Assessing technical debt by identifying design flaws in software systems

R. Marinescu

Tough time-to-market constraints and unanticipated integration or evolution issues lead to design tradeoffs that usually cause flaws in the structure of a software system. Thus, maintenance costs grow significantly. The impact of these design decisions, which provide short-term benefits at the expense of the system's design integrity, is usually referred to as technical debt. In this paper, we propose a novel framework for assessing technical debt using a technique for detecting design flaws, i.e., specific violations of well-established design principles and rules. To make the framework comprehensive and balanced, it is built on top of a set of metrics-based detection rules for well-known design flaws that cover all of the major aspects of design such as coupling, complexity, and encapsulation. We demonstrate the effectiveness of the framework by assessing the evolution of technical debt symptoms over a total of 63 releases of two popular Eclipse projects. The case study shows how the framework can detect debt symptoms and past refactoring actions. The experiment also reveals that in the absence of such a framework, restructuring actions are not always coherent and systematic, not even when performed by very experienced developers.

Introduction

Software systems must evolve continuously to cope with requirements and environments that are always changing [1]. Stringent time to market constraints and fast emerging business opportunities require swift, but profound, changes of the original system. Therefore, in most software projects, the focus is on immediate completion and on choosing a design or development approach that is effective in the short term, even at the price of increased complexity, and higher overall development cost in the long term [2]. Cunningham introduced the term *technical debt* [3] to describe this widespread phenomenon. According to this metaphor, *interest payments* represent the extra maintenance effort that will be required in the future, due to the hasty, inappropriate design choices that are made today, and *paying the principal* means the effort required to restructure the part of the systems that were improperly designed in the past. As in financial debt, there are two options: continuously paying the interest or paying down the principal, by refactoring the design affected by flaws into a better one and consequently gaining through reduced future interest payments [4].

When debt is incurred, at first, there is a sense of rapid feature delivery, but later, integration, testing, and "bug" (i.e., flaw) fixing become unpredictable, to the point where the effort of adding features becomes so high that it exceeds the cost of writing the system from the ground up [5]. The cause of this unfortunate situation is usually less visible: The internal quality of the system design is declining [6], and duplicated code, overly complex methods, noncohesive classes, and long parameter lists are just a few signs of this decline [7]. These issues, and many others, are usually the symptoms of higher-level design problems, which are usually known as design flaws [8], design smells [7], or anti-patterns [9].

Technical debt is not always bad. Often, the decision to incur debt is the correct one, especially when considering nontechnical constraints [10], but most of the time, incurring debt has a negative impact on the quality of a system's design. In this paper, we propose a framework for assessing technical debt by exposing debt symptoms at the design level. Our framework is based on detecting a set of relevant

Digital Object Identifier: 10.1147/JRD.2012.2204512

design flaws and measuring the negative impact of each detected flaw instance on the overall quality of the system design.

Related work

The literature proposes various approaches [11–15] to detect design flaws in object-oriented systems. Although these techniques are valuable for enabling the identification of individual problems, to our knowledge, there are no publications proposing a framework to support the overall assessment of technical debt symptoms by aggregating the results of individual design flaw-detection techniques. The idea of aggregating basic quality indicators into a quality model is not new. The literature proposes various models [16–19], all based on the Factor-Criteria-Metric (FCM) de-compositional approach introduced by McCall and Boehm [1]. An FCM model aims to "operationalize" a quality goal by reducing it to several factors (e.g., maintainability and portability), but these factors are too abstract and need to be expressed by a layer of criteria (e.g., coupling and complexity) that are eventually measured by metrics.

Although FCM models are widely used, they have at least one significant limitation: These models are unable to directly expose debt symptoms because their basic indicators are fine-grained code metrics, which when analyzed in isolation can hardly reveal design flaws [20]. As a consequence, corrective actions (i.e., restructurings) are hampered.

Another noteworthy approach is proposed in the context of the SONAR quality management platform [21], where several plug-ins [19–23] claim to measure technical debt, all based on the idea that indicators of internal quality can be converted to system-level measurements of effort and money. Estimating the effort or financial costs associated with technical debt is certainly a desirable goal; however, confirming Fowler's criticism [4], all of these approaches show two major deficiencies: 1) they do not provide any empirical evidence on how well these estimations reflect the actual effort needed to remove technical debt; and 2) they rely on the unrealistic assumption that the effort associated with basic restructuring actions is proportional with a single code metric; for example, it is hardly possible to infer the "cost to split a method" [22] solely based on the cyclomatic complexity (or conditional complexity) of a method. The approach presented in this paper overcomes the limitations of the aforementioned approaches, as it raises the abstraction level by building on top of detected instances of design flaws instead of metrics. As a result, the framework provides additional benefits in three ways:

1. *Assessment*—The framework helps assess the status of a system's design by identifying the flaws that affect the system and by measuring their negative impact on the overall design. Such an assessment is particularly relevant when a company acquires a software system from another company [10]. In such cases, it is essential to identify signs of a fragile design structure, which are usually the result of technical debt accumulated over time by a team that was unwilling or incapable of paying down the "principal," i.e., restructuring the design.
2. *Monitoring*—Whereas in financial debt, one knows from the beginning the debt value, when making a development decision, it is hardly possible to know its actual impact beforehand, even when knowing that the decision will incur technical debt. By quantifying debt symptoms, this framework makes it possible to continuously monitor the evolution of the system during a period when such a critical design decision is implemented.
3. *Restructuring*—Technical debt is reduced when a team decides to refactor the design [4] instead of paying the additional development effort of maintaining a flawed design fragment. By revealing the type, location, and severity of design flaws, this assessment framework allows a team to prioritize refactoring and to refactor systematically.

The remainder of the paper is organized as follows. First, we define the actual assessment framework and discuss the various instantiation details. Then we describe the two analyzed projects, the instruments, and the analysis process, and we present and discuss the results. We conclude by summarizing the advantages of the approach and several future developments.

Framework for assessing technical debt symptoms

Building the assessment framework involves four steps: 1) select a set of relevant design flaws, 2) define rules for the detection of each design flaw, 3) measure the negative influence of each detected flaw instance, and, eventually, 4) compute an overall score that summarizes the design quality status of a system by taking into account the influence of all detected flaws. For each of these steps, we distinguish in our description between the conceptual side and the decisions made for the actual implementation with their impact on the case study (see the section "Case study").

Select relevant design flaws

It is impossible to establish an objective and general set of rules that would automatically lead to high-quality design. However, there is a common understanding of some general, high-level characteristics of a good design [1]; notably, Coad and Yourdon [24] identify four essential traits of a good object-oriented design: low coupling, high cohesion, moderate complexity, and proper encapsulation. Consequently, many authors have formulated design principles, rules, and heuristics [25–27] that would help

Table 1 Informal description of the design flaws used to instantiate the assessment framework.

Design flaw	Description
God Class	An excessively complex class that breaks encapsulation by directly using data from other classes.
Schizophrenic Class	A class that has a very low cohesion, as it defines a large interface that is used by disjoint groups of clients.
Refused Parent Bequest	A subclass that shows a weak connection to its superclass, by insufficiently using the hierarchy-specific methods and data inherited from the superclass.
Data Class	A class that does not encapsulate its data and usually does not provide significant functionality, allowing other classes to use its data directly.
Code Duplication	A method that has a significant number of code lines duplicated with at least another method, from a different class.
Brain Method	A method that is overly long, with statements showing a deep nesting level and an over-complex branching structure.
Data Clumps	A long parameter list, which appears over and over again in many other methods throughout the system.
Intensive Coupling	A method that calls an excessive number of other method from one or several other classes.

developers design better object-oriented systems. In addition, the software refactoring community [7, 9] describes design flaws as a means to highlight frequently encountered situations where the aforementioned design principles and rules are violated.

The framework can be used with any type of design flaw, ranging from fine-grained ones that affect methods (e.g., code duplication) to architectural flaws that occur at the level of subsystems (e.g., cyclic dependencies [27]). The mixture of flaws that are included in an actual framework instantiation should reflect the assessment focus.

Implementation decisions
For the concrete instantiation of the framework, we selected eight design flaws (**Table 1**) based on the following characteristics:

1. *Significant*—We include only design flaws that are well described in the literature and for which empirical studies or experience reports (e.g., [28, 29]) indicate a high occurrence rate and a significant negative impact on the perceived quality of the analyzed systems (e.g., high correlation with flaws).
2. *Balanced*—We select a minimal set of design flaws that cover in a balanced manner the aforementioned traits of good design identified by Coad and Yourdon [24].
3. *Accurate*—We use only design flaws that proved in our earlier studies [8, 12] to be automatically detectable with a high level of accuracy (i.e., good precision and recall).

Detect design flaws
In order to detect design flaws, various detection techniques can be used. Some of these techniques are based on Prolog* rules to describe structural anomalies [11], whereas others are metrics-based [12, 14, 15] or use a dedicated specification language [13] that also allows taking into account correlation among flaws. We build on our previous experience of detecting design flaws using a metrics-based approach called

detection strategy [8, 12], in which design flaws are expressed as composite logical conditions based on metrics, where each such condition captures a significant symptom of a flaw. The detection strategy is used to identify those design fragments (e.g., classes and methods) that satisfy the condition.

Starting from the informal description of design flaws [7, 9, 27], we have defined a large set of detection strategies [8, 12, 15]. Each detection strategy was defined using the Goal-Question-Metric (GQM) methodology [30], which defines a three-level quantification model. The first, i.e., the conceptual level, defines the measurement goal, which in our case is to detect the presence of a particular design flaw. On the second level, the goal is refined in the form of a set of questions, which in this particular case capture the specific traits of a design flaw. Finally, each question is associated with one or more metrics that answer the question in a measurable way.

Example—Riel informally describes the God Class design flaw as follows [27]: *Top-level classes in a design should share work uniformly.* [...] *Beware of classes that access directly data from other classes.* [...] *Beware of classes with much noncommunicative behavior.*

First, we identify the symptoms that appear in the description of the design flaw, which include the following: excessive class complexity, direct access to instance variables defined in other classes, and low class cohesion. Each of the three symptoms can be measured using a software metric. We use Weighted Method Count (WMC) to measure complexity, Access To Foreign Data (ATFD) to quantify the usage of external instance variables, and Tight Class Cohesion (TCC) to measure the internal cohesion of a class. Consequently, the detection strategy for God Class can be expressed as follows [12]:

$$\texttt{God Class} = (\texttt{WMC} > \texttt{VERY\_HIGH}) \\ \texttt{and}\ (\texttt{ATFD} > \texttt{FEW})\ \texttt{and}\ (\texttt{TCC} < \texttt{LOW}).$$

Implementation decisions
This detection rule shows that the most sensitive part of any metrics-based technique is the selection of concrete threshold values. This issue is beyond the scope of this paper, but details are discussed extensively in [12, 15]. Likewise, a full description of the eight detection rules is available in the aforementioned publications.

Measure the impact of design flaws
Each design flaw affects to some extent the overall quality of a system, but not all have the same negative impact. Thus, in order to increase the accuracy of the assessment framework, the negative impact of each design flaw instance has to be quantified. We take into account the following three factors:

1. *Influence* ($I_{\text{flaw\_type}}$)—This factor expresses how strongly a type of design flaw affects the criteria of good design. We rely on Coad and Yourdon's four criteria of good design [24] (i.e., low coupling, high cohesion, moderate complexity, and proper encapsulation) and propose a three-level scale (high, medium, low) to characterize the negative influence of a design flaw on each of the four criteria. The actual $I_{\text{flaw\_type}}$ values are the result of assigning numerical values to each of the three levels and computing a weighted arithmetic mean between the four criteria, where each weight represents the relative importance of a design criterion in a given assessment scenario.
2. *Granularity* ($G_{\text{flaw\_type}}$)—In general, a flaw that affects methods has a smaller impact on the overall quality than one that affects classes; consequently, we assign a weight to each design flaw according to the type of design entities (e.g., class and method) that it affects.
3. *Severity* ($S_{\text{flaw\_instance}}$)—The first two factors refer to design flaw types, which means that all instances of the same flaw weigh equally; however, because not all cases are equal, we define for each flaw a severity score based on the most critical symptom of the flaw, measured by one or more metrics. To allow comparisons among design flaws, severity scores have a lower limit of 1 (low) and an upper limit of 10 (high).

Based on the three factors, we compute the *Flaw Impact Score* (*FIS*) of a design flaw instance as follows:

$$FIS_{\text{flaw\_instance}} = I_{\text{flaw\_type}} \times G_{\text{flaw\_type}} \times S_{\text{flaw\_instance.}}$$

Implementation decisions
Assessing quality is far from being an objective matter that can be nailed down to a generally applicable formula. Therefore, in instantiating the framework for assessing the case studies (see the section "Case study"), we made several decisions for each of the three factors that determine the *FIS* value:

1. *Influence*—The actual mapping between the eight design flaws and criteria was performed as follows: we asked seven senior software designers to assign one of the three influence levels for each flaw-criteria pair, and we chose the most common answer (which in all cases was selected by at least five designers). The results are summarized in **Table 2**. In assigning the numerical values to the three-level influence scale, we used a geometric sequence with ratio 2 (high = 2, medium = 1, and low = 0.5). By experimenting in the past with various

Table 2 Overview of the factors that influence the impact of each design flaw.

Design flaw	Influence				Granularity	Severity
	Coupling	Cohesion	Complexity	Encapsulation		
God Class	Medium	Medium	Medium	High	Class	Number of data members used from other classes
Schizophrenic Class	Medium	High	Medium	Low	Class	Number of disjoint groups of clients using the interface of the class
Refused Parent Bequest	High	Low	Medium	Low	Class	Ratio of inheritance-specific members used from the superclass
Data Class	Low	Medium	Low	High	Class	Number of non-encapsulated data members and the number of other classes using the data.
Code Duplication	Low	Medium	High	Low	Method	Length of duplicated code and number of operations sharing that code
Brain Method	Medium	Medium	High	Medium	Method	Operation length and nesting level of statements
Data Clumps	Medium	Low	Medium	High	Method	Number of methods where the repeated parameters appear
Intensive Coupling	High	Medium	Low	Low	Method	Number of methods called from a single class

assignment schemas on several known systems, we learned that this geometric sequence provides the most accurate classification of the systems, compared with the perception of human experts. Furthermore, using the ratio 2 ensures a well-balanced assessment matrix (Table 2) with respect to the four criteria of good design. In the assessment scenario, we considered each of the four criteria to be equally important, and therefore, $I_{\text{flaw\_type}}$ was computed as a simple arithmetic mean. For example, for the God Class flaw $I_{\text{flaw\_type}}$ is 1.25, i.e., $(1+1+1+2)/4$.

2. *Granularity*—Based on our extensive previous experience in assessing object-oriented systems [12], we considered a Class flaw to be three times more influential than a Method flaw.
3. *Severity*—The severity scores are computed on the basis of the metrics presented in Table 2 and using the implementation available in inFusion [31], the assessment tool used to perform the case study. Severities are computed by measuring how many times the value of a chosen metric exceeds a given threshold. For example, in God Class, the severity is based on the ATFD metric (see the section "Detect design flaws"), which has threshold 4. Considering two God Classes `Foo`$(\text{ATFD} = 20)$ and `Bar`$(\text{ATFD} = 50)$, the severity of Foo is 5, i.e., 20/4, but because of the upper bound of severity scores, the severity of `Bar` is 10 and not 12.5. We limit the severity score to an upper bound in order to avoid skewing the overall value, because of extreme outliers. In the future, we plan to refine the formulas for computing the severity of a design flaw instance by taking into account information about the evolution of the flawed entity; for example, a God Class that has barely changed over time may be considered less harmful than another one that has changed substantially in each release [32].

Compute the overall score

In order to get a quantifiable overview of design debt in a system, the various instances of flaws must be aggregated.

Table 3 Size characteristics of the two systems used in the case study.

System	No. of Releases	Releases					
		First			Last		
		Number	Year	Lines of code	Number	Year	Lines of code
Java Development Tools	31	1.0.0	2001	272,539	3.7.1	2011	1,187,252
Eclipse Modeling Framework	32	1.0.2	2003	127,888	2.7.1	2011	389,439

For this purpose, we define the Debt Symptoms Index (DSI) as follows:

$$DSI = \frac{\sum_{\text{all\_flaw\_instances}} FIS_{\text{flaw\_instance}}}{KLOC},$$

where $KLOC$ represents the number of thousands of lines of code of the system. Normalizing the DSI relatively to the code size of systems makes its values comparable among systems of different sizes.

Case study

In this section, we describe using the framework to assess the design debt in two Eclipse* projects. The goal of the case study was to demonstrate how the proposed framework can track the evolution of design debt over time and how it can expose the presence of debt symptoms in the source code.

Projects and instruments

We have applied the assessment framework to two of the oldest and most popular Eclipse projects: the Java* Development Toolkit (JDT) and the Eclipse Modeling Framework (EMF). The JDT project defines a full-featured Java Integrated Development Environment that provides a large number of views, editors, wizards, and refactoring tools [33]. The EMF project is a modeling framework and code-generation facility for building applications on the basis of a structured data model [34]. As summarized in **Table 3**, for each project, we analyzed more than 30 public releases, following their evolution, from the earliest versions (8 to 10 years ago) until the end of 2011. During this period, the source code of both projects grew significantly in size: In JDT, there is now four times more code than in the initial version, whereas in EMF, there is three times more code.

We selected the two projects because they fulfill several criteria, which are particularly important for the goals of this case study:

1. *Popularity*—A large userbase forces a project to constantly adapt in order to address the new, complex, and sometimes conflicting requirements, which inevitably lead to technical debt.
2. *Maturity*—A long history of releases ensures that the evolution of technical debt symptoms can be followed over an extensive period; it is only in the long run that one can notice significant variations of debt because of changing conditions.
3. *Open*—Open-source projects ensure that the presented case study is reproducible and may be enlarged in the future.
4. *Proficiency*—Experienced developers know the principles of software design very well and are therefore aware of the consequences of breaking them. This allowed us to reveal how such developers address the symptoms of increasing debt, even in the absence of a systematic approach such as the one described in this paper.

Instruments

The case study was performed as follows: All 63 releases were downloaded from the official website of the two projects [33, 34]. For each release, the source code, together with the depending libraries, was extracted and analyzed using inFusion [31], a quality-assessment tool that automatically detects a large set of design flaws. For this case study, we extended inFusion to compute the DSI according to the specifications described in the section "Framework for assessing technical debt symptoms." In addition, inFusion was extended with the capability of comparing two consecutive releases and identifying all the changes that occurred, in terms of number of instances and severities of detected design flaws. This feature, together with inFusion's built-in source code exploration features enabled us to perform a detailed analysis of the evolution patterns for each design flaw, as described in the section "Evolution of design flaws."

Overview of DSI evolution

Figure 1 shows how the DSI changed over time in each of the two systems. In order to put the DSI values into perspective, the figure also shows the evolution of source code size. Based on this information, we discuss two interesting aspects: the variations of DSI over time and the relation between code growth and design debt.

DSI variations

JDT starts (release 1.0) with an extremely high DSI value, but in release 2.0, there is a massive improvement,

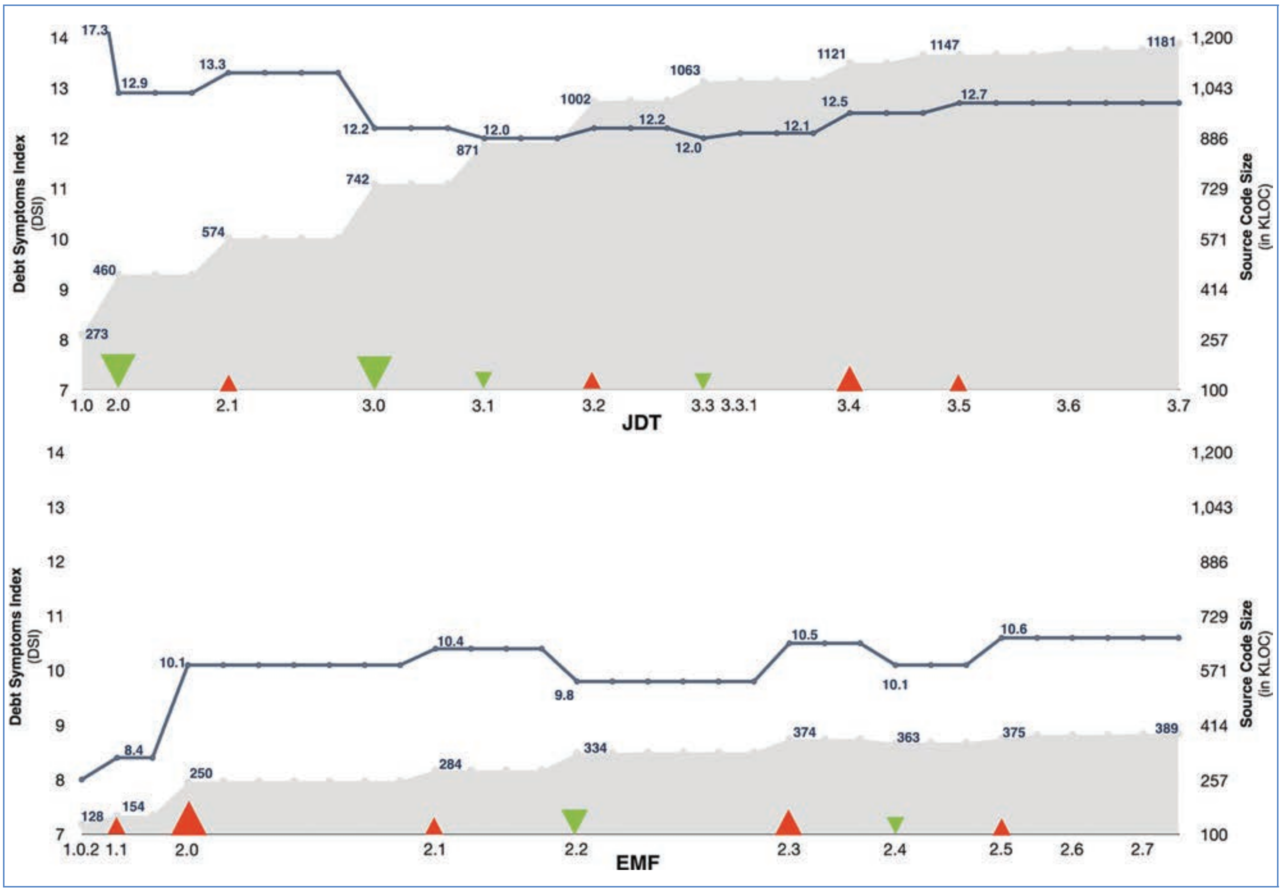


Figure 1

Evolution of code size and design debt in JDT and EMF. The blue line plots the evolution of *DSI*, whereas the gray area shows the source code growth over time. The triangles indicate versions where a significant debt decrease (green) or increase (red) was recorded.

as the *DSI* is reduced by 25%. Prior to release 3.3, design quality improves substantially and steadily. During this period, there are only two small and temporary setbacks, which occurred in releases 2.1 and 3.2. After 3.3.1, the next two major releases show consecutive increases of debt symptoms. Finally, after release 3.5, the design enters a very stable phase, with almost no variations of *DSI*.

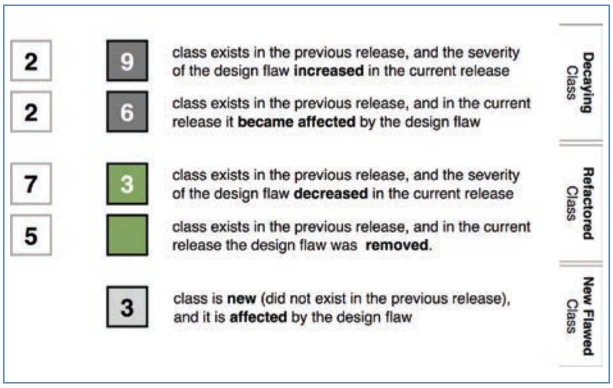
EMF has a very different story. It starts with a lower *DSI* value, but each of the first three major releases adds significant debt. In total, an extra 30% of *DSI* was added. After release 2.5, we observe a standstill in the evolution of design debt.

To put the DSI values of JDT and EMF into perspective, it is worth mentioning that in a different, not yet published, study, we measured the *DSI* in 140 popular open-source Java systems. Eighty percent of these systems had *DSI* values between 7 and 17, with a median value of 10.5.

DSI and code growth

Prior to JDT release 3.3, there were several releases where the code grew abruptly, with each release increasing the size of the code base by at least 25%. Surprisingly, in JDT 2.0 and 3.0, where the code size grew by 68% and 29%, respectively, the design quality appeared to improve substantially. In addition, the only major code growth associated with a decline of design quality is recorded in JDT release 2.1.

EMF shows again a different evolution. From the first release, until release 2.2, the code base almost triples in size, but it seems that functionality was added while neglecting the impact on design quality. A potential reason could be that EMF started with a low *DSI* value, so design debt was probably of no concern in the early releases. EMF 2.2 is the first release where the *DSI* is reduced. From there, up to EMF 2.5, we notice an alternation of debt-reducing releases (2.2 and 2.4) and debt-increasing releases (2.3 and 2.5). After release 2.5, the system enters



2
9
class exists in the previous release, and the severity of the design flaw **increased** in the current release
2
6
class exists in the previous release, and in the current release it **became affected** by the design flaw
Decaying Class
7
3
class exists in the previous release, and the severity of the design flaw **decreased** in the current release
5
class exists in the previous release, and in the current release the design flaw was **removed**.
Refactored Class
3
class is **new** (did not exist in the previous release), and it is **affected** by the design flaw
New Flawed Class

Figure 2

The various ways in which the state of a design flaw can change between two releases of a system. Squares represent classes, and the numbers contained within the squares represent the *FIS* value for a design flaw.

a very stable phase, where in spite of a slight code growth, there is no change in the *DSI*.

Interpreting a change of the *DSI* value between two releases, in the context of a significant code growth, is not trivial. For example, a lower *DSI* value is not necessarily the result of restructuring actions performed on the existing code; it may be equally the result of better-designed code being added. Therefore, in order to understand what caused the change of the *DSI* value in the evolution of the two systems, we performed a fine-grained analysis, as described below.

Evolution of design flaws

A change of the *DSI* value, between two releases, is determined by the design quality of the newly added code and by the modifications performed on the existing code, some of which reduce debt, whereas others increase it. Therefore, we analyzed each pair of consecutive releases where a significant change of the *DSI* was recorded, by looking for changes in the existence and severity of design flaw instances.

Observing the evolution of an index is generally useful for better understanding a measured system; however, we went further and refined this evolution analysis: For each type of design flaw, we compared the classes in the current release with those in the previous release and distinguished three types of classes (**Figure 2**):

1. *Decaying Class*—a class that exists in both releases and which, in the current release, either becomes affected by the flaw or has an increased flaw severity.
2. *Refactored Class*—a class that has been affected by the flaw in the previous release and which, in the current release, is either not affected anymore by that flaw or has a reduced flaw severity. This is a very important case, because it indicates that developers are aware of the debt caused by that particular flaw and have considered it worthwhile to change it in order to improve its design quality.
3. *New Flawed Class*—a newly introduced class that is affected by the flaw from the start.

For each type of design flaw, and for each type of change (i.e., decaying, refactored, and newly flawed), we compute the *Delta DSI* (*DDSI*) by adding all the differences for the *FIS* between the two releases (current and previous):

$$DDSI = \sum_{\text{instances}} \left| \frac{FIS_{\text{current\_instance}}}{KLOC_{\text{current}}} - \frac{FIS_{\text{previous\_instance}}}{KLOC_{\text{previous}}} \right|.$$

In order to match the identity of the class in the two releases, we compare the fully qualified names of classes. In addition, in order to capture the cases where a class has been moved from one package to another, we use a pattern-based matching algorithm [35].

We deliberately ignored in our analysis classes that have been deleted from the previous to the current release. The deletion of a flawed class can be interpreted as a refactoring, but deletion can also have different reasons (e.g., extract some classes to a new project). Therefore, we decided to take a conservative approach and count as refactorings only classes that exist in both versions.

Occurrence of design flaws

The results of applying the aforementioned procedure on the two case studies is summarized in **Figure 3** using one evolution chart for each of the eight design flaws. A chart depicts the evolution of *DDSI* for each of the three types of changes: decaying (dark gray area), newly flawed (light gray area), and refactored classes (green line).

In order to understand the occurrence of flaws, we examine the gray areas in Figure 3 and notice that the overall impact of various design flaws, as well as their evolution patterns, is very different. Schizophrenic Class, Data Class, and Intensive Coupling have a very minor influence in both JDT and EMF. The presence of these flaws occurs mostly in the first releases and is probably due to classes that have not yet reached a stable state.

In JDT, the newly added classes have a constant and significant *DDSI* value for the God Class flaw. In addition, previously existing God Classes tend to become gradually worse. In EMF, God Class shows a different pattern: In the second release, there is a moderate growth, but beginning with release 2.3, almost no new instances were detected.

The impact on the two projects is reversed when it comes to Refused Parent Bequest; in EMF, the flaw occurs massively in the new classes of several early releases, whereas the problem seems less present in JDT. However, an interesting observation applies to both systems. In all cases,

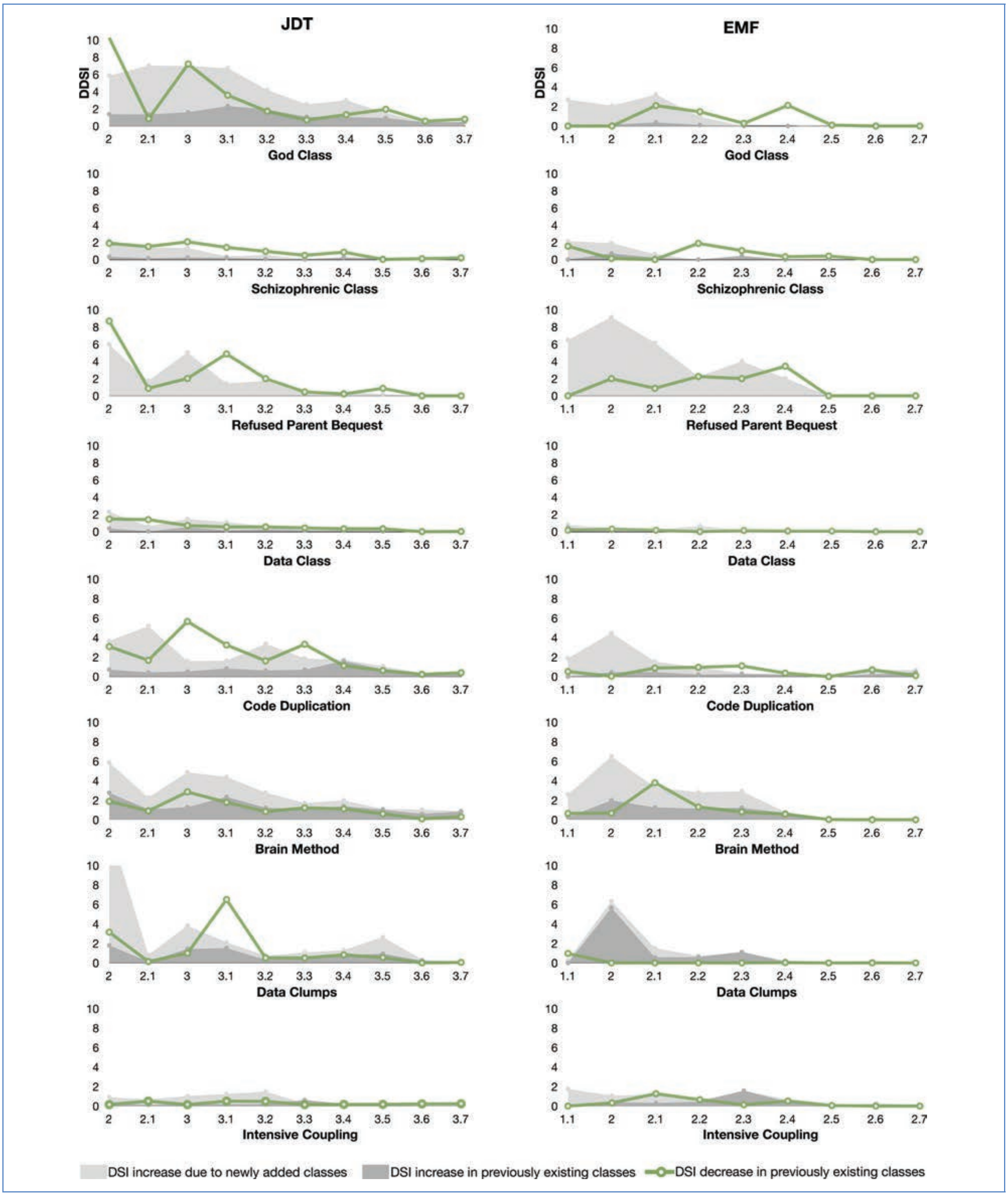


Figure 3

Evolution of *Delta DSI* for each type of design flaw, broken down by the three types of affected classes: decaying (dark gray), newly flawed (light gray), and refactored (green).

this problem occurs when a class is first created, and there are no cases where the flaw appears or becomes more severe for an existing class in a later release. A possible explanation, confirmed by a partial manual investigation, is that subclasses are not yet completely implemented when they are introduced in the hierarchy; in the beginning, they have a weaker coupling to their base class, which becomes stronger in subsequent releases, when the derived class reaches its complete implementation.

Another flaw that occurs differently in the two systems is Data Clumps; in JDT 2.0, this flaw appears massively in new classes. Unlike in JDT, in EMF, most of the Data Clumps instances occur almost exclusively in classes that have existed before that release.

Brain Method is a flaw that significantly affects both systems similarly. It occurs in new and in previously existing classes, where new overly complex methods tend to appear or existing Brain Methods tend to get worse.

Code Duplication occurs moderately in both systems. Among the two systems, JDT is certainly more affected, because new cases appear in almost all releases, both in new classes, especially in 2.1 and 3.2, but also in previously existing ones, especially in 3.4. In EMF, the flaw is present mainly in the new classes of early releases.

Refactoring actions

In the final part of this study, we looked for signs of refactoring actions, aimed to reduce the symptoms of technical debt. In performing this analysis, we considered the following three aspects:

1. *Diligence*—How intense are the efforts of improving classes affected by design flaws?
2. *Reactivity*—Are major refactoring actions correlated with an increase of debt symptoms in the previous releases?
3. *Coherence*—Are there cases where, for the same flaw type, we see both a decrease of *DSI* (due to refactoring actions in some classes) and an increase of *DSI* (due to new flawed classes and/or existing classes with an increased severity)?

In general, the diligence of refactoring actions is substantial and persistent over time for all those design flaws that significantly affect the systems. As one noteworthy exception, we did not detect any corrective actions for Data Clumps in EMF, despite their significant occurrence in release 2.0. This is a sign that EMF developers did not consider this to be a problem, whereas in JDT, Data Clumps were substantially refactored in release 3.1.

Concerning reactivity, Figure 3 shows that the main refactoring actions for a design flaw occur when one or two previous releases exhibit a significant increase of *DSI* related to that flaw. In some cases, the improvement actions are observed in the immediately following release; for example, Code Duplication in JDT 3.0 and Brain Method in EMF 2.1. In many other cases, a significant refactoring action is noticeable only after several releases. The best example here is the reduction of debt symptoms caused by God Class in EMF 2.4; a significant refactoring occurred despite that almost no new such classes were detected in EMF 2.2 and 2.3. Thus, the refactoring was focused mainly on God Classes introduced in the first three major releases (1.1 to 2.1).

Another noteworthy refactoring case concerns Refused Parent Bequest in EMF. After the massive debt symptoms recorded in 2.0 and 2.1, beginning with release 2.2, there is a steady improvement effort. By analyzing the evolution of several sample classes, we found the following explanation: Between releases 2.2 and 2.4, there is a stepwise refinement of class hierarchies that leads to a stronger connection between base and derived classes.

Refactoring does not always indicate a coherent attempt at tackling a design flaw. We noticed two very different situations: In some cases, a significant refactoring action is counterbalanced, in the same release, by an increase of *DSI* due to the same flaw, for example, Brain Method in JDT 3.0 and EMF 2.1, or God Class in JDT 3.0. In other cases, the improvements related to a flaw in a release surpass by far the occurrence of new negative signs, for example, God Class in EMF 2.4 or Code Duplication in JDT 3.0.

The common point between the two situations is that design flaws are often not coherently targeted. One example illustrates this point very well. In JDT 3.1, there was a massive, system-wide refactoring action aimed at addressing the Data Clumps flaw by replacing a group of parameters in all methods with a *parameter object* [7], but the refactoring was not performed systematically; in `ISourceElementRequestor`, the change was performed, whereas the very similar interface `IDocumentElementRequestor` was left unchanged. This shows that refactoring is not always performed coherently.

Lessons learned

Although our intuition says that a steep growth in code size would harm the design, the data from the two analyzed systems show that abrupt growths of code size do not necessarily increase debt and smooth growth does not guarantee that technical debt is under control.

Debt symptoms can have significant variations, and both systems show an alternation of decay and improvement phases, but the starting point is very important: JDT started with a high level of debt symptoms, and although reduced significantly, it stays above the values measured along the evolution of EMF.

The most significant variations of *DSI* occur in the earlier releases, whereas later in the lifetime of the systems, the *DSI* tends to stabilize completely. Furthermore, with one notable exception (JDT 3.3.1), the *DSI* suffers almost no variations during minor releases, which is a sign that none of the two development teams use the minor releases to perform refactorings (also see the section "Evolution of design flaws").

The different evolution patterns for JDT and EMF can be explained, at least partially, by their inherent differences. JDT must model the complex and constantly evolving elements of the Java language; and at the same time, it must implement a rich set of complex features aimed to support development (e.g., cross-referencing and refactorings). These two dimensions lead unavoidably to significant tangling and cross-cutting concerns, and this is amplified by the code optimizations needed to ensure a good time performance during parsing. All of these may explain the higher *DSI* values measured in the early releases of JDT. By contrast, the evolution of *DSI* in EMF could be explained by the fact that the application domain is easier to model in an object-oriented manner. In addition, the evolution of requirements in EMF is easier to foresee, because the meta-model in EMF is more stable than the specifications of various Java versions. Whereas some design flaws (e.g., Brain Method, Refused Parent Bequest, and Code Duplication) play a significant role in both JDT and EMF, not all design flaws have the same negative impact in both systems, and the same flaw may have different evolution patterns in different systems.

Beyond the results of this case study, the fact that design flaws are strongly correlated with external debt signs (e.g., flaws or excessive changes) is confirmed by a recent exploratory study [28], which analyzed more than 50 releases of four systems (including Eclipse) and 13 design flaws. This extensive study conclusively revealed that classes involved in design flaws are more fault- and change-prone than others, and size alone cannot explain the fact that classes with design flaws are more likely to undergo fault-fixing changes.

Threats to validity

In any experimental study, there are factors that can be viewed as possible influences and threats to validity [36]. Threats to external validity concern the possibility of generalizing our results. To make our results as generalizable as possible, we used two different Java systems and studied their evolution for more than 30 releases each, but we cannot be sure that the findings will be valid for other domains, applications, or development teams with a different level of experience in object-oriented design. More case studies are needed in order to establish whether the aforementioned observations concerning the evolution of technical debt are applicable in a different context. This experiment also did not include any design flaws that affect subsystems (e.g., Cyclic Dependencies), because their detection accuracy requires a precise specification of the subsystem structure, which is often unavailable for systems such as the ones that we analyzed. However, a recent study [37] has shown that approximately 80% of all architecture problems in programs were related to design flaws such as those described in this paper. This means that our framework can be used with relatively high confidence even if architectural flaws are not included.

Threats to internal validity concern any confounding factors that could have influenced the results of our study. This threat is mainly related to the assumption that cases of significant improvement in terms of *DSI* (see Figure 3) are the result of developers acknowledging the negative effect of a particular flaw. Although this assumption is in line with other studies [38, 39], we cannot exclude other factors influencing the decision to change a class. To limit this threat, we performed random manual inspections of classes in order to determine whether the changes were focused on removing the design problem or whether other changes unrelated to design quality concerns were performed on those classes. To a large extent, the results of these inspections support the assumption, but in order to draw any solid conclusions on the actual causes of the changes between releases, a detailed study is needed that would include interviews with the developers of the two Eclipse projects.

Conclusion

In this paper, we present an assessment framework for exposing and quantifying the symptoms of technical debt at the design level. The goal is to increase the visibility of design flaws that result from debt-incurring design changes. We have used the framework to analyze 63 releases of two well-known Eclipse projects: JDT and EMF.

Assessing design symptoms of technical debt is complex. It requires both a coarse-grained perspective to monitor the evolution of debt over time, and it requires a more detailed perspective that enables locating and understanding individual flaws, which can lead in turn to a systematic refactoring. The proposed framework provides both.

The provided assessment is quantitative and informative, because it directly measures technical debt in terms of major design flaws that incur future evolution and maintenance costs. This is particularly important because it provides concrete input for concrete corrective actions.

The framework is useful, because the case study showed that the framework characterizes quality aspects that the developers of the two systems have considered important. This is revealed by the fact that for all flaws, significant signs of restructuring actions have been noticed.

We have observed that the tool support offered by such a framework is necessary. Without it, corrective actions are not always coherent and systematic, and their benefit can be neutralized by new instances of the same flaws.

Although this framework is useful for capturing design symptoms of technical debt, it also has its limits. The experiment describes in detail how the framework can be customized and applied in a concrete assessment scenario, but the conclusions of the case study cannot be generalized, considering the few analyzed systems and the limited number of design flaws that were included in the actual instantiation of the framework.

We are aware that debt symptoms can also be project-specific; for example, an application with no internationalization support that is acquired by a company for which internationalization is mandatory automatically incurs debt [10]. Assuming that detection rules for project-specific debt symptoms can be defined, the framework is sufficiently flexible to accommodate them next to the general rules for design quality presented here. In addition, the framework is not aimed to measure or predict the actual effort or financial cost associated with technical debt [4]. This is a highly challenging task that needs to be addressed in the future.

Acknowledgments

I want to thank G. Ganea, I. Verebi, and A. Trifu, without whose work this paper would not have been possible. In addition, I am grateful to M. Minea, T. Gîrba, and C. Marinescu for their exceptional help in reviewing this manuscript. I also want to thank the Software Governance Team at IBM Thomas J. Watson Research Center (Hawthorne, New York) for all the exciting discussions on technical debt, which eventually triggered the writing of this paper. Finally, I want to thank C. Pickover and the three anonymous reviewers for their insightful comments that significantly strengthened the paper.

*Trademark, service mark, or registered trademark of Meridian Project Systems, Inc., Eclipse Foundation, Inc., Sun Microsystems, Inc., in the United States, other countries, or both.

References

1. R. S. Pressman, *Software Engineering: A Practitioner's Approach*, 7th ed. New York: McGraw-Hill, 2010.
2. S. McConnell, Technical debt. [Online]. Available: http://blogs.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx
3. W. Cunningham, "The WyCash portfolio management system," *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29–30, Apr. 1993.
4. M. Fowler. [Online]. Available: http://martinfowler.com/bliki/TechnicalDebt.htmlTechnical debt
5. C. Sterling, *Managing Software Debt*. Boston, MA: Addison-Wesley, 2011.
6. M. Lehman, "Laws of software evolution revisited," in *Proc. Eur. Workshop Softw. Process Technol.*, 1996, vol. 5, pp. 108–124.
7. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Boston, MA: Addison-Wesley, 1999.
8. R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proc. Int. Conf. Softw. Maint.*, 2004, vol. 20, pp. 350–359.
9. W. H. Brown, R. C. Malveau, and T. J. Mowbray, *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. London, U.K.: Wiley, 1998.
10. T. Klinger, P. Tarr, P. Wagstrom, and C. Williams, "An enterprise perspective on technical debt," in *Proc. Workshop Manag. Tech. Debt.*, 2011, vol. 2, pp. 35–38.
11. O. Ciupke, "Automatic detection of design problems in object-oriented reengineering," in *Proc. TOOLS*, 1999, vol. 30, pp. 18–32.
12. M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Heidelberg, Germany: Springer Verlag, 2006.
13. N. Moha, Y. G. Guéhéneuc, L. Duchien, and A. F. Le Meur, "DECOR: A method for the specification and detection of code and design smells," *IEEE Trans. Softw. Eng.*, vol. 36, no. 1, pp. 20–36, Jan./Feb. 2010.
14. M. J. Munro, "Product metrics for automatic identification of bad smell design problems in Java source-code," in *Proc. Int. Symp. Softw. Metrics*, 2005, vol. 11, pp. 15–24.
15. A. Trifu, *Towards Automated Restructuring of Object-Oriented Systems*. Karlsruhe, Germany: Kit Sci. Publ., 2008.
16. J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Trans. Softw. Eng.*, vol. 28, no. 1, pp. 4–17, Jan. 2002.
17. F. Deissenboeck, S. Wagner, M. Pizka, S. Teuchert, and J. F. Girard, "An activity-based quality model for maintainability," in *Proc. Int. Conf. Softw. Maintenance*, 2007, vol. 23. pp. 184–193.
18. R. G. Dromey, "A model for software product quality," *IEEE Trans. Softw. Eng.*, vol. 21, no. 2, pp. 146–162, Feb. 1995.
19. J.-L. Letouzey, "The SQALE method for evaluating technical debt," in *Proc. Workshop Manag. Tech. Debt*, Jun. 2012, vol. 3.
20. R. Marinescu and D. Raiu, "Quantifying the quality of object-oriented design: The Factor-Strategy model," in *Proc. Work. Conf. on Rev. Eng.*, 2004, vol. 11, pp. 192–201.
21. SonarSource, Sonar. [Online]. Available: http://www.sonarsource.org
22. SonarSource, Technical Debt. [Online]. Available: http://docs.codehaus.org/display/SONAR/Technical+Debt+Calculation
23. SonarSource, Total Quality Plugin. [Online]. Available: http://docs.codehaus.org/display/SONAR/Total+Quality+Plugin
24. P. Coad and E. Yourdon, *Object-Oriented Design*, 2nd ed. London, U.K.: Prentice-Hall, 1991.
25. R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ: Prentice-Hall, 2002.
26. B. Meyer, *Object-Oriented Software Construction*. Upper Saddle River, NJ: Prentice-Hall, 1988.
27. A. Riel, *Object-Oriented Design Heuristics*. Boston, MA: Addison-Wesley, 1996.
28. F. Khomh, M. Di Penta, and Y. G. Guéhéneuc, "An exploratory study of the impact of code smells on software change-proneness," *Empir. Softw. Eng.*, vol. 17, no. 3, pp. 1–33, 2012.
29. R. Marinescu and C. Marinescu, "Are the clients of flawed classes (also) defect prone?" in *Proc. Int. Work. Conf. Source Code Anal. Manipul.*, 2011, vol. 11, pp. 65–74.
30. V. Basili and H. D. Rombach, "The TAME project: Towards improvement-oriented software environments," *IEEE Trans. Softw. Eng.*, vol. 14, no. 6, pp. 758–773, Jun. 1988.
31. Intooitus, *inFusion*. [Online]. Available: http://www.intooitus.com/products/infusion
32. D. Raiu, S. Ducasse, T. Gîrba, and R. Marinescu, "Using history information to improve design flaws detection," in *Proc. Conf. Softw. Maintenance Reeng.*, 2004, vol. 8, pp. 223–232.
33. Eclipse Foundation, Java Development Tools. [Online]. Available: http://eclipse.org/jdt
34. Eclipse Foundation, Eclipse Modeling Framework. [Online]. Available: http://www.eclipse.org/modeling/emf
35. K. Kontogiannis, R. De Mori, M. Bernstein, M. Galler, and E. Merlo, "Pattern matching for design concept localization," in *Proc. Work. Conf. Reverse Eng.*, 1995, vol. 2, pp. 96–105.
36. D. T. Campbell, J. C. Stanley, and N. L. Gage, *Experimental and Quasi-Experimental Designs for Research*. Boston, MA: Houghton Mifflin, 1963.
37. I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. von Staa, "On the relevance of code anomalies for identifying architecture

degradation symptoms," in *Proc. 16th Conf. Softw. Maintenance Reeng.*, 2012, pp. 277–286.
38. S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding refactorings via change metrics," *ACM SIGPLAN Notices*, vol. 35, no. 10, pp. 166–177, Oct. 2000.
39. S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object-Oriented Reengineering Patterns*. Waltham, MA: Morgan Kaufmann, 2003.

Received March 2, 2012; accepted for publication April 9, 2012.

Radu Marinescu *LOOSE Research Group, Universitatea "Politehnica" din Timioara, Bvd. V. Pârvan 2, Timisoara 300223, Romania (radu.marinescu@cs.upt.ro).* Dr. Marinescu is an Associate Professor in the Computer and Software Engineering Department. He received a B.S. degree in computer science from the Politehnica University of Timisoara in 1997 and M.S. and Ph.D. degrees (*magna cum laude*) in software engineering jointly from the aforementioned university and the Technical University of Karlsruhe (Germany) in 1998 and 2002, respectively. He subsequently founded the LOOSE Research Group working on software maintenance and evolution as well as software measurement. In 2006, he received an IBM Eclipse Innovation Award and in 2009 the IBM John W. Backus Award for his work on quality assessment in object-oriented systems. He is coauthor of the book *Object-Oriented Metrics in Practice* and author or coauthor of more than 30 technical papers. Dr. Marinescu is currently serving as Vice President of the National Research Council, the Romanian equivalent of the U.S. National Science Foundation.