Lecture 3
Parnas’ Information Hiding
Announcement

- SSE: Students in Software Engineering
  - http://www.edge.utexas.edu/sse/
- Software Engineering Reading Group
  - 11AM - 12PM on every other friday
Announcement

- Reading assignments are due 10 PM. Sorry for confusion last time!
- We will grade the best 20 reviews instead of 24 reviews to give you more time for your project.
Announcement

• Tool evaluation paper expectation
• Tutorials on program representations are linked. (Review them before Lecture 8.)
• KWIC example code is available on the blackboard
• Can you post a thread on the discussion board?
Announcement

- Exemplary literature survey papers and tool evaluation papers are on the blackboard.
- Questions?
Example Project (1): History-based Code Completion

- Motivation: Programmers need assistance in remembering long API names. Most modern IDEs provide code completion feature to increase productivity and to prevent compilation errors.

- Problem: existing code completion algorithms only suggest candidate APIs based on starting alphabets but do not consider history of which code completion suggestions that programmers took in the past.

- Approach: propose a new algorithm that considers the history of which code completion suggestions programmers took in the past.
Example Project (1): *History-based Code Completion*

- **Implementation:** implement a new code completion algorithm in Eclipse

- **Evaluation Plan:** Download some code from OSS, remove some API calls and see which APIs are suggested by your algorithm and compare those suggestions with the ones suggested by the default code completion algorithm in Eclipse

- **Identify strengths and weaknesses of your algorithm and suggest future directions**
Example (2): Library Installation Suggestion based on Open Source

- **Motivation:** When programmers download open source software, they often subsequently need to identify and install libraries as some libraries’ source code cannot be released together due to licensing reasons.

- **Problem:** Programmers currently do not have much support other than reading README files and searching for needed libraries on the web. Even when they find libraries, their versions may not be compatible with the current version of software.
Example (2): Library Installation Suggestion based on Open Source

- **Approach:** Your web-service takes the URL of OSS and README files or a web-manual. It does some keyword analysis to identify which libs are required. It automatically runs a google search to locate these libraries and rank and suggestion them. Your web-service can also accommodate users’ input and maintain a set of compatible configurations.

- **Evaluation Plan:** Download some OSSs. Install them yourself by manually finding required libraries and checking them by running the application. Compare that results with your system’s suggestions.

- **Identify strengths and weaknesses of your algorithm and suggest future directions**
Example Project (3): Suggestion of when to refactor

- Motivation: Code decays without refactoring.
- Problem: Programmers need to know when to refactor. Refactoring tends to take a low priority.
- Return on refactoring investment depends when to refactor.
Example Project (3): Suggestion of when to refactor

- **Approach:** Identify code smells and suggest refactoring opportunities. Use metrics to identify bad smells. Program invariants (precondition/post conditions)

- **Implementation:**

- **Evaluation plan:** Compare with existing IDE-refactorings. Users studies in real tasks.
Example Project (3): Suggestion of when to refactor

- **Motivation:** Programmers often need to refactor to prevent code decay due to duplicated code.

- **Problem:** The return on refactoring investments depends on how often those code actually require similar changes. If programmers refactor too early, the refactoring may turn out to be unnecessary. If programmers refactor too late, the return on refactoring investments may be marginal.
Example Project (3): Suggestion of when to refactor

• Approach: We propose an algorithm that recommends the appropriate timing for refactoring code duplicates based on their change history.

• Implementation:
  • propose several algorithms for recommending when to refactor code

• Evaluation plan:
  • Apply your algorithm to OSS history and produce cost-benefit models
Parnas’ Information Hiding

• What problem did Parnas discuss in the paper?
Modularization

• What does Parnas mean by a “module?”
• What do you mean by a “module” in practice? an object, or class
Modularization

• Expected Benefits

• Unexpected Pitfalls?
KWIC Requirements

- Input: an ordered set of lines where
  - each line is an ordered set of words
  - each word is an ordered set of characters
- Output: all circular shifts of all lines in alphabetical order
KWIC Requirements

• Input: an ordered set of lines where each line is an ordered set of words and each word is an ordered set of characters
  • My name is Miryung Kim
  • Software Evolution
• All circular shifts of all lines
  • My name is Miryung Kim
  • name is Miryung Kim My
  • is Miryung Kim My name
  • Miryung Kim My name is
  • Kim My name is Miryung
  • Software Evolution
  • Evolution Software
KWIC Requirements

- All circular shifts of all lines in alphabetical order
  - *Evolution Software*
  - *Kim My name is Miryung*
  - *Miryung Kim My name is*
  - *My name is Miryung Kim*
  - *Software Evolution*
  - *is Miryung Kim My name*
  - *name is Miryung Kim My*
Modularization 1
Modularization 2

Master Control

Input
Circular Shift
Alphabetizing
Output
Line Storage
Comparison

Modularization 1
Nodes: 5
Edges: 9

Modularization 2
Nodes: 6
Edges: 10
What are differences between two alternative designs?

• Both are decompositions.
• Both share data representations and access methods.
• Is the modularization 1 bad? Why?
## Changeability Assessment: Modularization 1

<table>
<thead>
<tr>
<th>Changes</th>
<th>MasterControl</th>
<th>Input</th>
<th>CircularShift</th>
<th>Alphabetizer</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>InputFormat</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A Single Storage</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Packing characters</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Index for CS</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Search or Partial</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
### Changeability Assessment: Modularization 2

<table>
<thead>
<tr>
<th>Changes</th>
<th>MasterControl</th>
<th>Input</th>
<th>CircularShift</th>
<th>Alphabetizer</th>
<th>Output</th>
<th>LineStorage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Format</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A Single Storage</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Packing characters</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Index for CS</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Search or Partial</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
## Changeability Comparison

### Modularization 1:

<table>
<thead>
<tr>
<th>Changes</th>
<th>Master Control</th>
<th>Input</th>
<th>Circular Shift</th>
<th>Alphabetizer</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Format</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A Single Storage</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Packing characters</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Index for CS</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Search or Partial Alphabetizer</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

### Modularization 2:

<table>
<thead>
<tr>
<th>Changes</th>
<th>Master Control</th>
<th>Input</th>
<th>Circular Shift</th>
<th>Alphabetizer</th>
<th>Output</th>
<th>Line Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Format</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A Single Storage</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Packing characters</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Index for CS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Search or Partial Alphabetizer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
Independent Development

- Modularization 1: The decision to store line indices and word indices must be communicated among all module developers
- Modularization: API names and types
Functional Decomposition vs. Information Hiding

- Functional decomposition (Flowchart approach)
  - Each module corresponds to each step in a flow chart.

- Information Hiding
  - Each module corresponds to a design decision that are likely to change and that must be hidden from other modules.
  - Interfaces and definitions were chosen to reveal as little as possible.
Connecting Design Principles to Source Code for Improved Ease of Change

Vibha Sazawal
Department of Computer Science and Engineering
University of Washington

Now a professor at University of Maryland, College Park
These slides are borrowed from Dr. Sazawal’s talk.
The design snippets approach

Goals

- help programmers make decisions related to ease of change
- remain easy to use in the context of existing code

Insight: these goals can be achieved by

- partial views of a system
- that are co-displayed with code, and
- provide a bridge between code and design principles

These views are called design snippets.
• Four specific types of design snippets
  – derived from the **information hiding** principle
    * information hiding snippet
    * type assumptions snippet
  – derived from the **low coupling** principle
    * dependencies snippet
    * de facto interfaces snippet
Design principles: information hiding and low coupling

Information hiding [Parnas72, Parnas84]

- “details that are likely to change should be the secrets of separate modules”
- “the only assumptions that should appear in the interfaces between modules are those that are considered unlikely to change”

Low coupling [Yourdon78]

- helps reduce the effects of interface change
- helps programmers extract subsets of systems
Problem: a gap between design principles and code

There is no direct mapping between design terms (such as secret, volatile, and assumption) and code.

The gap between design principles and code

• complicates adherence to design principles
• results in design decision errors
Design snippets bridge the gap

Design snippets approach

- accommodate common mappings between design principles and Java code
  - example: module $\Rightarrow$ class
- present information that is
  - needed to follow design principles
  - relevant to the current Java file
**Information hiding snippet**

**Goal:** help programmers hide implementation details

**Side-by-side view:** for comparison of interface and implementation

**Secret types:** non-parameter, non-field types used by a class
Secret types, together with private members, provide a useful, succinct view of implementation details.

```
- Implementation of Bank
  - private fields
    - m_currency_manipulator has type CurrencyManipulator
  - Initializers
  - secret types used (broken down by method)
    - static initializer 1 uses
      - class USDollarsManipulator (calls constructor)
    - main(String[]) uses
      - class BankAccount (calls makeWithdrawal, calls constructor, calls getBalance, calls makeDeposit)
      - class Money (passed to method makeWithdrawal, passed to method makeDeposit, calls constructor)
      - class BankException (catches instance, calls getMessage)
      - class System (accesses std)
      - class PrintStream (calls print)
      - class String (passed to method print)
```
**Type assumptions snippet**

**Goal:** identify assumptions that may violate information hiding
- **casts** to parameters and return values

Why focus on type assumptions?
- often symptoms of larger problems with information sharing
- casts in client code are often hidden to maintainers
Recap

- Information Hiding principle means “identify design decisions that are likely to change and hide them within each module.”

- It does not mean using OO language, using abstract data types, using built-in libraries, using of message passing, etc.

- But what happens if you cannot anticipate what are likely to change?