Towards Triaging Code-Smell Candidates via Runtime Scenarios and Method-Call Dependencies

Thorsten Haendler
WU Vienna, Austria
thorsten.haendler@wu.ac.at

Stefan Sobernig
WU Vienna, Austria
stefan.sobernig@wu.ac.at

Mark Strembeck
WU Vienna, Austria
Secure Business Austria
Complexity Science Hub Vienna
mark.strembeck@wu.ac.at

ABSTRACT

Managing technical debt includes the detection and assessment of debt at the code and design levels (such as bad smells). Existing approaches and tools for smell detection primarily use static program data for decision support. While a static analysis allows for identifying smell candidates without executing and instrumenting the system, such approaches also come with the risk of missing candidates or of producing false positives. Moreover, smell candidates might result from a deliberate design decision (e.g., of applying a particular design pattern). Such risks and the general ambivalence of smell detection require a manual design and/or code inspection for reviewing all alleged smells.

In this paper, we propose an approach to obtain tailorable design documentation for object-oriented systems based on runtime tests. In particular, the approach supports a tool-supported triaging of code-smell candidates. We use runtime scenario tests to extract execution traces. Based on these execution traces, different (automatically derived) model perspectives on method-call dependencies (e.g., dependency structure matrices, DSMs; UML2 sequence diagrams) are then used as decision support for assessing smell candidates. Our approach is implemented as part of the KaleidoScope tool which is publicly available for download.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**; **Software design tradeoffs**; *Dynamic analysis*; *Object oriented development*;

KEYWORDS

design documentation, code smell, technical debt, execution trace, scenario-based testing, software behavior, decision support, dependency structure matrix, Unified Modeling Language (UML2)

XP '17 Workshops, May 22-26, 2017, Cologne, Germany

ACM ISBN 978-1-4503-5264-2/17/05…$15.00
https://doi.org/10.1145/3120459.3120468

1 INTRODUCTION

In general, Technical Debt (TD) can be introduced accidentally by hasty and ill-prepared design decisions, or intentionally in order to balance typical forces such as budget or time constraints. For example, TD can be caused by project-schedule pressure or a lack of developers' skills [54]. Debt items at the design level are typically caused by code smells (see, e.g., [13]) or violations to known design patterns (see, e.g., [15]), which negatively affect the code quality regarding its maintainability and evolvability.

Managing TD includes the tasks of detecting, evaluating and repaying debt [1]. In particular, the detection and evaluation of code smells is a complex task, because many types of smells are only hard to identify by manually inspecting source code and/or design documentation. Moreover, the size of a code base and/or the lack of design documentation can render a manual inspection even more difficult. Thus, automation and decision support for smell-detection as well as design/code critique tools have been proposed to simplify corresponding inspection tasks (for an overview, see, e.g., [10]).

Popular tools for smell detection [9, 37] and for design critique [3, 19, 55] provide visualizations of different dependencies or metrics as decision support. Such analysis tools usually leverage static program analysis techniques. While a static analysis allows for finding smell candidates without executing the corresponding software program [53], a static analysis also comes with the risk of missing smell candidates or of producing false positives. For example, smell candidates might result from a deliberate design decision (e.g., of applying a particular design pattern). This risk incurs time and effort to manually review each candidate, which may, again, exceed the time and effort that has been saved by the automated analysis (see, e.g., [36]).

To reduce this review effort for smell candidates, we present an approach that provides the following contributions:

- We use scenario-based runtime tests to extract execution traces of the system under test. Based on these execution traces, different views on method-call dependencies are derived as decision support (e.g., dependency structure matrices, DSMs; UML2 sequence diagrams).
- We implemented a software tool called *KaleidoScope* to obtain tailorable object-oriented design documentation based on runtime tests to support software engineers in reviewing smell candidates.

Method-Call Dependencies. Code-smell candidates can be identified by quantifiable phenomena at the level of code structures and behaviors. These include dependencies between code units (such

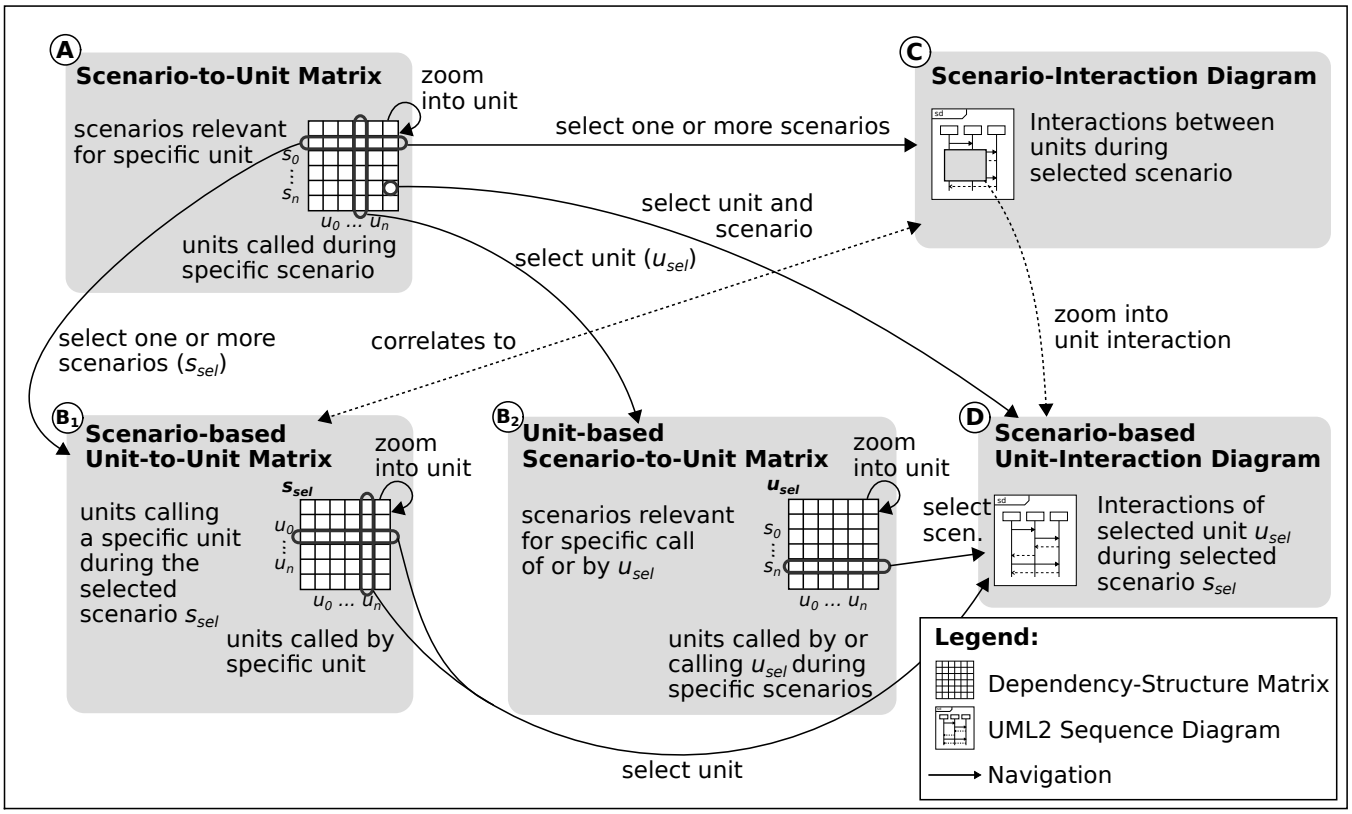


Figure 1: Scenario & runtime perspectives on method-call dependencies for triaging code-smell candidates.

as dependencies between classes or methods). Data on call dependencies between methods can be gathered by different means and at different binding times. Current approaches for smell detection and/or refactoring particularly harvest method dependencies based on static program analysis [29, 41, 43]. In this paper, we suggest the use of runtime scenario tests to record concrete method calls (messages). This way, *actual* (runtime) method-call dependencies can be investigated and related to each other in the context of a test-suite structure.

Scenarios & Scenario-based Testing. Scenarios are a popular means for modeling intended or actual behavior of software systems on different abstraction levels by describing action and event sequences (see, e.g., [2, 20, 21]). System scenarios can, for example, be modeled via UML2 sequence diagrams [35]. Scenarios can also help in selecting and specifying relevant tests (see, e.g., [31, 40]). For instance, in Behavior-driven Development [33], scenarios are used for analyzing the behavior of software components. In addition, scenarios are applied for reviewing and investigating different aspects of software design and architecture (see, e.g., [24, 26]).

Paper structure. The remainder of this paper is structured as follows. Section 2 gives an overview of our approach. In particular, it motivates the relevance of method-call dependencies for smell detection, explains challenges in the evaluation of selected smell types, and presents perspectives and navigation paths based on scenarios and method-call dependencies. In Sect. 3, we introduce our *KaleidoScope* tool. Here, we describe how the proposed model perspectives are derived and explain how *KaleidoScope* can be applied for assessing corresponding smell candidates. In Sect. 4, we discuss further assessment options and reflect on the approach's limitations. Section 5 discusses related work and Sect. 6 concludes the paper.

2 APPROACH

Figure 1 depicts the main artifacts produced in our approach. The data obtained from scenario-based test-execution traces (e.g. actual call dependencies triggered by executing scenarios, and the contextual data) can be aggregated, filtered, and presented in a number of ways. For example, the approach allows for the generation of different dependency structure matrices (DSMs; [8]). The DSMs Ⓐ, Ⓑ$_1$ and Ⓑ$_2$ provide the correlations between scenarios and system units in different tailorable ways. For example, a unit can be a package, a class, or a method. By selecting a specific unit, the dependencies of the corresponding sub-units are depicted (*zoom into unit*). In particular, the matrices display method-call frequencies (e.g., in terms of heatmaps). Among other things, it can be distinguished between the amount of calls in total or the amount of different sub-units called.

The scenario-to-unit matrix in Ⓐ shows whether and how specific units are covered by test scenarios. Different navigation options exist for this perspective: For instance, by selecting one or multiple

scenarios, the (active/passive) call dependencies between specific system units can be investigated in the scenario-based unit-to-unit matrix (in Ⓑ1). By selecting a system unit, the method calls from or to the selected unit and their target or source units respectively as well as the corresponding scenarios are displayed in Ⓑ2.

Moreover, the dependencies between units in terms of mutual method calls are displayed via UML2 sequence diagrams [35]. For example, the sequence diagram in Ⓒ reflects the interactions during a selected scenario (or set of scenarios). The sequence diagram in Ⓓ reports on the interactions of a selected unit (with other units) during a selected scenario (detail view of Ⓒ by zooming into the unit interaction). In addition to the DSMs, these sequence diagrams include behavioral details of methods; e.g., in terms of the order of method calls (relative/absolute order), transitive dependencies to other methods, the calling method(s) as well as passed arguments and return values.

2.1 Triaging Procedure

Smell triage is the systematic process of determining a smell candidate's condition as either `false positive`, `intentional`, or bad smell as well as its priority for refactoring planning. This is akin to systematic forms of bug triage (see, e.g., [6]). The triage of a candidate includes two basic decision steps (see Fig. 2).

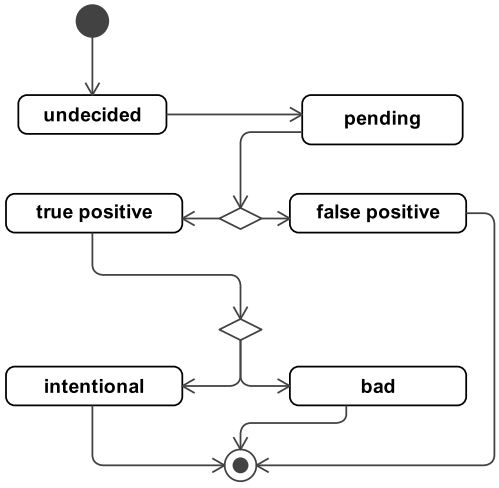


Figure 2: Key candidate states during a triage.

At *first*, in a symptom-based investigation and assessment, the smell candidates are identified and assessed on basis of different phenomena (e.g., method-call dependencies and aggregates thereof). This evaluation can be assisted by smell-detection tools and is about filtering out `false positives` based on candidate sets obtained from these detection tools. In a *second* step, a smell candidate (i.e., now a `true positive` in terms of the detection technique) is re-assessed in light of a) its structural and behavioral context and of b) the related design decisions such as documented or recovered design rationale. After this, a smell candidate has been rated either as an `intentional` or bad smell. In case of a bad smell, finally the effort and change impact of potential refactorings on the program and the test suite are estimated for planning and prioritizing refactoring steps.

2.2 Triaging Smell Candidates

Subsequently, we focus on a set of code smells, which manifest in terms of dependencies between units (e.g., packages, classes, methods) via method calls. First, we address functionally similar methods (as a specific kind of DuplicateCode). Second, we demonstrate the usability for a group of smells concerning modularization problems (as kinds of a Broken Modularization). For each smell, we first describe relevance, symptoms, challenges in evaluating the smell as well as common refactoring options. We close by detailing the decision support required by the smells.

Duplicate Code. This smell occurs if two or more pieces of code show an identical or similar structure or behavior (see, e.g., [13, 49]). Duplicate code in general is problematic, since its redundancy produces problems in maintenance or evolution activities. A set of duplicate methods is also an indicator for a DuplicateAbstraction design smell.

Functionally similar methods are introduced, for example, in case the same or similar functionality is developed independently by different developers. Thus, such code clones (type-4 clones [39]) seldom show syntactic similarity. Wagner et al. [52] figured out that less than 1% of clones show full similarity, and less than 12% only partial similarity. Different approaches for identifying functionally similar code exist (see, e.g., [14, 22, 47, 48]). In particular, these approaches use different criteria for evaluating behavioral similarity of methods: black box (similar input/output behavior), AST-based or PDG-based approaches (similar internal behavior). A variation of the latter is assessing the dependencies to other methods, i.e. the set of methods called by or calling the candidate methods. This way, smell candidates are methods calling the identical or largely overlapping sets of other methods.

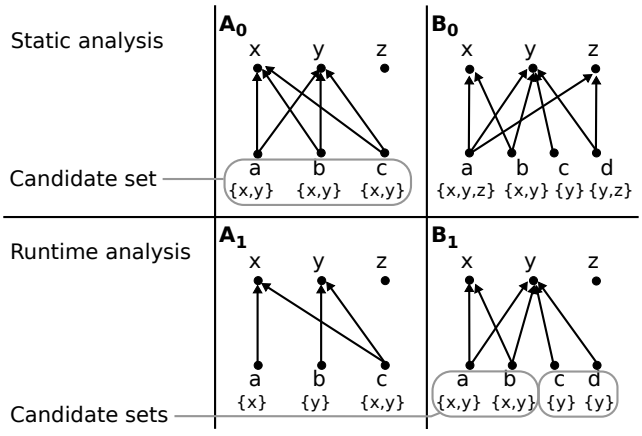


Figure 3: Towards spotting candidates for functionally similar methods.

Figure 3 depicts the challenges in evaluating functionally similar methods via method-call dependencies. Two exemplary method-dependency graphs are depicted (A and B). Not all method dependencies identified using static analysis (see A_0 and B_0) are likely to appear during system execution. In particular, parametrization and dependency injection may render a different picture of actual dependencies (see A_1 and B_1). From this angle, while A_0 shows an identical set of called methods {x,y}, since A_1 reveals that during execution the called sets are not identical: {x},{y}, and {x,y}. In contrast, B_0 represents a false negative, since in B_1 two pairs of identical call sets ({y} and {x,y}) are identified (hidden candidate). This way, both A_0 and B_1 can serve as starting point for the triage procedure.

Besides the called sets, other challenges occur in comparing the behavior of the candidate methods, such as the order of actually called methods, the transitive dependencies (to methods called by the targeted methods), the given usage context of the methods (usage scenario, calling methods), and the actually passed arguments and return values.

Broken Modularization. Modularization is the principle of creating cohesive and loosely coupled modules (e.g, packages, classes, methods). Core enabling practices to achieve modularization include (see, e.g., [49]):

- localize related data and methods
- decompose abstractions
- avoid cyclic dependencies (see, e.g., [28])
- limit dependencies between modules

Modularization smells violate one or more of these principles, often caused by misplaced methods or fields. For example, the FeatureEnvy smell contradicts these principles.[1] Restoring the principles above demands for refactoring these smells by performing one or multiple MoveMethod or MoveField factorizations.

In particular, **FeatureEnvy** manifests as a method that seems more interested in a class other than the one it actually is defined in [13]. In other words, a FeatureEnvy candidate can be identified by measuring the amount of called foreign methods or fields (owned by other classes) in relation to self calls.

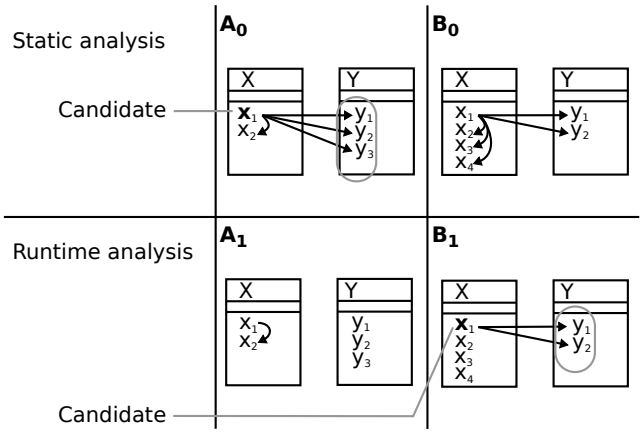


Figure 4: Towards spotting candidates for FeatureEnvy.

[1]Other relevant modularization smells are, for example: CyclicDependency, DataClumps, DataClass, MessageChain, and DispergedCoupling.

Figure 4 depicts the challenges in evaluating a FeatureEnvy candidate using method-call dependencies for two given examples A and B. Not all dependencies identified in static program data (A_0 and B_0) become manifest during system execution (A_1 and B_1). While method x_1 in A_0 contains 3 inter-class and 1 intra-class call definitions and, thus, can be considered as a smell candidate, it turns out to be not significant for the scope of selected system scenarios, since only the class-internal method call is actually triggered (see A_1). In turn, method x_1 in B_0 provides static references to three class-internal and two external methods. Depending on a given threshold, it might not be identified as a smell candidate. In contrast, during scenario execution, only the two inter-class calls can be tracked. This result suggests for further investigation of x_1 (as a hidden smell candidate).

Decision Support. The discussed examples illustrate the challenges for assessing smell candidates based on symptoms in method-call dependencies. As pointed out, by applying runtime scenarios, hidden candidates can be identified and the relevance of candidates (from a scenario perspective) can be investigated. In support of investigating dependencies between different program units (e.g., classes, methods), dependency structure matrices (DSMs) are a popular means (see, e.g., [8]). Essential for scenario-driven smell assessment are especially the scenario coverage of units as well as the call relations between different (kinds of) units during scenario execution (see Fig. 1). The second step in smell triage includes the re-assessment of given candidates based on the structural and behavioral design context of candidates for deciding whether the candidate is actually a `bad` or an `intentional` smell (e.g., as result of applying a particular design pattern). Design models such as UML2 sequence diagrams derived from executing runtime scenarios can assist in reviewing the method-call interactions. Our approach allows for tailoring the scope of the derived diagrams to narrow down on the relevant behavioral context of smell candidates (see Fig. 1).

3 TOOL SUPPORT

As an early proof of concept, we extended our existing tool *KaleidoScope*[2] [17] to support the proposed approach to scenario-based smell triage. *KaleidoScope* allows for recording test executions and the guided derivation of scenario-based artifacts from the execution-trace data (see Fig. 5). In support of our triage approach, *KaleidoScope* provides dependency structure matrices (DSMs) and UML2 sequence diagrams for different scopes (see Fig. 1).

3.1 Process Overview

KaleidoScope instruments a given test framework (e.g., *STORM* [45] or the BDD framework TclSpec [42]) to collect test-execution trace data and stores it in a corresponding trace model (see ① and ② in Fig. 5).[3] Driven by the developer's view selection (e.g., by selecting

[2]*KaleidoScope* is publicly available for download at [16]. For details on the derivation process, also see [17, 18].

[3]Given that a developer is interested in specific parts of the test suite only (e.g., a set of scenarios), she can select this subset for test execution. This way, the cost of instrumenting the test execution and processing the extracted trace data can be reduced.

scenario, unit and report type) ③, it aggregates the extracted test-execution trace data to corresponding matrices and creates scoped UML2 sequence diagrams ④.

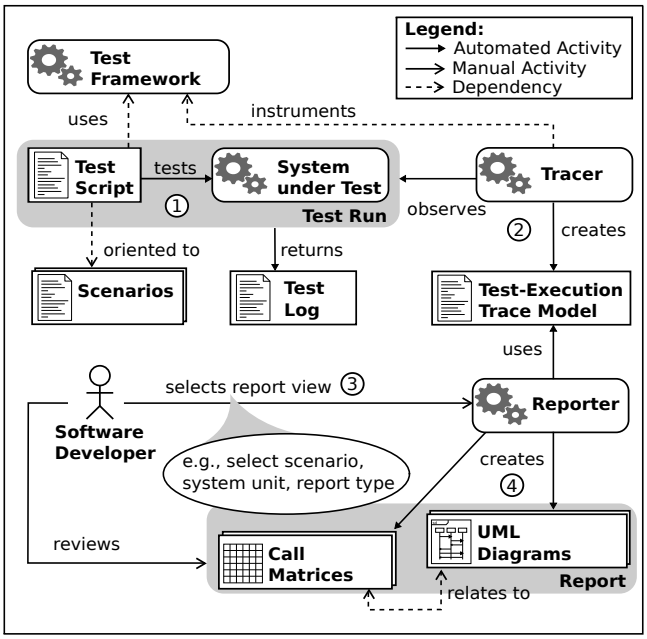


Figure 5: Conceptual overview of *KaleidoScope*.

In particular, *KaleidoScope* is implemented in NX/Tcl [32]. XMI test-execution trace models are extracted by using NX/Tcl introspection techniques. A set of Query/View/Transformation operational (QVTo) [34] mappings then transcripts the pair of trace model and view model (reflecting the user's view configuration) into scoped UML2 interaction models [35] (also handled in XMI). By integrating the *Quick Sequence Diagram* editor [44], the models are then automatically visualized as UML2 sequence diagrams. The derived dependency structure matrices can be processed and visualized using R.

3.2 Triage Walk-Throughs

In what follows, we want to briefly walk the reader through triaging the two smells exemplified in Sect. 2.2 (i.e., *functionally similar methods* and FeatureEnvy) using *KaleidoScope*. The underlying code examples used in these walk throughs are available from the resources section at [16].

Functionally Similar Methods. The activity diagram in Fig. 6 depicts a triage of a candidate for functionally similar methods. The triage steps are the following:

(1) *Analyze inter-method call dependencies*: The smell candidates are identified by assessing the methods via the overlapping sets of called methods during scenario-test execution. For example, an inter-method matrix (see Ⓑ1 in Fig. 1) shows three methods (candidate set) with an overlapping set of multiple called methods.

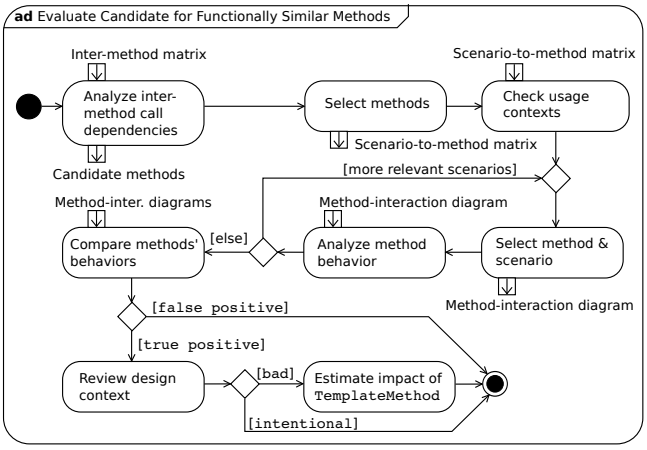


Figure 6: Decision process for assessing candidates for *functionally similar methods* using *KaleidoScope*.

(2) *Select methods*: In particular, the two candidate methods `getBillableAmount()` and `calculateAmount()` owned by two different classes call the identical set of methods. The user selects these two methods and *KaleidoScope* generates a scenario-to-method matrix (Ⓐ in Fig. 1), which reports on the scenario coverage of the corresponding methods.

(3) *Check the usage context*: Identifying the covering scenarios is a prerequisite for selecting the corresponding interaction diagram. Moreover, the scenario coverage can support in assessing the behavioral context of the candidate methods.

(4) *Select scenario*: After selecting a specific method and scenario, *KaleidoScope* automatically creates a corresponding scenario-based method-interaction diagram (Ⓓ in Fig. 1).

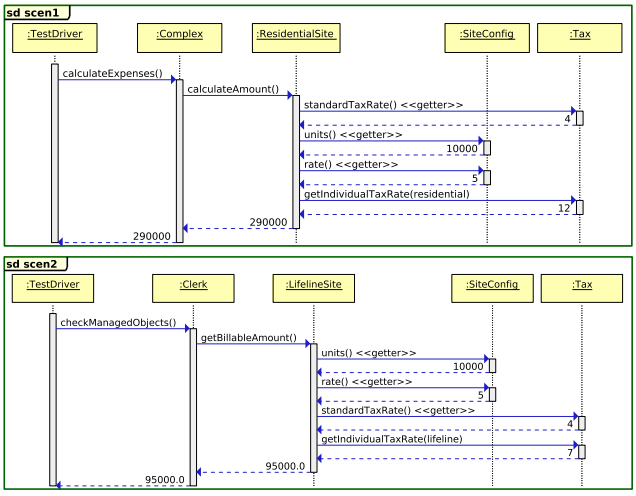


Figure 7: The generated scenario-based method-interaction diagrams help inspect on common and varying details of the candidate method `getBillableAmount()` of class `LifelineSite` and the candidate method `calculcateAmount()` of class `ResidentialSite`.

(5) *Analyze method behavior & Compare methods' behaviors*: The resulting interaction diagrams allow for inspecting behavioral details of the candidate methods (for the given example, see Fig. 7). These details include the orders of called methods relative to each other or in absolute terms, transitive method calls (chaining), the identity sets of calling and called objects, the calling methods, as well as the input/output behavior in terms of passed arguments and return values.

In this example, the preliminary triage decision is `true positive`. A first symptom is the identical set of called methods. Moreover, the set of participating classes (`SiteConfig` and `Tax`) is identical. The methods called by the candidates are partly in a relative order, i.e. `units()` is called before `rate()` and `standardTaxRate()` before `getIndividualTaxRate()`. Moreover three called methods show an identical input/output behavior, only `getIndividualTaxRate` differs. However, for deciding whether the smell is `bad` or `intentional`, further investigation is necessary (e.g. by reviewing documented or recovered design-rationale documentation; see *Review design context* in Fig. 6). So far, the two methods show a high level of behavioral similarity and are candidates for a `TemplateMethod` refactoring. For this reason, the relation between the classes (`Complex` and `Clerk`) and the behavioral context of the calling methods (`calculateExpenses()` and `checkManagedObjects()`) triggering the candidate methods need to be reviewed (see *Estimate impact of* TemplateMethod in Fig. 6). For a discussion on the investigation options provided by *KaleidoScope* regarding the refactoring impact, see Sect. 4.

FeatureEnvy. The activity diagram in Fig. 8 depicts a triage of a FeatureEnvy candidate using *KaleidoScope*. The steps involved are the following:

(1) *Analyze inter-class coupling*: In order to identify candidate classes, the call dependencies between classes are assessed. The scenario-based inter-class matrix in Table 1a) depicts the corresponding call dependencies between source and target classes. In particular, it shows the amount of different methods called during scenario execution (Ⓑ1 in Fig. 1). This way, it can be observed that class C triggers calls to more methods of other classes than to methods of the own (4 vs. 3, see Table 1a)), which can be considered suspicious ("smelly") behavior.

(2) *Select source class(es) & analyze method-to-class coupling*: In order to identify the corresponding methods, the user can select the candidate source-class C. *KaleidoScope* then automatically creates a corresponding method-to-class matrix. By inspecting the matrix in Table 1b), the candidate methods c2 and c3 can be identified, i.e., methods triggering more inter-class method calls than calls to methods of the own class (c2: 3 vs. 1 and c3: 1 vs. 0).

(3) *Select target class(es)*: To understand which methods are used by the candidate methods, the corresponding target classes (A and B) are selected by the user.

(4) *Analyze inter-method coupling*: The generated scneario-based inter-method matrix in Table 1c) finally reports on the call relations between the calling methods of class C and the called methods of classes A and B. In particular, in contrast to the two other matrices, it reflects the method-call frequencies.

In this example, the preliminary triage decision is `true positive`. Especially for method c2 during scenario execution, more call dependencies to methods of class B are identified than to features of the own class C. The high level of coupling between c2 and class B (in terms of multiple used methods, see Table 1c)) might be a first indicator for a bad smell. In order to decide whether the smell is `intentional` or bad, the related behavioral context needs to be reviewed (see *Analyze method behavior* in Fig. 8).

Our approach and *KaleidoScope* open up further investigation options regarding the behavioral candidate context and the change impact of refactoring options (such as MoveMethod for the FeatureEnvy candidate). Please see the discussion in Sect. 4.

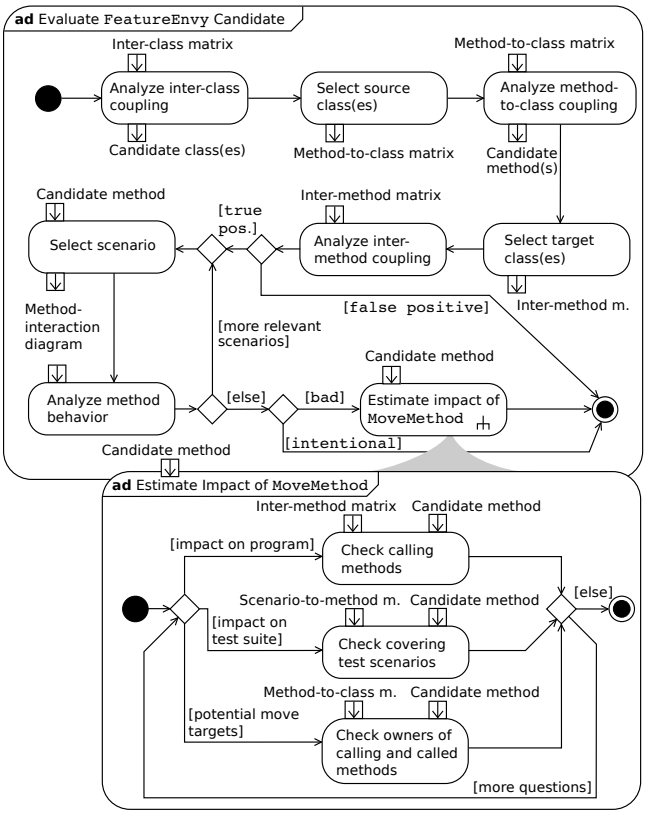


Figure 8: Decision process for assessing FeatureEnvy candidates (top) and for estimating the impact of MoveMethod refactorings (bottom) using *KaleidoScope*.

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array}\begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{pmatrix} 12 & \underline{2} & 0 & 0 \\ 0 & 17 & 0 & 0 \\ \underline{1} & \underline{3} & 3 & 0 \\ 0 & 0 & 0 & 9 \end{pmatrix} \end{array} \quad \text{(a)}$$

$$\begin{array}{c} \\ c1 \\ c2 \\ c3 \end{array}\begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{pmatrix} 0 & 0 & 2 & 0 \\ 0 & \underline{3} & 1 & 0 \\ \underline{1} & 0 & 0 & 0 \end{pmatrix} \end{array} \quad \text{(b)}$$

$$\begin{array}{c} \\ c1 \\ c2 \\ c3 \end{array}\begin{array}{c} \begin{array}{cccccc} a1 & a2 & b1 & b2 & b3 & b4 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \underline{1} & \underline{4} & 0 & \underline{2} \\ \underline{3} & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \quad \text{(c)}$$

Table 1: Exemplary dependency structure matrices (DSMs). The inter-class matrix in a) relates source classes (left) to target classes (top) and reports on the amount of different methods called during scenario execution. The inter-method matrix in c) reports on the method-call frequencies. Underlined numbers indicate inter-class method calls.

4 DISCUSSION

In this workshop paper, we reported on work-in-progess and thus limit ourselves to illustrative examples of smells, for which method-call dependencies are primary indicators. In particular, the presentation is limited to FeatureEnvy, as a relevant example of a Modularization smell, and *functionally similar methods* (kind of DuplicateCode) because they have been reported as being relevant and representative by empirical research on smells (see, e.g., [7, 23]). In the following, we discuss preliminary results regarding extended triaging questions (such as identifying intentional smells as well as estimating the change-impact of potential refactorings), the assessment of other smell types, and its application to large(r) software systems.

Bad vs. Intentional Smells. Ruling out smell candidates as being true positives with respect to a particular detection technique or tool does not necessarily imply a smell realizing a *bad* smell [11]. On the one hand, evidence from empirical work on smells and their actual impact on maintainability draws a inconclusive picture. On the other hand, design decisions (e.g., applying a certain design pattern) typically incur trade-offs including smelly structure and behavior. These challenges raise the barriers to adopting tool-assisted smell detection techniques, because they often require developers to review the code bases and/or the design specification manually all over again. We feel that our approach and *KaleidoScope* can assist in answering such extended triaging questions that directly depend on a design-decision context.

Consider, for example, that a combination of FeatureEnvy and CyclicDependency is first identified based on dependency structure matrices (DSMs) alone (or signalled by a conventional detection tool). To rule out the possibility of these smell occurrences being (intentional) consequences of applying a variant of the visitor pattern and its double-dispatch protocol [11], developers can investigate the corresponding method-interaction diagrams provided by *KaleidoScope* (see *Analyze method behavior* in Fig. 8). From these diagrams (for an example, see Fig. 9), the double-dispatch protocol (and naming conventions) hinting at visitor may become apparent; avoiding an otherwise more laborious direct code-inspection exercise.

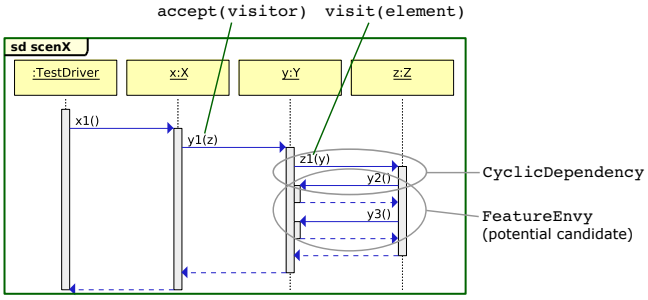


Figure 9: The generated scenario-based unit-interaction diagrams help identify `intentional` smells (e.g., a method-interaction diagram reflecting the double-dispatch protocol of the Visitor design pattern).

Refactoring Impact. The impact of potential refactorings on a system's maintainability and project priorities must be systematically considered in triaging decisions. Our approach provides artifacts as decision support to run a tentative change-impact analysis of possible refactoring options. Consider FeatureEnvy as an example for which one or several MoveMethod refactorings are commonly considered as action items. To prepare a MoveMethod refactoring, using *KaleidoScope*, a developer can tackle the following questions (see also *Estimate Impact of MoveMethod* in the bottom activity in Fig. 8).

- **Impact on program**: *Which calling methods depend on the candidate method to be moved*? The answer can be explored using our approach's inter-method matrix, for example.
- **Impact on test suite**: *Which scenario tests cover the method to be moved*? To prepare an answer, our approach offers the scenario-to-method matrix.
- **Move target**: *Which existing classes are eligible owners of the candidate method to be moved*? Candidate targets are classes coupled to the method, either as owning class of multiple called methods or as owner of methods calling the candidate method. Our approach suggests potential candidates providing method-to-class and class-to-method matrices.

This way, developers are supported in reviewing the effort of intended refactorings, which can help in estimating the costs for repaying the design debt incurred by a given FeatureEnvy instance.

Other Smell Types. We plan to systematize scenario-based triaging options for other smell categories, such as Abstraction, Encapsulation, Hierarchy as well as other Modularization design smells. This will also require us to include additional analysis techniques and program data (e.g., data and subclass dependencies) as well as additional visualization types (e.g., UML class diagrams) into *KaleidoScope* (e.g., to extract and visualize inheritance relations with scenarios as slicing criterion). Thereby, we intend to explore the further potential of using runtime scenarios for identifying and assessing smells. For instance, multiple smell types manifest via the usage context such as MissingAbstraction (a.k.a. DataClumps) or MultifacedAbstraction (a class with multiple responsibilities, similar to GodClass) [49]. The challenge in assessing MultifacedAbstraction candidates is to identify and to distinguish the class's responsibilities. Via the runtime scenarios, involved methods as well as calling and called methods of other classes, different usage contexts can be investigated. Our approach can help identify and assess such candidates. For example, the class-based scenario-to-method matrix (see ⓑ2 in Fig. 1) can be inspected, which reports on the (active/passive) dependencies of a selected candidate class to methods of other classes and the runtime scenarios triggering these calls.

Scalability. To explore our approach's capabilities and performance characteristics, we intend to apply our *KaleidoScope* to large(r) real-world and third-party software systems. We are currently preparing its application to a collection of Graph Product Line implementations (GPL; [27]), as a standard problem on software composition, as well as to the *BusinessActivity framework and runtime engine* [46], which features a comprehensive scenario-based test suite (including 357 test scenarios with about 30k individual

assertions). The matrices in Figs. 10 and 11 depict exemplary preliminary results of applying *KaleidoScope* to the *BusinessActivity framework*.

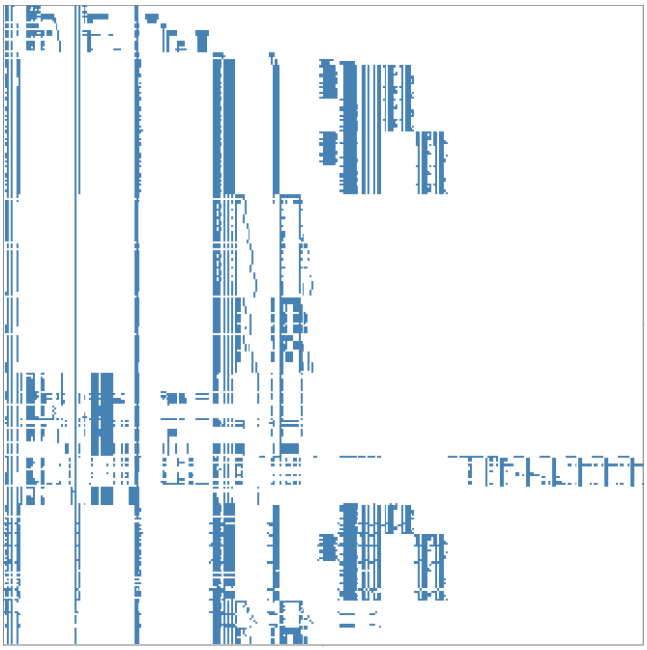
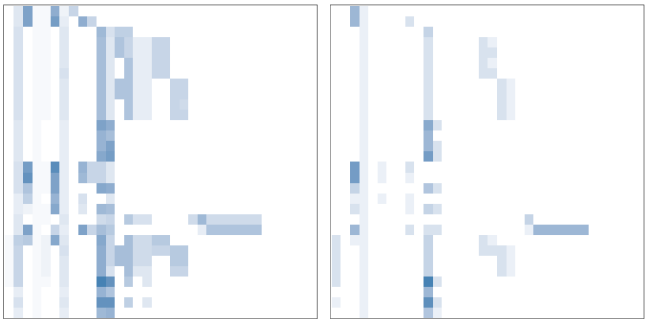


Figure 10: Exemplary generated scenario-to-method matrix reporting on the scenario coverage of methods (y-axis: 357 test scenarios, x-axis: selected methods; called vs. not called) in the *BusinessActivity framework* [46].



(a) (b)

Figure 11: Exemplary generated scenario-to-class matrices (y-axis: selected test scenarios, x-axis: selected classes). Matrix a) reports on the amount of different methods called, and matrix b) reports on the amount of different methods triggering inter-class method calls in the *BusinessActivity framework* [46].

5 RELATED WORK

To the best of our knowledge, data obtained from or visualization of test-based execution traces have not been applied for assessing smell candidates so far. Related work can be roughly divided into two groups: tools and approaches for (1) trace visualization and for (2) code or design critique.

On the one hand, multiple approaches exist for visualizing execution traces (see, e.g., [4, 50, 51]). Only a few approaches address the visualization of test-execution traces by including test information and test structure though. Thereby, the primary purpose of collecting and analyzing test-based execution traces is understanding the system's behavior [5]. Moreover, execution traces are also used for understanding or investigating the system's structure/architecture during runtime [38], e.g., also in terms of architecture slicing [25].

On the other hand, multiple approaches aim to assist in evaluating smells or debt items on different levels such as code smells [30] or architectural debt [12]. Other popular code and design critique tools (such as [3, 19, 55]) offer options to explore a program's design by visualizing different kinds of dependencies, e.g., in terms of dependency graphs or DSMs. Among these approaches, especially those are related that use method dependencies (and dependency metrics) for assessing smells and/or suggesting refactoring options [29, 41, 43]. In general, these approaches and tools mostly apply static-analysis techniques. So far, we are not aware of any smell-evaluation or detection tool that leverages runtime-analysis techniques. Our approach (built on runtime scenarios) complements these static-analysis tools in the sense that identified candidates can be further investigated (e.g. regarding scenario relevance or behavioral design context). Additionally, hidden candidates can be detected by our approach.

6 CONCLUSION

In this paper, we presented an approach for supporting the triage of code-smell candidates by analyzing method-call dependencies extracted from scenario-based runtime tests. Using our *KaleidoScope* tool, the test-execution traces are automatically processed into different model perspectives such as dependency structure matrices (DSMs) and UML2 sequence diagrams. The scenario-based analysis is proposed as a complement to existing critique tools built on static program data. This way, candidates identified by static-analysis tools can be proved as potentially false or not relevant from a runtime-scenario perspective. In turn, candidates can be identified which are not recognizable by applying static analysis tools (hidden candidates). However, the resulting views can also serve as decision support in the evaluation process independent from other tools. Besides assisting in the identification of code-smell candidates (symptom-based identification and assessment), the test-generated views also provide support for answering extended triaging questions regarding aspects of the system context; such as deciding whether a candidate is bad or intentional, or estimating the impact of potential refactorings.

In our future work, we will apply the approach for assessing smell candidates in larger software systems. Moreover, we will explore the approach's further possibilities for assessing other smell types, for identifying intentional smells as well as for refactoring planning. In addition, we plan to investigate, how human users benefit from using the provided perspectives for triaging smell candidates.

REFERENCES

[1] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, and others. 2010. Managing technical debt in software-reliant systems. In *WS Proc. FSE/SDP'10*. ACM, 47–52. DOI: http://dx.doi.org/10.1145/1882362.1882373

[2] John M. Carroll. 2000. Five reasons for scenario-based design. *Interact. Comput.* 13, 1 (2000), 43–60. DOI: http://dx.doi.org/10.1016/S0953-5438(00)00023-0

[3] CoderGears. 2017. JArchitect. (2017). http://www.jarchitect.com/ [last access: July 5, 2017].

[4] Bas Cornelissen, Danny Holten, Andy Zaidman, Leon Moonen, Jarke J Van Wijk, and Arie Van Deursen. 2007. Understanding execution traces using massive sequence and circular bundle views. In *Prog. ICPC'07*. IEEE, 49–58. DOI: http://dx.doi.org/10.1109/ICPC.2007.39

[5] Bas Cornelissen, Arie Van Deursen, Leon Moonen, and Andy Zaidman. 2007. Visualizing testsuites to aid in software understanding. In *Proc. CSMR'07*. IEEE, 213–222. DOI: http://dx.doi.org/10.1109/CSMR.2007.54

[6] Davor Cubranic and Gail C Murphy. 2004. Automatic bug triage using text categorization. In *Proc. SEKE'04*. 92–97. [last access: July 5, 2017].

[7] Marco D'Ambros, Alberto Bacchelli, and Michele Lanza. 2010. On the impact of design flaws on software defects. In *Proc. QSIC'10*. IEEE, 23–31. DOI: http://dx.doi.org/10.1109/QSIC.2010.58

[8] Steven D Eppinger and Tyson R Browning. 2012. *Design structure matrix methods and applications*. MIT press.

[9] Nikolaos Tsantalis et al. 2017. JDeodorant. (2017). https://github.com/tsantalis/JDeodorant [last access: July 5, 2017].

[10] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. 2012. Automatic detection of bad smells in code: An experimental assessment. *J. Object Technology* 11, 2 (2012), 5–1. DOI: http://dx.doi.org/10.5381/jot.2012.11.2.a5

[11] Francesca Arcelli Fontana, Jens Dietrich, Bartosz Walter, Aiko Yamashita, and Marco Zanoni. 2016. Antipattern and code smell false positives: Preliminary conceptualization and classification. In *Proc. SANER'16*, Vol. 1. IEEE, 609–613. DOI: http://dx.doi.org/10.1109/SANER.2016.84

[12] Francesca Arcelli Fontana, Riccardo Roveda, and Marco Zanoni. 2016. Tool support for evaluating architectural debt of an existing system: An experience report. In *Proc. SAC'16*. ACM, 1347–1349. DOI: http://dx.doi.org/10.1145/2851613.2851963

[13] Martin Fowler. 2009. *Refactoring: improving the design of existing code*. Pearson Education India.

[14] Mark Gabel, Lingxiao Jiang, and Zhendong Su. 2008. Scalable detection of semantic clones. In *Proc. ICSE'08*. IEEE, 321–330. DOI: http://dx.doi.org/10.1145/1368088.1368132

[15] Erich Gamma. 1995. *Design patterns: elements of reusable object-oriented software*. Pearson Education India.

[16] Thorsten Haendler. 2017. KaleidoScope. Institute for Inform. Syst. and New Media. WU Vienna. (2017). http://nm.wu.ac.at/nm/haendler [last access: July 5, 2017].

[17] Thorsten Haendler, Stefan Sobernig, and Mark Strembeck. 2015. Deriving Tailored UML Interaction Models from Scenario-Based Runtime Tests. In *Proc. ICSOFT'15*. Springer, 326–348. DOI: http://dx.doi.org/10.1007/978-3-319-30142-6_18

[18] Thorsten Haendler, Stefan Sobernig, and Mark Strembeck. 2016. Deriving UML-based Specifications of Inter-Component Interactions from Runtime Tests. In *Proc. SAC'16*. ACM, 1573–1575. DOI: http://dx.doi.org/10.1145/2851613.2851981

[19] hello2morrow. 2017. Sonargraph. (2017). https://www.hello2morrow.com/products/sonargraph [last access: July 5, 2017].

[20] I. Jacobson. 1992. *Object-oriented software engineering: A use case driven approach*. ACM.

[21] Matthias Jarke, X Tung Bui, and John M Carroll. 1998. Scenario management: An interdisciplinary approach. *Requirements Eng.* 3, 3 (1998), 155–173. DOI: http://dx.doi.org/10.1007/s007660050002

[22] Elmar Juergens, Florian Deissenboeck, and Benjamin Hummel. 2010. Code similarities beyond copy & paste. In *Proc. CSMR'10*. IEEE, 78–87. DOI: http://dx.doi.org/10.1109/CSMR.2010.33

[23] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. 2009. Do code clones matter?. In *Proc. ICSE'09*. IEEE, 485–495. DOI: http://dx.doi.org/10.1109/ICSE.2009.5070547

[24] Rick Kazman, Gregory Abowd, Len Bass, and Paul Clements. 1996. Scenario-based analysis of software architecture. *IEEE Softw.* 13, 6 (1996), 47–55. DOI: http://dx.doi.org/10.1109/52.542294

[25] Taeho Kim, Yeong-Tae Song, Lawrence Chung, and Dung T Huynh. 2000. Software architecture analysis: a dynamic slicing approach. *ACIS Int. J. Comput. & Inform. Sci.* 1, 2 (2000), 91–103.

[26] Philippe B Kruchten. 1995. The 4+1 view model of architecture. *IEEE Softw.* 12, 6 (1995), 42–50. DOI: http://dx.doi.org/10.1109/52.469759

[27] Roberto E. Lopez-Herrejon and Don Batory. 2001. A Standard Problem for Evaluating Product-Line Methodologies. In *Proc. GCSE'01*. Springer, Berlin, Heidelberg, 10–24. DOI: http://dx.doi.org/10.1007/3-540-44800-4_2

[28] Robert Cecil Martin. 2003. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR.

[29] Hayden Melton and Ewan Tempero. 2006. Identifying refactoring opportunities by identifying dependency cycles. In *Proc. ACSC'06*. ACM, 35–41.

[30] Emerson Murphy-Hill and Andrew P Black. 2010. An interactive ambient visualization for code smells. In *Proc. SOFTVIS'10*. ACM, 5–14. DOI: http://dx.doi.org/10.1145/1879211.1879216

[31] Clementine Nebut, F Fleurey, Y Le Traon, and JM Jezequel. 2006. Automatic Test Generation: A Use Case Driven Approach. *IEEE Trans. Softw. Eng.* 32, 3 (2006), 140–155. DOI: http://dx.doi.org/10.1109/TSE.2006.22

[32] Gustaf Neumann and Stefan Sobernig. 2015. Next Scripting Framework. API reference. (2015). https://next-scripting.org/xowiki/ [last access: July 5, 2017].

[33] Dan North. 2006. Introducing behaviour driven development. *Better Software Magazine* (2006).

[34] Object Management Group. 2015. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.2. (February 2015). http://www.omg.org/spec/QVT/1.2/ [last access: July 5, 2017].

[35] Object Management Group. 2015. Unified Modeling Language (UML), Superstructure, Version 2.5.0. (June 2015). http://www.omg.org/spec/UML/2.5 [last access: July 5, 2017].

[36] Sebastiano Panichella, Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. 2015. Would static analysis tools help developers with code reviews?. In *Proc. SANER'15*. IEEE, 161–170. DOI: http://dx.doi.org/10.1109/SANER.2015.7081826

[37] Ptidej. 2017. DECOR. (2017). http://www.ptidej.net/downloads/ [last access: July 5, 2017].

[38] Tamar Richner and Stéphane Ducasse. 1999. Recovering high-level views of object-oriented applications from static and dynamic information. In *Proc. ICSM'99*. IEEE, 13–22. DOI: http://dx.doi.org/10.1109/ICSM.1999.792487

[39] Chanchal Kumar Roy and James R Cordy. 2007. A Survey on Software Clone Detection Research. *Tech. Rep.* (2007). http://research.cs.queensu.ca/TechReports/Reports/2007-541.pdf [last access: July 5, 2017].

[40] Johannes Ryser and Martin Glinz. 1999. A scenario-based approach to validating and testing software systems using statecharts. In *Proc. ICSSEA'99*.

[41] Vitor Sales, Ricardo Terra, Luis Fernando Miranda, and Marco Tulio Valente. 2013. Recommending move method refactorings using dependency sets. In *Proc. WCRE'13*. IEEE, 232–241. DOI: http://dx.doi.org/10.1109/WCRE.2013.6671298

[42] Arthur Schreiber. 2013. TclSpec. (2013). https://github.com/arthurschreiber/tclspec [last access: July 5, 2017].

[43] Danilo Silva, Ricardo Terra, and Marco Tulio Valente. 2014. Recommending automated extract method refactorings. In *Proc. ICPC'14*. ACM, 146–156. DOI: http://dx.doi.org/10.1145/2597008.2597141

[44] Markus Strauch. 2016. Quick Sequence Diagram editor (sdedit). (2016). http://sdedit.sourceforge.net/ [last access: July 5, 2017].

[45] Mark Strembeck. 2011. Testing policy-based systems with scenarios. In *Proc. SE'11*. ACTA Press, 64–71. DOI: http://dx.doi.org/10.2316/P.2011.720-021

[46] Mark Strembeck and Jan Mendling. 2011. Modeling process-related RBAC models with extended UML activity models. *Inform. Softw. Technol.* 53, 5 (2011), 456–483. DOI: http://dx.doi.org/10.1016/j.infsof.2010.11.015

[47] Fang-Hsiang Su, Jonathan Bell, Kenneth Harvey, Simha Sethumadhavan, Gail Kaiser, and Tony Jebara. 2016. Code relatives: detecting similarly behaving software. In *Proc. FSE'16*. ACM, 702–714. DOI: http://dx.doi.org/10.1145/2950290.2950321

[48] Fang-Hsiang Su, Jonathan Bell, Gail Kaiser, and Simha Sethumadhavan. 2016. Identifying functionally similar code in complex codebases. In *Proc. ICPC'16*. IEEE, 1–10. DOI: http://dx.doi.org/10.1109/ICPC.2016.7503720

[49] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. 2014. *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann.

[50] Jonas Trumper, Jurgen Dollner, and Alexandru Telea. 2013. Multiscale visual comparison of execution traces. In *Proc. ICPC'13*. IEEE, 53–62. DOI: http://dx.doi.org/10.1109/ICPC.2013.6613833

[51] André Van Hoorn, Jan Waller, and Wilhelm Hasselbring. 2012. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proc. ICPE'12*. ACM, 247–248. DOI: http://dx.doi.org/10.1145/2188286.2188326

[52] Stefan Wagner, Asim Abdulkhaleq, Ivan Bogicevic, Jan-Peter Ostberg, and Jasmin Ramadani. 2016. How are functionally similar code clones syntactically different? An empirical study and a benchmark. *PeerJ Comput. Sci.* 2 (2016), e49. DOI: http://dx.doi.org/10.7717/peerj-cs.49

[53] BA Wichmann, AA Canning, DL Clutterbuck, LA Winsborrow, NJ Ward, and DWR Marsh. 1995. Industrial perspective on static analysis. *Softw. Eng. J.* 10, 2 (1995), 69–75. DOI: http://dx.doi.org/10.1049/sej.1995.0010

[54] Aiko Fallas Yamashita and Leon Moonen. 2013. Do developers care about code smells? An exploratory survey.. In *Proc. WCRE'13*, Vol. 13. IEEE, 242–251. DOI: http://dx.doi.org/10.1109/WCRE.2013.6671299

[55] ZEN PROGRAM. 2017. NDepend. (2017). http://www.ndepend.com/ [last access: July 5, 2017].