Perspectives on Automated Correction of Bad Smells

Javier Pérez, Yania Crespo
{jperez, yania}@infor.uva.es
Universidad de Valladolid

IWPSE-EVOL 2009
Aug 24-25 2009, Amsterdam
Introduction
Refactorings

Refactorings are structural transformations that can be applied to (originally) the source code of a software system to perform design changes (to improve it) without modifying its observable behaviour.

  - behaviour preserving invariants
  - preconditions
  - low-level refactorings
  - big refactorings

Refactorings are **structural transformations** that can be applied to (originally) the source code of a software system to perform design changes (to improve it) without modifying its observable behaviour.

  - behaviour preserving invariants
  - preconditions
  - low-level refactorings
  - big refactorings

Refactorings are structural transformations that can be applied to (originally) the source code of a software system to perform design changes (to improve it) without modifying its observable behaviour.

  - behaviour preserving invariants
  - preconditions
  - low-level refactorings
  - big refactorings

Refactorings are structural transformations that can be applied to (originally) the source code of a software system to perform design changes (to improve it) without modifying its observable behaviour.

  - behaviour preserving invariants
  - preconditions
  - low-level refactorings
  - big refactorings
Refactorings

Refactorings are structural transformations that can be applied to (originally) the source code of a software system to perform design changes (to improve it) without modifying its observable behaviour.

  - behaviour preserving invariants
  - preconditions
  - low-level refactorings
  - big refactorings

Refactorings

Refactorings are structural transformations that can be applied to (originally) the source code of a software system to perform design changes (to improve it) without modifying its observable behaviour.

  - behaviour preserving invariants
  - preconditions
  - low-level refactorings
  - big refactorings

Refactorings are structural transformations that can be applied to (originally) the source code of a software system to perform design changes (to improve it) without modifying its observable behaviour.

  - behaviour preserving invariants
  - preconditions
  - low-level refactorings
  - big refactorings

Examples of Refactorings

- Encapsulate Field
- Move Method, Field
- Rename Class, Field, Method
- Extract Method
- Replace Conditional with Polymorphism
- Pull Up Method, Extract Superclass
- ...
Introduction

Bad Smells

Problems encountered in the software’s structure (code or design), that can be detected statically, that do not produce compile or run-time errors, but negatively affect software quality factors. In fact, this negative effect on quality factors could lead to real compile and run-time errors in the future.

- Referred as “defects”, “flaws”, “disharmonies”, “antipatterns”, … by other authors.
- Bad smells are corrected with refactorings.
Bad Smells

Problems encountered in the software’s structure (code or design), that can be detected statically, that do not produce compile or run-time errors, but negatively affect software quality factors. In fact, this negative effect on quality factors could lead to real compile and run-time errors in the future.

- Referred as “defects”, “flaws”, “disharmonies”, “antipatterns”, ... by other authors.
- Bad smells are corrected with refactorings
Bad Smells

Problems encountered in the software’s structure (code or design), that can be detected statically, that do not produce compile or run-time errors, but negatively affect software quality factors. In fact, this negative effect on quality factors could lead to real compile and run-time errors in the future.

- Referred as “defects”, “flaws”, “disharmonies”, “antipatterns”, … by other authors.
- Bad smells are corrected with refactorings.
Bad Smells

Problems encountered in the software’s structure (code or design), that can be detected statically, that do not produce compile or run-time errors, but negatively affect software quality factors. In fact, this negative effect on quality factors could lead to real compile and run-time errors in the future.

- Referred as “defects”, “flaws”, “disharmonies”, “antipatterns”, … by other authors.
- Bad smells are corrected with refactorings.
Bad Smells

Problems encountered in the software’s structure (code or design), that can be detected statically, that do not produce compile or run-time errors, but negatively affect software quality factors. In fact, this negative effect on quality factors could lead to real compile and run-time errors in the future.

- Referred as “defects”, “flaws”, “disharmonies”, “antipatterns”, …by other authors.
- Bad smells are corrected with refactorings
Bad Smells

Problems encountered in the software’s structure (code or design), that can be detected statically, that do not produce compile or run-time errors, but negatively affect software quality factors. In fact, this negative effect on quality factors could lead to real compile and run-time errors in the future.

- Referred as “defects”, “flaws”, “disharmonies”, “antipatterns”, ... by other authors.
- Bad smells are corrected with refactorings
Introduction

Bad Smells

Problems encountered in the software’s structure (code or design), that can be detected statically, that do not produce compile or run-time errors, but negatively affect software quality factors. In fact, this negative effect on quality factors could lead to real compile and run-time errors in the future.

- Referred as “defects”, “flaws”, “disharmonies”, “antipatterns”, . . . by other authors.
- Bad smells are corrected with refactorings.
Bad Smells

Problems encountered in the software’s structure (code or design), that can be detected statically, that do not produce compile or run-time errors, but negatively affect software quality factors. In fact, this negative effect on quality factors could lead to real compile and run-time errors in the future.

- Referred as “defects”, “flaws”, “disharmonies”, “antipatterns”, ... by other authors.
- Bad smells are corrected with refactorings
Examples of Bad Smells

- Duplicated Code
- Temporary Field
- Data Class, Feature Envy
- Large Class, Large Method
- Refused Bequest
- ...
### Example of Data Class

**DataClass**

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
</table>

- getName()
- setName(n)
- getValue()
- setValue(v)

### Data Classes

“These are classes that have fields, getting and setting methods for the fields, and nothing else.”
Correction of bad smells has become a popular way to deal with the decay of the software’s structure that occurs during the evolution of a system.
Historical Analysis
Historical Analysis

Brief History of Bad Smell Management

Heuristics for Good OO Design
(1996 Riel) "Object-Oriented Design Heuristics"

Bad Smells Catalogs
(1999 Fowler) "Refactoring: Improving the ..."

Precise Smell Specifications
(2006 Lanza, Marinescu) "OO Metrics ..."

Automated Smell Detection
(2008 Moha) "DECOR: Détectection et Correction ..."

Correction Catalogs
(2004 Kerievsky) "Refactoring to Patterns"

Precise Correction Specifications
(2008 Trifu) "Towards Automated Restructuring ..."

Automated Correction

Detection

Correction

Perspectives on Bad Smell Correction
24-25 Aug 2009
On Detection
Bad Smell Detection: Fowler 1999 (Data Class)

Data Classes

“These are classes that have fields, getting and setting methods for the fields, and nothing else.”

- Textual description
Bad Smell Detection: Marinescu 2006 (God Class)

- Class uses directly more than a few attributes of other classes
  - ATFD > FEW
- Functional complexity of the class is very high
  - WMC ≥ VERY HIGH
- Class cohesion is low
  - TCC < ONE THIRD

Composition of metrics-based filters.
Bad Smell Detection: Moha 2008 (Spaghetti Code)

Rule Cards: DSL to specify bad smells to automate detection.
Composition of structural, metrics-based, and lexical filters.
Rule Cards: DSL to specify bad smells to automate detection.

Composition of structural, metrics-based, and lexical filters.
Bad Smell Detection: Moha 2008 (Spaghetti Code)

- Rule Cards: DSL to specify bad smells to automate detection.
- Composition of structural, metrics-based, and lexical filters.
Rule Cards: DSL to specify bad smells to automate detection.

Composition of structural, metrics-based, and lexical filters.
On Correction
Data Classes

In early stages these classes may have public fields. If so, you should immediately apply Encapsulate Field before anyone notices. If you have collection fields, check to see whether they are properly encapsulated and apply Encapsulate Collection if they aren’t. Use Remove Setting Method on any field that should not be changed. Look for where these getting and setting methods are used by other classes. Try to use Move Method to move behavior into the data class. If you can’t move a whole method, use Extract Method to create a method that can be moved. After a while you can start using Hide Method on the getters and setters.

- Textual and informal description.
Long Method

If a method is long because it contains numerous versions of an algorithm and conditional logic to choose which version to use at runtime, you can shrink the size of the method by applying **Replace Conditional Logic with Strategy**.

- Defines refactorings that involve introduction or removal of design patterns.
- “More structured” correction specifications for particular cases.
Historical Analysis

On Correction

Bad Smell Correction: Trifu 2004 (God Class) 2008 (Schizophrenic Class)

```java
strategy god_class{
    interface{
        Class godClass;
        List functionalityCollections;
    }
    List fc;
    int THRESHOLD = 0.7
    int i = 0;
    foreach (fc in functionalityCollections) {
        i = i + 1
        node move_or_extract (9) {
            branch {
                description("Move methods to satellite data classes.");
                quality{
                    favors(efficiency, complexity, understandability, modifiability);
                    disfavors(legibility, resconsumption);
                }
                Class dc;
                // find the class that the current functionality collection is most coupled to
                // and coupling must be at least as strong as specified in the threshold
                find_data_class(fc, dc, THRESHOLD);
                // call sequence to move methods in current functionality collection to identified
                // class. If class is nil, no moves occur and next branch is tried.
                if (dc == nil) abort;
                move_methods_to_datacls(fc, dc);
            }
            branch {
                description("Move methods to new classes");
                quality{
                    favors(understandability, modifiability);
                    disfavors(complexity, legibility, resconsumption, efficiency);
                }
                // generate a name for the new class. Optionally, this could be entered by the user.
                String name = godClass.className + n2a(i);
                extract_to_class(fc, name);
            }
        }
    }
}
```

Algorithmic correction. Quality factors. Very particular cases.
Bad Smell Correction: Trifu 2004 (God Class) 2008 (Schizophrenic Class)

```java
branch {
    description("Move methods to satellite data classes.");
    quality{
        favors(efficiency, complexity, understandability, modifiability);
        disfavors(legibility, resconsumption);
    }

    Class dc;
    // find the class that the current functionality collection is most coupled to
    // and coupling must be at least as strong as specified in the threshold
    find_data_class(fc, dc, THRESHOLD);

    // call sequence to move methods in current functionality collection to identified
    // class. If class is nil, no moves occur and next branch is tried.
    if (dc == nil) abort;
    move_methods_to_datacls(fc, dc);
}
```

- Algorithmic correction. Quality factors. Very particular cases.
Bad Smell Correction: Trifu 2004 (God Class) 2008 (Schizophrenic Class)

1: Let $O$ be the schizophrenic class
2: Check that we have an identity based decomposition of data in $O$, based on the identities of the encapsulated abstractions. // An action oriented topology in the case of the encapsulated abstractions would require a complete redesign of the fragment, which is outside the scope of design flaws in general (see 6.1.1)
3: Encapsulate all attributes in $O$ with public accessors // The public visibility is only temporary, in order to make moving functionality around easier
4: Identify all the abstractions $A_i$, that need to be separated and establish their future interfaces
5: Create empty classes that correspond to each of $A_i$
6: **if** $O$ has subtypes // we assume that they contain valid specializations for one or more of the abstractions contained in $O$ **then**
7: Establish those abstractions from $A_i$ that are affected by each specialization of $O$
8: Create appropriate subtypes for the classes that correspond to these abstractions
9: **end if**

Algorithmic correction. Quality factors. Very particular cases.
Bad Smell Correction: Trifu 2004 (God Class) 2008 (Schizophrenic Class)

10: **for all** attributes $a_i$ in $O$ do
11: Find the natural place for $a_i$ in one of the newly created classes, including helpers, based on the $A_i$ determined before and apply “move field” [Fow99]
12: if $a_i$ is an array or collection then
13: Decide between keeping the structured type and having an association multiplicity of 1 to the host class, or increasing the association's multiplicity and replacing the collection or array with only one of its elements. In the latter case, the class interface and the implementation of the facade need to be adapted accordingly
14: end if
15: end for

- Algorithmic correction. Quality factors. Very particular cases.
Bad Smell Correction: Trifu 2004 (God Class) 2008 (Schizophrenic Class)

16: **for all** methods $m_i$ in $O$ do
17:     **if** $m_i$ can be unambiguously assigned to one of the new classes **then**
18:         Apply “move method” [Fow99] to move $m_i$’s body to the respective class
19:         **if** $m_i$ is specialized in one of $O$’s subclasses **then**
20:             Apply “move method” [Fow99] to move the overriding method into the appropriate specialization
21:         **end if**
22:     **else**
23:         Apply “extract method” and “move method” [Fow99] to break up the original method, based on the attribute clusters that determine the encapsulated abstractions, and reunite functionality with its associated data
24:         **if** $m_i$ was previously specialized in one of $O$’s subclasses **then**
25:             Apply “extract method” and “move method” [Fow99] to break up the original overriding method, based on the attribute clusters that determine specializations of the encapsulated abstractions, and reunite functionality with its associated data
26:         **end if**
27:     **end if**

**Algorithmic correction. Quality factors. Very particular cases.**
Bad Smell Correction: Trifu 2004 (God Class) 2008 (Schizophrenic Class)

28:   if $m_i$ had public visibility then
29:      Implement “facade” [GHJV96] method in $O$, delegating to the appropriate abstraction(s).
30:   end if
31: end for
32: Create initialization methods in the facade $O$, or adapt its constructors to instantiate and wire together all newly defined classes and their specializations
33: Reduce data and accessor visibility as much as possible in all of the newly created classes

- Algorithmic correction. Quality factors. Very particular cases.
Model view of bad smell correction

- Correction Strategy
  - bad smell
- System Entity
- Metric
- Quality Factor
  - change
  - favoured
  - disfavoured
- Quality-Based Condition
- Behaviour-Preserving Transformation
  - Precondition
  - Transformation
  - Apply Refactoring
  - Introduce Pattern
  - Remove Pattern
- Algorithm
- Code Query
  - Structural
  - Numerical
  - Lexical
Model view of bad smell correction

- **Correction Strategy**
  - bad smell

- **System Entity**

- **Metric**
  - Quality Factor
    - favoured
    - disfavoured
    - change

- **Quality-Based Condition**

- **Behaviour-Preserving Transformation**
  - Precondition
  - Transformation

- **Algorithm**

- **Code Query**
  - Structural
  - Numerical
  - Lexical
  - Apply Refactoring
  - Introduce Pattern
  - Remove Pattern

- **Apply Refactoring**

- **Introduce Pattern**

- **Remove Pattern**

**Keywords:**
- Model view of bad smell correction
- Correction Strategy
- System Entity
- Metric
- Quality Factor
- Quality-Based Condition
- Behaviour-Preserving Transformation
- Algorithm
- Code Query
- Structural
- Numerical
- Lexical
- Apply Refactoring
- Introduce Pattern
- Remove Pattern

**Authors:**
- Javier Pérez (UVa)

**Date:**
- 24-25 Aug 2009
Model view of bad smell correction

- Correction Strategy
  - System Entity
  - Metric
  - Quality Factor
  - Quality-Based Condition
  - Behaviour-Preserving Transformation
  - Precondition
  - Transformation
- Algorithm
  - Code Query
    - Structural
    - Numerical
    - Lexical
    - Apply Refactoring
    - Introduce Pattern
    - Remove Pattern

Javier Pérez (UVa)
Model view of bad smell correction

Correction Strategy
- bad smell

System Entity

Metric

Quality Factor
- favoured
- disfavoured

Quality-Based Condition
- change

Behaviour-Preserving Transformation

Algorithm

Code Query
- Structural
- Numerical
- Lexical

Precondition

Transformation
- Apply Refactoring
- Introduce Pattern
- Remove Pattern
Model view of bad smell correction

Correction Strategy
- bad smell

System Entity
- Algorithm
- Code Query
  - Structural
  - Numerical
  - Lexical
  - Apply Refactoring
  - Introduce Pattern
  - Remove Pattern

Metric
- Quality Based Condition
  - Quality Factor
    - favoured
    - disfavoured
  - Change

Behaviour-Preserving Transformation
- Precondition
- Transformation
Model view of bad smell correction

Correction Strategy
- bad smell

System Entity

Metric

Quality Factor
- favoured
- disfavoured

Quality-Based Condition

Algorithm

Code Query
- Structural
- Numerical
- Lexical

Behaviour-Preserving Transformation
- Precondition
- Transformation
- Apply Refactoring
- Introduce Pattern
- Remove Pattern

Javier Pérez (UVa) Perspectives on Bad Smell Correction 24-25 Aug 2009 19 / 34
Brief History of Bad Smell Management

- **Heuristics for Good OO Design**
  - (1996 Riel) "Object-Oriented Design Heuristics"

- **Bad Smells Catalogs**
  - (1999 Fowler) "Refactoring: Improving the ..."

- **Precise Smell Specifications**
  - (2006 Lanza, Marinescu) "OO Metrics ..."

- **Automated Smell Detection**
  - (2008 Moha) "DECOR: Détection et Correction ..."

- **Correction Catalogs**
  - (2004 Kerievsky) "Refactoring to Patterns"

- **Precise Correction Specifications**
  - (2008 Trifu) "Towards Automated Restructuring ..."

- **Automated Correction**
  - ???
Brief History of Bad Smell Management

Heuristics for Good OO Design
(1996 Riel) "Object-Oriented Design Heuristics"

Bad Smells Catalogs
(1999 Fowler) "Refactoring: Improving the ..."

Precise Smell Specifications
(2006 Lanza, Marinescu) "OO Metrics ..."

Automated Smell Detection
(2008 Moha) "DECOR: Détectie et Correction ..."

Correction Catalogs
(2004 Kerievsky) "Refactoring to Patterns"

Precise Correction Specifications
(2008 Trifu) "Towards Automated Restructuring ..."

Automated Correction
???
Current Issues of Automated Correction

1. Applicability and instantiation of the desired transformation
   - The precondition may not hold when applying the transformation

2. Dependencies and conflicts between transformations
   - One transformation can enable or disable another.

3. Parallel transformations in team development
   - They may be difficult to merge

4. The heuristical nature of correction specifications
   - Imprecise specifications that have to be instantiated for a precise system.
Current Issues of Automated Correction

1. Applicability and instantiation of the desired transformation
   - The precondition may not hold when applying the transformation

2. Dependencies and conflicts between transformations
   - One transformation can enable or disable another.

3. Parallel transformations in team development
   - They may be difficult to merge

4. The heuristical nature of correction specifications
   - Imprecise specifications that have to be instantiated for a precise system.
Current Issues of Automated Correction

1. Applicability and instantiation of the desired transformation
   - The precondition may not hold when applying the transformation

2. Dependencies and conflicts between transformations
   - One transformation can enable or disable another.

3. Parallel transformations in team development
   - They may be difficult to merge

4. The heuristical nature of correction specifications
   - Imprecise specifications that have to be instantiated for a precise system.
Current Issues of Automated Correction

1. Applicability and instantiation of the desired transformation
   - The precondition may not hold when applying the transformation

2. Dependencies and conflicts between transformations
   - One transformation can enable or disable another.

3. Parallel transformations in team development
   - They may be difficult to merge

4. The heuristical nature of correction specifications
   - Imprecise specifications that have to be instantiated for a precise system.
Our Proposal
Refactoring Strategy

Automation-prone specification of the steps to perform a sequence of behaviour preserving transformations, which are addressed to achieve a particular goal.

- Refactoring strategies are instantiated to **refactoring plans**.

Refactoring Plan

Specification of a refactoring sequence which matches a system redesign proposal, and that can be immediately executed to modify the system, without changing the system’s behaviour, in order to obtain that desirable system redesign.
Refactoring Strategies & Refactoring Plans

Refactoring Strategy

Automation-prone specification of the steps to perform a sequence of behaviour preserving transformations, which are addressed to achieve a particular goal.

- Refactoring strategies are instantiated to refactoring plans.

Refactoring Plan

Specification of a refactoring sequence which matches a system redesign proposal, and that can be immediately executed to modify the system, without changing the system’s behaviour, in order to obtain that desirable system redesign.
Refactoring Strategies & Refactoring Plans

Refactoring Strategy

Automation-prone specification of the steps to perform a sequence of behaviour preserving transformations, which are addressed to achieve a particular goal.

- Refactoring strategies are instantiated to refactoring plans.

Refactoring Plan

Specification of a refactoring sequence which matches a system redesign proposal, and that can be immediately executed to modify the system, without changing the system’s behaviour, in order to obtain that desirable system redesign.
Refactoring Strategies & Refactoring Plans

Refactoring Strategy

Automation-prone specification of the steps to perform a sequence of behaviour preserving transformations, which are addressed to achieve a particular goal.

- Refactoring strategies are instantiated to refactoring plans.

Refactoring Plan

Specification of a refactoring sequence which matches a system redesign proposal, and that can be immediately executed to modify the system, without changing the system’s behaviour, in order to obtain that desirable system redesign.
Refactoring Strategies & Refactoring Plans

**Refactoring Strategy**

Automation-prone specification of the steps to perform a sequence of behaviour preserving transformations, which are addressed to achieve a particular goal.

- Refactoring strategies are instantiated to **refactoring plans**.

**Refactoring Plan**

Specification of a refactoring sequence which matches a system redesign proposal, and that can be immediately executed to modify the system, without changing the system’s behaviour, in order to obtain that desirable system redesign.
Refactoring Strategies & Refactoring Plans

Refactoring Strategy
Automation-prone specification of the steps to perform a sequence of behaviour preserving transformations, which are addressed to achieve a particular goal.

- Refactoring strategies are instantiated to **refactoring plans**.

Refactoring Plan
Specification of a refactoring sequence which matches a system redesign proposal, and that can be immediately executed to modify the system, without changing the system’s behaviour, in order to obtain that desirable system redesign.
Refactoring Strategies & Refactoring Plans

Refactoring Strategy

Automation-prone specification of the steps to perform a sequence of behaviour preserving transformations, which are addressed to achieve a particular goal.

- Refactoring strategies are instantiated to refactoring plans.

Refactoring Plan

Specification of a refactoring sequence which matches a system redesign proposal, and that can be immediately executed to modify the system, without changing the system’s behaviour, in order to obtain that desirable system redesign.
### Refactoring Strategy

Automation-prone specification of the steps to perform a sequence of behaviour preserving transformations, which are addressed to achieve a particular goal.

- Refactoring strategies are instantiated to **refactoring plans**.

### Refactoring Plan

Specification of a refactoring sequence which **matches a system redesign proposal**, and that can be immediately executed to modify the system, without changing the system’s behaviour, in order to obtain that desirable system redesign.
Refactoring Strategies & Refactoring Plans

Refactoring Strategy

Automation-prone specification of the steps to perform a sequence of behaviour preserving transformations, which are addressed to achieve a particular goal.

- Refactoring strategies are instantiated to **refactoring plans**.

Refactoring Plan

Specification of a refactoring sequence which matches a system redesign proposal, and that can be immediately executed to modify the system, without changing the system’s behaviour, in order to obtain that desirable system redesign.
Refactoring Strategies & Refactoring Plans

Refactoring Strategy

Automation-prone specification of the steps to perform a sequence of behaviour preserving transformations, which are addressed to achieve a particular goal.

- Refactoring strategies are instantiated to **refactoring plans**.

Refactoring Plan

Specification of a refactoring sequence which matches a system redesign proposal, and that can be immediately executed to modify the system, without changing the system’s behaviour, in order to obtain that desirable system redesign.
Our Proposal

Refactoring Strategies & Refactoring Plans

Refactoring Strategy

Automation-prone specification of the steps to perform a sequence of behaviour preserving transformations, which are addressed to achieve a particular goal.

- Refactoring strategies are instantiated to **refactoring plans**.

Refactoring Plan

Specification of a refactoring sequence which matches a system redesign proposal, and that can be immediately executed to modify the system, without changing the system’s behaviour, in order to obtain that desirable system redesign.
Our Proposal

Refactoring Strategies & Refactoring Plans

---

Javier Pérez (UVa)
Perspectives on Bad Smell Correction
24-25 Aug 2009 24 / 34
Our Proposal

Refactoring Strategies & Refactoring Plans

- Transformation
  - Simple
  - Composed
- Add
- Remove
- Replace
- System Entity
- Node
- Edge
- AST Element
- Behaviour-Preserving Transformation
- Conflict
- Dependency
- Arguments
- Precondition
- Refactoring
- Introduce Pattern
- Remove Pattern

Goal
- Refactoring Plan
- Quality-Based Condition
  - Quality Factor
  - Control Constructs
  - Code Query
- Refactoring Strategy
- Composition
- favoured
- disfavoured

Transformation Composition
- Composed
- Simple
- Add
- Remove
- Replace
- System Entity
- Node
- Edge
- AST Element
- Behaviour-Preserving Transformation
- Conflict
- Dependency
- Arguments
- Precondition
- Refactoring
- Introduce Pattern
- Remove Pattern

Javier Pérez (UVa)
Perspectives on Bad Smell Correction
24-25 Aug 2009 24 / 34
Our Proposal

Refactoring Strategies & Refactoring Plans

<table>
<thead>
<tr>
<th>Goal</th>
<th>Refactoring Strategy</th>
<th>Refactoring Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Instantiated</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Quality-Based Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structural</td>
</tr>
<tr>
<td>Numerical</td>
</tr>
<tr>
<td>Lexical</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Quality Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>favoured</td>
</tr>
<tr>
<td>disfavoured</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composed</td>
</tr>
<tr>
<td>Simple</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node</th>
<th>Edge</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>AST Element</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Behaviour-Preserving Transformation</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>System Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precondition</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composition</td>
</tr>
<tr>
<td>Dependency</td>
</tr>
<tr>
<td>Conflict</td>
</tr>
<tr>
<td>Arguments</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bad Smell</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Control Constructs</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Code Query</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Remove Smell</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Refactoring</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Introduce Pattern</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Remove Pattern</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Node</th>
<th>Edge</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Transform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
</tr>
<tr>
<td>Remove</td>
</tr>
<tr>
<td>Replace</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Incorrect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remove</td>
</tr>
<tr>
<td>Replace</td>
</tr>
<tr>
<td>Add</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Behaviour-Preserving Transformation</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>System Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precondition</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composition</td>
</tr>
<tr>
<td>Dependency</td>
</tr>
<tr>
<td>Conflict</td>
</tr>
<tr>
<td>Arguments</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bad Smell</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Control Constructs</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Code Query</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Remove Smell</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Refactoring</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Introduce Pattern</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Remove Pattern</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Node</th>
<th>Edge</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Transform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
</tr>
<tr>
<td>Remove</td>
</tr>
<tr>
<td>Replace</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Incorrect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remove</td>
</tr>
<tr>
<td>Replace</td>
</tr>
<tr>
<td>Add</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Behaviour-Preserving Transformation</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>System Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precondition</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composition</td>
</tr>
<tr>
<td>Dependency</td>
</tr>
<tr>
<td>Conflict</td>
</tr>
<tr>
<td>Arguments</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bad Smell</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Control Constructs</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Code Query</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Remove Smell</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Refactoring</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Introduce Pattern</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Remove Pattern</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Node</th>
<th>Edge</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Transform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
</tr>
<tr>
<td>Remove</td>
</tr>
<tr>
<td>Replace</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Incorrect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remove</td>
</tr>
<tr>
<td>Replace</td>
</tr>
<tr>
<td>Add</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument</th>
</tr>
</thead>
</table>
Refactoring Strategies & Refactoring Plans

Transformation

Simple

Composed

Goal

Refactoring Strategy

Refactoring Plan

instantiates

Composition

Quality Factor

favoured
disfavoured

Transformation

Apply Transformation

Remove Smell

Bad Smell

Correction Strategy

Control Constructs

Code Query

Quality-Based Condition

Structural

Numerical

Lexical

Metric

AST Element

Add

Remove

Replace

System Entity

Precondition

Behaviour-Preserving Transformation

Node

Edge

AST Element

arguments

conflict
dependency

Refactoring

Introduce Pattern

Remove Pattern

Javier Pérez (UVa)

Perspectives on Bad Smell Correction

24-25 Aug 2009

24 / 34
Refactoring Planning
Automated Planning

Definition

Automated planning is an artificial intelligence technique to generate sequences of actions that will achieve a certain goal when they are performed.

Example: Getting apples and a book.

The state of the world: at (grocery) AND not (have (apples))
Actions: buy (apples); moveTo (bookstore)
Goals: have (book) AND have (apples)
Classical Planning Operators (STRIPS)

- **World’s state**: list of terms

- **Operators**:
  - definition: name + arguments
  - precondition
  - effect list (add): terms to add to the state
  - effect list (deletes): terms to remove from the state

- **Problem**:
  - initial state
  - goal: list of terms

- **General planning approach**: chain operators by matching their effects and preconditions
Smell Correction with HTN Planning

- **World’s state:** AST represented by first order logic formulas
- **Operators:** refactorings & refactoring’s substeps
- **Tasks:**
  - refactorings strategies
  - smell correction strategies
- **Goals:** Execute a smell correction strategy
- **Planning Problem:** Execute a particular smell correction strategy over a particular version of a system
Basic transformations are specified in terms of preconditions and effects.

Refactoring strategies are specified as additional knowledge to the planner (tasks).

The user requests the correction of a bad smell or the application of a refactoring.

The instantiation of the strategy is not hard-coded in an algorithm.

The planner selects the applicable operators and their execution order.
Refactoring Planning

- Basic transformations are specified in terms of preconditions and effects.
- Refactoring strategies are specified as additional knowledge to the planner (tasks).
  - The user requests the correction of a bad smell or the application of a refactoring.
  - The instantiation of the strategy is not hard-coded in an algorithm.
  - The planner selects the applicable operators and their execution order.
Basic transformations are specified in terms of preconditions and effects.

Refactoring strategies are specified as additional knowledge to the planner (tasks).

The user requests the correction of a bad smell or the application of a refactoring.

The instantiation of the strategy is not hard-coded in an algorithm.

The planner selects the applicable operators and their execution order.
Refactoring Planning

- Basic transformations are specified in terms of preconditions and effects.
- Refactoring strategies are specified as additional knowledge to the planner (tasks).
- The user requests the correction of a bad smell or the application of a refactoring.
- The instantiation of the strategy is not hard-coded in an algorithm.
- The planner selects the applicable operators and their execution order.
Basic transformations are specified in terms of preconditions and effects.

Refactoring strategies are specified as additional knowledge to the planner (tasks).

The user requests the correction of a bad smell or the application of a refactoring.

The instantiation of the strategy is not hard-coded in an algorithm.

The planner selects the applicable operators and their execution order.
Conclusions
Conclusions

- Bad Smell correction has evolved through the development of more adequate specifications
  - which have led to better automatisation of the activity.

- The activity has clearly manifested an empirically nature.
  - The detection and correction knowledge is specified heuristically.

- Bad smell correction can keep being improved
  - by working on specification and automation of the correction activity.
Conclusions

- **Bad Smell correction has evolved through the development of more adequate specifications**
  - which have led to better automatisation of the activity.

- **The activity has clearly manifested an empirically nature.**
  - The detection and correction knowledge is specified heuristically.

- **Bad smell correction can keep being improved**
  - by working on specification and automation of the correction activity.
Conclusions

- Bad Smell correction has evolved through the development of more adequate specifications
  - which have led to better automatisation of the activity.

- The activity has clearly manifested an empirically nature.
  - The detection and correction knowledge is specified heuristically.

- Bad smell correction can keep being improved
  - by working on specification and automation of the correction activity.
Expected Contributions

- Refactoring Plans and Refactoring Strategies
  - as a way to write bad smell correction specifications

- Automated instantiation of refactoring plans by using HTN planning
Implementing a prototypical infrastructure
  - Reusing existing tools

Implementing refactoring strategies
  - to apply refactorings
  - to correct bad smells

Choosing systems to experiment with
Perspectives on Automated Correction of Bad Smells

Javier Pérez, Yania Crespo
{jperez, yania}@infor.uva.es
Universidad de Valladolid

IWPSE-EVOL 2009
Aug 24-25 2009, Amsterdam