Reading 8: Avoiding Debugging

6.005 Prime Objective

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

The topic of today's class is debugging – or rather, how to avoid debugging entirely, or keep it easy when you have to do it.

First Defense: Make Bugs Impossible

The best defense against bugs is to make them impossible by design.

One way that we've already talked about is **static checking (../01-static-checking/#static_checking_dynamic_checking_no_checking)** . Static checking eliminates many bugs by catching them at compile time.

We also saw some examples of **dynamic checking** in earlier class meetings. For example, 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. Older languages like C and C++ silently allow the bad access, which leads to bugs and security vulnerabilities (//en.wikipedia.org/wiki/Buffer_overflow) .

Immutability (immunity from change) is another design principle that prevents bugs. An *immutable type* is a type whose values can never change once they have been created.

String is an immutable type. There are no methods that you can call on a String that will change the sequence of characters that it represents. Strings can be passed around and shared without fear that they will be modified by other code.

Java also gives us *immutable references* : variables declared with the keyword `final` , which can be assigned once but never reassigned. 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.

Consider this example:

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

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

```
vowels = new char[] { 'x', 'y', 'z' };
vowels[0] = 'z';
```

You'll find the answers in the exercise below. Be careful about what `final` means! It only makes the *reference* immutable, not necessarily the *object* that the reference points to.

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 a 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:

```
/**
 * @param x  requires x >= 0
 * @return approximation to square root of x
 */
public double sqrt(double 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 requirement that `x` should be nonnegative, `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 bad call 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:

```
/**
 * @param x  requires x >= 0
 * @return approximation to square root of x
 */
public double sqrt(double 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` :

```
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:

```
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:

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

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

```
  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 brought the program to that point. 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! Java's designers did this because checking assertions can sometimes be costly to performance. For most applications, however, assertions are *not* expensive compared to the rest of the code, and the benefit they provide in bug-checking is worth that small cost in peformance.

So you have to enable assertions explicity by passing `-ea` (which stands for *enable assertions*) to the Java virtual machine. In Eclipse, you enable assertions by going to Run → Run Configurations → Arguments, and putting -ea in the VM arguments box. It's best, in fact, to enable them by default by going to Preferences → Java → Installed JREs → Edit → Default VM Arguments, as you hopefully did in the Getting Started (../../getting-started/#config-eclipse) instructions.

It's always a good idea to have assertions turned on when you're running JUnit tests. You can ensure that assertions are enabled using the following test case:

```
@Test(expected=AssertionError.class)
public void testAssertionsEnabled() {
    assert false;
}
```

If assertions are turned on as desired, then `assert false` throws an `AssertionError`. The annotation `(expected=AssertionError.class)` on the test expects and requires this error to be thrown, so the test passes. If assertions are turned off, however, then the body of the test will do nothing, failing to throw the expected exception, and JUnit will mark the test as failing.

Note that the Java `assert` statement is a different mechanism from the JUnit methods `assertTrue()`, `assertEquals()`, etc. They all assert a predicate about your code, but are designed for use in different contexts. The `assert` statement should be used in implementation code, for defensive checks inside the implementation. JUnit `assert...()` methods should be used in JUnit tests, to check the result of a test. The `assert` statements don't run without `-ea`, but the JUnit `assert...()` methods always run.

What to Assert

Here are some things you should assert:

Method argument requirements, like we saw for `sqrt`.

Method return value requirements. 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:

```
public double sqrt(double x) {
    assert x >= 0;
    double r;
    ... // compute result r
    assert Math.abs(r*r - x) < .0001;
    return r;
}
```

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:

```
switch (vowel) {
  case 'a':
  case 'e':
  case 'i':
  case 'o':
  case 'u': return "A";
  default: assert 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:

```
// 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 typed by a human 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 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 instead.

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 requirement that the array be sorted. Asserting this requirement 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 after 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:

```
// 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:

```
boolean found = list.remove(x);
assert found;
```

For 6.005, you are required to have assertions turned on, all the time. Make sure you did this in Eclipse, following the instructions in the **Getting Started handout (../../getting-started/#config-eclipse)** . If you don't have assertions turned on, you will be sad, and the staff won't have much sympathy.

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 before you move on. 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.

Our class on testing talked about two techniques that help with this:

- **Unit testing** (../03-testing/#unit_testing_and_stubs) : 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** (../03-testing/#automated_testing_and_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 software 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 into 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.

One kind of encapsulation is access control (//docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html) , using `public` and `private` to control the visibility and accessibility of your variables and methods. A public variable or method can be accessed by any code (assuming the class containing that variable or method is also public). A private variable or method can only be accessed by code in the same class. Keeping things private as much as possible, especially for variables, provides encapsulation, since it limits the code that could inadvertently cause bugs.

Another kind of encapsulation comes from **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. A method parameter's scope is the body of the method. A local variable's scope extends from its declaration to the next closing curly brace. 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:

```
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:

```
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:

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

…then the only place where `i` can be changed is within the for statement – in fact, only in the … parts that we'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:

  ```
  int i;
  for (i = 0; i < 100; ++i) {
  ```

  which makes the scope of the variable the entire rest of the outer curly-brace block containing this code, you should do this:

  ```
  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. Global variables are often used as a shortcut to provide a parameter to several parts of your program. It's better to just pass the parameter into the code that needs it, rather than putting it in global space where it can inadvertently reassigned.

Summary

In this reading, we looked at some ways to minimize the cost of debugging:

- Avoid debugging
    - make bugs impossible with techniques like static typing, automatic dynamic checking, and immutable types and references
- 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

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.