Evolving Software Quality Knowledge

Daniel Speicher
dsp@cs.uni-bonn.de
Functionality

Quality

known structure of good design

unknown good design

known structure of bad design

unknown bad design
Overview

1. Unsound Generic Quality Knowledge
2. Logic Representation of Code Structure
3. Correctable Represented Quality Knowledge
4. Case Study: Leveraging Expressed Intentions
5. Towards Coding Culture Evolution
UNSOUND GENERIC QUALITY
KNOWLEDGE
Feature Envy Considered Bad

• Objects bring data and behavior together.
• A method has *feature envy*, if it operates for the major part on data of another class.
• Breaks encapsulation.
Visitor Pattern Considered Good

• The visitor pattern places functionality that operates on the data of certain objects (elements) in separate classes (visitors).

• Reasons
  – Elements build a complex object structure
  – Functionality belongs rather to the whole structure than to single elements.
  – Functionality is expected to change more frequently.
  – Functionality is used only in specific configurations.
Visitors Tend to Have Feature Envy

```java
public void visit(DependentTask p) {
    double cost = hourlyRate * p.getTimeRequired();
    cost *= p.getDependencyFactor();
    totalCost += cost;
}
```
Intended Data-Behavior Separation

• Separation of data into an extra object to share this data
  – *data class*: *extrinsic state* in *flyweight*, *context* in *interpreter*
  – *feature envy*: *flyweight* in *flyweight*, *expressions* in *interpreter*
• Separation of data to store it independently (in memento)
  – *data class*: *memento*
  – *feature envy*: *originator*
• Separation of the behavior to change it dynamically
  – *feature envy*: *concrete state*, *concrete strategy*
• Separation of the behavior to localize logic
  – *feature envy*: *concrete mediator*
Further Pattern Leading to Smells

- **factory, builder, singleton, facade, mediator**
  - central responsibility
  - strong afferent coupling
    - *shotgun surgery* [LM06]
- **concrete factory, builder and its director, facade, concrete mediator, concrete visitor, context in state**
  - coordinate/configure objects
  - *intensive/dispersed coupling*
  - might attract high complexity.
- **adapter, composite, abstract decorator, concrete decorator**
  - inserted into type hierarchies
  - *refused parent bequest*
  - *tradition breaker*
- **adapter, abstraction in bridge, abstract decorator, facade, proxy, concrete mediator, context in state**
  - add level of indirection
  - *middle man* [FBB+99]
Fact Representation of Java-Code Structures
Smell Detection Strategies
Lightweight Pattern Definition

LOGIC REPRESENTATION OF CODE STRUCTURE
Fact Representation of Java-Code

double cost = hourlyRate * p.getTimeRequired();

method “ProjectCostVisitor.visit(DependentTask p)”

body of the method

local(300, 299, 297, 20, ‘cost’, 301)
operation(301, 300, 297, [302, 303], ‘*’, 0)
getField(302, 301, 297, null, ‘hourlyRate’, 222)
call(303, 301, 297, 304, ‘getTimeRequired’, [ ], 111)
ident(304, 303, 297, ‘p’, 298)

basic type “double”

IDs of the AST nodes

method “DependentTask.getTimeRequired()”

parameter “DependentTask p”

field “ProjectCostVisitor.hourlyRate”
Predicates

\[
\text{method\_contains\_call(}\text{Method, Call}) \; :\; call(\text{Call, \_}, \text{Method, \_}, \_\text{, \_}, \_\text{, \_}).
\]

\[
\text{call\_calls\_method(}\text{Call, CalledMethod}) \; :\; call(\text{Call, \_}, \_\text{, \_}, \_\text{, \_}, \_\text{, \_}, \_\text{, CalledMethod}).
\]

\[
\text{method\_calls\_method(}\text{Calling, Called}) \; :\; \\
\text{method\_contains\_call(}\text{Calling, Call}), \\
\text{call\_calls\_method(}\text{Call, Called}).
\]
feature_usage_structure(S) :-
  source_method(S),
  not(abstract(S)).
Structure: Roles

\[
\text{method}(S, M) :- \\
\text{own\_class}(S, C) :- \\
\text{own\_field}(S, F) :- \\
\text{own\_class}(S, C), \\
\text{type\_contains\_field}(C, F), \\
\text{private}(F).
\]
method_accesses_foreign_field(S, M, F) :-
    method(S, M),
    foreign_field(S, F),
    method_accesses_field(M, F).

public void visit(DependentTask p) {
    double cost = hourlyRate * p.getTimeRequired();
    cost *= p.getDependencyFactor();
    totalCost += cost;
}
Metrics

access_to_foreign_data(M, V) :-
    count(F, method_accesses_foreign_field(M, M, F), V).

locality_of_attribute_access(M, V) :-
    count(F, method_accesses_own_field(M, M, F), AOF),
    count(F, method_accesses_foreign_field(M, M, F), AFF),
    V is AOF / (AOF + AFF).

foreign_data_provider(M, V) :-
    count(C, method_accesses_foreign_class(M, M, C), V).
Detection Strategy

\[
\text{feature_envy}(M) : - \\
\text{feature_usage_structure}(M), \\
\text{access_to_foreign_data}(M, \text{ATFD}), \quad \text{ATFD} > 2, \\
\text{locality_of_attribute_access}(M, \text{LAA}), \quad \text{LAA} < 0.3, \\
\text{foreign_data_provider}(M, \text{FDP}), \quad \text{FDP} \leq 5.
\]
Lightweight Pattern Definition

```prolog
visitor_pattern(P) :-
    declared_as_visitor(P).
visitor(P, V) :-
    visitor_pattern(P), P = V.
concrete_visitor(P, C) :-
    visitor(P, V), sub_type(V, C), not(interface(C)).
method_in_concrete_visitor(P, M) :-
    concrete_visitor(P, C), type_contains_method(C, M).
visited_element(P, E) :-
    visitor(P, V), type_contains_method(V, M),
    method_has_parameter(M, R), parameter_has_type(R, E).
field_in_visited_element(P, F) :-
    visited_element(P, E), type_contains_field(E, F).
```
```java
public void visit(DependentTask p) {
    double cost = hourlyRate * p.getTimeRequired();
    cost *= p.getDependencyFactor();
    totalCost += cost;
}
```
Tying Classes to the Pattern

To let the developers tie their class via a naming convention to the pattern, there should be predicate like:

\[
\text{declared\_as\_visitor}(V) \leftarrow \\
\text{class\_name\_ends\_with}(V, 'Visitor'), \\
\text{not}(\text{subclass}(V, P), \\
\text{class\_name\_ends\_with}(P, 'Visitor')).
\]

To tie it to the pattern via an annotation, another predicate should be used:

\[
\text{declared\_as\_visitor}(V) \leftarrow \\
\text{class\_annotated\_with}(V, 'Visitor').
\]
Intention Aware Smell Detection

CORRECTABLE REPRESENTED
QUALITY KNOWLEDGE
Natural Odor

Smell Detection checks

Expected Smell has a Role
Intention Aware Smell Detection

data_class(C) :-
    named_internal_type(C),
    not(natural_odor(data_class, C)),
    weight_of_class(C, WOC), WOC < 3.34,

natural_odor(data_class, E) :-
    visited_element(_, E).
Intended Collaboration

- Smell Detection uses [Coupling] Metric
- Role To (has a) Expected Relation
- From (has a) Uses | Used By | Forwards
- [J10]
method_accesses_foreign_field(S, M, F) :-
    method(S, M),
    foreign_field(S, F),
    not(intended_field_access(M, F)),
    method_accesses_field(M, F).

intended_field_access(M, F) :-
    visitor_pattern(P),
    method_in_concrete_visitor(P, M),
    field_in_visited_element(P, F).
CASE STUDY: LEVERAGING EXPRESSED INTENTIONS
Intensive / Dispersed Coupling

- Intensive: Many dependencies to a few other classes
- Dispersed: Dependencies to many other classes
- “Additionally, based on our practical experience, we impose a minimal complexity condition on the function, to avoid the case of configuration operations (e.g., initializers, or UI configuring methods) that call many other methods. These configuration operations reveal a less harmful (and hardly avoidable) form of coupling [...].” [LM06, p.121]
- "Method has few nested conditionals" measured by MAXNESTING > 1
Hypotheses

(0_c) Methods with names starting with “init”, “create”, “build” or “make” are configuration methods.

(1_c) Flat methods are configuration methods.

(2) Coupling in flat methods is no design problem.

(3_c) Coupling in configuration methods is no design problem.

(0_t) Methods with names starting with “test” or containing the term “Test” in their name or in the name of the enclosing class are test methods.

(1_t) Flat methods are test methods.

(2) Coupling in flat methods is no design problem.

(3_t) Coupling in test methods is no design problem.
Method Names

- Starting with “init”, “create”, “build” or “make”, e.g.:
  - buildAssociation()
  - buildAssociationEnd()
  - buildAssociationRole()
  - buildClass()
  - buildConnection()
  - buildInterface()
  - buildNode()
  - buildPopup()
  - buildStereotype()
  - buildTaggedValue()
  - create()
  - createDiagram()
  - createDiagramElement()
  - createEvent()
  - createFigText()
  - createMultiplicity()
  - createNewItem()
  - createPropPanel()
  - init()
  - initFigs()
  - initialiseListener()
  - initialize()
  - initNotationProviders()
  - initWizard()
  - makeKey()
  - makePanel()

- Starting with “test” or containing the term “Test” in their name or in the name of the enclosing class, e.g.:
  - testBasics()
  - testCreates()
  - testPredicate2()
  - TestableModelElements()
  - testGetMetaModelName()
  - testIsValidStereoType()
  - testValidObjectCheck()
  - testGetHelp()
  - testSingleton()
  - testCreateDiagram()
  - testSetUp()
  - TestZargoFilePersister.suite()
  - TestXmi.notestXmiRoseUml14()
  - TestUMLUseCaseDiagram.setUp()
  - TestUMLClassDiagram.tearDown()
  - TestTargetListener.targetRemoved()
  - TestStateBodyNotationUml.checkGenerated()
  - TestProfileDefault.setUp()

- Review of randomly chosen methods built strong trust into the naming rules.
- Meaningful method names are another supporting criterion.
- Precision of the naming convention very high.
Hypotheses

\( \checkmark_{(0_c)} \) Methods with names starting with “init”, “create”, “build” or “make” are configuration methods.

\( \checkmark_{(0_t)} \) Methods with names starting with “test” or containing the term “Test” in their name or in the name of the enclosing class are test methods.

\( (1_c) \) Flat methods are configuration methods.

\( (1_t) \) Flat methods are test methods.

\( (2) \) Coupling in flat methods is no design problem.

\( (2) \) Coupling in flat methods is no design problem.

\( (3_c) \) Coupling in configuration methods is no design problem.

\( (3_t) \) Coupling in test methods is no design problem.
## All Methods

<table>
<thead>
<tr>
<th>Max Nesting</th>
<th>config</th>
<th>test</th>
<th>other</th>
<th>all</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>772</td>
<td>880</td>
<td>6416</td>
<td>8068</td>
</tr>
<tr>
<td>2</td>
<td>318</td>
<td>110</td>
<td>2468</td>
<td>2896</td>
</tr>
<tr>
<td>&gt;2</td>
<td>152</td>
<td>53</td>
<td>2128</td>
<td>2333</td>
</tr>
</tbody>
</table>

Total:
- 1242
- 1043
- 11012
- 13297

62% are flat. 88% have nesting not bigger than 2.

Only 10% of the flat methods are configuration methods.

61% are flat. => Excluding improves performance.
Hypotheses

(0c) Methods with names starting with “init”, “create”, “build” or “make” are configuration methods.

(1c) Flat methods are configuration methods.

(2) Coupling in flat methods is no design problem.

(3c) Coupling in configuration methods is no design problem.

(0t) Methods with names starting with “test” or containing the term “Test” in their name or in the name of the enclosing class are test methods.

(1t) Flat methods are test methods.

(2) Coupling in flat methods is no design problem.

(3t) Coupling in test methods is no design problem.
Test of ($3_c$) and ($3_t$)

- Coupling in *configuration methods* is hardly avoidable as all the decoupled classes need to be instantiated and connected somewhere.
- Reviewed the 13 *configuration methods* with the highest coupling intensity (22 - 116) having one of the smells.
- Result: Even they are clearly understandable and the coupling is not harmful.
- The same is true for the 13 *test methods* with the highest coupling intensity (28 - 68).
Methods with names starting with "init", "create", "build" or "make" are configuration methods.

Flat methods are configuration methods.

Coupling in flat methods is no design problem.

Coupling in configuration methods is no design problem.

Methods with names starting with "test" or containing the term "Test" in their name or in the name of the enclosing class are test methods.

Flat methods are test methods.

Coupling in flat methods is no design problem.

Coupling in test methods is no design problem.
Test of (2)

• Reviewed the 13 methods with highest coupling intensity (10 - 16) within the other methods with no nesting, but with one of the smells.
• Impression not that clear as in the two cases before.
• Still the coupling did not require any refactoring.
### Hypotheses

<table>
<thead>
<tr>
<th>Hypothesis</th>
<th>Configuration Methods</th>
<th>Test Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0&lt;sub&gt;c&lt;/sub&gt;)</td>
<td>Methods with names starting with “init”, “create”, “build” or “make” are configuration methods.</td>
<td>Methods with names starting with “test” or containing the term “Test” in their name or in the name of the enclosing class are test methods.</td>
</tr>
<tr>
<td>(1&lt;sub&gt;c&lt;/sub&gt;)</td>
<td>Flat methods are configuration methods.</td>
<td>Flat methods are test methods.</td>
</tr>
<tr>
<td>(2)</td>
<td>Coupling in flat methods is no design problem.</td>
<td>Coupling in flat methods is no design problem.</td>
</tr>
<tr>
<td>(3&lt;sub&gt;c&lt;/sub&gt;)</td>
<td>Coupling in configuration methods is no design problem.</td>
<td>Coupling in test methods is no design problem.</td>
</tr>
</tbody>
</table>
## Intensive Coupling

<table>
<thead>
<tr>
<th>Max Nesting</th>
<th>config</th>
<th>test</th>
<th>other</th>
<th>all</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>19</td>
<td>57</td>
<td>26</td>
<td>102</td>
</tr>
<tr>
<td>2</td>
<td>88</td>
<td>97</td>
<td>19</td>
<td>204</td>
</tr>
<tr>
<td>&gt;2</td>
<td>209</td>
<td>234</td>
<td>29</td>
<td>472</td>
</tr>
</tbody>
</table>

|       | 316    | 388   | 74    | 778  |

- 41% of methods with intensive coupling are configuration methods.
- 50% of methods with intensive coupling are tests methods.
### Dispersed Coupling

<table>
<thead>
<tr>
<th>Max Nesting</th>
<th>config</th>
<th>test</th>
<th>other</th>
<th>all</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>27</td>
<td>143</td>
<td>32</td>
<td>202</td>
</tr>
<tr>
<td>2</td>
<td>60</td>
<td>24</td>
<td>213</td>
<td>297</td>
</tr>
<tr>
<td>&gt;2</td>
<td>52</td>
<td>34</td>
<td>568</td>
<td>654</td>
</tr>
</tbody>
</table>

68% of methods with dispersed coupling are other methods with a nesting of at least 2
Result

• Ignoring flat methods increases efficiency:
  – $8086/13297 = 61\%$ are flat

• Ignoring configuration methods and test methods increases effectiveness!
  – Config/test: $704/778 = 90\%$ or $340/1153 = 29\%$.
  – Flat methods: $102/778 = 13\%$ or $202/1153 = 18\%$

=> Smell detection should make use of expressed intentions.
Respecting Configuration Methods

configuration_method(M) :-
    declared_as_configuration_method(M).

natural_odor(intensive_coupling, E) :-
    configuration_method(E).

declared_as_configuration_method(M) :-
    source_method(M), method_name(M, N),
    member(P, ['init', 'create', 'build', 'make']), starts_with(N, P).
TOWARDS CODING CULTURE EVOLUTION
(Coding-)Culture-Evolution

Short time workflow (incomplete)

- Smell found
  - Justified?
    - Y: Refactoring
    - N: Reason is known structure
      - Y: Assign structure
      - N: Define structure

Long time activities

- Reviewing structure assignments
- Reviewing structure definitions
- Reviewing smell definitions
Overview

1. Unsound Generic Quality Knowledge

2. Logic Representation of Code Structure

3. Correctable Represented Quality Knowledge

4. Case Study: Leveraging Expressed Intentions

5. Towards Coding Culture Evolution
Literature


[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, January 1995.


