Lecture 10: Dynamic Analysis, Part 1

The best way to ensure the quality of the software you build is to design it carefully from the start. The parts will fit together more cleanly, and the functionality of each part will be simpler, so you'll make fewer errors implementing it. But it's hard not to introduce some errors during coding, and an effective way to find these is to use *dynamic* techniques: that is, those that involve executing the program and observing its behaviour. In contrast, *static* techniques are ones that you use to ensure quality before you execute: by evaluating the design and by analyzing the code (either by manual review, or by using tools such as a type checker).

Some people mistakenly rely on dynamic techniques, rushing through specification and design on the assumption that they can fix things up later. There are two problems with this approach. The first is that problems in the design get enmeshed, by implementation time, with implementation problems, so they are harder to find. The second is that the cost of fixing an error in a software artifact is known to increase dramatically the later in development it is discovered. In some early studies at IBM and TRW, Barry Boehm discovered that a specification error can cost 1000 times more to fix if not discovered until implementation!

Other people mistakenly imagine that only static techniques are necessary. Although great strides have been made in technology for static analysis, we are still far from being able to catch all errors statically. Even if you have constructed a mathematical proof that your program is correct, you would be foolish not to test it.

The fundamental problem with testing is expressed in a famous aphorism of Dijkstra's:

> *Testing can reveal the presence of errors but never their absence.*

Testing, by its very nature, is incomplete. You should be very wary of making any assumptions about the reliability of a program just because it has passed a large battery of tests. In fact, the problem of determining when a piece of software is sufficiently reliable to release is one that plagues managers, and for which very little guidance exists. It is therefore best to think of testing not as a way to establish confidence that the program is right, but rather as a way to find errors. There's a subtle but vitally important difference between these viewpoints.

10.1 Defensive Programming

Defensive programming is an approach to increasing the reliability of a program by inserting redundant checks. Here's how it works. When you're writing some code, you figure out conditions that you expect to hold at certain points in the code – invariants, in other words. Then, rather than just assuming that these invariants hold, you test them explicitly. These tests are called *runtime assertions*. If an assertion fails – that is, the invariant evaluates to false – you report the error and abort the computation.

10.1.1 Guidelines

How should you use runtime assertions? First, runtime assertions shouldn't be used as a crutch for bad coding. You want to make your code bug-free in the most effective way. Defensive programming doesn't mean writing lousy code and peppering it with assertions. If you don't already know it, you'll find that in the long run it's much less work to write good code from the start; bad code is often such a mess it can't even be fixed without starting over again.

When should you write runtime assertions? As you write the code, not later. When you're writing the code you have invariants in mind anyway, and writing them down is a useful form of documentation. If you postpone it, you're less likely to do it.

Runtime assertions are not free. They can clutter the code, so they must be used judiciously. Obviously you want to write the assertions that are most likely to catch bugs. Good programmers will typically use assertions in these ways:

- At the start of a procedure, to check that the state in which the procedure is invoked is as expected – that is, to check the precondition. This makes sense because a high proportion of errors are related to misunderstandings about interfaces between procedures.
- At the end of a complicated procedure, to check that the result is plausible – that is, to check the postcondition. In a procedure that computes square roots for example, you might write an assertion that squares the result to check that it's (roughly) equal to the argument. This kind of assertion is sometimes called a *self check*.
- When an operation is about to be performed that has some external effect. For example, in a radiotherapy machine, it would make sense to check before turning on the beam that the intensity is within reasonable bounds.

Runtime assertions can also slow execution down. Novices are usually much more concerned about this than they should be. The practice of writing runtime assertions for testing the code but turning them off in the official release is like removing seat belts from a car after the safety tests have been performed. A good rule of thumb is that if you think a runtime assertion is necessary, you should worry about the performance cost only when you have evidence (eg, from a profiler) that the cost is really significant.

Nevertheless, it makes no sense to write absurdly expensive assertions. Suppose, for example, you are given an array and an index at which an element has been placed. It would be reasonable to check that the element is there. But it would not be reasonable to check that the element is nowhere else, by searching the array from end to end: that would turn an operation that executes in constant time into one that takes linear time (in the length of the array).

10.1.2 Catching Common Exceptions

Because Java is a safe language, its runtime environment – the Java Virtual Machine (JVM) – already includes runtime assertions for several important classes of error:

- Calling a method on a null object reference;
- Accessing an array out of bounds;
- Performing an invalid downcast.

These errors cause unchecked exceptions to be thrown. Moreover, the classes of the Java API themselves throw exceptions for erroneous conditions.

It is good practice to catch all these exceptions. A simple way to do this is to write a handler at the top of the program, in the main method, that terminates the program appropriately (eg, with an error message to the user, and perhaps attempting to close open files).

Note that there are some exceptions that are thrown by the JVM that you should no try to handle. Stack overflows and out-of-memory errors, for example, indicate that the program has run out of resources. In these circumstances, there's no point making things worse by trying to do more computation.

10.1.3 Checking the Rep Invariant

A very useful strategy for finding errors in an abstract type with a complex representation is to encode the rep invariant as a runtime assertion. The best way to do this is to write a method

public void checkRep ()

that throws an unchecked exception if the invariant does not hold at the point of call. This method can be inserted in the code of the data type, or called in a testbed from the outside.

Checking the rep invariant is much more powerful than checking most other kinds of invariants, because a broken rep often only results in a problem long after it was broken. With *checkRep*, you are likely to find catch the error much closer to its source. You should probably call *checkRep* at the start and end of every method, in case there is a rep exposure that causes the rep to be broken between calls. You should also remember to instrument observers, since they may mutate the rep (as a benevolent side effect).

There is an extensive literature on runtime assertions, but interestingly, there seems to be very little knowledge in industry of how to use *repCheck*.

Usually, checking the rep invariant will be too computationally intensive for the kind of runtime assertion that you would leave in production code. So you should use *checkRep* primarily in testing. For production checks, you should consider at which points the code may fail because of a broken representation invariant, and insert appropriate checks there.

10.1.4 Assert Framework

Runtime assertions can clutter up the code. It's especially bad if a reader can't easily tell which parts of the code are assertions and which are doing the actual computation. For this reason, and to make the writing of assertions more systematic and less burdensome, it's a good idea to implement a small framework for assertions.

Some programming languages, such as Eiffel, come with assertion mechanisms built-in. And requests for such a mechanism have topped all other requests for changes to Java. There are also many 3rd party tools and frameworks for adding assertions to code and for controlling them.

In practice, though, it's easy to build a small framework yourself. One approach is to implement a class, *Assert* say, with a method

public static void assert (boolean b, String location)

that throws an unchecked exception when the argument is false, containing a string indicating the location of the failed assertion. This class can encapsulate logging and error reporting. To use it, one simply writes assertions like this:

Assert.assert (x != null, "MyClass.MyMethod");

It's also possible to use Java's reflection mechanism to mitigate the need to provide location information.

10.1.5 Assertions in Subclasses

When we study subtyping, we'll see how the pre- and post-conditions of a subclass should be related to the pre- and post-conditions of its superclass in certain ways. This suggests opportunities for additional runtime checking, and also for subclasses to reuse assertion code from superclasses.

Surprisingly, most approaches to checking assertions in subclasses have been conceptually flawed. These recent papers explains why this is, and shows how to develop an assertion framework for subclasses:

- Robby Findler, Mario Latendresse, and Matthias Felleisen. Behavioral Contracts and Behavioral Subtyping. *Foundations of Software Engineering*, 2001.
- Robby Findler and Matthias Felleisen. Contract Soundness for Object-Oriented Languages. *Object-Oriented Programming, Systems, Languages, and Applications*, 2001.

see http:
www.cs.rice.edu/~robby/publications/.

10.1.6 Responding to Failure

Now we come to the question of what to do when an assertion fails. You might feel tempted to try and fix the problem on the fly. This is almost always the wrong thing to do. It makes the code more complicated, and usually introduces even more bugs. You're unlikely to be able to guess the cause of the failure; if you can, you could probably have avoided the bug in the first place.

On the other hand, it often makes sense to execute some special actions irrespective of the exact cause of failure. You might log the failure to a file, and/or notify the user on the screen, for example. In a safety critical system, deciding what actions are to be performed on failure is tricky and very important; in a nuclear reactor controller, for example, you probably want to remove the fuel rods if you detect that something is not quite right.

Sometimes, it's best not to abort execution at all. When our compiler fails, it makes sense to abort completely. But consider a failure in a word processor. If the user issues a command that fails, it would be much better to signal the failure and abort the command but not close the program; then the user can mitigate the effects of the failure (eg, by saving the buffer under a different name, and only then closing the program).