JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2004

Vol. 7, No. 2, Special Issue OOPS Track at SAC 2007, February 2008

AOP➥HiddenMetrics: Separation, Extensibility and Adaptability in SW Measurement

Walter Cazzola
DICo - Department of Informatics and Communication,
Università degli Studi di Milano
cazzola@dico.unimi.it

Alessandro Marchetto
Fondazione Bruno Kessler - IRST
marchetto@fbk.eu

Traditional approaches to dynamic system analysis and metrics measurement are based on system code (both source, intermediate and executable code) instrumentation or need *ad hoc* support by the run-time environment. In these contexts, the measurement process is tricky, invasive and the results could be affected by the process itself making the data not germane.
Moreover, the tool based on these approaches are difficult to customize, extend and often use since their properties are rooted at specific system details (e.g., special tools such as bytecode analyzers or virtual machine goodies such as the debugger interface) and require high efforts, skills and knowledges to be adapted.
Notwithstanding its importance, software measurement is clearly a nonfunctional concern and should not impact on the software development and efficiency. Aspect-oriented programming provides the mechanisms to deal with this kind of concern and to overcome the software measurement limitations.
In this paper, we present a different approach to dynamic software measurements based on aspect-oriented programming and the corresponding support framework named AOP➥HiddenMetrics. The proposed approach makes the measurement process highly customizable and easy to use reducing its invasiveness and the dependency from the code knowledge.

Keywords: Software Metrics, AOP, Separation of Concerns.

1 INTRODUCTION

Aspect-oriented programming (AOP) [9, 8] is a powerful technique to better modularize object-oriented programs by introducing crosscutting concerns in a safe and noninvasive way. Each aspect-oriented approach is characterized by a *join point model* (JPM) consisting of the *join point*s, a mechanism for identifying the join points (*pointcut*s) and a mechanism for raising effects at the join points (*advice*).

Cite this article as follows: Walter Cazzola and Alessandro Marchetto: AOP➥HiddenMetrics: *Separation, Extensibility and Adaptability in SW Measurement*, in Journal of Object Technology, vol. 7, no. 2, Special Issue OOPS Track at SAC 2007, February 2008, pp. 53–68 http://www.jot.fm/issues/issues_2008_2/article3

The advice is woven at the selected join points, i.e., the *weaving process* looks at the application bytecode for the points described by a pointcut and instruments those points with the advice code.

The aspect-oriented mechanisms better address functionality that orthogonally crosscut the whole implementation of the application. The measurement process is a typical crosscutting concern whose implementation tangles the code of many objects in the system. Software metrics and their measurement are logically self-contained and easy to be modularized and kept separated from the application code but their measurement strictly depends on what they are calculating and it is intimately bound to the application code.

The aspect-oriented programming provides the mechanisms for reducing the invasiveness necessary to measure the software, and therefore for widening the software measurement applicability. In our view, the metrics can be realized by aspects and the weaving process will instrument an application with the software measurement code. This approach does not require either the application code or the knowledge of its implementation. The software measurement process can be easily plugged into and unplugged from the application code.

In this paper we are going to present our AOP➥HiddenMetrics framework based on this idea to support the dynamic software measurement in a noninvasive way. Furthermore, we will show how to use it and how to extend it to support new assets and metrics.

The rest of the paper is organized as follows: in section 2 we give an overview of the AOP➥HiddenMetrics framework; in particular, we will present the supported metrics, the framework model and implementation. Section 3 shows the framework at work. Section 4 discusses the benefits/drawbacks of the approach and examines some related work and, finally, in section 5 we draw up some ideas for future works and conclusions.

2 AOP➥HiddenMetrics

The AOP➥HiddenMetrics framework is an *adaptable* tool to support *noninvasive* and *modular* software measurements. It is basically oriented to dynamic measurements neglecting more static and traditional metrics. Adaptability and noninvasiveness are achieved by exploiting the aspect-oriented technology; this renders quite easy to extend the framework by supporting new metrics (more details in the rest of the section).

A Glance at the Supported Metrics

The AOP➥HiddenMetrics framework focuses its efforts on analyzing the assets of a Java applications during its execution, hereafter *system under analysis* (SUA).

The framework supports many different measures on several software assets. The supported metric suite merges some existing metrics, in particular, it is inspired by [1,2,11] and [13]. To provide a uniform exposition, we refer to classes and aspects as *module*s and to methods and advice as *operation*s. The considered software assets are:

Coupling. The coupling represents the connection degree between two or more software components at run-time. The considered metrics are: coupling on module call (CMC), coupling on field access (CFA), response for a module (RFM).

Cohesion. The cohesion represents the degree to which software elements within a module are related each other (at run-time) and work together. The considered metrics are: field use (FU), and lack of cohesion of operations (LCOO).

Memory Usage. In this category, we consider only the *memory usage* (MU) metric; it computes the minimum and the maximum quantity of memory allocated by a module during its execution.

Concurrency. The CONCUR metric measures how much concurrent is a program. In Java, threads are the concurrency unit, so, CONCUR will measure how many threads have been activated during the SUA execution.

Code execution. The EXEC metric measures how many times a portion of code is executed and it accesses to each component of the SUA.

Code coverage. Code coverage (CC) and Dead code (DC) metrics analyze the differences between the static and running SUA code. For instance, the dead code metric measures the declared code that is not reachable by executing the SUA.

These metrics are from [1, 2, 11] and [13]; for sake of brevity we do not further explain them. The metric suite should not be considered exhaustive nor fixed since our goal is to support software measurement process through an easy to adapt and extend tool.

Framework Rationale

The AOP➥HiddenMetrics framework exploits AspectJ [8] to render the measurement process noninvasive, pluggable and unpluggable and to provide a tool to easily enhance the supported metric suite and measured assets. The metrics are implemented as aspects that will be woven to the SUA on demand. These aspects encapsulate the code necessary to measure the SUA and describe how the integration will take place; the weaving process will bind the process measurement code to the SUA code and the metric will be computed during the SUA execution. To measure an application, the user has to:

- define a set of SUA executions and code the related test cases (e.g., through the JUnit[1] framework);
- choose the metrics and weave the corresponding aspects to the SUA bytecode (and test cases);
- execute the test cases and wait for the aspect to collect the measures.

The key idea behind the AOP➥HiddenMetrics model consists in encapsulating the computation of a metric in an aspect and weaving it to the SUA bytecode. In this way, the software measurement process is independent of the SUA as long as the aspect is not tailored on the SUA code. Moreover, the whole measurement process can be plugged and unplugged without any specific knowledge of the SUA code or its availability.

In general, the computation of a metric is characterized by what must be counted, e.g., the number of method calls. This information identifies which points (in the AOP parlance join points) must be taken in consideration during the measurement and drives the definition of a set of *pointcut*s to select those *join point*s. The advice will collect the data related to the measurement process. As an example, the CMC (coupling on module call) metric takes in consideration all the interactions of a module with another through its operations. In this case, the pointcuts must select all the calls to operations belonging to another module and the corresponding advice will count these calls. The aspects implementing the previously described metric suite are bundled with the AOP➥HiddenMetrics framework as a case study. This suite can be easily extended with new metrics by adding the aspects to measure them.

In particular, the AOP➥HiddenMetrics framework does not analyze the SUA bytecode but "patches" (through the weaving process) it with the code to compute the metric during the SUA execution. The weaving process allows the user to enable/disable the measurement process on the SUA. Furthermore, the measurement process can be easily customized by coding ad hoc test cases that selectively run the SUA. For instance, to calculate a metric on a specific set of packages rather than on the entire SUA we have only to tune the pointcuts on these packages.

The main challenge of encapsulating the computation of a metric in an aspect consists of coding it not tailored on a given SUA but applicable to any SUA. For instance, in [13], the authors try a similar approach but the aspects are built on the chosen SUA hindering their reuse. We got a more general approach by capturing the join points activated by the test cases instead of selecting them on the whole application. In this way, the aspects do not depend on the SUA implementation rather the dependency of the SUA is confined to the test case that is defined in terms of the SUA methods by definition.

[1] http://www.junit.org.

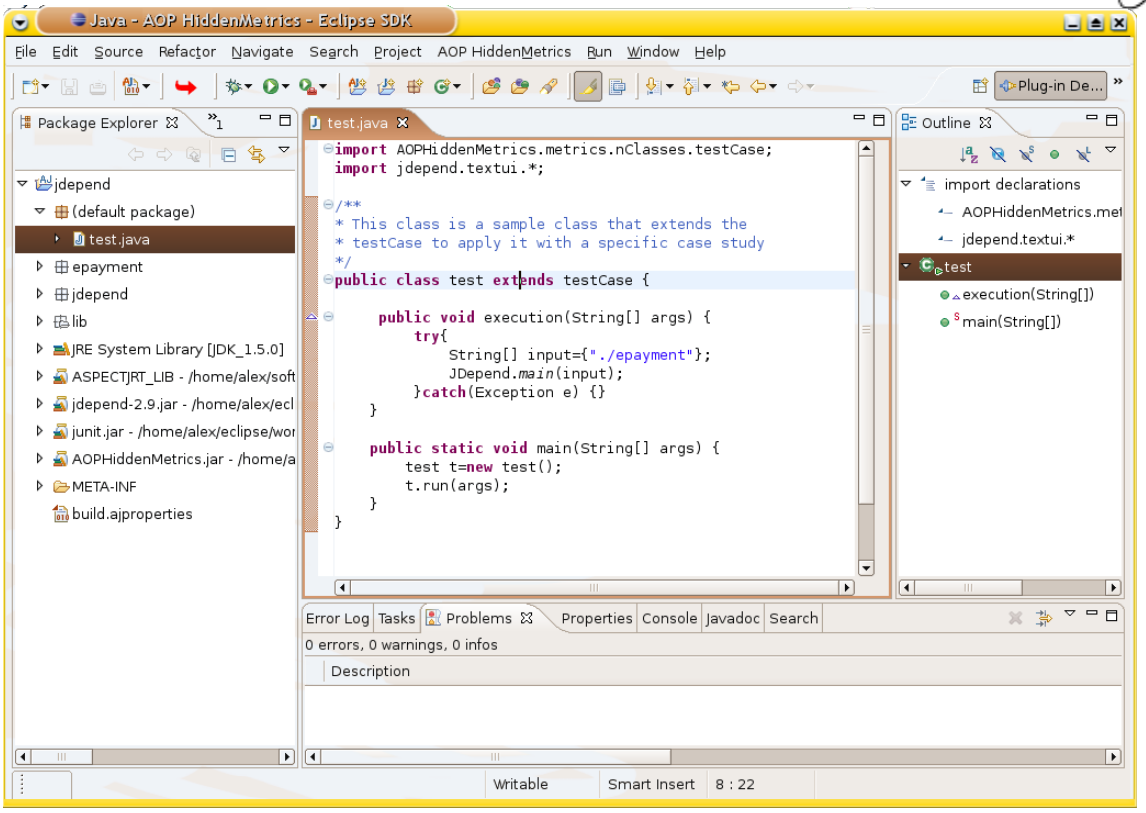


Figure 1: **AOP➥HiddenMetrics Eclipse Screenshot**

Therefore, our framework deals with the measurement process as it were a non-functional crosscutting concern and the aspect-oriented techniques allow the modularization of this concern and the noninvasive analysis of the SUA.

Software implementation

The AOP➥HiddenMetrics framework is basically implemented by three components:

- a library of aspects and ancillary classes realizing the supported metrics;
- a viewer for the data collected by the woven metrics during the SUA execution; and
- an Eclipse[2] plug-in (Fig. 1) to ease the framework use.

The woven aspects calculate the corresponding metrics during the SUA execution and output the collected data as XML files. The viewer graphically shows the data (thanks to XSLT transformations) to ease their analysis, understanding and

[2] http://www.eclipse.org.

```
public aspect LCOO {
  // jps excluded from the measurement process.
 pointcut excluded_executions():
  cflow(call(* java..*.*(..))) ||
  cflow(call(* javax..*.*(..))) ||
  withincode(AOPHiddenMetrics..*);

  // jps on the method executions.
 pointcut methods_executions():
  (execution(* *..*.*(..)) ||
   execution(static * *..*(..)) ||
   adviceexecution()) && !excluded_executions();

  // jps on the access to the fields.
 pointcut access_to_fields():
  (get(* *) || set(* *)) && !excluded_executions();

 before(): access_to_fields() {
  /* it inspects the thisJoinPointStaticPart and stores which field has been accessed
     and by which method. */
 }
 Object around(): methods_executions() {
  /* it inspects the thisJoinPointStaticPart and retrieves the used methods. */
 }
}
```

Listing 1: the LCOO metric aspectualization.

interpretation. The Eclipse plug-in allows the user to select and unselect the metrics to compute.

In the rest of this section, we will focus on the aspect library since the metrics aspectualization is symptomatic of our approach and (we believe) more interesting.

In the library, each package supports a different metric and provides its aspectualization, a set of ancillary classes to support the measurement process, the data collecting and reporting and, finally, a skeleton class for the test cases. The skeleton class must be extended when the user wishes to tailor the measurement process on a specific SUA. In particular, to automate the SUA execution and measurement, the user must adapt the skeleton class to invoke the starting method of the instrumented SUA with the necessary inputs.

In [7], AOP is expressed in terms of *quantification* and *obliviousness*. The SUA is not prepared to be analyzed by our tool, it is unaware of the metric presence and work (*obliviousness property*). On the other side, the aspects are not tailored on the SUA; they provide the basic mechanisms for measuring a generic SUA without assumptions on the SUA behavior and structure but dynamically adapting to it (*quantification property*). These concepts are essential for the AOP➥HiddenMetrics

```
public aspect CMC {
  // jps excluded from the measurement process.
 pointcut excluded_executions(): ...
 pointcut methods_executions():
  (call(static * *..*(..)) || execution(* *..*.*(..)) ||
   call(* *..*.*(..)) || call(*..*.new(..)) ||
   execution(*..*.new(..))) && !excluded_executions();
 before(): methods_executions() {
  Object caller = thisJoinPoint.getThis();
  String cls = "";
  if (caller == null)
    cls = thisEnclosingJoinPointStaticPart.
        getSignature().getDeclaringTypeName();
  else cls = caller.getClass().getName();
  String mthCls = thisJoinPointStaticPart.
        getSignature().getDeclaringTypeName();
  if (!mthCls.equals(cls)) {
    String[] mthName = getMethodName(thisJoinPoint);
    _fun_data.add_data(cls, mthName, 1);
  }
 }
 public String[] getMethodName(JoinPoint matched) {
  /* it reflectively extracts information on the method from «matched» and returns
     them. */
 }
}
```

Listing 2: the CMC metric aspectualization.

framework; the gained *independence* of the metrics from the SUA implementation renders our framework general-purpose and usable in different contexts without extra efforts.

Let us consider the LCOO metric and its aspectualization (Listing 1). To compute this metric, we have to know:

1. which methods belong to a class, and
2. which attributes they access.

Therefore, we need two pointcuts: `access_to_fields()`, and `methods_executions()`. The former selects all the accesses to the attributes and the latter all the called methods. The advice associated with the pointcuts respectively computes the number of methods accessing a field and the total number of methods.

Generally speaking, the AspectJ pointcuts allow us to analyze a Java software without knowing it because are based on given and generic join points as method calls or field accesses and rarely the metric aspectualization has to be tailored on a

specific join points such as a call to a method of a given name. This specific behavior only occurs when we want to measure a specific execution trace but, in this case, the dependency is confined to the test case realizing that execution trace.

The advice parts are in charge of processing the information at the join points and computing the metrics. To this aim, the advice can exploit reflection (through the special reference variable: **`thisJoinPoint`**) to extract the data (e.g., the target object and its type, the called method, the caller object, and so on) necessary to compute the metric at the matched join point maintaining the independence of the SUA. Listing 2 shows the aspect for the abovementioned CMC metric putting the emphasis on the advice.

3 AOP➥HiddenMetrics AT WORK

Before going on, we recall that the AOP➥HiddenMetrics does not introduce new metrics but it only supports the measurement process. In this section we will show how our framework works and prove that it produces sound results and how it can be extended.

Adding New Metrics

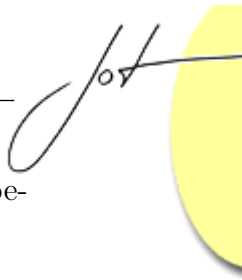
There are two main steps to support new metrics: *metric definition* and *aspect building*. The metric definition consists of defining (i) the entities to measure (e.g., operations, fields); and (ii) how they are aggregated (e.g., in classes, in aspects).

In the aspect building step, we have to write an aspect that encapsulates the measurement process for the new metric defined in the first step. Thus, we have to:

1. define a set of pointcuts to select/exclude some execution points (i.e., join points) as required by the measurement process;
2. write the advice to analyze the selected joint points and collect the data about the metric; and
3. write a set of ancillary classes to support the measurement, e.g., to temporary store data.

In this first example, we extend our framework to support the dynamic response for a module (d-RFM) metric; a variant of the well-known RFM metric that calculates the coupling between modules in terms of the number of different operations *directly* invoked from an invoked operation. Following our stepwise algorithm we have:

1. to define the metric characteristics
   i. to count how many operations are used by a module that are defined in another;



ii. to aggregate the used operations by module to compute the coupling between modules.

2. to build the corresponding aspect

   i. the pointcut (`dRFMcall()`) must select the execution of all the operations (methods, constructors or advice) invoked from an operation included library calls

```
pointcut dRFMcall():
 cflow(execution(* *..*.*(..))) &&
   ( adviceexecution() ||
     execution(* *..*.*(..)) ||
     execution(*.new(..))
   ) && !within(AOPHiddenMetrics..*);
```

   ii. in the advice associated with the `dRFMcall()` pointcut must collect the data about callee and caller modules; it uses **`thisJoinPoint`** to extract these data from the join point. Since the code is quite similar to that in listing 2 for sake of brevity we do not describe it any further.

   iii. the intermediate results need to be stored and an `Hashtable` can do the work.

In this second example, we are going to support the measurement of a new asset: exceptions. In particular, we add the support for the EXCP metric that measures frequencies and types of the exceptions raised during the execution of a given system scenario. To this regard, information about the exception handlers and the raised exception need to be recorded when an exception is trapped.

1. to define the metric characteristics

   i. to compute frequencies and types of raised exceptions;

   ii. to aggregate the raised exceptions by module;

2. to build the corresponding aspect

   i. the pointcut `EXCPtrap()` must capture all the raised exceptions

```
pointcut EXCPtrap(Exception e) :
  args(e) && handler(Exception+) &&
        !within(AOPHiddenMetrics..*);
```

   the related data are grasped through the `args` primitive pointcut

   ii. the advice associated with the `EXCPtrap()` pointcut extracts the data related to the trapped exception.

```
before(Exception e) : EXCPtrap(e) {
  StackTraceElement[] st = e.getStackTrace();
  Vector v = new Vector();
  String classOfExc = st[0].getClassName();
  v.add(classOfExc); // type of the raised exception «e»
  v.add(st[0].getMethodName()); // who has raised the exception «e»
  v.add(st[0].getLineNumber()); // where the exception «e» occurred
  if (!_data.ex_contains(classOfExc))
       _data.set_ex(classOfExc,v);
}
```

iii. the computed results are stored in the predefined storage unit (`storeData`).

Measuring the SUA

To show the framework at work, we have measured the JDepend[3] application. JDepend is a tool that computes several metrics on Java applications through static analysis. It is composed of more than 3000 lines of code implementing about 50 classes subdivided into 4 packages.

To measure JDepend, we have generated a test case class by specializing the abstract test cases class as follows.

```
import AOPHiddenMetrics.metrics.nClasses.TestCase;
import jdepend.textui.*;

/* This class is a sample class that extends the TestCase class
   to apply it with a specific case study. */
public class Test extends TestCase {
  public void execution(String[] args) {
    try {
      String[] input = { "./epayment" };
      JDepend.main(input);
    } catch(Exception e){e.printStackTrace();}
  }

  public static void main(String[] args) {
    Test t = new Test();
    t.run(args);
  }
}
```

The code for the test case is also in the main canvas of the Eclipse screenshot (Fig. 1). Note that JDepend can display its reports through several different devices (e.g., plain text file, XML files, or on a graphical GUI). The test case we have written

[3] http://clarkware.com/software/JDepend.html.

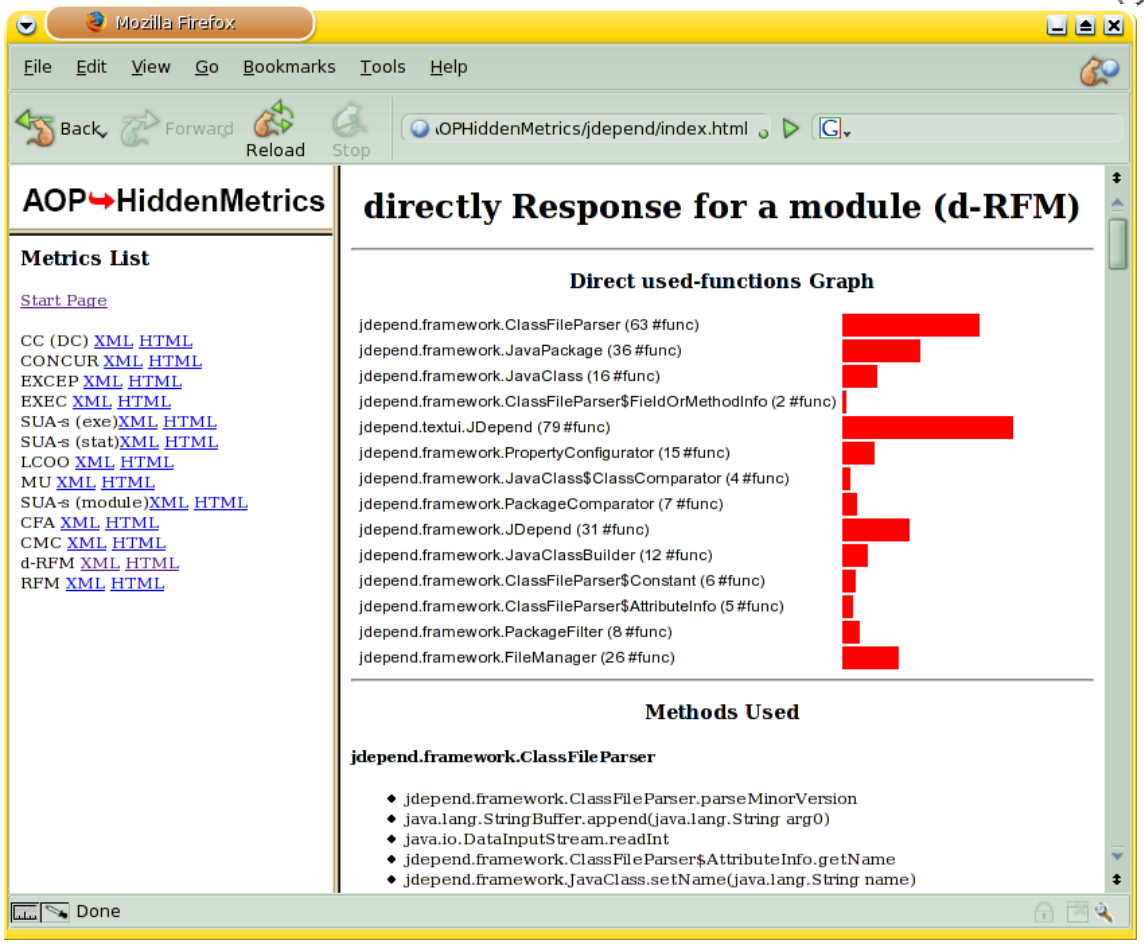


Figure 2: **AOP➥HiddenMetrics on the jDepend tool.**

forces JDepend to produce its report as a plain text file, this means that only the code related to this device will be measured and not the whole application. This choice is to emphasize that the AOP➥HiddenMetrics framework is quite different from the other ones such as JDepend and AOPMetrics since it performs dynamic analysis. Thence, the calculated measures are *strictly related* to the application execution trace we choose in the test case. For instance, we know that the JDepend tool is composed of 50 classes but considering our test case the AOP➥HiddenMetrics framework considers only 14 classes as shown in Fig. 2. This is due to the fact that several classes are not used by JDepend to report the computed measures as text file, — e.g., all the classes related to the graphic interface. Therefore, to perform an analysis of the whole tool we have to write a test case that covers all the possible execution traces (when possible).

Once the test cases have been written and the metrics to measure have been chosen, the AOP➥HiddenMetrics framework weaves the aspects for the chosen metrics on the SUA and on the test cases and finally executes the test cases. During the

execution the woven code computes the metrics and stores the results into XML files that can be shown through our viewer. For instance, Figure 2 shows a screenshot of the browser reporting the data collected measuring the JDepend application, in particular it is displaying the results for the *directly response for a module* (d-RFM) metric.

4 DISCUSSION AND RELATED WORK

Conventional approaches to software measurements are based on code/bytecode instrumentation or supported by specific JVM enhancement, e.g., JVMTI. In this section, we will discuss advantages and drawbacks of the aspect-oriented solution with respect to the traditional approaches and glance at some related work.

In [11], Mitchell et al. show how to adapt two common object-oriented metrics (coupling and cohesion) to be applied at run-time. To collect the results of the measurement process, the authors explore the use of a modified JVM and the JVM Debug Interface (JVMDI) to run the SUA. In [12], the authors explore the use of BCEL [3] to instrument the bytecode of the SUA. This last approach requires the modification of the application bytecode to dynamically acquire the data and adds overhead to the software execution but provides a more object-level accuracy rather than the JVMs instrumentation. Similar approaches have been followed in [4] and [1].

On the other hand, [13] adopts the aspect-oriented technology to profile the software execution and to increase the program comprehension of a given fragment of Java code. In particular, AOP is used to trace the methods execution for a single code fragment (e.g., for a specific class), to define all the calls to a given code fragment, and to define if the same fragment is really used by another Java code fragment. In [6], Figueiredo et al. present a set of metrics capturing several object- and aspect-oriented artifacts (properties as separation of concerns, coupling, cohesion, and size). They introduce an aspect-oriented tool to measure these metrics but they focus on static measurements. AOPMetrics [14] is a tool that calculates a set of object- and aspect-oriented metrics for Java applications. This tool uses the AspectJ compiler to compile the entire SUA source code and then computes the metrics on the application's syntax tree.

The AOP➥HiddenMetrics framework that we have presented in this paper is quite different from all of these tools/approaches. It focuses on the computation of a set of dynamic metrics related to some software assets. The framework dynamically collects data from the SUA execution and computes the metrics on them. In this way, the user can analyze different execution traces and behaviors of the same SUA. In other words, differently from the traditional approaches based on the static analysis of the application source code, the AOP➥HiddenMetrics framework can measure the same software artifact on different execution contexts.

The AOP➥HiddenMetrics framework (similarly to [13]) uses AOP to perform its dynamic analysis and define a noninvasive approach to measure software systems.

The use of an aspect to analyze software executions renders the approach more powerful and flexible than the traditional approaches (such as [11,4,1]) since it lets us perform a very object-level accuracy analysis such as the bytecode instrumentation but working at a higher-level of abstraction. Moreover, the AOP➥HiddenMetrics framework can be easily extended to support the measurement of new metrics by adding new aspects and applying them to the SUA.

With respect to the approach described in [13], our framework can analyze the whole Java application based on the user-defined executions and independently of its structure and behavior. Moreover, it is a complete framework that profiles the software executions, analyzes these executions to compute the metric-data, stores this data in files and defines some views of these data to help the user to understand and analyze them.

To compare classical and aspect-oriented approaches we consider three assets:

- the accuracy of the performed analysis;
- the developer effort, i.e., the effort required to a developer to implement a measurement-tool using a given approach; and
- the user effort, i.e., the effort required to a user to measure systems by using the built tool.

Summarizing, through an aspect-oriented approach and code instrumentation we analyze a SUA at object-level and calculate metrics related to several software attributes. While, by using a JVM-based approach we may (also) extract information about the JVM itself such as its state and the allocated memory. By using AOP rather than code instrumentation it is difficult to calculate metrics related to lines of code (e.g., statements, conditions) since we (usually) cannot define pointcuts at this level but this comes from the limits of AspectJ [10, 15]. The aspect-oriented approaches better fit the analysis of both aspect- and object-oriented SUA. Using JVM-based approaches is difficult to focus on specific elements (e.g., objects) but the whole system will be analyzed even if our analysis could be confined in a portion of the SUA code or on a given trace execution since it is difficult or impossible to disable the instrumentation on per object basis. On the contrary, an aspect-oriented technique lets the user define pointcuts to confine its analysis to specific portions of the SUA code.

To implement measurement-tools using AOP appears to be easier than other approaches since it requires (only) a bit knowledge about the adopted aspect-oriented language and because through AOP we "transparently" control our code instrumentator. In other terms, we do not use specific tool (e.g., BCEL that works at bytecode level) or special JVMs for producing instrumented version of systems to be analyzed. On the contrary, through pointcuts and advice we easily control our instrumentation that is automatically done during the weaving process. Furthermore, an AOP-based measurement-tool is more flexible and easy to extend or customize (e.g., for analyzing a specific set of classes) rather than other approaches that require more efforts

and skills to modify/adapt its instrumentator. For instance, in the case we would like to qualify an existing measurement-tool to calculate a new metric. Using a JVMs approach we need to modify and reconfigure the virtual machine to add this feature and this is not an easy task. Through a JVMTI approach, we need to verify if the provided events notification mechanism allows us to capture the needed information and, in this case, we need to update the instrumentator. Otherwise, we cannot implement the new metric. Using code instrumentation, we need to update the instrumentator and then (also) the metrics-calculation module. These operations may require high efforts, skills, and specific knowledge to be done since we need to work at code level (e.g., on bytecode) with *ad hoc* tools. Using an AOP-based approach we need to define the pointcuts of interest for the new measure and then write their related advice to calculate the metric. Thus, the use of AOP simplify implementation, development and maintenance of measurement-tools.

Finally, a measurement-tool based on AOP is easy to apply for analyzing and measuring applications since user needs (only) to weave the applications bytecode with the metric/aspects used to trace executions and calculate metrics. Thanks to the weaving process user does not need to manage several versions of SUA (instrumented/not instrumented) and/or *ad hoc* developed JVMs as well as required by other approaches.

5 CONCLUSIONS AND FUTURE WORKS

In this paper we document our tree-steps experience: (i) we study the state of the art in terms of approaches used to perform dynamic analysis; (ii) we develop the (first) AOP-based tool for measuring dynamic assets of systems; and (iii) we apply it to Java applications (e.g., JDepend).

The AOP➥HiddenMetrics framework analyzes the applications and collects data by using a set of aspects (written in AspectJ) to perform a noninvasive software measurement. To measure a given SUA the user must weave the chosen metrics to the SUA bytecode. Then, through predefined test cases she/he executes the woven SUA and automatically computes and collects the metrics data.

Our future work will focus on augmenting the set of metrics supported by the AOP➥HiddenMetrics framework. In particular, this extension will regard metrics strictly related to aspect-oriented specific software assets such as pointcut measurements, coupling between aspects, and so on (for example, see [4] and [5]). Furthermore, we are going to simplify the AOP➥HiddenMetrics customization mechanism by implementing a tool that drives the user during the customization process. Finally, to support our considerations, we are driving a comparative test that involves several approaches to software measurement such as AOP, source and bytecode instrumentation, JVM-based.

References

[1] E. Arisholm, L. C. Briand, and A. Føyen. Dynamic Coupling Measurement for Object-Oriented Software. *IEEE Trans. Softw. Eng.*, 30(8):491–506, Aug. 2004.

[2] M. Ceccato and P. Tonella. Measuring the Effects of Software Aspectization. In *Electronic Proceedings of the 1st Workshop on Aspect Reverse Engineering (WARE 2004)*, Delft, The Netherlands, Nov. 2004.

[3] M. Dahm. Byte Code Engineering. In *Java-Informations-Tage*, pages 267–277, 1999.

[4] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic Metrics for Java. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*, pages 149–168, Anaheim, California, USA, Oct. 2003. ACM Press.

[5] B. Dufour, C. Goard, L. Hendren, O. de Moor, G. Sittampalam, and C. Verbrugge. Measuring the Dynamic Behaviour of AspectJ Programs. In J. Vlissides, editor, *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*, pages 150–169, Vancouver, BC, Canada, Oct. 2004. ACM Press.

[6] E. Figueiredo, A. Garcia, C. Sant'Anna, U. Kulesza, and C. Lucena. Assessing Aspect-Oriented Artifacts: Towards a Tool-Supported Quantitative Method. In F. Brito e Abreu, C. Calero, M. Lanza, G. Poels, and H. A. Sahraoui, editors, *Proceedings of the 9th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'05)*, Glasgow, Scotland, July 2005.

[7] R. E. Filman and D. P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In *Proceedings of OOPSLA 2000 Workshop on Advanced Separation of Concerns*, Minneapolis, USA, Oct. 2000.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and B. Griswold. An Overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, LNCS 2072, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *11th European Conference on Object Oriented Programming (ECOOP'97)*, Lecture Notes in Computer Science 1241, pages 220–242, Helsinki, Finland, June 1997. Springer-Verlag.

[10] C. Koppen and M. Störzer. PCDiff: Attacking the Fragile Pointcut Problem. In *Proceedings of the European Interactive Workshop on Aspects in Software (EIWAS'04)*, Berlin, Germany, Sept. 2004.

[11] A. Mitchell and J. F. Power. Toward a Definition of Run-Time Object-Oriented Metrics. In *Proceedings of the 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'03)*, Darmstadt, Germany, July 2003.

[12] A. Mitchell and J. F. Power. Using Object-Level Run-time Metrics to Study Coupling Between Objects. In *Proceedings of the 2005 ACM Symposium on Applied Computing (ACM SAC'05)*, pages 1456–1463, Santa Fe, New Mexico, USA, Mar. 2005. ACM.

[13] D. Ng, D. R. Kaeli, S. Kojarski, and D. H. Lorenz. Program Comprehension Using Aspects. In *In Proceedings of the ICSE Workshop on Directions in Software Engineering Environments (WoDiSEE'2004)*, Edinburgh, Scotland, May 2004.

[14] M. Stochmiałek. AOPMetrics. Master's thesis, Wrocław University of Technology, Poland, 2005.

[15] T. Tourwé, K. Gybels, and J. Brichau. On the Existence of the AOSD-Evolution Paradox. In *Proceedings of the Workshop on Software-engineering Properties of Languages for Aspect Technologies (SPLAT'03)*, Boston, Massachusetts, Apr. 2003.

ABOUT THE AUTHORS

Walter Cazzola (Ph.D.) is currently an assistant professor at the Department of Informatics and Communication (DICo) of the Università degli Studi di Milano, Italy. His research interests include reflection, aspect-oriented programming, programming methodologies and languages. He has written and has served as reviewer of several technical papers about reflection and aspect-oriented programming. He can be reached at cazzola@dico.unimi.it.

Alessandro Marchetto (Ph.D.) is currently an assistant researcher at the Center for Scientific and Technological Research (IRST) of the Bruno Kessler Foundation in Trento (www.fbk.eu/irst). His primary research interests include quality, verification and testing of Software Systems and, in particular, of Web-based systems. He can be reached at marchetto@fbk.eu.