Sextant: A Tool to Specify and Visualize Software Metrics for Java Source-Code

Victor Winter, Carl Reinke, and Jonathan Guerrero
Department of Computer Science, University of Nebraska at Omaha
Omaha, Nebraska, USA 68182

Abstract—The success of software development efforts typically require guidance derived from a deep understanding of its design space. Well-crafted software metrics can impart invaluable insight into the nature of software and can provide the underpinnings for informed decisions involving design and implementation trade-offs. Leveraging metrics to their full capacity requires (1) the ability to quickly develop custom metrics that are domain-specific and even application specific, and (2) the ability to present metrics-related information in a comprehensible fashion.

This paper describes a tool, called Sextant, whose purpose is to achieve the goals stated in the previous paragraph. An overview of Sextant is given followed by two in-depth real-world examples of how the use of metrics can positively impact software development efforts.

Index Terms—software metrics, source code analysis, software visualization, program transformation, Java, Sextant

I. INTRODUCTION

Conceptually speaking, a central concept in the study of software metrics, in the limit, is the art and science of leveraging that which can be measured in order to make predictive statements about that which cannot be measured. Stated more succinctly: Indirect measurement plays an important role in software metrics.

Example: Studies have shown that complexity measures, such as McCabe’s cyclomatic complexity [1], have statistical correlations to software quality (e.g., if a Java method has a cyclomatic complexity exceeding a certain threshold, then that method will have a high likelihood of containing a defect). When viewed in nonstatistical terms, such an inference is neither sound nor complete. Denaro et al. [2] takes the idea of indirect measurement one step further by exploring the idea of constructing fault-proneness models for certain classes of software. The common thread here is that these approaches strive to measure program correctness indirectly. Specifically, these approaches are assurance mechanisms to be used in lieu of formal verification.

For theoretical reasons (Gödel’s incompleteness theorem) as well as practical reasons (combinatorial complexity and current tool limitations), a number of important questions about software artifacts cannot be directly answered. Nevertheless, the success of software development efforts typically require guidance derived from a deep understanding of the threats within their design space as well as the mitigation of those threats (e.g., do not pursue a development trajectory resulting in a system whose complexity is unmanageable). More often than not, dangers in the design space emanate from software attributes which cannot be directly measured. This is the challenge faced not only by those who study software metrics. It is a challenge for all who design, build and maintain large software systems.

A. Practical Limitations

In practice, the domain of discourse in software metrics is limited (among other things) by two important factors: (1) the measurement capabilities of tools, and (2) the ability to represent the results of a measurement in a comprehensible fashion. Metrics which cannot be implemented or whose results cannot be understood have little practical value. The impact of these two factors on the discipline should not be underestimated for, in practical terms, they ultimately establish bounds on the vocabulary and language used to think about software metrics. It was Dijkstra who said: “A Programming Language is a tool that has profound influence on our thinking habits”. This observation also applies to the domain of software metrics.

Historically, software metrics focused on attributes of software that could be readily measured (i.e., for which tools could be quickly developed). Classic examples of static measurements include source-lines of code (SLOC), number of methods per class (NOM), and method size. The data produced from such analysis are numbers and sets of numbers. Data from such measurements can be aggregated through statistical techniques, such as mean, variance, and standard deviation. However, it should be noted that numbers, in-and-of-themselves, are not suitable as a universal formalism for conveying information in a comprehensible fashion. For example, positional information (e.g., the distribution of elements satisfying a given metric across a design), clustering information, and emergent properties of the system-as-a-whole are more effectively conveyed in graphical terms. Initially, this led to the use of charts and graphs to represent certain kinds of data. However, more recently, formalisms such as treemaps and complex networks (i.e., graphs) are increasingly used to visualize “non-numeric” information. Tools such as TreeMap [3] and Cytoscape [4] provide extensive general-purpose capabilities for enhancing aspects of data deemed to be of interest. An important dimension that needs to be further
developed is the connection between software metrics and such visual representations.

Over the years, properties captured by software metrics have become increasingly sophisticated. Complexity measures such as McCabe and Halstead provide significant insight into characteristics of software. Cohesion and coupling metrics, such as those developed by Chidamber and Kemerer [5], have also been developed to capture essential properties of a design.

This brings us to modern times where the scope of software metrics has been further expanded encompassing source code analysis rules such as those used by PMD, Checkstyle, and FindBugs. Challenges faced by these kinds of metrics include: (1) obtaining a comprehensive overview of software quality [6], (2) displaying analysis results in manner suitable to the needs of software metrics-based reasoning, and (3) the level-of-effort associated with implementing new rules of this kind.

B. Contribution

This paper describes a tool called Sextant capable of specifying and visualizing custom software metrics. Emphasis is placed on how custom metrics, such as those that can be defined in Sextant, can effectively impact software development efforts. Towards this end, two detailed examples are given showing how Sextant can be used in practice. In these examples, particular attention is given to the interplay between metrics specification and visualization. Specifically, the understanding obtained from the visualization of one metric leads to the specification and visualization of another. The dynamics in these examples provide an additional case-in-point supporting the importance of interactive use of metrics.

The remainder of the paper is as follows: Section II gives an overview of Sextant, its novel features and its design goals. Section III gives a detailed discussion of two real-world instances where interactive software metrics played an important role. Section IV looks at some related work, and Section V concludes.

II. OVERVIEW OF SEXTANT

Sextant is a Java source-code analysis tool under development at the University of Nebraska at Omaha (UNO). From an implementation standpoint, Sextant represents a non-trivial extension of the TL system [7][8] (a general-purpose program transformation system) specialized to the domain of the Java programming language.

One of the primary design goals of Sextant is to provide a tool facilitating specification and visualization of custom analysis rules (e.g., domain specific or even application specific analysis rules).

In this paper, the terms (software) metric and analysis rule are interchangeable. We classify as a metric any process that gathers information about a software system. For example, our use of the term analysis rule/metric encompasses both (1) source-code analysis rules of the kind that can be specified by tools such as PMD, Checkstyle, and FindBugs, as well as (2) more traditional software metrics such as cyclomatic complexity, fan-in, fan-out, number of methods per class, statement count per method, and so on.

A. Architectural Overview

Sextant analysis rules are based on information drawn from several software models. At present, there are two models of central importance. The first model, a syntactic model, is the parse tree of the source code. Parse trees correspond to compilation units (i.e., Java files) and are generated using GLR parser technology provided by the TL system. Parse trees are well-suited to analysis and manipulation through standard primitives provided by program transformation systems such as matching and generic traversal.

The second model, a compound attributed graph (CAG), is a semantic model capturing structural, subtype, and reference dependencies among the fields, methods, constructors, types, and packages. The CAG also associates an attribute list with each node and edge. Table I gives an overview of the attributes associated with nodes and edges within the CAG.

Information in the CAG is made accessible to Sextant’s transformation-based analysis rules through two mechanisms. The first mechanism is a positional system [9] that establishes a relation between parse (sub)trees and corresponding contexts within the CAG. It is this relation that makes it possible to correctly resolve references to types, fields, local variables, methods, and constructors during the course of generic traversals which are a key enabling mechanism within program transformation systems. For example, the resolution of the simple identifier x, encountered during a generic top-down traversal of a parse tree, is dependent upon the context in which it occurs.

The second mechanism is a library of semantic queries which, thanks to the positioning system, can be accessed during the course of transformation. Functionality presently provided by this library includes things like: (1) resolving a reference, (2) determining the type of a reference, (3) determining whether one type is a subtype of another, (4) determining whether one declaration shadows or overrides another, and so on.

B. Collecting Information

Sextant provides table and set datatypes for collecting information associated with analysis rules. These constructs are suitable for storing information related to a wide variety of custom metrics. In particular, sets can be used to model arbitrary predicates. Within Sextant, the implementation of such predicates is accomplished through user-defined transformation-based analysis rules. In this fashion, Sextant is open-ended with respect to the definition of metrics – any source-code analysis rule can be interpreted as a metric, be they PMD-style rules focusing on violations of coding
conventions or rules such as those specified by FindBugs that are more semantic in nature.

In contrast, tables provide an abstraction well-suited to gathering standard numeric-based software metrics such as: (1) number of fields, methods, classes, and packages in an application, (2) lines of code per method, (3) cyclomatic complexity, and (4) fan-in/fan-out. Sextant also provides a set of statistical operators (e.g., sum, mean, variance, standard deviation) which can be applied to tables containing numerical data.

Subject to certain restrictions, information collected can be output in a comma-separated-value (CSV) format or an in-house developed graphflow file format (described in Section II-C).

C. Visualization

To facilitate comprehension, Sextant can generate software models which can be visualized using third party tools such as Cytoscape, TreeMap, and GraphViz. For example, Sextant can output the CAG of a targeted code base in a JavaScript Object Notation (JSON) format. This dot-JSON file can then be loaded into Cytoscape, an open source platform providing extensive and sophisticated capabilities for visualizing large complex networks (i.e., graph structures). Similarly, metrics derived from tables and sets can be output in CSV format and viewed using TreeMap [3], and parse trees can be output as dot-files and subsequently viewed via GraphViz.

To date, our primary efforts in custom visualization have centered around visualization-based analysis of Sextant’s CAG, through Cytoscape. Through a set of plugins, providing Java-specific visualization and analysis functions, we have extended Cytoscape’s general-purpose visualization and analysis capabilities. Our plugins are able to extract from the CAG a variety of views, the two most important being (1) a structural view of the elements in the CAG (e.g., fields belong to classes and top-level classes belong to packages), and (2) an inheritance view showing the subtype relation of the types in the model. More abstract views are also possible showing class-to-class, class-to-package, and package-to-package dependencies. Our plugins also provide suitable mechanisms for creating subviews as well as showing direct (i.e., fan-out) and transitive dependency relationships of selected software elements.

Recently we have developed a plugin that provides the ability to color nodes in such graphical networks according to a given set of rules stored in a file. A rich language for specifying coloring is provided. More specifically, match expressions can be written involving attributes associated with nodes and edges. Conditions associated with individual attribute matching are expressed using java.lang.regex, including the use of capture groups.

III. APPLICATION

In this section, we present two coloring examples that are driven by real-world problems and make use of actual code bases. The aim of these examples is to provide empirical evidence showing that metrics, specifically custom metrics, can positively impact software development in practice.

A. Usage Analysis of Generic Types

At Sandia National Laboratories, a hardware implementation of the JVM, called the SCore processor, has been developed for use in high-consequence embedded systems [10]. Similar to the Java Smart Card, the SCore is a restricted version of the JVM and does not support the full range of bytecodes (e.g., bytecodes pertaining to floating point operations are not supported). The SCore also has limited support for native methods.

In order to facilitate embedded application development, a subset of the Java Standard Edition Base Libraries has been targeted for migration [11] to the SCore platform. Specifically, portions of java.lang, java.util, and java.io have been targeted for migration.

To perform the desired migration, we have developed a source-code analysis and manipulation migration tool, called Monarch. The goal of Monarch’s migration is to produce a version of the targeted portion of the base libraries compatible with the SCore platform. This migration involves (1) automated removal of methods, constructors, and fields having dependencies on unsupported features, and (2) manual re-implementation of “must-have” functionality in order to remove dependencies on unsupported features. Both these activities must be performed in a manner that preserves the correctness of the original code base (e.g., shadowing/overriding properties must be considered when removing a field/method declarations). A reduced functionality is acceptable, an altered functionality must avoided if possible and done with great care in unavoidable cases.

Conservative (i.e., functionally safe) removal-based approximations can lead to excessive removal due to issues such as shadowing, overriding, and overloading. Additional over-approximations arise as a result of the undecidability of points-to analysis. And finally, approaches to addressing these challenges are restricted by the fact that the embedded application using the migrated library is proprietary and may not itself be subjected to migration (or related analysis).

1) Using Software Metrics to Focus Development Efforts:

Adding to the challenges mentioned in the previous paragraphs, Monarch’s source-code analysis capabilities are not complete. In particular, the development team has yet to implement full support for generic types. This is no small task since Java generics can be extremely complex [12]. For this reason we believe the pursuit of a development goal to fully handle generics would take a prohibitive amount of time, resulting in failure of the Monarch project to meet an upcoming short-term deadline.

Given this situation, our plan is to try to implement only those generic analysis capabilities that are needed for the migration task at hand. The hope of Monarch’s development teams is that only modest generic analysis capabilities will be needed. To determine whether this is indeed the case we have developed software metrics to reveal how generics are actually used in the targeted code base. Our conjecture is that (1) there are not too many generic types, (2) generic type declarations are predominantly simple in nature (e.g., class Box<T>)
and do not make use of exotic constraints nor have excessive inferencing requirements, (3) instances of generic types (e.g., Box<String>) are not wide-spread, and (4) uses of type variables (e.g., T and V) are also not wide-spread.

Metrics showing the extent to which generics are used may reveal that conditions exist in which the analysis needed for the migration of generics could be performed manually. If so, this would provide us with a fall-back position in the case where sufficient automated analysis capabilities cannot be developed in the allotted time.

Our first step in metrics-based analysis is to develop a graph-based software metric in which generic classes are colored red and generic interfaces are colored blue. This coloring can be projected onto any Cytoscape view derived from Sextant’s CAG. Figure 1 shows a structural view of the targeted code base to which the coloring has been applied.

In the graph shown, textual information is associated with nodes and edges. For example, selecting a node will show, in a results panel, a variety of information including: (1) the canonical name of the node, (2) the node kind (i.e., field, method, constructor, class, enum, or interface), (3) visibility, (4) the file and line number of the node in the code base – from which navigation to source code is supported, (5) the list of elements the selected node is directly dependent on as well as the list of elements that directly depend on the selected node, (6) the list of elements that the node is transitively dependent upon as well as the list of elements that transitively depend on the node, and (7) the names of the coloring rules applied to the node.

Through visual inspection and node selection of the model shown in Figure 1, we can quickly determine that a number of interfaces in the package java.util are generic. Outside of java.util, there are only two generic interfaces (Comparable and Iterable) and one generic class (Enum).

Next we develop metrics-based views that color (1) all instances of generic types, (2) all uses of generic type variables, and (3) a composite view showing all colorings simultaneously. Due to space constraints we only present the generic-type uses and composite coloring which are respectively shown in Figures 2 and 3. It is worth noting that Sextant’s coloring algorithm permits nodes to be colored by multiple rules. In such instances rule colors are blended in a straightforward additive fashion.

At this stage in our analysis we are in a position to begin conducting some really interesting metrics-based analysis. For example, as previously mentioned, migration will remove elements (i.e., nodes) having direct or indirect dependencies on features that are not supported by the SCore platform. Unsupported features include: (1) reflection (i.e., dependencies on java.lang.Class), (2) floating point numbers, and (3) unsupported native methods. An important question now is:
Fig. 4. A coloring revealing dependencies on `java.lang.Class`, floats, and native methods.

Fig. 5. A coloring showing the intersection of generic instances and unsupported features.

“What elements that use generics will be removed during migration due to dependencies on unsupported features?” Specifically, if all generic instances will be removed during migration anyway (due to dependencies on unsupported features), then our development effort with respect to generic support will be dramatically reduced. If only a handful of generic instances survive, then a manual migration of these elements may be justified (depending on other metrics such as complexity measures) and our migration efforts can be re-targeted to more pressing matters.

The view shown in Figure 4 represents a coloring of dependencies on the unsupported features mentioned in the previous paragraph. Purple nodes have direct dependencies on unsupported features while nodes colored orange have indirect (i.e., transitive) dependencies on unsupported features. It is worth mentioning that our coloring plugin has capabilities for specifying the coloring distinctions for nodes having indirect (i.e., transitive) dependencies on colored nodes. New colors (e.g., orange) may be assigned to transitive-dependently nodes, colors may be combined, and colors may be faded using a decay rate (e.g., the more distant the dependency, the lighter the color).

The view in Figure 5 shows a combination of generic instance coloring (green) and unsupported feature coloring (black for both direct and indirect dependencies). The nodes colored bright green are those that contain generic instances and have no unsupported dependencies. There are three such nodes in the code base: 1 field, 1 standard method, and 1 method belonging to an anonymous class occurring within a method having an unsupported dependency which will be removed during migration. Thus, there are only two instances of generic types that need to be handled by our migrator. Navigating to the source code shows us that the two generic instances are: `Comparator<String>` and `java.util.Map<String,String>`. Further navigation shows the generic declarations to be `Comparator<T>` and `Map<K,V>`. In particular, exotic constraints are not used (e.g., bounding and wildcards).

Because of this analysis, in our next step we target the development of capabilities to perform resolution of type variables as well as instances of such types, which Figure 3 shows occur extensively in the `java.util` interfaces. A similar analysis has been conducted to determine the complexity of this task. For example, abstract methods that occur in interfaces can only use type variables to denote the method’s return type and the types of its formal parameters. Thus, analysis capabilities in such cases need not be extended to include instances of such types (e.g., `T myT`) nor references based on such instances (e.g., `myT.foo()`).

B. Validation of Generated Software

Class initialization is part of the linking phase of the JVM [13]. In Java, the initialization of a class takes place at most once during execution and involves executing the `<clinit>` method associated with a class. This method is generated by the compiler and contains code realizing all class variable initializers as well as static initializers. From an operational standpoint, class variable initializers and static initializers are executed in the order in which they syntactically appear in the class. The method `<clinit>` cannot be invoked explicitly through a source code level call, but rather may only be invoked internally by the JVM in response to the first active use of a class.

Due to complexity and hardware overhead, class initialization is oftentimes not supported in hardware implementations of the JVM. An exception to this is the SCore platform (described in Section III-A) which has a clever implementation of class initialization. Naturally, this capability of the SCore should be comprehensively tested.

In a project unrelated to the topic of this paper we have developed a set of transformations capable of generating a set of Java tests exercising the `<clinit>` behavior of a
TABLE II 
ABSTRACT MODEL USED TO GENERATE TESTS.

| A → OBJ | B → OBJ | C → D A | D → A B C | E → C A |

targeted JVM [14]. Our approach to generating Java tests is transformation-based. Specifically, the input to our test generator is an abstract model of a Java class hierarchy specifying the dependencies among static fields. An example of such an abstract model is shown in Table II. The abstract model shown specifies static dependencies between the user-defined classes A, B, C, D, and E. For example, the expression “D → A B C” encodes that D is a subtype of A and has static dependencies on B and C (in that order).

Given an abstract model an executable Java test-suite is generated in which all “interesting” class initialization possibilities are exercised. More specifically, a set of class hierarchies are created conforming to the abstract model. In this set, each class hierarchy corresponds to a single class initialization test. Within a class hierarchy each class contains a static field \( x \) whose initialization dependencies on the static fields of other classes in the test conform to the abstract model. A single test consists of an invocation-expression containing a “minimal” sequence of static field references (e.g., the initialization of \( A.x \) has references to \( B.x \) and \( C.x \)) having the property that its evaluation triggers invocation of all \(<\text{clinit}>\) methods in the class hierarchy.

Direct monitoring of the invocation of \(<\text{clinit}>\) methods can be accomplished through aspect-orientation and reflection. Monitoring \(<\text{clinit}>\) invocations can also be accomplished indirectly via a monitor class. To initiate a computation to be monitored, our transformation-based test generator produces a test driver that when called will evaluate the invocation-expression for each test. Within a monitor class, observed \(<\text{clinit}>\) evaluation sequences are compared to pre-computed sequences based on the abstract model. An individual test is said to “pass” when its pre-computed and monitored \(<\text{clinit}>\) invocation sequences match.

As the data in the tables suggest, there is a considerable complexity gap between abstract and concrete test models. The primary reason for this is due to the exploration of the combinatorial possibilities inherent in class initialization (i.e., first-use orderings among a collection of classes). In the worst case, combinatorial exploration has \( O(n!) \) complexity where \( n \) is the number of classes in the abstract model. Standard source code inspection can be used to validate that a given set of tests achieve their desired purpose. We would like to strengthen this validation argument through metrics-based visualizations. For example, we would like to validate through visualization the symmetrical nature between individual tests. Failure to display certain kinds of symmetry is conclusive evidence that the test generator is in error. Specifically, the following properties are necessary (though not sufficient) to assure the correctness of the test generator:

- **Structure** - all class hierarchies within a test set should have the same structure.
- **Completeness** - each instance of a class belonging to the abstract model should declare a static integer field \( x \). For example, it is to join points matching the pointcut “\( \text{set}(\text{int } * .x) \)” that after advice can be woven monitoring the invocation of \(<\text{clinit}>\) methods.
- **Test harness** - each test should have its own harness.
- **Monitor** - each test should have its own monitor.
- **Test driver** - a test suite should have exactly one driver.

Figure 6 shows the structural view of the Java code base generated from the model in Table II. In this view, test harness classes are colored red, monitor classes are colored green, and occurrences of the static integer field \( x \) are colored blue. In the lower right corner, the small structure colored black is the test driver. This view helps strengthen the correctness argument for the test generator by providing visual confirmation that the test suite possesses the properties previously mentioned.

IV. RELATED WORK

Recoder [15] is a Java-based framework for analyzing and transforming Java source code. Recoder provides capabilities for parsing and pretty-printing Java source code. Analysis capabilities include name and type resolution as well as a bidirectional refers-to relation. Transformations operate on syntax trees, are expressed in Java and are semantically informed. The transformational framework provided by Recoder is similar to the framework provided by Sextant. A notable difference is
that in Sextant, transformations are expressed in a special-purpose transformation language TL (instead of Java), and semantic analysis is performed in the functional programming language SML.

SolidSX [16] is a tool in which CAG-based software model can be visually explored. Information in a CAG can be visualized in terms of treemaps, table lenses and radial views. SolidSX is relatively open-ended with respect to the information contained in a CAG. The creation of a CAG lies outside the scope of SolidSX and must be provided by some other tool such as Recoder. Sextant’s use of TreeMap, Cytoscape enhanced with Sextant plugins, and GraphViz provides similar visualization capabilities to SolidSX.

Polymetric views [17] provides a graphic-based representation of software that utilizes node size, position, and color (including shading) to convey information about software. In particular, software visualization is extended to include software metrics. Polymetric visualizations have been used to assist in reverse engineering and to help characterize system evolution.

AspectMaps [18] is a tool that provides the ability to view crosscutting concerns within a code base. Software elements that can be visualized include, packages, classes, aspects, and methods. Aspects are associated with colors and aspect dependencies (i.e., where aspects apply within a code base) are painted accordingly. If multiple aspects apply to an element it is colored black.

V. Conclusion

This paper takes the position that software metrics have an important role to play in a wide range of activities within the software life cycle. Software metrics, whose definition also encompasses rule-based analysis of the kind used by PMD and FindBugs, can give insight into the nature of software along a number of important dimensions. An interactive environment, such as that provided by Sextant, can amplify the information obtained from metrics. It is important to note that interactivity creates necessary preconditions for feedback-based exploration of software. However, in order to be effective interactive environments must provide (1) a rich framework for the creation of domain/application-specific metrics, and (2) sophisticated means to understand the information provided by such metrics. Sextant is a tool whose design goal is to meet these needs for software activities involving Java source code.

At a more conceptual level, an important characteristic of software metrics is that they can and do continue on in places where type systems must stop. It is beyond these borders of undecidability and tractability that software metrics (especially domain and application-specific metrics) reign supreme. Thus, we end this paper with a recapitulation of the initial statement:

A central concept in the study of software metrics, in the limit, is the art and science of leveraging that which can be measured in order to make predictive statements about that which cannot be measured.

Acknowledgment

The authors would like to thank Farokh Bastani and Bojan Cukic for insightful comments on a draft of this paper.

References