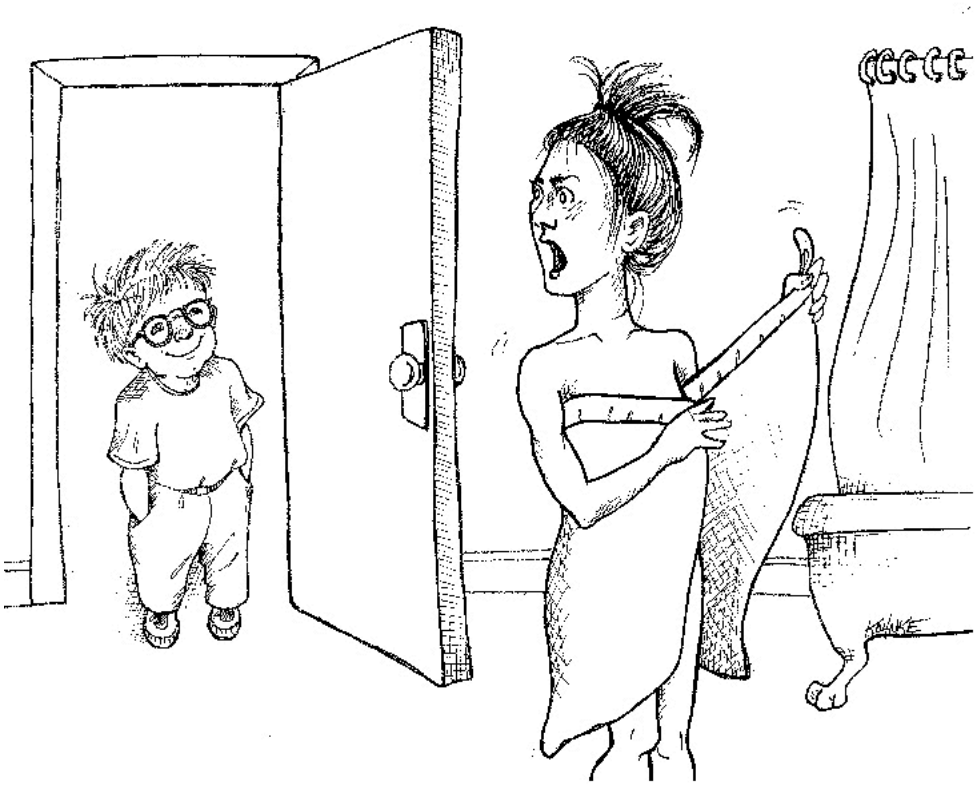
26

Clean 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 section, we look at practices and techniques to keep the boundaries of our software clean.

Third-Party IoT Framework: Lots o' Boundaries

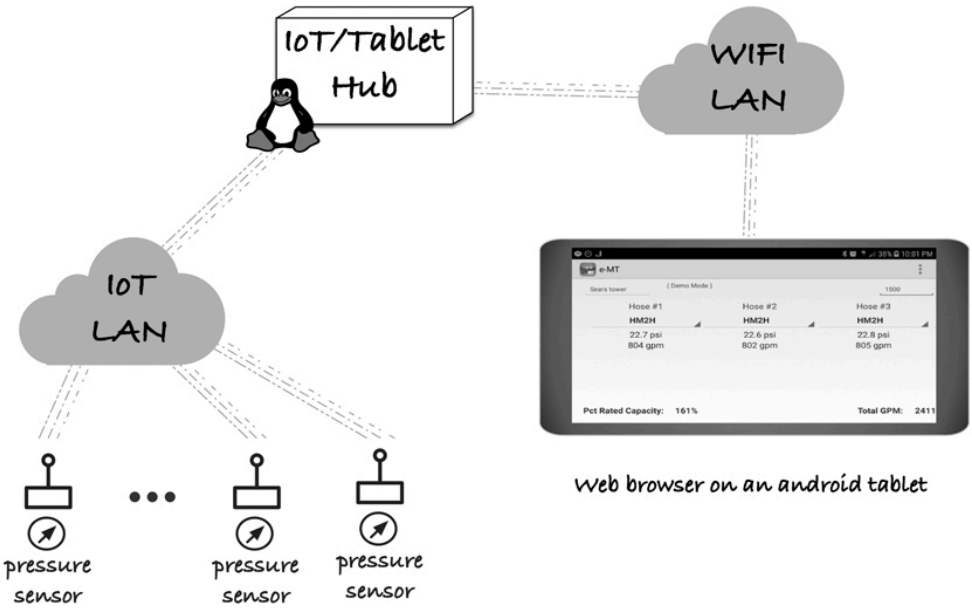
My brother had an idea for automating the manual process that users of his product perform. The product is used by a technician testing a high-capacity water pump. The pump is tested by flowing water and manually

measuring and recording water pressure at several nozzles. Given the geometry of the nozzles, pressure is converted into flow rate.

The pump is permanently installed in a commercial or industrial building; water flows outside, onto the curb, and down a storm drain. Pump capacities are measured in thousands of gallons per minute. A manual test can take 20 to 30 minutes, and a lot of water is wasted in the process. Typically, pressure measurements are made with a physical gauge at several locations: inside the building next to the pump, and outside at the nozzles, with records kept manually on a clipboard.

The big idea was to measure pressure with an electronic gauge that could report its pressure once per second and have the readings shown on a tablet computer or smart phone. The first design used Bluetooth to communicate from the sensor to the tablet. It worked great in the lab, but in the field there was always a brick wall (or worse) in the way. Bluetooth could not penetrate the brick wall. So we dreamed up the following idea.

With an off-the-shelf Internet of Things (IoT) infrastructure, we could get the water pressure measurements through brick walls.



We selected an IoT vendor with infrastructure that let us build our application in Python. Their existing radio system, along with our production code in Python, would make it possible for us to get a proof-of-concept prototype developed quickly.

We set off to build a prototype, knowing we might need a different vendor once we went to market. Why did we doubt this vendor? At the prototype stage, we weren't that concerned with cost. That could change by the time we went to market. Also, in the rapidly growing IoT market, this vendor may go out of business or get eaten by another IoT supplier. We wanted to preserve our business logic investment, so in our prototype, we needed clean code, separation of concerns, and a limited dependency on our vendor's infrastructure.

Nicely, our vendor provided a simple "hello world" IoT application to get started. Their "hello world" demo allowed us to send messages between a small Linux box (it was kind of like an industrial Raspberry Pi) and the little microcontroller-driven radios that would be married to our pressure sensors. We could morph their "hello world" example code into a layer that would contain all the dependencies on the vendor infrastructure.

Something I learned the hard way leads me to prefer starting with a working system and focusing on small changes to grow functionality and not break stuff. You know, it's easier to keep a system working than to fix it after you break it! Test-driven development (TDD) allowed me to see the value of small, verified changes.

The vendor had an elaborate application infrastructure we could take advantage of. It had a lot of capabilities. We just would not need everything it offered. We wanted to use the radios for some simple message passing, not marry the vendor (until death do us part). Consequently, we used very little of the vendor's very wide interface.

Let's start with the sensor and the hub getting to know each other. The vendor makes this very easy. The hub and radios interact using a remote procedure call (RPC) mechanism. The hub has a full Python environment while the radio is limited to MicroPython.

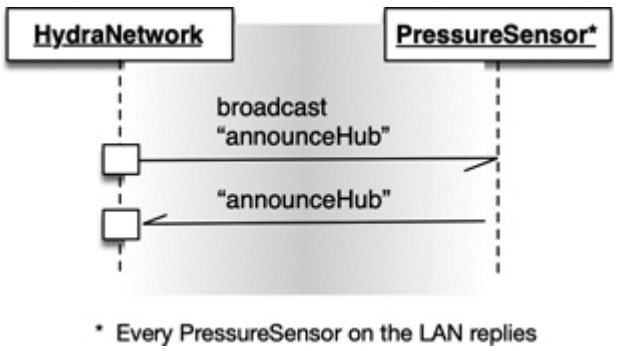
The project is code-named Hydra, after the many-headed monster. The class `HydraNetwork` encapsulates all dependencies in the hub on the IoT vendor. The functions are low complexity and delegate to the communications infrastructure. Starting at the beginning, initialization, the hub announces itself to all the radios listening, then polls the framework for replies.

```
# In the hub
class HydraNetwork:
  …
  def announceHub(self):
    self.broadcast('announceHub', '')
    self.poll_a_second()
```

Through the magic of the infrastructure RPC mechanism, the `announceHub()` function is called in the radio. The radio remembers the hub's address and sends an acknowledgment to the hub.

```
# In the radio
def announceHub():
  global hub_addr
  hub_addr = rpcSourceAddr()
  info = nodeIdentity()
  sendAck(hub_addr, 'announceHub', info)
```

This sequence chart shows the asynchronous communications between the `HydraNetwork` hub and its `PressureSensors` on the IoT LAN.



Now, going back to the hub, the infrastructure RPC facility calls the hub's `HydraNetwork.processAck()` function.

```
# In the hub
class HydraNetwork:
  …
  def processAck(self, command, payload):
    sensor_id = self.source_addr()
    self.ack_handler.handle_reply(sensor_id, command,
```

The `ack_handler` is an injected dependency, which happens to be an instance of `HydraDataSource`. The method `handle_reply()` passes the received bookkeeping and payload to other parts of the Hydra system. As it turns out, there is nothing to do for the `'announceHub'` ACK from the radio.

The `HydraDataSource` initiates a network PSI[1] read when its `flow_update()` method is called. The `flow_update()` method is called once per second during a flow test. Each update has a `sync_token` so that asynchronous measurement replies can be grouped together by second. When a new second starts, a report is saved from the previous second. More on that later.

---

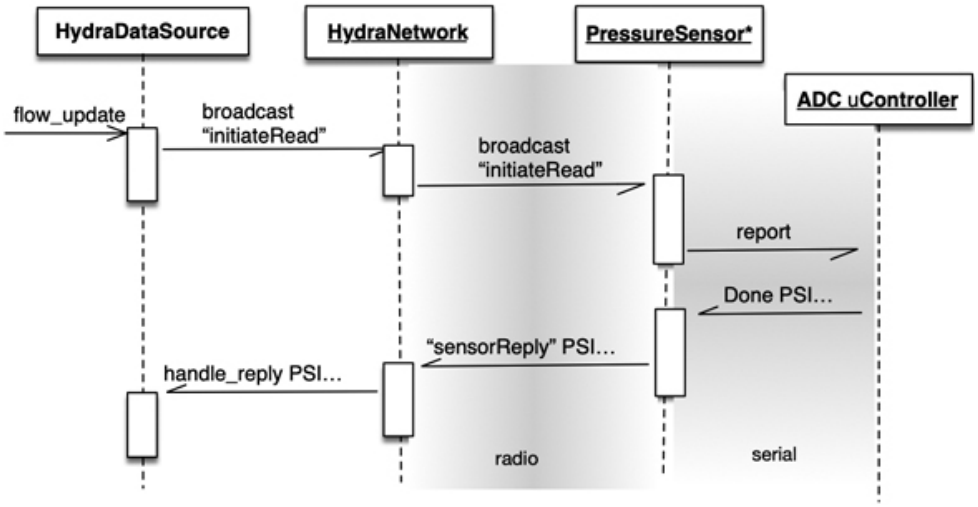
1. Pressure. Pounds per Square Inch.

```
# In the hub
class HydraDataSource:
   …
   def flow_update(self):
        self.sync_token.next()
        self.network.broadcast(
                        'initiateRead',
                         self.sync_token.value())
        self.report = self.reply_dispatcher.report()
```

When `'initiateRead'` is broadcast to the sensor network, each radio replies asynchronously. Again, via the IoT framework, `initiateRead()` is called in the `PressureSensor` module.

First, `initiateRead()` responds with an ACK and then calls `reportMeasurement(sync_token)` to get the latest PSI value.

I neglected to tell you about another boundary. There is a separate microcontroller responsible for reading the analog-to-digital converter (ADC) and doing the needed math. The call to `reportMeasurement()` writes to a serial connection asking the ADC microcontroller for the next measurement.




```
# In the radio
def initiateRead(sync_token):
    global hub_addr
    hub_addr = rpcSourceAddr()
    sendAck(hub_addr, 'initiateRead', sync_token)
    reportMeasurement(sync_token)

def reportMeasurement(sync_token):
    print 'RA ' + sync_token
@setHook(HOOK_STDIN)
def measurementDone(reply):
    stdin_info = getInfo(STDIN_INFO)
    if stdin_info != STDIN_RECEIVED:
      reply = 'Error: Invalid STDIN_INFO'+
                                str(stdin_info)
    sendAck(hub_addr,'SensorReply',reply[:MAX_SIZE])
```

It takes a little time to get a new ADC value and convert it to PSI. When the measurement is ready, the ADC microcontroller sends back a CSV string using the serial connection. The `HOOK_STDIN` notifies `measurementDone()`, providing a reply. If there is no serial communication error, the reply includes the `sync_token` and sensor configuration. The reply is sent as a `SensorReply` ACK to the hub.

Meanwhile, back at the hub, the hub is collecting ACK messages. ACK messages are sent to `handle_reply()`. A `'SensorReply'` ACK is given to the `reply_dispatcher.sensor_update()`. The `reply_dispatcher` knows the message format, and where to send it once it is parsed. The payload is a simple set of CSVs. The

`reply_dispatcher` and collaborators are test driven and free from IoT dependencies.

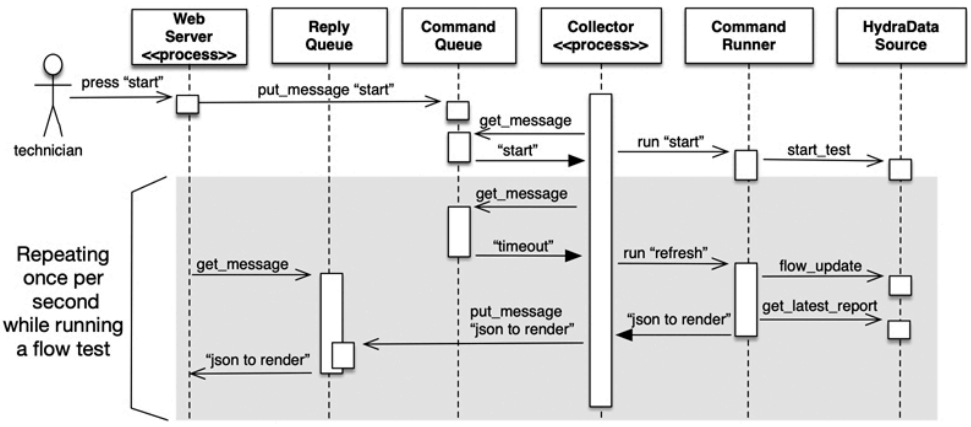
```
class HydraDataSource:
  …
  def handle_reply(self, sensor_id,
                   command, payload):
   sync_token = self.sync_token.value
    if command == 'SensorReply':
      result = self.reply_dispatcher.
                   sensor_update(sensor_id,
                                 sync_token,
                                 payload)
```

Because `HydraNetwork` encapsulated all IoT framework dependencies, we could substitute a different vendor's IoT infrastructure.

UI/Application Boundary

Next, let's look at the UI/application boundary. The user interface of the system is a Web browser on a tablet or smart phone. We used the popular Python/Flask Web server framework. The application runs on a small Linux box as two processes.

The `Collector` process collects measurements from the `HydraNetwork` . The `WebServer` process runs the customized Flask Web server in the main thread. The user has buttons to start and stop the flow test, which comes to the `Collector` in the `CommandQueue` . Measurement updates are sent asynchronously to the `ReplyQueue` once a synchronized group of measurements is collected.



The `Collector` process entry point is the `measurement_loop()` function, which waits on the `CommandQueue` with a one-second timeout. Each timeout initiates a `refresh` / `flow_update` of the measurement data all the way down to the ADC microcontroller (not shown in this diagram).

```
# Collector process entry function
def measurement_loop(command_q, reply_q,
                     command_runner, update_interval):
 command = None
  while command != HYDRA_EXIT:
    command = get_message(command_q, update_interval)
    if command == None:
      Command = HYDRA_REFRESH
    reply = command_runner.run(command)
    put_message(reply_q, reply)
```

There are a number of application objects not shown, but then they are not at the boundary. We'll look at a bigger picture soon.

Let's take a quick look at `hydra_run()`. This function is called from `main()` with its `data_source` injected. In production, a `HydraDataSource`, with its `HydraNetwork`, is injected. During development and testing, we have a simulated data source where a network is not needed.

The `hydra_run()` function puts everything in motion and is called from `main()`. The following code starts the `Collector` process running the Web server in the main thread of execution.

```
def hydra_run(port, data_source, polling_rate,
                                 logger):
  logger.info("Hydra start")
  command_runner = CommandRunner(data_source,
                                 logger)
  collector = Process(
                target=measurement_loop,
                args=(command_q, reply_q,
                      command_runner, polling_rate))
  collector.start()
```

```
    sleep(1)
    logger.info("Hydra collector started")
    run_hydra_web_server(port, command_q, reply_q)
    logger.info("Hydra web server exited")
    command_q.put(HYDRA_EXIT)
    collector.join()
    logger.info("Hydra shut down")
```

This is a little messy, as all the dependencies come together here. On the plus side, the code is not complex and probably won't change much.

Let's review some of the important attributes of our clean boundaries.

About the IoT isolation:

- The business logic is free from the radio system knowledge.
- IoT-dependent functions are low complexity.
- IoT-dependent functions are unlikely to change often or much.

About the UI isolation:

- Business logic is separate from the UI.
- Communications between the application to the Web server rely on a simple queuing mechanism.

About concurrency:

- None of the business logic is concerned with concurrency.
- Concurrency mechanisms are in low-complexity functions with a low risk of changes.

About object creation and binding:

- Object creation is separate from object usage.
- During initialization, the concrete objects are created and bound to processes and to each other.
- Binding is based on a need-to-know basis.

SOLID and Hexagonal Architecture

We've looked at the little pieces; now let's look at a bigger picture and the relationship between the classes in the design. Following the SOLID

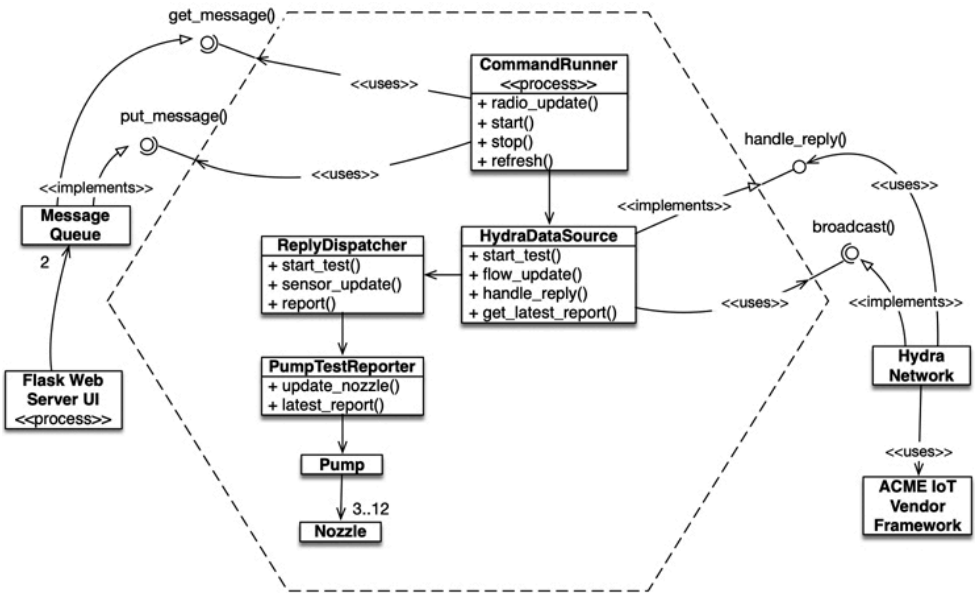
principles and isolating dependencies of the outside world naturally leads to something that Alistair Cockburn calls the *Hexagonal architecture*[2] (aka Ports and Adapters).

---

2. [Hex].

Inside the hexagon is all the product's logic and behavior. Instead of a big ball of mud inside, responsibilities are allocated to separate classes, functions, and domain objects. Outside the hexagon is the execution environment. The application depends on service abstraction layers at the edge of the hexagon.

There are two `MessageQueue` instances: the `CommandQueue` and the `ReplyQueue`. The UI, through the `CommandQueue`, drives the system. The user starts a test with a button on the Web page, which writes the "start" command to the `CommandQueue`. The start command works its way to the sensors that start collecting measurements. The `CommandRunner` does a timed wait for new UI commands. Each one-second timeout causes the `CommandRunner` to initiate a new update from the sensors. The previous second's sensor readings are bundled into a report that is sent to the `ReplyQueue`. The following diagram shows how that message starts the flow test. We also saw the connection of the `CommandRunner` to the queues in the `measurement_loop()` in a code snippet earlier.



The `HydraNetwork` encapsulates the dependencies on the ACME IoT vendor framework. `HydraNetwork.broadcast()` tells all sensors in the network to report their latest measurements. As individual radio replies are received, `HydraNetwork` calls `handle_reply()`, feeding the latest `PressureSensor` messages into the Hydra application. Hydra knows nothing about the ACME IoT.

The interfaces of `broadcast()` and `handle_reply()` have more to do with what the application wants from the ACME IoT than what the ACME IoT offers. `HydraDataSource` knows it has a network of sensors, but only needs it to broadcast the need for an update and to receive the updates. *It's important that the interfaces at the boundary of the hexagon are about the Hydra application's needs and not about ACME's offerings*. The ACME IoT has facilities that are very close to our needs, so the adapter is very thin and has low complexity.

As our product, or the IoT vendor offerings, evolve, we may want to change vendors. If the time comes where we need to replace the ACME IoT or support multiple vendors, the `HydraNetwork` class would have to be evolved.

Imagine some new IoT vendor, BrandX, who could provide significant cost savings. Unfortunately, BrandX does not have a broadcast built in, as ACME does. Instead, each radio has to be addressed individually. The BrandX `HydraNetwork` implementation could implement `broadcast()`, but it would have knowledge of each sensor connection to do so. That layer might become nontrivial, but we could change the vendor without the risk of touching the Hydra core application.

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*.[3]

---

3. [BeckTDD], pp. 136–137.

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.

Not having done concurrent programming in Python before, I thought it would be a good idea to learn how the concurrency and communications mechanisms work before using them. I could simply read the docs and integrate their usage into my production code, or I could experiment and learn from the comfort of test cases.

From looking at the documentation, I decided to test-drive two functions, `put_message()` and `get_message()`, that put and get messages from a queue.

Here are the tests that helped me know for sure that I understood how Python's `Queue` works.

```
class QueueTest(unittest.TestCase):
    def test_get_message_returns_message(self):
    q = Queue(maxsize=1)
    self.assertTrue(put_message(q, 'hey'))
    self.assertEqual('hey', get_message(q, 1.0))

  def test_get_message_None_when_timeout(self):
    q = Queue(maxsize=1)
```

```
    self.assertEqual(None, get_message(q, 0.01))

  def test_get_message_None_empty_no_wait(self):
    q = Queue(maxsize=1)
    self.assertEqual(None, get_message(q, 0.0))

  def test_put_message_True_for_success(self):
    q = Queue(maxsize=1)
    self.assertTrue(put_message(q, 'hey'))
    self.assertEqual('hey', get_message(q, 1.0))

  def test_put_message_False_when_full(self):
    q = Queue(maxsize=1)
    self.assertTrue(put_message(q, 'hey1'))
    self.assertFalse(put_message(q, 'hey2'))
    self.assertEqual('hey1', get_message(q, 1.0))
```

Now that we know how `Queue` functions work, let's use them to communicate with a `Process`. First, we define a function, `worker()`, that will be executed on a separate `Process`. It gets from `in_q`, puts to `out_q`, and then exits. At the time, we expected to use this approach to separate the Hydra core from the `WebServer`.

```
def worker(in_q, out_q):
  message = get_message(in_q, 0.05)

  if message:
    put_message(out_q, message)

class ProcessLearningTest(unittest.TestCase):

  def setUp(self):
    self.command_q = Queue(maxsize=1)
    self.reply_q = Queue(maxsize=1)
    self.process = Process(target=worker,
                           args=(self.command_q,
                           self.reply_q,))
    self.process.daemon = True

  def test_thread_q_message_echos(self):
    put_message(self.command_q, 'Do this')
    self.process.start()
    reply = self.reply_q.get(True, 0.1)
    self.process.join()
```

```
      self.assertEqual('Do this', reply)

    def test_thread_message_timeout(self):
      self.process.start()
      reply = get_message(self.reply_q, 0.1)
      self.process.join()
      self.assertEqual(None, reply))
```

These learning tests cost nothing. We have to learn mechanics 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 that is incompatible with our tests, we will find out right away.

In this case, who expects Python to change? So, there is no future payback. Well, actually, Python did change. While converting the Hydra code from Python 2 to Python 3, the packaging changed; some edits were needed to compile and run the tests. Nicely, our learning tests showed us the behavior was preserved.

Regardless of whether you need the learning provided by the learning tests, a clean boundary should be supported by a set of boundary 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 further 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 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 hardware and the API had not been designed yet. So we would have to work out the details later.

To keep from being blocked, we defined our own interface to the nonexistent hardware. 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 is that it's under our control. This helps keep client code more readable and focused on what it is trying to accomplish. On the client side of the boundary, we are not burdened by the many implementation details. If we had a hardware abstraction layer (HAL), we probably would have started depending on that. HALs usually are not all that abstract from the application perspective. Not knowing the HAL, we had no way to depend on it. Instead of a liability, for now it was an advantage.

In the following diagram, you can see that we insulated the `CommunicationsController` classes from the transmitter API (which was out of our control and currently undefined). By using our own

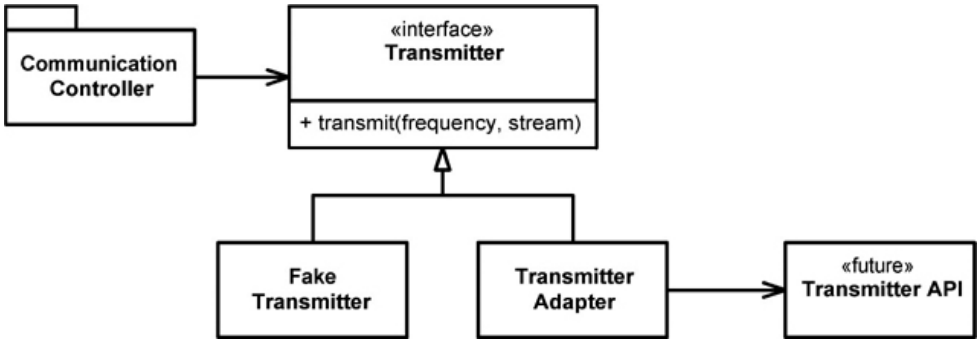
application-specific interface, we kept our `CommunicationsController` code clean and expressive. Once the transmitter API was defined, we wrote the `TransmitterAdapter` to bridge the gap. The Adapter[4] encapsulated the interaction with the API and provides a single place to change when the API evolves.

---

4. See the Adapter pattern in [GOF95].

---

5. See more about seams in [WELC].



This design also gives us a very convenient seam[5] in the code for testing. Using a suitable `FakeTransmitter`, we can test the `CommunicationsController` classes. Once we have the `TransmitterAPI`, we can also create boundary tests 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 third-party dependencies in an ADAPTER to convert from our application-centered 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.