8 Boundaries

by James Grenning

We seldom control all the software in our systems. Sometimes we buy third-party packages or use open source. Other times we depend on teams in our own company to produce components or subsystems for us. Somehow we must cleanly integrate this foreign code with our own. In this chapter we look at practices and techniques to keep the boundaries of our software clean.

Using Third-Party Code

There is a natural tension between the provider of an interface and the user of an interface. Providers of third-party packages and frameworks strive for broad applicability so they can work in many environments and appeal to a wide audience. Users, on the other hand, want an interface that is focused on their particular needs. This tension can cause problems at the boundaries of our systems.

Let's look at `java.util.Map` as an example. As you can see by examining Figure 8-1, `Maps` have a very broad interface with plenty of capabilities. Certainly this power and flexibility is useful, but it can also be a liability. For instance, our application might build up a `Map` and pass it around. Our intention might be that none of the recipients of our `Map` delete anything in the map. But right there at the top of the list is the `clear()` method. Any user of the `Map` has the power to clear it. Or maybe our design convention is that only particular types of objects can be stored in the `Map`, but `Maps` do not reliably constrain the types of objects placed within them. Any determined user can add items of any type to any `Map`.

Figure 8-1 The methods of `Map`

If our application needs a `Map` of `Sensors`, you might find the sensors set up like this:

```
Map sensors = new HashMap();
```

Then, when some other part of the code needs to access the sensor, you see this code:

```
Sensor s = (Sensor)sensors.get(sensorId );
```

We don't just see it once, but over and over again throughout the code. The client of this code carries the responsibility of getting an `Object` from the `Map` and casting it to the right type. This works, but it's not clean code. Also, this code does not tell its story as well as it could. The readability of this code can be greatly improved by using generics, as shown below:

```
Map<Sensor> sensors = new HashMap<Sensor>();
…
Sensor s = sensors.get(sensorId );
```

However, this doesn't solve the problem that `Map<Sensor>` provides more capability than we need or want.

Passing an instance of `Map<Sensor>` liberally around the system means that there will be a lot of places to fix if the interface to `Map` ever changes. You might think such a change to be unlikely, but remember that it changed when generics support was added in Java 5. Indeed, we've seen systems that are inhibited from using generics because of the sheer magnitude of changes needed to make up for the liberal use of `Map`s.

A cleaner way to use `Map` might look like the following. No user of `Sensors` would care one bit if generics were used or not. That choice has become (and always should be) an implementation detail.

```
public class Sensors {
private Map sensors = new HashMap();

public Sensor getById(String id) {
  return (Sensor) sensors.get(id);
}

//snip
}
```

The interface at the boundary (`Map`) is hidden. It is able to evolve with very little impact on the rest of the application. The use of generics is no longer a big issue because the casting and type management is handled inside the `Sensors` class.

This interface is also tailored and constrained to meet the needs of the application. It results in code that is easier to understand and harder to misuse. The `Sensors` class can enforce design and business rules.

We are not suggesting that every use of `Map` be encapsulated in this form. Rather, we are advising you not to pass `Map`s (or any other interface at a boundary) around your system. If you use a boundary interface like `Map`, keep it inside the class, or close family of classes, where it is used. Avoid returning it from, or accepting it as an argument to, public APIs.

Exploring and Learning Boundaries

Third-party code helps us get more functionality delivered in less time. Where do we start when we want to utilize some third-party package? It's not our job to test the third-party code, but it may be in our best interest to write tests for the third-party code we use.

Suppose it is not clear how to use our third-party library. We might spend a day or two (or more) reading the documentation and deciding how we are going to use it. Then we might write our code to use the third-party code and see whether it does what we think. We would not be surprised to find ourselves bogged down in long debugging sessions trying to figure out whether the bugs we are experiencing are in our code or theirs.

Learning the third-party code is hard. Integrating the third-party code is hard too. Doing both at the same time is doubly hard. What if we took a different approach? Instead of experimenting and trying out the new stuff in our production code, we could write some tests to explore our understanding of the third-party code. Jim Newkirk calls such tests *learning tests*.[1]

In learning tests we call the third-party API, as we expect to use it in our application. We're essentially doing controlled experiments that check our understanding of that API. The tests focus on what we want out of the API.

Learning `log4j`

Let's say we want to use the apache `log4j` package rather than our own custom-built logger. We download it and open the introductory

documentation page. Without too much reading we write our first test case, expecting it to write “hello” to the console.

```
    @Test
  public void testLogCreate() {
    Logger logger = Logger.getLogger(“MyLogger”);
    logger.info(“hello”);
  }
```

When we run it, the logger produces an error that tells us we need something called an `Appender`. After a little more reading we find that there is a `ConsoleAppender`. So we create a `ConsoleAppender` and see whether we have unlocked the secrets of logging to the console.

```
    @Test
  public void testLogAddAppender() {
    Logger logger = Logger.getLogger(“MyLogger”);
    ConsoleAppender appender = new ConsoleAppender();
    logger.addAppender(appender);
    logger.info(“hello”);
  }
```

This time we find that the `Appender` has no output stream. Odd—it seems logical that it’d have one. After a little help from Google, we try the following:

```
    @Test
  public void testLogAddAppender() {
    Logger logger = Logger.getLogger(“MyLogger”);
    logger.removeAllAppenders();
    logger.addAppender(new ConsoleAppender(
        new PatternLayout(“%p %t %m%n”),
        ConsoleAppender.SYSTEM_OUT));
    logger.info(“hello”);
  }
```

That worked; a log message that includes “hello” came out on the console! It seems odd that we have to tell the `ConsoleAppender` that it writes to the console.

Interestingly enough, when we remove the `ConsoleAppender.SystemOut` argument, we see that “hello” is still

printed. But when we take out the `PatternLayout`, it once again complains about the lack of an output stream. This is very strange behavior.

Looking a little more carefully at the documentation, we see that the default `ConsoleAppender` constructor is "unconfigured," which does not seem too obvious or useful. This feels like a bug, or at least an inconsistency, in `log4j`.

A bit more googling, reading, and testing, and we eventually wind up with Listing 8-1. We've discovered a great deal about the way that `log4j` works, and we've encoded that knowledge into a set of simple unit tests.

Listing 8-1 `LogTest.java`

```
public class LogTest {
private Logger logger;

@Before
public void initialize() {
   logger = Logger.getLogger("logger");
   logger.removeAllAppenders();
   Logger.getRootLogger().removeAllAppenders();
}
@Test
public void basicLogger() {
   BasicConfigurator.configure();
   logger.info("basicLogger");
}
@Test
public void addAppenderWithStream() {
logger.addAppender(new ConsoleAppender(
   new PatternLayout("%p %t %m%n"),
   ConsoleAppender.SYSTEM_OUT));
  logger.info("addAppenderWithStream");
}
   @Test
   public void addAppenderWithoutStream() {
    logger.addAppender(new ConsoleAppender(
```

```
            new PatternLayout("%p %t %m%n")));
          logger.info("addAppenderWithoutStream");
        }
    }
```

Now we know how to get a simple console logger initialized, and we can encapsulate that knowledge into our own logger class so that the rest of our application is isolated from the `log4j` boundary interface.

Learning Tests Are Better Than Free

The learning tests end up costing nothing. We had to learn the API anyway, and writing those tests was an easy and isolated way to get that knowledge. The learning tests were precise experiments that helped increase our understanding.

Not only are learning tests free, they have a positive return on investment. When there are new releases of the third-party package, we run the learning tests to see whether there are behavioral differences.

Learning tests verify that the third-party packages we are using work the way we expect them to. Once integrated, there are no guarantees that the third-party code will stay compatible with our needs. The original authors will have pressures to change their code to meet new needs of their own. They will fix bugs and add new capabilities. With each release comes new risk. If the third-party package changes in some way incompatible with our tests, we will find out right away.

Whether you need the learning provided by the learning tests or not, a clean boundary should be supported by a set of outbound tests that exercise the interface the same way the production code does. Without these *boundary tests* to ease the migration, we might be tempted to stay with the old version longer than we should.

Using Code That Does Not Yet Exist

There is another kind of boundary, one that separates the known from the unknown. There are often places in the code where our knowledge seems to drop off the edge. Sometimes what is on the other side of the

boundary is unknowable (at least right now). Sometimes we choose to look no farther than the boundary.

A number of years back I was part of a team developing software for a radio communications system. There was a subsystem, the "Transmitter," that we knew little about, and the people responsible for the subsystem had not gotten to the point of defining their interface. We did not want to be blocked, so we started our work far away from the unknown part of the code.

We had a pretty good idea of where our world ended and the new world began. As we worked, we sometimes bumped up against this boundary. Though mists and clouds of ignorance obscured our view beyond the boundary, our work made us aware of what we *wanted* the boundary interface to be. We wanted to tell the transmitter something like this:

> *Key the transmitter on the provided frequency and emit an analog representation of the data coming from this stream.*

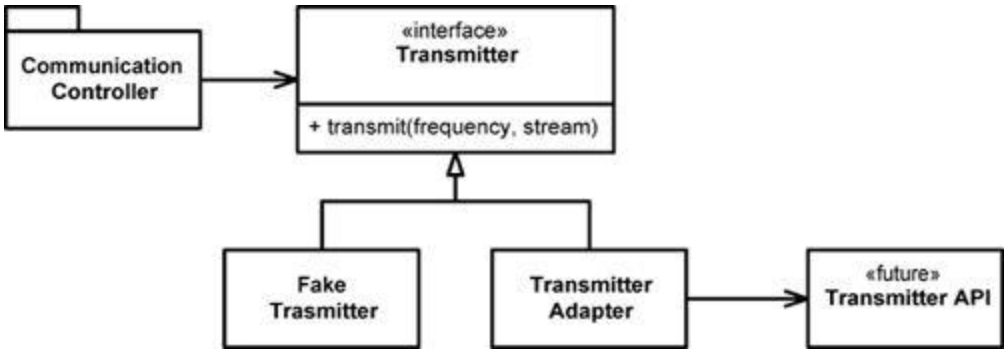
We had no idea how that would be done because the API had not been designed yet. So we decided to work out the details later.

To keep from being blocked, we defined our own interface. We called it something catchy, like `Transmitter`. We gave it a method called `transmit` that took a frequency and a data stream. This was the interface we *wished* we had.

One good thing about writing the interface we wish we had is that it's under our control. This helps keep client code more readable and focused on what it is trying to accomplish.

In Figure 8-2, you can see that we insulated the `CommunicationsController` classes from the transmitter API (which was out of our control and undefined). By using our own application specific interface, we kept our `CommunicationsController` code clean and expressive. Once the transmitter API was defined, we wrote the `TransmitterAdapter` to bridge the gap. The ADAPTER[2] encapsulated the interaction with the API and provides a single place to change when the API evolves.

Figure 8-2 Predicting the transmitter



This design also gives us a very convenient seam[3] in the code for testing. Using a suitable `FakeTransmitter`, we can test the `CommunicationsController` classes. We can also create boundary tests once we have the `TransmitterAPI` that make sure we are using the API correctly.

Clean Boundaries

Interesting things happen at boundaries. Change is one of those things. Good software designs accommodate change without huge investments and rework. When we use code that is out of our control, special care must be taken to protect our investment and make sure future change is not too costly.

Code at the boundaries needs clear separation and tests that define expectations. We should avoid letting too much of our code know about the third-party particulars. It's better to depend on something *you* control than on something you don't control, lest it end up controlling you.

We manage third-party boundaries by having very few places in the code that refer to them. We may wrap them as we did with `Map`, or we may use an ADAPTER to convert from our perfect interface to the provided interface. Either way our code speaks to us better, promotes internally consistent usage across the boundary, and has fewer maintenance points when the third-party code changes.

Bibliography

[BeckTDD]: *Test Driven Development,* Kent Beck, Addison-Wesley, 2003.

[GOF]: *Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison-Wesley, 1996.

[WELC]: *Working Effectively with Legacy Code*, Addison-Wesley, 2004.