Today

• Part 1 (Discussion)(~60 mins)
  • Discussion of readings

• Break!

• Part 2 (Lecture)(60 mins)
  • Editing Code

• Part 3 (In class activity)(~20 mins)
  • Design exercise
Editing Code

• What types of edits do developers make?
• What mistakes occur? How can they be prevented?
• How can developers edit at a level of abstraction beyond lines and characters?

• Techniques we will examine today
  • Structured editors
  • Editable program views
  • Copy & paste reuse
  • Refactoring
  • Exploratory programming
Structured Editors: Motivation

• Syntax can be hard
  • Have to learn the right syntax (challenging for programming or language novices)
  • Getting syntax wrong creates errors

• What if we could have a development environment where it was impossible to have a syntax error
Structured Editors: Idea

- Developers edit code through commands that create program elements
  - e.g., create an if statement through a keyboard shortcut or drag & drop

- Edits are semantic rather than syntactic
  - Individual elements expose specific elements they support
  - Cannot make edits that crosscut element structure
Cornell Program Synthesizer

- Introduced key concepts

```
IF (condition)
  THEN statement
ELSE statement
```

```
IF (k > 0)
  THEN statement
ELSE PUT SKIP LIST (‘not positive’);
```

What happened?

- Structured editors make unstructured edits hard
  - Hard to add / remove lines that crosscut structure
  - Hard to copy and paste in ways that crosscut structure
  - If you already know the syntax, may be slower to select syntax from command or drag and drop than it is to type

- But… if you don’t know the syntax at all, can be helpful
  - ➔ Extensive use of syntax directed editors in programming environments for novice programmers
Example: Alice

http://www.alice.org/3.1/Materials/Videos/01.BriefTour.mp4

Alice: Lessons Learned from Building a 3D System for Novices. Matthew Conway, Steve Audia, Tommy Burnette, Dennis Cosgrove, Kevin Christiansen, Rob Deline, Jim Durbin, Rich Gossweiler, Shuichi Kogi, Chris Long, Beth Mallory, Steve Miale, Kristen Monkaitis, James Patten, Jeffrey Pierce, Joe Schochet, David Staak, Brian Stearns, Richard Stoakley, Chris Sturgill, John Viega, Jeff White, George Williams, and Randy Pausch, CHI 2000
Example: Scratch

https://vimeo.com/65583694

Example: TouchDevelop

https://www.touchdevelop.com/home
https://www.youtube.com/watch?v=ve2E90wh-wk
Editable program views

- Expressing code edits through textual changes can be time consuming
  - extra boilerplate, code duplication, etc.

- Key idea: Enable developers to instead interact with abstracted view of code
  - Use edits to abstract view to edit underlying code
Registration-based language abstractions

Copy & paste code reuse

- A very common way to edit code is by copying existing code. —> copy & paste reuse
- Creates code duplication
  - But… ok if this code duplication does not represent new abstraction

- Studies have attempted to understand when code duplication introduced by copy & paste is bad

- Many tools to detect code clones introduced by copy & paste

Slides for this section adapted from 05-899D Human Aspects of Software Development Spring 2011, “Software Evolution” by YoungSeok Yoon
Why do developers copy & paste code?

- structural template (the most common intention)
  - relocate, regroup, reorganize, restructure, refactor
- semantic template
  - design pattern
  - usage of a module (following a certain protocol)
  - reuse a definition of particular behavior
  - reuse control structure (nested if-else or loops)

Why do developers copy & paste?

- Forking
  - Hardware variations
  - Platform variation
  - Experimental variation
- Templating
  - Boiler-plating due to language in-expressiveness
  - API/Library protocols
  - General language or algorithmic idioms
- Customization
  - Bug workarounds
  - Replicate and specialize

Properties of copy & paste reuse

• Unavoidable duplicates (e.g., lack of multiple inheritance)

• Programmers use their memory of C&P history to determine when to restructure code
  • delaying restructuring helps them discover the right level of abstraction

• C&P dependencies are worth observing and maintaining

Code clone genealogies

- Investigates the validity of the assumption that code clones are bad
- Defines clone evolution model
- Built an automatic tool to extract the history of code clones from a software repository

Table 1: Description of Two Java Subject Programs

<table>
<thead>
<tr>
<th>Program</th>
<th>carol</th>
<th>dnsjava</th>
</tr>
</thead>
<tbody>
<tr>
<td>URL</td>
<td>carol.objectweb.org</td>
<td><a href="http://www.dnsjava.org">www.dnsjava.org</a></td>
</tr>
<tr>
<td>LOC</td>
<td>7878 ~ 23731</td>
<td>5756 ~ 21188</td>
</tr>
<tr>
<td>duration</td>
<td>26 months</td>
<td>68 months</td>
</tr>
<tr>
<td># of check-ins</td>
<td>164</td>
<td>905</td>
</tr>
</tbody>
</table>

Table 2: Clone Genealogies in carol and dnsjava

\( min_{token} = 30, sim_{th} = 0.3 \)

<table>
<thead>
<tr>
<th></th>
<th>carol</th>
<th>dnsjava</th>
</tr>
</thead>
<tbody>
<tr>
<td># of genealogies</td>
<td>122</td>
<td>140</td>
</tr>
<tr>
<td>false positive</td>
<td>13</td>
<td>15</td>
</tr>
<tr>
<td>true positive</td>
<td>109</td>
<td>125</td>
</tr>
<tr>
<td>locally unfactorable</td>
<td>70 (64%)</td>
<td>61 (49%)</td>
</tr>
<tr>
<td>consistently changed</td>
<td>41 (38%)</td>
<td>45 (36%)</td>
</tr>
</tbody>
</table>

Refactoring: Motivation

“Refactoring is 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 1999]


Slides for this section adapted from 05-899D Human Aspects of Software Development Spring 2011, “Software Evolution” by YoungSeok Yoon
First tool: A Refactoring Tool for Smalltalk

(Very) brief story of refactoring

• Started with academic work defining idea of refactoring
• Academic work for tools quickly followed (e.g., [Brant TPOS97])
  • Built in real IDE for Smalltalk from beginning
• Disseminated by agile thought leaders like Martin Fowler
• Adopted into mainstream IDEs like Eclipse, Visual Studio
• Became standard accepted feature of IDES
• Research continued
  • Do developers use refactoring tools?
  • Could they use them more?
  • How could refactoring tools better support developers?
Developers manually perform refactorings not yet supported by tools

- About 70% of structural changes may be due to refactorings
- About 60% of these changes, the references to the affected entities in a component-based application can be automatically updated
- State-of-the-art IDEs only support a subset of common low-level refactorings, and lack support for more complex ones

<table>
<thead>
<tr>
<th>Type of refactoring</th>
<th># detected</th>
<th>Eclipse support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convert anonymous class to nested²</td>
<td>12</td>
<td>√</td>
</tr>
<tr>
<td>Convert nested type to top-level</td>
<td>19</td>
<td>√</td>
</tr>
<tr>
<td>Convert top-level type to nested</td>
<td>20</td>
<td>×</td>
</tr>
<tr>
<td>Move member class to another class</td>
<td>29</td>
<td>√</td>
</tr>
<tr>
<td>Extract package</td>
<td>16</td>
<td>×</td>
</tr>
<tr>
<td>Inline package</td>
<td>3</td>
<td>×</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type of refactoring</th>
<th># detected</th>
<th>Eclipse support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pull up field/method</td>
<td>279</td>
<td>√</td>
</tr>
<tr>
<td>Push down field/method</td>
<td>53</td>
<td>√</td>
</tr>
<tr>
<td>Extract interface</td>
<td>28</td>
<td>√</td>
</tr>
<tr>
<td>Extract superclass</td>
<td>15</td>
<td>×</td>
</tr>
<tr>
<td>Extract subclass</td>
<td>4</td>
<td>×</td>
</tr>
<tr>
<td>Inline superclass</td>
<td>4</td>
<td>×</td>
</tr>
<tr>
<td>Inline subclass</td>
<td>7</td>
<td>×</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type of refactoring</th>
<th># detected</th>
<th>Eclipse support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extract constant interface</td>
<td>5</td>
<td>√</td>
</tr>
<tr>
<td>Inline constant interface</td>
<td>2</td>
<td>×</td>
</tr>
<tr>
<td>Extract class</td>
<td>95</td>
<td>×</td>
</tr>
<tr>
<td>Inline class</td>
<td>31</td>
<td>×</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type of refactoring</th>
<th># detected</th>
<th>Eclipse support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Information hiding</td>
<td>751</td>
<td>×</td>
</tr>
<tr>
<td>Generalize type</td>
<td>107</td>
<td>√</td>
</tr>
<tr>
<td>Downcast type</td>
<td>85</td>
<td>×</td>
</tr>
<tr>
<td>Introduce factory</td>
<td>19</td>
<td>√</td>
</tr>
<tr>
<td>Change method signature</td>
<td>4497</td>
<td>√</td>
</tr>
<tr>
<td>Introduce parameter object</td>
<td>4</td>
<td>×</td>
</tr>
<tr>
<td>Extract method*</td>
<td>45</td>
<td>√</td>
</tr>
<tr>
<td>Inline Method*</td>
<td>31</td>
<td>√</td>
</tr>
</tbody>
</table>

Larger study by Murphy-Hill

- Extensive study using 4 data sets spanning
  - > 13,000 developers, > 240,000 refactorings
  - > 2500 developer hours, > 3400 commits

- Data sets
  - Users (collected by Murphy et al. in 2005)
  - Everyone (collected by Eclipse Usage Collector)
  - Toolsmiths (refactoring tool developers)
  - Eclipse CVS

Results

• The Rename refactoring tool is used much more frequently by ordinary programmers than by the toolsmiths.
• About 40% of refactorings performed using a tool occur in batches (i.e., refactorings of the same kind within 60 secs).
• About 90% of configuration defaults or refactoring tools remain unchanged when programmers use the tools.
• Messages written by programmers in commit logs do not reliably indicate the presence of refactoring.
• Programmers frequently *floss refactor* (i.e., interleave refactoring with other programming activities).
• About half of the refactorings are not high-level. —> refactoring detection tools that look exclusively for high-level refactorings will not detect them.
• Refactorings are performed frequently.
• Almost 90% of refactorings are performed *manually*, without the help of tools.
• The kind of refactoring performed with tools differ from the kind performed manually.
Exploratory Programming

• Developers sometimes explore programs without knowing a priori what behavior they want to create or the best way to implement it

• Goal: enable developers to explore variations in programs
Backtracking in programming

Fig. 1. An example of a node evolution history, which contains three backtracking instances. The node first appeared in the code as “getHeight();” (v1), changed a few times (v2 through v5), and finally ended up back at the original code (v6). The different contents are symbolized as capital letters A, B, and C. There are three backtracking instances in this node history indicated as black backward arrows.

Fig. 2. An example output of our analyzer, showing the history of a statement node. Each row maps to each version (v1, v2, ..., v5). This node contains a single backtracking instance, which is v1...v5. The edit operation IDs were originally 6-digits long (e.g., 184263), but were shortened for brevity.

Fig. 3. Distribution of all the detected backtracking sizes

Fig. 5. The identified backtracking tactics

Supporting backtracking

http://www.cs.cmu.edu/~azurite/

https://www.youtube.com/watch?v=blbIBdlUGIc