Section 1: Debugging + Code Reasoning

Alex Mariakakis

cse331-staff@cs.washington.edu (staff-wide)

Outline

• Introduction
• Reasoning about code
• IDEs – Eclipse
• Debugging

Reasoning About Code

• Two purposes
  o Prove our code is correct
  o Understand why code is correct
• Forward reasoning: determine what follows from initial conditions
• Backward reasoning: determine sufficient conditions to obtain a certain result

Forward Reasoning

// {x >= 0, y >= 0}
y = 16;
//
x = x + y
//
x = sqrt(x)
//
y = y - x
//

// {x >= 0, y >= 0}
y = 16;
//
x = x + y
//
x = sqrt(x)
//
y = y - x
//
Forward Reasoning

// {x >= 0, y >= 0}
y = 16;
// {x >= 0, y = 16}
x = x + y
// {x >= 16, y = 16}
x = sqrt(x)
// {x >= 4, y = 16}
y = y - x

Forward Reasoning

// {true}
if (x>0) {
    // {x > 0}
    abs = x
    // {x > 0, abs = x}
}
else {
    // {x <= 0}
    abs = -x
    // {x <= 0, abs = -x}
}
//

Forward Reasoning

// {true}
if (x>0) {
    // {x > 0}
    abs = x
    // {x > 0, abs = x}
}
else {
    // {x <= 0}
    abs = -x
    // {x <= 0, abs = -x}
}
// {x > 0, abs = x OR x <= 0, abs = -x}
Forward Reasoning

```java
// {true}
if (x>0) {
    // {x > 0}
    abs = x
    // {x > 0, abs = x}
} else {
    // {x <= 0}
    abs = -x
    // {x <= 0, abs = -x}
} // {x > 0, abs = x OR x <= 0, abs = -x}
// {abs = |x|}
```

Backward Reasoning

```java
// a = x + b;
// c = 2b - 4
// {a + c > 0}
x = a + c
// {x > 0}
```

Backward Reasoning

```java
// a = x + b;
// c = 2b - 4
// {a + c > 0}
x = a + c
// {x > 0}
```

Backward Reasoning

```java
// {x + 3b - 4 > 0}
a = x + b;
// {a + 2b - 4 > 0}
c = 2b - 4
// {a + c > 0}
x = a + c
// {x > 0}
```

Implication

- Hoare triples are just an extension of logical implication
  - Hoare triple: \( (P) S (Q) \)
  - \( P \rightarrow Q \) after statement \( S \)

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>P → Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>
**Implication**

- Hoare triples are just an extension of logical implication
  - Hoare triple: \( \{P\} \text{S} \{Q\} \)
  - \( P \rightarrow Q \) after statement S
- Everything implies true
- False implies everything

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>P → Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

**Weaker vs. Stronger**

- If \( P_1 \rightarrow P_2 \), then
  - \( P_1 \) is stronger than \( P_2 \)
  - \( P_2 \) is weaker than \( P_1 \)
- Weaker statements are more general, stronger statements say more
- Stronger statements are more restrictive
- Ex: \( x = 16 \) is stronger than \( x > 0 \)
- Ex: “Alex is an awesome TA” is stronger than “Alex is a TA”

**Weakest Precondition**

- The most lenient assumptions such that a postcondition will be satisfied
- If \( P^* \) is the weakest precondition for \( \{P\} \text{S} \{Q\} \), then \( P \rightarrow P^* \) for all \( P \) that make the Hoare triple valid
- \( WP = \wp(S, Q) \), which can be found using backward reasoning
  - Ex: \( \wp(x = y+4, x > 0) = y+4>0 \)

**What is Eclipse?**

- Integrated development environment (IDE)
- Allows for software development from start to finish
  - Type code with syntax highlighting, warnings, etc.
  - Run code straight through or with breakpoints (debug)
  - Break code
- Mainly used for Java
  - Supports C, C++, JavaScript, PHP, Python, Ruby, etc.
- Alternatives
  - NetBeans, Visual Studio, IntelliJIDEA

**Eclipse shortcuts**

<table>
<thead>
<tr>
<th>Shortcut</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl + D</td>
<td>Delete an entire line</td>
</tr>
<tr>
<td>Alt + Shift + R</td>
<td>Refactor (rename)</td>
</tr>
<tr>
<td>Ctrl + Shift + O</td>
<td>Clean up imports</td>
</tr>
<tr>
<td>Ctrl + /</td>
<td>Toggle comment</td>
</tr>
<tr>
<td>Ctrl + Shift + F</td>
<td>Make my code look nice ©</td>
</tr>
</tbody>
</table>

**Eclipse Debugging**

- System.out.println() works for debugging…
  - It’s quick
  - It’s dirty
  - Everyone knows how to do it
- …but there are drawbacks
  - What if I’m printing something that’s null?
  - What if I want to look at something that can’t easily be printed (e.g., what does my binary search tree look like now)?
- Eclipse’s debugger is powerful…if you know how to use it
Double click in the gray area to the left of your code to set a breakpoint. A breakpoint is a line that the Java VM will stop at during normal execution of your program, and wait for action from you.

Click the Bug icon to run in Debug mode. Otherwise your program won't stop at your breakpoints.

Controlling your program while debugging is done with these buttons:

- **Step Into**: Steps into the method at the current execution point – if possible. If not possible then just proceeds to the next execution point. If there's multiple methods at the current execution point step into the first one to be executed.

Play, pause, stop work just like you'd expect.
**Eclipse Debugging**

**Step Over**
Steps over any method calls at the current execution point. Theoretically program proceeds just to the next line.

**BUT, if you have any breakpoints set that would be hit in the method(s) you stepped over, execution will stop at those points instead.**

**Step Out**
Allows method to finish and brings you up to the point where that method was called.

**Useful if you accidentally step into Java internals (more on how to avoid this next).**

Just like with step over though you may hit a breakpoint in the remainder of the method, and then you'll stop at that point.

**Enable/disable step filters**
There's a lot of code you don't want to enter when debugging, internals of Java, internals of JUnit, etc.

You can skip these by configuring step filters.

**Checked items are skipped.**

**Stack Trace**
Shows what methods have been called to get you to current point where program is stopped.

You can click on different method names to navigate to that spot in the code without losing your current spot.

**Variables Window**
Shows all variables, including method parameters, local variables, and class variables, that are in scope at the current execution spot. Updates when you change positions in the stackframe. You can expand objects to see child member values. There's a simple value printed, but clicking on an item will fill the box below the list with a pretty format.

Some values are in the form of ObjectName (id=x), this can be used to tell if two variables are referring to the same object.

**Variables that have changed since the last break point are highlighted in yellow.**

You can change variables right from this window by double clicking the row entry in the Value tab.
Variables that have changed since the last break point are highlighted in yellow. You can change variables right from this window by double clicking the row entry in the Value tab.

There's a powerful right-click menu:

- See all references to a given variable
- See all instances of the variable's class
- Add watch statements for that variables value (more later)

Show Logical Structure

Expands out list items so it’s as if each list item were a field (and continues down for any children list items)

Breakpoints Window

Shows all existing breakpoints in the code, along with their conditions and a variety of options.

Double clicking a breakpoint will take you to its spot in the code.

Enabled/Disabled Breakpoints

Breakpoints can be temporarily disabled by clicking the checkbox next to the breakpoint. This means it won’t stop program execution until re-enabled.

This is useful if you want to hold off testing one thing, but don’t want to completely forget about that breakpoint.

Hit count

Breakpoints can be set to occur less-frequently by supplying a hit count of n.

When this is specified, only each n-th time that breakpoint is hit will code execution stop.
Conditional Breakpoints

Breakpoints can have conditions. This means the breakpoint will only be triggered when a condition you supply is true. This is very useful for when your code only breaks on some inputs!

Watch out though, it can make your code debug very slowly, especially if there’s an error in your breakpoint.

Eclipse Debugging

Disable All Breakpoints

You can disable all breakpoints temporarily. This is useful if you’ve identified a bug in the middle of a run but want to let the rest of the run finish normally.

Don’t forget to re-enable breakpoints when you want to use them again.

Eclipse Debugging

Break on Java Exception

Eclipse can break whenever a specific exception is thrown. This can be useful to trace an exception that is being “translated” by library code.

Expressions Window

Used to show the results of custom expressions you provide, and can change any time.

Not shown by default but highly recommended.

Expressions Window

These persist across projects, so clear out old ones as necessary.

Expressions Window

Used to show the results of custom expressions you provide, and can change any time.

Resolves variables, allows method calls, even arbitrary statements “2+2”

Beware method calls that mutate program state – e.g. stk1.clear() or in.nextLine() – these take effect immediately.
Demo!!!