Continuous Quality Assessment with inCode

George Ganea[a,*], Ioana Verebi[a,*], Radu Marinescu[a]

[a]*LOOSE Research Group, Universitatea "Politehnica" Timişoara, Bvd. V.Parvan 2, Timişoara, Romania*

Abstract

In spite of the progress that has been made over the last ten years in the research fields of software evolution and quality assessment, developers still do not take full advantage of the benefits of new assessment techniques that have been proposed by researchers. Beyond social factors, we believe that there are at least two main elements that contribute to this lack of adoption: (i) the insufficient integration of existing techniques in mainstream IDEs and (ii) the lack of support for a continuous (daily) usage of QA tools. In this context this paper introduces INCODE, an Eclipse plugin aimed to transform quality assessment and code inspections from a standalone activity, into a continuous, agile process, fully integrated in the development life-cycle. But INCODE not only assesses continuously the quality of Java systems; it also assists developers in taking restructuring decisions, and even supports them in triggering non-standard, complex refactorings. The main focus of the paper is to introduce those features of INCODE that make it innovative, and to describe its architecture and key design decisions.

Keywords: quality assessment, design quality, design problems, code smells, software metrics, Eclipse plugin

1. Introduction

There is no perfect software design. Like all human activities, the process of designing software is error prone and object-oriented design makes no exception. The flaws of the design structure, also known as "bad smells" [Fow99] have a strong negative impact on quality attributes such as flexibility or maintainability. Thus, the identification, detection and correction of these design flaws is essential for the evaluation and improvement of software quality.

In spite of the fact that all of today's major IDEs provide extensive quality assurance modules that use metrics to control design quality, in the last years it became more and more clear that the major problem when it comes to assessing design quality is that when metrics are used in isolation, they are too fine grained to quantify comprehensively one investigated design aspect [Mar04] (*e.g.,* distribution of system's intelligence among classes). Thus, in most cases individual measurements do not provide relevant clues regarding the cause of a problem. In other words, a metric value may indicate an anomaly in the code but it leaves the engineer mostly clueless concerning the real cause of the anomaly. As a consequence of the previous remark, in practice it is very hard to correlate an abnormal metric value with a concrete restructuring measure, that would improve the quality of the system's design.

The typical usage scenario of a quality assessment module (and/or methodology) is currently this: a developer, feeling that something is wrong with the design/code, is using the QA module provided by (or available for) her IDE to compute a suite of metrics; noticing some abnormal metric values, she must *infer* what the *real* design problem is from the informal description of the metric's interpretation model. This is not easy at all, especially when the analysis occurs long after that code/design fragment has been created, and/or the code was written by someone else. But even after finding out what the problem is, correcting the design flaws moves the developer to another world, where she must compose the proper restructuring solution using the basic refactorings available in her IDE. This is again a challenging and painstaking operation.

We believe that this process is so tedious because of two reasons: (i) metrics used to detect design flaws are only "detection atoms", and, therefore incapable of pointing out to relevant correction (restructuring) solutions; (ii) refactorings, as they are used now, are also only the "correction atoms", and therefore they do not represent the correction solution for all but non-trivial design problems.

Certainly, the issue of adoption of academic technology by the industry is a complex matter. Telea *et al.* quote senior managers [TVS10] stating that adoption depends on measurable added value brought by a tool and the corresponding cost. Furthermore, Bessey *et al.* [BBC+10] indicate that adoption might be challenged by diversity in the language dialects, compilers and platforms used, as well as social restrictions (dis)allowing certain modifications, while Kemerer [Kem92] emphasizes that the learning curve plays also a significant role.

*Corresponding Author

Email addresses: georgeganea@gmail.com (George Ganea), ioanaverebi@gmail.com (Ioana Verebi), radu.marinescu@cs.upt.ro (Radu Marinescu)

In this paper we introduce INCODE[1], an Eclipse plugin aimed to transform quality assessment and code inspections from a standalone activity, into a continuous, agile process, fully integrated in the development life-cycle. INCODE supports programmers with continuous detection of design and code problems (*e.g.,* duplicated code [Fow99], classes that break encapsulation [Rie96], or misplaced methods). INCODE identifies and locates specific design flaws *as they appear*, which improves over the efficiency of performing code reviews from time to time, because it allows developers to address a problem at the best possible moment *i.e.,* when the entire design context is fresh to them. Furthermore, INCODE provides *contextualized explanations* for each instance of a detected problem, and it guides engineers in correcting the detected flaws.

While the improved efficiency of continuous quality code analysis over periodic code reviews needs to be rigorously evaluated, we believe that it can be used as a legitimate assumption, which is worthwhile to be rigorously evaluated in the coming future. Furthermore, in order be truly efficient, continuous assessment needs not only adequate tools, but also a clear process (methodology). While this paper touches the issue of the assessment process, its main focus is to show that, from a technical point of view, tools assessing quality continuously are feasible, even when used on large-scale projects.

The rest of the paper is organized as follows: next, we will illustrate the way INCODE supports continuous quality assessment by means of a typical usage scenario that "interweaves" current development and quality assessment. Section 3 presents systematically INCODE's operation mode and its main features, followed by a discussion (Section 4) about its architecture and the key decision that makes it possible for INCODE to run continuously during development in spite of the complex analysis that it has to perform. Section 5 discusses time and memory performance of INCODE. The paper is wrapped-up by a discussion of related work (Section 6) and a series of conclusive remarks and a look at future perspectives.

2. A typical scenario of continuous assessment

Before describing in more detail the features of INCODE, we will illustrate by a simple yet significant example its main trait: the capacity to automate in a continuous manner the detection of design problems, as well as its context-sensitive support for automatic restructuring.

The scenario starts with Lisa, a typical developer, beginning to write the Rectangle class (see Figure 1[2] - **Step 1**) in Eclipse's editor window. The very moment she finishes writing the 4 lines of the Rectangle class and saves the file, a red square – which is an INCODE *marker* – appears on the left side ruler of her editor, next to the class definition. This notifies her that INCODE has detected a potential design problem. By reading the code, you might have noticed that Rectangle is a class that defines four public attributes, opening the gates for breaking encapsulation. This problem is known in the literature as *Data Class* design flaw [Fow99, Rie96]. In the past we defined metrics-based techniques, called *detection strategies* [LM06] for automatically detecting such design flaws[3]. So, every time Lisa is changing the file, on save INCODE executes it's metrics-based detection rules and displays a red marker for each design flaw that has been detected in that file.

By noticing the red INCODE marker, Lisa wants to find out what the problem is. Because INCODE markers behave exactly like the standard Eclipse ones which signalize compiler errors or warnings, it comes natural to Lisa to click on it (**Step 2**) and find out that the *Data Class* problem has been detected.

Next, when Lisa decides to find out more about the problem, the INCODE TIPS view is opened and she can read a detailed description of what the problem is. One important thing here is that the description is not a presentation of what *Data Class* means in *general*, but an explanation of why Rectangle, *in particular*, is reported by INCODE as a *Data Class* (see text box next to **Step 2**). By reading the text you will also notice that INCODE TIPS includes a section of *Refactoring Tips*, which in this particular case advises Lisa to encapsulate the four attributes of Rectangle because they have no reason to be declared public (as no one is using them from outside the class). Again, the noteworthy aspect here is the *context-sensitive nature* of the refactoring advice.

Finally, there is another thing about this description that needs to be emphasized: while both the detection of the problem and the additional description are based on a significant amount of metrics and further dependency analysis, Lisa is not required to have an understanding of these in order to use INCODE. Lisa does not even need to know exactly what a *Data Class* is. We believe that this is an essential trait for any QA tool in order to facilitate a wide adoption by developers: it has to hide the complexity of the powerful analysis techniques that it uses under a presentation that is easy to understand by developers. It's our tools that have to learn the language of developers, not vice-versa.

Now, assume that after reading the description of the problem, Lisa decides to momentarily ignore it and move on, writing class Client that uses the data members exposed by Rectangle (**Step 3**). Again, a red marker appears left to the line where the definition of method calculate starts. This is because calculate is affected by the *Feature Envy* design problem [Fow99], and INCODE detects it based on the strategy defined in [LM06].

The first thing that Lisa may notice is that the Rectangle's *Data Class* description has been updated (**Step 4a**) [4]; the description now remarks the usage of Client.calculate using Rectangle's data. But, the most remarkable change is the one that occurs in the *Refactoring Tips* part, as now, by the fact that three of Rectangle's public members are used from a *single* external

[1] INCODE can be installed using the standard Eclipse mechanism for plugin installation, using the following update: http://www.intooitus.com/research/incode/

[2] The sequence of steps in our example are summarized in form of numerical labels (which we are going to refer to in the following as **Step X**)

[3] Detection strategies are composed of logical metrics-based conditions that identify those design fragments that are fulfilling the condition

[4] In case Lisa didn't close the INCODE TIPS view in the meantime, the update occurs automatically

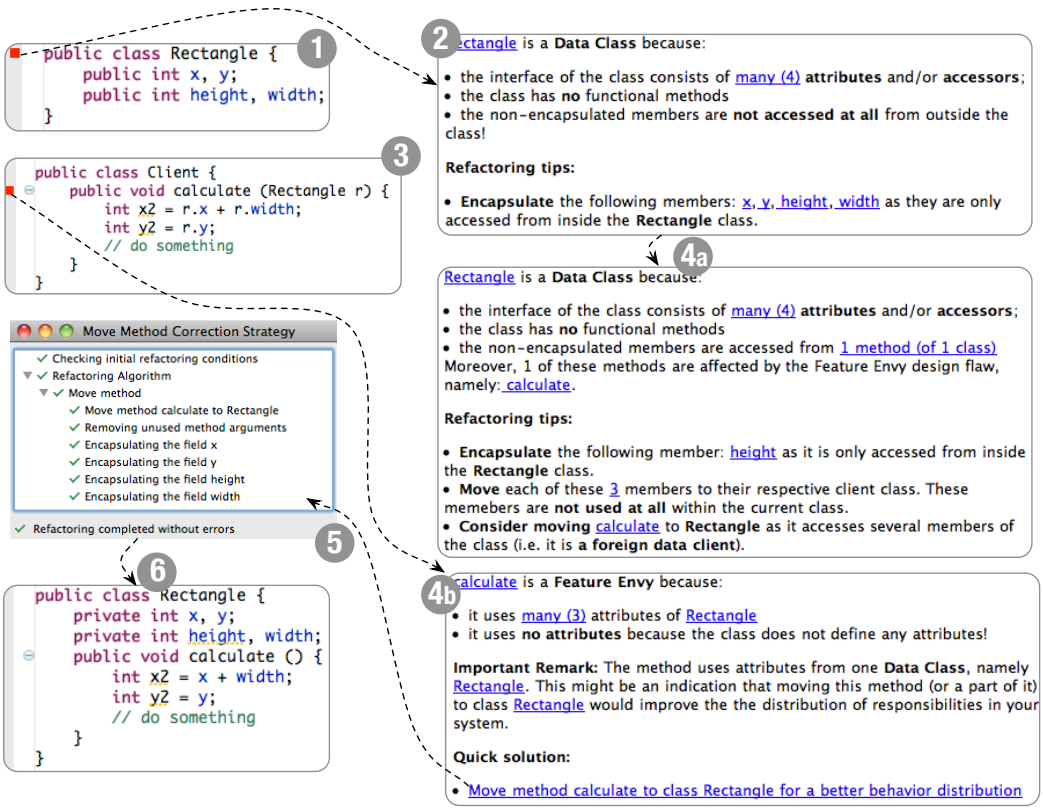

Figure 1: The various steps of using INCODE for design assessment, going from writing a piece of code to the eventual restructuring.

class, INCODE can give different refactoring advices (see description in **Step 4a**); the initial refactoring suggestion is maintained for the height data member, but for the others, due to the new usage context, there are two options: moving the three data members to the Client class or vice-versa, move the calculate method to the Rectangle class.

Beyond the update of Rectangle's problem description, there is one even more interesting thing for Lisa to see: the description of the *Feature Envy* problem detected for calculate (**Step 4b**). Apart from the characteristics already emphasized while presenting the Rectangle's *Data Class* description (*i.e.,* context-sensitivity, continuous update, refactoring advices), there are two additional aspects of INCODE TIPS revealed here: (i) the description contains a remark on the fact that the "foreign" data used by calculate are defined in a class that has been detected as *Data Class*; (ii) it has an additional refactoring section, called *Quick Solution* indicating that INCODE has detected an actual refactoring that Lisa can launch in order to solve the *Feature Envy* problem. In other words, INCODE TIPS supports the design improvement by (i) providing information about meaningful *correlations* detected between various design flaws and (ii) by identifying cases where "clear-cut" restructuring solutions exist and consequently by providing support for automating the code transformation process.

If Lisa decides to solve the problem by moving calculate to Rectangle, she selects the link below *Quick Solution* which, in result, will start the restructuring process (**Step 5**). As seen in Figure 1 the restructuring process consists of a sequence of basic refactorings, which lead to a significantly better solution (**Step 6**) than a simple restructuring like the Eclipse's built-in *Move Method* refactoring.

3. Quality Assessment with INCODE

We used the previous section to illustrate INCODE's most frequent usage scenario *i.e.,* quality assessment performed *continuously* during development. Next, we are going to cover systematically INCODE's main features as a quality assurance tool.

INCODE *Markers.* When the Eclipse workbench is started, INCODE begins to analyze in background the source file currently active in the editor. As seen in Figure 2, when a design problem is detected, INCODE places a *red marker* on the ruler (see[5] **a**), next to the affected class or method. As mentioned before, INCODE markers are similar to those used by Eclipse to indicate compiler errors or warnings. The presence of these markers is very dynamic: as new code is written, or code is modified new markers may appear, or existing markers may disappear.

INCODE TIPS *View.* By launching the *quick fix* for an INCODE marker (see **b**), the INCODE TIPS view (see **c**) is opened, providing a wealth of contextualized information on the particular

[5]In this entire passage we will refer the various parts of Figure 2 by the corresponding labels depicted there

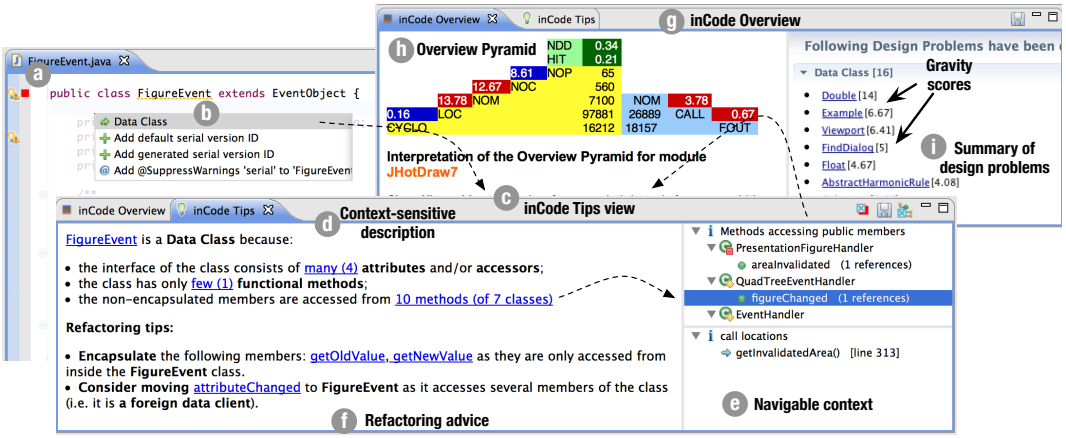

Figure 2: INCODE marks automatically, and updates continuously, code entities with design problems. The INCODE TIPS view can be used to better understand the causes, context and possible remedies for a particular instance of a design problem.

cause (see **d**) of a detected problem. Although the design problems are detected using metrics-based rules [LM06], the description of the problem is in terms of design concepts, rather than numbers; in other words, the description is *hiding* the unnecessary complexity of working with software metrics. Thus, we list the actual entities and relations that contribute to a specific design flaw, as well as the actions that the user can take to improve on these. For example, in the case of a Feature Envy, where the detection rule states that: (i) the method uses many external attribues, from few classes and (ii) it uses none or close to no attributes from its class [LM06], we list the actual external attributes that are used, grouped by class and also the attributes from its own class that the method is using, if any. Also, the description contains a set of hyperlinks that enable the developer to "zoom-in" in exploring in detail the relevant *context* of that problem (see **e**). Furthermore, the description may contain *significant correlations* of this problem with other design flaws detected by INCODE elsewhere in the project, as you have seen in Figure 1 - **4b**).

The other significant part of the INCODE TIPS view is the one that provides *concrete refactoring advices* (see **f**) *i.e.,* hints on how that particular problem instance can be corrected, by taking into account the *entire context of dependencies* of that class/method. Furthermore, if INCODE detects that in a given context a predefined Eclipse refactoring, or a composed restructuring (defined by INCODE) can be applied, the suggested code transformation can be triggered directly from the INCODE TIPS view, as seen in Figure 1. Currently, the set of refactorings that can be triggered automatically is limited, but we are working on extending both the *number of contexts* where INCODE can perform a design improvement, as well as the number of performable restructurings that involve the correlation of multiple atomic refactorings [Ver09].

Detected problems. As mentioned before, we believe that metrics used in isolation cannot help in detecting *real* design problems [Mar04]. Therefore, in INCODE we use *detection strategies* to quantify design problems [LM06]. Currently INCODE detects four well-known design problems related to an improper distribution of intelligence among classes, namely *God Class*, *Data Class*, *Feature Envy* and *Code Duplication* [6]. These four analyses where carefully chosen as to stress the most computation intensive aspects of our model and also because their complexity is comparable to the rest of the design flaws from [LM06]. Moreover, these problems are oftentimes correlated and as such INCODE can use these correlations in order to provide more meaningful information to the user. Thus, while the detection is based on object-oriented metrics, developers do not have to interact directly with metrics. Instead, they can reason about the quality of their design at the conceptual level that is more convenient for them.

INCODE *Overview.* While the continuous assessment mode is what makes INCODE outstanding, we also need a way to "zoom-out" in order to see the global design quality picture for a system. INCODE addresses this need by providing a view that reveals the overall quality of a system, namely INCODE OVERVIEW (see Figure 2). This view has two components: (i) the *Overview Pyramid* (see **h**) that captures the key characteristics of the system/package in terms of complexity, coupling and shape of class hierarchies [LM06]; and (ii) a categorized list of detected design problems (see **i**). From here, the problematic classes and methods can be inspected closer in order to understand for each case the particular causes and the suggested correction steps.

The number of low-level design problems can be overwhelming when analyzing large systems. Therefore in INCODE we compute a *gravity score* (see Figure 2) for each of the four design problems, which indicates for each detected instance how severe the symptoms of the design problem are. As the detection of the design problem is based on metrics [LM06], the *gravity scores* are computed based on how much the various key measures are beyond the thresholds used in the detection rule. For example, a *Data Class* instance with 40 public attributes has a (significantly) higher gravity score than another

[6] A detailed description of these analyses can be found in [LM06]

instance with only 6 public attributes. In order to make the gravity scores comparable, each factor that enters that gravity score is normalized [Mar12], based on a threshold for each metric (a more detailed discussion on how thresholds are selected can be found in [LM06] and [Tri08]). Thus, the categorized lists of design problems are sorted descending based on the gravity score (see **i**). This allows developers to prioritize the inspection and (hopefully) the correction of the most critical design fragments.

4. INCODE's Architecture

At the surface, INCODE is simply an Eclipse plugin for code analysis, with a focus on the detection and correction of design problems. However, looking closer at its internals, INCODE is far more than what developers can perceive by interacting with the user interface. In this section we describe the architecture of INCODE from three different perspectives: first, we will describe how INCODE interacts with the Eclipse ecosystem *i.e.,* how it interacts with and benefits from the comprehensive infrastructure provided by Eclipse. Second, we will reveal that INCODE is not only an analysis tool: it is an analysis framework, that easily allows to plug-ing various analysis ranging from metrics, to software visualizations, detection rules for design problems and complex structure-based analyses. Last, but not least, Section 4.3 describes the design decisions that make it possible for INCODE to run continuously, without affecting the current development activities, an extremely challenging task especially for large-scale projects.

4.1. INCODE *as an Eclipse plugin*

As seen in Figure 3, INCODE has the following workflow: based on the continuously updated Java model, extracted from the source by Eclipse JDT, we create our own model with the subset of design information required for computing the various metrics, for defining and running the higher-level detection strategies and eventually for performing the needed refactorings.

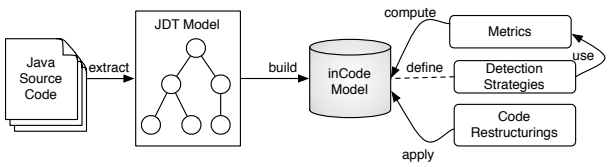

Figure 3: INCODE's Workflow

Figure 3 also reveals another important aspect: the higher-level analyses (Detection Strategies) depend on the metrics component, they do not interact directly with the Eclipse API.

INCODE is an Eclipse plugin; the first pillar on which INCODE relies is the core of the *Eclipse Platform*, from which it mainly uses three components: (i) the *workspace* to get access to the resources of the *Eclipse projects* and their interdependencies; (ii) the *workbench* for a smooth integration of INCODE's user-interface elements; and (iii) the *incremental project build* mechanism of Eclipse. The workbench is used for setting the *markers* which INCODE uses to annotate the Java source files with the detected design problems, and for defining the *views* that INCODE needs for presenting its analysis results. INCODE defines its own incremental builder in order to be able to selectively update its cached information. The incremental project builder is used to get the resources that changed since the latest build and also the resource change delta, which describes the actual changes on a particular resource.

INCODE*'s Dependencies on Eclipse JDT.* The most important dependency of INCODE is on the plugins belonging to the Eclipse JDT project *i.e.,* the plug-ins that implement a Java IDE. INCODE needs information about the program elements that are involved in its various metrics, detection rules or visualizations. Most of this information is provided by the *Java Model*, which is Eclipse's representation of a Java project, providing a light weight model of program elements like package fragments, compilation units, types, methods *etc.*.

However, the *Java Model* is a only a coarse-grained model of Java programs, ranging from java project to type members. Many analyses in INCODE need more in-depth information like methods calls and variable accesses. For this, we need to build the Abstract Syntax Tree (AST) of a file , in which all program elements are available and all bindings (cross-references) are resolved. Because building the AST of a file comes at the price of a considerable performance overhead, INCODE relies on ASTs only when genuinely needed.

4.2. INCODE *as an analysis framework*

INCODE is more than an Eclipse plugin that computes metrics and detects design problems. In the past we developed a wealth of metrics and quality assessment techniques using the iPLASMA analysis environment [MMM+05, LM06]. Therefore INCODE is designed to (i) *inherit* that vast number of analyses defined and implemented in the past; (ii) make it easy to create new analyses on top of the existing ones; and (iii) keep to a minimum the dependency of analyses on the Eclipse platform.

In order to achieve these goals, concrete analyses must not be defined in terms of a concrete meta-model; therefore INCODE is designed with an *additional abstraction layer* on top of a concrete meta-model, namely a *meta-meta model* [BG01]. As seen in the bottom-right side of Figure 4 every type of program entity is modeled as an AbstractEntity that has a specific EntityType; for example, each class is an AbstractEntity object sharing a "*class*" EntityType object. Each EntityType can be characterized by a set of *relations* and *properties* that can be computed on program entities of a particular type. Recalling the example of the "*class*" EntityType object, some examples of relations (groups) are the "*set of methods defined within a class*" or the "*set of classes that are extending it*", while examples of properties for a "class" EntityType can be its"*number of attributes*" or the "*is abstract*" property. Thus, the inCode meta-model is an instantiation of our meta-meta model, we define what entity type can hold what kind of properties (e.g. a class can hold the metric NOM) and what kind of relations (e.g. a method can hold the group of methods it calls). In general, the meta-meta-modelling approach is similar to the one found in the MOOSE

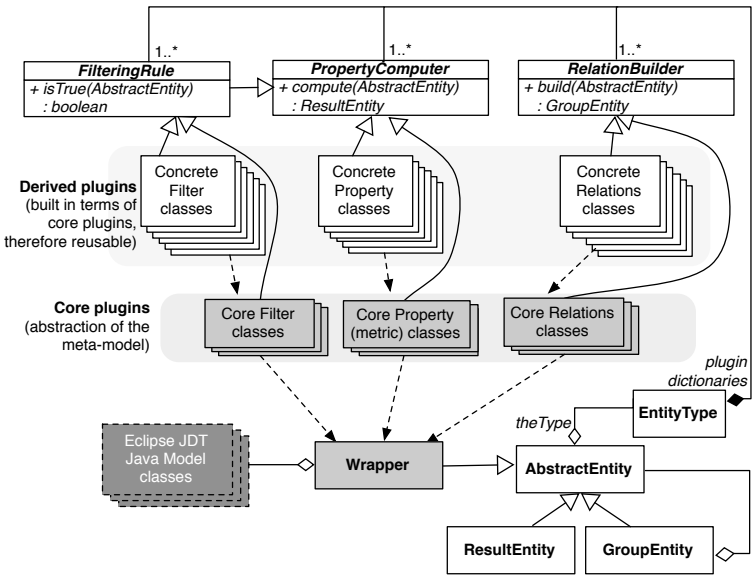

Figure 4: The INCODE system of plugins

analysis environment [NDG05], namely the FAMIX family of language-independent meta-models [DTD01]. As opposed to FAMIX, where the meta-model is explicitly defined via a set of classes, inCode's meta-model is rather implicit, as all model elements are instances of the Wrapper class. Each Wrapper instance can have a different EntityType and as a result will be able to have different properties and relations. For example a Wrapper instance that wraps an IMethod from the JDT model will always have the "method" EntityType and will be able to have properties such as LOC, CYCLO and relations such as "called methods" and "client methods".

The extraction and computation of these properties and relations is encapsulated in different classes associated with a particular EntityType. Thus, RelationBuilder (see Figure 4) is defining the interface for all INCODE plugins that will return a *group of program elements* (AbstractEntity objects) that are associated in some way with the program element passed as a parameter; each concrete RelationBuilder defines a relation between a program element (*e.g.,* a class) and a set of other program elements.

Similarly, PropertyComputer is an interface for all the INCODE plugin classes that will compute and return some property for a given program element (passed as a parameter). All metrics are implemented in INCODE as such PropertyComputer classes. A special type of PropertyComputer plugins are those that return a boolean result (*i.e.,* true/false, like the aforementioned "*is abstract*" property). These INCODE filters are important because all the metrics-based rules for detecting design problems (*e.g., God Class*, *Feature Envy etc.*) are in fact such boolean properties.

All these concrete properties and relations fall into two categories, depending on the way they are computed/extracted:

1. *Core Plugins* – these are the property and relation plugins that provide information by extracting it directly from the code, by using a parsing infrastructure. For example, the "*set of methods defined within a class*" is such a *core relation* plugin and "*method is abstract*" is a *core property/filter* plugin because they both extract information directly from the Eclipse *Java Model*.
2. *Derived Plugins* – these are all the property and relation plugins that can be defined *on top* of the core plugins. For example, the *"Number of Methods (NOM)"* metric is a *derived property* plugin because it can be computed as the cardinality of the "*set of methods defined within a class*" (core) relation plugin.

The most important aspect about these two types of INCODE plugins is that the number of *core plugins* – which are dependent on the parsing infrastructure, and thus not reusable – is significantly lower than the rest of the plugins, which are built directly or indirectly on top of these (see Figure 4). Thus, by designing this system of plugins we were able to reuse the wealth of (derived) plugins (*i.e.,* most of the metrics and all detection strategies) defined in IPLASMA [MMM+05, LM06]. We could not reuse the core plugins from IPLASMA, as Recoder [LH00] was used in order to extract data.

4.3. INCODE *as a continuous code analyzer*

One of the biggest implementation challenges is to ensure that INCODE responds to every change in the program without affecting the overall performance of the Eclipse workbench. While Eclipse provides mechanisms for tracking changes, the actual source-code analysis that has to be done behind the scene (*e.g.,* metrics computation or clone detection) is *resource intensive*, both in terms of computing time and memory usage. Therefore, we took a series of design decisions without which INCODE would have never been more than a good idea. We are going to briefly enumerate these decisions here:

INCODE *always works on a separate thread.* In order to have continuously up-to-date results, we defined *listeners* on two types of events: (i) code changes in the current file, that lead (on save) to the launch of the *Java Builder*; (ii) changes of the current/selected file in the editor. When these events occur, INCODE starts a *marker job* by which the current file is analyzed. This job always runs on a new *separate* thread; this ensures that the user-interface is always responsive. Moreover, in order to avoid the danger of multiple *marker jobs* being simultaneously active, we use *concurrent data structures* and define a mechanism which cancels all running *marker jobs* on a file, whenever the editor's focus is switched to a different source file.

INCODE *works with lightweight model objects.* In INCODE all analyses assume a certain source-code model. In a static analysis environment, we would have employed the Eclipse Java Model (and ASTs) only as a fact extractor for populating our own model. But this is not feasible when INCODE has to react permanently to code changes. The solution to staying connected to the constantly updated Eclipse model, while still keeping the definition of analyses decoupled from the Eclipse API, is to *wrap Java Model entities* into model entities of the INCODE meta-model (see Wrapper class in Figure 4). The Wrapper class is the link between the iPlasma meta-meta model and Eclipse JDT. INCODE creates a Wrapper instance for every JDT model element instance (IProject, IPackage, IType, IMethod, IField). When the JDT model changes (usually during an incremental build, triggered by a modify-save operation), the incremental builder that INCODE defines gets notified of the changed JDT model elements and as a result INCODE can discard "dirty" wrappers that contain references to the now obsolete JDT model elements.

Minimizing the usage of Eclipse ASTs during model construction. Assessing design quality requires a detailed analysis of cross-references which occur at the method level, by means of method invocations and variables accesses; and this requires a complete parsing of the source-file, and visiting the resulting ASTs. Knowing that, in Eclipse, constructing an AST is extremely time consuming, we optimize the process by visiting each AST only once; even if the initial goal for creating an AST is to extract the calls and accesses for one method, INCODE *prefetches* the calls and accesses for all methods of that file, and after that we immediately destroy the AST object.

Intelligent caching. Many metrics are based on time-consuming cross-referencing information. Therefore, INCODE is designed to avoid any unnecessary (re-)computations, by means of *caching* and *selective update*. Thus, for each entity, the *core relations* (*e.g.,* group of calls) are cached, which allows properties (*e.g.,* metrics) to be recomputed very fast. When a change occurs, we identify the program entities that are affected by the respective change and consequently update only these entities, ensuring that no unnecessary computation is performed. Caching comes at a price: memory consumption, which can become an issue for large projects. To solve this problem, we engineered a solution that keeps references only to light-weight model elements from the JDT, while the heavy AST nodes (necessary only for local variables and parameters), are computed on demand and then disposed as soon as they are not needed. As a result, INCODE runs smoothly even on large Java projects with a minimal performance overhead.

4.4. Effort

INCODE is the result of more than 18 person months of effort, an estimate that does not include the initial learning curve of Eclipse and also iPlasma. While we could reuse a large part of the iPlasma framework (see 4.2), implementing the continuous assessment platform required a large amount of effort, resulting in a code base roughly the same size of the reused iPlasma code base (out of a total of approximately 38.000 lines of code, 18.000 represent the analysis framework). INCODE was developed in several iterations: (i) we started with dummy examples to see if the major challenges can be overcome, (ii) implemented a first running version and (iii) went through major refactorings a couple of times, in order to meet requirements of speed of execution and memory footprint, ease of use and smooth IDE integration.

5. Performance

So far we introduced the concept of continuous quality assessment and presented the key features of INCODE as well its general architecture. Next, we will evaluate INCODE and see, considering the significant number and complexity of the analyses used to asses a system's quality, if is it *feasible* to have INCODE being used by developers for *continuous* assessment *i.e.,* run INCODE and still perform in parallel "usual" development, without a negative performance impact?

Our measurements asses the execution time of opening a file in the editor. As described in the previous sections, opening a file in the editors triggers INCODE, which analyzes all the program elements defined in that file, in order to detect design problems. Note that although INCODE is focused only on assessing the program elements in the current file, the computations have system-wide implications; for example, checking for code duplication requires each method in the file to be compared with all the other methods in the entire system. Thus, we conducted these measurements as follows: we selected five well-known open source systems (ranging from average sized projects, of over 100.000 lines of code and large projects, that have over 1.4 MLOC), for each of the five systems we selected two source files based on their size; more exactly for each system we picked up the *largest* file (*Sample 1*) and a *random, average size* file (*Sample 2*). Using these files we measured the execution time of INCODE on three scenarios for each of the projects: (i) open *Sample 1* immediately after starting Eclipse; (ii) open *Sample 2* immediately after starting Eclipse; (iii) open *Sample 2* after *Sample 1 i.e.,* open a file *after* INCODE has already analyzed another file. This is the reason why we exercised the third scenario *i.e.,* measuring the execution time for the *second opened file*, as we expect the INCODE execution times on the subsequent files to be lower than the ones needed for the analysis of the first file.

System	Sample 1 (largest file)	Execution Time for Sample 1 with heap size = 2GB		Sample 2 (random file of average size)	Execution Time for Sample 2 (heap size = 2GB)		
		no duplication check	check duplication		no duplication check	check duplication	**opened after Sample 1 (check duplication)**
JHotDraw 7.4.1	137 KB	10 s.	18 s.	21 KB	3 s.	10 s.	1.2 s.
JEdit 4.3pre18	148 KB	10 s.	23 s.	6 KB	7 s.	12 s.	0.9 s.
ArgoUML 0.24	128 KB	9 s.	25 s.	4 KB	2 s.	17 s.	0.4 s.
Vuze 4511-23	151 KB	21 s.	99 s.	3 KB	2 s.	81 s.	0.1 s.
Eclipse - JDT 3.3.1	373 KB	16 s.	233 s.	4 KB	1 s.	212 s.	0.1 s.

Figure 5: Time performance when working with INCODE in continuous mode

By analyzing the numbers summarized in Figure 5 we can make the following observations [7]:

- The execution time dramatically decreases to a *marginal value* (*i.e.,* around a second per file) when a file is opened after another file has been analyzed. While the numbers in Figure 5 show the case where the average sized file is opened after the largest file, we made similar observations even when inverting the order of opening the files.
- When a file is first analyzed, a significant amount of time is used by checking for code duplication; and the time is substantially increasing when the project is large. This is expected considering the previous remarks. However, for the medium-sized projects the INCODE's "*first-opened*" execution time is below 30 seconds, even for the largest file and even when code duplication is checked. While 30 seconds may seem a long time, remember that INCODE runs on a separate thread (see Section 4.3), which means that the developer is never kept waiting by the INCODE thread. Developers can perform their work in parallel with INCODE. Thus, these execution times should be understood simply as the time in which INCODE works in parallel with the developer.

6. Related work

The work described in this paper is primarily related with other tools and approaches that are focused on quality assessment in general, and in particular with those that investigate how the design of object-oriented systems can be better assessed and improved.

Techniques for Quality Assessment. In the literature on object-oriented design we find many definitions of design rules and heuristics [Rie96, Mey97, Mar02], as well as a significant enumeration of design flaws [Rie96], usually called *bad smells* [Fow99] or *anti-patterns* [BMMM98]. Unfortunately, in almost all cases, these are defined informally, which makes their automatic detection hardly possible. As a result, a significant number of approaches have been proposed for detecting such design problems. Ciupke [Ciu99] proposes an approach in which the rules are specified in terms of queries, usually implemented as Prolog clauses, while Emder and Moonen propose an approach based on detecting structural regularities of code smells [vM02]. In this paper, we follow a metrics-based approach like the one that we defined in [Mar04] and which has been continued by Munro [Mun05] with a systematic description of smells and by Lungu [Lun04] with an attempt to sort potential instances of design problems, an approach which is related to the gravity score that we use in INCODE. More recently, Moha *et al.* proposed DECOR [NM10], a comprehensive framework for the specification design problems in form of combinations of metrics and other structural characteristics; based on DECOR the author generates a number of detection techniques for a set of specific design problems. While having efficient assessment techniques is essential, it is not sufficient from the point of view of quality improvement. Therefore, in the last years we notice a significant tendency towards approaches that bridge the gap between the detection and the correction of design problems, like JDeodorant [MF07, NT09].

Tools for Quality Assessment. When it comes to tools for quality assessment we have to distinguish between two categories: *standalone* QA tools and tools *integrated* in IDEs. There are a number of commercial standalone tools like *Klocwork Insight* [Klo10], *Structure 101* [Sof10], as well as dedicated open-source QA tools like *PMD* [Cop05] or *Checkstyle* [ECK10]. Furthermore there is also a set of source code analysis environments like *MOOSE* [DGN05], *Sonar* [Sou10] or *ConQAT* [FD06]. All of these are very mature, feature-rich tools which contain many innovative QA techniques. However, they are separated from the place where the code is produced, namely the IDEs. Thus, there is an unfortunate separation between quality assessment and the actual development, and consequently a separation between the people who write the software and those who assess its quality. As a result, the feedback loop between code and reviews is weak and inefficient, as developers learn about the various design problems only from time to time (*i.e.,* after each code review), and usually only long after the code has been written. This makes it very hard to solve *all* problems, because of the many problems that cumulate over time; and it's also hard to solve them efficiently because the context of that design fragment is probably long forgotten.

Because of this drawback, many of the aforementioned tools seek some form of integration with the development process, either by providing means of integration in the build process [Klo10, Sou10] and/or by integration in IDEs [Klo10] [Cop05]

[7] The entire evaluation, including all performance measurements, has been performed on a MacBook Pro, Intel Core 2 Duo processor at 2.2 GHz, and 4 GB RAM (@667 MHz)

[TCC08]. While the integration in the build process is definitely a step forward in the right direction – Sonar being a glorious [Sou10] illustration of this category – it still has the disadvantage that all problems are reported in a centralized manner and most of the times they lack providing the *exact context* of a design problem.

Concerning the quality assessment tools that are integrated in IDEs in general and in Eclipse in particular, the biggest problem is that their level of integration is very shallow. Most of these are *de facto* standalone tools that lack any synergy [Zel07] with the other components of the development environment. In almost all cases the analyses have to be triggered by the developer and thus do not run continuously during development. In Eclipse one notable exception is the *Checkstyle* plugin [ECK10] that detects code-style violations by using a project builder, which means that whenever the build process is started the files are analyzed by *Checkstyle* as well. This plugin has the same continuous analysis approach like INCODE, however, the problems that it detects are more related to coding style (*e.g.,* code conventions) than to object-oriented design. Other tools such as [EKKM08] use Eclipse's incremental builder in order to maintain and query an XML fact file database of cross-artifact information.

One more remark: all of today's major IDE's provide quality assurance modules that use metrics to control design quality [Met10, Ins10, Gmb10]. However, in the last years it became more and more clear that when metrics are used in isolation they are too fine grained to quantify comprehensively one investigated design aspect (*e.g.,* distribution of system's intelligence among classes). Thus, in most cases individual measurements do not provide relevant clues regarding the cause of a problem. In other words, a metric value may indicate an anomaly in the code but it leaves the engineer mostly clueless concerning the real cause of the anomaly. Consequently, in practice it is very hard to correlate an abnormal metric value with a concrete restructuring measure, that would improve the quality of the system's design.

7. Lessons learned

The main requirement of INCODE is that it must run in a timely matter and that it should not slow down the development process. We will detail a few of the lessons learned while trying to accomplish these goals and also how it could be further improved.

The biggest performance improvement was encountered when we decided to wrap Java Model elements, rather than an ASTNode, in an attempt to give up the memory hungry ASTs completely. This was not 100% achievable mainly because the Java Model does not have any cross-reference information. We still needed the AST for this type of information (called methods, return type), but we decided to cache all the information the parsed AST has to offer. Also, when constructing the AST of a method in a file, we extract the AST information for all the other methods of the file, as the AST was already built. After all the information is extracted from the AST, it is discarded in order to make room for the next file.

To further enhance performance, the authors would turn to Eclipse itself. Eclipse builds its own AST (for the java file currently open in the editor) that is not accessible via the API. Tool developers can not reuse this AST and must build their own. This is obviously suboptimal, especially considering the fact that a given Eclipse user could have any number of QA tools installed, and each tool has to build its own AST. Alternatively, the Java Model could provide type and type members cross-reference information, thus eliminating the need for expensive ASTs.

As an alternative solution to the Eclipse improvements mentioned earlier, in order to avoid rebuilding the ASTs, we could use aspect oriented programming to define a join point in Eclipse's internal AST builder, so that we would avoid an AST rebuild.

8. Conclusions and future work

In this paper we state that quality assessment must be transformed into a continuous, largely automated, process which comes closer to the daily development activities and implicitly closer to developers. In addition to periodical code reviews we need means to monitor *constantly* the evolution of design quality. In this context, we introduced INCODE as a means to automate such a continuous quality assessment process, which is integrated in the development life-cycle. We believe that INCODE will increase the productivity of programmers and the quality of their projects in two ways: (i) by warning developers about the occurrence of a design problem *as it appears*, which is far more efficient than the classic code inspections; and (ii) by providing *contextualized explanations* for each instance of a detected problem, instead of confronting programmers with just dry numbers (i.e., abnormal metrics values), which are hard to interpret in a way that leads to real code/design improvements.

While INCODE is currently detecting seven well-known design and architectural problems we have shown that it is built on a solid and flexible architecture that will allow us to rapidly increase the number of quality analyses in the coming months. Moreover, INCODE can be successfully used on very large projects of 1.4 MLOC, like Eclipse JDT.

At the same time, the continuous *detection* of design problems needs to be complemented by a continuous approach for *restructuring* the flawed design fragments. Currently, performing any code restructuring, except for the atomic refactorings, is still a manual, time-consuming and risky process, that involves extensive and costly human expertise. We plan to extend the correction component in INCODE by going beyond simple refactorings. This correction component is envisioned to act as a "wizard" that guides the developer through a contextualized correction plan that will be defined for each design flaw. An incipient form of a correction plan has been already defined in [Ver09] and [TM05] and implemented in INCODE (see correction step in Figure 1 but we need to significantly extend both the *number of contexts* where INCODE can perform a design improvement, as well as the number of performable restructurings that involve the correlation of multiple atomic refactorings.

Finally, probably the most important aspect of the future work, is to perform an extensive evaluation concerning the us-

age of INCODE and, more general, of employing a continuous quality assessment approach. We intend to perform in the close future a series of exploratory studies that would address the following research questions: (i) Does a continuous quality assessment approach improve the design quality of software, over traditional code reviews? (ii) Does the systematic usage of INCODE lead to less new design problems (as a result of improving the design skills of developers)? (iii) How does the usage of INCODE affect programmers' productivity?

References

[BBC+10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[BG01] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the OMG/MDA framework. In *Proceedings of Automated Software Engineering (ASE'01)*, pages 273–282, Los Alamitos CA, 2001. IEEE Computer Society.

[BMMM98] William J. Brown, Raphael C. Malveau, Hays W. McCormick, III, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley Press, 1998.

[Ciu99] Oliver Ciupke. Automatic detection of design problems in object-oriented reengineering. In *Proceedings of TOOLS 30 (USA)*, pages 18–32, 1999.

[Cop05] Tom Copeland. *PMD Applied*. Number 0-9762214-1-1. Centennial Books, 2005.

[DGN05] Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Moose: an agile reengineering environment. In *Proceedings of ESEC/FSE 2005*, pages 99–102, September 2005. Tool demo.

[DTD01] Serge Demeyer, Sander Tichelaar, and Stéphane Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

[ECK10] The Checkstyle Plugin for Eclipse. http://eclipse-cs.sourceforge.net/, 2010.

[EKKM08] Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. Defining and continuous checking of structural program dependencies. In *Proceedings of the 30th international conference on Software engineering*, pages 391–400. ACM, 2008.

[FD06] T. Seifert F. Deissenboeck, M. Pizka. Tool support for continuous quality assessment. In IEEE Computer Society Press, editor, *13th IEEE International Workshop on Software Technology and Engineering Practice*, volume 076952639X, pages 127–136, 2006.

[Fow99] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[Gmb10] Odysseus Software GmbH. Stan - structure analysis for java. http://stan4j.com/, 2010.

[Ins10] Instantiations. CodePro analytix. http://www2.instantiations.com/codepro/analytix, 2010.

[Kem92] Chris F. Kemerer. Now the learning curve affects case tool adoption. *Software, IEEE*, 9(3):23–28, 1992.

[Klo10] Klocwork. Klocwork insight. http://www.klocwork.com/products/insight, 2010.

[LH00] Andreas Ludwig and Dirk Heuzeroth. Metaprogramming in the large. In *Jarzabek (Eds.), Generative and Component-Based Software Engineering. Lecture Notes in Computer Science*, pages 443–452. Springer, 2000.

[LM06] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[Lun04] Mircea Lungu. Conformity strategies: Measures of software design rules. Master's thesis, Politehnica University of Timisoara, September 2004.

[Mar02] Robert Cecil Martin. *Agile Software Development. Principles, Patterns, and Practices*. Prentice-Hall, 2002.

[Mar04] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 350–359, Los Alamitos CA, 2004. IEEE Computer Society Press.

[Mar12] R. Marinescu. Assessing technical debt by identifying design flaws in software systems. *IBM Journal of Research and Development*, 56(5), 2012.

[Met10] Eclipse metrics plugin. http://sourceforge.net/projects/metrics/, 2010.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.

[MF07] A. Chatzigeorgiou M. Fokaefs, N. Tsantalis. Jdeodorant: Identification and removal of feature envy bad smells. In IEEE Computer Society Press, editor, *23rd IEEE International Conference on Software Maintenance (ICSM'2007)*, pages 519–520. IEEE Computer Society, IEEE Computer Society Press, September 2007.

[MMM+05] Cristina Marinescu, Radu Marinescu, Petru Mihancea, Daniel Ratiu, and Richard Wettel. iPlasma:an integrated platform for quality assessment of object-oriented design. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005)*, pages 77–80, 2005. Tool demo.

[Mun05] M.J. Munro. Product metrics for automatic identification of "bad smell" design problems in java source-code. In IEEE Computer Society Press, editor, *Proceedings of the 11th International Software Metrics Symposium*, September 2005.

[NDG05] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of moose: an agile reengineering environment. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 1–10, New York, NY, USA, 2005. ACM.

[NM10] Laurence Duchien Anne-Françoise Le Meur Naouel Moha, Yann-Gaël Guéhéneuc. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, January 2010.

[NT09] Alexander Chatzigeorgiou Nikolaos Tsantalis. Identification of move method refactoring opportunities. *IEEE Transactions On Software Engineering*, 35(3):347–367, May/June 2009.

[Rie96] Arthur Riel. *Object-Oriented Design Heuristics*. Addison Wesley, Boston MA, 1996.

[Sof10] Headway Software. Structure 101. http://www.headwaysoftware.com, 2010.

[Sou10] Sonar Source. Sonar. http://www.sonarsource.org/, 2010.

[TCC08] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. Jdeodorant: Identification and removal of type-checking bad smells. In *Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering*, pages 329–331, Washington, DC, USA, 2008. IEEE Computer Society.

[TM05] Adrian Trifu and Radu Marinescu. Diagnosing design problems in object oriented systems. In *Proceedings of 12th Working Conference on Reverse Engineering (WCRE 2005), 7-11 November 2005, Pittsburgh, PA, USA*, pages 155–164, Los Alamitos CA, 2005. IEEE Computer Society.

[Tri08] Adrian Trifu. *Towards Automated Restructuring of Object Oriented Systems*. PhD thesis, Universität Karlsruhe, 2008.

[TVS10] Alexandru Telea, Lucian Voinea, and Hans Sassenburg. Visual tools for software architecture understanding: A stakeholder perspective. *Software, IEEE*, 27(6):46–53, 2010.

[Ver09] Ioana Verebi. Automation of complex design restructurings. Master's thesis, Politehnica University of Timisoara, 2009.

[vM02] Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Proc. 9th Working Conf. Reverse Engineering*, pages 97–107. IEEE Computer Society Press, October 2002.

[Zel07] Andreas Zeller. The future of programming environments: Integration, synergy, and assistance. In *2007 Future of Software Engineering*, FOSE '07, pages 316–325, Washington, DC, USA, 2007. IEEE Computer Society.