Disciplinary Agency in Test-Driven Design (DRAFT of early March 2004)

Brian Marick (marick@testing.com)

Andrew Pickering's The Mangle of Practice *is about how practice - doing things - causes change to happen in science and technology. He uses "the mangle" to name the way that machines and instruments, scientific facts and theories, goals and plans, skills, social relations, rules of evidence, and so forth all come together and are changed through practice.*

In this paper, I present a detailed case study of the programming practice usually called "test-driven design." I show how Pickering's analysis, particularly his notion of "disciplinary agency," applies well to that practice. However, the flavor of this case study is different than those in his book. Its "dance of agency" gives the lead to disciplinary agency. Disciplinary agency is less source of resistance, more a causal force in modeling and goal setting.

Why the difference? Pickering's book presents an ontology. I suggest that ontologies, too, are mangled in practice.

Setting the scene

> So, I don't start with a story like "The game has Squares." I start with something like: "Player can place a piece on a square."[...]
>
> What I am not doing is worrying about overall game design. [...] [Ideally], I let the design emerge.
>
> -- William Caputo[1]

> Beck has those rules for properly-factored code: 1) runs all the tests, 2) contains no duplication, 3) expresses every idea you want to express, 4) minimal number of classes and methods. When you work with these rules, you pay attention *only* to micro-design matters.
>
> When I used to watch Beck do this, I was *sure* he was really doing macro design "in his head" and just not talking about it, because you can see the design taking shape, but he never seems to be doing anything directed to the design. So I started trying it. What I experience is that I am never doing anything directed to macro design or architecture: just making small changes, removing duplication, improving the expressiveness of little patches of code. Yet the overall design of the system improves. I swear I'm not doing it.
>
> -- Ron Jeffries[2]

[1] testdrivendevelopment Yahoogroups mailing list, March 9 2003.

The world, I want to say, is continually *doing things*, things that bear upon us not as observation statements upon disembodied intellects but as forces upon material beings.

-- Andrew Pickering[3]

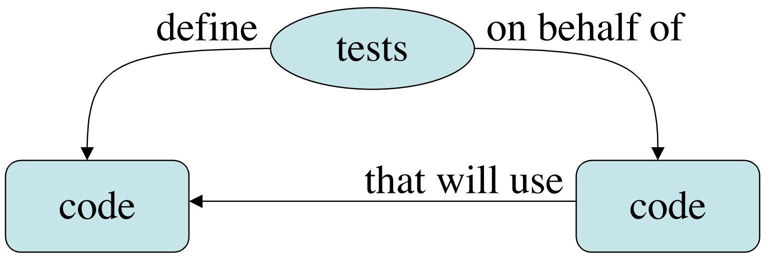
A coding example

On February 8, 2004, I wrote a program on American Airlines flight 505 from Chicago to San Francisco. After I landed, I realized I'd seen a nice example of disciplinary agency. This is my best reconstruction of what I did.

The context is a small sample application I use to demonstrate test-driven design. Test-driven design is a practice of writing a test, watching it fail, writing just enough code (program statements) to make the test pass (while continuing to make all previous tests pass), cleaning up the code without changing its externally-visible behavior, and repeating with the next test.

This example is about cleaning up the code. (In the jargon, that's called *refactoring*.)

Most test-driven design is done with small tests written in a programming language. They're used to help programmers think about the interface to, and behavior of, the code they're writing. The thinking is from the point of view of a later programmer writing yet more code that uses the code now being written.



But it's not "code all the way out". At some point, there's a boundary between the code and the outside world. If one believes in test-driven design, that boundary should also be defined by tests.

In my application, the outermost boundary is defined by tests written as annotated tables. You'll find the one in question on the next page. It describes how a veterinary clinician gives orders during the progress of a medical case in a veterinary clinic. Ignore the first line for now. The lines in bold font represent clinician actions: what a clinician would today write on a clipboard. The lines in italics check that the software gives the right people the right tasks at the right intervals as the result of an order.

---

[2] Agile Manifesto authors' mailing list, July 19 2001.

[3] *The Mangle of Practice* (1995), p. 6.

This particular test is the third one used to build the application. The first two tests define billing behavior: that intensive care costs US$40/day whereas normal board costs US$16, that clients have to pay even if the patient dies, that pathology costs $80, and so on. I mention this because much of the cleaning up involves comparing the code used to pass this last test with the code used to pass previous tests.

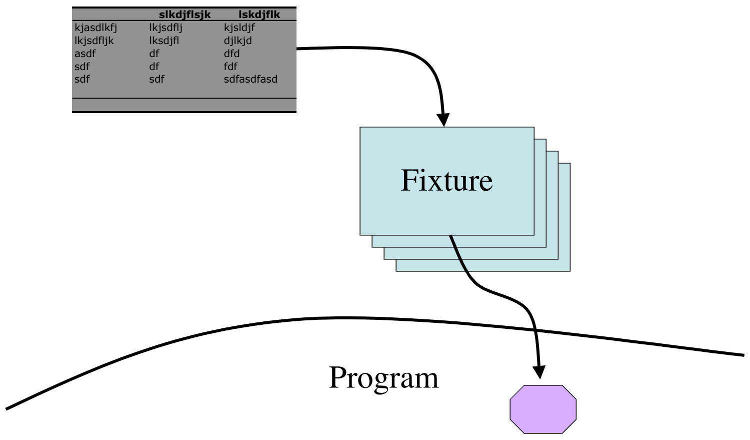
Orders are given to caretakers and students. Some orders depend on whether the animal's in intensive care or not.

AnimalProgressFixture					
new case	**Betsy**	**Rankin**			Rankin brings in a cow
diagnosis	**severe mastitis**				
order	**intensive care**				
order	**soap**				subjective objective assessment and plan
check	*student does*	*soap*	*daily*		
check	*student monitors*	*temperature*	*6 hours*		because in intensive care
check	*caretaker does*	*milking*	*never*		no one milks - milking has to be ordered.
check	*student does*	*milking*	*never*		
order	**milking**				
check	*caretaker does*	*milking*	*12 hours*		
check	*student does*	*milking*	*3 hours*		between student and caretaker, cow gets milked every three hours
diagnosis	**mild mastitis**				now finish treatment
order	**normal board**				
check	*student does*	*soap*	*daily*		
check	*student monitors*	*temperature*	*never*		
check	*student does*	*milking*	*never*		
check	*caretaker does*	*milking*	*12 hours*		

…continued…

order	**discharge**				By the way, discharge should cancel all orders, except that caretakers should keep milking until the owner picks up. And SOAPs are always done.
check	*student does*	*soap*	*daily*		
check	*student monitors*	*temperature*	*never*		
check	*student does*	*milking*	*never*		
check	*caretaker does*	*milking*	*12 hours*		

For this to be a useful test, something has to take the information in the table, convert it into a form a program can understand, deliver it to the program, and compare the results to the expected results found in the table. I'm using a tool called FIT (Cunningham 2002). In that tool, the "something" is called a "fixture." The diagram below shows a fixture sucking information out of a table and sending it into the program.



The octagon represents the piece (or pieces) of the program that will have to change to allow the table's test to pass.

One refinement to this style of programming is to write the needed code in the fixture itself, waiting for something to cause you to move it out into the program. What I hope to demonstrate is that disciplinary agency can make that decision - and that it can make it well, in the sense that the resulting code looks as if it had been the product of conscious intention all along.

Let's begin.

The code and how it was made

My goal in this coding episode was first to make the test pass (such that all the *check* statements become true statements about the behavior of the code), then to make the code clean. Before showing the code, let me describe how it was created. (If you're not familiar with programming, Appendix A might help at this point. It describes how to read the completed code, which should help you read this section about its creation.)

You'll recall that the first line of the test was a mysterious "AnimalProgressFixture." That names the kind of fixture that handles that table. Each line in the table is a step in the test. Most of the lines invoke methods of the fixture. The first cell is the name of the method and the rest are its arguments. FIT takes care of finding the matching method and invoking it. Arguments are converted to match the method's declaration. So *order | normal board* passes the string "*normal board*" to the method *order*, which would look like this:[4]

```
public void order(String order) {
      … whatever it takes to issue the order…
}
```

Lines beginning with "check" are handled differently. Consider *check | student does | soap | daily*. The cell "daily" is the expected result, what is to be checked. The cell "student does" denotes a method that takes a single argument (here, the string "soap"). Since "student does" is not valid Java, it is "camel cased" into "studentDoes", which is. So this line invokes *studentDoes("soap")*, which returns a Java String. That string is compared to "daily". If the comparison fails, the table cell is colored red; if it succeeds, the cell is colored green.

The process of making a test pass begins by running it. The line *check | student does | soap | daily* initially failed because *studentDoes* didn't even exist, much less give the right answer. To fix it, I created the method *studentDoes*, then made it return "daily". That code looked like this:

```
public String studentDoes(String task) {
      return "daily";
}
```

That's sufficient to make that check pass, and canonical test-driven design practice is to do nothing more than that. Later, I had to make this method answer the question "how often does the student do milking?" The simplest way to make that work was to add an *if* statement so that *studentDoes* returns "daily" for the "soap" task, "never" for milking. (Here, and in what follows, I use bold font to draw your attention to what's changed in the new version.)

[4] This behavior is due to a class called StepFixture that I wrote. It's not currently part of the FIT distribution.

```
public String studentDoes(String task) {
   if (task.equals("soap")) {
      return "daily";
   }  else if (task.equals("milking")) {
      return "never";
   }
   return "error";[5]
}
```

However, later rows in the table reveal that students will milk if the animal is in intensive care and the clinician ordered milking. So "never" is sometimes the wrong answer to the question *studentDoes("milking")*. So *studentDoes* had to become more complicated. In particular, it had to have some memory of what the clinician has ordered. The easiest way to do that[6] was to make a variable that holds the student milking frequency, set it with the appropriate values in the appropriate places, and return it in *studentDoes*, which now looks like this:

```
public String studentDoes(String task) {
   if (task.equals("soap")) {
      return "daily";
   } else if (task.equals("milking")) {
      return studentMilkingFrequency;
   }
   return "error";
}
```

This process of adding variables and *if* statements continued until the code passed all the *check* statements.

With that as preface, glance through the following code, looking for ugliness. I've bolded the code that was added to pass this test.

[5] This extra return value is there solely because the Java compiler will whine if I don't put it in. It serves no purpose as far as the tests are concerned.

[6] "Do the simplest thing that could possibly work" is a common enough slogan in test-driven development circles that people use the acronym DTSTTCPW without feeling the need to explain it. A related slogan is "make it work, make it good, make it fast", the last referring to how quickly the program calculates the right answer. "Good" is sometimes taken to mean "easy to revise". Bolder people will forthrightly refer to their own aesthetics (which are taken to be aligned with revisability). For my purposes here, "good" means "obeying disciplinary agency of a certain sort."

```
public class AnimalProgressFixture extends fit.StepFixture {
   private int balance = 0;
   private int dailyCharge = 0;
   private String studentMilkingFrequency = "never";
   private String caretakerMilkingFrequency = "never";
   private String studentTemperatureFrequency = "6 hours";

   public void newCase(String animal, String owner) {
   }

   public void diagnosis(String diagnosis) {
   }

   public void order(String order) {
      maybeUpdateDailyCharge(order);
      maybeUpdateBalance(order);
      if (order.equals("milking")) {
         studentMilkingFrequency = "3 hours";
         caretakerMilkingFrequency = "12 hours";
      } else if (order.equals("normal board")) {
         studentTemperatureFrequency = "never";
         studentMilkingFrequency = "never";
      }
   }

   public void charge(int amount) {
      balance += amount;
   }

   public int balance() {
      return balance;
   }

   public void dayPasses() {
      balance += dailyCharge;
   }

   public void payment(int amount) {
      balance -= amount;
   }
```

… continued…

```
    public void dayPasses() {
        balance += dailyCharge;
    }

    public void payment(int amount) {
        balance -= amount;
    }

    public void dead() {
    }

    public String studentDoes(String task) {
        if (task.equals("soap")) {
            return "daily";
        } else if (task.equals("milking")) {
            return studentMilkingFrequency;
        }
        return "error";
    }

    public String studentMonitors(String measurement) {
        return studentTemperatureFrequency;
    }

    public String caretakerDoes(String task) {
        return caretakerMilkingFrequency;
    }

    ////// Private methods

    private void maybeUpdateDailyCharge(String order) {
        if (order.equals("intensive care")) {
            dailyCharge = 40;
        } else if (order.equals("normal board")) {
            dailyCharge = 16;
        }
    }

    private void maybeUpdateBalance(String order) {
        if (order.equals("pathology")) {
            balance += 80;
        }
    }
}
```

To my mind, this code is a disaster waiting to happen. There are too many variables that keep track of almost the same thing, too many *if* statements scattered about. It's hard to see what's going on.

For example, I think the test is inadequate. We see that the clinician specifically orders SOAPs to be done, and yet the response to *check* | *student does* | *soap* can never be anything but "daily". That's purely an artifact of "doing the simplest thing that could possibly work" in the face of a test that does not ask the question "What if the clinician doesn't order a SOAP?" (Is it really done regardless? If so, why have the explicit order?)

Similarly, the variable *studentTemperatureFrequency* starts out at "6 hours" despite the test annotation that the six-hourly check is because the animal is in intensive care. I could get away with that only because there's no test in which the animal does not start out in intensive care.

To my mind, the proper response to these two oddities is to add one or more new tests that express what should happen in the currently undescribed cases. But the existence of these oddities is obscured by the messiness of the code, so they're easier to overlook than they should be.

It's time to work the code.[7]

[7] The analogy between a potter working clay and a programmer working code is due to Ward Cunningham: <http://www.artima.com/intv/clay3.html>.

Refactoring

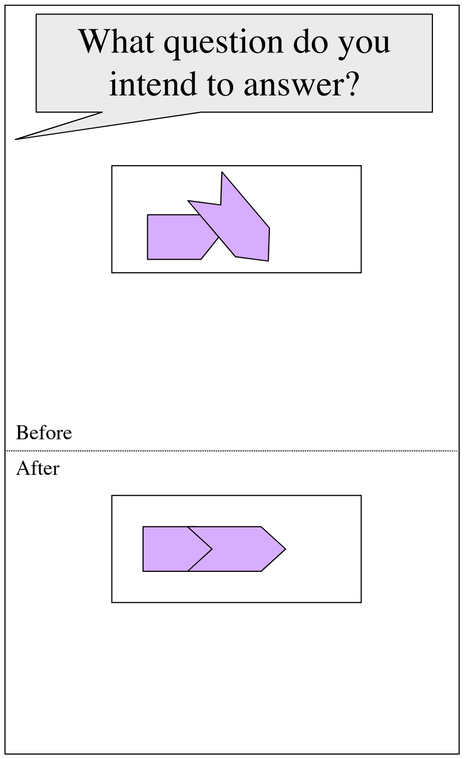
I scanned the code, looking for violations of my internal rules for well-formed code, rules quite similar to those described by Jeffries in the second scene-setting quote. The first thing I happened to see was this:

```
public String caretakerDoes(String task) {
   return caretakerMilkingFrequency;
}
```

Does the body of this code express the same thing as the declaration? No. The declaration says this code answers a question about a task, but the body answers the question only about the task of milking. My discipline tells me that the code should be changed to bring the two into alignment.

But it doesn't tell me how. I cannot change the method name to *caretakerMilkingFrequency()* because the *caretakerDoes* was chosen by the Customer to be meaningful to her[8]. I could change the body to something like this:

```
public String caretakerDoes(String task) {
   return caretakerFrequency(task);
}
```



But that just moves the problem somewhere else (to the body of *CaretakerFrequency).* No, the problem is that the intent of the test (as expressed in the table and resultant method declaration) is that milking is but one of many possible caretaker tasks. So the body of the method should echo this intent. A simple solution is to replace the special-purpose variable *caretakerMilkingFrequency* with an object that uses task names to find frequencies. (I think of this as moving *Milking* out of the name.)

[8] "Customer" is a term of art in Extreme Programming (Beck XX). The Customer describes to the programmers what the business needs. In this case, part of the description was through tests. As is always the case, much of the description is conversational. It's important that the programmers not impose the happenstances of their implementation on the Customer's vocabulary. It is the case, though, that the joint vocabulary of the whole team (Customer and programmers) becomes mangled (in Pickering's sense) during the project. That would be an interesting case study in its own right.

The Customer in this scenario is my wife, who is in fact a clinician at a veterinary clinic.

Java provides such an object in the form of a HashMap. I can replace:

```
        private String caretakerMilkingFrequency = "never";
```
with:

```
        private HashMap caretakerFrequency = new HashMap();
        caretakerFrequency.put("milking", "never");
```

Where I earlier wrote this:

```
        caretakerMilkingFrequency = "12 hours";
```

I can write:

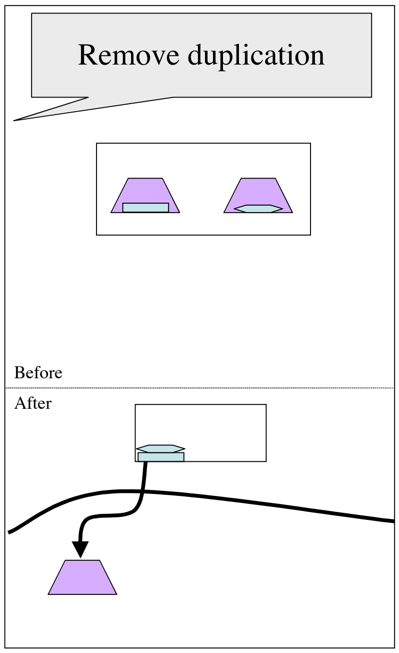
```
        caretakerFrequency.put("milking", "12 hours");
```

And now *caretakerDoes* expresses intention better:

```
        public String caretakerDoes(String task) {
           return (String) caretakerFrequency.get(task);
        }
```

I can do the same thing with the similar-looking method *studentMonitors*:

```
        public String studentMonitors(String measurement) {
           return (String) studentFrequency.get(measurement);
        }
```



But at this point disciplinary agency steps forward to say "Remove duplication!" The bodies of my new versions of *caretakerDoes* and *studentMonitors* look too similar. The difference between them is who's doing the task - caretaker or student - but that difference is obscured by their similar appearance.[9]

Again, I have a choice about how to remove duplication. I would prefer to see this:

```
   public String studentMonitors(String measurement) {
      return frequency.of("student", measurement);
   }

   public String caretakerDoes(String task) {
      return frequency.of("caretaker", task);
   }
```

[9] I have to confess it's the duplication of the *(String)* that grates the most. Better programming languages, like Ruby, would not have that. Would I then have been so quick to remove duplication? How would the end result have changed?

I like the way that reads - almost English. I also like it because there's another routine that can be changed in the same way. This:

```
public String studentDoes(String task) {
   if (task.equals("soap")) {
      return "daily";
   } else if (task.equals("milking")) {
      return studentMilkingFrequency;
   }
   return "error";
}
```

can become this:

```
public String studentDoes(String task) {
   return frequency.of("student", task);
}
```

That gets rid of *if* logic, which my discipline tells me to do. It also gets rid of the "error" return that was an annoying imposition of a know-it-all compiler. Finally, I also have a good excuse to set the SOAP frequency in the same place as everything else, instead of handling it specially inside the body of *studentDoes*.

In short, I move all the initialization into one place:[10]

```
public AnimalProgressFixture() {
   frequency.set("caretaker", "milking", "never");
   frequency.set("student", "soap", "daily");
   frequency.set("student", "temperature", "6 hours");
   frequency.set("student", "milking", "never");
}
```

And the only other mention of specific frequencies (like "daily") is when they're assigned to a task as the result of an order:

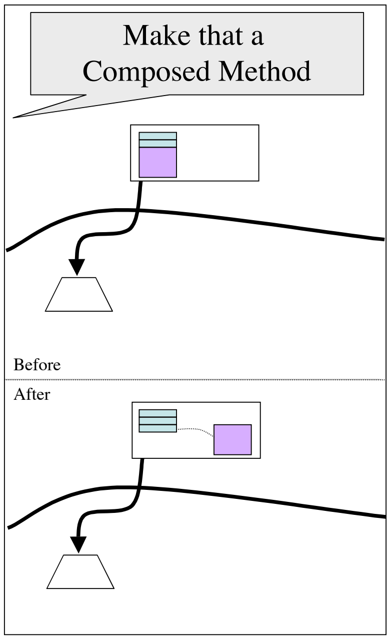
```
public void order(String order) {
   maybeUpdateDailyCharge(order);
   maybeUpdateBalance(order);
   if (order.equals("milking")) {
      frequency.set("student", "milking", "3 hours");
      frequency.set("caretaker", "milking", "12 hours");
   } else if (order.equals("normal board")) {
      frequency.set("student", "temperature", "never");
      frequency.set("student", "milking", "never");
   }
}
```

[10] This method is called a *constructor*. It's run automatically when an AnimalProgressFixture is created. So it serves to initialize the frequencies before the test runs. There was no constructor in the original code because there was nothing to initialize.

Although it was not my goal to localize behavior in a constructor and the *order* method, it was the result, and it's a result I'm (provisionally) pleased with. It's nice when following disciplinary agency's directives lead to happy results I wasn't aiming for. As we work through this example, you'll see more of that.

A couple of other things have happened. The methods *studentMonitors* and *studentDoes* used to look rather different. Now they look identical save for names ("monitors", "does"; "measurement", "task") that are synonyms as far as the code is concerned. The customer uses different terminology in different cases but has yet to give the code a reason to distinguish them. The form of the code now emphasizes that.

The object *frequency* is nothing that Java provides. I had to create a new class, named Frequency. (I'm not showing the code. Unlikely as it may seem, I'm trying to be brief. The code is straightforward.) The need to eliminate duplication has started pushing code out of the fixture and into the program proper.

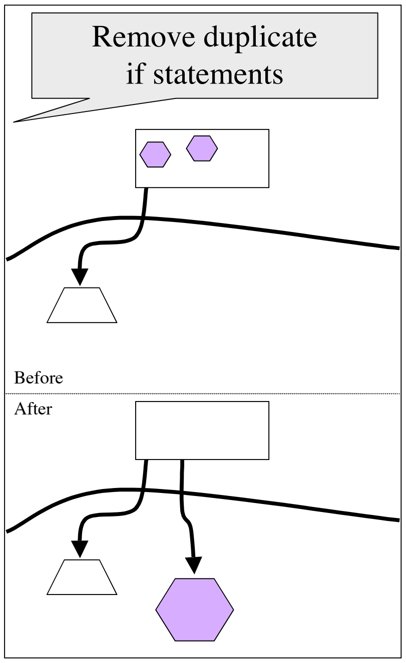


Having completed that, disciplinary agency immediately takes over again. Look at the *order* method (just shown). It's at two levels of detail. In the first two lines, the code is saying things like "maybe update the daily charge appropriately…"; in the rest, it's saying "if, specifically, the order is 'milking', set the frequency to three hours…".

Methods that are all at the same level of detail are called "Composed Methods" (Beck XX). My discipline tells me to make composed methods. That's simple to do by extracting a new method:

```
public void order(String order) {
   maybeUpdateDailyCharge(order);
   maybeUpdateBalance(order);
   maybeUpdateFrequency(order);
}

private void maybeUpdateFrequency(String order) {
   if (order.equals("milking")) {
      frequency.set("student", "milking", "3 hours");
      frequency.set("caretaker", "milking", "12 hours");
   } else if (order.equals("normal board")) {
      frequency.set("student", "temperature", "never");
      frequency.set("student", "milking", "never");
   }
}
```

That's incredibly simple to do. Given my programming tools, it's a matter of highlighting some code, typing ctrl-apple-M, then typing the new method's name. (More about tools later.) But doing so focused my attention once more on *if* statements. Both the code in the new *maybeUpdateFrequency* and the older *maybeUpdateDailyCharge* check whether an order is "normal board":



```
public void maybeUpdateDailyCharge(String order) {
   if (order.equals("intensive care")) {
      dailyCharge = 40;
   } else if (order.equals("normal board")) {
      dailyCharge = 16;
   }
}
```

My discipline tells me it should only be checked in one place. I should find some way to consolidate these two *if*s. A way that comes to mind is to turn an order from a String into an object that knows how to set frequencies. I'll start by dealing with the *if* in *maybeUpdateFrequencies*. Once I've made my change, the *order* method will look like this:

```
public void order(String orderName) {
   Order order = Order.from(orderName);
   order.maybeUpdateFrequency(frequency);
   maybeUpdateDailyCharge(orderName);
   maybeUpdateBalance(orderName);
}
```

where different kinds of Orders know how to do appropriate frequency updates. For example, an order to milk causes this method to be chosen:[11]

```
static class MilkingOrder extends Order {
   public void maybeUpdateFrequency(Frequency frequency) {
      frequency.set("student", "milking", "3 hours");
      frequency.set("caretaker", "milking", "12 hours");
   }
}
```

The *if* statement seems to have vanished, but it's merely moved into Order's *from* method, which knows which kind of Order to create:

```
public static Order from(String orderName) {
   if (orderName.equals("milking")) {
      return new MilkingOrder();
   } else …
```

[11] Appendix A describes in more detail how the code works.

I still have two *if* statements, one in Order's *from* and one in AnimalProgressFixture's *maybeUpdateDailyCharge*. The next step is to merge the latter into the former by moving the charge-manipulation code into the appropriate kind of Order. The fact that the daily charge for intensive care is US$40 moves into IntensiveCareOrder:

```
  static class IntensiveCareOrder extends Order {
    public void maybeUpdateDailyCharge(
                                AnimalProgressFixture animalProgressFixture) {
      animalProgressFixture.dailyCharge = 40;
    }
  }
```

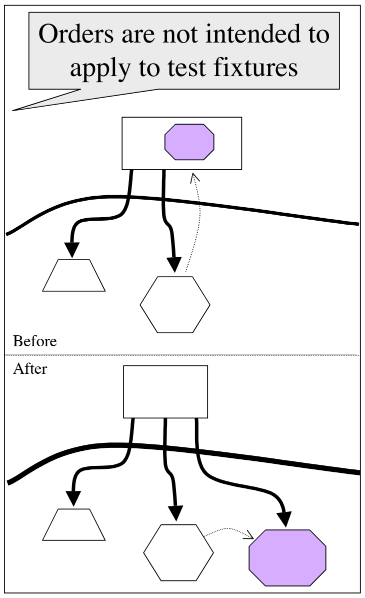
The fact that normal board is US$16 moves into the NormalBoardOrder:

```
static class NormalOrder extends Order {
    public void maybeUpdateDailyCharge(
                                AnimalProgressFixture animalProgressFixture) {
      animalProgressFixture.dailyCharge = 16;
    }
  }
```

AnimalProgressFixture's *maybeUpdateDailyBoard* no longer does anything, so it - and its *if* statement - can be removed. There is now only one check anywhere for the string "normal board" (in Order's *from* method).

At this point, *order* looks like the following. (I have yet to change *maybeUpdateBalance* in a way parallel to *maybeUpdateDailyCharge*.)

```
        public void order(String orderName) {
            Order order = Order.from(orderName);
            order.maybeUpdateFrequency(frequency);
            order.maybeUpdateDailyCharge();
            maybeUpdateBalance(orderName);
          }
```



But once again the discipline of intention-revealing code comes into play. Notice that the *maybeUpdateDailyCharge* methods refer to an AnimalProgressFixture. There's no sense in which anyone - customer or programmer - intends Orders to apply to AnimalProgressFixtures, which are nothing but glue between the tests and the code that really solves the Customer's problems. Instead, orders should apply to something else… a definite thing with its own purpose… something that needs its own name… How about a Bill?

I move everything about balances and payments into a new Bill class:

```
public class Bill {
   private int balance = 0;
   private long dailyCharge = 0;

   public void charge(int amount) {
      balance += amount;
   }

   public int balance() {
      return balance;
   }

   public void setDailyCharge(int dailyCharge) {
      this.dailyCharge = dailyCharge;
   }

   public void applyDailyCharge() {
      balance += dailyCharge;
   }

   public void payment(int amount) {
      balance -= amount;
   }
}
```

These familiar methods are now grouped together into an object that is, pleasingly, only about one thing: money. Now the AnimalProgressFixture's *charge* method can just ask the Bill for information:

```
public void charge(int amount) {
   bill.charge(amount);
}
```

or create Orders that update the bill and the frequency:

```
public void order(String orderName) {
   Order order = Order.from(orderName);
   order.maybeUpdateFrequency(frequency);
   order.maybeUpdateBill(bill);[12]
}
```

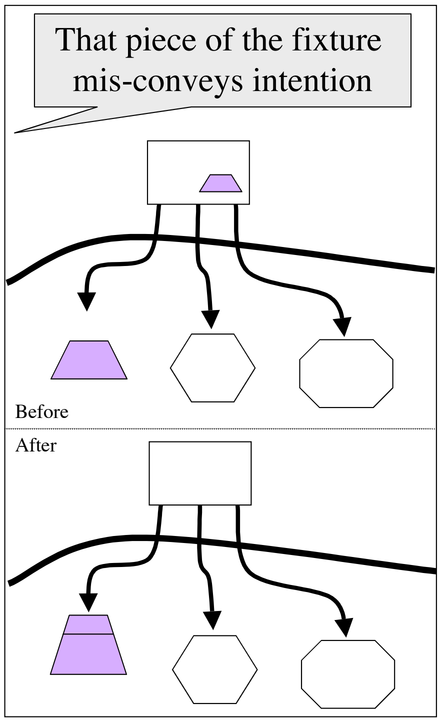
The fixture is now not doing much - it's starting to look more like "glue" and less like program logic.

```
public class AnimalProgressFixture extends fit.StepFixture {
   private Frequency frequency = new Frequency();
   private Bill bill = new Bill();

   public AnimalProgressFixture() {
      frequency.set("caretaker", "milking", "never");
      frequency.set("student", "soap", "daily");
      frequency.set("student", "temperature", "6 hours");
      frequency.set("student", "milking", "never");
   }

   public void newCase(String animal, String owner) {
   }

   public void diagnosis(String diagnosis) {
   }

   public void order(String orderName) {
      Order order = Order.from(orderName);
      order.maybeUpdateFrequency(frequency);
      order.maybeUpdateBill(bill);
   }

   public void charge(int amount) {
      bill.charge(amount);
   }

   public int balance() {
      return bill.balance();
   }
```

…continued…

[12] The old *order* updated the balance and the daily charge separately. But there's no reason for the fixture to care - or even know about - what orders do to bills, so I merged the two into a single *maybeUpdateBill*.

```
    public void dayPasses() {
        bill.applyDailyCharge();
    }

    public void payment(int amount) {
        bill.payment(amount);
    }

    public void dead() {
    }

    public String studentDoes(String task) {
        return frequency.of("student", task);
    }

    public String studentMonitors(String measurement) {
        return frequency.of("student", measurement);
    }

    public String caretakerDoes(String task) {
        return frequency.of("caretaker", task);
    }
}
```



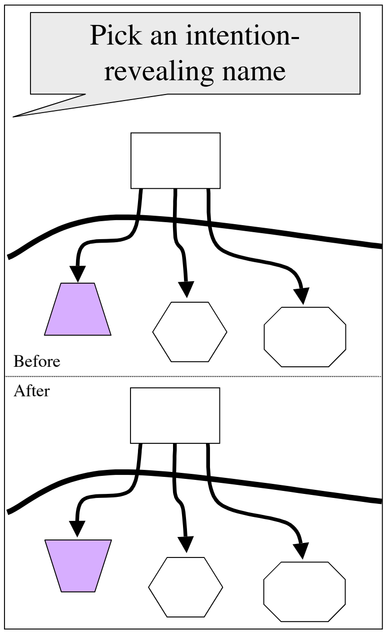
But the constructor at the top is annoying. Why should setting up an AnimalProgressFixture be all about *frequency*? And it is also suspicious that it's the longest routine. (The longest routine is always suspicious.)

That code should be moved into the Frequency class:

```
class Frequency {
    public Frequency() {
        …
        set("caretaker", "milking", "never");
        set("student", "soap", "daily");
        set("student", "temperature", "6 hours");
        set("student", "milking", "never");
    }
```

Notice how the "6 hours" stands out. Something is probably wrong here - it's more likely that the starting "temperature" should be "never". Another test is needed. (Recall that I noticed this in the original fixture, but I claim that it's more noticeable now.)

Does anything else jump out?



The name "Frequency" grates a bit. If I were given to personifying things, I'd say that Mr. Intention Revealing is sleeping restlessly. Something's wrong here. The word "frequency" isn't one that I've heard the customer use. There's a mismatch between the world outside the code and the world inside the code. I would prefer, along with Evans (XXX), that the two be tuned to each other.

What word would work better? One that comes to mind is "Schedule". The Frequency object holds the schedule of tasks for all the people who do tasks. So I rename Frequency and change *frequency.of(person, task)* to *schedule.frequencyOf(person, task)*. I'm still a little uneasy about the name, but it's better, and I know I can easily change it later.

Thinking about the Story

Having told the story, I want to answer two questions. First, is this a "manglish" story, one that fits Pickering's model? Unsurprisingly, I think it is. Second, is this a *boring* manglish story, or does it deviate in any interesting ways from Pickering's case studies?

On the latter question, I'll let you be the judge.

The mangle of refactoring practice

Certain key ideas recur in Pickering's stories. Do they occur here?

Is the story **open ended**? Yes. I did not predict at the beginning what the program would look like at the end. Or, to be more accurate, I predicted that it would look like the last time I made these tests pass, but I was wrong. I didn't anticipate the Schedule class. Instead, I expected to have Students and Caretakers who knew their own schedule. I've written a program that passed these tests before, and I did end up with a Student and a Caretaker class, and I saw no reason why that history wouldn't repeat itself.

Open-endedness is a typical feature of test-driven design, as the quotes that opened this paper suggest. We practitioners expect the final shape of the program to surprise us. Our literature is fond of the noun "emergence". We tell each other stories of how refactoring sessions lead to new classes with unforeseen capabilities.[13]

Similarly, there is a strong element of **contingency**. Something trivial caused me to end up in a different place in this programming episode than when I implemented the exact same tests once before. Was it that I fixed messiness in a different order this time? Was it

[13] For a legendary story, see http://c2.com/cgi/wiki?WhatIsAnAdvancer

that I was marginally more disciplined? Was it increased cosmic ray flux at 35,000 feet? Who knows?

Is there a **dance of agency** and a **decentering of the human subject**? Check. The story is one of alternation. My discipline told me of a problem, then fell silent (mostly) to let me fix it. The pictures I use to adorn the story capture the feeling that disciplinary agency is speaking to me to set a direction for my next application of human agency. Since, in this dance, it's very much the discipline that takes the lead, I think it fair to say the human subject (me) is not at the center of the story.

Is there **tuning**? Yes, though the story only shows a hint of it. The hint is in the naming. "Schedule" is a new idea. I asked the customer, and she has no notion of a schedule as a distinct thing that keeps track of everyone's tasks. Instead, her model of a clinic has independent agents - students and caretakers - who are assigned tasks and are thereafter responsible for carrying them out.

If the story of this program going forward is typical, there will be more tuning coming up: the language the customer uses to talk about the problems the program is to solve will tune itself to concepts embodied as names in the program, while the program's names will continue to adjust to the customer's model of the world. There will likely come a point of **interactive stabilization**, though that point will never be fixed forever. For the moment, though, the only point of interactive stabilization is that disciplinary agency has nothing more to say about how this version of the program should be changed.

Because all of these Pickeringesque words work for this story, I declare it a manglish story.

Differences

In the four case studies that make up the bulk of *The Mangle of Practice*, human and nonhuman agency have distinct jobs. For the most part, nonhuman agency *resists* and human agency *accommodates*.[14] Nonhuman agency doesn't make much of an effort to help out.

In my case, it felt as if nonhuman agency *were* helping out. At each point at which I finished a task, I asked my discipline what to do next, and it told me.

Before going into that further, am I in fact talking about disciplinary agency in Pickering's sense? It seems to me I am. Pickering talks of disciplinary agency as a matter of "particular routinized ways of connecting marks and symbols with one another" (p. 115). That's what my story is about. Someone familiar with my discipline can look at "before" and "after" snapshots of the code and describe the connections in routine ways. "Extract Method was used to convert the method into a Composed Method" is not a

[14] I'm sure the careful reader of *Mangle* - hi, Andy! - can point to cases where each does a different job - indeed, the phrase "dance of agency" suggests a parity that "dialectic of resistance and accommodation" does not, but I maintain the overall tendency is as I've described. I expect that's because of the particular episodes Pickering described. The quaternion story differs from mine because Hamilton was making a breakthrough whereas I'm just doing some mundane programming. The free quark story differs because Murpurgo was trying to make a machine do something never done before, whereas I'm making Java do the kinds of things it's often used for.

different kind of statement than "he multiplied the two equations and used the associative law".

But Pickering also says that disciplinary agency "leads us through a series of manipulations within an established conceptual system" (p. 115), and that is not quite the case here. In his story of Hamilton's discovery of quaternions - the story that brings disciplinary agency to the fore - Pickering talks of new conceptual structures being modeled on old ones. The modeling is divided into three iterated phases: *bridging*, a free human move that sends exploration off in a particular direction; *transcription*, a forced exercise of habits or routines appropriate to the new starting point, often uncovering resistance (the bridging extension doesn't work); and *filling*, a free human move to answer questions unanswered by either the original bridging extension or the transcription moves.

In my story, disciplinary agency doesn't lead: it points. It does the bridging for me: "Make the next version like this one, but without this particular duplication."[15] Now I, the human, do a bit of filling. Which of several possible ways of eliminating the duplication do I choose? Having done that, I move to transcription, rote work. For example, having decided that I will remove a duplicate *if* by creating an Order class and its various specific subclasses (IntensiveCareOrder and the like), the path from the current code to the next version was one of applying a set of standard moves well-described in the literature (see Fowler XX, Wake XX, and especially Kerievsky XX).

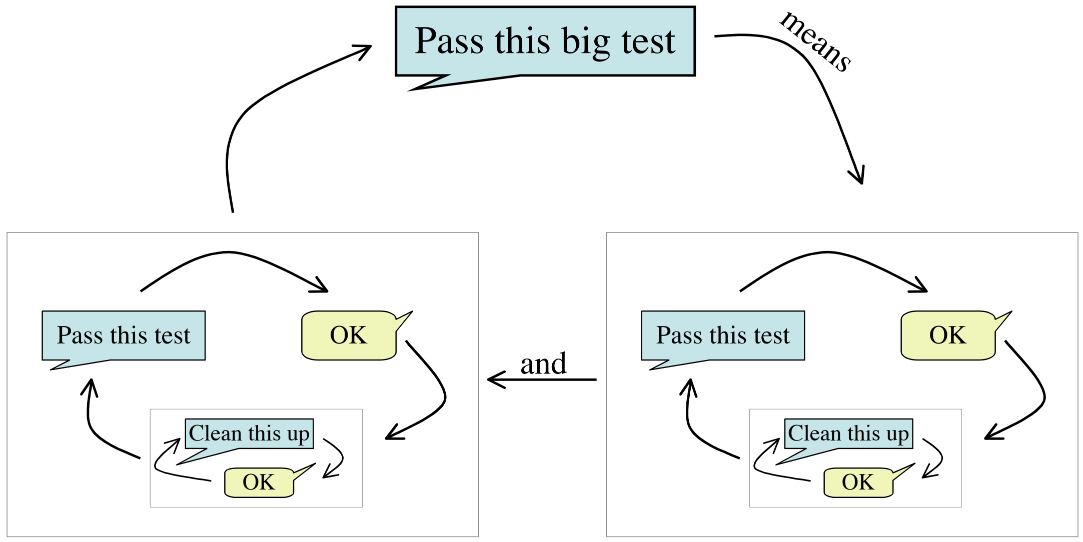
I want to call the disciplinary agency that points *impulsive agency*. I chose the name because "impulse" has the connotation of giving something a push, which is what the agency did to me, but also because "impulsive" suggests a liveness and even whimsicality that is appropriate to this context.

I single out impulsive agency for a name because it appears repeatedly in test-driven programming. Programmers are continually ceding responsibility that is traditionally their own to some outside agency.

For example, it's traditionally a programmer's responsibility to consider carefully the totality of the requirements for a task, think carefully about a design that would satisfy those requirements, then write code that implements the design. In test-driven design, though, the programmer gives up the desire for totality. Instead, a series of tests is processed, one at a time, in a tight iterative loop. First the test is presented, then the code is tweaked to make it pass, then the code is cleaned up. Repeat ad infinitum and watch the design emerge. The tests serve as impulses for programmer action, and the programmer trusts that discipline will produce a result as good as the more thoughtful approach.

[15] I could choose which of several possible bridging moves to start with. I could ponder whether I should first remove *this* duplication or, instead, make *that* method more intention-revealing. In practice, I don't consciously do that - I simply take the first one that catches my attention. I also have some choice in considering whether some particular similarity between two methods counts as duplication or whether a particular chunk of code reveals intention well enough. And, indeed, different programmers can make different judgments on such matters. but this seems to me more a matter of differences in their disciplines than a matter of choice: once I've perceived a duplication, I *must* seek to remove it. So I contend these human choices are "in the noise" - the real action is in the rules of the discipline.

Impulsive agency acts at several levels.



At the lowest level - the level of my story - it acts to direct refactoring. At the next level, as just described, it provides tests that goad the programmer into writing code. This is not non-human agency in Pickering's sense, since a human (typically the programmer herself) creates each test to move the program one step closer to a goal. Nevertheless the tests are impulsive in my sense. Those tests, in turn, are derived from some test of larger scope, typically created through conversation between a programmer and a customer.[16] Given the mandate to satisfy that test, the programmer decides which pieces of the program to change, then acquires or creates lower-level tests to direct those changes.[17]

Before discussing this in more detail, let me say something about what it means for the programmer to say "OK" in response to an impulse. In the text describing my refactorings, I pretended I had completely free choice to respond to discipline's direction. That's misleading.

For example, consider my decision to create an Order class to remove duplicate *if* statements. I could (perhaps) have come up with a different way, but why bother? This is a classic use of subclassing and runtime polymorphism. It's what languages like Java are *for*. Moreover, as a consequence of making that change, I isolated a bunch of similar *if* statements into one of Order's methods:

[16] These top-level tests are less common than the lower-level tests. Often, the direction to the programmer is given solely in the form of conversation with the customer, rather than through conversation+test. Nevertheless, the conversation serves as an impulse: a pushing of the programmer in a direction, without any commitment that the place she comes to rest is the final, predictable-in-advance goal.

[17] My description of the creation of the original code to clean up didn't look like the picture on this page. That's because the program is simple enough that the Big Test could be understood as direct instructions to change what - at that point - was a small amount of code. I didn't need to divide the problem of passing that test into smaller problems, each with its own test or tests.

```
public class Order {
   public static Order from(String orderName) {
      if (orderName.equals("milking")) {
         return new MilkingOrder();
      } else if (orderName.equals("intensive care")) {
         return new IntensiveCareOrder();
      } else if (orderName.equals("normal board")) {
         return new NormalBoardOrder();
      } else if (orderName.equals("pathology")) {
         return new PathologyOrder();
      } else {
         return new Order();
      }
   }
```

This method represents no innovation; it's a standard way of dealing with a standard problem. It even has its own name: Factory Method (Gamma et al 1994). Quite a lot of what I and other programmers do is adapt standard solutions to recurrent problems. In the jargon, we "apply patterns".[18]

Now that the fad for chaos theory has passed, I feel safe in using the word "attractor" for this effect. So I claim that this style of programming consists of being given a push in a particular direction by impulsive agency, applying free choice in the details of moving in that direction, but having that free choice frequently captured by attractors.[19] Once the impulse has been satisfied, the programmer turns to outside (in some sense) agency for the next impulse, all the while serenely confident that the end result will be something that is fit for its purpose and internally coherent and aesthetically pleasing.

I'm most interested in that serene confidence - whence comes it? - but let me first finish discussing this particular kind of dance of agency. I've presented the most extremely decentered picture of programming. In my rhetoric, the human is not doing much at all: just being pushed and attracted, ad infinitum. There is lively debate among programmers about where the center should lie and how much humans actually do. Ron Jeffries is one of the foremost proponents of a decentered view. Yet, as I quoted him at the beginning of this paper, he at first "was *sure* [Beck] was really doing macro design 'in his head' and just not talking about it." There are people today equally sure that Jeffries is

[18] The term "pattern" is due to the architect Christopher Alexander, who has had an enormous effect on the practice of programming. An inadequate definition of a pattern is "a solution to a recurrent problem in a context". In Alexander's formulation, the context is the result of applying earlier, "higher-level," patterns. So, for example, Alexander (XX) will begin designing a house by using a pattern that helps decide where a house should stand on a particular plot of land. The next pattern would determine the orientation of the house. This can continue on through the layout of the rooms, decisions about how to shape the outside space, where windows should be placed, and so on - all the way down to the width of the trim. For the most part, the patterns guide free choice. (Although Alexander is pretty dogmatic about the width of the trim.)

[19] An attractor may also figure in Hamilton's story. Pickering mentions in a footnote <find this> that Hamilton had previously toyed with non-commutative algebras, so I could argue that his background knowledge of what such algebras do was an attractor when he was dealing with a problem posed by transcription.

unconsciously doing design, that it is *impossible* for someone with roughly forty years of programming experience *not* to be thinking forward and guiding the design. And there are certainly many programmers who explicitly move the center toward themselves: they do more upfront thinking about design, they plan ahead.

Nevertheless, I think the majority of test-driven programmers do less up-front design than they used to, are more willing to accept the surprising emergence of structure, and would perhaps even agree that there's an alternation between their "doing" and the "doing" of something else. So I believe my particular variant dance of agency fits both Pickering's model and test-first programming practice.

The interesting question to me is why we test-driven programmers think this works. My answer is that it is *made* to work by constructing an ontology in which the programming world consists of helpful attractors and impulsive agency whose whimsicality doesn't often push you down paths from which you can't readily recover. It is a world without scary resistance, one in which software is actually *soft* - malleable, shapeable, workable. This world has some precedent. Consider the worldview that Cornel West attributes to Ralph Waldo Emerson in his (West's) *The American Evasion of Philosophy*.

1. Emerson held that "the basic nature of things, the fundamental way the world is, is itself incomplete and in flux" (p. 15). Moreover, the world and humans are bound up together: the world is the result of the work of people, and it actively solicits "the experimental makings, workings, and doings of human beings" (p. 15).
2. Emerson believed that this basic nature makes the world joyous. It gives people an opportunity to exercise their native powers with success, because the world is fundamentally supportive of human striving.
3. And finally, Emerson believed that human powers haven't yet been fully unleashed, but they could be through the "genius of individuals willing to rely on and trust themselves" (p. 16).

The test-first programmer ontology, I think, shares points (1) and (2). It differs from point (3) in being less of a cult of individuality. The ontology locates genius in assemblages of tools, techniques, disciplines, and teams more than in individuals. It is, I claim, the construction of those assemblages that has allowed programmers like me to believe in (1) and (2), which are *very* far from the normal "software engineering" ontology.

In that ontology, the world is corrupted by entropy. Software will inevitably decay into what Foote and Yoder (XX) call a "big ball of mud," a tangled mass of unmaintainable code in which each change makes future changes incrementally harder. In this ontology, the sensible response is to fight against entropy as long as possible. The main weapon is control of change. Your goal is to understand everything the software will have to do at the very beginning, so that you build it right the first time and then never change it. Since that goal is impossible, the secondary goal is to anticipate the kind of changes that will happen, then build the software to accommodate those. Since that's *also* impossible - the world has a way of coming up with changes you didn't foresee - the tertiary goal is to resist changes ("can't be done, nope") as long as possible and, when overruled, make them as small and localized as possible.

It's not such a cheery worldview. It contrasts with what's now called the "Agile" worldview, which counsels "embracing change" (the subtitle of Beck XX) in the expectation that change will - if properly managed - lead to programs that are ever more capable of supporting change.

So how did this ontology emerge? A good answer awaits a longer study, but my tentative speculation is that it began as an accommodation to a major episode of resistance. The accommodation led to an increasingly successful practice. The increasing success cast increasing doubt on the ontology of inevitable decay, which became both tacitly and explicitly replaced by the new ontology. That new ontology feeds back into the mangle of supporting practices and tools, bringing to bear new conceptual resources (like the rhetoric of emergence in Complex Adaptive Systems (Schwaber XX, Highsmith XX) and the theorizings of scholars like Pickering.

Let's start with Smalltalk.

Smalltalk is both a programming language and a programming environment. That is, it's a large program used to write Smalltalk programs, with the twist that it's a program that's itself written in Smalltalk. Any Smalltalk programmer has direct access to (almost) all of the Smalltalk code used to make up the environment itself. That allows many powerful and flexible things.

The Smalltalk environment is quite old as programs go. I have on my bookshelf two volumes describing the 1980 version, and it was already quite sophisticated then. The sophistication is of a certain form. Like certain other languages, Smalltalk is an exercise in pushing certain simple and elegant ideas about structuring programs as far as they can go. Over the years, its implementers found they could go quite far indeed. As they continually went further, found more power in the ideas, they revised the environment itself to demonstrate and use those ideas. Perhaps because the environment was a flagship demonstration of the ideas, perhaps because the implementers were researchers without time pressure, perhaps because they were really very good - for whatever reason, Smalltalk avoided the entropy that plagues programs.

The Smalltalk environment was also characterized by support for a highly interactive and exploratory programming style. Smalltalk programmers were used to writing small amounts of code and getting almost instant feedback. Their tools gave them very sophisticated tools for understanding what was happening with their programs.

The future looked bright for Smalltalk. Its devotees believed that it was obviously superior to the alternatives and soon vast numbers of people would be programming in it. But, for various reasons still disputed today, Smalltalk failed in the market.

But there was one exception: financial services. Smalltalk started to be used for things like internal support programs in insurance companies (such as programs to calculate values for new types of policies), and payroll programs, and the like.

That's a significant change. At that time, internal "IT" jobs were decidedly *not* the glamour jobs for programmers. Graduates from the elite or even semi-elite schools looked down on such jobs, and they were staffed by lesser programmers (at least according to the standards of the elite schools).

And yet the Knights of the Square Bracket[20] *needed* to keep programming in Smalltalk, so the many of them joined the IT world. That's an interestingly different one from Smalltalk's original world:

- There is a constant flood of externally-generated change requests that have to be finished so fast that there seems to be no time for the loving crafting of internal structure that Smalltalk was known for.
- The programs have to cope with (implement) business rules that make no ahistorical sense. Most business's rules are a mass of special cases. One rule exists because some salesperson twenty years ago was only able to close a deal by promising a change to one little bit of the standard policy. Another rule exists because it was inherited from a small company acquired fifteen years ago. And so on. These special cases push away from the sort of elegant and concise programs that elite programmers are taught to value. The result is very different from taking one powerful and simple idea and seeing how far it could go. The Knights moved from what might (oversimplistically) be called a world of hedgehogs to what is definitely a world of foxes.[21]
- The people doing the programming are not elite programmers and have less of a drive to construct elegant and concise solutions.

Bam! Resistance.

I interpret what happened next in two ways. First, the Knights tried to accommodate the resistance by finding techniques to allow them to write their familiar style of program in this radically different environment. Second, they turned part of their attention away from elegant, powerful, and emergent program structures to elegant, powerful, and emergent *human* structures. As a consequence, there's a historical trajectory toward a collection of techniques (like test-driven design) that decenter human planning in favor of a fluid reaction to the flow of discipline and impulses from "outside".

I know of no short way to talk about the resulting practices, their relationship to where the Knights started, and their relationship to the IT environment. So let me pass over that with only the claim that it seems manglish to me. Instead, I'll talk about why I believe it's fair to talk about constructing an ontology.

The first key thing is that the practices *worked*. They produced decent software, at least as good as that produced by conventional techniques. More significantly, they produced enthusiastic programmers. The practices were more aligned to their goals: joyful work, craftsmanship, a feeling of steady progress.

[20] People who will do almost anything to keep programming in Smalltalk. So-called because Smalltalk is unusual among programming languages for using square brackets for grouping. See http://wiki.cs.uiuc.edu/VisualWorks/The+%22I'll+give+up+Smalltalk+when+they+pry+my+cold+fingers+from+the+browser%22+club+

[21] From Isaiah Berlin (XX): "The fox knows many things, but the hedgehog knows one big thing." Berlin was writing of authors. Hedgehogs have one overarching principle they can apply universally; foxes have a myriad of truths that they apply situationally.

The second key thing is that the practices *kept working better*. The techniques are typically easy to start with and produced fairly immediate gratification. They come with free tools that support them. The tool industry (both free and expensive) has been tuning tools to the practices, making them (and the programmers that use them) more successful. Trickier problems (how do you refactor database schema? how do you develop graphical user interfaces test-first?) are being chipped away at, and the results are being published both informally and formally. In the terminology of Lakatos (XX), we have here a progressive research programme. That is, the world of problems seems to be giving way to a "hard core" of postulates.[22]

In such a situation, one where formerly knotty tasks keep getting easier, and the hard core is continually throwing up what Lakatos calls novel confirmations, *and* the hard core represents a radical shift from prior practice and belief (in this case, the shift away from the individual and conceptual toward the social (team-oriented) and material[23]), it is natural to think you've stumbled on a new discovery about the way the world is: an ontology like Emerson's, but with the self-reliant individual replaced by an assemblage of people, disciplines, and tools.

I'm a partisan, so I find this a hopeful sign where some see a retreat from rationality.[24] This new ontology will make progress more likely: if the world truly is beckoning us on, we'll be less inclined to give up on making programs ever more responsive. It is, I think, through the mangling of ontologies in practice that the like-minded gain the resolve to create a new world.

[22] In the case of the agile methods, the postulates are to favor (1) individuals and interactions over processes and tools, (2) working software over comprehensive documentation, (3) customer collaboration over contract negotiation, and (4) responding to change over following a plan. That's from the Agile Manifesto (XX), so it's a tad vague. See Cockburn XX or Schwaber XX for more exposition.

[23] Today there are programmers happily programming in large bullpens, writing code only in pairs, planning by scribbling on 3x5 cards, and working without any notion of "code ownership". If you'd predicted that a decade ago, when they were fighting to stay in their semi-private offices and out of cubicles, they would have thought you mad.

[24] At a workshop where I submitted a position paper titled "Agile methods, the Emersonian worldview, and the dance of agency", one colleague - a person sympathetic to the agile methods - quoted his M.D. wife as saying that if, in fact, I believed this position paper, her diagnosis would be clinical schizophrenia. Judge for yourself: http://www.visibleworkings.com/papers/agile-methods-and-emerson.html

Appendix A - programming by example

In this section, I'll explain the code used in this paper in more detail. Let's begin with the untidy version of the fixture that causes all the tests to pass. It begins like this:

```
public class AnimalProgressFixture extends fit.StepFixture {
  … lots of stuff …
}
```

This defines a new *class* (kind) of *object*. An object is a distinct program entity that bundles together *state* (memory) and *behavior*. The class definition describes the behavior and the kinds of state that all objects "of that class" share. Let me start with a simpler example:

```
public class Worker {
  … lots of stuff …
}
```

Here's how you create a particular Worker:

```
Worker someWorker = new Worker();
```

someWorker is a *variable*. Think of it as naming a particular new Worker.

Let's suppose that all we care about is money. A worker has a particular salary, which can be paid at periodic intervals. We pay *someWorker* by sending it a *message*, like this:

```
someWorker.payYourself();
```

How does *someWorker* know what to do with the *payYourself* message? By looking for the *method definition* in the class, which would look something like this:

```
public class Worker {
   public void payYourself() {
      … whatever needs to be done to pay yourself…
   }
   …
}
```

How does the Worker know how much to pay itself? It has to be told its salary:

```
someWorker.setSalary(100000);
```

Here, the 1000000 (presumably representing an amount of currency) is the *argument* to the message. This *message send* tells the object to execute (run, perform) the following method:

```
public class Worker {
   …
   private int salary = 0;

   public void setSalary(int amount) {
      salary = amount;
   }
   …
}
```

Look at *setSalary* first. It declares that its argument is an integer (*int*) and gives it a name (*amount*), then stashes the value away as part of *someWorker*'s state by *assigning* it to variable *salary*. In every Worker, the name *salary* refers to the stashed salary. If the salary is never assigned any other value, it's zero.

Now's perhaps a good time to talk about the two "private" and "public" annotations. Something that's public is available to other objects. Any object can set *someWorker*'s salary through *setSalary*. But no object besides *someWorker* itself can change *someWorker*'s private *salary* directly. We'll see more of this in a moment.

We have almost enough to understand AnimalProgressFixture. Let's suppose there are special kinds of Workers. For example, my wife's a professor, which means she doesn't pay FICA (social security). That's different from most workers. But in many other ways, she's just the same. She does have a salary, she does pay Federal income tax, and so forth. It would be convenient to say, "Professors are pretty much Workers, except for the following differences…" In the Java programming language, that looks like:

```
public class Professor extends Worker {
   public void payYourself() {
      … special ways that Professors pay themselves…
   }
   …
}
```

Suppose you make a new professor:

```
Worker drDawn = new Professor();
```

And you set a salary and pay it:

```
drDawn.setSalary(100000);
drDawn.payYourself();
```

What happens? There are two versions of *payYourself*. One belongs to Worker, and one to Professor. Since Dawn is, specifically, a Professor, the Professor version will be used. The Java language automatically knows to use the most specific one. This is called *runtime polymorphism*. It's *polymorphic* because a single message corresponds to several methods with the same name (in different classes). It's *runtime* because Java picks which one to use at the last moment.

In contrast, there's only one version of *setSalary*, the one belonging to Worker. All Workers set their salary the same way, so that's the only *setSalary* Java will ever use.

Programmers would say that Professor *inherits* the *setSalary* method from Worker, but it *overrides* Worker's *payYourself* method.

The starting code

So now I hope you understand this:

```
public class AnimalProgressFixture extends fit.StepFixture {
```

Our program will contain one or more AnimalProgressFixture objects, each of which is a kind of fit.StepFixture. That means they *inherit* a bunch of behavior from fit.StepFixture. That behavior is all concerned with interpreting tables. What's in the AnimalProgressFixture is all about the content of those tables. Here's some of that:

```
private int balance = 0;
private int dailyCharge = 0;
private String studentMilkingFrequency = "never";
private String caretakerMilkingFrequency = "never";
private String studentTemperatureFrequency = "6 hours";
```

Here's the state that the AnimalProgressFixture works with. *balance* should be familiar - it will be used just like a Worker's *salary*. But not everything this fixture works with will be a number. It also works with *strings*, like "never", which are just sequences of characters.

Here's a method definition:

```
public void newCase(String animal, String owner) {
}
```

It's invoked by a table entry like "new case | betsy | rankin". The class fit.StepFixture knows how to turn that entry into a message to this AnimalProgressFixture. But *newCase* does nothing. That's because there's nothing in any test that forced me to write any code in here. (My discipline is to only write code required to make a failing test pass.) Presumably someday there will be tests that make the names of animals and owners relevant.

Here's a method that does things:

```
public void order(String order) {
   maybeUpdateDailyCharge(order);
   maybeUpdateBalance(order);
   if (order.equals("milking")) {
      studentMilkingFrequency = "3 hours";
      caretakerMilkingFrequency = "12 hours";
   } else if (order.equals("normal board")) {
      studentTemperatureFrequency = "never";
      studentMilkingFrequency = "never";
   }
}
```

The first thing it does is send two messages. But notice that these look different. Instead of being of the form ***object.****message(argument)*, no *object* is given. That's shorthand for sending the message to this object itself, not another one. If you look further down in the code, you can see the methods *maybeUpdateDailyCharge* and *maybeUpdateBalance*.

After that, there's an *if* test. The awkward-looking *order.equals("milking")* says "if the order is the string 'milking' then…"

Finally, let me complete the explanation with two more methods:

```
public void charge(int amount) {
   balance += amount;
}

public int balance() {
   return balance;
}
```

I hope the first is now familiar, except for the odd jargon "+=". That means that *balance*'s value should be increased by the value of *amount*.

The second is different because of "int" in front of "balance()". Everything else we've seen up to now has been labeled "void". When a message invokes a method, that method can return some value to its invoker. But the "void" has said that each of the methods so far has nothing to return. The "int" indicates that the method returns an integer. Specifically, it returns the value of the state variable *balance*. (Remember that no outside object can see the *balance* variable directly, so this is the only way for an outsider to get at it.)

Order handling

In the discussion, I show how I remove *if* statements by making orders into their own classes. Specifically, I take advantage of runtime polymorphism (as with the *Professor extends Worker* example above). First, I create an Order class. It doesn't do much of anything:

```
public class Order {

   public static Order from(String orderName) {
      … described later…
   }

   public void maybeUpdateSchedule(Schedule schedule) {
   }

   public void maybeUpdateBill(Bill bill) {
   }
}
```

Here, an Order is an object that could conceivably update both a Bill and a Schedule. But the generic, default Order actually does nothing. It is the special case classes (called *subclasses*) that do the work. Here's the subclass for the order to milk, the MilkingOrder:

```
static class MilkingOrder extends Order {
    public void maybeUpdateSchedule(Schedule schedule) {
        schedule.set("student", "milking", "3 hours");
        schedule.set("caretaker", "milking", "12 hours");
    }
}
```

A MilkingOrder is just an Order, except it has its own special version of *maybeUpdateSchedule*. That method *overrides* Order's version to do something: specifically, to change how often the student and caretaker milk. Notice that the MilkingOrder doesn't affect the bill. An owner is charged the same whether or not the cow is milked.

In contrast, PathologyOrder changes the bill but does not change what the student or caretakers do:

```
static class PathologyOrder extends Order {
    public void maybeUpdateBill(Bill bill) {
        bill.charge(80);
    }
}
```

Appendix B - programming jargon

Gosh, I don't think this is going to help.

argument
: A data value passed to a **method** when **sending a message**. Within the **invoked method**, the value is **assigned** to a **variable**.

assignment
: Assignment associates a particular value with a **variable**. Thereafter, you can use the variable to obtain the value.

behavior
: The externally visible results of **sending a message** to an **object**.

body, method
: The body of the **method** is the actual code that produces a result when the method is **invoked**. Contrast to its **declaration**.

class
: A class describes what's common to a bunch of **objects**. It describes the **messages** that can be sent to one of these objects and the kinds of **state** that the objects maintain. You can think of a class as a Platonic ideal object.

declaration, method
: The method declaration gives the **method's** name and describes each of its **arguments**. It does not describe how the method does whatever it does.

definition, method
: Same as the method **body**.

fixture
: An object that understands how to read a test table and convert rows within it into **message sends** (to the object itself).

inheriting a method
: Suppose Professor is a **subclass** of Worker. Then Professor inherits each **method** defined in Worker that Professor does not itself define.

invoking a method
: See **sending a method**.

Java
: A particular programming language.

method
: A method contains code that does some useful chunk of **behavior**. All the action in a program happens when methods are **invoked** by **objects**.

message
: A **message** is a way to refer to an **object**'s **method.**

message sends
: See **sending a message**.

object
: An object is the fundamental bundle of "stuff" within a program. It is described by a **class**. The class might say, for example, that its objects have **strings** called *address* and *name*. But each object will have its own value for those two values. When objects are **sent messages**, they typically perform some action that depends in part on the values of the state.

overriding a method

Suppose Professor is a **subclass** of Worker and that both Professor and Worker define a **method** named *foo*. Then Professor's *foo* overrides Worker's *foo*. If a professor **object** is sent the *foo* **message**, its *foo* will be used rather than Worker's.

recursion

See **recursion**. (Sorry: traditional joke.)

refactoring

When code is refactored, its externally-visible **behavior** and **declaration** stay the same, but its internal structure or form is changed.

sending a message

A **message** is sent to an object *receiver* by an object *sender* when *sender* names one of *receiver*'s **methods** and supplies values for each of that method's **arguments**. In **Java**, a message send would look like this:

receiver.methodName(value1, value2)

state

The memory of an object. A value that is retained until its replaced by some other value.

string

A sequence of text characters, like "hello".

subclass

A **class** that's a specialized version of some other class. For example, a SociologySeminar would be a subclass of the Seminar class. It would **inherit** those **methods** of Seminar that it did not **override**.

variable

A variable is a name that's used to hang onto some value for later use.