Ninth International Workshop on Managing Technical Debt (MTD 2017)

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

Thorsten Haendler, Stefan Sobernig, and Mark Strembeck

Vienna University of Economics and Business (WU Vienna)

thorsten.haendler@wu.ac.at

Assessment of smell candidates

Smell-detection tools produce ***false positives*** and/or miss smell candidates (due to applied detection technique: mostly static program analysis)

In general, smells also might result from a deliberate design decision (Arcelli Fontana et al., 2016; ***intentional smell***)

Smell Triage

A) *symptom-based identification and assessment*

B) *re-assessment of true positives*

- structural and behavioral context
- design decisions
- change impact and prioritization of potential refactorings

→ *effort/time for manual re-assessment*

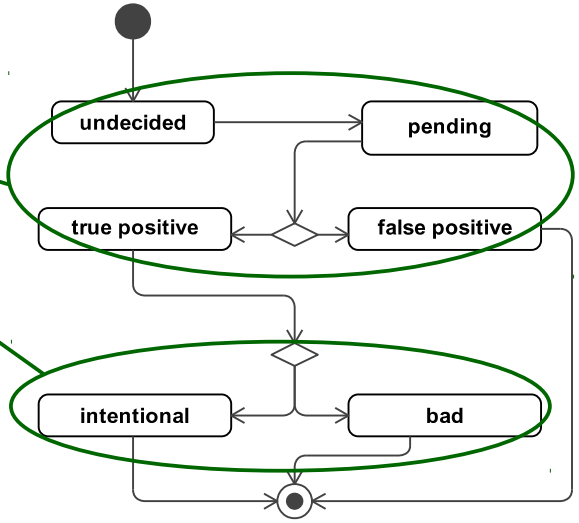


Fig. 1: Candidate states during triage

Approach: Decision support based on runtime scenarios and method-call dependencies

- ***Runtime Scenarios***
  Scenario-based runtime tests (e.g., BDD tests)
  → exemplary intended behavior

Given...
When...
Then...

```
set pES [::STORM_i::TestScenario new -name
         pushOnEmptyStack -testcase PushElement]
$pES expected_result set 1
$pES setup_script set {
  [::Stack getInstance] pop
}
$pES preconditions set {
  {expr {[[::Stack getInstance] size] == 0}}
  {expr {[[::Stack getInstance] limit get] == 4}}
}
$pES test_body set {
  [::Stack getInstance] push [::Element new -name e5 -value 1.9]
}
$pES postconditions set {
  {expr {[[::Stack getInstance] size] == 1}}
  {expr {[[[::Stack getInstance] top] name get] eq "e5"}}
  {expr {[[[::Stack getInstance] top] value get] == 1.9}}
}
$pES cleanup_script set {
  [::Stack getInstance] limit set 4
}
```

- ***Method-Call Dependencies***
  Multiple code smells manifest via call dependencies e.g., `FeatureEnvy`, `CyclicDependency`, `MessageChain`, *Functionally similar methods* (kind of `DuplicateCode`)

- *Reverse-Engineering* ***Design Perspectives*** *(using runtime analysis)*
  - dependency structure matrices (DSMs)
  - UML2 sequence diagrams

Scenario-driven smell assessment

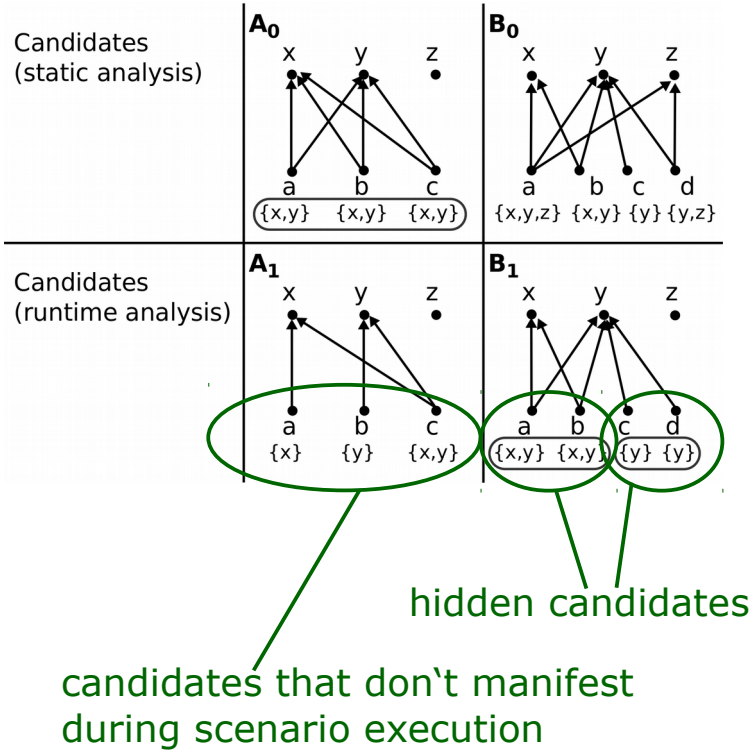
1. Identification of ***hidden*** candidates

2. Assessment of given candidates

   a) Check ***scenario-relevance*** of candidates

   b) Review scenario-scoped behavioral and structural ***candidate context*** (e.g., for identifying intentional smells such as applied design patterns)

Fig. 2: Example: Spotting candidates for *functionally similar methods* (kind of `DuplicateCode`)



Tailorable design perspectives derived from runtime scenarios

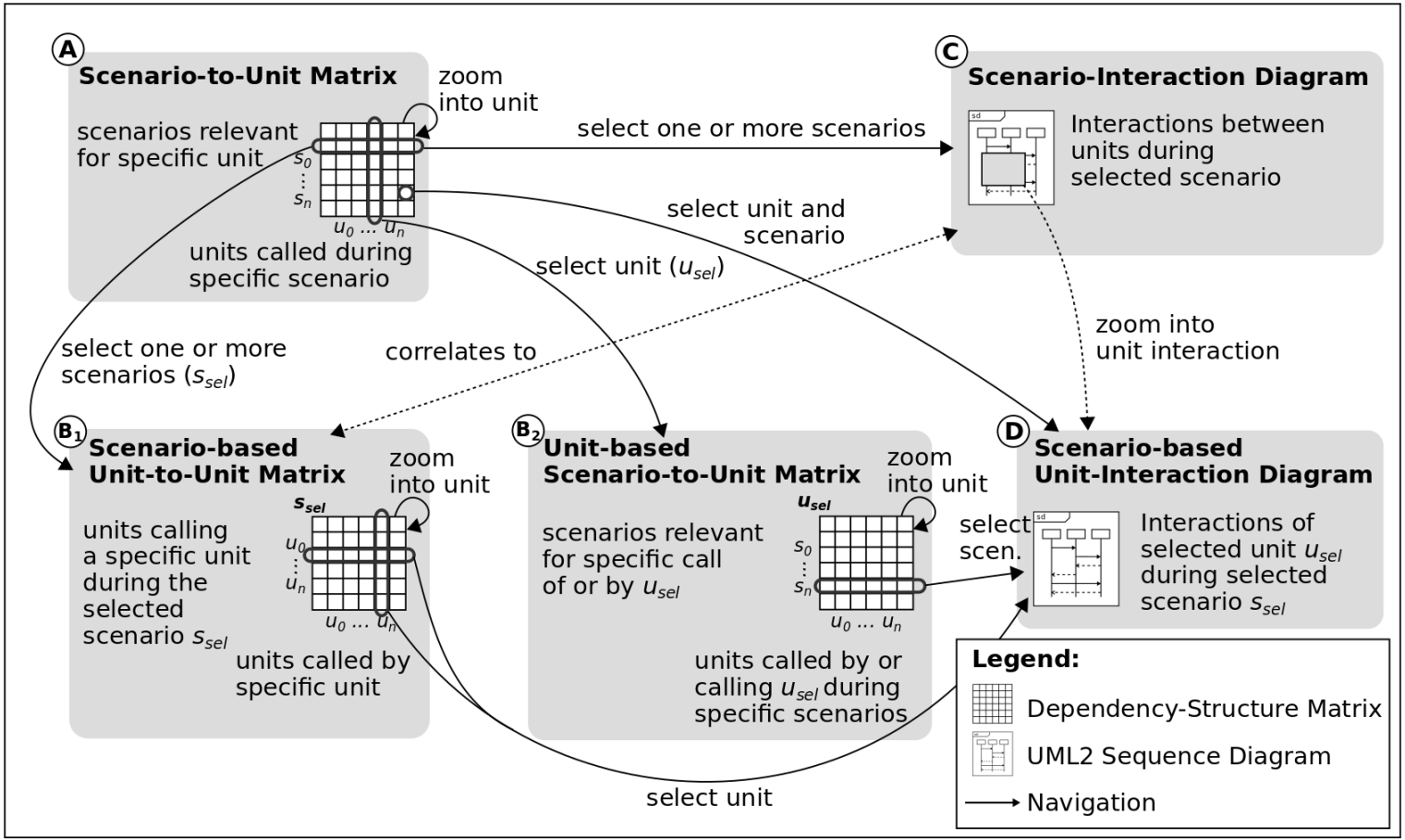


Fig. 3: Scenario & runtime perspectives on method-call dependencies for triaging smell candidates

Software prototype: KaleidoScope

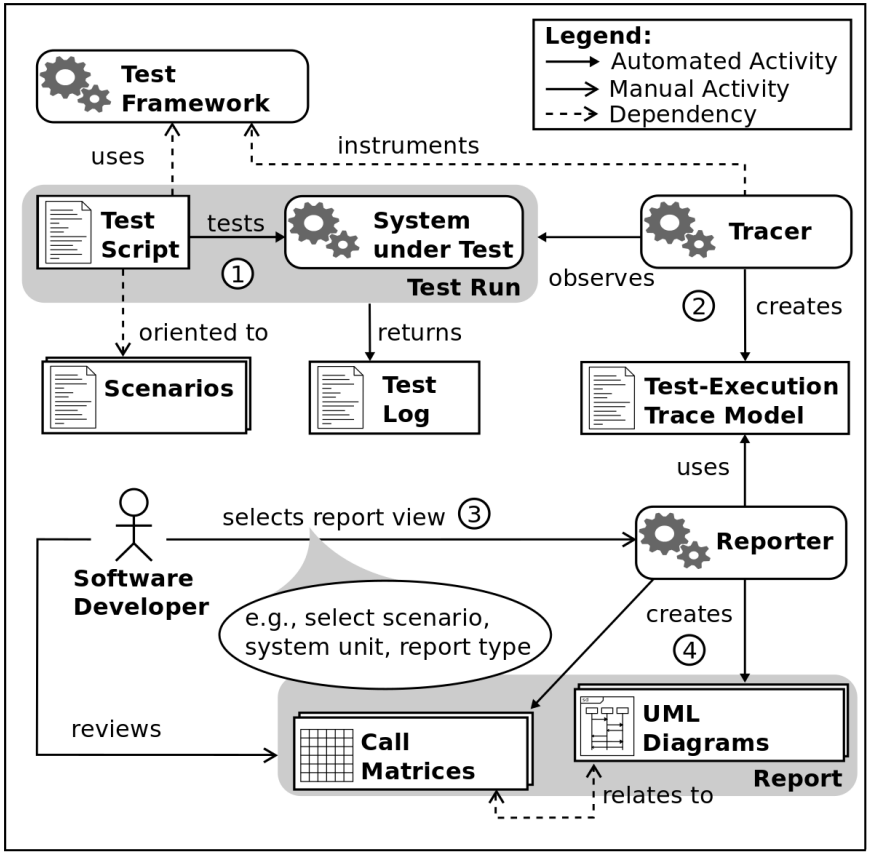


Fig. 4: Conceptual overview of ***KaleidoScope*** (publicly available for download at http://nm.wu.ac.at/nm/haendler)

Tracer Component

- instruments the test framework (e.g., *TclSpec/STORM*)
- creates *XMI* trace model

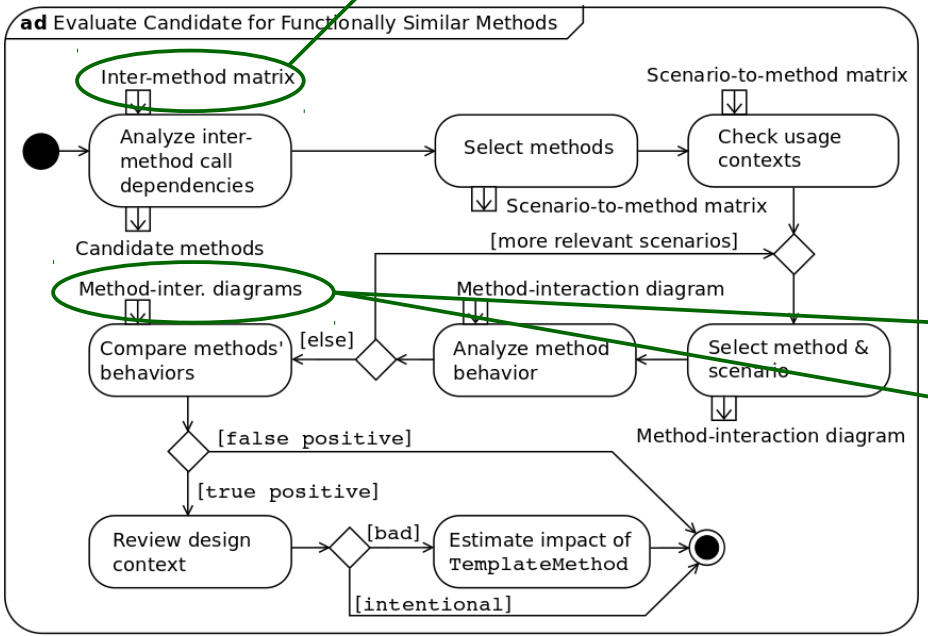
Reporter Component

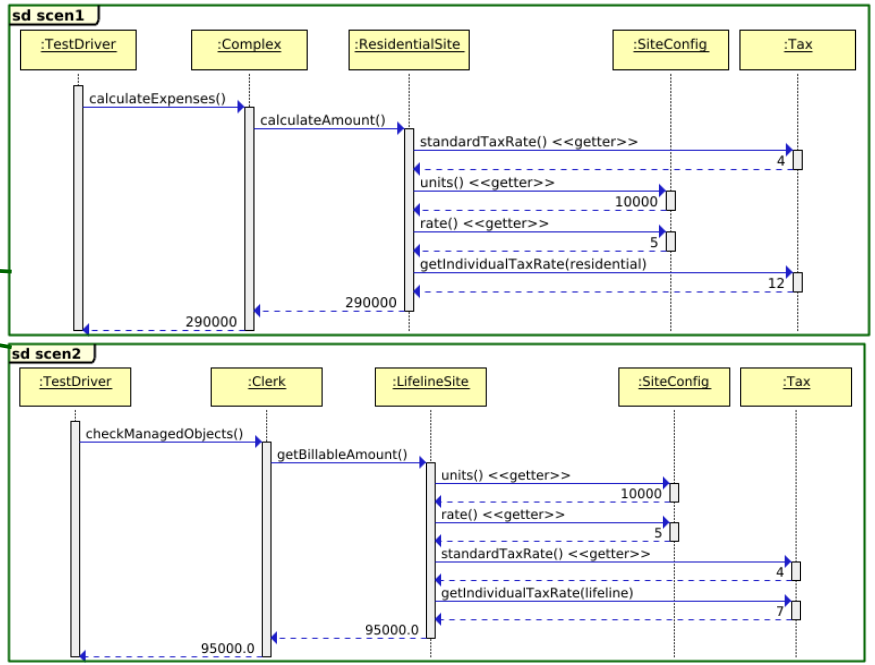
- parametric transformation
- UML models created using *QVTo mappings* and visualized in diagrams using *Quick Sequence Diagram Editor*
- matrices visualized using *R*

Simple example:
Assessing candidates for *functionally similar methods*

Overlapping set of called methods:
scenario-based inter-method matrix

further assessment criteria: order of method calls, i/o behavior, usage context (calling methods/classes, scenarios):
generated method-interaction diagrams





Figs. 5 & 6: Process for assessing FSM candidates (above) and exemplary generated method-interaction diagrams (righthand).

Summary

Decision support for triaging smell candidates

- reflecting method-call dependencies obtained from scenario test-execution traces
- providing different tailorable design perspectives (DSMs, UML2 sequence diagrams)
- complementing static-analysis tools

Prototypical implementation ***KaleidoScope***

Work in progress

- support for other smell types
- assisting in extended triaging questions (*bad vs. intentional* and refactoring planning)
- large(r) software systems
- experiments on the approach's benefits for human users

Discussion 1/4

Support for other code & design smells

Abstraction, Hierarchy, Encapsulation and other Modularization smells

→ also include data and subclass dependencies
→ additional design views (e.g., UML class diagrams)

Further potential of using scenarios

Example:
`MultifacedAbstraction`
(and `MissingAbstraction`)

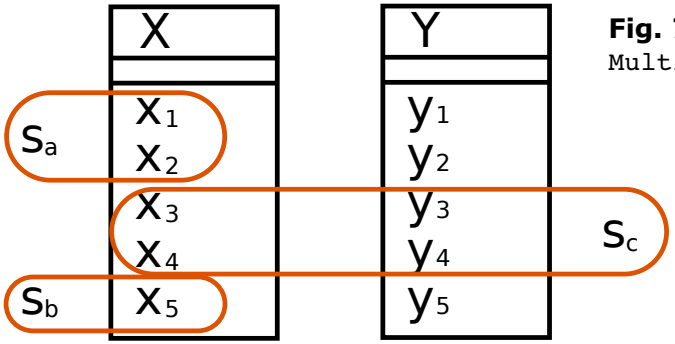


Fig. 7: `MultifacedAbstraction` example

Bad vs. Intentional
smell false positives in terms of design patterns (Arcelli Fontana et al., 2016)

→ behavioral context for identifying such intentional smells

Example: `Visitor` DP

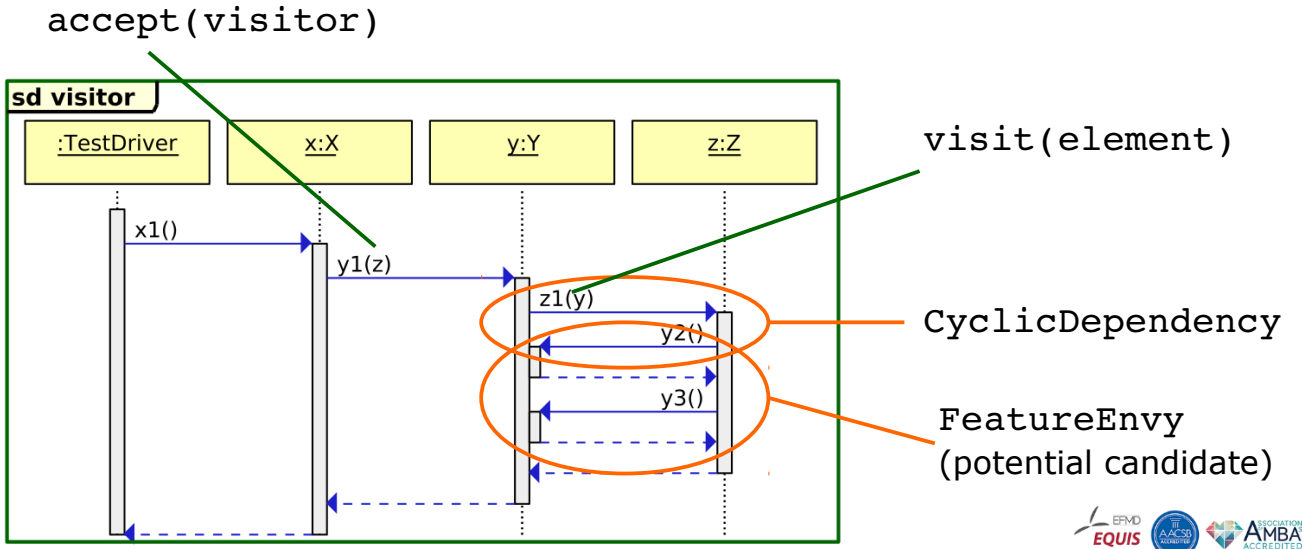


Fig. 8: Exemplary auto-generated method-interaction diagram

Discussion 3/4

Change-Impact Analysis
Impact of potential refactorings on system and test suite

Example: `MoveMethod`

Analysis	Exemplary Question	Perspective
Impact on program	*Which calling methods depend on the candidate method to be moved?*	scenario-based inter-method matrix
Impact on test suite	*Which scenario tests cover the method to be moved?*	scenario-to-method matrix
Move target	*Which existing classes are eligible owners of the candidate method to be moved?*	class-to-method and method-to-class matrices

Discussion 4/4

Larger application examples

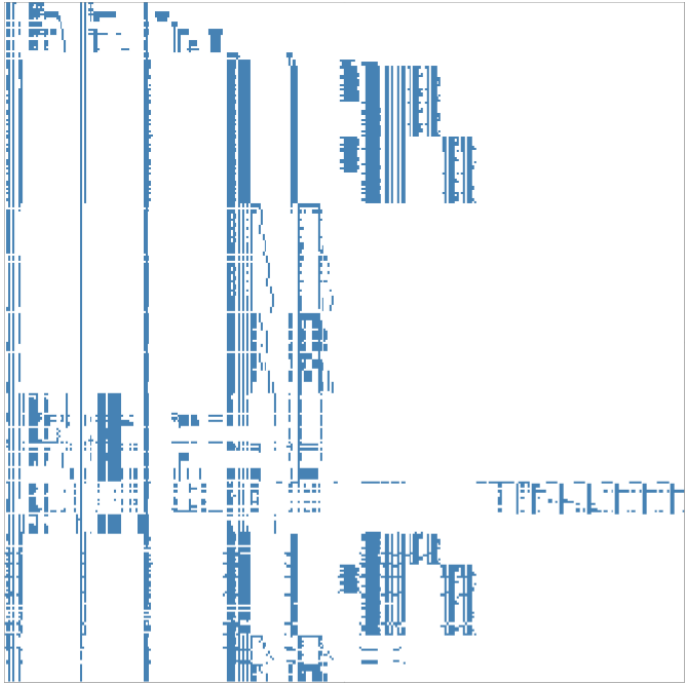


Fig. 9: **Scenario-to-method matrix**: called vs. not called (y-axis: test scenarios, x-axis: selected methods)

System under analysis:
357 test scenarios
~30k assertions

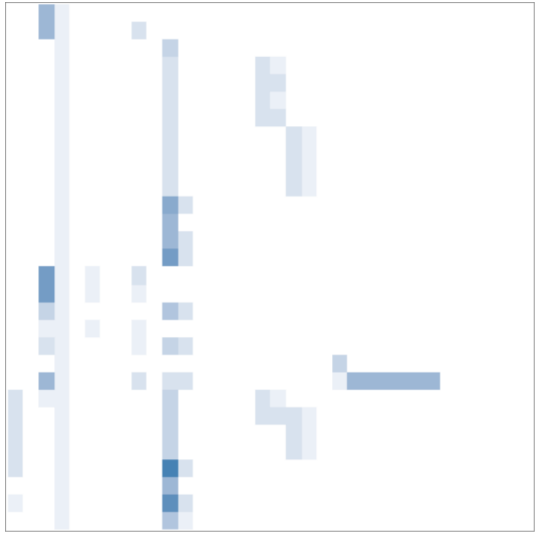


Fig. 10: **Scenario-to-class matrix**: amount of different methods triggering inter-class method calls (y-axis: selected test scenarios, x-axis: selected classes).

Thank you for your attention!

WIRTSCHAFTS UNIVERSITÄT WIEN VIENNA UNIVERSITY OF ECONOMICS AND BUSINESS

Questions & Discussion

thorsten.haendler@wu.ac.at

EFMD EQUIS ACCREDITED · AACSB ACCREDITED · ASSOCIATION OF MBAS ACCREDITED