Program Transformation

CS 695 / SWE 699: Programming Tools
Fall 2023
Today

• Part 1 (Lecture) (~60 mins)

• 10 min break!

• Part 2 (In-Class Activity) (50 mins)

• Part 3 (Group work time)
Logistics

- HW 2 due next week
Overview

- What is program transformation
- Applications of program transformation
  - Improving the design of code
  - Editable programming views
  - API & language migration
  - Security fixes
Program transformation

- Problem
  - You already have some existing code
  - You want to change it
  - Your change impacts several parts of the code

- --> Program Transformation
Key Ideas

• Transformations are repetitive edits
  • Need to change all call sites to add new parameter to signature
  • Need to change all call sites to respond to new API change
  • Need to change code snippets of a specific type to make them more secure
• Doing this work manually is tedious and error prone
  • Might miss something
• Programming tools can do some of this work for the developer
Some Key Design Dimensions

- What types of transformations are possible?
  - What types of changes can be described?
- How much control does a developer have over the transformation?
  - All or nothing? Change the behavior of a transformation that doesn't work right?
- What can a developer see before or after a change has been made?
  - # of place changed? Diff of each change? Errors where the change didn't work?
Challenges

- What if a transformation might sometimes insert a defect?

- What if there's more than one way to do the transformation, which a developer might want to choose between?
Improving the design of code
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>100</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>

Fixing code duplication

• Code duplication happens because there is a missing abstraction.
• Instead of one piece of code being called 10 times with different parameters to achieve different behaviors, have 10 copies of code with behavior hardcoded

• How can we make it easier to redesign code to create the abstractions that we just realized we needed?
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

Figure 2 - Screenshot of Refactoring Browser during extract code as method refactoring

(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

How developers refactor

• The RENAME refactoring tool is used much more frequently by ordinary programmers than by the developers of refactoring tools
• About 40 percent of refactorings performed using a tool occur in batches
• About 90 percent of configuration defaults in refactoring tools are not changed when programmers use the tools
• Programmers frequently floss refactor, that is, they interleave refactoring with other types of programming activity
• About half of refactorings are not high level, so refactoring detection tools that look exclusively for high-level refactorings will not detect them
• Refactorings are performed frequently
• Close to 90 percent of refactorings are performed manually, without the help of tools
Editable program views
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

• More control than a traditional refactoring tool --> transformation to be done controlled by the developer
Editable views

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

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


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.
API & Language Migration
API & Language Migration

• You wrote code in Java, now you want to use it in a Python project.

• React just introduced 3 new features. You want to update your code to use these.
Change

• What happens when an upstream system introduces a change?
  • Backwards compatible change: upstream system provides everything they did before and more
    • Nothing needs to change on downstream system
    • Just have new functionality to be used
  • Breaking change: upstream system no longer fulfills contract it did before
    • Method might be deprecated, renamed, or changed in its behavior
  • Burden of change
    • Downstream system will not work until is updated to work with new version
Program transformation for API migration

- Describe a code pattern to find all the code you want to update
- Describe a code pattern that describes what the code should be updated to
Specifying program transformations

Before:

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

Match template:

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

After:

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

Rewrite template:

```go
for [:_] := 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.

```go
func (c *SymbolCollector) addContainer(...) {
    if fields.List != nil {
        for _, field := range fields.List {
            if field.Names != nil {
                for _, fieldName := range field.Names {
                    c.addSymbol(field, fieldName.Name)
                }
            }
        }
    }
}
```

(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.
Comby.dev

- Tool for writing code transformations

Find and refactor

Comby is ideal for touching up pieces of code. Use it to translate code like this Python 2 to 3 fixer on the right to replace deprecated methods. Easily write one-off refactor or a collection of quick fixes customized to your project. Comby makes finding and changing code easier than regex alone allows and avoids pitfalls like escaping parentheses, quotes, or multiline changes.

```python
comby 'failUnlessEqual([a],[b])' 'assertEqual([a],[b])' example.py

--- example.py
+++ example.py
@@ -1,6 +1,6 @@
     def test(self):
         r = self.parse("if 1 fooze", 'r3')
-        self.failUnlessEqual(
+        self.assertEqual(  
          r.tree.toStringTree(),  
          '(if 1 fooze)'  
        )
```

Python 2 to 3 fixers reference
Fixers: reusable transformations

Python 2 to 3 fixers

import
Detects sibling imports and converts them to relative imports.

imports
Handles module renames in the standard library.

imports2
Handles other modules renames in the standard library. It is separate from the imports fixer only because of technical limitations.

input
Converts input(prompt) to eval(input(prompt)).

intern
Converts intern() to sys.intern().

isinstance
Fixes duplicate types in the second argument of isinstance(). For example, isinstance(x, (int, int)) is converted to isinstance(x, int) and isinstance(x, (int, float, int)) is converted to isinstance(x, (int, float)).

itertools_imports
Removes imports of itertools.ifilter(), itertools.izip(), and itertools.imap(). Imports of itertools.ifilterfalse() are also changed to itertools.filterfalse().

itertools
Changes usage of itertools.ifilter(), itertools.izip(), and itertools.imap() to their built-in equivalents. itertools.ifilterfalse() is changed to itertools.filterfalse().

long
Renames long to int.

map
Wraps map() in a list call. It also changes map(None, x) to list(x). Using from future.builtins import map disables this fixer.

metaclass
Converts the old metaclass syntax (__metaclass__ = Meta in the class body) to the new (class X(metaclass=Meta)).

methodattrs
Fixes old method attribute names. For example, meth.im_func is converted to meth.__func__. 
Security fixes
Security fixes

- Code that is insecure can be characterized by code patterns
- Find all of the code that matches an insecure pattern
- Describe how it should be changed to fix it
<table>
<thead>
<tr>
<th>Name</th>
<th>Problem</th>
<th>Mechanics of Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Add Audit Intercept</td>
<td>How can you make it easy to add and change auditing events?</td>
<td>Developer specifies where to intercept data to audit, and how to create audit data. Transformation adds a component that intercepts requests and responses, and creates audit events.</td>
</tr>
<tr>
<td>2. Add Perimeter Filter</td>
<td>How can you enforce input validation policies on incoming data?</td>
<td>Developer specifies where the input is checked and what are validation policies. Transformation adds a policy enforcement component and delegates requests to it.</td>
</tr>
<tr>
<td>3. chroot Jail</td>
<td>How can you prevent an attacker from corrupting important files?</td>
<td>Developer specifies the constrained environment for a program. Transformation creates a jail environment for a process and runs it inside a chroot jail.</td>
</tr>
<tr>
<td>4. Decorated Filter</td>
<td>How can you apply multiple input validation policies?</td>
<td>Developer specifies the target input and validation policies. Transformation adds a decorator to the input.</td>
</tr>
<tr>
<td>5. Exception Shielding</td>
<td>How can you preserve application behavior when rectified user inputs cause an unexpected state?</td>
<td>Developer specifies exception type and insertion point. Transformation inserts exception, and obfuscates the error message produced by the exception.</td>
</tr>
<tr>
<td>6. Safe Library Replacement</td>
<td>How can you prevent injection attacks when sanity checks fail to sufficiently validate inputs and the function that uses the inputs are also vulnerable?</td>
<td>Developer specifies the unsafe functions and safe alternatives. Transformation searches and replaces unsafe functions with safe functions.</td>
</tr>
<tr>
<td>7. Secure Logger</td>
<td>How can you ensure that system events are logged timely and in a secure manner?</td>
<td>Developer specifies the messages to log, and policies to retain confidentiality and integrity. Transformation adds a logging component that encrypts and signs logged data.</td>
</tr>
<tr>
<td>8. Unique Location for each Write Request</td>
<td>How can you prevent data corruption caused by insufficient locking mechanism when multiple processes write to the same file?</td>
<td>Developer specifies the section of a program that writes to a shared file and new file creation policy. Transformation modifies the write request so that a new file is created for each write request.</td>
</tr>
</tbody>
</table>

Developer loved, Security trusted.

Find and automatically fix vulnerabilities in your code, open source dependencies, containers, and IaC — powered by Snyk’s industry-leading security intelligence and DeepCode AI.
10 min break
Intuita CodeMods

**Code migrations at lightning speed.**
Save days of manual work by running open-source automation recipes to automate framework upgrades, such as Next.js, right from your IDE or CLI.

**Build Automations with AI**
Simply run `intuita learn` and let AI instantly build your custom code automation recipe. Automate the boring and get back to shipping more new features to end-users.

[https://www.intuita.io/](https://www.intuita.io/)
In-Class Activity

• In groups of 2, try out CodeMod Studio.
  • https://codemod.studio/
  • Identify a JS migration problem (e.g., migrate old React code to use newer features such as WebHooks)
  • Read docs to understand how to build a CodeMod for your problem.
  • Build a CodeMod and try it out with sample code snippets.