Design Patterns and Refactoring
CSE 403

Outline
- Design Patterns
- Refactoring
- Refactoring patterns

Resources
- CSE 503 Sp ’04 lecture, CSE 403 Sp ’05
- Gamma, Helm, Johnson, Vlissides ("Gang of four"): Design Patterns: Elements of reusable object-oriented software
- Shalloway and Trott: Design Patterns Explained
- Martin: Agile Software Development

Design Patterns
- Is design mostly *routine* or *innovative*?
- Design Patterns are a way of recording design knowledge
- Christopher Alexander first described patterns in architecture

What is a pattern
- Pattern name
- Problem
- Solution
- Consequences

Gang of Four patterns

<table>
<thead>
<tr>
<th>Scope</th>
<th>Basic</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>factory method</td>
<td>structural</td>
</tr>
<tr>
<td>object</td>
<td>abstract factory</td>
<td>adapter (object)</td>
</tr>
<tr>
<td></td>
<td>builder</td>
<td>bridge</td>
</tr>
<tr>
<td></td>
<td>prototype</td>
<td>composite</td>
</tr>
<tr>
<td></td>
<td>singleton</td>
<td>decorator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>facade</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flyweight</td>
</tr>
<tr>
<td></td>
<td></td>
<td>proxy</td>
</tr>
</tbody>
</table>


Problem: delay choice of type

Typical OOP program hard-codes type choices

```c
void AppInit () {
    #if MAC
        Window w = new MacWindow(...);
        Button b = new MacButton(...);
    #else
        Window w = new XpWindow(...);
        Button b = new XpButton[...];
    #endif
    w.Add(b);
}
```

We want to easily change the app’s "look and feel", which means calling different constructors.

Factory method

Wrap the constructors in "factory methods"

```c
class LookAndFeelFactory {
    LookAndFeelFactory () {
        Window CreateWindow (...);
        Button CreateButton (...);
    }

    void AppInit (LookAndFeelFactory factory) {
        Window w = factory.CreateWindow(...);
        Button b = factory.CreateButton(...);
        w.Add(b);
    }
}
```

Problem: selection of an algorithm depends on client or data

- You have a set of algorithms that do basically the same thing, but implemented differently
- Want to separate the algorithm from the implementation

Strategy

- A Strategy specifies the interface for how the different algorithms will be used
- Concrete strategy classes implement the algorithms
- Context forwards client requests to appropriate concrete strategy
- Example: Sockets

Refactoring: Motivational Examples

- What is common among the following?
  1. \( x = ((p<=1) ? (p?0:1) : (p==4)?2:(p+1)); \)
  2. while (*a++ = *b++) {
  3. \( 1 + 1/1 + 1/(1+1/(1+1)) + ... \) = ?

Refactoring – What Is It?

- What is refactoring?
  - Modifying code to improve its structure without changing functionality
    - "the 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)

- What is the opposite of refactoring?

- Why might one want to do it?
Refactoring – Why Do It?

Why is it necessary?
- A long-term investment in the quality of the code and its structure
- Without proper maintenance, code tends to "rot" as its structure deteriorates when quick last-minute fixes are made and unplanned features are added
- Doing no refactoring may save on costs in the short term but pays huge interest in the long run
  "Don't be penny-wise but hour-foolish!"

Why fix it if it ain't broken? Every module has three functions:
- (a) to execute according to its purpose;
- (b) to afford change;
- (c) to communicate to its readers.
  If it doesn't do one or more of these, it's broken.

Refactoring – When to Do It?

- Refactoring is necessary from a business standpoint too
  - Helps with predictable schedules and high output at lower cost
  - ROI for improved software practices is 500% (1) or better
  - By doing refactoring a team saves on unplanned defect-correction work

- When is refactoring necessary?
  - Best done continuously, along with coding and testing
  - Very hard to do late, much like testing
  - Often done before plunging into version 2

Types of Refactoring

- Renaming (methods, variables)
- Naming (extracting) "magic" constants
- Extracting common functionality into a service / module / class / method
- Extracting code into a method
- Changing method signatures
- Splitting one method into several to improve cohesion and readability (by reducing its size)
- Putting statements that semantically belong together near each other
- Exchanging risky language idioms with safer alternatives
- Clarifying a statement (that has evolved over time or that is hard to "decipher")
- Performance optimization


Refactoring patterns

- From
  http://industriallogic.com/xp/refactoring/catalog.html
- E.g., Chain Constructors, Extract Adapter, Introduce Null Object, Replace Conditional Logic with Strategy

Chain constructors

- Problem: You have constructors that contain duplicate code.
- Chain the constructors together to obtain the least duplicate code.
Summary: Top Reasons for Refactoring

- Improving readability (and hence productivity)
- Responding to a change in the spec/design by improving cohesion
  - Or anticipating such a change

“If bug rates are to be reduced, each function needs to have one well-defined purpose, to have explicit single-purpose inputs and outputs, to be readable at the point where it is called, and ideally never return an error condition.” Steve Maguire -- “Writing Solid Code”

Language Support for Refactoring

- Modern development environments (e.g., Eclipse) support:
  - variable/method/class renaming
  - method or constant extraction
  - extraction of redundant code snippets
  - method signature change
  - extraction of an interface from a type
  - method inlining
  - providing warnings about method invocations with inconsistent parameters
  - help with self-documenting code through auto-completion

- Older development environments (e.g., vi, Emacs, etc.) have little or no automated support
  - Discourages programmers from refactoring their code