Towards a Framework for Software Design Defects Correction with Refactoring Plans

Javier Pérez
jperez@infor.uva.es

Universidad de Valladolid
Université de Mons-Hainaut

Fundamental Aspects of Software Evolution
FNRS Contact Group on Fundamental Computer Science
May 22nd 2008, University of Namur

Introduction

Software Design Defects

Definition

Design defects are “bad” solutions to recurring design problems in object-oriented systems. Design defects are problems resulting from bad design practices. They include problems ranging from high-level and design problems, such as antipatterns, to low-level or local problems, such as code smells. (Moha, 2008)

- Why is important to deal with design defects?

Software Design Defects

Definition

Design defects are "bad" solutions to recurring design problems in object-oriented systems. Design defects are problems resulting from bad design practices. They include problems ranging from high-level and design problems, such as antipatterns, to low-level or local problems, such as code smells. (Moha, 2008)

- Why is important to deal with design defects?

Software Design Defects

Definition

Design defects are “bad” solutions to recurring design problems in object-oriented systems. Design defects are problems resulting from bad design practices. They include problems ranging from high-level and design problems, such as antipatterns, to low-level or local problems, such as code smells. (Moha, 2008)

- Why is important to deal with design defects?

Software Design Defects

Definition

Design defects are “bad” solutions to recurring design problems in object-oriented systems. Design defects are problems resulting from bad design practices. They include problems ranging from high-level and design problems, such as antipatterns, to low-level or local problems, such as code smells. (Moha, 2008)

- Why is important to deal with design defects?

Software Design Defects

Definition

Design defects are “bad” solutions to recurring design problems in object-oriented systems. Design defects are problems resulting from bad design practices. They include problems ranging from high-level and design problems, such as antipatterns, to low-level or local problems, such as code smells. (Moha, 2008)

- Why is important to deal with design defects?

Software Design Defects

Definition

Design defects are “bad” solutions to recurring design problems in object-oriented systems. Design defects are problems resulting from bad design practices. They include problems ranging from high-level and design problems, such as antipatterns, to low-level or local problems, such as code smells. (Moha, 2008)

- Why is important to deal with design defects?

Software Design Defects

Definition

Design defects are “bad” solutions to recurring design problems in object-oriented systems. Design defects are problems resulting from bad design practices. They include problems ranging from high-level and design problems, such as antipatterns, to low-level or local problems, such as code smells. (Moha, 2008)

- Why is important to deal with design defects?

Motivation

- Software evolution “happens”.
- Software design decays:
  - changes are applied hastily
  - “design debt” appears (Kerievsky, *Refactoring To Patterns*)
- Design decay can manifest through design defects, which affect software quality factors:
  - maintainability
  - reusability
  - comprehensibility
  - . . .

Motivation

- Software evolution “happens”.
- Software design decays:
  - changes are applied hastily
  - “design debt” appears (Kerievsky, *Refactoring To Patterns*)
- Design decay can manifest through design defects, which affect software quality factors:
  - maintainability
  - reusability
  - comprehensibility
  - . . .

Motivation

- Software evolution “happens”.
- Software design decays:
  - changes are applied hastily
  - “design debt” appears (Kerievsky, *Refactoring To Patterns*)
- Design decay can manifest through design defects, which affect software quality factors:
  - maintainability
  - reusability
  - comprehensibility
  - . . .

Software Design Defect Management

- The ideal is to prevent design defects, but
- . . . design defects appear, so
- systematic ways to detect and correct design defects are needed.

Software Design Defect Management

- The ideal is to prevent design defects, but
- . . . design defects appear, so
- systematic ways to detect and correct design defects are needed.

Software Design Defect Management

- The ideal is to prevent design defects, but
- . . . design defects appear, so
- systematic ways to detect and correct design defects are needed.

Software Design Defect Management

- Techniques to **detect design defects** and to **suggest design changes** are maturing:
  - Structural patterns to find defects (Moha, DECOR project)
  - Metrics to detect “bad smells” (Marinescu, 2006; Crespo et al., 2005).
  - Formal/Relational Concept Analysis to propose reorganisation of OO entities (Moha et al., 2006; Prieto et al., 2003).
  - Software inconsistency management (Mens, 2006)
- The change suggestions given:
  - are not directly applicable over a system,
  - are usually given in terms of refactorings.

Software Design Defect Management

- Techniques to **detect design defects** and to **suggest design changes** are maturing:
  - Structural patterns to find defects (Moha, DECOR project)
  - Metrics to detect “bad smells” (Marinescu, 2006; Crespo et al., 2005).
  - Formal/Relational Concept Analysis to propose reorganisation of OO entities (Moha et al., 2006; Prieto et al., 2003).
  - Software inconsistency management (Mens, 2006)
- The change suggestions given:
  - are not directly applicable over a system,
  - are usually given in terms of refactorings.

Refactorings to Correct Design Defects

- **Refactorings** are structural transformations that can be applied to a software system to perform design changes without modifying its behaviour.
- **Current approaches** to improve a system design with refactorings focus in:
  - Individual refactoring steps.
  - Detecting refactoring opportunities.
  - Assisting the developer in executing the refactoring

Refactorings to Correct Design Defects

- **Refactorings** are structural transformations that can be applied to a software system to perform design changes without modifying its behaviour.
- **Current approaches** to improve a system design with refactorings focus in:
  - Individual refactoring steps.
  - Detecting refactoring opportunities.
  - Assisting the developer in executing the refactoring

Objective of a Defect Correction Framework

1. Instantiate defect removing suggestions into a correction plan which could be effectively applied.
   - through refactorings, because systems' behaviour should be preserved.

Objective of a Defect Correction Framework

1. Instantiate defect removing suggestions into a correction plan which could be effectively applied.
    - through refactorings, because systems' behaviour should be preserved.

Refactoring Plans

- We pretend to introduce a new concept: **Refactoring Plans**

Definition

A **Refactoring Plan** will be a specification of a refactoring sequence which matches a system redesign proposal, so that it can be automatically executed to modify the system in order to obtain that desirable system redesign without changing the system's behaviour.

Refactoring Plans

- We pretend to introduce a new concept: **Refactoring Plans**

Definition

A **Refactoring Plan** will be a specification of a refactoring sequence which matches a system redesign proposal, so that it can be automatically executed to modify the system in order to obtain that desirable system redesign without changing the system's behaviour.

Refactoring Plans

- We pretend to introduce a new concept: **Refactoring Plans**

Definition

A **Refactoring Plan** will be a specification of a refactoring sequence which matches a system redesign proposal, so that it can be automatically executed to modify the system in order to obtain that desirable system redesign without changing the system's behaviour.

Refactoring Plans

- We pretend to introduce a new concept: **Refactoring Plans**

Definition

A **Refactoring Plan** will be a specification of a refactoring sequence which matches a system redesign proposal, so that it can be automatically executed to modify the system in order to obtain that desirable system redesign without changing the system's behaviour.

Refactoring Plans

- We pretend to introduce a new concept: **Refactoring Plans**

Definition

A **Refactoring Plan** will be a specification of a refactoring sequence which matches a system redesign proposal, so that it can be automatically executed to modify the system in order to obtain that desirable system redesign without changing the system's behaviour.

Refactoring Plans

- We pretend to introduce a new concept: **Refactoring Plans**

Definition

A **Refactoring Plan** will be a specification of a refactoring sequence which matches a system redesign proposal, so that it can be automatically executed to modify the system in order to obtain that desirable system redesign without changing the system's behaviour.

Refactoring Plans

- We pretend to introduce a new concept: **Refactoring Plans**

Definition

A **Refactoring Plan** will be a specification of a refactoring sequence which matches a system redesign proposal, so that it can be automatically executed to modify the system in order to obtain that desirable system redesign without changing the system's behaviour.

Goals of a Framework for Refactoring Plans

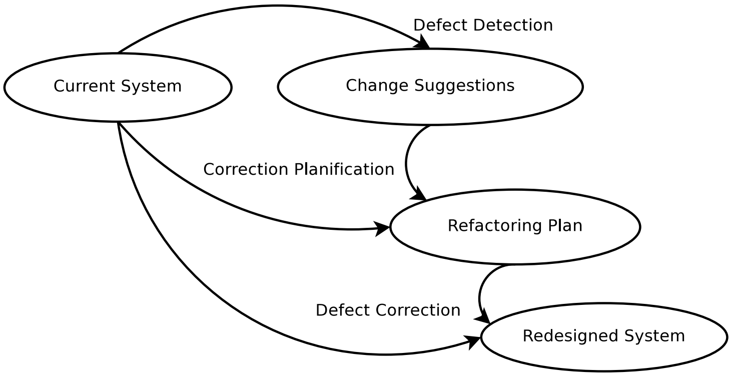
1. **Support to automatic or assisted generation of refactoring plans**
2. **To provide very high level (big) refactorings** for design improvement, using refactoring plan generation altogether with the defect detection techniques that suggest redesign proposals.

Goals of a Framework for Refactoring Plans

1. **Support to automatic or assisted generation of refactoring plans**
2. **To provide very high level (big) refactorings** for design improvement, using refactoring plan generation altogether with the defect detection techniques that suggest redesign proposals.

Design Defect Correction

General Defect Correction Process



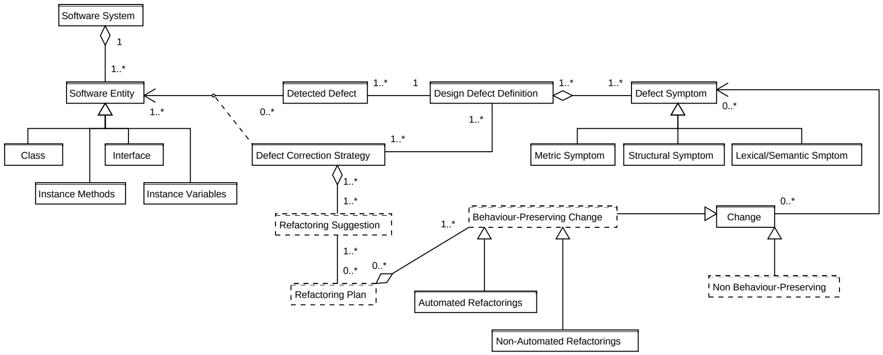
Goals of a Framework for Refactoring Plans

1. **Support to automatic or assisted generation of refactoring plans**
2. **To provide very high level (big) refactorings** for design improvement, using refactoring plan generation altogether with the defect detection techniques that suggest redesign proposals.

Goals of a Framework for Refactoring Plans

1. **Support to automatic or assisted generation of refactoring plans**
2. **To provide very high level (big) refactorings** for design improvement, using refactoring plan generation altogether with the defect detection techniques that suggest redesign proposals.

A Framework sketch



Generating Refactoring Plans

Refactoring Plan Questions

- Given **a software system** as the source of the transformation, **a redesign proposal**, and **a set of refactorings** that can be used as transformation operations:
  1. Does a refactoring plan, which transforms the source, according to the redesign proposal, using the provided refactorings, exist?
     - additional non-refactoring transformations could be needed
  2. When a refactoring plan exists, can it be generated and executed automatically?
     - How to deal with a semi-automated solution, with additional user input?

Refactoring Plan Questions

- Given **a software system** as the source of the transformation, **a redesign proposal**, and **a set of refactorings** that can be used as transformation operations:
  1. Does a refactoring plan, which transforms the source, according to the redesign proposal, using the provided refactorings, exist?
     - additional non-refactoring transformations could be needed
  2. When a refactoring plan exists, can it be generated and executed automatically?
     - How to deal with a semi-automated solution, with additional user input?

Refactoring Plan Questions

- Given **a software system** as the source of the transformation, **a redesign proposal**, and **a set of refactorings** that can be used as transformation operations:
  1. Does a refactoring plan, which transforms the source, according to the redesign proposal, using the provided refactorings, exist?
     - additional non-refactoring transformations could be needed
  2. When a refactoring plan exists, can it be generated and executed automatically?
     - How to deal with a semi-automated solution, with additional user input?

Subproblems

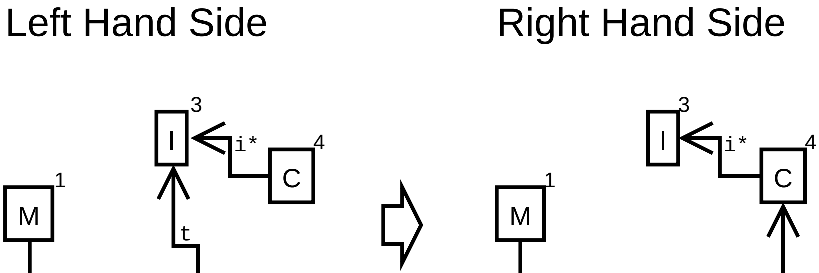
- We have divided the problem of **automatic generation of refactoring plans** in:
  - Definition and formalization of the “Refactoring Plan” concept
  - Representation of Software
  - Formalization of Refactorings
  - Elaboration of techniques to obtain refactoring plans

Formalising Refactorings

- Any refactoring formalization method must allow:
  - to deal with **system structure**.
  - to **check** behaviour preserving **conditions**.
- We will use **Graph Transformations** because:
  - Representing and managing structural information is straightforward with graphs.
  - This approach has already been validated (Mens et al., 2005).
- With Graph Transformation:
  - **Software** is represented as **graphs**.
  - **Refactorings** are represented as **graph transformation rules**.

Other refactoring formalization approaches:
  - First Order Logic (Kniessel, Köch, 2002).

Example of a Graph Transformation Rule



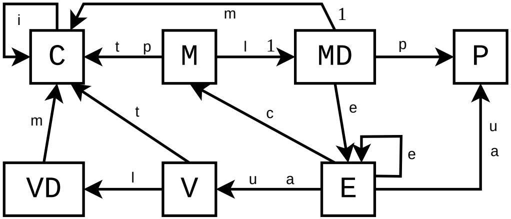
Software Representation: Program Graphs

- A graph representation for Object-Oriented Software is needed. We must represent:
  - elements of OO paradigm (classes, fields, methods, ...)
  - structural relationships
  - method bodies
- We have chosen the software representation part from the refactoring formalization of (Mens et al., 2005). This representation:
  - uses directed type graphs.
  - is language independent, lacking specific language constructions.
  - has been simplified to be as flexible as possible.

Software Representation: Program Graphs

- A graph representation for Object-Oriented Software is needed. We must represent:
    - elements of OO paradigm (classes, fields, methods, ...)
    - structural relationships
    - method bodies
- We have chosen the software representation part from the refactoring formalization of (Mens et al., 2005). This representation:
    - uses directed type graphs.
    - is language independent, lacking specific language constructions.
    - has been simplified to be as flexible as possible.

Software Representation: Java Program Graphs



Software Representation: Java Program Graphs

- For real systems, it is necessary to extend the graph format, adding:
  - elements for specific languages
  - more detailed representation of method bodies
- We have extended program graphs for Java: **Java Program Graphs**.
- Our graph representation format adds:
  - Java concepts such as visibility, interfaces, packages, . . .
  - More detailed representation of method bodies, with new node types, attributes and relationships.

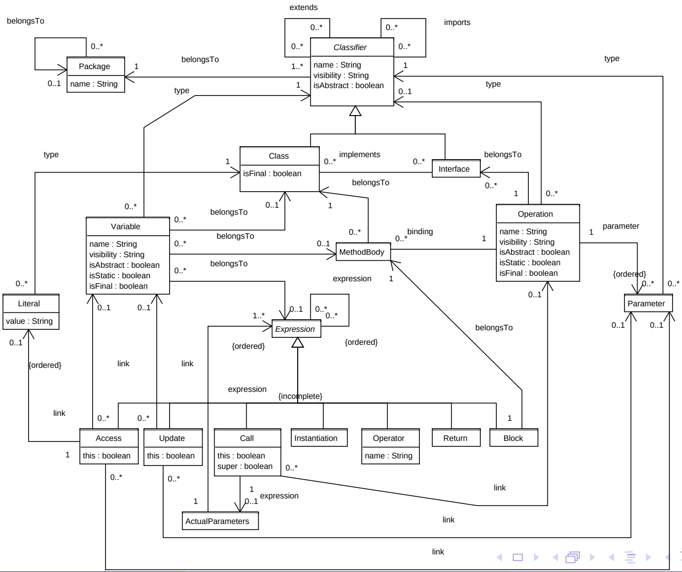
Software Representation: Java Program Graphs

- For real systems, it is necessary to extend the graph format, adding:
  - elements for specific languages
  - more detailed representation of method bodies
- We have extended program graphs for Java: **Java Program Graphs**.
- Our graph representation format adds:
  - Java concepts such as visibility, interfaces, packages, . . .
  - More detailed representation of method bodies, with new node types, attributes and relationships.

Software Representation: Java Program Graphs

- For real systems, it is necessary to extend the graph format, adding:
  - elements for specific languages
  - more detailed representation of method bodies
- We have extended program graphs for Java: **Java Program Graphs**.
- Our graph representation format adds:
  - Java concepts such as visibility, interfaces, packages, . . .
  - More detailed representation of method bodies, with new node types, attributes and relationships.

Software Representation: Java Program Graphs



Possible Approaches to Obtain Refactoring Plans

- We are exploring two aproaches:
  - Searching forwards
  - Searching backwards

Searching forwards

- **approach**
  - Suggested changes are turned into a simplified version of the sstem's desirable design.
  - Available refactorings are applied in a state space search way.
  - Refactoring pre and postconditions guide the search.
- **Advantages**
  - Every possible path is being explored
  - Relatively easy to implement
- **Problems**
  - Size of the state space
  - Possible infinite process

Searching Backwards

- **approach**
  - Dependencies between refactorings are computed
  - Iteratively, refactorings which enable the application of the desired change are added to the plan.
- **Advantages**
  - More efficient than searching backwards
- **Problems**
  - More difficult to implement with current Graph Transformation tools

Open questions

- Can complex refactorings be represented and analysed with current GT tools?
- Can searching be reduced to finite process?

Conclusions and Future Work

Conclusions

- Automatic generation of refactoring plans will provide very high level refactorings to improve the design of existing code.
- The Main subproblems and the research strategy have been introduced.
- Graph transformation can be used as the underlying formalism, specifically the programmed graph rewriting approach.
  - Representing Java programs with Java Program Graphs.
  - The graph transformation formalism could provide support to refactorings formal analysis, enabling searching for refactoring plans.

Conclusions

- Automatic generation of refactoring plans will provide very high level refactorings to improve the design of existing code.
- The Main subproblems and the research strategy have been introduced.
- Graph transformation can be used as the underlying formalism, specifically the programmed graph rewriting approach.
    - Representing Java programs with Java Program Graphs.
    - The graph transformation formalism could provide support to refactorings formal analysis, enabling searching for refactoring plans.

Conclusions

- Automatic generation of refactoring plans will provide very high level refactorings to improve the design of existing code.
- The Main subproblems and the research strategy have been introduced.
- Graph transformation can be used as the underlying formalism, specifically the programmed graph rewriting approach.
  - Representing Java programs with Java Program Graphs.
  - The graph transformation formalism could provide support to refactorings formal analysis, enabling searching for refactoring plans.

Future Work

- Main future tasks will be directed to:
  - Further definition of the “Refactoring Plan” concept.
  - Explore the expressivenss of GT tools
  - Analyse termination and correctness conditions of the searching approaches.

Towards a Framework for Software Design Defects Correction with Refactoring Plans

Javier Pérez
jperez@infor.uva.es

Universidad de Valladolid
Université de Mons-Hainaut

Fundamental Aspects of Software Evolution
FNRS Contact Group on Fundamental Computer Science
May 22nd 2008, University of Namur