Refactoring

Software Engineering
Kent State University

Mathematics: Factor

- factor
  - One of two or more quantities that divides a given quantity without a remainder, e.g., 2 and 3 are factors of 6; a and b are factors of ab
- factoring
  - To determine or indicate explicitly the factors of

Software Engineering: Factoring

- factor
  - The individual items that combined together form a complete software system:
    - identifiers
    - contents of function
    - contents of classes and place in inheritance hierarchy
- factoring
  - Determining the items, at design time, that make up a software system
Refactoring

- Process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure [Fowler'99]
- A program restructuring operation to support the design, evolution, and reuse of object oriented frameworks that preserve the behavioral aspects of the program [Opdyke'92]

Specifics

- Source to source transformation
- Remain inside the same language, e.g., C++ to C++
- Does not change the programs behavior – according to a test suite
- Originally designed for object-oriented languages, but can also be applied to non-object oriented language features, i.e., functions

Levels of Software Changes

- High Level -
  - Features to be added to a system
  - e.g., New feature
- Intermediate Level
  - Change design (factoring)
  - e.g., Move a member function
- Low Level
  - Change lines of code
  - e.g., Changes in (a least) two classes
Relationship to Design

- Not the same as “cleaning up code”
  - May cause changes to behavioral aspects
  - Changes often made in a small context or to entire program
- Core practice in agile (XP) methodologies
- Views design as an evolving process
- Strong testing support to preserve behavioral aspects

Some Example Refactorings

- Introduce Explaining Variable
- Rename Method
- Move Method
- Pullup Method
- Change Value to Reference
- Remove Parameter
- Extract Hierarchy

Why: Design Preservation

- Code changes often lead to a loss of the original design
- Loss of design is cumulative:
  - Difficulties in design comprehension
  - Difficulties in preserving design
  - More rapid decay of design
- Refactoring improves the design of existing code
Why: Comprehension

- Developers are most concerned with getting the program to work, not about future developers
- Refactoring makes existing code more readable
- Increases comprehension of existing code, leading higher levels of code comprehension
- Often applied in stages

Why: Debugging

- Improved program comprehension leads to easier debugging
- Increased readability leads to the discovery of possible errors
- Understanding gained during debugging can be put back into the code

Why: Faster Programming

- Counterintuitive argument made by Fowler
- Good design is essential for rapid development
- Poor design allows for quick progress, but soon slows the process down
  - Spend time debugging
  - Changes take longer as you understand the system and find duplicate code
When?

- Adding Functionality
  - Comprehension of existing program
  - Preparation for addition
- Debugging
  - Comprehension of existing program
- Code Review
  - Preparation for suggestions to other programmers
  - Stimulates other ideas

Refactoring Catalog

- Collected by Fowler [Addison Wesley 2000]
- Refactoring entry composed of:
  - Name
  - Summary
  - Motivation
  - Mechanics
  - Examples
- Based on Java

Categories

- Composing Methods
  - Creating methods out of inlined code
- Moving Features Between Objects
  - Changing of decisions regarding where to put responsibilities
- Organizing Data
  - Make working with data easier
Categories (cont)

- Simplifying Conditional Expressions
- Making Method Calls Simpler
  - Creating more straightforward interfaces
- Dealing with Generalization
  - Moving methods around within hierarchies
- Big Refactorings
  - Refactoring for larger purposes

Composing Methods

- Extract Method
- Inline Method
- Inline Temp
- Replace Temp with Query
- Introduce Explaining Variables
- Split Temporary Variable
- Remove Assignments to Parameters
- Replace Method with Method Object
- Substitute Algorithm

Example: Remove Assignments to Parameters

```java
int discount (int inputVal, int quantity, int yearToDate)
{
    if (inputVal > 50) inputVal -= 2;
    ...
}

int discount (int inputVal, int quantity, int yearToDate)
{
    int result = inputVal;
    if (inputVal > 50) result -= 2;
    ...
}
```
Moving Object Features

- Move Method
- Move Field
- Extract Class
- Inline Class
- Hide Delegate
- Remove Middle Man
- Introduce Foreign Method
- Introduce Local Extension

Organizing Data

- Self Encapsulate Field
- Replace Data Value with Object
- Change Value to Reference
- Change Reference to Value
- Replace Array with Object
- Duplicate Observed Data
- Change Unidirectional Association to Bidirectional
- Change Bidirectional Association to Unidirectional

Organizing Data (cont)

- Replace Magic Number with Symbolic Constant
- Encapsulate Field
- Encapsulate Collection
- Replace Record with Data Class
- Replace Type Code with Class
- Replace Type Code with Subclasses
- Replace Type Code with State/Strategy
- Replace Subclass with Fields
Simplifying Conditional

- Decompose Conditional
- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Fragments
- Remove Control Flag
- Replace Nested Conditional with Guard Clauses
- Replace Conditional with Polymorphism
- Introduce Null Object
- Introduce Assertion

Example: Decompose Conditional

```java
if (date.before(SUMMER_START) || date.after(SUMMER_END))
    charge = quantity * _winterRate + _winterServiceCharge;
else
    charge = quantity * _summerRate;

if (notSummer(date))
    charge = winterCharge(quantity);
else
    charge = summerCharge(quantity);
```

Simplifying Method Calls

- Rename Method
- Add Parameter
- Remove Parameter
- Separate Query from Modifier
- Parameterize Method
- Replace Parameter with Explicit Methods
- Preserve Whole Object
- Replace Parameter with Method
Simplifying Method Calls (cont)
- Introduce Parameter Object
- Remove Setting Method
- Hide Method
- Replace Constructor with Factory Method
- Encapsulate Downcast
- Replace Error Code with Exception
- Replace Exception with Test

Dealing with Generalization
- Pull Up Field
- Pull Up Method
- Pull Up Constructor Body
- Push Down Method
- Push Down Field
- Extract Subclass
- Extract Superclass
- Extract Interface

Dealing with Generalization (cont)
- Collapse Hierarchy
- Form Template Method
- Replace Inheritance with Delegation
- Replace Delegation with Inheritance
Example: Push Down Method

Example: Extract Subclass

Example: Extract Interface
Big Refactorings

- Tease Apart Inheritance
  - Split an inheritance hierarchy that is doing two jobs at once
- Convert Procedural Design to Objects
- Separate Domain from Presentation
  - GUI classes that contain domain logic
- Extract Hierarchy
  - Create a hierarchy of classes from a single class where the single class contains many conditional statements

Convert Procedural Design

- Take each record type and turn it into a dumb data object with accessors
- Take all procedural code and put it into a single class
- Take each long method and apply Extract Method and the related factorings to break it down. As you break down the procedures use Move Method to move each one to the appropriate dumb data class
- Continue until all behavior is removed from the original class

Example
Extract Method

- Create a new method, and name it after the intention of the method (name it by what it does, not by how it does it)
- Copy the extracted code from the source method into the new target method
- Scan the extracted code for references to any variables that are local in scope to the source method

Extract Method (cont)

- See whether any temporary variables are used only within this extracted code. If so, declare them in the target method as temporary variables
- Look to see whether any local-scope variable are modified by the existing code (See Split Temporary Variable and Replace Temp with Query)
- Pass into the target method as parameters local scope variables that are read from the extracted code

Extract Method (cont)

- Compile when you have dealt with all the locally-scoped variables
- Replace the extracted code in the source method with a call to the target method
- Compile and test
Tools

- Smalltalk Refactoring Browser
  - Development environment written in Smalltalk
  - Allows for Smalltalk source code to transform Smalltalk source code
  - Comments as a first-class part of the language
- XRefactory
  - Allows standard refactorings for C++

Challenges

- Preservation of documentary structure (comments, white space etc.)
- Processed code (C, C++, etc.)
- Integration with test suite
- Discovery of possible refactorings
- Creation of task-specific refactorings

Limitations

- Tentative list due to lack of experience
- Database
  - Database schema must be isolated, or schema evolution must be allowed
- Changing Published Interfaces
  - Interfaces where you do not control all of the source code that uses the interface
  - Must support both old and new interfaces
  - Don't publish interfaces unless you have to
Design Patterns

- Source
  - Design Patterns: Discovered in source designs
  - Refactorings: Discovered in informal version histories of source code
- Provides standard names
  - Design Patterns: GOF
  - Refactorings: Fowler's Catalog

Design Patterns (cont)

- Provides Mechanics
  - Design Patterns: Static program structure
  - Refactorings: Algorithmic changes
- Simple to Complex
  - Design Patterns: Complex patterns are a combination of simpler patterns
  - Refactorings: Complex refactorings are a combination of simpler refactorings