Clean Boundaries

Christopher Simpkins
chris.simpkins@gatech.edu

Library Providers versus Library Users

Tension between providers of an interface and users of an interface

- Providers want to produce maximally flexible and useful functionality for a wide audience.
- Users want to focus on their needs.

This tension is particularly acute in standard library classes. Consider `java.util.Map` ...

A Key-Value Sensors Data Structure

Say you have an application that needs to maintain sensors that are identified by ids. This is precisely what `Map`s are for, so you use a `Map`. But ...

- your code passes around sensors, and most parts of your app should only retrieve sensors from the `Map`, and
- `java.util.Map` includes methods that allow users to delete items from the map.

The provider of `Map` is providing flexibility we don't want.

How to Deal with Overly General Library Classes

If we use `Map` to hold sensors, we end up with code like this:

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

There's a concept here that's begging to be represented in our system:

```
public class Sensors {
  private Map<String, Sensor> sensors = new HashMap<>();

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

- Wrapping not necessary for every use of `Map`, but generally not a good idea to pass `Map`s around.
- This "wrapping" technique is a good general approach to deal with overly general third-party libraries at boundaries.

Exploring and Learning Boundaries

When you need a capability, you can write the code yourself, or use a library that provides the capability.

- If a libary is available, “buy, don’t build.”
- Still have to learn the library.

Instead of studying documentation, write *learning tests*.

Learning `log4j`

We write a simple test the way we think the library should work:

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

When the test runs, we get an error saying that we need an `Appender`, so we look up `Appender`s in thr docs and update our test:

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

This time we get an error about a missing output stream, ...

Learning Tests Lead to Boundary Tests

... so we read a little more and come up with:

```
@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");
}
```

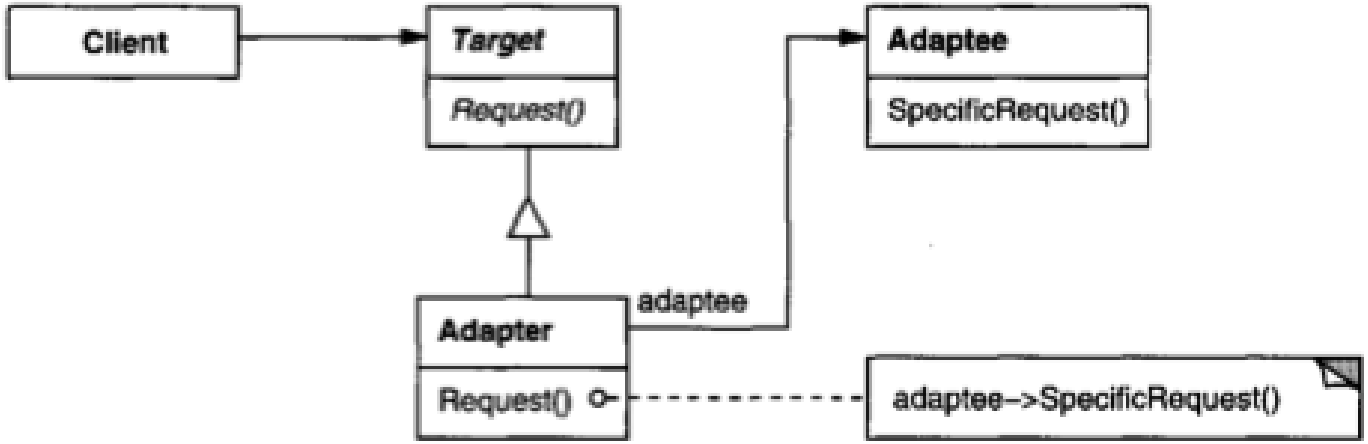
After a few more iterations of writing tests and reading docs, we know how `log4j` works and we have a set of tests with example code.

- These tests are *boundary tests* that use the library the same way as our production code.
- When a new version of the library is released, we can test it with our boundary tests before integrating it with production code.

The Adapter Pattern[1] (A.K.A Wrapper)

Intent: Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Structure



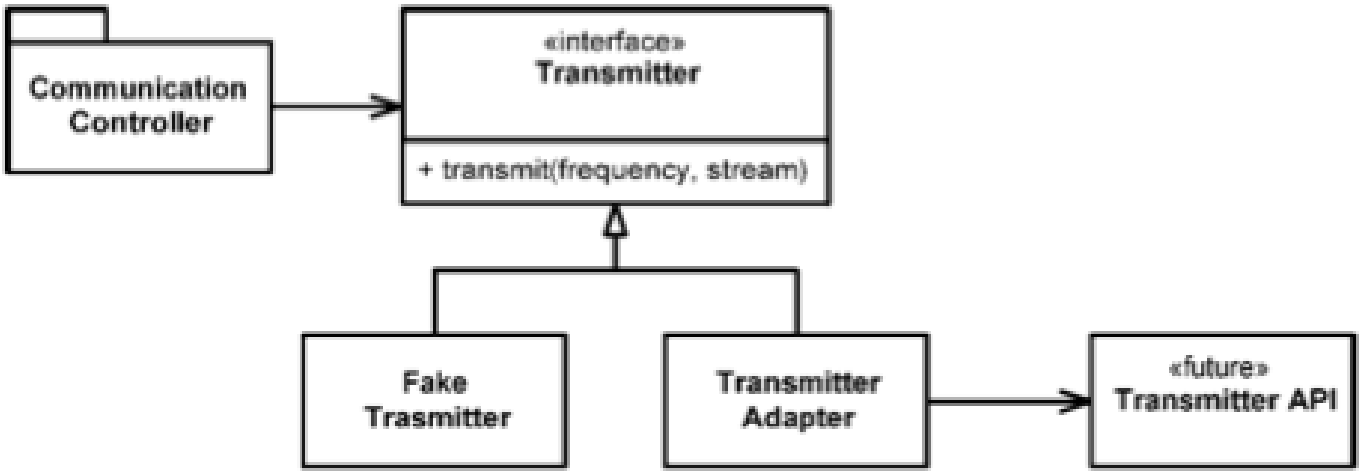
Participants

- **Target** defines the domain-specific interface that Client uses.
- **Client** collaborates with objects conforming to the Target interface.
- **Adaptee** defines an existing interface that needs adapting.
- **Adapter** adapts the interface of Adaptec to the Target interface.

[1] GoF, *Design Patterns*, 1994

Integrating Future Code

- Imagine we're a team writing an application that will use a hardware transmitter, but the transmitter's software is handled by another team that hasn't defined their software interface.
- We can define our own interface the way *we* want it to work.
- While we're waiting for the transmitter team, we create a fake implementation to work with.
- When the transmitter team finally gives us their interface, we can write an adapter to fit it to our interface.
- The rest of our code is unaffected.



Final Thoughts on Boundaries

- Third-party libraries out of our control.
- Boundary tests minimize surprises when migrating to new versions of libraries.
- Minimize the number of places in our code that accesses third-party libraries.
- Sequester third-party libraries in boundary code using the adapter pattern.