Today

- Assertions
- Scope minimization
- Systematic debugging

The topic of today’s lecture is debugging. First how to avoid it, second how to keep it easy, and third how to do it systematically when you have to.

First Defense: Make Bugs Impossible

We’ve already talked about static typing. Static typing eliminates many runtime type errors.

We also saw some examples of dynamic checking in the early lectures. Java makes array overflow bugs impossible by catching them dynamically – if you try to use an index outside the bounds of an array or a List, then Java automatically produces an error. Languages like C and C++ don’t do this.

Immutability (immunity from change) is a major design principle in this course. Immutable types are types whose values can never change once they have been created. Immutable objects like Strings can be passed around and shared without fear that they will be modified. Why is a String a better data type than a char array? We’ll talk a lot more about immutability in a future lecture.

Java also gives us immutable references: variables that are assigned once and never reassigned. To make a reference immutable, declare it with the keyword `final`:

```java
final int n = 5;
```

If the Java compiler isn’t convinced that your final variable will only be assigned once at runtime, then it will produce a compiler error. So `final` gives you static checking for immutable references.

It’s good practice to use final for declaring the parameters of a method and as many local variables as possible. Like the type of the variable, these declarations are important documentation, useful to the reader of the code and statically checked by the compiler.

Be careful about what final means! It only makes the reference immutable, not necessarily the object that the reference points to. Consider this example:

```java
final char[] vowels = new char[] { 'a', 'e', 'i', 'o', 'u' };
```

The `vowels` variable below is declared final, but is it really unchanging? Which of the following statements will be illegal (caught by statically by the compiler), and which will be allowed?

```java
vowels[0] = 'z';
```

```java
vowels = new char[] { 'x', 'y', 'z' };
```

Second Defense: Localize Bugs

If we can’t prevent bugs, we can try to localize them to a small part of the program, so that we don’t have to look too hard to find the cause of the bug. When localized to a single method or small module, bugs may be found simply by studying the program text.

We already talked about fail fast: the earlier a problem is observed (the closer to its cause), the easier
it is to fix.

Let's begin with a simple example:

```java
public double sqrt (double x) // requires: x >= 0 // returns: approximation to square root of x { ... }
```

Now suppose somebody calls `sqrt` with a negative argument. What's the best behavior for `sqrt`? Since the caller has failed to satisfy the precondition, `sqrt` is no longer bound by the terms of its contract, so it is technically free to do whatever it wants: return an arbitrary value, or enter an infinite loop, or melt down the CPU. Since the failed precondition indicates a bug in the caller, however, the most useful behavior would point out the bug as early as possible. We do this by inserting a runtime assertion that tests the precondition. Here is one way we might write the assertion:

```java
public double sqrt (double x) // requires: x >= 0 // returns: approximation to square root of x {
    if (! (x >= 0)) throw new AssertionError ();
    ...
}
```

When the precondition is not satisfied, this code terminates the program by throwing an `AssertionError` exception. The effects of the caller's bug are prevented from propagating.

Checking preconditions is an example of defensive programming. Real programs are rarely bug-free. Defensive programming offers a way to mitigate the effects of bugs even if you don't know where they are.

**Assertions**

It is common practice to define a procedure for these kinds of defensive checks, usually called `assert`:

```java
assert (x >= 0);
```

This approach abstracts away from what exactly happens when the assertion fails. The failed assert might exit; it might record an event in a log file; it might email a report to a maintainer.

Assertions have the added benefit of documenting an assumption about the state of the program at that point. To somebody reading your code, `assert(x>=0)` says "at this point, it should always be true that x >= 0." Unlike a comment, however, an assertion is executable code that enforces the assumption at runtime.

In Java, runtime assertions are a built-in feature of the language. The simplest form of the assert statement takes a boolean expression, exactly as shown above, and throws `AssertionError` if the boolean expression evaluates to false:

```java
assert x >= 0;
```

An assert statement may also include a description expression, which is usually a string, but may also be a primitive type or a reference to an object. The description is printed in an error message when the assertion fails, so it can be used to provide additional details to the programmer about the cause of the failure. The description follows the asserted expression, separated by a colon. For example:

```java
assert (x >= 0) : "x is " + x;
```

If x = -1, then this assertion fails with the error message

```java
x is -1
```

along with a stack trace that tells you where the assert statement was found in your code and the sequence of calls that arrived there. This information is often enough to get started in finding the bug.
A serious problem with Java assertions is that assertions are off by default. If you just run your program as usual, none of your assertions will be checked! You have to enable assertions explicitly by passing `-ea` (which stands for "enable assertions") to the Java virtual machine:

```
java -ea MyClass
```

(In Eclipse, you specify `-ea` in the Run dialog under VM arguments.)

If you don’t like Java’s assert mechanism, it’s easy enough to roll your own, by writing a procedure that tests a boolean expression and throws an exception if the expression is false. Since assert is now a reserved word in Java, however, you must name your procedure something else:

```java
public static void assertAlways (boolean b) { if (!b) throw new RuntimeException ("assertion failure");}
```

In fact, the assertTrue method in JUnit is basically implemented this way. Since it’s a static method, you don’t have to be writing a unit test to use assertTrue. Just call Assert.assertNotNull().

Here’s a handy rule for when to use an assertion that’s always enabled (like assertAlways or assertTrue), and when to use an assertion that is disabled in production code:

**Use Java's built-in assert only for expensive assertions.** Here, expensive depends on the context. Checking whether an array is sorted would be expensive in the context of a log-time binary search method; but it would be quite cheap in the context of a method that writes the array to disk.

**Otherwise use assertAlways or assertTrue.** Constant-time checks should always fall in this category, unless the constant factor is large or the code is performance-critical.

### What to Assert

Here are some things you should assert:

- **Preconditions** to a method, like we just saw for sqrt’s argument.
- **Postconditions.** Assertions are useful for checking the result of a method before returning it. This kind of assertion is sometimes called a *self check*. For example, the sqrt method might square its result to check whether it is reasonably close to x:

  ```java
  public double sqrt (double x)// requires: x >= 0// returns: approximation to square root of x {
  assert (x >= 0);
  double r;...
  // compute result r assert (Math.abs(r*r - x) < .0001 ); return r;
  }
  ```

- **Rep invariants.** The rep invariant of an abstract data type should be asserted whenever an object changes, in order to prove that the invariant is still established. We’ll talk about rep invariants in the abstract data types lectures.

- **Loop invariants.** When you’re implementing a nontrivial algorithm with a loop or a recursion, the algorithm often has a key property that is invariant, true for every iteration of the loop or recursive call. For example,...

- **Covering all cases.** If a conditional statement or switch does not cover all the possible cases, it is good practice to use an assertion to block the illegal cases:

  ```java
  switch (vowel) {
  case 'a': return "A";
  case 'e':
  case 'i':
  case 'o':
  case 'u':
  ```
default: Assert.assertTrue(false);
}

The assertion in the default clause has the effect of asserting that vowel must be one of the five vowel letters.

When should you write runtime assertions? As you write the code, not after the fact. When you’re writing the code, you have the invariants in mind. If you postpone writing assertions, you’re less likely to do it, and you’re liable to omit some important invariants.

**What Not to Assert**

Runtime assertions are not free. They can clutter the code, so they must be used judiciously. Avoid trivial assertions, just as you would avoid uninformative comments. For example:

```java
// don’t do this
x = y + 1;
assert (x == y+1);
```

This assertion doesn’t find bugs in your code. It finds bugs in the compiler or Java virtual machine, which are components that you should trust until you have good reason to doubt them. If an assertion is obvious from its local context, leave it out.

Never use assertions to test conditions that are external to your program, such as the existence of files, the availability of the network, or the correctness of input given by the user. Assertions test the internal state of your program to ensure that it is within the bounds of its specification. When an assertion fails, it indicates that the program has run off the rails in some sense, into a state in which that it was not designed to function properly. Assertion failures therefore indicate bugs. External failures are not bugs, and there is no change you can make to your program in advance that will prevent them from happening. External failures should be handled using exceptions, which are discussed in more detail later in this lecture.

Many assertion mechanisms are designed so that assertions are executed only during testing and debugging, and turned off when the program is released to users. Java’s assert statement behaves this way. The advantage of this approach is that you can write very expensive assertions that would otherwise seriously degrade the performance of your program. For example, a procedure that searches an array using binary search has a precondition that the array be sorted. Asserting this precondition requires scanning through the entire array, however, turning an operation that should run in logarithmic time into one that takes linear time. You should be willing (eager!) to pay this cost during testing, since it makes debugging much easier, but not when the program is released to users.

However, disabling assertions in release has a serious disadvantage. With assertions disabled, a program has far less error checking when it needs it most. Novice programmers are usually much more concerned about the performance impact of assertions than they should be. Most assertions are cheap, so they should not be disabled in the official release.

Since assertions may be disabled, the correctness of your program should never depend on whether or not the assertion expressions are executed. In particular, asserted expressions should not have side-effects. For example, if you want to assert that an element removed from a list was actually found in the list, don’t write it like this:

```java
// don’t do this
assert (list.remove (x));
```

If assertions are disabled, the entire expression is skipped, and x is never removed from the list. Write it like this instead:

```java
boolean found = list.remove (x); assert (found);
```
**Incremental Development**

A great way to localize bugs to a tiny part of the program is incremental development – build only a bit of your program at a time, and test that bit thoroughly. That way, when you discover a bug, it’s more likely to be in the part that you just wrote, rather than anywhere in a huge pile of code.

The testing lecture talked about two techniques that help with this:

- **Unit testing**: when you test a module in isolation, you can be confident that any bug you find is in that unit -- or maybe in the test cases themselves.
- **Regression testing**: when you’re adding a new feature to a big system, run the regression test suite as often as possible. If a test fails, the bug is probably in the code you just changed.

**Modularity & Encapsulation**

You can also localize bugs by better design.

**Modularity.** Modularity means dividing up a system into components, or modules, each of which can be designed, implemented, tested, reasoned about, and reused separately from the rest of the system. The opposite of a modular system is a monolithic system – big and with all of its pieces tangled up and dependent on each other.

A program consisting of a single, very long main() function is monolithic – harder to understand, and harder to isolate bugs in. By contrast, a program broken up in small functions and classes is more modular.

**Encapsulation.** Encapsulation means building walls around a module (a hard shell or capsule) so that the module is responsible for its own internal behavior, and bugs in other parts of the system can’t damage its integrity.

We’ll talk more about encapsulation using public and private modifiers when we get to abstract data types, but one important kind of encapsulation that is relevant to the kinds of methods we’re writing now is variable scope. The scope of a variable is the portion of the program text over which that variable is defined, in the sense that expressions and statements can refer to the variable. Keeping variable scopes as small as possible makes it much easier to reason about where a bug might be in the program. For example, suppose you have a loop like this:

```java
for (i = 0; i < 100; ++i) {
    ...
    doSomeThings();
    ...
}
```

...and you’ve discovered that this loop keeps running forever – i never reaches 100. Somewhere, somebody is changing i. But where? If i is declared as a global variable like this:

```java
public static int i;
...
for (i =0; i < 100; ++i) {
    ...
    doSomeThings();
    ...
}
```

...then its scope is the entire program. It might be changed anywhere in your program: by doSomeThings(), by some other method that doSomeThings() calls, by a concurrent thread running
some completely different code. But if \( i \) is instead declared as a local variable with a narrow scope, like this:

```java
for (int i = 0; i < 100; ++i) {
    ...
    doSomeThings();
    ...
}
```

... then the only place that \( i \) can be changed is within the for statement – in fact, only in the ... parts that I’ve omitted. You don’t even have to consider doSomeThings(), because doSomeThings() doesn’t have access to this local variable.

**Minimizing the scope of variables** is a powerful practice for bug localization. Here are a few rules that are good for Java:

- **Always declare a loop variable in the for-loop initializer.** So rather than declaring it before the loop:
  ```java
  int i;
  for (i = 0; i < 100; ++i) {
      ...
      doSomeThings();
      ...
  }
  ```
  which makes the scope of the variable the entire rest of outer curly-brace block containing this code, you should do this:
  ```java
  for (int i = 0; i < 100; ++i) {
      ...
  }
  ```
  which makes the scope of \( i \) limited just to the for loop.

- **Declare a variable only when you first need it, and in the innermost curly-brace block that you can.** Variable scopes in Java are curly-brace blocks, so put your variable declaration in the innermost one that contains all the expressions that need to use the variable. *Don’t* declare all your variables at the start of the function – it makes their scopes unnecessarily large. (But note that in languages without static type declarations, like Python and Javascript, the scope of a variable is normally the entire function anyway, so you can’t restrict the scope of a variable with curly braces, alas.)

- **Avoid global variables.** Very bad idea, especially as programs get large.

**Last Ditch Defense: Debug Systematically**

Sometimes you have no choice but to debug, however — particularly when the bug is found only when you plug the whole system together, or reported by a user after the system is deployed, in which case it may be hard to localize it to a particular module. For those situations, we can suggest a systematic strategy for more effective debugging.

**Reproduce the Bug**

Start by finding a small, repeatable test case that produces the failure. If the bug was found by regression testing, then you’re in luck; you already have a failing test case in your test suite. If the bug was reported by a user, it may take some effort to reproduce the bug. For graphical user interfaces and multithreaded programs, a bug may be hard to reproduce consistently if it depends on timing of events or thread execution.

Nevertheless, any effort you put into making the test case small and repeatable will pay off, because you’ll have to run it over and over while you search for the bug and develop a fix for it. Furthermore, after you’ve successfully fixed the bug, you’ll want to add the test case to your regression test suite, so that the bug never crops up again. Once you have a test case for the bug, making this test work becomes your goal.
Understand the Location and Cause of the Bug

To localize the bug and its cause, you can use the scientific method:

(a) Study the data. Look at the test input that causes the bug, and the incorrect results, failed assertions, and stack traces that result from it.

(b) Hypothesize. Propose a hypothesis, consistent with all the data, about where the bug might be, or where it cannot be. It's good to make this hypothesis general at first. Here's an example. You're developing a web browser, and a user has found that displaying a certain web page causes the browser to crash. You might hypothesize that the bug is not in the networking code that fetches the page from the server, but in one of the modules that parses the web page or displays it.

(c) Experiment. Devise an experiment that tests your hypothesis. The experiment might be a different test case. In our web browser example, you might test your hypothesis by downloading the page to disk and loading it from a disk file instead of over the network. Another experiment inserts probes in the running program — print statements, assertions, or debugger breakpoints. It's tempting to try to insert fixes to the hypothesized bug, instead of mere probes. This is almost always the wrong thing to do, because your fixes may just mask the true bug. For example, if you're getting a NullPointerException, try to understand what's going on first; don't just insert a little test that avoids the exception without fixing the real problem.

(d) Repeat. Add the data you collected from your experiment to what you knew before, and make a fresh hypothesis.

Bug localization by binary search. Debugging is a search process, and you can sometimes use binary search to speed up the process. For example, in a web browser, the web page might flow through 10 modules before the program crashes. To do a binary search, you would divide this workflow in half, guessing that the bug is found somewhere in the first 5 modules, and insert probes (like breakpoints or print statements) after the fifth module to check its results. From the results of that experiment, you would further divide in half.

Prioritize your hypotheses. When making your hypothesis, you may want to keep in mind that different parts of the system have different likelihoods of failure. For example, old, well-tested code is probably more trustworthy than code recently added. Java library code is probably more trustworthy than yours (except where it depends on your code for correct behavior, as HashMap depends on your classes properly implementing the object contract). The Java compiler and runtime, operating system platform, and hardware are increasingly more trustworthy, because they are more tried and tested. You should trust these lower levels until you've found good reason not to.

Make sure your source code and object code is up to date. Pull the latest version from the repository, and delete all your *.class files and recompile everything (in Eclipse, this is done by Project / Clean).

Swap components. If you have another implementation of a module that satisfies the same interface, and you suspect the module, you may try swapping in the alternative. For example, if you suspected java.util.ArrayList, you could swap in java.util.LinkedList instead. If you suspect the binarySearch() method, then substitute a simpler linearSearch() instead. If you suspect the Java runtime, run with a different version of Java. If you suspect the operating system, run your program on a different OS. If you suspect the hardware, run on a different machine. You can waste a lot of time swapping unfailing components, however, so don't do this unless you have good reason to suspect a component.

Get help. It often helps to explain your problem to someone else, even if the person you're talking to has no idea what you're talking about. Lab assistants and fellow 6.170 students usually do know what you're talking about, so they're even better.
Sleep on it. If you’re too tired, you won’t be an effective debugger. Trade latency for efficiency.

Fix the Bug

Once you’ve found the bug and understand its cause, the third step is to devise a fix for it. Avoid the temptation to slap a patch on it and move on. Ask yourself whether the bug was a coding error, like a misspelled variable or interchanged method parameters, or a design error, like an underspecified or insufficient interface. Design errors may suggest that you step back and revisit your design, or at the very least consider all the other clients of the failing interface to see if they suffer from the bug too.

Think also whether the bug has any relatives. If I just found a divide-by-zero error here, did I do that anywhere else in the code? Try to make the code safe from future bugs like this.

Also consider what effects your fix will have. Will it break any other code?

Finally, after you have applied your fix, add the bug’s test case to your regression test suite, and run all the tests to assure yourself that (a) the bug is fixed, and (b) no new bugs have been introduced.
Debugging Example

Let’s practice our skills on this program. Can you debug it?

/**
 * Can a thousand monkeys typing for a thousand years write a little bit of
 * Shakespeare’s plays? Let’s find out.
 *
 * Note: this class is NOT meant as an example of good coding style.
 * It also has a few bugs lurking in it, because we’re using it as an example
 * of debugging.
 */
public class MonkeyShakespeare {

    public static void main(String[] args) throws Exception {

        // Read in all of Shakespeare from a file
        List<String> allLines = new ArrayList<String>();
        BufferedReader reader = new BufferedReader(new FileReader("shakespeare.json"));
        String line = reader.readLine();
        while (line != null) {
            // Extract the play text from lines of the form:
            //    "text_entry": "A bird of my tongue is better than a beast of yours."
            if (line.contains("text_entry")) {
                line = line.substring(19, line.length() - 2); // keep only the part between
                allLines.add(line);
            }
        }
        reader.close();

        // Build an index that maps each word to the set of lines that contain the word
        Map<String, Set<String>> index = new HashMap<String, Set<String>>();
        for (String line2 : allLines) {
            StringTokenizer tokenizer = new StringTokenizer(line2, // string to split up into words ("tokens")
                                                    ",.!?;:\-\=\+\<\>\-$/!###&@*", // space & punctuation
                                                    false // don't keep the spaces and punctuation
                                                    );
            while (tokenizer.hasMoreElements()) {
                String word = tokenizer.nextToken();
                word = word.toLowerCase();
                Set<String> linesContainingWord = index.get(word);
                if (linesContainingWord == null) {
                    // First time we've seen this word -- create a set for it
                    linesContainingWord = new HashSet<String>();
                    index.put(word, linesContainingWord);
                } else {
                    linesContainingWord.add(line);
                }
            }
        }
    }
}
// count the frequency of each letter in the words actually used, so that the
// monkeys have a fair shot at typing ETAOIN more often than VKXJQZ.
// letterDistribution.get(c) counts how often a character c appears in the index
// Maps<Character, Integer> letterDistribution = new HashMap<Character, Integer>();
int sumOfLetterDistribution = 0; // sum of all counts in letterDistribution
for (String word : index.keySet()) {
  for (int i = 0; i < word.length(); ++i) {
    letterDistribution.put(word.charAt(i),
      letterDistribution.get(word.charAt(i)) + 1);
    ++sumOfLetterDistribution;
  }
}

// Count the frequency of each word length, too, so that the
// monkeys are typing THE more often than HONORIFICABILITUDINITATIBUS.
// lengthDistribution.get(len) counts the number of words of length len
// Maps<Integer, Integer> lengthDistribution = new HashMap<Integer, Integer>();
int sumOfLengthDistribution = 0; // sum of all counts in letterDistribution
for (String word : index.keySet()) {
  lengthDistribution.put(word.length(),
    lengthDistribution.get(word.length()) + 1);
  ++sumOfLengthDistribution;
}

// Set those monkeys going!
// Type one word at a time, and if we ever generate a word found
// somewhere in Shakespeare, print it.
Random random = new Random(); // our random monkey
while (true) {
  // Monkey first decides how long the word will be,
  // randomly distributed according to lengthDistribution
  int monkeyWordLength = 1;
  int rand = random.nextInt(sumOfLengthDistribution);
  while (rand >= lengthDistribution.get(monkeyWordLength)) {
    rand -= lengthDistribution.get(monkeyWordLength);
    ++monkeyWordLength;
  }

  // Now monkey picks each letter in the word,
  // randomly distributed according to letterDistribution
  String monkeyWord = "";
  for (int i = 0; i < monkeyWordLength; ++i) {
    char c = 'a';
    int rand2 = random.nextInt(sumOfLetterDistribution);
    while (rand2 >= letterDistribution.get(c)) {
      rand2 -= letterDistribution.get(c);
      ++c;
    }
    monkeyWord += c;
  }

  // Is the monkey's word in Shakespeare?
  Set<String> matchingLines = index.get(monkeyWord);
  if (matchingLines != null) {
    String randomMatchingLine = new ArrayList<String>(matchingLines)
      .get(random.nextInt(matchingLines.size()));
    System.out.println(monkeyWord + " : " + randomMatchingLine);
  }
}
Some (but not all) of the problems in the code are identified below. Can you explain why these are problems?

What’s the risk in the code below?

```java
// extract the play text from lines of the form:
//    "text_entry": "A bird of my tongue is better than a beast of yours.",
if (line.contains("text_entry")) {
    line = line.substring(19, line.length()-2); // keep only the part between double-quotes
    ...
}
```

Why does Java allow you to reuse the variable name `word`, but not the variable name `line`?

```java
String line = reader.readLine();
...
for (String line : allLines) { // <= causes compile error
...
    String word = tokenizer.nextToken();
...
    for (String word : index.keySet()) { // <= doesn’t cause compile error
...
```

What’s the risk in the code below?

```java
StringTokenizer tokenizer = new StringTokenizer(
    line2, // string to split up into words ("tokens")
    " ,.!?;:[]\"\`~@$%^&*", // space & punctuation
    false // don’t keep the spaces and punctuation
);
```

```java
Set<String> linesContainingWord = index.get(word);
if (linesContainingWord == null) {
    // first time we’ve seen this word -- create a set for it
    linesContainingWord = new HashSet<String>();
    index.put(word, linesContainingWord);
} else {
    linesContainingWord.add(line);
}
```
What's the bug in the code below?

```java
for (char c = 'a'; c < 'z'; ++c) {
    letterDistribution.put(c, 0);
}
```

What's smelly about the code below?

```java
assert 'a' <= word.charAt(i) && word.charAt(i) <= 'z';
letterDistribution.put(word.charAt(i),
letterDistribution.get(word.charAt(i)) + 1);
```

What's risky about this line?

```java
for (int len = 1; len <= "HONORIFICABILITUDINITATIBUS".length(); ++len)
lengthDistribution.put(len, 0);
```

In the code below, the first block has been copied-and-pasted and edited to form the second block. Risky! The programmer made two mistakes in editing. Which one is caught automatically by Java’s static typing, and which one is not? Will that one be caught automatically by dynamic checking?

```java
// count the frequency of each letter in the words actually used, so that the
// monkeys have a fair shot at typing ETAOIN more often than VKXJQZ.
// letterDistribution.get(c) == the number of times c appears in the words
Map<Character, Integer> letterDistribution = new HashMap<Character, Integer>();
int sumOfLetterDistribution = 0;
for (String word : index.keySet()) {
    for (int i = 0; i < word.length(); ++i) {
        if (!letterDistribution.containsKey(word.charAt(i)))  {
            letterDistribution.put(word.charAt(i), 0);
        }
        letterDistribution.put(word.charAt(i),
letterDistribution.get(word.charAt(i)) + 1);
    ++sumOfLetterDistribution;
}
}
// count the frequency of each word length, too, so that the
// monkeys are typing THE more often than HONORIFICABILITUDINITATIBUS.
// lengthDistribution.get(len) is the number of words of length len in the index
Map<Integer, Integer> lengthDistribution = new HashMap<Integer, Integer>();
int sumOfLengthDistribution = 0;
for (String word : index.keySet()) {
    if (!lengthDistribution.containsKey(word.length())) {
        lengthDistribution.put(word.length(), 0);
    }
    lengthDistribution.put(word.length(),
        lengthDistribution.get(word.length()) + 1);
    ++sumOfLetterDistribution;
}
```
Summary

avoid debugging
  - make bugs impossible with techniques like static typing

keep bugs confined
  - failing fast with assertions keeps a bug’s effects from spreading
  - incremental development and unit testing confine bugs to your recent code
  - scope minimization reduces the amount of the program you have to search

debug systematically
  - reproduce bug as a test case, and put it in your regression suite
  - find the bug using the scientific method
  - fix the bug thoughtfully, not slapdash

Thinking about our three main measures of code quality:

• Safe from bugs. We’re trying to prevent them and get rid of them.
• Easy to understand. Techniques like static typing, final declarations, and assertions are additional documentation of the assumptions in your code. Variable scope minimization makes it easier for a reader to understand how the variable is used, because there’s less code to look at.
• Ready for change. Assertions and static typing document the assumptions in an automatically-checkable way, so that when a future programmer changes the code, accidental violations of those assumptions are detected.