C++- und Ruby- und ... Refactoring für Eclipse

iX-Konferenz Bessere Software! 2006

Prof. Peter Sommerlad
HSR - Hochschule für Technik Rapperswil
Institut for Software
Oberseestraße 10, CH-8640 Rapperswil
peter.sommerlad@hsr.ch
http://ifs.hsr.ch
http://wiki.hsr.ch/PeterSommerlad
Assumptions/Questions

- Talk in English or deutsch?
- I assume that you ask questions when you have them. Please interrupt me.
- I assume that you are not yet very familiar with the concept of Refactoring
  - short explanation is optional
- I assume that you are familiar with object technology and either C++, Ruby or Java
Peter Sommerlad
peter.sommerlad@hsr.ch

- **Work Areas**
  - Refactoring Tools (C++, Ruby,...)
  - Decremental Development (make SW 10% its size!)
  - Modern Software Engineering
  - Patterns
    - Pattern-oriented Software Architecture (POSA)
    - Security Patterns

- **Background**
  - Diplom-Informatiker (Univ. Frankfurt/M)
  - Siemens Corporate Research - Munich
  - itopia corporate information technology, Zurich (Partner)
  - Professor for Software HSR Rapperswil, Head Institute for Software

- **Credo:**
  - People create Software
    - communication
    - feedback
    - courage
  - Experience through Practice
    - programming is a trade
    - Patterns encapsulate practical experience
  - Pragmatic Programming
    - test-driven development
    - automated development
  - Simplicity
    - fight complexity
Overview

- **Introduction to Refactoring**
  - Elementary Refactorings (optional)

- **Refactoring Plugins**
  - typical approach

- **C++ Refactoring Plugins**
  - Why is it so hard?
  - Cerp Demo

- **Ruby Refactoring Plugin**
  - What is easier?
  - "Demo"

- **Outlook**
  - future Refactoring features and languages
  - RadRails, C++, Python, PHP, groovy, PL/1, Ada, Cobol...?
  - Funding?
Introduction
Refactoring

"Improving the Design of Existing Code" -- Martin Fowler

- after we learned to solve the programming problem, solve it in a better way.

- process:
  - smell "bad code"
  - then refactor it
  - run all (unit-) tests
Improving Design

• **improving means to eliminate violation of principles of good design, i.e.,**
  o bad names – uncommunicative code
  o duplication – DRY principle, OAOO, SPOT
  o lack of cohesion – too large
  o bad dependencies – wrong direction or circular

• **goals**
  o understandable program code
  o achieve conceptual integrity
  o simplicity – not simplistic
  o testability (see iX conference 2005)
Refactoring tools and Eclipse

• Java JDK provides a good support for refactoring automation
  o eases development
  o refactoring possible without too many unit tests

• Eclipse
  o provides language independent framework for refactoring tools
  o Development Toolkit plugins for many other languages available

Most non-Java Toolkit plugins lack adequate Refactoring support
Benefits and Drawbacks of automated Refactoring

• Fowler's book describes Refactoring as a manual process of (very) small steps
  o tedious work
  but
  o makes programmers think on what to achieve

• automated Refactoring
  o less tedious, easier to undo, no/less automatic re-testing required
  but
  o lulls programmers, can refactor in circles without code improvement, refactor without sense of good design, tools not perfect
Some Important Refactorings (0)

• Rename
  o good names are important
  o while we program, we learn better names for fields, variables, classes, methods or functions
  o renaming benefits heavily from automatic support
    ➢ find all places where this named program element is used by tracking references from a definition
    ➢ no need to blindly change the name string everywhere
    ➢ no need for manual checking if name means something else in another place
    ➢ better than sed –e 's/oldname/newname/g' 😊
Some Important Refactorings (1)

• Change Bidirectional Association with Unidirectional
  - lower coupling, remove circular dependency
  - no automation, requires thinking and planning
  - other simpler refactorings support this (ie. Move)
Some Important Refactorings (2)

• Extract Class
  o improve cohesion
  o smaller classes

```
C1
  field1
  field2
  m1()
  m2()

C2
  field2
  m2()
```
Some Important Refactorings (3)

- **Extract Method**
  - reuse code
  - less duplication

```
C1

+m()

m() {
  before();
  a();
  b();
  after();
}
```

```
extracted() {
  a();
  b();
}
```

```
C1

+m()
  -extracted()

m() {
  before();
  extracted();
  after();
}
```
Some Important Refactorings (4)

- **Move Field**
  - to better cohesion,
Some Important Refactorings (5)

• Move Method
  - move code where it belongs

C1
m()

C2

C1

C2
m()
Principles of Refactoring tools
Principles workings of Refactoring tools

• **based on abstract syntax trees AST**
  - internal representation of program code generated by a parser
  - with name bindings (know what a symbol means) aka symbol table

• **automated Refactoring mechanics (in Eclipse)**
  - ask for user input (e.g., new name)
  - check preconditions (correct selection)
  - modify AST
  - re-generate source code
  - check postconditions
  - visualize changes
  - perform changes
Hard parts of automated Refactoring

• Comments
  o typical parsers ignore comments
  o comments are often not part of an AST

• Language redundancy
  o different syntax for same semantics
  o often identical AST representation

• Preserve semantics and compilability
  o moving elements can change meaning
  o moving code can break compiles
     sequential nature of source code
AST parsing Example simplified

```
a = check ? "ok" : "failed" # some check
```

- `a` (lvar)
- `check` (lvar)
- "ok" (str_literal)
- "failed" (str_literal)
- `some check` (comment)
- `condition`
- `then_part`
- `else_part`
- `if_expr`
- `assignment`

Classic abstract syntax tree

Comment-extended abstract syntax tree
Creating Refactoring Plugin Infrastructure

**Tasks:**

- extend parser and AST to keep comments
  - JDT loses comments in some cases (or used to)
  - C++ especially hard with preprocessor

- generate code from AST without losing comments
  - pretty-printing in context, e.g., nesting levels
  - might need to rely on original formatting

- automate tests for this
  - have to deal with all syntax idiosyncrasies

- understand name bindings for necessary semantic checks for individual refactorings
  - can require extending AST
Re-generate Code from AST

```plaintext
a = if check
   "ok"
else
   "failed"
end # some check
```

```plaintext
: "failed" # some check
```
Additional features

• (re-)generating source code from an extended AST allows additional features

• Pretty printing
  o configurable consistent layout of source code

• Code generation
  o generate typical source code pieces
    ➢ i.e. generate getters/setters/accessors
  o actually changing the code's semantics
    ➢ generate constructor using fields
    ➢ override method
Our Projects
Our Refactoring projects

- CERP – simple C++ Refactoring
- Ruby Refactoring
- CDT – C++ Refactoring
- Python
- Rails
- PHP
- groovy
- your language here... (PL/1, ABAP, Cobol?)
C++ Refactoring in Eclipse

• CDT provides support for C++ development in Eclipse
  o mainly supported by companies providing development tools for embedded platforms
  o Refactoring not in their focus (yet)
  o CDT's evolution provides some interesting and duplicate design elements and features
    ➢ i.e. several parsers, several source representations

• C/C++ is especially hard to deal with
  o #includes, Macros, templates
  o hand-coded parsers – not grammar generated
  o redundancy of definitions with declarations
Example of C++ specific problems

• Preprocessor
  o `#include`
    ➢ classical compiler physically includes
  o comments
    ➢ already removed by preprocessor
  o macros
    ➢ can do strange things to syntax in original source

• Separate compilation units
  o `.c, .cpp files`
  o package and project concept by convention or separate tools
Eclipse CDT specific challenges

- **different generations of source code representation and corresponding parsers**
  - CModel - elementary information of position of elements
    - today deprecated
  - DOM – the AST of a single translation unit
    - "completely" parsed code representation
  - PDOM – "persistent" DOM
    - not yet complete (but almost now)
  - Rename-Refactoring – own parser and "DOM"
    - copy-paste reuse of DOM parser

- **neither represents comments or all preprocessing information**
Demo CERP

- Diploma thesis by Guido Zgraggen and Christian Cavegn
  - 8 weeks of intense work fall 2005
- uses CDT's legacy CModel to find pieces of code
- generates source code without relying on complete AST
  - usable, but problems with some C++ syntax
- "interesting" C++ Refactoring: synchronize Method
  - adjust Definition with Declaration or vice versa
- provides
  - Declare Method - generates method declaration from impl.
  - Extract Baseclass
  - Implement [from] Baseclass
  - Hide Method – make method private
  - Implement Method – generated from declaration
  - Separate Class – moves a class' definition into separate file
  - Synchronize Method [signature]
CERP Menu
Extract Baseclass Dialog

Methods to extract:
- ~ExtractBaseClass()
- void method1()
- void method2()
- void doStuff()
- void doOtherStuff()
Refactoring Preview Wizard

```
#include <iostream>

#define EXTRACTBASECLASS_H
#define EXTRACTBASECLASS_H

class ExtractBaseClass {
public:
    ExtractBaseClass();
    ExtractBaseClass();
    virtual void method1();
    virtual void method2();
    protected:
    void doStuff();
    void doOtherStuff();
    private:
    void methodA();
    void methodB();
    void print() {
        std::cout << "Extract Base Class Demo";
    }
    int xWert;
    int yWert;
};

#undef EXTRACTBASECLASS_H
```

```
#include "BaseClass.h"

class ExtractBaseClass : virtual protected BaseClass {
public:
    ExtractBaseClass();
    ExtractBaseClass();
    virtual void method1();
    virtual void method2();
    protected:
    void doStuff();
    void doOtherStuff();
    private:
    void methodA();
    void methodB();
    void print() {
        std::cout << "Extract Base Class Demo";
    }
    int xWert;
    int yWert;
};

#undef EXTRACTBASECLASS_H
```
Ruby Refactoring
Easier...

- **Ruby has a simpler syntax**
  - with some problems when re-arranging code
  - parentheses are optional in some places but required for disambiguating

- **Dynamic typing eases code generation and refactoring**
  - everything is an object

- **JRuby as parser needed adjustments**

- **RDT needed extension**

- **Rails team enthusiastic supporters**
Demo RDT Features

- Generate Accessors
- Generate Constructor using Fields
- Override Method
- Rename Local Variable
- Push Down Method
- Extract Method
- Extract Class
- Rename Class, Method, Field
- Inline Class, Method, Field
- Split Temporary Variable
- Move Method, Field
- Convert Local Variable to Field
- Merge Class' Parts from different Files
Extract Method Refactoring
Generate Accessors
Inline Temporary
Merge Class Parts

Select Class Parts

Salesman

Description
Check the parts of a class that should be merged. All parts will be merged together into the selected class part.

- getQuota
- multiplier

class Salesman < Employee
def getQuota
  @quota = 2
end

def multiplier
  1.5
end

class Engineer < Employee
def getQuota
  @quota
end
end
Push Down
rename local
Split Temporary Variable

temp = \( 2 \times (_\text{height} + _\text{width}) \)
puts temp

temp = _\text{height} \times _\text{width}
puts temp

Choose new names:
perimeter
area
Convert Local Variable to Field
Conclusion and Outlook
Difficulties of developing Refactoring plugins

- **Unit Testing**
  - workbench expensive to start
    - decouple your code and tests from workbench as good as possible
    - run tests as often as possible and as quick as possible
  - many tests required
    - write configurable tests
    - files with test data, not code with test cases
    - quick to add and extend tests

- "Fixing" and extending existing Plugins - Logistics
  - establish collaboration with plugin projects
  - de-couple from plugin projects

- **frequent releases of Eclipse platform and projects**
  - especially with volatile projects like CDT, hard to keep up
Our Refactoring projects

• CERP – simple C++ Refactoring
• Ruby Refactoring
• CDT – C++ Refactoring
• Python
• Rails
• PHP
• groovy
• your language here... (PL/1, ABAP, Cobol?)
Future Refactoring and Plugin projects @ IFS

- **incorporate Ruby Refactoring into official RDT and RadRails projects**
  - RubyOnRails-specific Refactorings

- **extend C++ CDT Refactoring**
  - still much work to do, but basic infrastructure available:
    - comments aware parser and AST
    - AST rewriting
    - virtual preprocessing keeps original source

- **Python Refactoring**
  - current semester student project

- **Projects, Plans and Ideas for**
  - Quick-Fixes for FindBugs™ (Java)
  - groovy, PL/1(?), ABAP (?), Cobol(?), Ada(?) - Refactoring
  - Business-Process Refactoring
Have fun Refactoring!

- Remember: **Refactor to simplify your code**

- or order specific Refactoring features for your organization

- Do not hesitate to contact me for further information and feedback:
  - peter.sommerlad@hsr.ch

Online-Resources:
- http://ifs.hsr.ch
- http://ifs.hsr.ch/cerp/updatesite - CERP
- http://r2.ifsoftware.ch – Ruby Refactoring Plugin