Habanero Extreme Scale Software Research Project
Comp215: Design, Testing and Debugging revisited

Zoran Budimlić (Rice University)
So, you read the Week10 Assignment?

DONT PANIC
Text adventure game

I need a way to represent the game world!
   Start writing class structure for rooms, items, characters, inventory...

I need a way to communicate with the player!
   Start writing UI parser for English, multiple-choice actions,

I need a web frontend!
   Wander into JavaScript...

Oh, and how to show the player where he is?
   Start writing fancy text-based graphics to show the room contents

Wait, I want the cool points too!
   Start adding cool characters, items, and rooms
How to design “big” things?

Example: “Build me a text adventure game”

Q: How do you approach a “big” problem?
A: Break it down into smaller problems!

Create a skeleton game, start implementing parts of it, mock up everything else

   Always test the code you’re creating along the way

Top-down approach leads much quicker to a solution, with fewer bugs
“Measuring programming progress by lines of code is like measuring aircraft building progress by weight.”

- Bill Gates
Modularity is your friend

Build your code to interact with a “game world”, not a specific design
Make it easy to remove one back-end and add another one later
You can start with JSON as-is to represent your world, change it later if needed

Build your game interactions in a general-purpose way
Start off with something simple like “north”, “east”, e.t.c., plan to replace later

Iterate quickly, discover whether your design fits the requirements, then change course if necessary
Versus coding and coding for days, only to be stuck with the wrong design

Plan on throwing away a good chunk of your Week 1 code
It’s not wasted effort! You’re probably going to do it anyway, so you might as well plan on doing it

Use the stuff we gave you!
JSON Parser, Lists, Trees, Treaps, Simple Web Server
Correctness is your #1 priority

Correct but slow always beats buggy but (fast/pretty/elegant)

Always think carefully how your changes are affecting the code
- Regression tests are your friend
- Some systems won’t even let you commit until you pass the regression tests

When you find a bug, write more regression tests
- Perfect moment to add more tests
- Test for the exact bug that you discovered, and any related ones you might think of

Beautiful code is (usually) bug-free code

Complex code is very seldom bug-free
- Even if you didn’t introduce a bug, if it’s too complex, next change will likely introduce a (harder to find) bug
The cost of a bug in software life cycle

Source: carpgemini.com
“When debugging, novices insert corrective code; experts remove defective code.”

- Richard Pattis
So how do we write/debug beautiful code?

Gradually!

First write “ugly” (easier to test and debug) code, then make it “pretty”

You’ll be tempted to skip the last step, but don’t

When debugging, you might need to reverse the process

Outline your lambdas, create temporaries for the intermediate results

IntelliJ compiler will make it easier to get the types right

IntelliJ debugger will make it easier to see what is going on
So how do we write/debug beautiful code?

```java
json.UtilsTest:
Parser.IValue testVal = Utils.updatePathMatchesRegex(ParserTest.bigComparison,
    List.of(".*", ".*", "title"),
    oval -> oval.map(val -> javaString(val.asJString().
        toUnescapedString().
        toUpperCase()).
    .get();
```
So how do we write/debug beautiful code?

```java
IList<String> path = List.of(".*", ".*", "title");

Function<Optional<Parser.IValue>, Optional<Parser.IValue>> func =
    oval -> oval.map(val -> {
        Parser.JString tmp = val.asJString();
        String tmp2 = tmp.toUnescapedString();
        String tmp3 = tmp2.toUpperCase();
        Parser.JString jstring = javaString(tmp3);
        return jstring;
    });

Optional<Parser.IValue> testValOptional = Utils.updatePathMatchesRegex(ParserTest.bigComparison,
    path,
    func);

Parser.IValue testVal2 = testValOptional.get();

assertEquals(bigComparison2, testVal2);
```
“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.”

- Martin Fowler
Pick your algorithm carefully

An $O(n \log(n))$ algorithm may be much more complex than an $O(n^2)$ algorithm

But aren’t you supposed to always pick the faster one?

Know your input domain

If all you ever do is sorting lists that are at most 10 elements, then $O(n^2)$ might be good enough

In fact, it might be faster than $O(n \log(n))$

In real-time systems, a “slower on average” algorithm that always meets the deadline may be much better than a faster algorithm that sometimes misses it

In COMP 215, a correctly implemented slower algorithm will always get you more points than an incorrectly implemented faster algorithm

Make sure you document these kinds of decisions
“Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”

- Brian W. Kernighan
Know your tools

What the IntelliJ can do for you

- Type checking
- Type inference
- Refactoring
- Suggest changes to your code

  Make sure you know exactly what the changes will do

- Generate test functions for you

What the runtime can do

- Check for annotations (function contracts)
- Check type casts

Use Mockito to mock parts of your code that you still haven’t done
Mock up your unfinished code

Parts of your code will unavoidably depend on other parts

You *do not* want to wait until you finish everything to start testing/debugging!

Use Mockito to mock up the classes that your code uses

Create “mock” versions of classes that you want to use
They use the “real” interface but have none of your code inside
You tell them how to respond
You then do your tests of the code using those classes

All kinds of uses
Example: make a fake thief character that always appears in the room you are in
Printing vs. logging

Old school
System.out.println("running constructor");

New school
import edu.rice.Log;

class Foo {
    private static final String TAG = "Foo";
    public Foo() {
        Log.i(TAG, "running constructor");
    }
}
Log levels, etc.

Information, warning, error
Log.i(), Log.w(), Log.e()
You can adjust the level via Log.setLogLevel() and see more or less output
Optional lambda “lazy” form if you want “fast” when logging is disabled
Log.i(TAG, () -> "current input: " + input.toString());

private static final String
If you copy-and-paste the code from one class to another, the TAG will be displayed correctly
Really new school: Function wrapping

Log.java wrappers:

```java
public static <A,B> Function<A,B> iWrap(
    String tag,
    Function<A, B> func) {

    return input -> {
        B output = func.apply(input);
        Log.i(tag, ()->{String.format("%s -> %s",
            input.toString(), output.toString())));
        return output;
    };
}
```
Composing RPN stack operations

// \( f(x) = (x + 10) \times 27 \)

RPNCalculator.CalcOp f2 =
    RPNCalculator.numberPusher(10)
    .andThen(RPNCalculator::add)
    .andThen(RPNCalculator.numberPusher(27))
    .andThen(RPNCalculator::multiply);
Composing RPN stack operations

// f(x) = (x + 10) * 27

RPNCalculator.CalcOp f2 =
    RPNCalculator.numberPusher(10)
    .andThen(RPNCalculator::add)
    .andThen(RPNCalculator.numberPusher(27))
    .andThen(RPNCalculator::multiply);

assertEquals(
    Optional.of(List.of((3.0 + 10.0) * 27.0)),
    f2.apply(RPNCalculator.of(3.0)));

Nothing happens until this call
Wrapping the functions

// f(x) = (x + 10) * 27

RPNCalculator::CalcOp f3 =
    Log.iWrap(TAG, RPNCalculator::numberPusher(10))
    .andThen(Log.iWrap(TAG, RPNCalculator::add))
    .andThen(Log.iWrap(TAG, RPNCalculator::numberPusher(27)))
    .andThen(Log.iWrap(TAG, RPNCalculator::multiply));

Oct 23, 2015 3:17:05 PM RPNCalculatorTest
INFO: Optional[3.0] -> Optional[10.0 3.0]
Oct 23, 2015 3:17:05 PM RPNCalculatorTest
INFO: Optional[10.0 3.0] -> Optional[13.0]
Oct 23, 2015 3:17:05 PM RPNCalculatorTest
INFO: Optional[13.0] -> Optional[27.0 13.0]
Oct 23, 2015 3:17:05 PM RPNCalculatorTest
INFO: Optional[27.0 13.0] -> Optional[351.0]
“If debugging is the process of removing software bugs, then programming must be the process of putting them in.”

- Edsger Dijkstra
Live Coding

Decompose testUpdatePathMatchesRegex test