

jCT: A Java Code Tomograph

Markus Lumpe, Samiran Mahmud, and Olga Goloshchapova
Faculty of Information & Communication Technologies
Swinburne University of Technology
Hawthorn, AUSTRALIA
{mlumpe,smahmud,ogoloshchapova}@swin.edu.au

Abstract—We are concerned with analyzing software, in particular, with its nature and how developer decisions and behavior impact the quality of the product they produce. This is the domain of empirical software engineering where measurement seeks to capture attributes affecting the product, process, and resources of software development. A well-established means to study software attributes is metrics data mining. But, even though a variety of frameworks have emerged that can distill desired measures from software systems (e.g., *JHawk* or *SonarJ*), a systematic approach to collecting measures from large data sets has still eluded us. For this reason, we have developed the Java Code Tomograph (jCT), a novel framework for metrics extraction and processing. jCT offers an extensible measurement infrastructure with built-in support for the curated repositories *Qualitas Corpus* and *Helix*. With jCT, large-scale empirical studies of code within the same software system or across different software systems become feasible. In this paper, we provide an overview of jCT’s main design features and discuss its operation in relation to the effectiveness of the framework.

Index Terms—Software metrics; Data mining; Reasoning about programs.

I. INTRODUCTION

In diagnostic medicine, the terms CT scan or MRI stand for a medical imaging technique to visualize the internal structures of a human body. The aim of this technique is to produce, in a non-invasive manner, a three-dimensional image through *computed tomography* – a process that joins a series of two-dimensional images through superimposition to yield the corresponding three-dimensional representation. Computed Tomography offers two important benefits. First, its inherent high resolution allows for a very precise analysis. Second, the raw data from a scan can be rearranged so that different diagnostic tasks can be performed simultaneously without having the patient scanned another time.

High resolution data mining and the ability to slice-and-dice the resulting information are also two important tenants in empirical software engineering. Software engineering, like many other disciplines, relies on a high level approach for evolving the knowledge in the field [1]. A mere observational practice is not enough. Software engineering can only succeed, if it is equipped with a rigorous experimental component that provides support (a) to study theories about the nature of software, (b) to investigate application domains and programming paradigms, and (c) to analyze developer behavior and decisions as manifested in the code they produce [2]–[4].

Measurement is the fundamental element here [2]. To better understand the process being used to build a software system

and the software system itself, we can identify a set of characterizing properties (or attributes), define associated software metrics, and perform an empirical analysis to interpret the data [5]. These measurement activities enable us to improve development processes and practices, and provide us with the means to perform quality assurance within available project resources [1]. However, “*measurement is never better than the empirical operations by which it is carried out*” [6]. Precision is paramount. The quality and quantity of data resulting from the measurement process must not arbitrarily limit analysis. Suitable post-processing can eliminate potential noise, if necessary.

Rigorous measurement is a key feature of the *Java Code Tomograph* (jCT), an extensible metrics extraction infrastructure for Java code. jCT is both a high-resolution data miner and a powerful tool for performing both quantitative and qualitative empirical analyses. jCT’s extraction engine supports the complete Java-6 class file specification [7] including the proposed extensions for dynamic programming [8]. Moreover, jCT provides built-in support to traverse *Qualitas Corpus* [2] and *Helix* [9], two recently emerged *curated* open-source software repositories to facilitate repeatable empirical studies. At present, jCT is the only system that is designed to work with these repositories.

II. JCT

We have developed jCT framework specifically to help understand software systems from empirical data. That is, to be effective, we require it to (a) process non-trivial software systems, (b) support the addition of new measures easily, and (c) yield a natural and adaptive approach for meaningful interpretation of metrics data.

jCT is a stand-alone Java application and it operates on bytecode rather than source code. We found [3], [4] that class files, which contain simultaneously both IL-bytecode and meta data, yield nearly the same information as source files. Bytecode can, therefore, be considered isomorphic to source code. Moreover, due to the absence of any source-level syntactic sugar and auxiliary information, bytecode provides a more direct access to the information we seek.

Except for the internal representation of Java bytecode, we do not use a dedicated meta-model to map the measured artifacts to language-neutral representations (e.g., FAMIX [10]) in jCT. Instead, we employ the semantics of the language (i.e., Java-6, see Table I) as the source for metrics data mining. The

TABLE I
STRUCTURE OF A COMPILED JAVA CLASS [7].

General	Fully-qualified class name Super class name Interfaces implemented Modifiers Constant Pool: numeric, string, and type constants Annotation* Attribute*
Field*	Modifiers, name, and type Annotation* Attribute*
Method*	Modifiers, name, return type, and parameter types Annotation* Attribute* (including Java IL-bytecode)

data retrieved from a Java class file (*i.e.*, *.class*) is stored in a jCT-internal `JavaClassFile` instance. No information is omitted. Although, this approach implies that each supported language requires its own extraction process, the benefit is that we are able to capture all of the raw data. There is no need to compromise. Raw data can always be aggregated to a level suitable for analysis.

One of the reasons for relying on raw data rather than a full-featured meta model is due to our experiences developing similar analysis frameworks and code instrumentation libraries (*e.g.*, CLI [11]). The use of a meta model increases the demands on and the complexity of the techniques and algorithms governing the meaningful interpretation of the differences in the mapped languages. There may even be a need to reverse a meta model mapping to capture the true semantics of a language element [10]. Consequently, we argue that a full-featured meta model creates a remarkable engineering challenge and maintenance overhead, which is orthogonal to the purpose of metric extraction itself. By using the Java-6 specification [7] as the model for representing classes in memory, jCT avoids these meta model-related issues.

Nevertheless, in order to analyze a given class, we usually have to have access to the underlying bytecode instructions of that class. However, in the Java class file itself, bytecode is just a sequence of binary numbers [7]. To interpret it, we need to convert these numbers into instances of `ILInstruction`, the internal jCT representation of the Java IL-bytecode instruction set [12]. But, this is a very expensive operation in terms of memory consumption. We, therefore, adopt a technique used in operating systems or graphical user interface programming and treat bytecode as a *resource* that is loaded on demand. In other words, before measures can be extracted for a class, we have to *lock* its bytecode and map it to instances of `ILInstruction`. Once metrics extraction for that class is completed, these instances can be released. For this reason, in jCT we perform metrics extraction *class-by-class* for each measure.

jCT embeds itself into software measurement and analysis as shown in Figure 1. Every analysis starts with an initial research question. To explore that question, we need artifacts and a suitable set of metrics definitions to mine those artifacts. Running jCT yields metrics data sets, raw data to be post-

processed by statistical software, summarized (visually, if necessary), or compared against existing data sets. We perform these operations either to generate the knowledge we seek or to trigger additional extraction steps with possibly new and refined research questions and metrics definitions.

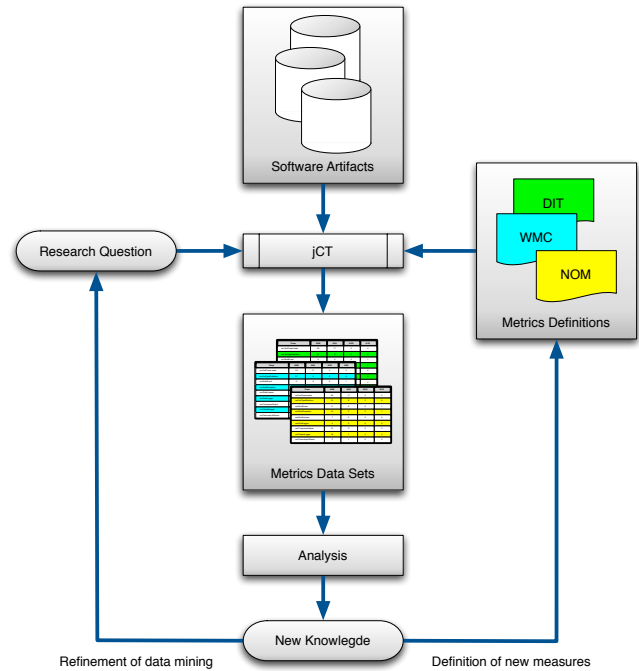


Fig. 1. jCT-induced measurement process.

By design, the extraction engine of jCT does not provide any default measures *per se*. Instead, we use a light-weight extension mechanism to supply desired metrics definitions to it. This mechanism is *viewpoint-agnostic*, that is, when defining a new measure we are not bound to adhere to any predefined analysis model. The resulting flexibility allows us to tailor every measure to the specific needs of the intended analysis. For example, we can use jCT not only to capture a variety of object-oriented class and method metrics (*e.g.*, the CK metrics suite [5]), but also to emulate *javap* to build our own Java class file disassembler or to emit class and method graphs for independent graph-based analyses.

However, in jCT, every metrics definition must define the required recording process, per class file, as a subclass of `MetricsCollector`. This abstraction defines jCT's extension point to weave a newly defined measure into the metrics extraction engine. This approach enforces a strict *separation-of-concerns*. A metrics definition only needs to specify what we have to distill from a class file in order to meet the associated objectives. Applying the measure to all classes of a software system is the responsibility of the metrics extraction engine.

A jCT-supported measure has to implement at least four methods: `flushHeader`, `setup`, `setOption`, and `extractDataPass1`. The methods `flushHeader` and `setup` define auxiliaries to emit suitable data column header

TABLE II
MEMORY PROFILING STATISTICS.

Runtime Object Allocation				
Active Instances			Total Instances	
Rank	Type	Size in %	Type	Size in %
1	char []	34.59	char []	31.51
2	ConstantUtf8	8.53	int []	24.34
	RedBlackTree.RedBlackNode	8.07		
3	byte []	6.68	ILInstruction	12.67
4	LineNumberInfo	5.94	ILArgument	5.58
	LocalVariableInfo	5.17		
5	ContantNameAndType	4.66	ILShortIndexToConstantPoolArgument EntryIterator	4.22 3.20
	ConstantPoolEntry []	4.48		

information and to initialize local variables for the recording process, respectively, if necessary. But, it is the method `setOption` that links a defined measure with the extraction engine. jCT uses a configuration manager that, at the beginning, instantiates all available metrics definitions. That is, in order for a metrics definition to be recognized in jCT, a measure needs to announce its capabilities, in terms of appropriate command line attributes, to the metrics extraction engine. This is the role of the `setOption` method.

The attributes measured may exhibit mutual dependencies, like *Coupling Between Object classes* (CBO) [5]. For this reason, jCT’s metrics extraction engine employs a 2-pass tactic for metrics extraction. By default, a metrics definition always has to implement the first pass (*i.e.*, `extractDataPass1`). The second pass (*i.e.*, `extractDataPass2`) provides an empty default behavior, defined in `MetricsCollector`, and only needs to be overridden when a measure requires it. For example, the CK-measure *Weighted Methods per Class* (WMC) [5] can be collected in one pass, whereas CBO takes two passes to process in order to resolve type dependencies.

At present, we have implemented more than 500 measures to capture a multitude of aspects at varying degrees of granularity. We use the file system to store the metrics data sets. These sets are hierarchically structured. jCT uses a “binning” strategy to organize metrics data depending on the analysis task performed. This enables us not only to conduct comparative studies of different systems, for example, part of Qualitas Corpus or Helix, but also to run evolutionary studies on the same system over time.

Finally, jCT provides built-in analysis support for Qualitas Corpus [2] and Helix [9] that come with the promise to enable equally repeatable large-scale empirical studies of code and the validation of software metrics in general. Combined, Qualitas Corpus and Helix offer over 1,200 releases from more than 100 open-source Java software systems. To extract metrics data from Qualitas Corpus, for example, jCT reads the `.properties` file [2] located in the root directory of every system in the corpus. This file contains meta data, which jCT can use to construct the input data set necessary to perform metrics extraction. In particular, we use the `sourcepackages` entry to determine, which classes are *core* (as defined by Qualitas Corpus [2]) and should be considered for analysis. This step is required since contemporary

software systems, in general, make extensive use of third-party libraries. Therefore, we need a mechanism to select the right set of artifacts for analysis.

III. JCT’S RUNTIME PROFILE

So, how does jCT cope with large data sets and what is its typical resource consumption profile? To answer this question we run an experiment on a Mac Pro equipped with one 2.66 GHz Quad-Core processor, 8GB 1066 MHz DDR3 memory, and running Mac OS X 10.6.8 and the Qualitas Corpus version 20101126r [2]. This release contains 106 Java software systems with a median system size of 853 classes, where `jasml-0.10` is the smallest system comprising only 49 classes and `netbeans-6.9.1` is the biggest one containing a total of 32,475 classes. For each system we distilled approx. 140¹ different object-oriented class, method, and field measures relating to both size and complexity (*e.g.*, *Number of Methods* [3], *Number of Getters* [4], or *Response for a Class* [5]).

When running a metrics extraction with jCT, 80% of the runtime memory is devoted to just 14 built-in Java and jCT types, with strings (*i.e.*, `char []`) requiring more than 30% alone. Strings, in terms of both active instances and total instance, naturally require the most memory. The garbage collector of JVM does not guarantee the removal of unreferenced strings from the internal string pool. As a consequence, a string-intensive Java application, like jCT, may eventually run out of memory. For this reason, we use a dedicated string heap that can be cleared between individual extraction tasks in order to remove unused strings from memory.

The data shown in Table II also demonstrates two important aspects in metrics extraction with jCT. The most memory-intensive active instances are comprised exclusively of class file and dataset related entities. That is, only the data needed to represent Java classes in jCT is kept in memory at all times. On the other hand, the most memory-intensive total instances are associated the light-weight meta model for the analysis of IL-bytecode (*e.g.*, `int []`, `ILInstruction`, and `ILArgument`). However, these instances are released as soon as the metrics extraction for the host class has been completed,

¹We record separate counts for storage and visibility modifiers. For example, the *Number of Methods* of a class yields 24 sub-measures: 6 for **static** methods, 6 for instance methods, 6 for **abstract static** methods, and 6 for **abstract** instance methods. It is this detailed breakdown that allows for a fine-tuned analysis of developer behavior [3], [4].

hence easing the cumulative demand on memory for metrics extraction in jCT.

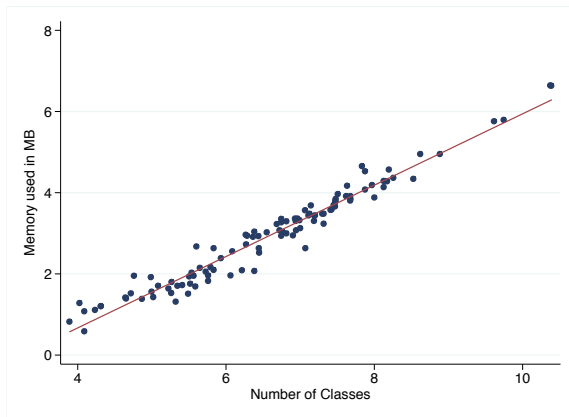


Fig. 2. Memory consumption per system size (log scale).

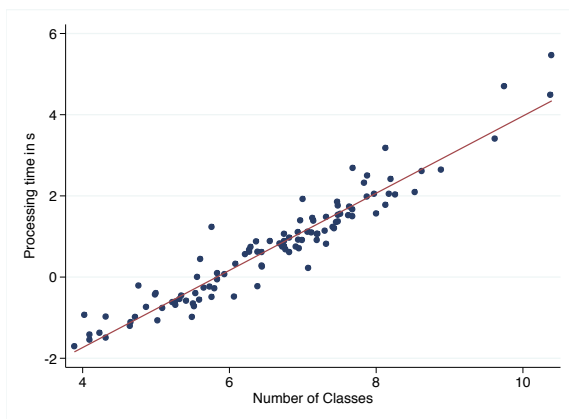


Fig. 3. Processing time per system size (log scale).

Metrics extraction with jCT is $O(n)$ (cf. Figure 2 and Figure 3). Both memory consumption and running time are proportional to the size of the system being analyzed. We observe a strong positive correlation between system size, in terms of number of classes, and memory consumption (*i.e.*, $\rho = 0.98$) and running time (*i.e.*, $\rho = 0.96$). For our selected 140 metrics, we find that the analysis of a class with jCT takes approx. 3ms to complete and requires around 20kB runtime memory on average. However, these values are estimates and can vary (*i.e.*, $\pm 20\%$ from the mean) depending on the nature of the analyzed classes and systems as well as the inherent complexity of the measures being collected.

IV. CONCLUSION AND FUTURE WORK

An important element in improving software development with respect to both the process and the product is the ability to monitor and measure it [5]. Measurement is fundamental to engineering. However, whereas traditional engineering disciplines had centuries to perfect measurement, software engineering has only completed the first steps. So, if we are to succeed in advancing the knowledge in the field, then we

need an accessible, yet comprehensive, approach to measure software.

We have designed jCT to close this gap. jCT is an extensible framework that comprises the features which empower us to perform precision data mining and analysis of Java-based software systems effectively. In addition, jCT is also tuned to work with Qualitas Corpus [2] and Helix [9], the largest currently available curated open-source Java repositories. This makes jCT an ideal tool to conduct evolutionary and comparative studies of software. Experiments must be repeatable. An *ad hoc* collection of systems cannot achieve this. Both Qualitas Corpus and Helix have been carefully designed not only to reduce the costs on empirical studies, but also to provide a representative sample of software artifacts. With jCT's built-in support for new metrics definitions, we can perform empirical studies, which are both repeatable and extensible.

Analysis techniques are constantly improving and the number of suitable systems for analysis increases steadily. We will, therefore, continue to explore new repositories and languages in order to grow the jCT framework and make it more versatile. This will enable us not only to study a greater variety of solution designs and their associated effects, but also to perform studies to capture the impact of the means to construct software, the chosen programming languages, to the way software engineers utilize them in practice.

REFERENCES

- [1] V. R. Basili, "The Role of Experimentation in Software Engineering: Past, Current, and Future," in *Foundations of Empirical Software Engineering*, B. Boehm, Rombach, H. D., and M. V. Zelkowitz, Eds. Springer, 2005, pp. 1–13.
- [2] T. Ewan, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies," in *Proceedings of 17th Asia Pacific Software Engineering Conference*, Sydney, Australia, Dec. 2010, pp. 336–345.
- [3] R. Vasa, M. Lumpe, P. Branch, and O. Nierstrasz, "Comparative Analysis of Evolving Software Systems Using the Gini Coefficient," in *Proceedings of 25th IEEE International Conference on Software Maintenance*. Edmonton, Alberta: IEEE Computer Society, Sep. 2009, pp. 179–188.
- [4] M. Lumpe, S. Mahmud, and R. Vasa, "On the Use of Properties in Java Applications," in *Proceedings of the 21st Australian Software Engineering Conference*, Auckland, New Zealand, Apr. 2010, pp. 235–244.
- [5] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, Jun. 1994.
- [6] S. Stevens, "On the Theory of Scales of Measurement," *Science*, vol. 103, no. 2684, pp. 677–680, Jun. 1946.
- [7] A. Buckley, "JSR 202: Java(TM) Class File Specification Update," Dec. 2006, <http://jcp.org/en/jsr/detail?id=202>.
- [8] J. Rose, "JSR 292: Supporting Dynamically Typed Languages on the Java(TM) Platform," Feb. 2011, <http://jcp.org/en/jsr/detail?id=292>.
- [9] R. Vasa, "Growth and Change Dynamics in Open Source Software Systems," Ph.D. dissertation, Swinburne University of Technology, Faculty of Information and Communication Technologies, Oct. 2010.
- [10] S. Ducasse, "Reengineering Object-Oriented Applications," University of Berne, Institute of Computer Science and Applied Mathematics, IAM-03-008, Tech. Rep., Sep. 2001.
- [11] M. Lumpe, "Using Metadata Transformations to Integrate Class Extensions in an Existing Class Hierarchy," in *Proceedings of the Fourth ASIAN Symposium on Programming Languages and Systems (APLAS 2006)*, N. Kobayashi, Ed. Sydney, Australia: Springer, LNCS 4279, Nov. 2006, pp. 290–306.
- [12] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, 2nd ed. Addison-Wesley, Apr. 1999.