Extracted from:

Debug It!

Find, Repair, and Prevent Bugs in Your Code

The Pragmatic Programmers

Debug It!

Find, Repair, & Prevent Bugs in Your Code

Paul Butcher

Edited by Jacquelyn Carter

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

http://www.pragprog.com

Printed in the United States of America.

ISBN-10: 1-934356-28-X

ISBN-13: 978-1-934356-28-9

Printed on acid-free paper.

B1.0 printing, June 17, 2009

Version: 2009-6-16

Chapter 10

Teach Your Software to Debug Itself

Plenty has been written about how to write good software. Much less has been written about how to create software that is easy to debug.

The good news is that if you follow the normal principles of good software construction—separation of concerns, avoiding duplication, information hiding and so on—as well as creating software that is well structured, easy to understand and easy to modify, you will also create software that is easy to debug. There is no conflict between good design and debugging.

Nevertheless, there are a few additional things that you can put in place that will help when you find yourself tracking down a problem. In this chapter we'll cover some approaches that can make debugging easier or even, on occasion, unnecessary:

- Validating assumptions automatically with assertions.
- Debugging builds.
- Detecting problems in exception handling code automatically.

10.1 Assumptions and Assertions

Every piece of code is built upon a platform of myriad assumptions—things that have to be true for it to behave as expected. More often than not, bugs arise because one or more of these assumptions are violated or turn out to be mistaken.

Joe Asks...

Do I Need Assertions if I Have Unit Tests?

Some people argue that automated unit tests are a better solution to the problem that assertions are trying to solve. This line of thought probably arises to some extent from the unfortunate fact that the functions provided by JUnit to verify conditions within tests are also (confusingly) called assertions.

It isn't a question of either/or, but of both/and. Assertions and unit tests are solving related, but different problems. Unit tests can't detect a bug that isn't invoked by a test. Assertions can detect a bug at any time, whether during testing or otherwise.

One way to think of unit tests is that they are (in part) the means by which you ensure that all of your assertions are executed regularly.

It's impossible to avoid making such assumptions and pointless to try. But the good news is that not only can we verify that they hold, we can do so automatically with *assertions*.

What does an assertion look like? In Java, they can take two forms—the first, simpler form is:

```
assert «condition»;
```

The second form includes a message that is displayed if the assertion fails:

```
assert «condition» : «message»;
```

Whichever form you use, whenever it's executed an assertion evaluates its condition.[1] If the condition evaluates to true, then it takes no action. If, on the other hand, it evaluates to false, it throws an AssertionError exception, which normally means that the program exits immediately.

So much for the theory, how does this work in practice?

1. If assertions are enabled, which we'll get to soon.

An Example

Imagine that we're writing an application that needs to make HTTP requests. HTTP requests are very simple, comprising just a few lines of text. The first line specifies the method (such as GET or POST), a URI and which version of the HTTP protocol we're using. Subsequent lines contain a series of key/value pairs (one per line).[2] For a GET request, that's it (other requests might also include a body).

We might define a small Java class HttpMessage that can generate GET requests as follows:[3]

```
public class HttpMessage {

    private TreeMap<String, String> headers = new TreeMap<String, String>();

❶   public void addHeader(String name, String value) {
        headers.put(name, value);
    }

❷   public void outputGetRequest(OutputStream out, String uri) {
        PrintWriter writer = new PrintWriter(out, true);

        writer.println("GET " + uri + " HTTP/1.1");
        for (Map.Entry<String, String> e : headers.entrySet())
            writer.println(e.getKey() + ": " + e.getValue());
    }
}
```

It's very simple—addHeader()❶ just adds a new key/value pair to the headers map and outputGetRequest()❷ generates the start line, followed by each key/value in turn.

Here's how we might use it:

```
HttpMessage message = new HttpMessage();

message.addHeader("User-Agent", "Debugging example client");
message.addHeader("Accept", "text/html,test/xml");

message.outputGetRequest(System.out, "/path/to/file");
```

Which will generate the following:

```
GET /path/to/file HTTP/1.1
Accept: text/html,text/xml
User-Agent: Debugging example client
```

2. See the *Hypertext transfer protocol* [iet99] specification for further details.
3. Of course, you wouldn't write this code yourself given the number of well-debugged HTTP libraries available. But it's a nice simple example for our purposes.

Joe Asks...

How Do I Choose a Good Assert Message?

An early reviewer spotted a poster in, of all places, Google's Beijing offices that read "Make sure that your error messages aid in debugging, and don't just tell you that you need to debug."

The example that they cited was an assertion of the general form:

```
assert_lists_are_equal(list1, list2);
```

If this fails, it tells you that the lists are not equal. You still have to go through the code trying to find where the lists started to differ. It would be better to highlight the first element where the difference occurs, whether the order has changed, or something else that gives you a head-start diagnosing the problem.

So far, so simple. What could possibly go wrong?

Well, our code is very trusting. It's just taking what it's given and passing it through as-is. Which means that if it's called with bad arguments it will end up generating invalid HTTP requests. If, for example, addHeader() is called like this:

```
message.addHeader("", "a-value");
```

We'll end up generating the following header, which is sure to confuse any server we send it to:

```
: a-value
```

We can automatically detect if this happens by placing the following assertion at the start of addHeader():

```
assert name.length() > 0 : "name cannot be empty";
```

Now, if we call addHeader() with an empty string, when assertions are enabled the program exits immediately with:

```
Exception in thread "main" java.lang.AssertionError: name cannot be empty
    at HttpMessage.addHeader(HttpMessage.java:17)
    at Http.main(Http.java:16)
```

Wait a Second—What Just Happened?

Let's take a moment to reflect on what we've just done. We may have only added a single, simple line of code to our software, but that line has achieved something profound. We've taught our software to debug itself. Now, instead of us having to hunt down the bug, the software itself notices when something's gone wrong and tells us about it.

Hopefully this happens during testing, before the embarrassment of it being discovered by a user, but assertions are still helpful when tracking down bugs reported from the field. As soon as we find a way to reproduce the problem, there's a good chance that our assertions will immediately pinpoint the assumption that's being violated, dramatically saving time during diagnosis.

Example, Take Two

Now that we've started down this road, how far can we go? What other kinds of bugs can we detect automatically?

Detecting empty strings is fair enough, but are there any other obviously broken ways in which our class might be used? Once we start thinking in this way, we can find plenty.

For a start, empty strings aren't the only way that we could create an invalid header—the HTTP specification defines a number of characters that aren't allowed to appear in header names. We can automatically ensure that we never try to include such characters by adding the following to the top of addHeader():[4]

```
assert !name.matches(".*[\\(\\)<>@,;:\\\"/\\[\\]\\?=\\{\\} ].*") :
    "Invalid character in name";
```

Next, what does the following sequence of calls mean?

```
message.addHeader("Host", "somewhere.org");
message.addHeader("Host", "nowhere.com");
```

HTTP headers can only appear once in a message, so adding one twice

4. Don't worry too much about the hairy regular expression in this code—it's just matching a simple set of characters. It looks more complicated than it might because some of the characters need to be escaped with backslashes, and those backslashes themselves also need to be escaped.

has to be a bug.[5] A bug that we can catch automatically by adding the following to the top of addHeader():

```
assert !headers.containsKey(name) : "Duplicate header: " + name;
```

Other checks we might consider (depending on exactly how we foresee our class being used) might include:

- Verifying that outputGetRequest() is only called once and that addHeader() isn't called afterwards.
- Verifying that headers we know we always want to include in every request are always added.
- Checking the values assigned to headers to make sure that they are of the correct form (that the Accept header, for example, is always given a list of MIME types).

So much for the example—are there any general rules we can use to help us work out what kind of things we might assert?

Contracts, Pre-Conditions, Post-Conditions and Invariants

One way of thinking about the interface between one piece of code and another is as a *contract*. The calling code promises to provide the called code with an environment and arguments that confirm to its expectations. In return, the called code promises to carry out certain actions or return certain values that the calling code can then use.

It's helpful to consider three types of condition that, taken together, make up a contract:

Pre-conditions: The pre-conditions for a method are those things that must hold before it's called in order for it to behave as expected. The pre-conditions for our addHeader() method are that its arguments are non-empty, don't contain invalid characters, and so on.

Post-conditions: The post-conditions for a method are those things that it guarantees will hold after it's called (as long as it's pre-conditions were met). A post-condition for our addHeader() method is that the size of the headers map is one greater than it was before.

5. Note to HTTP specification lawyers—I am aware that there are occasions where headers can legitimately appear more than once. But they can always be replaced by a single header that combines the values and for the sake of a simple example, I'm choosing to ignore this subtlety.

Invariants: The invariants of an object are those things that (as long as its method's pre-conditions are met before they're called) it guarantees will *always* be true. That the cached length of a linked list is always equal to the length of the list, for example.

If you make a point of writing assertions that capture each of these three things whenever you implement a class, you will naturally end up with software that automatically detects a wide range of possible bugs.

Switching Assertions On and Off

One key aspect of assertions that we've already alluded to is that they can be disabled. Typically we choose to enable them during development and debugging, but disable them in production.

In Java, we switch assertions on and off when we start the application by using the following arguments to the java command:

```
-ea[:<packagename>...|:<classname>]
-enableassertions[:<packagename>...|:<classname>]
              enable assertions
-da[:<packagename>...|:<classname>]
-disableassertions[:<packagename>...|:<classname>]
              disable assertions
-esa | -enablesystemassertions
              enable system assertions
-dsa | -disablesystemassertions
              disable system assertions
```

In other languages, assertions are enabled and disabled using other mechanisms. In C and C++ for example, we do so at build time using conditional compilation.

Why might we choose to switch them off? There are two reasons—efficiency and robustness.

Evaluating assertions takes time and doesn't contribute anything to the functionality of the software (after all, if the software is functioning correctly, none of the assertions should ever do anything). If an assertion is in the heart of a performance critical loop, or the condition takes a while to evaluate (thinking back to our earlier example, an assertion that involved parsing the HTTP message to check that it's well-formed) it is possible to have a detrimental effect on performance.

A more pertinent reason for disabling assertions, however, is robustness. If an assertion fails, the software unceremoniously exits with a

terse and (to an end-user) unhelpful message. Or if our software is a long-running server, a failed assertion will kill the server process without tidying up after itself, leaving data in who knows what state. While this may be perfect acceptable (desirable in fact) when we're developing and debugging, it almost certainly isn't what we want in production software.

Instead, production software should be written to be *fault tolerant* or to *fail safe* as appropriate. How you go about achieving this is outside the scope of this book, but it does bring us onto the thorny subject of *defensive programming*.

Defensive Programming

Defensive programming is one of the many terms in software development that means different things to different people. What we're talking about here is the common practice of achieving small-scale fault tolerance by writing code that operates correctly (for some definition of correctly) in the presence of bugs.

Software should be robust in production, fragile when debugging

But defensive programming is a double-edged sword—from the point of view of debugging, it just makes our lives harder. It transforms what would otherwise be simple and obvious bugs into bugs that are obscure, difficult to detect, and difficult to diagnose. We may want our software to be as robust as possible in production, but it's much easier to debug *fragile* software that falls over immediately when a bug manifests.

A common example is the almost universal **for**-loop idiom, in which, instead of writing:

```
for (i = 0; i != iteration_count; ++i)
    «Body of loop»
```

We write the following defensive version:

```
for (i = 0; i < iteration_count; ++i)
    «Body of loop»
```

In almost all cases both loops behave identically iterating from zero to iteration_count - 1. So why do so many of us automatically write the second, not the first?[6]

6. Actually, this idiom is starting to fall out of favor in the C++ community thanks to the Standard Template Library, but nevertheless there are millions of examples in existence.

The reason is because if the body of the loop happens to assign to i so that it becomes larger than iteration_count, the first version of our loop won't terminate. By using < in our test instead of != we can guarantee that the loop will terminate if this happens.

The problem with this is that if the loop index *does* become larger than iteration_count, it almost certainly means that the code contains a bug. And whereas with the first version of the code we would immediately notice that it did (because the software hung inside an infinite loop), now it may not be at all obvious. It will probably bite us at some point in the future and be very difficult to diagnose.

Another example. Imagine that we're writing a function that takes a string and returns true if it's all uppercase, false otherwise. Here's one possible implementation in Java:

```
public static boolean allUpper(String s) {
    CharacterIterator i = new StringCharacterIterator(s);

    for (char c = i.first(); c != CharacterIterator.DONE; c = i.next())
        if (Character.isLowerCase(c))
            return false;

    return true;
}
```

A perfectly reasonable function—but if for some reason we pass null to it, our software will crash. With this in mind, some developers would add something along these lines to the beginning:

```
if (s == null)
    return false;
```

So now the code won't crash—but what does it *mean* to call this function with null? There's an excellent chance that any code that does so contains a bug, which we've now masked.

Assertions provide us with a very simple solution to this problem. Wherever you find yourself writing defensive code, make sure that you protect that code with assertions.

So now our protective code at the start of allUpper() becomes:

```
assert s != null : "Null string passed to allUpper";
if (s == null)
    return false;
```

And our earlier **for**-loop becomes:

```
for (i = 0; i < iteration_count; ++i)
```

Assertions and Language Culture

A programming language is more than just syntax and semantics. Each language has one or more communities built up around with their own idioms, norms, and practices. How (or if) assertions are habitually used in a language depends in part on that community.

Although assertions can be used in any language, they are more widespread in the C/C++ community than any other of the major languages. In particular, they aren't particularly widely used in Java, probably because they only became officially supported in Java 1.4 (although there are signs that assertions are catching on within the wider Java community with JVM-based languages such as Groovy and Scala encouraging their use).

In part, this may be because there are more opportunities for things to go wrong in C/C++. Pointers can wreak havoc if used incorrectly, strings and other data structures can overflow. These kinds of problems simply can't occur in languages like Java and Ruby.

But that doesn't mean that assertions aren't valuable in these languages—just that we don't need to use them to check for this kind of low-level error. They're still extremely useful for checking for higher-level problems.

```
    «Body of loop»
assert i == iteration_count;
```

We now have the best of both worlds—robust production software and fragile development/debugging software.

Assertion Abuse

As with many tools, assertions can be abused. There are two common mistakes you need to avoid—assertions with side-effects and using them to detect errors instead of bugs.

Cast your mind back to our HttpMessage class and imagine that we want to implement a method which removes a header we added previously. If we want to assert that it's always called with an existing header, we might be tempted to implement it as follows (the Java remove() method returns null if the key doesn't exist):

```
public void removeHeader(String name) {
    assert headers.remove(name) != null;
}
```

The problem with this code is that the assertion contains a *side-effect*. If we run the code without assertions enabled, it will no longer behave correctly because, as well as removing the check for null, we're *also* removing the call to remove().

Better (and more self-documenting) would be to write it as:

```
assert headers.containsKey(name);
headers.remove(name);
```

An assertion's task is to check that the code is working as it should, not to affect *how* it works. For this reason, it's important that you test with assertions disabled as well as with assertions enabled. If any side-effects have crept in, you want to find them before the user does.

Assertions are not an error-handling mechanism

Assertions are a bug-detection mechanism, not an error-handling mechanism. What is the difference? Errors may be undesirable, but they *can* happen in bug-free code. Bugs, on the other hand, are impossible if the code is operating as intended. Here are some examples of conditions that almost certainly should not be handled with an assertion:

- Trying to open a file and discovering that it doesn't exist.
- Detecting and handling invalid data received over a network connection.
- Running out of space while writing to a file.
- Network failure.

Error-handling mechanisms such as exceptions or error codes are the right way to handle these situations.

We've mentioned that assertions are typically disabled in production builds and enabled in development or debug builds. But what exactly is a debug build?

10.2 Debugging Builds

Many teams find it helpful to create a *debugging build*, which differs from a release build in various ways designed to help reproduce and diagnose problems.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Debug It!'s Home Page
http://pragprog.com/titles/pbdp
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/pbdp.

Contact Us

Online Orders:	www.pragprog.com/catalog
Customer Service:	support@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com
Contact us:	1-800-699-PROG (+1 919 847 3884)