Refactoring

Recap: Class Diagrams
- Class diagrams represent design structure
- Three parts: name, attribute, operations
- Visibility, attribute type, multiplicity
- Association, association multiplicity
- Generalization i.e. interface impl, subclassing
- Composition i.e. class A contains class B
- Applied information hiding, DSM, layering
Recap: Interaction Diagrams

- Represent functionality
- Two major parts:
  - Life line boxes (represent life time of an object)
  - Messages between objects (instances)
- Also looked at some special case notations
  - Instances: parameterized instances, singleton instances, array elements
  - Messages: creation, destruction, conditional messages, mutually exclusive conditional messages, asynchronous messages, polymorphic messages
  - Abstractions: looping, iteration over collection, nesting of frames, decomposition

Recap: Responsibility Assignment

- Information Expert
- Creator
- Controller
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations
Eclipse Tutorial

- Open source IDE
- Available from http://www.eclipse.org
- Lot of interesting & advance features
- Demo: refactoring features in eclipse
  - Extract method
  - Pull up method
  - Push down method

Refactoring

- Reorganize instructions in a programs, perhaps rewriting
- Key: Preserve semantics of the program
- Objective: Improve structure (aka design), improve readability
- Doesn’t add/remove any functionality
Value Argument

- Refactoring consumes resources
  - Doesn’t add any new feature
  - Impact on near term profit = 0

- Why Improve Structure?

Value Argument

- Readability
  - More readable less chances of inadvertent errors
  - Adding features in future will be less cumbersome
  - Invest today, reap later
Is semantics preserved?

- Informal & imprecise verification: Write test suite
  - Rigorous test suite development before refactoring
  - Test after you refactor

- Question: Why do I call it imprecise verification?

- Formal verification: Prove that certain transformation always preserves the semantics

Refactoring that you don't see

- Compiler optimizations
  - Refactoring for performance improvement

Example code:

```c
radius = ..;
while(index < Length){
    array[index] = 2 * pi * radius;
}
```
Loop Invariant Optimization

radius = ..;
perimeter = 2 * pi * radius
while(index < Length){
array[index] = perimeter;
}

- Compiler optimizations go through precise verification

Common Refactoring Techniques

- Extract method
- Pull up method
- Push down method
- Consolidate conditional expressions
- Encapsulate Collection
- ...
- Alphabetical list at:
Extract Method: The Problem

Pull Up Method/Field/Constructor

- Methods with identical results on subclasses

Fowler: Refactoring, Page 322

- Move the method to superclass
Push Down Method

- Behavior on a superclass is relevant only for some of its subclasses.

  ![Diagram](image.png)

  **Fowler: Refactoring - Page 328**

- Move the behavior to the subclass

Consolidate Conditional Expression

```java
double disabilityAmount() {
    if (seniority < 2) return 0;
    if (monthsDisabled > 12) return 0;
    if (!isPartTime) return 0;
    // compute the disability amount
}
```

```java
double disabilityAmount() {
    if (!isNotEligableForDisability()) return 0;
    // compute the disability amount
}
```

**Fowler: Refactoring - Page 240**
Encapsulate Collection

- Method returns a collection
  - Breaks encapsulation
  - Allows clients to manipulate collection without objects’ knowledge

![Diagram showing method changes]

**Fowler: Refactoring - Page 208**

- Make a read-only view, provide methods to add remove elements

---

Summary

- Improve the structure of code
  - No value gain at the moment
  - Easier to add features later
  - Less chances of errors in maintenance tasks

- Key is to preserve semantics
  - Imprecisely ensure that by developing tests
  - Also, by code inspection
Exercise

- Can some refactoring techniques change the design structure matrix view of the system?

- If yes, which refactoring techniques change the DSM view and how?