EvoJava: A Tool for Measuring Evolving Software

Joshua Oosterman[1], Warwick Irwin[2], Neville Churcher[3]
Department of Computer Science and Software Engineering
University of Canterbury
Private Bag 4800, Christchurch, New Zealand
jjo54@uclive.ac.nz[1], {warwick.irwin[2], neville.churcher[3]}@canterbury.ac.nz

Abstract

This paper introduces EVOJAVA, a new tool for extracting static software metrics from a Java source code repository. For each version of a program, EVOJAVA builds a comprehensive model of the semantic features described by Java code (classes, methods, invocations, etc), and tracks the identity of these features as they evolve through sequential versions. This allows traditional software metrics to be recorded over time without losing traceability of software components, and permits calculation of new metrics that characterise the software evolution itself.

Keywords: Software metrics; software evolution; source repository mining; software visualisation.

1 Introduction

Software systems are commonly too large and complex for an individual to fully comprehend. Software product metrics attempt to mitigate the problem by aggregating and abstracting detail in order to expose salient characteristics of programs. Static software product metrics quantify aspects of programs that can be observed in source code (or other static artefacts such as UML diagrams). Examples include measures of size, complexity, coupling, and so on.

Although many software metrics are proposed in the literature, they have seen limited use in the software development industry. Many developers remain sceptical about their claimed benefits (Medha and Carolyn, 2008). Further work is needed to make metrics more useful, and new tools are needed to make measurement easier.

This paper describes EVOJAVA, a new tool that relies on static analysis of Java source code to measure software structure. EVOJAVA builds on earlier tools that could measure a snapshot of source code at some point in time, but which did not specifically support repeated measurement of a program's code as it evolved (Oosterman J. and Churcher, 2010, Irwin, 2007). A particular goal of EVOJAVA is the ability to preserve the identity of semantic software features (classes, methods, statements etc) across versions, despite renaming, moving or refactoring of parts of the code.

By adding the dimension of time to our models, we hope to enable further research that will clarify the behaviour of metrics over time and lead to improvements in the predictive power of metrics and ultimately increase their usefulness for code comprehension and project management. We would like to investigate, for example, whether source code refactoring – which aims to improve readability, understandability, extensibility or performance of code without modifying its functionality – does in fact lead to improved software quality measurements.

The inclusion of time in our models also enables a family of metrics that quantify aspects of software evolution, such as the lifetime of features or their rate of change. These evolution metrics should help to answer questions such as whether software quality metrics can be used to predict the likelihood that software maintenance will later be required.

Information overload is already a challenge for software engineers, and the inclusion of another dimension of measurement data compounds the problem. Visualisation techniques and software tool support are essential for effectively communicating measurements to developers. A visualisation prototype is included in this paper.

The rest of the report is structured as follows:
Section 2 reviews related technologies in the literature. Section 3 describes the design and implementation of EVOJAVA. Section 4 presents results collected from real world software by the tool, and related visualisations.

2 Background

2.1 Software Metrics

Among the best known metrics for Object-Oriented (OO) programming are the Chidamber and Kemerer (CK) metrics suite (Chidamber and Kemerer, 1994) and the MOOD (and later MOOD2) suite (Abreu and Melo, 1996). Popular CK and MOOD metrics include W*eighted Methods per Class* (WMC), *Depth of Inheritance Tree* (DIT), *Coupling Between Objects* (CBO), and *Attribute Hiding Factor* (MHF).

Many subsequent papers have addressed the CK and MOOD suites, with particular focuses on extending the set or validating their use (Chahal and Singh, 2009, Nagappan et al., 2005). A criticism of these metrics (and others), is that they are underspecified. The few industry applications built to measure metrics differ in the way they compute their values (Lincke et al., 2008). Even counting the number of methods in a class is not trivial; should inherited methods, constructors, or overloaded methods be counted? In order to be reproducible, metrics research must include precise specifications of metrics.

Older complexity metrics such as McCabe's cyclomatic complexity (McCabe, 1976) and NPATH (Nejmeh, 1988) can be run on procedural programs or functions, but remain applicable to OO programs. Both of these metrics measure aspects of a program's control flow graph, an acyclic graph representation of the possible control paths through a program. More nodes or paths in this graph should indicate a higher complexity.

The type of metrics that can be calculated is limited by the richness and completeness of the model used. This is discussed further in Section 2.2.

2.1.1 Automatic OO Design Evaluation

Since the introduction of Object Oriented (OO) programming, many informal OO guidelines have been proposed to increase quality. Examples include OO design heuristics (Riel, 1996) and *Code Smells* (Fowler, 1999). To varying degrees, these guidelines can be measured with quantitative methods using software metrics and rules. For example, the *Large Class Smell* could possibly be detected by counting lines of code, number of methods or number of instance variables. The *Acyclic Dependencies Principle* could be enforced with a topological sort algorithm. Others are impossible to automate, such as Riel's *Model the Real World* heuristic, which requires semantic knowledge of the problem domain.

A system called CODE CRITICK (OOSTERMAN J. AND CHURCHER, 2010) was built in our previous work to automate many of these rules. CRITICK evaluates Java programs, and provides a ranked list of violations. These metrics and rules were built on the JST semantic model (Irwin, 2007), and can be used to quantify aspects of quality in our software evolution research.

2.2 Static Analysis and Semantic Models

Static analysis involves the analysis of software artefacts, such as source code. Most modern Integrated Development Environments (IDEs) analyse source code to provide features such as syntax highlighting, auto completion and automated refactoring. Some compilers such as the Java compiler also provide more complex checking such as dead code detection. Raw source code alone is not suitable for such complex static analysis, so it must be processed first.

A file of source code in its simplest form is simply a string of characters which represents some sentence in a particular language's grammar. A scanner and parser are used to turn this string into a syntactic model, represented as a syntax tree. Metrics such as cyclomatic complexity or NPATH can be calculated from a syntax tree, but others such as coupling and cohesion cannot.

The syntax tree, while useful, does not fully model the high level concepts in OO source code such as packages, classes and relationships. A semantic model of the code can be built from the syntax trees, and this is in essence what a compiler does. Depending on the richness and accuracy of the semantic model, advanced analysis such as relationship metrics and automatic design evaluations can then occur.

We use the Java semantic model provided by JST (Irwin, 2007). The model accurately describes the relationships between packages, classes and methods in a Java codebase.

2.3 Version Control Systems

The use of a Version Control System (VCS) such as SUBVERSION[1] or MERCURIAL[2] is common practice in team-based software engineering. These systems enable multiple developers to concurrently read and modify the same code base. The history of source code is maintained in a repository so that developers can revert to an earlier version at any time. Repository storage is usually highly optimized using text compression techniques.

2.4 Source Repository Mining

As the VCS repository provides access to every past version of the source code, it enables retrospective analysis of the evolution of software. The process of extracting and processing information from a repository is known as Source Repository Mining (SRM). It has become a significant area of research in recent years (Robbes, 2007, Wedel et al., 2008). Two of the most popular applications of repository mining are defect prediction (Aversano et al., 2007, Nagappan et al., 2006), and the characterisation of software evolution (Robles et al., 2006). Our tool EVOJAVA builds on technology used in both of these areas, and should subsequently allow us to contribute back to this area, with detailed and accurate data. When mining a repository, it is also possible to extract the metadata about the change between each version change, such as the time, author, or change description (commit message). This can give information such as the size of contributions per developer, or which particular changes fixed a bug.

2.4.1 Defect Prediction

Defect prediction involves using software metrics to find areas of code that are likely to contain defects or bugs. Predictors are built in retrospect, by analysing both the source code and historical defect information. In essence, a predictor takes a collection of metric values from a module and then estimates the number of defects it contains using its internal model and past experience. In order to build an effective predictor, the metrics must be shown to have a strong empirical relationship to defect rate.

Diverse metrics are used in the defect prediction research, but they have common themes of measuring size, quality, and complexity. Metrics from the CK suite have been found to correlate with defect rates, as documented in a summary of the area by (English et al., 2009). Modules with more Lines of Code (LOC) have been strongly correlated to higher defect rates; this is hardly a surprising result.

A large scale study of Microsoft projects such as INTERNET EXPLORER and DIRECTX found that other metrics such as number of functions, fan in, cyclomatic complexity and inheritance depth were correlated to defect rates (Nagappan et al., 2006).

---

[1] http://subversion.apache.org/

[2] http://mercurial.selenic.com/

The EVOJAVA tool will enable us to collect these metrics, to detect correlations not only against defect rate, but refactoring and code rework.

2.4.2 Software Evolution

The other large area of SRM research is the characterisation of software evolution, by means of repository mining (Robles et al., 2006). For example, a tool called SOURCERER was build and then run over 38 million lines of java code to collect many evolution metrics (Bajracharya et al., 2009). Similarly, OHLOH[3] is a website which mines repositories of thousands of open source projects to provide information about longevity of projects, and popularity statistics for programming languages. Although the existing work has a large number of projects, the metrics and source code analysis is too superficial for our analysis. These results are perhaps suitable at project manager level, but are not fine-grained enough to perform metrics research at the class, method or line level.

Some researchers have focussed on designing and implementing extensible research frameworks for software evolution, in a similar vein to EVOJAVA. The benefits of such frameworks are that researchers can focus without specific questions without having to invest significant development time on the tools. The ALITHEIA CORE is such an extensible framework designed for software engineering research (Gousios and Spinellis, 2009). In addition to source code repositories, information is extracted from bug tracking systems and email servers. This system is very heavy weight, designed to have a distributed architecture. The KENYON framework (Bevan et al., 2005) was designed for similar reasons, with a focus on fact extraction, fact storage and scalability. Both systems stored rich amounts of metadata and integrated with multiple VCS systems. However, they currently only support evolution measurement at a higher level and plug-ins would need to be developed to accurately collect our target metrics. While relevant, the design goals for EVOJAVA are sufficiently different that we have designed a new system.

2.5 Limitations of a VCS

EVOJAVA was designed to be a general purpose, accurate tool for software evolution metric research. Mining from a VCS repository, as in much previous SRM research, is not the only option for such analysis. In fact, mining from a VCS repository has several downfalls (Robbes, 2007). In this section we discuss these reasons, and argue for our choice to mine a VCS repository.

The primary shortcomings of VCS repositories are that they are file-based and snapshot based. The term 'file-based' means that version control systems store information as files and folders the basic building blocks in a file system. While this allows them to store a wide range of information, it means that a significant amount of pre-processing is required to abstract the content into higher level semantic concepts such as classes, methods and relationships. The term snapshot-based refers to the granularity at which information is updated to the repository. The size of a *changeset* between any two revisions can be arbitrary, and the actual number of code level change actions such as refactorings is unknown. It is not possible given two consecutive revisions (or snapshots) to completely reconstruct the set of actual actions that took place, thus data is lost.

Robbes' proposed solution was a new technology called a change based repository. This required an IDE plug-in to be used, which would store semantic code level actions in a domain specific repository, at a significantly finer granularity. Although this approach reduced information loss, we still argue for the use of classic VCS repository mining in EVOJAVA for several reasons.

Previous work on static analysis has produced tools for parsing and modelling software systems. We believe that JST is accurate and powerful enough that the file-based aspect of repositories is not a problem.

Also, the implemented change based repository was only a prototype for one language (Smalltalk) and only worked for a specific IDE. This made the assumption that all code modification actions would occur using IDE actions, which is not a universal reality. Real world software may be developed using different tools or environments.

And lastly, VCS repository mining can be done after the fact. Repository mining techniques allow us to analyse the many existing open source projects over multiple years of development.

3 EvoJava

3.1 Requirements

EVOJAVA was designed to meet the following requirements:

1. Integrate with a SUBVERSION repository. The history of code versions is the primary data source for analysis.
2. Construct a temporal semantic model of Java programs. Existing models such as JST describe the static semantic concepts in software; EVOJAVA adds a time dimension.
3. Compare consecutive models and deduce the actual changes that occurred, without losing the identity of software features. This is a necessary prerequisite to building an accurate evolution model from a snapshot based repository.
4. Provide an API that enables configurable calculation of metrics, including traditional software metrics, CODE CRITICK metrics and new evolution metrics.
5. Condense results into comprehensible forms using visualisations.

System performance, in terms of memory usage and processing time, was not a priority, due to the research nature of this project.

3.2 System Architecture

The system was designed in a modular fashion so that it would be extensible and could be used in future work. The main modules of the system directly relate to each requirement identified above: the repository integration module, the evolution model, JSTDIFF, and the metric

[3] http://www.ohloh.net/

framework. In this section, the system workflow, architecture and interaction between modules are discussed. Important modules are then explained in detail in following sections.

The system requires two inputs items to run, the address of a SUBVERSION repository and an XML query file which specifies the metrics to collect. Once a run has finished, an XML file is produced containing the results. This workflow is that of the XML pipeline (Irwin and Churcher, 2001) – the output could be transformed using XSLT to XML input formats for our existing visualisations. Thus, although the system is internally complex, it can be viewed as a 'black box' for collecting evolution data.

The internal architecture of the system is depicted in Figure 1. The grey dotted rectangle represents the EVOJAVA system, and the items outside of this rectangle represent the input or output artefacts discussed above.

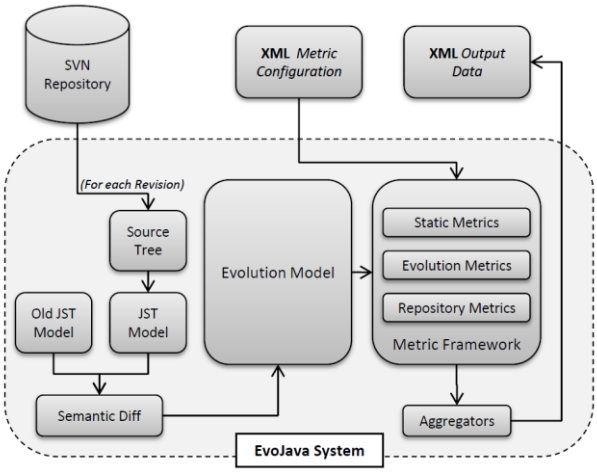

Figure 1: System Architecture

The repository integration module is responsible for several tasks. When a run commences, it will query the repository to fetch a list of version numbers, and branches. It will then incrementally check-out a copy of the code base at each version, allowing the rest of the system to process one version at a time.

The evolution model is a semantic model of a Java code base, modelling the concepts of classes, methods and packages. It is similar to the JST, except that has the dimension of time. The model is updated at each version to reflect the current state of the code base, whilst still remembering critical information from previous versions (such as a historical list of changes).

The metric framework is responsible for querying the evolution model to collect the metrics of interest. When the metrics are collected depends on the category of metric. *PerRevision* metrics, such as the traditional metrics LOC and WMC are run as the model updates to each version. *Evolution* metrics are those run only at the end of processing, such as the *Lifespan* or *ModificationRate* of a java method.

JSTDIFF is the last module of the system. JSTDIFF uses a set of algorithms and heuristics to determine the semantic changes that have occurred between two consecutive code versions. At each version, it used to calculate the *semantic diff*, which is used to update the evolution model, whilst still preserving identity to modified classes, methods and packages.

The JST is still the backbone to the system. It is richer that the evolution model so it is used by many of the *PerRevision* metrics. It is also used to parse each version of the code base, and build the semantic model required by JSTDIFF.

3.3 Subversion Integration

The EVOJAVA system interacts with a SUBVERSION repository in order to extract data. We chose to support SUBVERSION for the following several reasons. Firstly, SUBVERSION is centralised, so history of versions is stored on a single server. It would be harder to mine a distributed system as there can be no identifiable 'canon' repository. Secondly, SUBVERSION is one of the most popular systems, used by open source giants such as the Apache Software Foundation[4], and GCC[5]. Finally, the University of Canterbury Software Engineering department uses SUBVERSION. This allows us to mine our own projects, and apply context to the collected data. Although we currently only support SUBVERSION, generalising EVOJAVA at a later time should not prove difficult.

An open source, pure Java SUBVERSION library called SVNKIT was used in EVOJAVA as it provides an API for all of the tasks required.

Unfortunately, the code history is complicated by branching and merging in the repository. SUBVERSION allows you to create a branch, which is a clone of the entire code base, with its own parallel history. Typically this is used in order to create an unstable feature branch, or a stable release branch, which can be modified without affecting the main branch, usually known as *trunk*. The opposite, applying the changes from the history on one branch to another branch, is known as merging. Due to branching and merging, there may be many parallel histories for any code object, and not a single linear history.

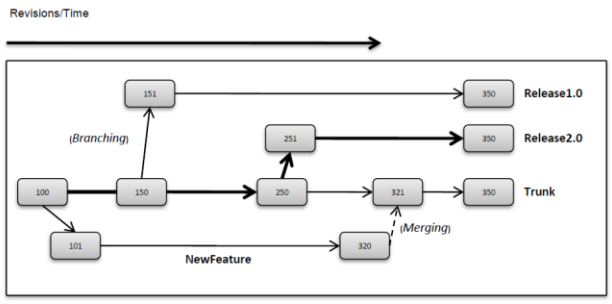

Figure 2: SVN Branching

EVOJAVA addresses this problem by automatically finding the logical path through the version tree from a user-specified endpoint of interest. It will backtrack through the tree, finding the branching revisions, and locate the logical origin of the code. Figure 2 depicts a nonlinear history, and the path of revisions used for analysis, given the endpoint *Release2.0*. Until recently,

[4] http://www.apache.org/

[5] http://gcc.gnu.org/

SUBVERSION would not keep record of merges in the repository. Consequently, when a set of changes from a branch is merged back into the trunk it will appear to EVOJAVA as a single large set of changes. This is largely unavoidable.

3.4 Evolution Model

The evolution model is effectively a lightweight version of the JST which models containment relationships between Java feature nodes such as packages, classes, interfaces and methods. However, each node in the model stores evolution information such as creation and deletion revisions as well as a history of changes. It does not aim to fully replace the rich JST model, but rather augment it with the time dimension.

Both JST and the evolution model are queried in the same fashion. The metrics are able to visit the models using the Visitor design pattern [GoF]. This allows the metrics to touch the nodes of interest, and then call their methods to collect data.

Each node in evolution model supports the concept of identity, even if its name or content changes drastically throughout its lifespan. Nodes are recognized using a unique id, built from the fully qualified Java name at the first version it appeared in.

3.5 JSTDiff

JSTDIFF is perhaps the most complex module in EVOJAVA. Its purpose is to determine the semantic changes that have occurred between two consecutive code versions. It is based on the existing UMLDIFF system, which required a custom semantic model, rather than the JST.

It is important to distinguish a semantic diff from a text diff. A text diff is produced by the VCS to display the text-level changes that have occurred between two versions, in terms of line and file additions and removals. The text diff fails to capture higher level Java semantic concepts. What is identified as a line addition in a text diff, could semantically actually be the addition of a field to a class, or statement to a method body.

JSTDIFF works by walking two JST models simultaneously, comparing the 'before' and 'after' nodes. The difficult part is determining which nodes in each are actually the same, despite having been renamed or modified between versions. When matching nodes, JSTDIFF matches nodes with the following priorities:

1. Completely Identical
2. Different, but have the same name[6]. This corresponds to a *Modified* change.
3. Structurally Similar, different name. This corresponds to a *Renamed* change.
4. Unresolved. The remaining nodes are marked as *additions* and *deletions*.

Several similarity heuristics are used to determine matches. Priorities 1 and 2 use a simple text similarity heuristic to determine name similarity.

[6] For example, it is a very unlikely a class is deleted and another is added immediately with the same name.

Priority 3 uses two heuristics. The first one is the percentage of identical or nearly identical lines in text body. The other one compares the similarity of the relationship sets of the two nodes. The set includes object names, and their relationship type to the node, such as containment, invocation, declaration, dependency etc. The maximum value from these two heuristics is used, for stability against different types of changes.

Once two nodes are matched, the actual changes are determined by comparing the attributes on the before and after node. The changes are combined into a tree model, which is then applied to the evolution model.

3.6 Metrics

Different categories of metrics can be collected using the metric framework in EVOJAVA.

The first category is the *PerRevision* metrics. These metrics run only on a snapshot, and don't account for evolution themselves. However, as they are run on each version, they can be used to describe evolution. The CK and MOOD OO metrics, as well as the CODE CRITICK system, are all *PerRevision* metrics.

The second category is the *Evolution* metrics, which utilise the EVOJAVA evolution model. *Evolution* metrics tell you something about the life span of an object, such as how frequently it was modified.

The last category is *Repository* metrics. These are enabled in EVOJAVA through the SVNKIT API, and could include metrics such as commit frequency and developer contribution size.

4 Results

We have used the EVOJAVA tool to gather data from the repositories of real world software projects. In this section we discuss the preliminary findings, to demonstrate the power and utility of the EVOJAVA tool.

At the University of Canterbury, 3rd year students can elect to take a year-long course, in which they must develop a large scale, real-world software project. The class usually consists of about 6 teams, each containing 6 students. We chose to use the 2010 student projects as a pilot study for our tool, for several reasons:

- The Scale of the projects is suitable for a pilot study. Most have between 200-500 revisions, and about 5,000 LOC.
- The teams are building individual projects, but to solve the same problem. This allows comparisons between repositories.
- Feedback on the development habits of students is useful for the department.
- The developers of the project are easily accessible, which is of mutual benefit to us and the students. Students are interested in the data collected by our tool, and in return we are able to query them about any irregularities in the data collected, to evaluate the tool and refine it.

As mentioned previously, EVOJAVA fits into the XML pipeline. XSLT transformations have been written to transform the XML output into CSV files, which were then graphed using Excel for this report.

4.1 Evolution Overview

Traditional 'static' metrics such as LOC, and the CK suite, can be run automatically on each previous version of the code. This is one of the features of EVOJAVA.

Figure 3 displays the total LOC of a group's project, over the first 550 revisions. Unsurprisingly, the graph is gradually trending upward, eventually reaching about 15,000. Gathering this data is a trivial task, but it is critical for deeper analysis, thus EVOJAVA is able to collect it. Graphs of this sort are common in both the literature and existing tools such as OHLOH.

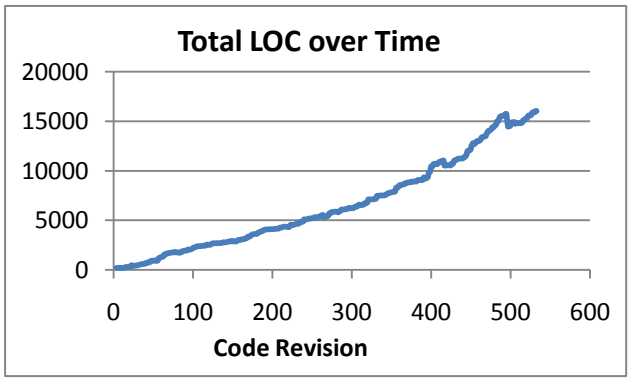

Figure 3: LOC over Time

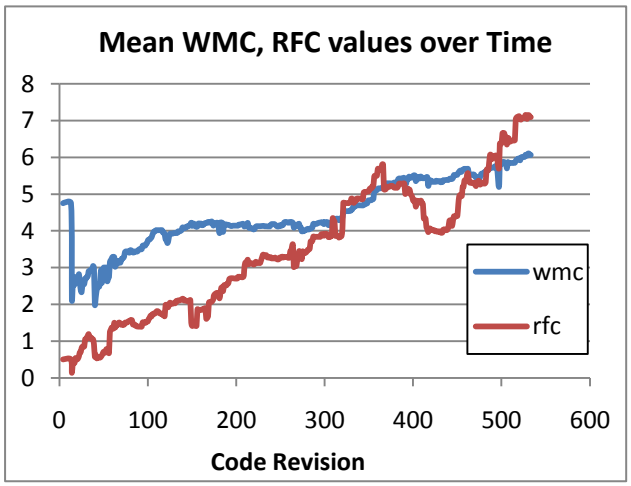

Figure 4: WMC, RFC over Time

Figure 4 shows some more detailed information. Weighted Methods per Class (WMC) and Response for Class (RFC) are two metrics in the CK OO Suite. These metrics must be measured on a per-class basis, and require a strong semantic model, such as the JST. This graph shows that on average, the number of methods per class, and the number of methods called per method, are both increasing over time.

4.2 Code Critick

The CODE CRITICK System developed in our previous work is now part of the EVOJAVA system, so we are able to characterise software evolution in regards to OO design.

CRITICK returns a ranked list of Violations to OO design rules and heuristics, which are implemented using various metrics and algorithms. Figure 6 displays the quantity of violations found against several CRITICK rules, over a project's lifetime. Note that this graph is normalised for project size, and the Y axis represents violations per 1,000 lines of code.

The presence of violations is common, and largely unavoidable, due to the nature of conflicting forces in OO design. Encapsulation related rules can also be very conflicting, and tend to make up the majority of violations found in student systems. For this graph, several rules were removed.

The results for the first few revisions are bound to be noisy due to the small, volatile nature of a project at this point.

The overall violation trends for other groups' projects were quite different to the one depicted above. In particular, another group frequently broke the *LargeClassSmell* and *LongParameterListSmell*, rules, but had no *SwitchStatementSmells* at all.

4.3 Change Metrics

The results discussed so far only measure software metrics against individual snapshots.

The real power of the EVOJAVA system is that it contains an evolution model, and preserves the identity of semantic elements (such as classes, methods, and packages) between versions, for a richer evolution analysis.

In addition to tracking semantic elements, between versions, the JSTDIFF subsystem of EVOJAVA serves to characterise the actual semantic changes that occur during an elements lifetime. Figure 5 shows the number of each detected type of semantic change, over the lifespan of a group's project. The revisions have been grouped into 5 100 revision buckets, and the quantity of each change type is shown as a bar.

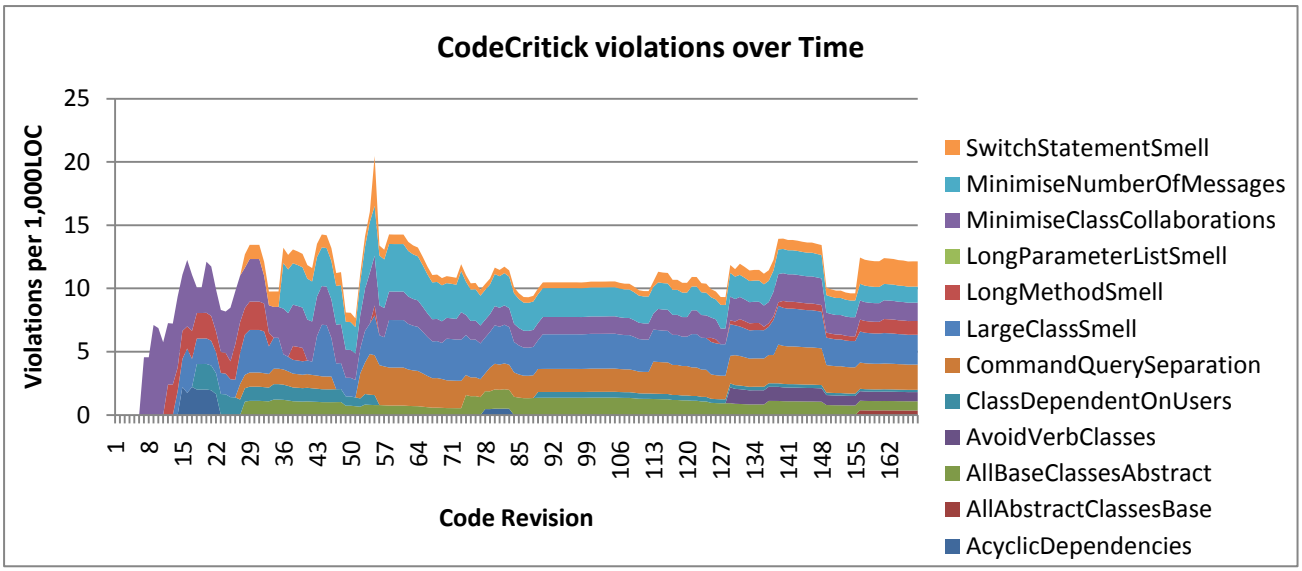

Figure 6: CodeCritick over Time

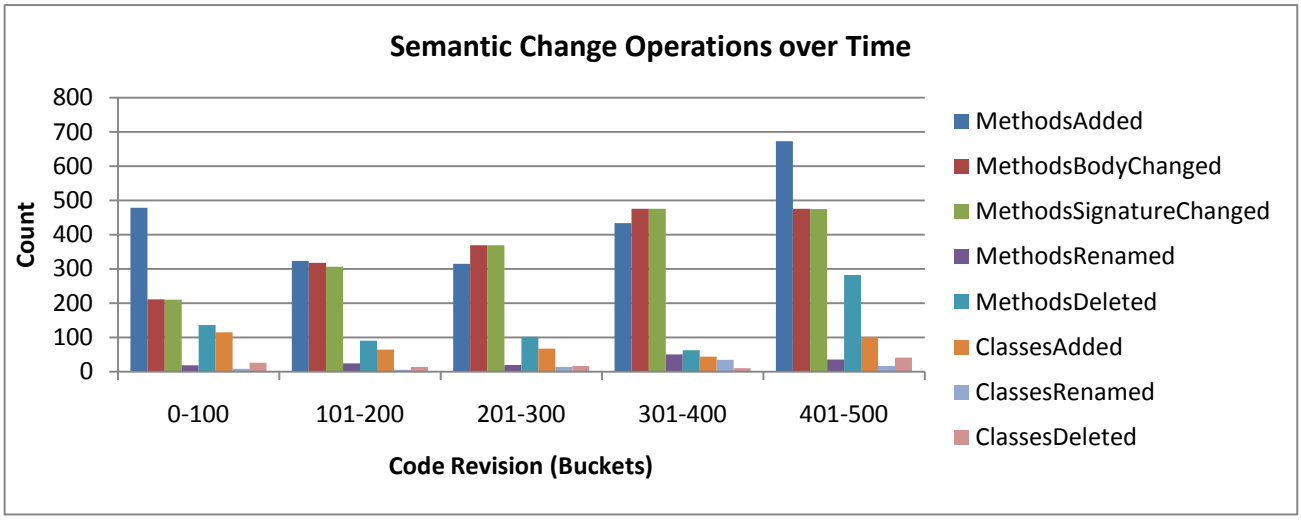

Figure 5: Change Types over Time

In the first and last buckets, the primary type change operation identified is *MethodAdded*. The change type *MethodsBodyChanged* is detected when one or more lines of code in the method have been added, removed, or modified, as detected by a SVN textual diff. *MethodsSignatureChanged* is detected when the name, visibility, return type or parameters of a method are changed. These change types are two of the most common in each bucket, as can be seen by the red and green bars in the graph.

In all buckets, the number of methods added outweighs the methods deleted, and the same is true for classes added and classes deleted. This is sensibly linked to Figure 3, where we see a steady increase in LOC for the same project.

In the last bucket, there were significantly more methods added *and* deleted. This suggests that significant code rework occurred at this point in development. This likely explains the apparent dip in RFC displayed between on Figure 4, during the same revision period.

Another interesting point is that throughout the entire lifespan of the project, relatively few rename operations were detected.

4.4 The Interactive heat map Visualisation

Although overview aggregate measures are useful for characterising software evolution, EVOJAVA is able to measure metrics, and track elements at a much finer granularity. Displaying this amount of information presents information overload issues, which need to be addressed. The interactive heat map, as depicted in Figure 7, is able to display such detailed information in a compact space. It is loosely based on CVSCAN (VOINEA ET AL., 2005).

Each row in the display represents an element in the system, depending on the selected granularity (in this case, each row is a method). Each column in the display represents a revision of the code, as extracted from Subversion. The shaded (non-white) cells in a row represent the versions that the particular element was present in. For example, in the first revision 3 methods

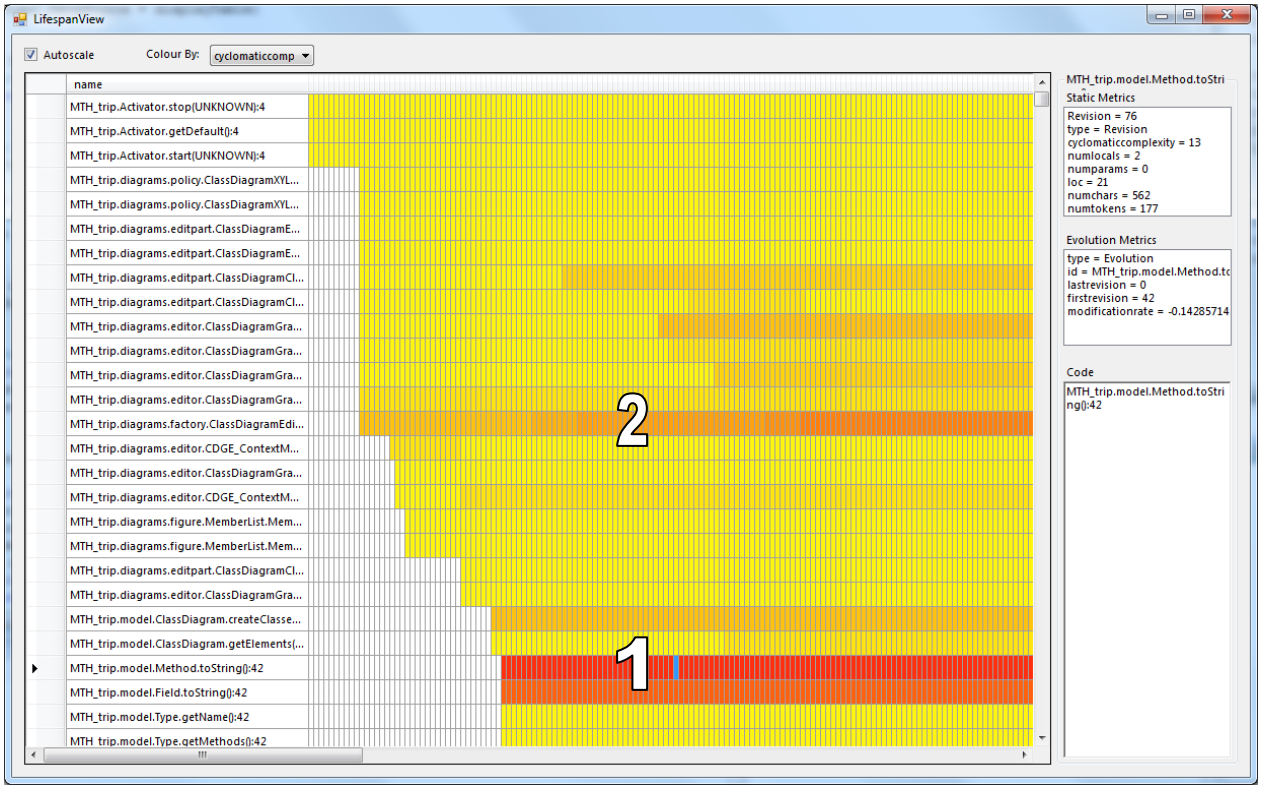

Figure 7: The Interactive heat map

were present. Several revisions later, 11 further methods were added.

The actual colour of each shaded cell represents the heat, or relative metric value, of that element at that particular version. For example, this heat map is shading cells based on the *CyclomaticComplexity* metric. The row marked 1 represents a method that has is theoretically quite complex. The method shown as the row marked 2 was mildly complex when it was added, but was modified several times subsequently to become more complex.

The heat map is interactive, as it allows users to select any cell in the display to view more information. Selecting a cell will display all of the available metric values for the selected element (row) at the selected version (column) in a text box at the top right.

If available, the source code is retrieved and displayed in a textbox to the bottom right.

4.5 Performance Considerations

As mentioned previously, EVOJAVA is primarily a research tool, correctness was valued ahead of system performance, and thus it was only identified as a secondary goal.

For a project with 200 revisions, and 5,000LOC, analysis took under 20 minutes on a modern desktop computer (2.8Ghz Quad-core, with 4GB DDR3 RAM). The same computer processed a 600 revision, 15,000LOC project in just under 3 hours.

The memory footprint of the system is low, as it processes versions incrementally, rather than in parallel. This allows analysis to occur comfortably on a regular desktop computer.

4.6 Future Work

The EVOJAVA system will be extended in our future work to accommodate these features:

1. Extension of JSTDIFF to detect of composite semantic change operations, such as refactorings.
2. Performance enhancements to the underlying JST model, for faster analysis
3. Potential integration with our Process Metrics plug-ins, to collect code change information at an even finer level.

In addition to these tool features, the system will be used for more in-depth software evolution analysis.

5 Conclusion

We have presented EVOJAVA, a new tool for extracting static software metrics from a Java source code repository. For each version of a program, EVOJAVA builds a comprehensive model of the semantic features described by Java code (classes, methods, invocations, etc), and tracks the identity of these features as they evolve through versions, using the novel JSTDIFF system.

We presented and discussed results collected from real world software projects, developed by student teams at the University of Canterbury. Traditional metrics, OO design metrics, and change metrics were all collected with the tool and discussed.

Finally, we presented a software evolution visualisation called the interactive heat map, and mention our future directions.

6 References

Abreu, F. B. E. & Melo, W. (1996): Evaluating the Impact of Object-Oriented Design on Software Quality. *METRICS '96: Proceedings of the 3rd International Symposium on Software Metrics.* Washington, DC, USA: IEEE Computer Society.

Aversano, L., Cerulo, L. & Del Grosso, C. (2007): Learning from bug-introducing changes to prevent fault prone code. *IWPSE '07: Ninth*

international workshop on Principles of software evolution. Dubrovnik, Croatia: ACM.

Bajracharya, S., Ossher, J. & Cristina Lopes (2009): Sourcerer: An internet-scale software repository. *SUITE '09: Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation.* Washington, DC, USA: IEEE Computer Society.

Bevan, J., Whitehead, E. J. J., Kim, S. & Godfrey, M. (Year): Facilitating software evolution research with kenyon. *In:* ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, 2005 Lisbon, Portugal. ACM, 177-186.

Chahal, K. K. & Singh, H. (2009): Metrics to study symptoms of bad software designs. *SIGSOFT Softw. Eng. Notes,* 34**,** 1-4.

Chidamber, S. R. & Kemerer, C. F. (1994): A Metrics Suite for Object Oriented Design. *IEEE Trans. Softw. Eng.,* 20**,** 476-493.

English, M., Exton, C., Rigon, I. & Cleary, B. (2009): Fault detection and prediction in an open-source software project. *PROMISE '09: Proceedings of the 5th International Conference on Predictor Models in Software Engineering.* Vancouver, British Columbia, Canada: ACM.

Fowler, M. (1999): *Refactoring: Improving the Design of Existing Code,* Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc.

Gousios, G. & Spinellis, D. (2009): Alitheia Core: An extensible software quality monitoring platform. *ICSE '09: Proceedings of the 31st International Conference on Software Engineering.* Washington, DC, USA: IEEE Computer Society.

Irwin, W. (2007): *Understanding and Improving Object-Orientated Software Through Static Software Analysis.*

Irwin, W. & Churcher, N. (Year): XML in the visualisation pipeline. *In,* 2001. 67.

Lincke, R. U., Lundberg, J. & L\"Owe, W. (2008): Comparing software metrics tools. *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis.* Seattle, WA, USA: ACM.

Mccabe, T. J. (1976): A complexity measure. *ICSE '76: Proceedings of the 2nd international conference on Software engineering.* San Francisco, California, United States: IEEE Computer Society Press.

Medha, U. & Carolyn, S. (2008): Why do programmers avoid metrics? *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement.* Kaiserslautern, Germany: ACM.

Nagappan, N., Ball, T. & Zeller, A. (2006): Mining metrics to predict component failures. *ICSE '06: Proceedings of the 28th international conference on Software engineering.* Shanghai, China: ACM.

Nagappan, N., Williams, L., Vouk, M. & Osborne, J. (2005): Early estimation of software quality using in-process testing metrics: a controlled case study. *3-WoSQ: Proceedings of the third workshop on Software quality.* St. Louis, Missouri: ACM.

Nejmeh, B. A. (1988): NPATH: a measure of execution path complexity and its applications. *Commun. ACM,* 31**,** 188-200.

Oosterman J., I. W. & Churcher, N. (2010): Code Critick: Using Metrics to Inform Design. *ASWEC '10.* University of Canterbury, Christchurch, New Zealand.

Riel, A. J. (1996): *Object-Oriented Design Heuristics,* Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc.

Robbes, R. (2007): Mining a Change-Based Software Repository. *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories.* Washington, DC, USA: IEEE Computer Society.

Robles, G., Gonzalez-Barahona, J. M., Michlmayr, M. & Amor, J. J. (2006): Mining large software compilations over time: another perspective of software evolution. *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories.* Shanghai, China: ACM.

Voinea, L., Telea, A. & Van Wijk, J. J. (2005): CVSscan: visualization of code evolution. *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization.* St. Louis, Missouri: ACM.

Wedel, M., Jensen, U. & G\"Ohner, P. (2008): Mining software code repositories and bug databases using survival analysis models. *ESEM '08: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement.* Kaiserslautern, Germany: ACM.