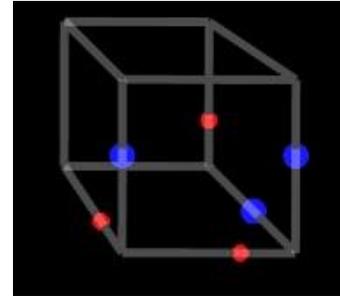
Fig 3. Visualization of multiple currencies through a cube-shaped matrix.

Other adjustable aspects of the model include changing the depth (z-scale) of the structure of the model. In the Nyack Floodplain model, where the depth of the water molecules is also important, exaggerating the z-scale allows ecologists to view the model in a different way. The floodplain is very long and wide, so in order to accurately represent it in its entirety, the depth appears to be fairly shallow. When the model is rotated so a cross-section can be seen, and we adjust the z-scale so the water-bed becomes deeper, then, not only can ecologists study the flow of the water molecules downstream, but they can also examine how deep the molecules flow into the gravel aquifer underlying the stream channel, and how long the water remains there.

OUTREACH

The use of the Digistar System allows for opportunities above and beyond those that accompany ordinary modeling software. The use of this software allows for a effective outreach opportunity that makes connections beyond the scope of our lab and research group. We have the ability to display our visualizations at a museum planetarium; a public venue with 3D projection capabilities and where 60,000 people pass through to view shows on a yearly basis. These capabilities provide us with the chance to communicate science and our research to those in academia as well as the general public, and to bring awareness to the dynamics of ecosystems and changing landscapes due to climate change or other factors. There is also the potential to target youth groups with the intent to stimulate interest in science and computing fields.

This outreach potential also allows for the interdisciplinary collaboration between not only

Computer Science, Land & Resource Sciences, and the museum, but also the School of Film & Photography. It is one of our future goals to make a professional documentary of the research that is currently taking place that will display current and future visualization products. These endeavors will be beneficial for communicating our research to the public, stakeholders, and groups whose mission is to advance the field of environmental science. One of these such groups is the Montana Institute on Ecosystems, a program created under the NSF EPSCoR Track I project whose mission is to stimulate research in the environmental sciences and engineering while addressing climate change effects in sustaining healthy ecosystems. Using software of this type to visualize these complex ecological models will help to understand links between landscape and processes in a way that will be visually stimulating and captivating. With the use of this software for our modeling purposes, one can ―step inside" the simulation of a river or ecosystem element and see for themselves a flow network in action or the effects of changes made within that network.

FUTURE WORK

There is still much work to be done with the visualization of multiple currencies. Currently, the only model featuring multiple currencies is the simple cube matrix. However, in order to create these multiple currency visualizations using real ecological data, we need to more seamlessly integrate the NEO simulation framework with the Digistar system. This would allow for the ability to output data directly from the NEO database and input the data into Digistar 4 with minimal intermediary formatting changes. There is also preliminary research focusing on the improvement of the accuracy of water molecule movement in hydrogeology models, similar to the one of the Nyack Floodplain.

As noted in the previous section, we would also like to extend the outreach of our visualization efforts. We plan to put together a short clip of the water molecule movements to be shown in the Taylor Planetarium. This clip would reach thousands of viewers of all ages and would highlight the work being done here at MSU.

REFERENCES

[1] Evans & Sutherland [online] 2011, http://www.es.com (Accessed: 25 March 2013).

[2] Buneman, P., Chapman, A., Cheney, J., 2006. Provenance Management in Curated Databases. ACM, pp. 539-550.

[3] Ebert, D., Favre, J., Peikert, R., ―Data Visualization." *Computers & Graphics* 26.2 (2002): 207-08. Print.

[4] Izurieta, C., Poole, G.C., Payn, R.A., Griffith, I., Nix, R., Helton, A., Bernhart E., Burgin, A.J., ―Development and Application of a Simulation Environment (NEO) for Integrating Empirical and Computational Investigations of System-Level Complexity." In Information Science and Applications (ICISA), 2012 International Conference on (pp. 1-6). IEEE.

[5] Google Earth [online] 2013, http://www.google.com/earth/index.html

[6] Walton, R., Chapman, R.S., Davis, J.E., ―Development and application of the wetlands dynamic water budget model," Wetlands 16:347-357, 1996.

[7] Poole G.C., Stanford J.A., Running S.W., Frissell C.A., Woessner W.W., Ellis B.K., ―A patch hierarchy approach to modeling surface and subsurface hydrology in complex flood-plain environments," Earth Surface Processes and Landforms 29:1259-1274, 2004.

[8] Helton A.M., Poole G.C., Payn R.A., Izurieta C., Stanford J.A., ―Relative Influences of the River Channel, Floodplain Surface, and Alluvial Aquifer on Simulated Hydrologic Residence Time in Montane River Floodplain," Elsevier Journal of Geomorphology. 2011.

APPENDIX C

A REPLICATION CASE STUDY TO MEASURE

THE ARCHITECTURAL QUALITY OF

A COMMERCIAL SYSTEM

APPENDIX C

A REPLICATION CASE STUDY TO MEASURE
THE ARCHITECTURAL QUALITY OF
A COMMERCIAL SYSTEM

Contribution of Authors and Co-Authors

Manuscript in Appendix C

Author: Derek Reimanis

Contributions: List Contributions Here, Single Spaced

Co-Author: Clemente Izurieta

Contributions: Provided feedback on early drafts of the manuscript and reviewed the final manuscript. Served as the instructor for technical debt, case study replications, and experimental design.

Co-Author: Rachael Luhr

Contributions: Wrote portion of first draft. Aided in development during early stages of the project. Provided feedback on early drafts of the manuscript and reviewed the final manuscript.

Co-Author: Lu Xiao

Contributions: Performed the initial study that was replicated. Served as CLIO expert and answered any questions regarding the tool.

Co-Author: Yuanfang Cai

Contributions: Performed the initial study that was replicated.

Co-Author: Gabe Rudy

Contributions: Served as software expert and industry liaison.

Manuscript Information Page

Derek Reimanis, Clemente Izurieta, Rachael Luhr, Lu Xiao, Yuanfang Cai, and Gabe Rudy

A Replication Case Study to Measure the Architectural Quality of a Commercial System

Derek Reimanis[7], Clemente Izurieta[7], Rachael Luhr[7], Lu Xiao[8], Yuanfang Cai[8], Gabe Rudy[9]

{derek.reimanis, clemente.izurieta, rachael.luhr}@cs.montana.edu, 01-406-994-3720
{lx52, yfcai}@cs.drexel.edu, 01-215-895-0298
rudy@goldenhelix.com, 01-406-585-8137

ABSTRACT

Context: Long-term software management decisions are directly impacted by the quality of the software's architecture. **Goal:** Herein, we present a replication case study where structural information about a commercial software system is used in conjunction with bug-related change frequencies to measure and predict architecture quality. **Method:** Metrics describing history and structure were gathered and then correlated with future bug-related issues; the worst of which were visualized and presented to developers. **Results:** We identified dependencies between components that change together even though they belong to different architectural modules, and as a consequence are more prone to bugs. We validated these dependencies by presenting our results back to the developers. The developers did not identify any of these dependencies as unexpected, but rather architectural necessities. **Conclusions:** This replication study adds to the knowledge base of CLIO (a tool that detects architectural degradations) by incorporating a new programming language (C++) and by externally replicating a previous case study on a separate commercial code base. Additionally, we provide lessons learned and suggestions for future applications of CLIO.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques – *Modules and interfaces*.

General Terms

Management, Measurement, Experimentation.

Keywords

Modularity violations, grime, technical debt, static analysis, architecture quality, case study, replication.

INTRODUCTION

Building confidence in previous results helps to increase the strength and the importance of findings. It is especially important to strive for external validation of results by independent researchers, as has been done by the replication study presented herein. To date, the field of Empirical Software Engineering lacks in the number of replication studies. Additionally, most of the existing guidelines found in the literature focus on formal experiments [3] [4] [7] [14]. In this paper, we present the findings of an external replication *case-study*. We present our results by borrowing from the existing experimentation terminology and we have structured our findings consistent with expected sections as delineated by Wohlin et al. [15].

The motivation behind this study stems from a desire to see if the techniques used by Schwanke et al. [13] to uncover architecture-related risks in a Java agile development environment (using architecture and history measures) can also be applied to a commercial C++ development environment. This is important because we wanted to evaluate the deployment of this technology in an industrial setting of a successful company with strict quality controls. We were also interested to see if the observations we make can be used to build consensus in explaining a form of architectural decay, where decay is defined as the structural breakdown of agreed upon solutions [6].

We applied CLIO [16], a tool designed to uncover modularity violations, to a commercial software system developed by a local bioinformatics company –Golden Helix[10]. The latter allowed us access to their software code base to investigate potential architectural disharmonies.

This paper is organized as follows: Section 2 discusses background and related work; Section 3 explains the importance of replication in empirical software engineering and our approach to classifying this study; Section 4 discusses the method followed by our replication; Section 5 explores how the method was carried out, including deviations and challenges encountered from the baseline method, results and developer feedback; Section 6 discusses the relation of our results to the baseline study. Section 7 discusses the

---

[1] Software Engineering Laboratory, Computer Science Dept., Bozeman, MT, USA

[2] Computer Science Dept., Drexel University, Philadelphia, PA, USA

[3] Golden Helix Inc., 203 Enterprise Blvd, Bozeman, MT, USA

[10] Golden Helix Inc.; http://www.goldenhelix.com

threats to validity in our study; and Section 8 concludes with lessons learned from this study and suggestions of future work.

BACKGROUND AND RELATED WORK

Modularity Violations

Baldwin and Clark [1] define a module as "a unit whose structural elements are powerfully connected among themselves and relatively weakly connected to elements in other units." Identifying violations in modules (hereafter referred to as modularity violations) is important because it allows developers to find code that exhibits bad structural design. Identifying such violations early in the lifecycle leads to proactive module refactoring. However, early detection of modularity violations is difficult because they do not always exhibit negative influences on the functionality of the software system. It is entirely possible

Table 1: Summary of different treatments between case studies		
Factor	*Baseline Project*	*Our Project*
Programming Language	Java	C++
# of Developers	Up to 20	Up to 11
Project Lifetime	2 years	4 years
# Source Files	900	3903 (1569 C++, 267 C, 2067 h)

for a system to function as intended, yet still contain modularity violations. If these violations are left uncorrected, they can lead to architectural decay, which would slowly cripple production.

Zazworka et al. [17] used the modularity violations findings from a CLIO case study and compared them to three other technical debt identification approaches. They found that modularity violations contribute to technical debt in the Hadoop open source software system. Technical debt [5] is a well-known metaphor that describes the tradeoffs between making short term decisions (i.e., time to market) at the expense of long term but high software quality (i.e., low coupling). The debt incurred during the lifetime of a software system can be measured as a function of cost (monetary or effort) with added interest. Often, debt happens because of quick and dirty implementation decisions –usually occurring when a development team is trying to meet a deadline. Technical debt is dangerous if not managed because it can result in a costly refactoring process. Techniques to slow down the accumulation of technical debt can benefit from early detection of modularity violations.

CLIO

CLIO was developed by Wong et al. [16] as a means to identify modularity violations in code. Wong et al. evaluated CLIO by running it on two different open source Java projects, Eclipse JDT[11] and Hadoop Common[12]. The results showed that hundreds of violations identified by CLIO were fixed in later versions of the software. CLIO finds violations within modules by looking not only at the source code of a project, but also at its version history. It helps developers identify unknown modular level violations in software. Although developers will identify some violations, specifically if the violations prove to be bothersome, the difficulty of finding all modularity violations is quite great. CLIO validates that its reports are useful by confirming that previously detected violations are indeed fixed in later versions of the software. The results that Wong et al. [16] obtained showed that CLIO could detect these modularity violations much earlier than developers who were manually checking for them. This means that CLIO can be used in software systems to identify modularity violations early in the development process to save time and money by not having to check for them manually.

Schwanke et al. [13] expanded upon this work by using CLIO on an agile industrial software development project. They looked specifically at the architectural quality of the software. They used a clustering algorithm to observe how files changed together without developer knowledge, and the impact that change had on the quality of the architecture, as measured by source code changes because of bugs. They reported several modularity violations to developers. The developers issued a refactoring because the modularity violations were (1) unexpected and (2) possibly harmful to their system. CLIO allowed them to see the exact number of files that were dependent on one another, and how those changes were affecting the structure of their project.

Replication in Software Engineering

Literature in the field concerning guidelines of replication studies only addresses experimental replication, not case study replication [4] [14]. Therefore, we have borrowed terminology from this literature to inform our work.

Importance of Replicating Case Studies

Experiment replication plays a key role in empirical software engineering [4] [14]. While many other domains construct hypotheses *in vitro*, software engineers are generally not granted that luxury. Empirical software engineering frequently involves humans, directly or indirectly, as experimental subjects, and human behavior is unpredictable and not repeatable

---

[11] The Eclipse Project; http://www.eclipse.org

[12] Apache Hadoop Common; http://hadoop.apache.org

in a laboratory setting. Coupled with the prohibitive costs of formal experimentation, software engineering empiricists must look for alternatives. Instead, we must rely on repeated case studies in various contexts to construct a knowledge base suitable for a scientific hypothesis. This process, while requiring exhaustive work, allows for consensus building that can provide the necessary support to generate scientific claims.

Categories of Replication

Shull et al. [14] discuss two primary types of replications; *exact replications* and *conceptual replications*. Exact replications are concerned with repeating the procedure of a baseline experiment as closely as possible. Conceptual replications, alternatively, attempt to use a different experimental procedure to answer the same questions as the baseline experiment. The study presented in this paper utilizes an exact replication method.

Shull et al. [14] divide exact replications into two categories: *dependent replications* and *independent replications*. In dependent replications, researchers keep all elements of the baseline study the same. In independent replications, researchers may alter elements of the original study. An independent replication follows the same procedure as the original study, but tweaks experimental treatments to come to the same or a different result. If treatments are changed and the same result is found, researchers can conclude that the treatment in question probably has little or no effect on the outcome. However, if changing a treatment leads to different results, that treatment needs to be explored further.

Using Shull's terminology, we categorized this study as an independent replication, with five major treatment differences from what would be considered a dependent replication. These differences are illustrated in Table 1. First, the baseline study used a software project written in Java as their only treatment to the programming language factor. In our case, the treatment is the C++ programming language. In other words, our study lies in the context of a C++ programming language, which may provide different results from the baseline. Second, the comparative sizes of the development groups differed. The baseline study featured a development group of up to 20 developers working on the project at any given point in time [13].The C++ system analyzed in this paper has had a total of eleven contributing developers in its four year lifetime. Third, the software project in the baseline study had been in development for two years, while the project covered in our study has been in development for four years. Finally, the project in the baseline study features 300 kilo-source lines of code (KSLOC) in 900 Java files. The project in our study has 1300 KSLOC across 3903 source files, of which 1836 have a .cpp/.c extension, and 2067 are header files. Surprisingly, both projects have a similar ratio of LOC per source file (333 LOC per source file).

Replication Baseline

In the selected baseline study, Schwanke et al. [13] reported on a case study that measured architecture quality and discovered architecture issues by combining the analysis of software structure and change history. They studied three structured measures (file size, fan-in, and fan-out) and four history measures (file change frequency, file ticket frequency, file bug frequency, and pair of file change frequency). Their study included two parts: 1) Exploring different software measures; and 2) Uncovering architecture issues using those measures.

1) Exploring different software measures: First, they explored the relationship between each pair of measures (structure and history) using Kendall's *tau-b* rank correlation [8], which showed the extent to which any two measures rank the same data in the same order. This study provided an initial insight on whether those measures were indicative of software quality, which was approximated by the surrogate file bug frequency. Then they studied how predictive those measures were of software faults. The data they used spanned two development cycles of the subject system, release 1 (R1) and release 2 (R2). They illustrated how predictive the calculated measures from R1 were for faults that appeared in R2 using Alberg diagrams [9].

2) Uncovering architecture issues: After validating the measures, they were used to discover architecture issues using three separate approaches. First, Schwanke et al. ranked all files by different measures –worst first. They found that the top ranked files (outliers) were quite consistent for different measures. They showed those outliers to the developers to obtain feedback about potential architecture issues; however, the developers gave little response because they could not visualize these issues. To generate responses from developers, they used a static analysis tool named *Understand*™[13] to visualize the position of those outliers in the architecture. Using this method, they were able to discuss many of the outlier files with the developers. In some cases, the developers pointed out how severe the problems were. Finally, they used CLIO to investigate the structure and history of pairs of files and grouped structurally distant yet historical coupled files into clusters. For each cluster, its structure was visualized using *Understand*™ in a structure diagram, which illustrated how clusters which cross-cut different architecture layers could be severe, and gave hints about why they were coupled in history.

Major Findings of the Baseline

Schwanke et al. found that by using CLIO they could identify, predict, and communicate certain architectural issues in the system [13]. They found that a few key interface files contributed to the majority of faults in the software. Additionally, they discovered that the file size and fan-out metrics are good predictors of future fault-

---

[13] Understand; http://www.scitools.com

proneness. In the absence of historical artifacts, files that contain high measures of these metrics typically have a higher number of architectural violations. Finally, unknown to the developers, some of these files violated modularity in the system by creating unwanted connections between layers. These violations were visualized and presented to the developers who issued a refactoring thereafter.

PROCEDURE

Following the procedure outlined in [13], our case study consisted of the following steps:

1) Data collection: The source code, version control history, and ticket tracking history of the software system in question were gathered.

2) Structure and history measurements: Measurements for common metrics were computed/collected across all versions of the software.

3) Validation: Measurements from the second-most recent release are correlated with fault measurements from the most recent release.

4) Prediction: Measurements from the most recent release are used to predict faults in upcoming future releases of the project.

5) Uncovering architecture problems: Measurements were sorted according to future fault impact and visualized. Outlier measurements present the most concern to system architecture quality, and were selected for further exploration.

6) Present findings to developers: Visualizations of the architecture of outlier modules were presented to developers with the intent of helping to realize the architectural quality of the system.

CASE STUDY

Setting

The project analyzed in this case study is named SNP & Variation Suite (SVS), and is the primary product of the bioinformatics company Golden Helix. We analyzed seven major releases of SVS.

SVS features 1.3 million lines of C++ source code spread out across 3903 source files. The project's structure is spread out across a total of 22 directories. In this study, we have chosen to define module as a directory, based on Parnas el al.'s definition [12]. We use the term directory and module interchangeably.

Eleven developers have contributed to this project over its four-year lifetime. The organization of the development group has an interesting hierarchy. The lead developer is also the Vice President of Product Development at Golden Helix. He plays a major role in not only developing SVS, but also in managing product development from a financial perspective. This means he has comprehensive knowledge of the software system when he makes management-related decisions,

and therefore, is more aware of the technical debt present in the software than business-oriented managers.

Motivation

This project was chosen for three reasons. First, Golden Helix is a local software company with its developing team in close proximity to the authors, and is well known for their generous contributions to the community. The process presented in this study is a great opportunity to inform Golden Helix of the architectural quality of their flagship software. Second, applying the CLIO tool in different commercial settings will help future applications of CLIO. By clearly outlining the strengths, weaknesses, and lessons learned at the end of the study, we hope to improve future applications of CLIO. Finally, no previous study that follows this methodology to detect modularity violations has considered a C++ project. Previous studies such as [16] [17] only looked at non-commercial Java projects. Using the C++ programming language as a treatment in this sense builds on the knowledge base of CLIO, extending what we know about this method.

Data Collection

Golden Helix strongly encourages developers to commit often, and keep commits localized to their section of change. These commits are stored in a Mercurial[14] repository, and the FogBugz[15] tool is used to track issues. Golden Helix switched repositories, from Apache Subversion (SVN)[16] to Mercurial, and ticket tracking tools, from Trac[17] to FogBugz, during the lifetime of SVS. Because this study focuses on the entirety of the project's lifetime, both the SVN repository and Trac ticket logs have been recovered and treated in the same manner as the current system. Each developer is expected to include references to ticket cases in their commits.

Similar to [13], the repository logs and issue tracking logs were extracted into a PostgreSQL[18] database. This allowed us to search for historical data using simple SQL queries.

We have grouped C/C++ source files and header source files together in this study. That is, for each C/C++ source file and its corresponding header file(s), the files are considered one and the same. For the remainder of this case study, we refer to the C/C++ source and corresponding header file pairs as a *file pair*. Measurements made in both files are aggregated together. There is a reason for doing this. Developers of SVS demand that source files and their corresponding

---

[14] Mercurial SCM; http://mercurial.selenic.com/
[15] FogBugz Bug Tracking;
https://www.fogcreek.com/fogbugz/
[16] Apache Subversion;
http://subversion.apache.org/
[17] Trac; http://trac.edgewall.org/
[18] PostgreSQL; http://www.postgresql.org/

header files be kept together in the same directory. When either a source file or a header file changes, the developers are expected to update the signatures in the corresponding file. This implies that any changes made to the latter are expected and hence do not constitute modularity violations. Our study is concerned with locating *unexpected* changes in modules of code. Therefore, including any information about header/source pairs changing together will lead to useless information.

Structure and History Metrics

Following the work of Schwanke et al. [13], the following metrics were gathered for all file pairs (*u)* across all seven versions of the software:

1. *File size*: The aggregated file size on disk of both elements in *u*, measured in bytes.

2. *Fan-in*: Within a project, *fan-in* of *u* is the sum of the number of references from any *v* (where *v* is defined identically similarly to *u*) pointing to *u*.

3. *Fan-out:* Within a project, *fan-out* of *u* is the sum of the number of references from *u* that point to any *v* (where *v* is defined identically similarly to *u*).

4. *Change frequency*: The number of times that any element in *u* is changed, according to the commit log. Commits where both elements of *u* are changed are only counted once.

5. *Ticket frequency*: The number of different FogBugz or Trac issue tickets referenced for which either element in *u* is modified. If both elements in *u* are modified with a reference to the same issue ticket, it is only counted once.

6. *Bug change frequency*: The number of different FogBugz or Trac **bug** issue tickets referenced for which either element in *u* is modified. If both elements in *u* are modified with a reference to the same **bug** issue ticket, it is only counted once.

7. *Pair change frequency*: For each file pair, *v*, in the project, the number of times in which *u* and *v* are modified in the same commit.

Validation

In an effort to validate the significance of our metric choices, several exploratory data analysis techniques were utilized. These include histogram inspection, scatter plot analysis, and correlation analysis. Although the system in question has gone through seven releases, in this paper we only present the results from the most recent release (release 7.5) and the release immediately preceding the most recent release (release 7). Hereafter, we refer to release 7.5 as the *present* state of the software, and release 7 as the *past*.

Similar to the baseline study, we found that data analysis across all other releases showed very similar results. The baseline study chose to focus their work on the most recent releases, because it is more representative of the system in the present time, and may provide better predictive power. We have followed suit because of the same reasons.

1) Histogram analysis

Histograms were generated for each metric in question. We focused on identifying distributions of each metric across releases. From the distributions, we identified outlier file pairs which Schwanke et al. [13] states are more prone to unexpected changes. For example, Figure 1 illustrates the change frequency metric across all releases of the software. The y-axis is shown as a logarithmic scale in base 4 to preserve column space. There is a typical exponential decay curve, suggesting that the majority of file pairs experienced few changes. However, there exist outliers with more than 180 changes per file (not shown, but aggregated to form the bin at x=180). This suggests that a surprising number of pairs (about 60) experience more than 180 changes. This is congruent with findings from [13] and their histogram analysis.
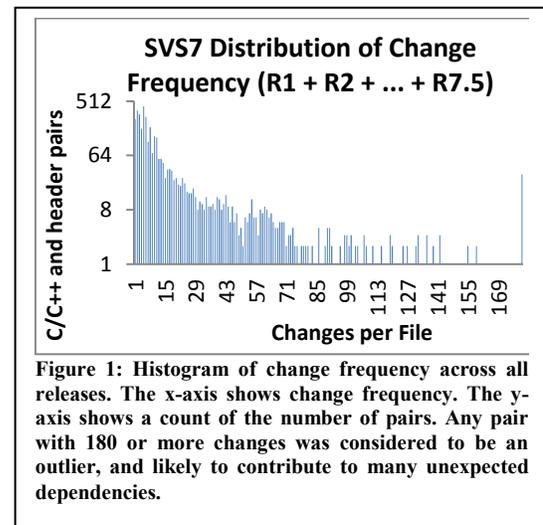
Figure 1: Histogram of change frequency across all releases. The x-axis shows change frequency. The y-axis shows a count of the number of pairs. Any pair with 180 or more changes was considered to be an outlier, and likely to contribute to many unexpected dependencies.

2) Scatter Plot Analysis

Scatter plots were constructed for each metric gathered. When constructing scatter plots, we plotted the measure in release 7.5 on the y-axis and the measure of other metrics from release 7 on the x-axis. This gave us the opportunity to identify a possible relationship between past and present measurements. Figure 2 shows a scatter plot of change frequency in release 7.5 versus fan-out in release 7. There appears to be a slight linear correlation between the two, suggesting that change frequency in future releases can be predicted from fan-out in current or past releases.

This graph suggests that the fan-out of current or past file pairs may be used to predict the change frequency of the pair in the future. Our scatter plot analysis

Table 2: *Tau-b* values for metric pairs

Tau-b table of metrics for svs7 + svs7.5						
R7+R7.5	fan-in	fan-out	file size	changes	tickets	bugs
fan-in	1	0.257	0.301	0.331	0.328	0.464
fan-out	0.257	1	0.441	0.417	0.416	0.637
size	0.301	0.441	1	0.293	0.273	0.510
changes	0.331	0.417	0.293	1	0.972	0.858
tickets	0.328	0.416	0.273	0.972	1	0.857
bugs	0.463	0.637	0.510	0.858	0.857	1

provided similar results as the baseline study by Schwanke et al [13].

3) Correlation Analysis

Rank-based correlation analysis was performed on the data to identify possible relationships between measurements in one release and fault measurements in a future release. Per the baseline study, we used the Kendall's *tau-b* rank correlation measure [8]. This non-parametric test was chosen instead of a Spearman or the parametric Pearson test because many of the values fall near zero. The Ordinary Least Squares (OLS) method of Spearman or Pearson performs poorly when many values fall near zero.

Kendall's *tau-b* value is found in a two-step process. First, the measurements taken from two metrics are ordered according to their values. Second, a calculation is performed which counts the number of values which appear in the same order. The calculation is shown below:

$$\tau_B(F,G) = \frac{concord(F,G) - discord(F,G)}{concord(F,G) + discord(F,G)}$$

Where *F* and *G* are two orderings of values taken from a file pair. *concord(F,G)* is a count of the number of times values appear in the same order. Alternatively, *discord(F,G)* is a count of the number of times values appear in different order. For this test, values of 0 in either *F* or *G* are ignored; that is, they are not counted by either *concord* or *discord*. The value produced falls in range [-1, 1], corresponding to the correlation between the orderings. A value of 1 indicates a perfect linear correlation. For the purpose of this study, and in agreement with [11], we consider

Figure 2: Scatter plot of release 7.5 change frequency and release 7 fan-out. Each data point represents a C/C++ and header pair. The x-axis plots the fan-out of pairs in release 7. The y-axis plots the change frequency of each pair in release 7.5. $R^2 \approx 0.2658$
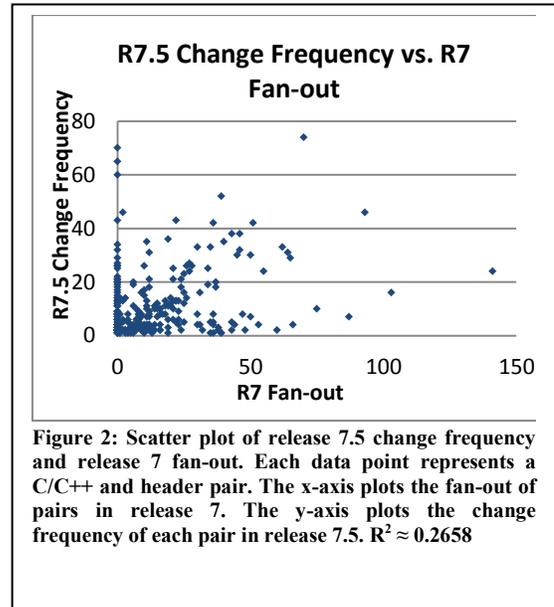
values at 0.6 or greater to be strong. Because this is a non-parametric statistical test, we cannot assume a normal distribution fits the data. Therefore, we cannot find an associated *p*-value for each *tau-b* value.

Table 2 shows the *tau-b* value calculated for each metric pair in release 7 and release 7.5. Each cell corresponds to the *tau-b* value as found by the previously described equation. The table is symmetric because the comparison of two ranked metric values is a symmetric property. Highlighted cells indicate a strong correlation.

The highlighted values in the bottom right quadrant of the table are expected correlations. The values report that, for example, as ticket frequency increases, bug change frequency increases as well. This is logically consistent because as developers add more tickets to their commits, more of these tickets will contain bug references. However, the correlation value for bugs vs. fan-out is an unexpected result. This number tells us that as the fan-out of a file pair increases, the number of bugs associated with that pair increases as well. Similar

results were found by [13], adding more power to hypothesis that fan-out and number of bugs increase together.

Using these three methods of exploratory data analysis, we identified likely correlations between metrics. In the validation step we analyze these correlations to see if they are indicative of bug-related changes in the future.

Prediction

Ostrand and Weyuker [10] introduced *accuracy*, *precision*, and *recall* measures from the information retrieval domain. We use various *recall* metrics to validate our prediction of future bugs. Recall is defined as the percentage of faulty files that are correctly identified as faulty files. As in the baseline case study, we calculate recall in three different ways. For every file pair *u*,

Faulty file recall: An instance occurs when either element in *u* is changed at least once in the release representing the future due to any bug ticket.

Fault recall: An instance is a tuple defined as <*u*, bug ticket reference>, where *u* is changed at least once due to the same bug ticket.

Fault impact recall: An instance is a triple defined as <*u*, commit number in the source control logs where *u* is changed, bug ticket reference> where the bug ticket is referenced in the same commit where *u* is changed in.

These three recall measures apply different emphasis to future fault prediction. *Faulty file recall* emphasizes future fault prediction least, because it treats all future bug-related changes to *u*, regardless of the number of instances, as one. This fails to capture instances where *u* is associated with more than one bug ticket. However, *Fault recall* does take this into account, because it considers multiple bug ticket references in an instance. Furthermore, *Fault impact recall* provides the highest granularity to allow for future fault prediction because it takes into account all changes *u* goes through. All three recall measures form an implied subsumption hierarchy.

Using these recall measures, we use Alberg diagrams [9] to plot release 7 measurements vs. release 7.5 faults. Alberg diagrams are based on the pareto principle, that roughly 20% of the files in a system are responsible for 80% of the faults. In this context, we use this same principle to estimate the accuracy of prediction models [9].

Figure 3 illustrates one Alberg diagram for this system. The x-axis shows 60 C/C++ source and header pairs, *u*, ordered in descending order according to their metric values from release 7. These 60 file pairs are selected based on their contribution to bug-related changes in release 7.5. The bug change frequency for *u* in release 7.5 is plotted on the y-axis. Any given point on the curve represents a C/C++ source and header pair. The oracle curve is a perfect predictor of release 7.5 bug change frequency for all *u*. As other curves get nearer to

the oracle curve, their accuracy for predicting release 7.5 bug change frequency increases.

The oracle curve from this Alberg diagram states that roughly 20% (actually 23.3%) of C/C++ source and header pairs contribute to 80% of bug change frequency in release 7.5. The values of fan-out and change frequency in release 7 for these pairs contributed from 40% to 50% of bug changes in release 7.5. These findings are slightly less than Schwanke et al.'s findings, yet are still noteworthy. This validates that selected metrics from earlier releases can be used to predict bug change frequency in future releases.
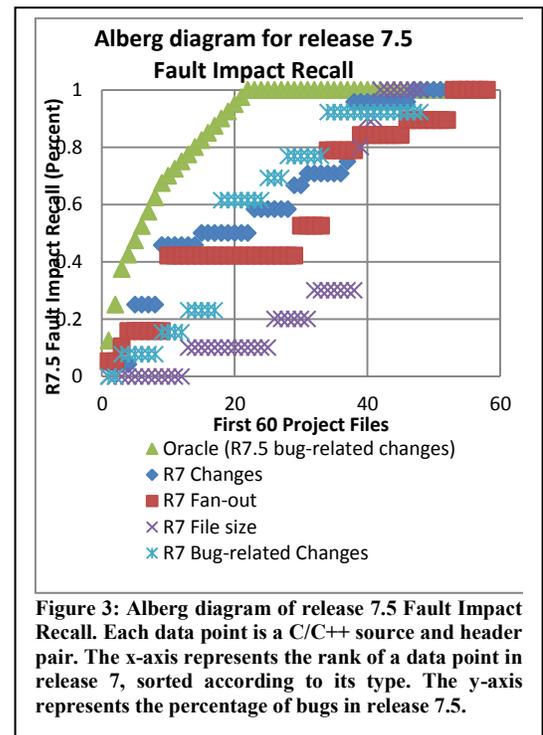
Figure 3: Alberg diagram of release 7.5 Fault Impact Recall. Each data point is a C/C++ source and header pair. The x-axis represents the rank of a data point in release 7, sorted according to its type. The y-axis represents the percentage of bugs in release 7.5.

Uncovering and Visualizing Architecture Problems

Once these measures have been validated as capable of predicting future faults, the problem of identifying file pairs which are more prone to unexpected changes arises. Next, we study the extent to which these pair affects other quality measures.

We utilized the static code analysis tool *Understand*™ to visualize graphs of interdependent components. *Understand*™ is a commercial product developed by Scientific Tools, Inc.13. *Understand*™ can find many structural features of code, including dependency listings of how pairs of C++ files depend on one another. Through visualization, we can analyze the extent to which these dependencies affect other pairs in the software system.

These graphs help differentiate expected and unexpected dependencies. If dependencies occur

between two pairs that are in the same module, we treat them as *expected* dependencies, consistent with the baseline study. This is based on the assumption that developers group files or classes together based on similar functionality. *Unexpected* dependencies are treated as dependencies that occur across different modules, also consistent with our baseline study. Our definitions of expected and unexpected dependencies were validated by the developers at Golden Helix.

Because we are concerned with how these dependencies are changing together, we define a ―distant‖ and ―local‖ change pair. Using Schwanke et al.'s [13] definitions, a pair of file pairs that change together, *change pair,* <u,v> is local if (1) *u* directly depends on *v*, (2) *v* directly depends on *u*, or (3) *u* and *v* belong to the same module. Any change pair which does not fit under this definition is a *distant change pair*.

Figure 4 illustrates a high level view of the dependencies between modules in SVS. Nodes in the graph represent modules, and edges represent dependencies between modules. The number on the edge refers to the exact number of dependencies. The modules shown contain the ten most frequent distant change pairs. This graph is nearly a complete graph, suggesting that modules have high coupling when distant change pair frequency is high.

Once change pairs have been classified as either local or distant, CLIO is used to (1) identify change pairs which historically have changed together frequently, and (2) cluster these pairs according to the scope of their change pair (local or distant). To identify frequent historic change pairs, we mine the PostgreSQL database built in the procedure described by section 5.1. To cluster the pairs, a ―single link‖ clustering algorithm is used [13].

Figure 4: High level view of the dependencies between modules containing the *ten* most frequent distant change pairs. The numbers on the edges represent the number of times all file pairs in a module depend on another module.
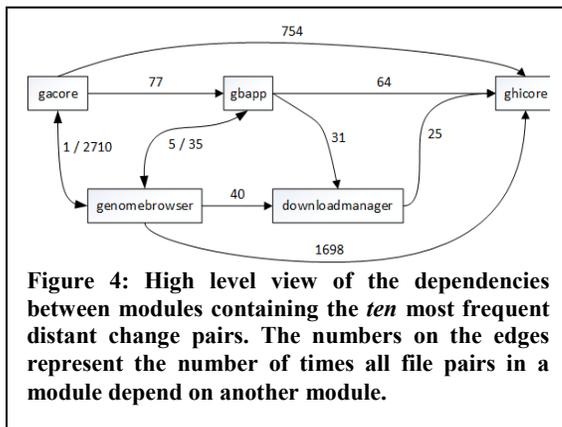
The clustering algorithm groups distant change pairs as follows: For each frequent, distant change pair <u, v>, cluster *u* and *v* together. Then, add all the local dependencies which contain either *u* or *v* to the cluster. We generated visualizations of these clusters that illustrate the number of dependencies across distant change pairs and presented these visualizations to developers.

Presenting Results to Developers

Visualizing architectural dependencies with graphs provided us with a convenient and intuitive medium that could be validated with developers. We presented all our data to the lead developer at Golden Helix. In summary, the lead developer at Golden Helix was not surprised by our findings. He indicated that several outlier file pairs were contributing to the majority of modularity violations in the code base. It was these pairs that also contributed to a large number of bugs in the most current releases. The lead developer was well aware of this, and more or less the extent to which this affected other files.

The majority of modularity violations and bugs occurred in packages representing highly customizable components of the SVS executable. These packages include the UI component, the core component, and a component that is concerned with reading in a large variety of complex file formats. We noticed that file pairs in these packages both heavily depend on and were depended upon by many others (i.e., they have high efferent and afferent coupling). However, the structure observed was the choice of the developers. The developers utilized these pairs as access points, or common files to reference when one component needed to be used. When these access point pairs were changed, they incurred a slew of changes in other modules in the system because of numerous, propagating dependencies. The developers saw this method as a necessary step in their development lifecycle.

DISCUSSION

The process of using CLIO to detect and measure architectural quality of software needs to be matured further. Developers were not surprised by the findings of CLIO, primarily because the findings pointed out known problems. Many of these problems are due to the many connections that exist between modules. From an academic sense this is a problem, because it is preferable to have few connection points between modules (coupling). Lower coupling between modules is indicative of better design, and helps localize possible future changes as well as allows for increased quality attributes (such as understandability) [2]. However, from the developers' perspective, familiarity with the code base was more important than traditional good design. The developers are content leaving the coupling between modules as is, because it makes the most sense for the SVS system. This finding is very interesting because it gives the impression that the results from tools such as CLIO should be system dependent. That is, although the results may appear useful, nothing can be learned unless an in-depth assessment of the software system in question has been made. These conclusions cannot be reached without evaluating and deploying laboratory tools in commercial grade environments.

We did find very similar results to the baseline, which is promising in helping extend power of the hypothesis that certain metrics can be used as better predictors of

software quality. We found that a select few files contributed to many modularity violations, and greatly influenced the number of bugs. While in our case the developers were not surprised by the results, the results are promising in that they clearly identify problem files in code. The baseline found that developers were not always aware of these modularity violations. In cases where developers may not be fully familiar with the structural connections across modules in their code base, this procedure provided significant insights.

We also identified and validated cases where structural metrics can be used as quality predictors for future releases. Both this study and Schwanke et al. [6] concur that the fan-out metric is a good predictor of future faults, as verified by correlation analysis and Alberg diagrams.

THREATS TO VALIDITY

There are several threats that threaten the validity of this study. One developer brought up the argument that, ―If a developer prefers to commit files more frequently than other developers, it would show up in the commit logs as having few change pairs. This would give misleading results because it would provide cases where too few files are being committed to account for changes across modules, or too many files are being committed which would make it appear that more dependencies exist.‖ This is a direct threat to the construct validity of our study. Although the developer‗s observation is correct, it did not have a large impact on our results. We identify files showing up in the commit logs together with a high frequency, and ignore cases where paired changes happen infrequently. This reinforces that such cases as described by the developer are unlikely to occur often. Regardless, the observation does shed light into a situation that will be mitigated in future studies.

A second threat to the construct validity is the fact that we grouped C/C++ source file and corresponding header files together. These file pairs consist of the aggregated information from their combined elements. Although a threat, it is mitigated by the following reason. The developers brought to our attention that both elements in the file pair are expected to belong to the same package, and are expected to change together. That is, if a C++ source file is updated, the developers expect to make changes to the signature of the header source file as well. Because both of these cases are expected changes, including both files separately in the study would be spurious information. Thus, we chose to group every C++ source and corresponding header file together.

A third threat to the construct validity of this study is the assumption that developers tag bugs correctly in the commit messages. As an external observer, the only method we have of identifying past-bugs in the software project is through analyzing historical artifacts. Therefore, we need to rely on the discipline of developers to (1) tag the bugs they focused on in a commit and (2) tag the bugs correctly. We have no way of knowing if either of these two conditions is not met.

External validity represents the ability to generalize from the results of a study. In this instance, we cannot generalize the results we found to other contexts. In other words, the results found in this study and the baseline only hold true for our specific contexts, however they helped in building consensus around our findings across different programming languages in commercial agile development environments. More replication studies are necessary to increase the power of these results.

CONCLUSION

This replication case study was performed to help us analyze how structural file metrics could be correlated with system quality, and to help us comprehend if similar observations performed in a Java commercial product could also be observed in its C++ counterpart. We have gathered structural metrics and identified correlations between them and future bug problems. We identified a select few outlier files which contribute to the majority of future bug problems. From these, we collected dependencies and visualized how extensively problems may propagate. We showed this information to the developers of Golden Helix and they were not surprised by the results. Rather than attempt to entirely eliminate distant-modules with frequently-changing dependencies, the developers preferred to keep a select-few files as connection points. When asked why, the lead developer explained that these connection points offer a single point of entry into a module. Any changes between modules would be reflected in the connection points only. The developers would rather be aware of a few files that are frequently problematic than issue a refactoring.

Challenges

Herein we describe some of the challenges we encountered while trying to perform this study.

1) Specific Tools: The baseline study featured the use of the commercial tool *Understand*™ for static analysis of code to gather metrics as well as to visualize results. Although the static analysis and visualizations provided high quality analysis, it is nearly impossible to replicate this case study without the use of this specific tool. Alternatives were considered, but the mechanistic formula used for analyzing files needed to be used as is, as other approaches would have constituted (in the opinion of the authors) a significantly large deviation from the baseline method that we would not have been able to call this a replication study.

2) Understanding the System: While we hope that manually performing the CLIO process eventually leads to an automated approach, this study suggests that such a hope may be far-fetched. Ultimately, a complete understanding of the system in question is necessary before any significant value can be taken from this tool.

Our results mean very little unless the developers actually make use of them.

3) Literature Coverage: The majority (entirety) of literature covering replications in Empirical Software Engineering refers to formal experiments, not case studies. We have borrowed the terminology from such literature in this study. This situation is not ideal because case studies have less power than formal experiments and therefore should be approached differently. Peer-reviewed literature needs to be published which outlines case study replication guidelines.

ACKNOWLEDGMENTS

REFERENCES

[1] Baldwin, C. and Clark, K. 2000. *Design Rules: The power of Modularity*. Vol. 1. MIT Press., Cambridge, MA.

[2] Bansiya, J. and Davis, C. G. 2002. A hierarchical model for object oriented design quality assessment. In *IEEE Transactions on Software Engineering* 28, 1 (Aug. 2002), 4-17. DOI=http://dx.doi.org/10.1109/32.979986.

[3] Basili, V. R., Selby, R. W., and Hutchens, D. H. 1986. Experimentation in Software Engineering. In *IEEE Transactions on Software Engineering* 12,7 (July 1986), 733-743. DOI=http://dx.doi.org/10.1109/TSE.1986.6312975.

[4] Brooks, A., Roper, M., Wood, M., Daly, J., and Miller, J. 2008. Replication's Role in Software Engineering. In *Guide to Advanced Empirical Software Engineering*, Shull, F., Singer, J., and Sjøberg, D. I. K. Springer London, Springer, 365-379. DOI=http://dx.doi.org/10.1007/978-1-84800-044-5_14.

[5] Cunningham, W. 1992. The Wycash portfolio management system. In *OOPSLA '92 Addendum to the proceedings on Object-oriented programming systems, languages, and applications* (Dec. 1992). OOPSLA '92. SIGPLAN ACM, New York, NY 29-30. DOI= http://dx.doi.org/10.1145/157709.157715.

[6] Izurieta, C. and Bieman, J. 2013. A multiple case study of design pattern decay, grime, and rot in evolving software systems. In *Software Quality Journal*, 21, 2 (June 2013), 289-323, DOI=http://dx.doi.org/10.1007/s11219-012-9175-x.

[7] Juristo, N. and Moreno, A. M. 2010. *Basics of Software Engineering Experimentation* (1st ed.). Springer Publishing Company, Incorporated.

[8] Kendall, M. G. 1938. A new measure of rank correlation. In *Biometrika*, 30 (1938), 81-93.

[9] Ohlsson, N. and Alberg, H. 1996. Predicting fault-prone software modules in telephone switches. In *IEEE Transactions on Software Engineering*, 22, 12 (Dec. 1996), 886-894, DOI= http://dx.doi.org/10.1109/32.553637.

[10] Ostrand, T. J. and Weyuker, E. J. 2007. How to measure success of fault prediction models. In *Fourth international workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting* (SOQUA '07). ACM, New York, NY, USA, 25-30. DOI= http://doi.acm.org/10.1145/1295074.1295080.

[11] Ott, R. and Longnecker, M.1993. *An introduction to statistical methods and data analysis*. Vol. 4. Duxbury Press, Belmont, CA.

[12] Parnas, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15, 12 (December 1972), 1053-1058.

[13] Schwanke, R., Xiao, L., and Cai, Y. 2013. Measuring architecture quality by structure plus history analysis. In *2013 35th International Conference on Software Engineering (ICSE)* (San Francisco, CA, May18 - 26 2013). ICSE '13. IEEE, San Francisco, CA, 891-900. DOI= http://dx.doi.org/10.1109/ICSE.2013.6606638.

[14] Shull, F. J., Carver, J. C., Vegas, S., and Juristo, N. 2008. The role of replications in Empirical Software Engineering. In *Empirical Software Engineering* 13, 2 (April 2008), 211-218. DOI= http://dx.doi.org/ 10.1007/s10664-008-9060-1.

[15] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. 2012 *Experimentation in software Engineering*. Springer Berlin Heidelberg. DOI= http://dx.doi.org/10.1007/978-3-642-29044-2.

[16] Wong, S., Cai, Y., Kim, M., and Dalton, M., 2011. Detecting software modularity violations. In *Proceedings of the 33rd International Conference on Software Engineering* (ICSE '11). ACM, New York, NY, USA, 411-420. DOI= http://doi.acm.org/10.1145/1985793.1985850.

[17] Zazworka, N., Vetro, A., Izurieta, C., Wong, S., Cai, Y., Seamon, C., and Shull, F. 2013. Comparing four approaches for technical debt identification. In *Software Quality Journal* (April 2013), 1-24, Springer US.

APPENDIX D

INPUT AND OUTPUT TABLES FROM THE

SIMPLIFIED TESTING GRAPH FOR THE

PARTICLE TRACKER ALGORITHM

Matrix Table for Test Graph

id	from_id	to_id	length
1001	1	2	160.234
1003	3	2	291.892
1004	2	4	191.111
1005	2	5	355.769
1006	4	6	80.32
1007	4	7	307.25
1008	5	8	203.768
1009	5	9	105.89
1010	1	10	283.702
1011	2	10	234.119
1012	5	10	267.88

Model Results Table for Test Graph

secs	uid	flux	velocity
0	1001	3.02	0.2436
0	1003	28.63	0.2213
0	1004	6.139	0.2213
0	1005	14.876	0.386
0	1006	0	0
0	1007	0	0
0	1008	2132.6	0.0123
0	1009	101.75	0.0015
0	1010	4.312	0.369
0	1011	2.12	0.212
0	1012	8.291	0.101
43200	1001	12.987	0.2549
43200	1003	28.789	0.219
43200	1004	6.285	0.219
43200	1005	14.93	0.402
43200	1006	0	0
43200	1007	0	0
43200	1008	2133.4	0.0234
43200	1009	0	0
43200	1010	4.486	0.378
43200	1011	2.23	0.223

43200	1012	8.367	0.128
86400	1001	12.34	0.2499
86400	1003	28.834	0.21359
86400	1004	6.35	0.21359
86400	1005	15.012	0.438
86400	1006	0	0
86400	1007	3.215	0.40839
86400	1008	2133.7	0.0345
86400	1009	0	0
86400	1010	4.679	0.384
86400	1011	2.26	0.226
86400	1012	-8.73	-0.156

Condition 1: Particles had to choose links correctly based on weighted flux with some element of randomness. The actual percentages of particles that follow certain links are very close to their expected percentages.

First juncture:	Expected:	Actual:
Link 1001	75.12%	73%
Link 1010	24.87%	27%
Second juncture:	Expected:	Actual:
Link 1011	9.16%	12.32%
Link 1004	26.53%	28.77%
Link 1005	64.3%	58.9%
Third juncture:	Expected:	Actual:
Link 1008	95.09%	97.26%
Link 1009	4.53%	1.37%
Link 1012	0.37%	1.37%

Condition 2: If the velocity reaches 0 in the middle of the link, the particle would stop moving, but would keep reporting its position. At time 43200, link 1009 loses velocity. The particle stops moving but continues reporting its position.

particleid	currentlinkid	currenttime	currentlocation
5	1009	42700	62.3352
5	1009	42800	62.4852
5	1009	42900	62.6352

5	1009	43000	62.7852
5	1009	43100	62.9352
5	1009	43200	63.0852
5	1009	43300	63.0852
5	1009	43400	63.0852

Condition 3: If a particle comes to a junction where none of the available outgoing links have a flux value greater than 0, the particle will wait at the junction until the flux becomes greater than 0. Link 1007 gains flux at time 86400. The particle waits at the beginning of the link until this happens.

particleid	currentlinkid	currenttime	currentlocation
25	1007	86000	0
25	1007	86100	0
25	1007	86200	0
25	1007	86300	0
25	1007	86400	40.839
25	1007	86500	81.678
25	1007	86600	122.517
25	1007	86700	163.356
25	1007	86800	204.195
25	1007	86900	245.034
25	1007	87000	285.873

Condition 4: If a particle comes to a junction and there is a link with positive velocity but the link is the wrong direction, the particle should never choose to go down this link. In all runs that were completed, link 1003 was never taken.