iPlasma:
An Integrated Platform for Quality Assessment of Object-Oriented Design

C. Marinescu, R. Marinescu, P.F. Mihancea, D. Raţiu, and R. Wettel

LOOSE Research Group
"Politehnica" University of Timişoara
Romania
{lrg}@cs.utt.ro

Abstract

To make software maintenance easier, a superior quality of its design and implementation process must be ensured. For this reason, existing software must be supported by automated systems for analysis, diagnose and design improvement, at a high level as well as at a level close to source code. iPlasma *is an integrated environment for quality analysis of object-oriented software systems that includes support for all the necessary phases of analysis: from model extraction (including scalable parsing for C++ and Java) up to high-level metrics-based analysis, or detection of code duplication.* iPlasma *has three major advantages: extensibility of supported analysis, integration with further analysis tools and scalability, as it was used in the past to analyze large-scale projects in the size of millions of code lines (e.g. Eclipse and Mozilla).*

1 Introduction

In order to increase the maintainability and the flexibility of a software system, the quality of its design and implementation must be properly assessed. For this purpose a lot of analyses are described in the state of the art literature. In order to apply them on large software systems we need a set of tools dedicated to this purpose.

During this tool demonstration we intend to present the iPlasma quality assessment platform. Based on a "hands-on" example, we are going to present how this suite of tools can be used to perform advanced analyses that assess the design quality of object-oriented software systems.

2 Overview

Figure 1 presents the layered structure of iPlasma[1] quality assessment platform. Notice that the tool platform, starts

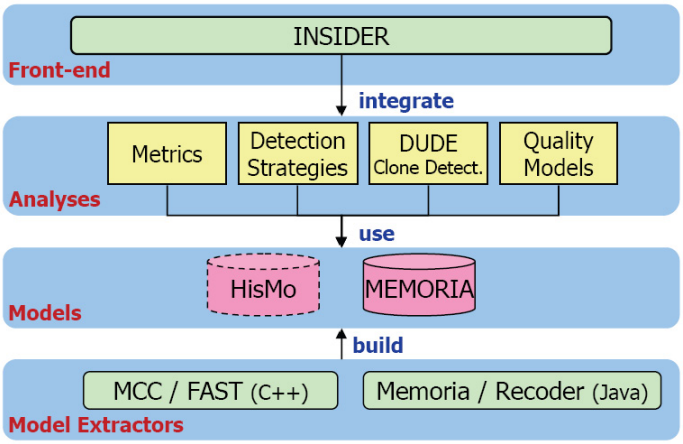

Figure 1. The iPlasma analysis platform

directly from the source-code (C++ or Java) and provides the complete support needed for all the phases involved in the analysis process, from parsing the code and building a model up to an easy definition of the desired analyses including even the detection of code duplication, all integrated by a uniform front-end, namely INSIDER.

Through the next paragraphs we are going to briefly introduce the main components of the iPlasma quality assessment platform.

[1] iPlasma stays for Integrated PLAtform for Software Modelling and Analysis.

3 Models and Model Extractors

The first step in analyzing the design of a software system is obtaining a model of it. Therefore, at the bottom level, we have tools for *model extraction*. Currently, we support two mainstream object-oriented languages *i.e.,* C++ and Java.

3.1 Model Extractors

An essential (and very painful) task in a process of software analysis is the construction of the proper model of the system. The purpose for constructing the model is to extract from the source code the information which is relevant from the point of view of a particular goal. Thus, for analyses focused on object-oriented *design* it is important to know the *types* of the analyzed system, the *functions* and *variables* and information about their *usages*, the *inheritance relations* between classes, the *call-graph* etc.

For Java systems we use the open-source parsing library called RECODER[2] to extract all these information in form of an object-oriented meta-model i.e., MEMORIA (see Section3.2).

MCC (Model Capture for C++) [6] is a tool which extracts the aforementioned design information from C++ source code (even incomplete code!), based on Telelogic's FAST parsing library. It receives as input a directory containing the source code and it produces a set of related (fully normalized) ASCII tables containing the extracted design information (including even information about templates). This information could be easily loaded in a RDBMS and interrogated in form of SQL queries. But, in IPLASMA we load this information in the MEMORIA *object-oriented model*, which is described next.

3.2 MEMORIA: A Unified Meta-Model

MEMORIA [3] is a meta-model that can represent Java and C++ systems in a uniform manner, by capturing design information (e.g., classes, method interactions). One of the key roles of Memoria is to provide a consistent model even in the presence of incomplete code or missing libraries, to allow the analysis of large systems and to ease the navigation within a system.

By analyzing only a single version of a system we miss important information related to the *evolution* of the system. Therefore we extended MEMORIA so that it can keep also information about multiple versions of a system. The result is the HISMO[8] meta-model which allows us to define on top of it various analyses related to the dynamics of the system during its evolution (e.g., logical couplings between modules can be detected).

4 Analyses

Based on the extracted information we can then define several types of design analyses (e.g., metrics, metrics-based rules for detecting design problems [5], quality models etc.). Next, we are going to discuss how these analyses are supported in IPLASMA.

4.1 Metrics

Our platform contains a library of more than 80 state-of-the-art and novel design metrics that can be applied at different levels of abstraction ranging from system-level metrics used to obtain an overview of the system to primitive metrics which describe the details within a single method. The metrics can be divided into the following categories: size metrics – measure the size of the analyzed entity (e.g., Lines of Code), complexity metrics – measure the complexity of the analyzed entity (e.g., Cyclomatic Complexity), coupling metrics – measure the data coupling between entities (e.g., Coupling Between Objects) and cohesion metrics – measure the cohesion of classes (e.g., Tight Class Cohesion).

4.2 SAIL: An Easy Way to Implement Structural Analyses

Design-related analyses can be implemented using almost any programming languages (e.g., Java). Unfortunately, almost all these implementations will be hard to reuse and understand and thus they hinder the correlation between the results of the implemented analyses. This happens because a single normal programming language does not provide all the proper mechanisms to ensure the easy implementations of many different analyses. Therefore we defined SAIL (Static Analysis Interrogative Language) [4] as a dedicated language for structural analyses, built on top of the MEMORIA meta-model. The language provides a set of powerful mechanisms which facilitate a concise and natural expression of the implemented analyses. For example, if we want to obtain the package named **my.package**, we would write the following code sequence:

```
Package myPack;
myPack = select (*) from sysPackages
         wherename = "my.package";
```

On the one hand, the efficient querying mechanism introduced in SAIL (i.e., the select statement) contributes decisively to reducing the complexity overhead found in structure based approaches when complex navigation must be combined with filtering. On the other hand, because query results can be stored in SAIL variables, it becomes possible to break down the excessively complex monolithic queries often encountered

in the repository based approach. In this manner, the understandability and the changeability of the analyses implementation have been increased. SAIL also provides modularity mechanisms that allow us to reuse analyses and to compound them into more complex types of code inspections.

4.3 Detection Strategies

A detection strategy [5] is a quantifiable expression of a rule by which design fragments that are conforming to that rule can be detected in the source code. Using this quantification mean we can automatically detect design flaws in a software system, design entities which present some deviations from good object-oriented design criteria, thus hindering the easy evolution of the system and affecting its design quality.

The major problem with any metrics-based approach, including detection strategies, is that a couple of threshold values must be used in order to detect those design entities which present some "abnormal" characteristics. The DSTM (Detection Strategy Tuning Machine) [7] implements a method metaphorically named tuning machine which can be used to establish the proper threshold values for a detection strategy. An important advantage of this method is that a detection strategy can be calibrated for a particular development environment. The drawback is that the method needs human feedback for a period of time in order to produce good results.

4.4 Dude: Detection of Code Duplication

An important design heuristic states that the source code of a software system must "say everything once and only once". Copying code, sometimes followed by slight adaptations is a wrong way of code reuse. The maintainability of such a system becomes a nightmare, since every further change has to be propagated in all the places where the code has been copied. Moreover, along with the code, bugs are copied as well.

DUDE (Duplication Detector) [9] is a tool that uses textual comparison at the level of line of code in order to detect fragments of duplicated code. Its powerful detection engine can also cover various "adaptations" of the duplicated code (such as variables renaming or statement insertion/removal). These modifications can scatter a monolithic copied block into many smaller, thus hardly noticeable, duplicated fragments. The duplication chain concept enables DuDe to recover larger blocks that were subject to such modifications, usually overlooked by traditional line-based approach clone detectors. Moreover, it provides flexibility by means of different thresholds, which are responsible for filtering out non-significant duplication chains.

5 Insider: the Integrating Front-End

As we have seen, assessing the design quality of an object-oriented system requires the collaboration of many tools. Using them independently can easily transform the analysis process into a nightmare, making it completely unscalable for usage on large-scale systems.

One of the key aspects of iPLASMA is that all these analyses are *integrated* and can be used in a uniform manner through a flexible front-end, called INSIDER [1]. In other words Insider is

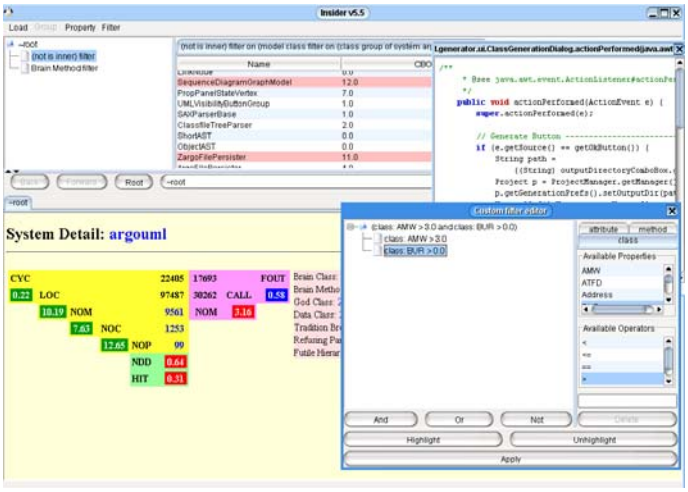

Figure 2. A snapshot of INSIDER

a front-end (see Figure 2) which offers the possibility to integrate independent analyses (in form of plugins) in a common framework. This approach makes INSIDER *open implemented* and thus easily extendable with any further needed analyses.

6 Conclusion

Although iPLASMA was developed as a research tool, it is not a "toy". It was successfully applied for analyzing the design of an important number of "real-world" systems including very large-scale systems (>1 MLOC), like **mozilla** (C++, 2.56 million LOC) and **eclipse**, (Java, 1.36 million LOC).

iPLASMA is available for free download at: **http://loose.utt.ro/iplasma/iplasma.zip**. We would appreciate very much if you could let us know about using iPLASMA. In return we would do our best to provide you with support and updates.

References

[1] C. Caloghera; *Evolutionary Integrated Analysis Environment for Software Systems*, Diploma thesis, "Politehnica" University of Timişoara, 2004.

[2] COMPOST Team; *Recoder Project*, http://recoder.sourceforge.net/, University of Karlsruhe.

[3] D. Raţiu; *Memoria: A Unified Meta-Model for Java and C++*, Master thesis, "Politehnica" University of Timişoara, 2004.

[4] C. Marinescu, R. Marinescu, T. Gîrba; *Towards a Simplified Implementation of Object-Oriented Design Metrics*, In Proceedings of Metrics 2005, Como, 2005 (to appear).

[5] R. Marinescu; *Detection Strategies: Metrics-Based Rules for Detecting Design Flaws*, In Proceedings of The 20th IEEE International Conference on Software Maintenance (ICSM), Chicago, 2004.

[6] P.F. Mihancea; *The Extraction of Detailed Design Information from C++ Software Systems*, Master thesis, "Politehnica" University of Timişoara, 2004.

[7] P.F. Mihancea, R. Marinescu; *Towards the Optimization of Automatic Detection of Design Flaws in Object-Oriented Software Systems*, In Proceedings of CSMR, Manchester, 2005.

[8] S. Ducasse, T. Gîrba, J.M. Favre; *Modeling Software Evolution by Treating History as a First Class Entity*, Workshop on Software Evolution Through Transformation (SETra 2004), 2004.

[9] R. Wettel; *Automated Detection of Code Duplication Clusters*, Diploma thesis, "Politehnica" University of Timişoara, 2004.