Editing Code

SWE 795, Fall 2019
Software Engineering Environments
Today

• Part 1 (Lecture)(~80 mins)
  • Editing Code

• Break!

• Part 2 (Discussion)(~60 mins)
  • Discussion of readings
What IDE features do you use when editing code?
Demo: JS in WebStorm
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
  • Systematic edits
  • 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

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.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

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>

Supporting systematic edits

• Developers sometimes make edits to multiple files that are very similar

• Tool idea: find commonality in edits between 2 or more examples, generalize to others
Example

Locating and applying systematic edits

1. ... method declaration(...) {
2. T$0$ v$0$ = v$1$.m$0$().m$1$();
3. DELETE: m$2$(v$2$.m$3$());
4. DELETE: m$2$(v$2$.m$4$());
5. while(v$0$.m$5$()) {
6. UPDATE: T$1$v$3$ = (T$1$v$0$.m$6$());
7. TO: T$2$v$4$ = v$0$.m$6$();
8. if(v$3$.m$7$()) {
9. INSERT: if(v$4$ instanceof T$1$) {
10. INSERT: T$1$v$3$ = (T$1$v$4$;
11. } ...
12. }

Fig. 2. SYDIT learns an edit from one example. A developer must locate and specify the other methods to change.

1. ... method declaration(...) {
2. Iterator v$0$ = u$0$.FieldAccessOrMethodInvocation .values().iterator();
3. while(v$0$.hasNext()) {
4. UPDATE: MVAction action = (MVAction)v$0$.next();
5. TO: Object next = v$0$.next();
6. if(action.m$0$()) {
7. } ...
8. }
9. INSERT: if(next instanceof MVAction) {
10. INSERT: MVAction action = (MVAction)next;
11. } ...
12. }

Fig. 3. LASE learns an edit from two or more examples. LASE locates other methods to change.

Fig. 4. Edit script from SYDIT abstracts all concrete names. Gray marks edit context, red marks deletions, and blue marks additions.

Fig. 5. Edit script from LASE abstracts code names that differ in the examples and uses concrete names for common ones. Gray marks edit context, red marks deletions, and blue marks additions.

Specifying program transformations

Before:
```go
if s != nil {
    for _, x := range s {
        ...
    }
}
```

Match template:
```go
if [var] != nil {
    for [:var] := range [var] {
        [body]
    }
}
```

After:
```go
for _, x := range s {
    ...
}
```

Rewrite template:
```go
for [:var] := range [var] {
    [body]
}
```

(a) Highlighted lines 2 and 4 contain redundant nil checks in Go code: iterating over a container in a for loop implies it is non-nil.

(b) Rewrite output simplifying the Go code above.

**Figure 1.** Top: A textual description for simplifying a nil check Go code, taken from the Go staticcheck tool. Bottom: Our match template and rewrite templates for the nil-check pattern above.

**Figure 2.** Redundant code pattern and simplification.
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
Domains for exploratory programming

- Learning programming through play
- Digital art and music: generative music, live coding, performance
- Data science: tasks analyzing data, building a machine learning model
- Software engineering: backtracking, commenting out or undoing different ideas; figuring out how an API should be used

Code quality tradeoffs

• Often associated with code being hard to read
  • If rapidly changing it, no sense in spending time making it clear and easy to read
  • “I know how to write code. And I know that I could write functions to reuse functions and I could try to modularize things better, and sometimes I just don’t care because why am I going to put effort in that if I’m not going to use it again?”
  • In TDD methodology, make it work (functional), make it right (easy to read), make it fast (performant) are 3 separate stages and should not progress till finished previous

Exploration process

- Backtracking: 2 or more edit run cycles that are close in time and affect the same code
- Exploration scale:
  - tuning a single variable or parameter to observe effect
  - iterating variations of a function
  - trying out different larger snippets of code
- Exploration duration: transient to long term
- Using exploratory history
  - Often use code history to understand a change or bug

Backtracking in programming

Exploration by data scientists

- Notebooks used in 3 ways: (1) preliminary scratch pad work, (2) production work, (3) shared work
- Scratch pad use: preliminary and short-lived, answers a specific question: how to debug a piece of code, test out example from internet, test if idea worth pursuing
  - "I was just testing to make sure I had the syntax right on these tuples." - IP13
  - “OK so can we do k-means on this dataset and like does it make sense” - IP11
- Sometimes occurs with individual cells, sometimes with whole notebooks

Iteration behavior

• Organizing the notebook
  • Examples:
    • Most recent code at the bottom
    • Debugging at the bottom
    • Function refs at top
  • Add cells where the original data analysis took place
• Expand then reduce
  • “So at the beginning it's usually a lot of little code cells that are one at a time... just making things work... I end up with this huge mess where there are several threads in sort of the same series. So I usually go back and start deleting things or combining cells” - IP17
• Cells enable viewing intermediate results
• Narrative structure: some used note book chronologically following steps in analysis; others were non-chronological, following important decisions

Supporting data scientists

Figure 3: The history tab opens the sidebar for Verdant containing three tabs: Activity (A), Artifacts (B & Fig. 5), and Search (C & Fig. 7). The Activity tab, shown open here, displays a list of events. A date (D) can be opened or collapsed to see what happened that day. Each row shows a version of the notebook (e.g. version #53) with a text description and visual minimap. The minimap shows cells added in green (see G) and deleted in red (F). In (E), a cell was edited and run (in blue), and the following cells were run but remained the same (in grey). The user can open any version (e.g., #53, H & Fig. 8) in a ghost notebook tab for quick reference.

Video:  https://dl.acm.org/citation.cfm?doid=3290605.3300322