Continuous Detection of Design Flaws in Evolving Object-Oriented Programs using Incremental Multi-pattern Matching

Sven Peldszus
Institute for Software Technology
University of Koblenz-Landau, Germany
speldszus@uni-koblenz.de

Géza Kulcsár, Malte Lochau
Real-Time Systems Lab
TU Darmstadt, Germany
{geza.kulcsar, malte.lochau}@es.tu-darmstadt.de

Sandro Schulze
Institute of Software Technology Systems
TU Hamburg-Harburg, Germany
sandro.schulze@tuhh.de

ABSTRACT

Design flaws in object-oriented programs may seriously corrupt code quality thus increasing the risk for introducing subtle errors during software maintenance and evolution. Most recent approaches identify design flaws in an ad-hoc manner, either focusing on software metrics, locally restricted code smells, or on coarse-grained architectural anti-patterns. In this paper, we utilize an abstract program model capturing high-level object-oriented code entities, further augmented with qualitative and quantitative design-related information such as coupling/cohesion. Based on this model, we propose a comprehensive methodology for specifying object-oriented design flaws by means of compound rules integrating code metrics, code smells and anti-patterns in a modular way. This approach allows for efficient, automated design-flaw detection through incremental multi-pattern matching, by facilitating systematic information reuse among multiple detection rules as well as between subsequent detection runs on continuously evolving programs. Our tool implementation comprises well-known anti-patterns for Java programs. The results of our experimental evaluation show high detection precision, scalability to real-size programs, as well as a remarkable gain in efficiency due to information reuse.

CCS Concepts

•**Software and its engineering** → *Maintaining software; Object oriented development; Software evolution;*

Keywords

design-flaw detection, continuous software evolution, object-oriented software architecture

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

ASE'16, September 3–7, 2016, Singapore, Singapore
ACM. 978-1-4503-3845-5/16/09...
http://dx.doi.org/10.1145/2970276.2970338

1. INTRODUCTION

Object-oriented programming offers software developers rich concepts for structuring initial program designs, in order to cope with the inherent complexity of nowadays large-scale software systems. In this regard, design patterns serve as default architectural templates for solving reoccurring programming tasks in compliance with object-oriented design principles like separation of concerns [1]. As nowadays software systems tend to become more and more long-living, their initial code bases have to be continuously maintained, improved and extended over a long period of time. In practice, corresponding evolution steps are frequently conducted in an ad-hoc (and often even undocumented) manner (e.g., in terms of fine-grained manual program edits). As a result, the initial program design may be prone to continuous erosion, eventually leading to structural decay whose negative side-effects are usually summarized under the notion of *software aging* [2]. In particular, *design flaws* potentially infecting object-oriented programs may seriously corrupt code quality, thus increasing the risk for introducing subtle errors during software maintenance and evolution [3, 4, 5].

Object-oriented refactorings have been proposed as effective counter-measure against design flaws by means of behavior-preserving program transformations to be repeatedly interleaved with object-oriented development workflows [6, 7, 8]. In fact, a manual identification of problematic code structures to be removed by applying appropriate refactorings is tedious, error-prone, or even impossible for larger-scale software projects. Various approaches have been recently proposed to assist and/or automate the identification of design flaws in object-oriented programs. The different attempts may be roughly categorized into three kinds of *symptoms*, potentially indicating object-oriented design flaws [9].

- *Software metrics* assess quality problems in program designs by means of quantified measures on structural code entities (e.g., high coupling and/or low cohesion of classes) [10, 11].
- *Code smells* qualify problematic, locally restricted code structures and anomalies in-the-small, i.e., at class-/method-/field-level (e.g., relatively large classes) [12, 13, 14, 15, 16].
- *Anti-patterns* qualify architectural decay in the large, involving several classes spread over the entire program (e.g., God Classes) [17, 3, 18].

Based on this taxonomy, a precise and reliable identification of actual occurrences of design flaws in real-world programs requires arbitrary combinations of software metrics with adjustable thresholds, as well as code smells and anti-patterns into *compound detection rules* [9]. However, most existing approaches lack a comprehensive formal foundation and uniform, yet modular representation of such design-flaw detection rules. Instead, specifically tailored detection routines are applied for every design flaw individually, and being re-evaluated from scratch for every program version anew during software evolution [11].

In this paper, we present a comprehensive methodology for specifying and automatically detecting design flaws in object-oriented programs. The approach utilizes a unified abstract program model comprising those high-level object-oriented code entities being relevant for a concise specification of well-known design flaws [19, 20]. Based on this model, compound design-flaw detection rules integrate software metrics, code smells and anti-patterns, and allow for arbitrary combinations thereof. The modular nature of the rule language allows for sharing similar symptoms among multiple design-flaw rule specifications. The corresponding pattern-matching routines derived from those rules incrementally augment the underlying abstract program model with qualitative and quantitative design-related information. This technique builds the basis for efficient design-flaw detection by systematically facilitating reuse of information among multiple detection rules, as well as between subsequent detection runs on continuously evolving programs. To sum up, we make the following contributions.

- A comprehensive rule-based methodology for object-oriented design-flaw specification based on a unified program model, integrating the entire spectrum of possible symptoms.
- An efficient technique for concurrently detecting multiple design flaws on continuously evolving programs based on incremental multi-pattern matching.
- A tool implementation automating the detection of well-known object-oriented code smells and anti-patterns in Java programs.
- Evaluation results gained from experimental applications of our tool to real-world Java programs. The results demonstrate high detection precision and scalability to real-size programs, as well as a remarkable gain in efficiency due to information reuse.

Please note that our tool implementation, as well as all experimental data sets, are available on our GitHub site[1].

2. BACKGROUND

Consider the sample Java program in Figure 1, serving as our running example in the remainder of this paper. The example comprises an extract from the source code of a simplified object-oriented catalog-software of a video-rental shop. The program design is further illustrated by the class diagram in Figure 2. Originally, the rental shop only possessed early horror classics, like THE BLOB from 1958. Later on, they decided to reward customers for frequent rentals by providing free bonus movies, like the 'refactored' remake of THE BLOB from 1988. Therefore, a software developer extended the existing shop system by adding the source code parts highlighted in gray in Figure 1. His design decision was

[1]http://github.com/Echtzeitsysteme/hulk-ase-2016

```
 1 public class Shop {
 2   private String shopName = "DemoShop";
 3   private List   customer = new ArrayList();
 4   private List   movies   = new ArrayList();
 5   public void addCustomer(String name){...}
 6   public void addMovie(String title, int year){...}
 7   public void rentMovie(Customer cu, Movie mo){...}
 9   public void addCustomer(String name, int bonus){
10     Customer movie = new Customer();
11              movie.setName(name);
12              movie.setBonus(bonus);
13     this.customer.add(customer)
14   }
15   public void addBonusMovie(String title, int year,
         int level){ .. }
16   public void rentBonusMovie(Customer customer,
         BonusMovie movie){...}
17   public void updateBonus(Customer customer, Movie
         movie){...}
18   public int calculateBonusPoints(BonusMovie movie,
         int bonus){...}
19 }
20 public class Customer {
21   private   String name;
22   protected List   rents = new ArrayList();
23   private int bonus;
24   public void setBonus(int bonus) {
25     this.bonus = bonus;
26   }
27   public String getBonus(){
28     return bonus;
29   }
30   public void setName(String name) {...}
31   public String getName(){...}
32 }
33 public class Movie {...}
35 public class BonusMovie {...}
```

Figure 1: Source Code of the Video Rental System

to add a new sub-class `BonusMovie` (line 35) to represent free bonus movies. To store the achieved bonus points of each customer, he added a field `bonus` as well as a corresponding *getter* and a *setter* method for this field (lines 24 to 29) to the class `Customer`. In lines 9 to 18, the creation and calculation of new bonus movies and the required bonus points are implemented by four additional methods in the class `Shop`. During a subsequent maintenance phase, a senior developer reviewed the evolved software system by re-investigating the class diagram in Figure 2 (again, highlighted in gray) to get an overview of the edits performed.

During code review, the senior developer relies on quantitative software metrics (e.g., number of class members) as well as qualitative characteristics of single classes, methods and fields. The recognition of such local symptoms of particular object-oriented entities (i.e., so-called code smells) therefore relies on manual judgments based on the available metric values. In this particular case, the senior developer considered the `Shop` class to be significantly larger than the other classes. In addition, the methods of class `Shop` intensively call getters and setters of other classes that, in turn, do not implement actual functionality, but rather only hold program data. Moreover, different methods in `Shop` do not call each other as they have different concerns, but mainly call methods in other classes, resulting in low cohesion of class `Shop` (cf. the review notes in Figure 2, bottom left), but high couplings of that class with other classes.

The presence of a predominant class like `Shop`, obliged with too much responsibility and degrading other classes to the mere task of storing data, constitutes a misuse of

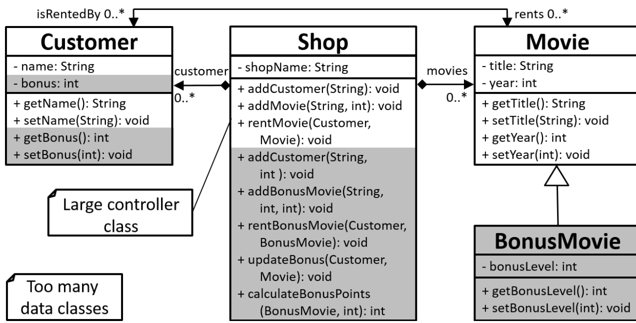

Figure 2: Class Diagram of the Running Example

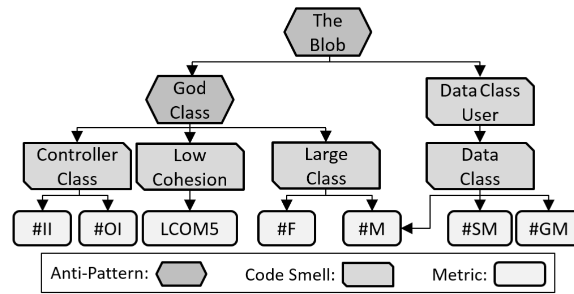

Figure 3: Modular Description of *The Blob* Anti-Pattern

object-oriented encapsulation concepts. This specific combination of symptoms therefore indicates a well-known object-oriented *design flaw*, usually referred to as *The Blob* anti-pattern [17]. As a consequence, software maintenance in subsequent evolution steps may be seriously obstructed and error-prone. For instance, suppose that the video shop later plans to offer on-line streaming, which is however excluded from the bonus system. This new functionality shall be naturally implemented in a new sub-class of the central `Shop` class. But, concerning the current implementation in Figure 2, this would mean that methods for handling the bonus system are also included in this new class, thus leading to erroneous behavior.

As illustrated by this example, manually performing comprehensive and precise design-flaw detection is tedious, error-prone, and becomes infeasible in a real-world software development scenario comprising hundreds of thousands of lines of source code and multiple developers involved. In this regard, design-flaw identification needs to be assisted and, at least up to some extent, automated. In particular, a comprehensive design-flaw specification and detection framework has to integrate the following techniques, corresponding to the aforementioned different kinds of symptoms.

- Selection and computation of quantifiable *software metrics* with adjustable thresholds,
- identification of *code smells* by means of fine-grained properties of object-oriented entities, including software metrics and locally restricted structural patterns occurring in particular classes, methods or fields, and
- detection of architectural *anti-patterns*, composed of various code smells and coarse-grained global program structures, as well as (static) semantic code patterns.

Revisiting our running example, the identification of the design flaw *The Blob* in the evolved program involves various code smells. First, the central class constitutes a so-called *Controller Class*, i.e., a class with a predominant amount of outgoing method calls to members of other classes, clearly exceeding the amount of incoming calls. Moreover, the central class constitutes a *Large Class* with *Low Cohesion*, i.e., having a high amount of outgoing method calls compared to those targeting the same class. The other classes centered around this central class serve as *Data Classes*, as their methods are mostly getters/setters providing accesses to fields storing program data. In this regard, the central class further takes the role of a *Data Class User* by making intensive use of Data Classes. Each of these code smells may be characterized by utilizing different kinds of established software metrics [21, 22, 23], amongst others:

- **#II/#OI:** number of incoming/outgoing method invocations of a class,
- **LCOM5:** a popular variant of the Lack-of-Cohesion-of-Methods metric,
- **#F/#M:** number of fields/methods in a class, and
- **#SM/#GM:** number of setters/getters in a class.

Based on these metrics, the aforementioned code smells may be described as follows.

- **Data Class.** A *Data Class* has a high ratio between the number of its getter-setter methods and all of its methods, denoted as
$$\frac{\#\text{SM}+\#\text{GM}}{\#\text{M}},$$
compared to all other classes in the program.
- **Large Class.** A *Large Class* has a significantly higher number of members (fields and methods), denoted as $\#F + \#M$, compared to the average of all classes in the program.
- **Low Cohesion.** A class with *Low Cohesion* has a significantly higher LCOM5 value compared to the average of all classes in the program.
- **Controller Class.** A *Controller Class* has a significantly smaller ratio between the number of incoming and outgoing method invocations, denoted as
$$\frac{\#\text{OI}}{\#\text{II}},$$
compared to the average of all classes in the program.
- **Data Class User.** A *Data-Class User* has significantly more accesses to Data Classes compared to the average of all classes in the program.

Finally, specifying fully-fledged architectural anti-patterns requires arbitrary combinations of code smells with global program structures and (static) semantic information. For instance, concerning the *The Blob* anti-pattern of our running example, we observe that the description and, therefore, the detection of some code smells being part of a *The Blob* anti-pattern implicitly depend on each other (e.g., in terms of "consists-of" relationships), thus inducing a (partial) (de-)composition hierarchy among the different symptoms constituting a design-flaw specification. Figure 3 visualizes such a modularized description of the *The Blob* anti-pattern, making explicit the dependency hierarchy among the respective software metrics and code smells, as initially proposed in [9].

In practice, not only one, but *multiple* different design flaws have to be taken into account during software main-

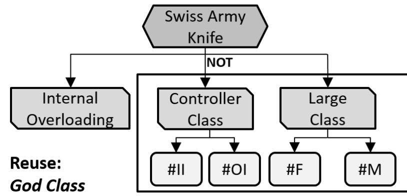

Figure 4: Modular Description of the *Swiss Army Knife* Anti-Pattern

tenance. Although differing in subtle details, the variety of design flaws documented in the literature usually share, up to a certain level of detail, similar symptoms. To this end, descriptions of anti-patterns may also include references to (parts of) other anti-patterns. For example, *The Blob* contains the anti-pattern *God Class* as integral part of its specification. A *God Class* is characterized as a relatively *Large Class* which monopolizes control over system behavior and unifies different system concerns [24]. This is indicated by the code smells *Large Class*, *Controller Class* and *Low Cohesion* in the description shown in Figure 3 [5].

Besides referencing entire anti-pattern descriptions within other anti-patterns, the same may be done on a more fine-grained level by *reusing* code-smell specifications in different anti-patterns, as demonstrated by the *Swiss Army Knife* anti-pattern in Figure 4. Intuitively, a *Swiss Army Knife* is a *Large Class* which has been designed to act as a super-interface for a huge variety of foreseeable default tasks [17]. In contrast to *The Blob*, a *Swiss Army Knife* does not take control over control-flows in the system—it rather fills the role of a universal toolkit. This role is captured by the *Internal Overloading* code smell, which means that there are many overloaded method implementations with identical names, but different signatures, one for each possible caller-context. To express further characteristics of a *Swiss Army Knife*, we are able to reuse code smells already specified for *God Class* and *The Blob*, namely *Controller Class* (a *Swiss Army Knife* does *not* have this property, also indicated on the corresponding edge in Figure 4) and *Large Class*. As illustrated by this example, modular description hierarchies not only permit conjunction (AND), but likewise negation (NOT) and disjunction (OR) of design-flaws, as well as arbitrary combinations thereof.

Reconsidering our example, not all the symptoms leading to *The Blob* appear at the same time as result of one single evolution step; instead, certain code smells may have already been introduced in steps preceding the extension shown in Figure 2. For instance, class `Movie` has been a *Data Class* and `Shop` a *Controller Class* with *Low Cohesion* already before the extension took place (i.e., white parts in Figure 2). After the extension (gray parts in Figure 2) has been performed, new *Data Classes* appear in the system, thus finally leading to *The Blob*. The intermediate information may therefore be collected and preserved over consecutive evolution steps to facilitate continuous design-flaw detection without re-evaluating the entire set of symptoms of compound design-flaws from scratch.

To summarize, a comprehensive methodology for a formalized specification and automated detection of design-flaws in object-oriented programs has to satisfy the following three requirements **R1**–**R3**.

(R1) Formal specification framework for modular description and systematic integration of different styles of established design-flaw symptoms, defined on a unified representation of object-oriented programs.

(R2) Automated derivation of incremental design-flaw detection routines, exploiting information reuse derived for multiple design flaws with similar symptoms while continuously investigating evolving programs.

(R3) Integration into existing software development workflows by making detection results accessible in a comprehensible manner, thus further supporting refactoring decisions [25].

3. RULE-BASED DESIGN-FLAW SPECIFICATION AND INCREMENTAL DETECTION

In this section, we present a rule-based framework for specifying and detecting object-oriented design-flaws that meets the aforementioned requirements **R1**–**R3**.

3.1 Program Model

As demonstrated in Section 2, effective object-oriented design-flaw identification requires an appropriate abstract representation of the program under consideration, even for a manual recognition. As object-oriented design flaws mostly occur at the level of high-level object-oriented program entities (classes, fields, methods), the Abstract Syntax Tree (AST) representation of a program is no appropriate representation for design-flaw characterization, because

- the AST comprises every fine-grained syntactic detail of the program, including instruction code in method bodies, and
- the AST does not make explicit further crucial (static) semantic information crosscutting the hierarchical program structure, such as field-access and method-call dependencies between classes.

Hence, instead of considering the entire AST, we illustrated the incremental detection of the *The Blob* anti-pattern for our running example by annotating related code-smell information on the corresponding UML class diagram representation in Figure 2. However, even if such a class diagram is available for existing software, it usually mainly serves documentation purposes, rather than being directly attached to, and continuously synchronized with, the (evolving) program code. The resulting lack of preciseness and accuracy of existing object-oriented design models, therefore, make them inappropriate for design-flaw specification and detection. Instead, we pursue in our approach to reverse-engineer and continuously update a unified *program model representation* from the AST of the program source. This program model is specifically tailored in such a way that it serves as a basis for a comprehensive rule-based design-flaw specification according to requirement **R1**. In particular, the program representation provided by the model abstracts from unnecessary details of the program AST, thus permitting comprehensive identification of, and navigability through program entities being relevant for design-flaw specification. In addition, the model is further augmented with design-related meta-information in terms of (approximated) static semantic dependencies (field accesses and updates, as well as method calls) and quantified structural properties (e.g., coupling/cohesion), derivable from those dependencies.

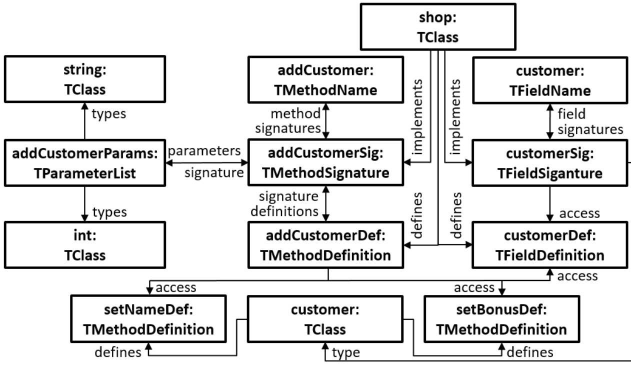

Figure 5: Program Model (Extract): Method `addCustomer`

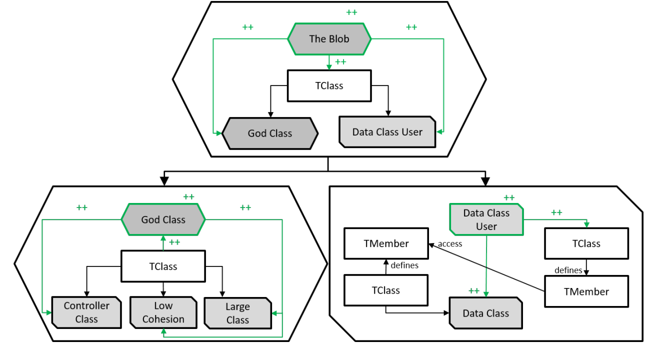

Figure 6: Detection Rule of *The Blob* Anti-Pattern

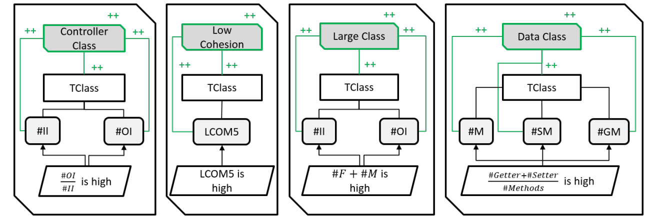

Figure 7: Detection Rules for Code Smells in *The Blob* Anti-Pattern

In order to facilitate information reuse among multiple design-flaws with shared symptoms as well as between subsequent detection runs on continuously evolving programs (cf. requirement **R2**), the program model further allows for annotating (intermediate) detection results to affected program elements (e.g., marking classes as *Large Classes*, *Data Classes* and/or *Controller Classes*). Those annotations are created and/or aggregated and incrementally refined during design-flaw detection, thus leading to an *annotated program model*, which is an extension of the program model. The accumulated information may be shared among multiple design-flaw rules, and is continuously synchronized (i.e., updated) with the (evolving) source code.

As an example, Figure 5 shows an extract of the program-model instance derived from the AST of the video shop system presented in Section 2. Here, the representation of method `addCustomer` from class `Customer` is enriched with further structural dependencies. Each program element is represented by one or more objects in the model, where each object is identified by a unique *name* and a *type*. Please note that we use the naming convention *name:type* in the illustration. For instance, classes are represented as objects of type `TClass` (e.g., `shop:TClass` on the top). The representation of methods (e.g., `addCustomer`) consists of three objects of types `Name`, `Signature` and `Definition`. These three properties of methods are particularly relevant for code-smell and anti-pattern specification including overloading and overriding. The labeled edges between object nodes represent different types of (syntactic and static semantic) dependencies among the corresponding program entities. In particular, *access* edges (denoting method calls and field accesses performed in `addCustomer`) are depicted at the bottom, including two method calls (`setNameDef` and `setBonusDef`) and, on the right-hand side, one field access to `customerDef`.

In the following section, we describe a design-flaw detection technique for incrementally annotating the program model with aggregated information about software metrics, code smells and anti-patterns

3.2 Design-Flaw Detection using Incremental Multi-Pattern Matching

We now devise a rule-based design-flaw specification approach and an accompanying detection process which integrates the different kinds of design-flaw symptoms (cf. Section 2) by addressing the requirements **R1**–**R3** (Sec. 2).

R1 – Modular Description.

To address **R1**, our design-flaw detection framework provides a modular description technique for all three kinds of design-flaw symptoms. For this purpose, code-smell and anti-pattern descriptions such as those illustrated in Figure 3 and Figure 4, as well as their combinations into compound design flaws, are formulated in terms of graphic *detection rules* defined on the program model. Figure 7 shows detection rules for the four code smells, occurring in *The Blob*, namely *Controller Class*, *Low Cohesion*, *Large Class* and *Data Class*. A detection rule consists of two parts: (i) (unmarked) black elements specify the *preconditions* of the rule, and (ii) green elements (marked with "++") specify the *annotation creations*. Correspondingly, the *execution* of a detection rule consists of two steps: (i) every (annotated) pattern within the program model is determined, matching the preconditions of the rule, and (ii) for each such pattern, annotations are created and assigned to the affected entities in the program model.

As an example, consider the rule for detecting a *Data Class* in Figure 7. Here, the precondition (i) matches those classes for which the ratio of the number of getters/setters and the number of all methods is *high*, compared to all other classes in the program. To evaluate this rule, the ratio is computed for each class and an appropriate threshold for 'high' is derived by deploying corresponding software metrics. In step (ii), for the respective `TClass` object of the class fulfilling the preconditions, a corresponding *Data Class* annotation is created and attached to that object. The creation of annotations for *Controller Class*, *Low Cohesion* and *Large Class* code smells is performed in a similar way.

R2 – Continuous Multi-Pattern Detection.

Concerning requirements of **R2**, detection routines are derived from detection rules by means of *pattern matching*. In

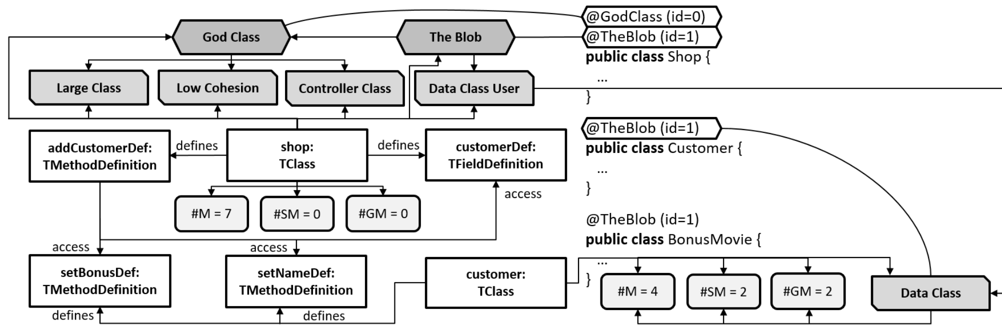

Figure 8: Annotated Program Model and Source Code of the Video Rental System

particular, we apply *incremental multi-pattern matching* to facilitate information reuse on program-model annotations as follows.

- *Incrementally* updating design-flaw annotations from the version history of evolving programs by preserving unaffected annotations and only updating those potentially affected by program changes.
- Detecting *multi-patterns* concurrently: annotations for software metrics, code-smells and anti-patterns are derived only once for a particular program version, and then shared among multiple design-flaws having those symptoms in common.

In our example program, when applying the *Data Class* rule to `customer:TClass` on the bottom, the class gets a *Data Class* annotation as seen on the left-hand side of Figure 8. The ratio of getter/setters to all methods is 1 (i.e., there are only getters/setters) whereas for `shop:TClass`, it is 0. Thus, the preconditions of the rules are fulfilled and the annotation is created.

Once the code smell annotations are created in the program model, the detection proceeds with applying anti-pattern detection rules. Figure 6 shows the three rules for detecting and annotating *The Blob*. The principle of executing an anti-pattern detection rule is exactly the same as for code smell detection rules. The difference lies in the preconditions which in this case also contain code smell annotations.

The *God Class* rule (bottom left) simply checks the presence of the respective code smell annotations as precondition and creates the *God Class* annotation accordingly. The *Data Class User* rule (bottom right) does not solely rely on the *Data Class* annotation, but it also takes into account an additional static semantic information in the program model: access between members of different classes. If there is an access from a class (in the rule on the right) to another one (in the rule on the left) which is a *Data Class*, the first one gets annotated as *Data Class User*. The rule for *The Blob* (Figure 6 on the top) relies on the two previously introduced rules and annotates a class which is a *God Class* and a *Data Class User* as *The Blob*.

The left-hand side of Figure 8 shows the complete annotation of our sample program model excerpt (Figure 5). As the `shop:TClass` has all the relevant code smell annotations, it becomes a *God Class* and as, moreover, it is also a *Data Class User*, we annotate this class as *The Blob* (annotations on the top of Figure 8). However, *The Blob* does not only consist of a single central class, but also of data classes it accesses (here, `customer:TClass`). In the annotated program model, annotations not only represent single, separated markers in the program code, but they also have structural links representing the associations between them. For example, in Figure 8, the *Data Class User* annotation of `shop` points to the *Data Class* annotation of `Customer` as prescribed by the *Data Class User* rule.

Considering the additional rules required for detecting *Swiss Army Knife* as described in Figure 4, we are able to reuse annotations for *Large Class* and *Controller Class* code-smells derived for the *The Blob* anti-pattern for the detection of *Swiss Army Knife* as well. Our detection rules therefore allow to express that a *Swiss Army Knife* should *not* be a *Controller Class*. Hence, as the only *Large Class* is also a *Controller Class* in our running example, no *Swiss Army Knife* is detected. To illustrate *incremental detection*, reconsider the program edit shown in Figure 2. After design-flaw detection has been performed on the original program, `Movie` is marked as *Data Class*. After editing the program, this annotation remains unaffected and the *Data Class* detection rule does not have to be re-executed as the respective annotation is preserved.

To summarize, the rule-based detection process reflects the three kinds of symptoms described in Section 2:

- calculating and integrating *software metrics* data into the program model by annotating corresponding values to respective program model elements,
- identifying and annotating *code smells* by evaluating code-smell detection rules, incorporating combinations of software metrics and (locally restricted) pattern matching concerning structural properties of particular classes, methods and fields, and
- detecting *anti-patterns* using anti-pattern detection rules, comprising code-smell annotations and structural as well as static semantic patterns involving multiple entities of the entire program.

R3 – Integration into Software Development.

To address requirement **RQ3**, detection results are offered to the developer by propagating program-model annotations into the source as Java annotations. To this end, the program model is continuously synchronized as illustrated on the right-hand side of Figure 8. In this presentation, only those anti-patterns are displayed, being of immediate relevance for subsequent software-maintenance steps. For traceability and documentation purposes, a unique identifier is assigned to every design-flaw.

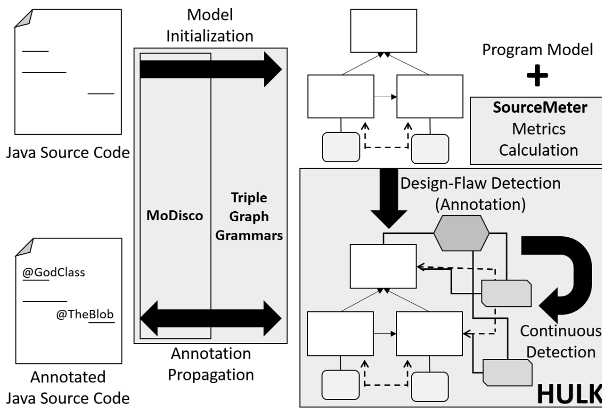

Figure 9: Implementation Architecture

4. IMPLEMENTATION

We now present an Eclipse-based implementation of the proposed design-flaw detection framework being applicable to Java programs. Our tool, called HULK [2] integrates components for the previously described program modeling, incremental multi-pattern matching and software-metrics computation on Java programs. For constructing the program model and for deriving corresponding pattern-matching routines from detection rules, we utilize the meta-CASE tool eMoflon[3]. For evaluating software metrics, we use the static analysis tool SourceMeter[4]. The implementation along with user instructions (as well as our evaluation results) are publicly available on the GitHub site of HULK[5]. Figure 9 illustrates the analysis steps performed on a given Java program and the corresponding components of HULK in detail.

Model initialization. The program model is constructed (arrow on the top) using the Eclipse plug-in MoDisco[6] as a code-to-model adapter. For bidirectional code-to-model synchronization, we apply *Triple Graph Grammars* (TGG) [27], to establish a correspondence between the MoDisco Java model and our program model for design-flaw detection.

Software-metrics calculation. Software metrics values are calculated by SourceMeter and attached to the respective program model elements (program model with metrics in the upper right corner).

Program-model annotations. Code smell and anti-pattern detection rules are executed to annotate the program model with design-flaw information. Our tool provides a graphical front-end for visually specifying detection rules (closely resembling Fig. 6-7). A rule specification is not attached to one specific design flaw, but rather might be (re-)used within arbitrary compound rule definition via rule references. Based on these rules, routines for incremental multi-pattern matching as described in the previous section, are automatically derivable including schedules for an appropriate application order according to their inter-dependencies. Our tool provides predefined rules for the anti-patterns *God Class*, *The Blob*, *Swiss Army Knife*, *Spaghetti Code* [17], and is easily extensible to incorporate further detection rules.

Continuous detection and annotation propagation. The TGG-based synchronization mechanism facilitates continuous (re-)detection as described in Section 3.2, by propagating/preserving annotations shared between multiple rules as well as between between subsequent detection runs on different program versions.

[2] The green monster Hulk is the only member of the Marvel universe known to be able to cope with 'The Blob'[26].

[3] http://www.emoflon.org/

[4] http://www.sourcemeter.com/

[5] http://github.com/Echtzeitsysteme/hulk-ase-2016

[6] http://eclipse.org/MoDisco/

5. EVALUATION

In this section, we present the evaluation results of applying the implementation of our proposed design-flaw detection technique on a corpus of 13 real-size Java programs from various application domains (cf. the first column in Table 1) to consider the following research questions.

RQ1 (Scalability): Is the proposed design-flaw detection technique *applicable* to real-size Java programs in a reasonable amount of time?

RQ2 (Precision): Does the proposed design-flaw detection technique produce sufficiently *precise* results?

RQ3 (Efficiency): To what extent does incremental multi-pattern detection improve the *efficiency* of the proposed design-flaw detection technique?

Our selection of subject systems relies on former experiments performed for related approaches, as well as on a standard catalog for analyzing evolution of Java systems [9, 18, 28, 29] to address **RQ3**. We selected open-source Java programs from different application domains, including systems for software developers as well as for end-users. We also aimed at including a range of different program sizes. Moreover, for two programs, *Gantt* and *Xerces*, there exists a manual detection oracle which allowed us to evaluate **RQ2** [18]. The particular program versions considered for the experiments, together with the URL for accessing source code, are included on our accompanying GitHub site.

5.1 Experimental Setup

We now describe the details on the experimental setup and methodology to obtain the results for answering **RQ1**–**RQ3**. Concerning **RQ1**, we applied our proposed detection technique to all subject systems considering the following anti-patterns: *God Class, The Blob, Swiss Army Knife, Spaghetti Code*. Moreover, we monitor the execution and measure execution times of each step, namely: (1) program-model initialization (Init), (2) calculation of software metrics (SM), and (3) design-flaw detection (Det).

For answering **RQ2**, we compare our approach with two approaches that pursue similar goals, but use different techniques: DECOR [9] and BDTEX [18]. DECOR also provides a rule-based approach for anti-patterns specification, but there are essential differences regarding the description formalism and detection mechanism. BDTEX is also similar to DECOR, but further incorporates machine learning.

To evaluate and compare the three approaches concerning **RQ2**, we rely on the standard detection quality metrics *precision* and *recall* [30]. However, these metrics require an oracle delivering correct detection results. For our evaluation, we rely on a set of anti-patterns to serve as oracles that has been manually identified as part of the evaluation of BDTEX [18]. This oracle contains manual judgments about *The Blob* anti-patterns in the programs *Gantt* and *Xerces* (cf. also Table 1). Based on this oracle, we conduct our evaluation as follows: *true positives* are those anti-patterns detected which are also predicted by the oracle, *false posi-*

Table 1: Program Statistics and Execution Times

Project	LOC	#C	Init [s]	SM [s]	**Det** [s]	AP
QuickUML	2,667	19	15.11	13.36	0.14	7
JSciCalc	5,437	121	38.84	13.89	0.16	6
JUnit	5,780	105	58.2	14.54	0.13	9
Gantt	21,331	256	620.47	35.46	0.56	12
Nutch	21,437	273	396.86	30.95	0.63	18
Lucene	25,472	276	447.03	33.15	0.49	24
log4j	31,429	394	691.61	44.61	0.95	29
JHotDraw	31,434	312	742.05	47.59	0.65	25
jEdit	49,829	514	1,251.56	58.38	1.64	38
PMD	53,214	860	1,461.75	80.34	4.2	38
JTransforms	71,348	36	1,363.07	157.29	0.14	16
Xerces	102,052	642	4,573.5	122.6	2.89	39
Azureus	201,527	1,623	12,275.55	233.88	17.2	91

tives are anti-patterns being detected, but not predicted by the oracle, and *false negatives* are those being predicted by the oracle, but not being detected.

Concerning **RQ3**, we consider the following sub-questions:

RQ3.1 (Multi-Patterns): To which extent does information reuse among multiple design-flaw rules speed-up detection compared to rule-by-rule detection?

RQ3.2 (Continuous Detection): To which extent does information reuse among subsequent detection runs on different program versions speed-up design-flaw detection compared to version-by-version detection?

For **RQ3.1**, we evaluate the speed-up achieved when detecting multiple anti-patterns in one detection run (as in **RQ1**), compared to detecting each design flaw separately. In particular, we perform metrics calculation and detection time measurements (cf. phases (2)–(3) in **RQ1**) for the following scenarios (where abbreviations indicate detected anti-patterns): Blob (*The Blob*), SAK (*Swiss Army Knife*), SC (*Spaghetti Code*), All (four in combination as in **RQ1**). Note that detection of *God Class* is included in *The Blob* detection. The *speed-up factor* $\mathbf{s_{multi}}$ is calculated as

$$\mathbf{s_{multi}} = 1 - \frac{\text{All}}{\text{Blob} + \text{SAK} + \text{SC}},$$

where the names represent the metrics calculation and detection time needed for those individual scenarios. For **RQ3.1**, we perform measurements on a previously stored program-model state, resulting from the experiments conducted for **RQ1** after phase (1) (i.e., model initialization). To answer **RQ3.2**, we selected a set of fine-grained program edits which frequently occur during continuous software evolution. In this regard, evolution steps do not comprise complicated structural program changes in the large, but rather consist in introducing or deleting particular methods and/or fields, as well as renaming operations, as can be observed in the evolution history of the Qualitas Corpus, a standard catalog for analyzing object-oriented system evolution [29]. For our measurements, we first perform detection on an unchanged program state. Afterwards, we perform a program edit and measure the detection times before and after that change. The speed-up $\mathbf{s_{evolution}}$ is presumably obtained through *incremental pattern matching* and is calculated according to the formula $\mathbf{s_{evolution}} = 1 - (t_\Delta / t_0)$, where t_0 represents the complete initialization and detection time of the unchanged program state and t_Δ denotes the time needed to recalculate design-flaw annotations after program edits. As edits are limited to very few program elements, we assume that re-detection without incremental matching requires the same detection time as for the unchanged program state, i.e., t_0. For experimental purposes, we simulate following program edits: *Delete Method* from *Gantt*, *Create Class*: inserting a fresh class into *JHotDraw*, *Create Method*: inserting a fresh method with a simple return statement into the `CommonPanel` class of *Gantt*, *Rename Class* in *QuickUML*.

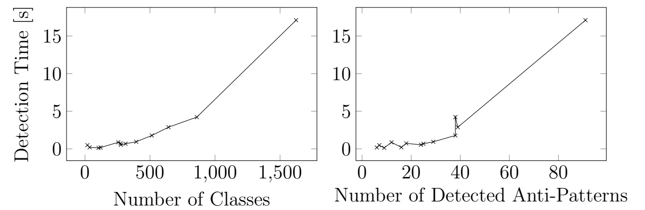

Figure 10: Detection Times for RQ1

5.2 Results and Discussion

We present and discuss the results of our experiments with respect to our research questions. All experiments have been performed on a Windows 10 PC with an Intel i5-3570K quad-core processor, 8 GB DDR3 RAM and JDK v1.8.0_91.

RQ1 – Scalability.

Table 1 lists the Java programs used as subject systems along with their size (LOC: Lines of Codes, #C: number of classes). In the next three columns, detailed results of execution times (phases (1)–(3), see Section 5.1) are given in seconds (median values out of 5 runs). The last column (AP) shows the number of detected anti-patterns. The first plot in Figure 10 depicts the relation between detection times (the time spent for actual detection after initialization and metrics calculation) and the number of classes in the program. The second plot in Figure 10 depicts the relation between detection times (similar to the first plot) and the number of anti-patterns found in the program. Both plots indicate a second-order polynomial increase as expected.

To answer **RQ1**, the results show that the time required for design flaw detection is reasonable also for larger-scale programs. In all cases, the time required for phase (3) is much lower than the execution times of phases (1) and (2). As our implementation supports continuous detection, initialization costs might be omitted later on in the case of evolving programs. The relatively low execution times of the detection itself make our approach applicable for frequent usage on real-size Java projects.

RQ2 – Precision.

The precision and recall values for detecting *The Blob* in *Gantt* and *Xerces* are given in Table 2, where the values for DECOR and BDTEX are based on the oracle and evaluation data in [18]. In contrast to other approaches, we decided to adjust our design-flaw rules to obtain high precision. We believe that for an automated design-flaw detection approach to gain user-acceptance for assisting continuous software development and maintenance, it is crucial to avoid too many

Table 2: **Comparison: Precision and Recall for Gantt and Xerces – Detection of *The Blob***

Approach	Gantt Precision [%]	Gantt Recall [%]	Xerces Precision [%]	Xerces Recall [%]
HULK	**100**	50	**100**	5
DECOR	87.5	63.6	90.6	100
BDTEX	21	100	21	100

Table 3: Execution Times and Speed-up for RQ3.1

Project	Blob [s]	SAK [s]	SC [s]	All [s]	s_{multi} [%]
QuickUML	18.17	12.23	10.35	12.29	69.8
JSciCalc	15.34	15.32	15.28	15.42	66.4
JUnit	16.70	16.66	16.40	16.70	66.4
Gantt	35.91	35.71	35.75	36.07	65.9
Nutch	30.47	30.30	30.42	30.70	66.4
Lucene	33.90	33.86	33.80	34.30	66.0
log4j	43.77	43.65	43.61	44.09	66.4
JHotDraw	48.00	46.05	46.33	47.80	66.7
jEdit	59.44	58.73	58.90	60.40	65.2
PMD	79.81	78.21	80.76	81.53	65.4
JTransforms	159.76	157.59	158.37	158.86	66.9
Xerces	110.96	109.22	109.54	111.56	66.2
Azureus	225.46	220.99	224.62	235.62	64.8

false alarms. However, aiming at high precision comes with a price regarding recall (i.e., precise detection rules tend to miss cases being considered as design flaws by other approaches) as can be seen in Table 2. For smaller programs with few design flaws (Gantt), we achieve 100% precision while still reaching a recall close to DECOR. Even for a large program with (presumably) many design flaws (Xerces), we achieve maximal precision, but reach a very low recall value due to our much more restrictive detection rules and a high number of *The Blob* anti-patterns demanded by the oracle. One of our subject systems, JHotDraw, is widely known for its excellent object-oriented design with no major anti-patterns as it has been developed as a design exercise for object-oriented design experts [18]. Our approach detects 0 *The Blob* anti-patterns in JHotDraw, while detection with BDTEX results in 1 false positive.

To summarize **RQ2**, we conclude our approach to guarantee high precision at the cost of low recall values. Nevertheless, the rule definitions allow for arbitrary adjustments in either directions.

RQ3 – Efficiency.

Concerning **RQ3.1**, Table 3 shows the execution times of metrics calculation and design-flaw detection (average value over 10 runs) for our four scenarios (cf. Sec. 5.1), along with the achieved speed-up s_{multi}. The reason for this value being close to 2/3 of the time required for every program is that, as already seen for **RQ1**, the detection times are negligible compared to metrics calculation, which only takes place once in case of multi-pattern matching (All). Evaluating the execution times for subsequent runs, standard deviation was in every case negligible (less than 8%).

Concerning **RQ3.2**, Figure 11 shows the ratio of t_0 and t_Δ as a bar-chart for the four basic program edits. The achieved speed-up $s_{evolution}$: *Delete Method*: 44.16%, *Create Class*: 42.84%, *Create Method*: 57.7%, *Rename Class*: 26.05%. As expected, edits on smaller programs (*QuickUML*) lead to a lower, but still remarkable speed-up, as the ratio of the edited program part to the whole is higher in this case. In general, the achieved speed-up factor is highly encouraging, continuous and incremental design-flaw detection being the main goal of our approach.

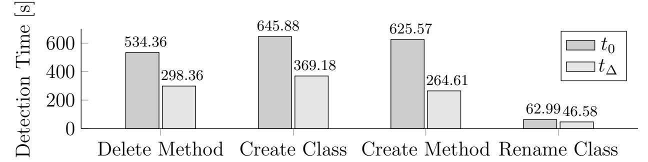

Figure 11: Bar Chart: Speed-up through Incremental Matching for Program Edits

5.3 Threats to Validity

A general threat to internal validity may arise from the selection of subject systems not being representative; to address this issue, we thoroughly investigated related literature for our selection to cover a broad spectrum regarding both size and application domains. In addition, most of the programs have been considered for evaluation purposes by comparable approaches. Another general issue is the NP-completeness of graph isomorphism used by pattern matching. However, in our case, we achieve polynomial complexity by restricting pattern matching using fixed entry points.

Concerning **RQ3.2**, we focus on a small set of self-defined program edits. Although our investigations show that typical evolution steps, not aiming at bug elimination but on structural improvement or program extensions, mainly comprise those kinds of edits, they are naturally limited in scope and are specific to the particular program. However, those edits constitute the most general building blocks of frequent evolution steps and, therefore, our experiments can be assumed to properly simulate evolution-related phenomena occurring in real-life evolving systems. Nevertheless, as part of future work, we plan to further investigate continuous design-flaw detection scenarios by emulating entire version histories available in real-life software repositories.

One major external threat to validity of our design-flaw detection experiments is the manifold possible interpretation of code smells and anti-patterns, resulting in non-comparable detection results, especially for **RQ2**. The standard literature on the description of code smells and anti-patterns largely comprise informal, prose-style circumscriptions of symptoms [17, 8, 13]. However, there is no generally accepted formalization of even the most widely studied design flaws so far. As a consequence, a missing unification of design-flaw descriptions thus leads to each detection tool having its own underlying design-flaw interpretation. Thus, when comparing design-flaw detection approaches, it is not the approaches competing but some (implicit or explicit) configurations of them. In our experiments, this has the most severe effect on the precision results answering **RQ2**, but it also threatens the validity of any performance measurement (**RQ1**, **RQ3**). A further threat for the results of **RQ2** is the lack of publicly available, curated oracles for anti-pattern detection. The only oracle available to us has been initiated for evaluating the BDTEX approach [18] and

has a limited scope. Our own experiences with our proposed technique have shown that our rule language is expressive enough to support a wide range of possible interpretations and adjustments of design-flaw characterizations.

Validity of the results for **RQ1** may be threatened by the lack of data for comparison with similar tools. Although some approaches such as DECOR [9] provide execution times, it is not possible to identify and distinguish the different phases (cf. (1)-(3) in Sec. 5.1) for comparison. Moreover, the available results are not reproducible due to missing information on measurement setup. In addition, they do not provide detailed information about their underlying program representation and its initialization.

A further external threat to validity concerning scalability (**RQ1** and **RQ3.1**) may arise from the choice of SourceMeter to calculate software metrics externally. Thereby, we do not have influence on the efficiency of metrics calculation and we did not evaluate further alternatives so far.

6. RELATED WORK

In this section, we categorize various design-flaw detection approaches based on the applied technique(s) and analyze the similarities and differences to our approach.

Metric-Based. A common method to detect code smells and anti-patterns is by means of software metrics. Simon et al. define a generic distance measure that can be applied to identify anomalies inducing certain refactorings in an underlying language [31]. Mäntylä makes use of atomic metrics to evaluate their applicability for code smell detection compared to human intuition, concluding that metric-based detection often contradicts human perception [13]. Munro proposes to capture informal code smell descriptions by means of a set of metrics and to identify possible occurrences in Java programs based on those metrics [10]. Compared to our approach, all of the above use rather simple metrics that are limited in their capability to detect code smells. Moreover, they use static thresholds for their decision process, whereas we derive relative thresholds from system properties.

More advanced approaches are proposed by Marinescu [11] and Kessentini et al. [32], respectively. Marinescu goes beyond simple metrics by applying different kind of filters and composition rules and using relative thresholds for metric values, thus achieving a better precision. Kessentini et al. formulate the detection as a combinatorial optimization problem over a set of metrics, based on previously evaluated examples and genetic algorithms to optimize smell descriptions. The main difference to our approach is that we additionally use structural information and provide a systematic methodology how code smells are combined to anti-patterns, thus, providing reuse opportunities for detected code smells.

An inverse approach is taken by Kessentini et al. [33], who measure the defectiveness of the software in terms of deviation from good coding practice, provided as design patterns. In a similar way, O'Keeffe et al. propose to formulate design problems and subsequent program refactoring as a metric-based optimization problem [34]. However, these approaches rather aim at directly performing improvement actions instead of detecting and annotating design flaws in the program for dedicated maintenance activities.

Machine Learning. Some detection methods employ machine learning techniques to detect code smells and anti-patterns. Khom et al. use a Bayesian approach to detect code smells [18]. However, as input for their approach, they still need the result of some code smell detection tool (here: DECOR [9]). Similarly, Fontana et al. propose a large-scale case study on common code smells and anti-patterns using different supervised ML techniques [16]. Finally, Maiga et al. propose an approach based on support vector machines (SVM) that also takes user feedback into account [35]. While all of these approaches may improve the detection results, we consider them as complementary techniques as they rely on external design-flaw data. Also, these approaches do not take structural properties into account.

Advanced Techniques. Some approaches exploit additional information such as program structures, represented by more precise program representations such as AST or PDG. A variety of such techniques have been proposed to detect code clones, a common smell in evolving systems. In particular, these techniques are token-based [36], AST-based [37], PDG-based [38], or use static analysis [39]. However, as they only focus on code clones, these approaches capture only a very limited subset of smells.

An approach that is more related to ours has been proposed by Van Emden et al. [14], who use a model-based approach to describe program structure and code smell characteristics. However, in contrast to our approach, they do not provide incremental detection for evolving systems. Moha et al. [9] propose a rule-based detection approach. Similar to us, they provide a taxonomy of code smells and anti-patterns and how they are related. Based on rules, they specify properties that must hold for a smell to be detected, including various metrics. Although similar from several viewpoints, their approach does not support detection information reuse.

Tools. Beyond the aforementioned approaches, several tools exist, such as CodeSonar, SourceMeter [40], PMD [41], or FindBugs [42, 41] that provide a rich set of analysis facilities, such as static analysis, software metrics, or fault analysis. While these analyses might provide valuable data (in fact, we are using SourceMeter for metrics), none of these tools provide comprehensive design-flaw detection.

7. CONCLUSION AND FUTURE WORK

We presented a comprehensive methodology for effectively and efficiently detecting design flaws in continuously evolving object-oriented programs. The approach utilizes an abstract program model to define detection rules, integrating different kinds of design-flaw symptoms. By applying incremental multi-pattern matching, the approach allows for systematically reusing detection information in order to reduce detection efforts. Our evaluation results further show scalability to real-size programs and high detection accuracy. As a future work, we plan to conduct further long-term experiments on continuously evolving systems, based on program edits derived from version histories of real-life software repositories. We also plan to extend our pattern language with logical constructs such as 'or', enlarge the catalog of design flaws using our framework and to conduct user studies to gain a better understanding about the impact of design flaws on software maintenance and evolution.

8. ACKNOWLEDGEMENTS

This work has been supported by the German Research Foundation (DFG) in the Priority Programme SPP 1593: Design For Future – Managed Software Evolution (LO 2198/2-1, JU 2734/2-1).

9. REFERENCES

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.

[2] D. L. Parnas, "Software Aging," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 1994, pp. 279–287.

[3] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An Empirical Study of the Impact of two Antipatterns, Blob and Spaghetti Code, on Program Comprehension," in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2011, pp. 181–190.

[4] D. Sjoberg, A. Yamashita, B. C. D. Anda, A. Mockus, T. Dyba *et al.*, "Quantifying the Effect of Code Smells on Maintenance Effort," *TSE*, vol. 39, no. 8, pp. 1144–1156, 2013.

[5] S. Olbrich, D. Cruzes, and D. I. Sjoberg, "Are All Code Smells Harmful? A Study of God Classes and Brain Classes in the Evolution of Three Open Source Systems," in *Proceedings of the International Conference on Software Maintenance (ICSM)*, 2010, pp. 1–10.

[6] W. Opdyke, "Refactoring Object-Oriented Frameworks," Ph.D. dissertation, University of Illinois, 1992.

[7] D. B. Roberts, "Practical Analysis for Refactoring," Ph.D. dissertation, University of Illinois, 1999.

[8] R. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[9] 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 Transactions on Software Engineering (TSE)*, vol. 36, no. 1, pp. 20–36, 2010.

[10] M. J. Munro, "Product Metrics for Automatic Identification of Bad Smell Design Problems in Java Source-Code," in *Proceedings of the International Software Metrics Symposium (METRICS)*. IEEE, 2005, pp. 15–15.

[11] R. Marinescu, "Detection Strategies: Metrics-based Rules for Detecting Design Flaws," in *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 2004, pp. 350–359.

[12] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and Why Your Code Starts to Smell Bad," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Press, 2015, pp. 403–414.

[13] M. Mäntylä, "Bad Smells in Software – A Taxonomy and an Empirical Study," Master's thesis, Helsinki University of Technology, Finland, 2003.

[14] E. Van Emden and L. Moonen, "Java Quality Assurance by Detecting Code Smells," in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. IEEE, 2002, pp. 97–106.

[15] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A Bayesian Approach for the Detection of Code and Design Smells," in *Proceedings of the International Conference on Quality Software (QSIC)*. IEEE, 2009, pp. 305–314.

[16] F. Fontana, M. Zanoni, A. Marino, and M. Mantyla, "Code Smell Detection: Towards a Machine Learning-Based Approach," in *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 2013, pp. 396–399.

[17] W. J. Brown, R. C. Malveau, H. W. McCormick, III, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., 1998.

[18] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "BDTEX: A GQM-based Bayesian Approach for the Detection of Antipatterns," *Journal of Systems and Software*, vol. 84, no. 4, pp. 559–572, 2011.

[19] N. V. Eetvelde and D. Janssens, "A Hierarchical Program Representation for Refactoring," *Electronic Notes in Theoretical Computer Science*, vol. 82, no. 7, pp. 91–104, 2003.

[20] T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens, "Formalizing Refactorings with Graph Transformations," *Journal of Software Maintenance and Evolution*, vol. 17, no. 4, pp. 247–276, 2005.

[21] L. C. Briand, J. W. Daly, and J. Wüst, "A Unified Framework for Cohesion Measurement in Object-Oriented Systems," *Empirical Software Engineering*, vol. 3, no. 1, pp. 65–117, 1998.

[22] L. C. Briand, J. W. Daly, and J. K. Wust, "A Unified Framework for Coupling Measurement in Object-Oriented Systems," *IEEE Transactions on Software Engineering (TSE)*, vol. 25, no. 1, pp. 91–121, 1999.

[23] B. Henderson-Sellers, *Object-oriented Metrics: Measures of Complexity*. Prentice-Hall, Inc., 1996.

[24] A. J. Riel, *Object-Oriented Design Heuristics*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[25] J. Bosch, "Software Architecture: The Next Step," in *European Workshop on Software Architecture*, 2004, pp. 194–199.

[26] S. Grant, "With Friends like These..." in *Marvel Fanfare Vol. 1 #7*. New York, NY, USA: Marvel, 1983.

[27] A. Schürr, "Specification of Graph Translators with Triple Graph Grammars," in *Proceedings of the International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*. Springer, 1995, pp. 151–163.

[28] Z. Ujhelyi, A. Horváth, D. Varró, N. I. Csiszár, G. Szőke, L. Vidács, and R. Ferenc, "Anti-pattern Detection with Model Queries: A Comparison of Approaches," in *Proceedings of the Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*. IEEE, 2014, pp. 293–302.

[29] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies," in *Asia Pacific Software Engineering Conference*, 2010, pp. 336–345.

[30] J. Davis and M. Goadrich, "The Relationship Between Precision-Recall and ROC Curves," in *Proceedings of the 23rd International Conference on Machine Learning*, ACM. ACM, 2006, pp. 233–240.

[31] F. Simon, F. Steinbruckner, and C. Lewerentz, "Metrics Based Refactoring," in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2001, pp. 30–38.

[32] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni, "Design Defects Detection and Correction by Example," in *Proceedings of the International Conference on Program Comprehension (ICPC)*. IEEE, 2011, pp. 81–90.

[33] M. Kessentini, S. Vaucher, and H. Sahraoui, "Deviance from Perfection is a Better Criterion Than Closeness to Evil when Identifying Risky Code," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. New York, NY, USA: ACM, 2010, pp. 113–122.

[34] M. O'Keeffe and M. Ó Cinnéide, "Search-based Refactoring: an Empirical Study," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 5, pp. 345–364, 2008.

[35] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y.-G. Guéhéneuc, and E. Aimeur, "SMURF: A SVM-based Incremental Anti-pattern Detection Approach," in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. IEEE, 2012, pp. 466–475.

[36] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code," *IEEE Transactions on Software Engineering (TSE)*, vol. 28, no. 7, pp. 654–670, 2002.

[37] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier, "Clone Detection using Abstract Syntax Trees," in *Proceedings of the International Conference on Software Maintenance (ICSM)*, 1998, pp. 368–377.

[38] J. Krinke, "Identifying Similar Code with Program Dependence Graphs," in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. IEEE, 2001, pp. 301–309.

[39] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," in *International Symposium on Static Analysis*. Springer, 2001, pp. 40–56.

[40] R. Ferenc, L. Langó, I. Siket, T. Gyimóthy, and T. Bakota, "Source Meter Sonar Qube Plug-in," in *Proceedings of the Working Conference on Source Code Manipulation and Analysis (SCAM)*. IEEE, 2014, pp. 77–82.

[41] I. F. Darwin, *Checking Java*. O'Reilly Media, 2007.

[42] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," in *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, 2004, pp. 92–106.