# 11

# Heuristics and Coffee


Over the past dozen years I have taught, and continue to teach, OO design to professional software developers. My courses are divided into morning lectures and afternoon exercises. For the exercises I will divide the class up into teams and have them solve a design problem using UML. The next morning we choose one or two teams to present their solutions on a white board, and we critique their designs.

I have taught these courses hundreds of times and have noticed that there is a group of design mistakes that are commonly made by the students. This chapter presents a few of the most common errors, shows why they are errors, and addresses how they can be corrected. Then it goes on to solve the problem in a way that I think resolves all the design forces nicely.

# The Mark IV Special Coffee Maker

During the first morning of an OOD class I present the basic definitions of classes, objects, relationships, methods, polymorphism, and so on. At the same time I present the basics of UML. Thus, the students learn the fundamental concepts, vocabulary, and tools of object oriented design.

During the afternoon I give the class the following exercise to work on: I ask them to design the software that controls a simple coffee maker. Here is the specification I give them.[1]

*The Mark IV Special Coffee Maker*

The Mark IV Special makes up to 12 cups of coffee at a time. The user places a filter in the filter holder, fills the filter with coffee grounds, and slides the filter holder into its receptacle. The user then pours up to 12 cups of water into the water strainer and presses the Brew button. The water is heated until boiling. The pressure of the evolving steam forces the water to be sprayed over the coffee grounds, and coffee drips through the filter into the pot. The pot is kept warm for extended periods by a warmer plate, which only turns on if there is coffee in the pot. If the pot is removed from the warmer plate while water is being sprayed over the grounds, the flow of water is stopped so that brewed coffee does not spill on the warmer plate. The following hardware needs to be monitored or controlled:

- The heating element for the boiler. It can be turned on or off.
- The heating element for the warmer plate. It can be turned on or off.
- The sensor for the warmer plate. It has three states: `warmerEmpty`, `potEmpty`, `potNotEmpty`.
- A sensor for the boiler, which determines whether there is water present. It has two states: `boilerEmpty` or `boilerNotEmpty`.
- The Brew button. This is a momentary button that starts the brewing cycle. It has an indicator that lights up when the brewing cycle is over and the coffee is ready.
- A pressure-relief valve that opens to reduce the pressure in the boiler. The drop in pressure stops the flow of water to the filter. It can be opened or closed.

The hardware for the Mark IV has been designed and is currently under development. The hardware engineers have even provided a low-level API for us to use, so we don't have to write any bit-twiddling I/O driver code. The code for these interface functions is shown in Listing 11–1. If this code looks strange to you, just keep in mind that it was written by hardware engineers.

1. This problem comes from my first book: [Martin1995], p. 60.

**Listing 11–1** CofeeMakerAPI.java

```
public interface CoffeeMakerAPI {
  public static CoffeeMakerAPI api = null; // set by main.

  /**
   * This function returns the status of the warmer-plate
   * sensor. This sensor detects the presence of the pot
   * and whether it has coffee in it.
   */
  public int getWarmerPlateStatus();

  public static final int WARMER_EMPTY = 0;
  public static final int POT_EMPTY = 1;
  public static final int POT_NOT_EMPTY = 2;

  /**
   * This function returns the status of the boiler switch.
   * The boiler switch is a float switch that detects if
   * there is more than 1/2 cup of water in the boiler.
   */
  public int getBoilerStatus();

  public static final int BOILER_EMPTY = 0;
  public static final int BOILER_NOT_EMPTY = 1;

  /**
   * This function returns the status of the brew button.
   * The brew button is a momentary switch that remembers
   * its state. Each call to this function returns the
   * remembered state and then resets that state to
   * BREW_BUTTON_NOT_PUSHED.
   *
   * Thus, even if this function is polled at a very slow
   * rate, it will still detect when the brew button is
   * pushed.
   */
  public int getBrewButtonStatus();

  public static final int BREW_BUTTON_PUSHED = 0;
  public static final int BREW_BUTTON_NOT_PUSHED = 1;

  /**
   * This function turns the heating element in the boiler
   * on or off.
   */
  public void setBoilerState(int boilerStatus);

  public static final int BOILER_ON = 0;
  public static final int BOILER_OFF = 1;

  /**
   * This function turns the heating element in the warmer
   * plate on or off.
   */
  public void setWarmerState(int warmerState);

  public static final int WARMER_ON = 0;
```

**Listing 11–1 (Continued)** CofeeMakerAPI.java

```
  public static final int WARMER_OFF = 1;

  /**
   * This function turns the indicator light on or off.
   * The indicator light should be turned on at the end
   * of the brewing cycle. It should be turned off when
   * the user presses the brew button.
   */
  public void setIndicatorState(int indicatorState);

  public static final int INDICATOR_ON = 0;
  public static final int INDICATOR_OFF = 1;

  /**
   * This function opens and closes the pressure-relief
   * valve. When this valve is closed, steam pressure in
   * the boiler will force hot water to spray out over
   * the coffee filter. When the valve is open, the steam
   * in the boiler escapes into the environment, and the
   * water in the boiler will not spray out over the filter.
   */
  public void setReliefValveState(int reliefValveState);

  public static final int VALVE_OPEN = 0;
  public static final int VALVE_CLOSED = 1;
}
```

## *A challenge*

If you want a challenge, stop reading here and try to design this software yourself. Remember that you are designing the software for a simple, embedded real-time system. What I expect of my students is a set of class diagrams, sequence diagrams, and state machines.

## *A common, but hideous, coffee maker solution*

By far the most common solution that my students present is the one in Figure 11–1. In this diagram we see the central `CoffeeMaker` class surrounded by minions that control the various devices. The `CoffeeMaker` contains a `Boiler`, a `WarmerPlate`, a `Button`, and a `Light`. The `Boiler` contains a `BoilerSensor` and a `BoilerHeater`. The `WarmerPlate` contains a `PlateSensor` and a `PlateHeater`. Finally there are two base classes, `Sensor` and `Heater`, that act as parents to the `Boiler` and `WarmerPlate` elements, respectively.

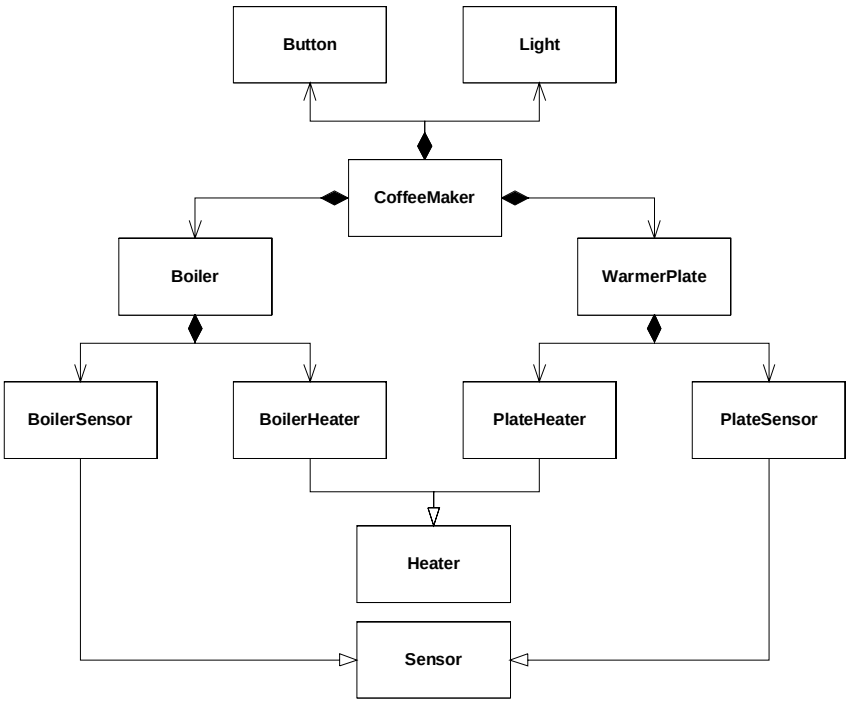


**Figure 11–1**
Hyper-concrete coffee maker.

It is hard for beginners to appreciate just how hideous this structure is. There are quite a few rather serious errors lurking in this diagram. Many of these errors would not be noticed until you actually tried to code this design and found that the code was absurd.

But before we get to the problems with the design itself, let's look at the problems with the way the UML is created.

## *Missing methods*

The biggest problem that Figure 11–1 exhibits is a complete lack of methods. We are writing a *program* here, and programs are about behavior! Where is the behavior in this diagram?

When designers create diagrams without methods they may be partitioning the software on something other than behavior. Partitionings that are not based upon behavior are almost always significant errors. It is the behavior of a system that is the first clue to how the software should be partitioned.

## *Vapor classes*

We can see how poorly partitioned this particular design is, if we consider the methods we might put in the class `Light`. Clearly the `Light` object just wants to be turned on or turned off. Thus we might put an `on()` and `off()` method in class `Light`. What would the implementation of those function look like? See Listing 11–2.

**Listing 11–2** `Light.java`

```
public class Light {
  public void on() {
    CoffeeMakerAPI.api.
      setIndicatorState(CoffeeMakerAPI.INDICATOR_ON);
  }

  public void off() {
    CoffeeMakerAPI.api.
      setIndicatorState(CoffeeMakerAPI.INDICATOR_OFF);
  }
}
```

There are some peculiar things about class `Light`. First, it has no variables. This is odd since an object usually has some kind of state that it manipulates. What's more, the `on()` and `off()` methods simply delegate to the `setIndicatorState` method of the `CoffeeMakerAPI`. So apparently the `Light` class is nothing more than a call translator. It's not really doing anything useful.

This same reasoning can be applied to the `Button`, `Boiler`, and `WarmerPlate` classes. They are nothing more than adapters that translate a function call from one form to another. Indeed, they could be removed from the design altogether without changing any of the logic in the `CoffeeMaker` class. That class would simply have to call the `CoffeeMakerAPI` directly instead of through the adapters.

By considering the methods, and then the code, we have demoted these classes from the prominent position the hold in Figure 11–1, to mere place holders without much reason to exist. For this reason, I call them *vapor classes*.

## *Imaginary abstraction*

Notice the `Sensor` and `Heater` base classes in Figure 11–1. The previous section should have convinced you that their derivatives were mere vapor, but what about base classes themselves? On the surface they seem to make a lot of sense. And yet, there doesn't seem to be any place for them.

Abstractions are tricky things. We humans see them everywhere, but many are not appropriate to be turned into base classes. These, in particular, have no place in this design. We can see this by asking ourselves who uses them.

No class in the system actually makes use of the `Sensor` or `Heater` class. If nobody uses them, what reason do they have to exist? Sometimes we might tolerate a base class

that nobody uses if it supplied some common code to the derivatives, but these bases have no code in them at all. At best their methods are abstract. Consider, for example, the `Heater` interface in Listing 11–3. A class with nothing but abstract functions and that no other class uses is officially useless.

**Listing 11–3** `Heater.java`

```
public interface Heater {
  public void turnOn();
  public void turnOff();
}
```

The `Sensor` class (Listing 11–4) is worse! Like `Heater`, it has abstract methods and no users. What's worse is that the return value of its sole method is ambiguous. What does the `sense()` method return? In the `BoilerSensor` it returns two possible values, but in `WarmerPlateSensor` it returns three possible values. In short, we cannot specify the contract of the *Sensor* in the interface. The best we can do is say that sensors may return `int`s. This is pretty weak.

**Listing 11–4** `Sensor.java`

```
public interface Sensor {
  public int sense();
}
```

What really happened here is that we read through the specification, found a bunch of likely nouns, made some inferences about their relationships, and then created a UML diagram based on that reasoning. If we accepted these decisions as an architecture and implemented them the way they stand, then we'd wind up with an all-powerful `CoffeeMaker` class surrounded by vaporous minions. We might as well program it in C!

## *God classes*

Everybody knows that god classes are a bad idea. We don't want to concentrate all the intelligence of a system into a single object or a single function. One of the goals of OOD is the partitioning and distribution of behavior into many classes and many functions. It turns out, however, that many object models that appear to be distributed are really the abode of gods in disguise. Figure 11–1 is a prime example. At first glance it looks like there are lots of classes with interesting behavior. But as we drill down into the code that would imple-

ment those classes we find that only one of those classes, `CoffeeMaker`, has any interesting behavior, and the rest are all imaginary abstractions or vapor classes.

# A Coffee Maker Solution

Solving the coffee maker problem is an interesting exercise in abstraction. Most developers new to OO find themselves quite surprised by the result.

The trick to solving this problem is to step back and separate its details from its essential nature. Forget about boilers, valves, heaters, sensors, and all the little details and concentrate on the underlying problem. What is that problem? The problem is, how do you make coffee?

How *do* you make coffee? The simplest, and most common solution to this problem, is to pour hot water over coffee grounds, and to collect the resulting infusion in some kind of vessel. Where do we get the hot water from? Let's call it a `HotWaterSource`. Where do we collect the coffee? Let's call it a `ContainmentVessel`[2].

Are these two abstractions really classes? Does a `HotWaterSource` have behavior that could be captured in software? Does a `ContainmentVessel` do something that software could control? If we think about the Mark IV unit, we could imagine the boiler, valve, and boiler sensor playing the role of the `HotWaterSource`. The `HotWaterSource` would be responsible for heating the water and delivering it over the coffee grounds to drip into the `ContainmentVessel`. We could also imagine the warmer plate and its sensor playing the role of the `ContainmentVessel`. It would be responsible for keeping the contained coffee warm, and also for letting us know whether there was any coffee left in the vessel.

## *Crossed wires*

How would you capture the previous discussion in a UML diagram? Figure 11–2 shows one possible schema. `HotWaterSource` and `ContainmentVessel` are both represented as classes, and are associated by the flow of coffee.

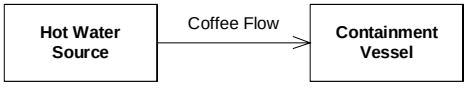


**Figure 11–2**
Crossed wires.

2. That name is particularly appropriate for the kind of coffee that *I* like to make.

The association shows an error that OO novices commonly make. The association is made with something physical about the problem instead of with the control of software behavior. The fact that coffee flows from the `HotWaterSource` to the `ContainmentVessel` is completely irrelevant to the association between those two classes.

For example, what if the software in the `ContainmentVessel` told the `HotWaterSource` when to start and stop the flow of hot water into the vessel? This might be depicted as shown in Figure 11–3. Notice that the `ContainmentVessel` is sending the start message to the `HotWaterSource`. This means that the association in Figure 11–2 is backwards. `HotWaterSource` does not depend upon the `ContainmentVessel` at all. Rather, the `ContainmentVessel` depends upon the `HotWaterSource`.

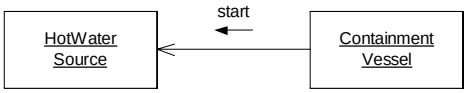


**Figure 11–3**
Starting the flow of hot water.

The lesson here is simply this: Associations are the pathways through which messages are sent between objects. They have nothing to do with the flow of physical objects. The fact that hot water flows from the boiler to the pot does not mean that there should be an association from the `HotWaterSource` to the `ContainmentVessel`.

I call this particular mistake *crossed wires* because the wiring between the classes has gotten crossed between the logical and physical domains.

## *The coffee maker user interface*

It should be clear that something is missing from our coffee maker model. We have a `HotWaterSource` and a `ContainmentVessel`, but we don't have any way for a human to interact with the system. Somewhere our system has to listen for commands from a human. Likewise the system must be able to report its status to its human owners. Certainly the Mark IV had hardware dedicated to this purpose. The button and the light served as the user interface.

Thus, we'll add a `UserInterface` class to our coffee maker model. This gives us a triad of classes interacting to create coffee under the direction of a user.

## *Use Case 1: User pushes brew button*

OK, given these three classes, how do their instances communicate? Let's look at several use cases to see if we can find out what the behavior of these classes is.

Which one of our objects detects the fact that the user has pressed the Brew button? Clearly, it must be the `UserInterface` object. What should this object do when the Brew button is pushed?

Our goal is to start the flow of hot water. However, before we can do that, we'd better make sure that the `ContainmentVessel` is ready to accept coffee. We'd also better make sure that the `HotWaterSource` is ready. If we think about the Mark IV, we're making sure that the boiler is full, and that the pot is empty and in place on the warmer.

So the first thing our `UserInterface` object does is to send a message to the `HotWaterSource` and the `ContainmentVessel` to see if they are ready. This is shown in Figure 11–4.

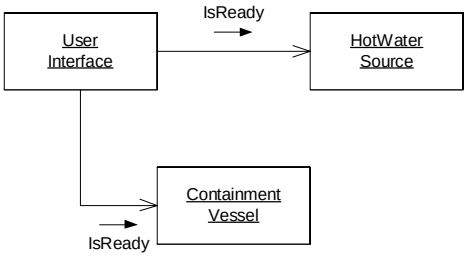


**Figure 11–4**
Brew button pressed, checking for ready.

If either of these queries returns false, then we refuse to start brewing coffee. The `UserInterface` object can take care of letting the user know that his request was denied. In the Mark IV case, we might flash the light a few times.

If both queries return true, then we need to start the flow of hot water. Probably the `UserInterface` object should send a `Start` message to the `HotWaterSource`. The `HotWaterSource` will then start doing whatever it needs to do to get hot water flowing. In the case of the Mark IV, it will close the valve and turn on the boiler. Figure 11–5 shows the completed scenario.

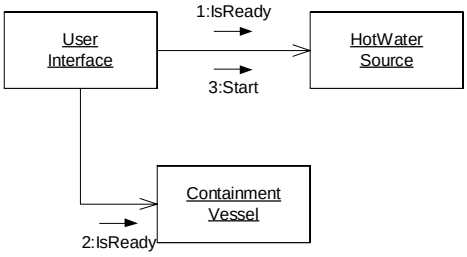


**Figure 11–5**
Brew button pressed, complete.

## *Use Case 2: Containment vessel not ready*

In the Mark IV we know that the user can take the pot off the warmer while coffee is brewing. Which one of our objects would detect the fact that the pot had been removed? Certainly it would be the `ContainmentVessel`. The requirements for the Mark IV tell us that we need to stop the flow of coffee when this happens. Thus the `Containment-Vessel` must be able to tell the `HotWaterSource` to stop sending hot water. Likewise, it needs to be able to tell it to start again when the pot is replaced. Figure 11–6 adds the new methods.

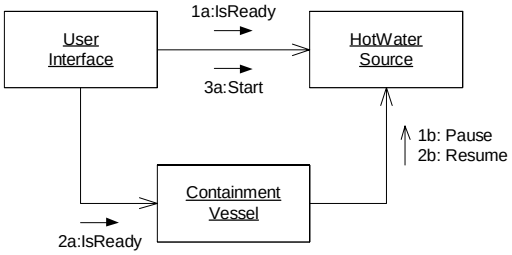


**Figure 11–6**
Pausing and resuming the flow of hot water.

## *Use Case 3: Brewing complete*

At some point we will be done brewing coffee, and we'll have to turn off the flow of hot water. Which one of our objects knows when brewing is complete? In the Mark IV's case the sensor in the boiler tells us that the boiler is empty. So our `HotWaterSource` would detect this. However, it's not hard to envision a coffee maker in which the `Containment-Vessel` would be the one to detect that brewing was done. For example, what if our coffee maker was plumbed into the water mains and therefore had an infinite supply of water? What if the water was heated by an intense microwave generator[3] as it flowed through the pipes into a thermally isolated vessel? What if that vessel had a spigot from which users got their coffee? In this case it would be a sensor in the vessel that would know that it was full, and that hot water should be shut off.

The point is that in the abstract domain of the `HotWaterSource` and `Containment-Vessel`, neither is an especially compelling candidate for detecting completion of the brew. My solution to that is to ignore the issue. I'll assume that either object can tell the others that brewing is complete.

Which objects in our model need to know that brewing is complete? Certainly the `UserInterface` needs to know since, in the Mark IV, it must turn the light on. It should

3. OK...I'm having a bit of fun. But, what if?

also be clear that the `HotWaterSource` needs to know that brewing is over, because it'll need to stop the flow of hot water. In the Mark IV it'll shut down the boiler and open the valve. Does the `ContainmentVessel` need to know that brewing is complete? Is there anything special that the `ContainmentVessel` needs to do, or to keep track of, once the brewing is complete? In the Mark IV it's going to detect an empty pot being put back on the plate, signalling that the user has poured the last of the coffee. This causes the Mark IV to turn the light *out*. So, yes, the `ContainmentVessel` needs to know that brewing is complete. Indeed, the same argument can be used to say that the `UserInterface` should send the `Start` message to the `ContainmentVessel` when brewing starts. Figure 11–7 shows the new messages. Note that I've shown that either `HotWaterSource` or `ContainmentVesslel` can send the `Done` message.

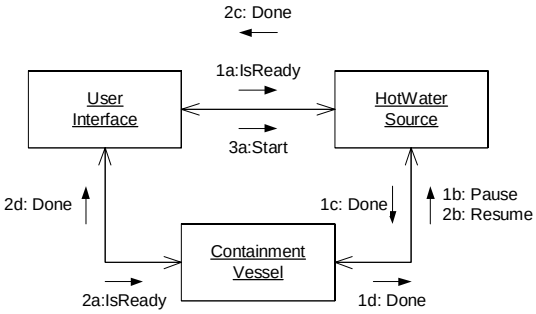


**Figure 11–7**
Detecting when brewing is complete.

## *Use Case 4: Coffee all gone*

The Mark IV shuts off the light when brewing is complete *and* an empty pot is placed on the plate. Clearly, in our object model, it is the `ContainmentVessel` that should detect this. It will have to send a `Complete` message to the `UserInterface`. Figure 11–8 shows the completed collaboration diagram.

From this diagram we can draw a class diagram with all the associations intact. This diagram holds no surprises. You can see it in Figure 11–9.

## *Implementing the abstract model*

Our object model is reasonably well partitioned. We have three distinct areas of responsibility, and each seems to be sending and receiving messages in a balanced way. There does not appear to be a god object anywhere. Nor do there appear to be any vapor classes.

So far, so good, but how do we implement the Mark IV in this structure? Do we just implement the methods of these three classes to invoke the `CoffeeMakerAPI`? This

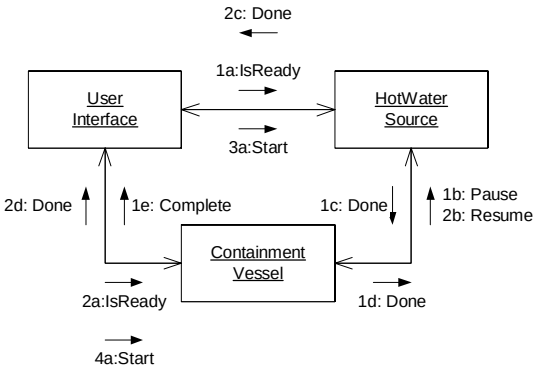


**Figure 11–8**
Coffee all gone.

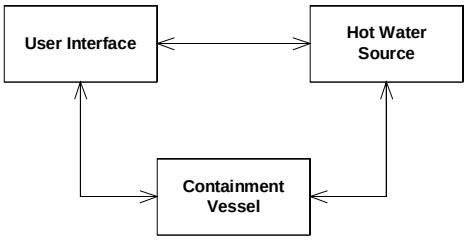


**Figure 11–9**
Class diagram.

would be a real shame! We've captured the essence of what it takes to make coffee. It would be pitifully poor design if we were to now tie that essence to the Mark IV.

In fact, I'm going to make a rule right now. None of the three classes we have created must ever know anything about the Mark IV. This is the Dependency Inversion Principle (DIP). We are not going to allow the high-level coffee making policy of this system to depend upon the low-level implementation.

OK, then how will we create the Mark IV implementation? Let's look at all the use cases again, but this time, let's look at them from the Mark IV point of view.

## *Use Case 1: User pushes Brew button*

Looking at our model, how does the `UserInterface` know that the Brew button has been pushed? Clearly, it must call the `CoffeeMakerAPI.getBrewButtonStatus()` function. Where should it call this function? We've already decreed that the `UserInterface` class itself cannot know about the CoffeeMakerAPI. So where does this call go?

We'll apply the DIP and put the call in a derivative of `UserInterface`. See Figure 11–10 for details.

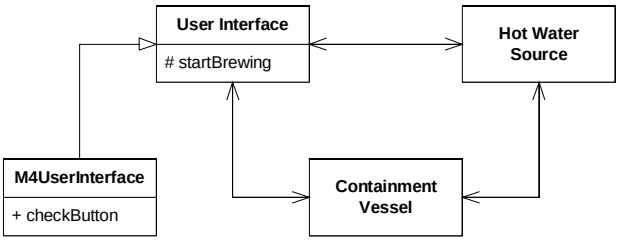


**Figure 11–10**
Detecting the Brew button.

We've derived `M4UserInterface` from `UserInterface`, and we've put a `checkButton()` method in `M4UserInterface`. When this function is called, it will call the `CoffeeMakerAPI.getBrewButtonStatus()` function. If the button has been pressed, it will invoke the protected `startBrewing()` method of `UserInterface`. Listings 11–5 and 11–6 show how this would be coded.

**Listing 11–5** `M4UserInterface.java`

```
public class M4UserInterface extends UserInterface {
  private void checkButton() {
    int buttonStatus =
        CoffeeMakerAPI.api.getBrewButtonStatus();
    if (buttonStatus == CoffeeMakerAPI.BREW_BUTTON_PUSHED) {
      startBrewing();
    }
  }
}
```

**Listing 11–6** `UserInterface.java`

```
public class UserInterface {
  private HotWaterSource hws;
  private ContainmentVessel cv;

  public void done() {}
  public void complete() {}
  protected void startBrewing() {
    if (hws.isReady() && cv.isReady()) {
      hws.start();
      cv.start();
    }
  }
}
```

You might be wondering why I created the protected `startBrewing()` method at all. Why didn't I just call the `start()` functions from `M4UserInterface`? The reason is simple, but significant. The `isReady()` tests, and the consequential calls to the `start()` methods of the `HotWaterSource` and the `ContainmentVessel` are high-level policy that the `UserInterface` class should possess. That code is valid irrespective of whether we are implementing a Mark IV and should therefore not be coupled to the Mark IV derivative. You will see me make this same distinction over and over again in this example. I keep as much code as I can in the high-level classes. The only code I put into the derivatives is code that is directly, and inextricably, associated with the Mark IV.

## *Implementing the `isReady()` functions*

How are the `isReady()` methods of `HotWaterSource` and `ContainmentVessel` implemented? It should be clear that these are really just abstract methods, and that these classes are therefore abstract classes. The corresponding derivatives `M4HotWaterSource` and `M4ContainmentVessel` will implement them by calling the appropriate `CoffeeMakerAPI` functions. Figure 11–11 shows the new structure, and Listings 11–7 and 11–8 show the implementation of the two derivatives.

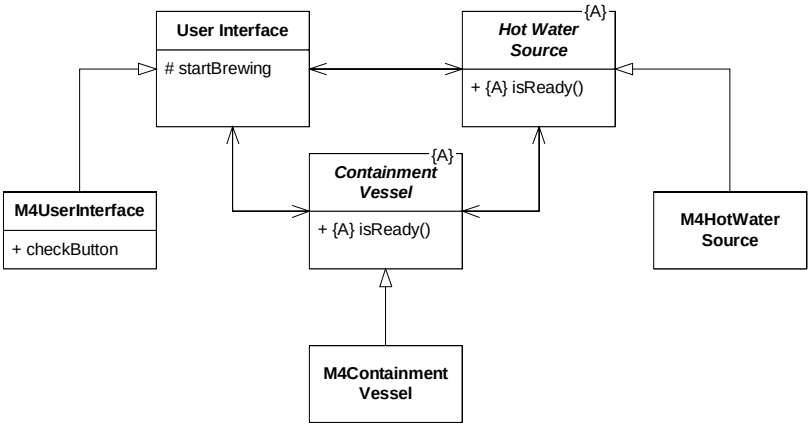


**Figure 11–11**
Implementing the `isReady` methods.

**Listing 11–7** `M4HotWaterSource.java`

```
public class M4HotWaterSource extends HotWaterSource {
  public boolean isReady() {
    int boilerStatus =
      CoffeeMakerAPI.api.getBoilerStatus();
    return boilerStatus == CoffeeMakerAPI.BOILER_NOT_EMPTY;
  }
}
```

**Listing 11–8** M4ContainmentVessel.java

```
public class M4ContainmentVessel extends ContainmentVessel {
  public boolean isReady() {
    int plateStatus =
      CoffeeMakerAPI.api.getWarmerPlateStatus();
    return plateStatus == CoffeeMakerAPI.POT_EMPTY;
  }
}
```

## *Implementing the `start()` functions*

The `start()` method of `HotWaterSource` is just an abstract method that is implemented by `M4HotWaterSource` to invoke the `CoffeeMakerAPI` functions that close the valve and turn on the boiler. As I wrote these functions I began to get tired of all the `CoffeeMakerAPI.api.XXX` structures I was writing, so I did a little refactoring at the same time. The result is in Listing 11–9.

**Listing 11–9** M4HotWaterSource.java

```
public class M4HotWaterSource extends HotWaterSource {
  CoffeeMakerAPI api;

  public M4HotWaterSource(CoffeeMakerAPI api) {
    this.api = api;
  }

  public boolean isReady() {
    int boilerStatus = api.getBoilerStatus();
    return boilerStatus == api.BOILER_NOT_EMPTY;
  }

  public void start() {
    api.setReliefValveState(api.VALVE_CLOSED);
    api.setBoilerState(api.BOILER_ON);
  }
}
```

The `start()` method for the `ContainmentVessel` is a little more interesting. The only action that the `M4ContainmentVessel` needs to take is to remember the brewing state of the system. As we'll see later, this will allow it to respond correctly when pots are placed on, or removed from, the plate. Listing 11–10 shows the code.

**Listing 11–10** M4ContainmentVessel.java

```
public class M4ContainmentVessel extends ContainmentVessel {
  private CoffeeMakerAPI api;
  private boolean isBrewing;

  public M4ContainmentVessel(CoffeeMakerAPI api) {
    this.api = api;
    isBrewing = false;
  }

  public boolean isReady() {
```

**Listing 11–10 (Continued)** M4ContainmentVessel.java

```
    int plateStatus = api.getWarmerPlateStatus();
    return plateStatus == api.POT_EMPTY;
  }

  public void start() {
     isBrewing = true;
  }
}
```

## *How does `M4UserInterface.checkButton` get called?*

This is an interesting point. How does the flow of control ever get to a place at which the `CoffeeMakerAPI.getBrewButtonStatus()` function can be called? For that matter, how does the flow of control get to where *any* of the sensors can be detected?

Many of the teams who try to solve this problem get completely hung up on this point. Some don't want to assume that there's a multithreading operating system in the coffee maker, and so they want to use a polling approach to the sensors. Others want to put multithreading in so that they don't have to worry about polling. I've seen this particular argument go back and forth for an hour or more in some teams.

The mistake that these teams are making (which I eventually point out to them after letting them sweat a bit) is that the choice between threading and polling is completely irrelevant. This decision can be made at the very last minute without harm to the design. Therefore it is always best to assume that messages can be sent asynchronously, as though there were independent threads, and then put the polling or threading in at the last minute.

The design so far has assumed that somehow the flow of control will asynchronously get into the `M4UserInterface` object so that it can call `CoffeeMakerAPI.getBrewButtonStatus()`. Now let's assume that we are working in a very minimal JVM that does not support threading. This means we're going to have to poll. How can we make this work?

Consider the `Pollable` interface in Listing 11–11. This interface has nothing but a `poll()` method. Now, what if `M4UserInterface` implemented this interface? What if the `main()` program hung in a hard loop just calling this method over and over again? Then the flow of control would continuously be reentering `M4UserInterface` and we could detect the Brew button.

**Listing 11–11** Pollable.java

```
public interface Pollable {
  public void poll();
}
```

Indeed, we can repeat this pattern for all three of the M4 derivatives. Each has its own sensors it needs to check. So, as shown in Figure 11–12, we can derive all of the M4 derivatives from `Pollable` and call them all from `main()`.

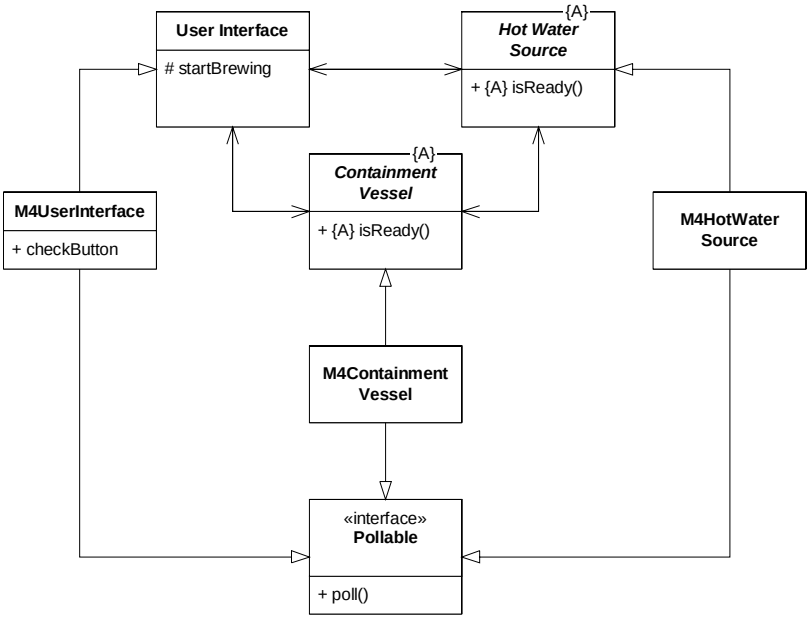


**Figure 11–12**
Pollable coffee maker.

Listing 11–12 shows what the main function might look like. It is placed in a class called `CoffeeMaker`. The `main()` function creates the implemented version of the `api`, and then creates the three M4 components. It calls `init()` functions to wire the components up to each other. Finally it hangs in an infinite loop calling `poll()` on each of the components in turn.

**Listing 11–12** `CoffeeMaker.java`

```
public class CoffeeMaker {
  public static void main(String[] args) {
    CoffeeMakerAPI api = new M4CoffeeMakerAPIImplementation();

    M4UserInterface ui = new M4UserInterface(api);
    M4HotWaterSource hws = new M4HotWaterSource(api);
    M4ContainmentVessel cv = new M4ContainmentVessel(api);

    ui.init(hws,cv);
    hws.init(ui,cv);
    cv.init(ui,hws);

    while(true) {
      ui.poll();
      hws.poll();
      cv.poll();
```

**Listing 11–12 (Continued)** CoffeeMaker.java

```
    }
  }
}
```

Now it should be clear how the `M4UserInterface.checkButton()` function gets called. Indeed, it should be clear that this function is really not called `checkButton()`. It is called `poll()`. Listing 11–13 shows what `M4UserInterface` looks like now.

**Listing 11–13** M4UserInterface.java

```
public class M4UserInterface extends UserInterface
                             implements Pollable {
  private CoffeeMakerAPI api;
  private HotWaterSource hws;
  private ContainmentVessel cv;

  public void init(HotWaterSource hws, ContainmentVessel cv) {
    this.hws = hws;
    this.cv = cv;
  }

  public M4UserInterface(CoffeeMakerAPI api) {
    this.api = api;
  }

  private void poll() {
    int buttonStatus = api.getBrewButtonStatus();
    if (buttonStatus == api.BREW_BUTTON_PUSHED) {
      startBrewing();
    }
  }
}
```

## *Completing the Coffee Maker*

The reasoning used in the previous sections can be repeated for each of the other components of the coffee maker. The result is shown in Listings 11–14 through 11–21.

## *The benefits of this design*

Despite the trivial nature of the problem, this design shows some very nice characteristics. Figure 11–13 shows the structure. I have drawn a line around the three abstract classes. These are the classes that hold the high-level policy of the coffee maker. Notice that all dependencies that cross the line point inward. Nothing inside the line depends upon anything outside. Thus, the abstractions are completely separated from the details.

The abstract classes know nothing of buttons, lights, valves, sensors, nor any other of the detailed elements of the coffee maker. By the same token, the derivatives are dominated by those details.

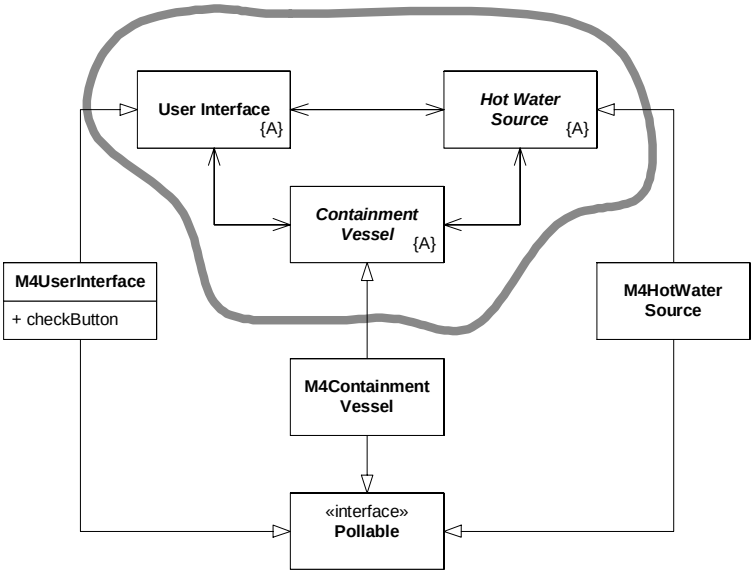


**Figure 11–13**
Coffee maker components.

Note that the three abstract classes could be reused to make many different kinds of coffee machines. We could easily use them in a coffee machine that is connected to the water mains and uses a tank and spigot. It seems likely that we could also use them for a coffee vending machine. Indeed, I think we could use it in an automatic tea brewer or even a chicken soup maker. This segregation between high-level policy and detail is the essence of object oriented design.

## *How did I really come up with this design?*

I did not just sit down one day and develop this design in a nice straightfoward manner. Indeed, my very first design for the coffee maker looked much more like Figure 11–1. However, I have written about this problem many times, and have used it as an exercise while teaching class after class. So this design has been refined over time.

The code you see below was created, test first, using the unit tests in Listing 11–22. I created the code based upon the structure in Figure 11–13, but put it together incrementally, one failing test case at a time.[4]

4. [Beck2002].

I am not convinced that the test cases are complete. If this were more than an example program, I'd do a more exhaustive analysis on the test cases. However, I felt that such an analysis would have been overkill for this book.

**Listing 11–14** UserInterface.java

```
public abstract class UserInterface {
  private HotWaterSource hws;
  private ContainmentVessel cv;
  protected boolean isComplete;

  public UserInterface() {
    isComplete = true;
  }

  public void init(HotWaterSource hws, ContainmentVessel cv) {
    this.hws = hws;
    this.cv = cv;
  }

  public void complete() {
    isComplete = true;
    completeCycle();
  }

  protected void startBrewing() {
    if (hws.isReady() && cv.isReady()) {
      isComplete = false;
      hws.start();
      cv.start();
    }
  }

  public abstract void done();
  public abstract void completeCycle();
}
```

**Listing 11–15** M4UserInterface.java

```
public class M4UserInterface extends UserInterface
                             implements Pollable {
  private CoffeeMakerAPI api;

  public M4UserInterface(CoffeeMakerAPI api) {
    this.api = api;
  }

  public void poll() {
    int buttonStatus = api.getBrewButtonStatus();
    if (buttonStatus == api.BREW_BUTTON_PUSHED) {
      startBrewing();
    }
  }

  public void done() {
    api.setIndicatorState(api.INDICATOR_ON);
  }
```

**Listing 11–15 (Continued)** M4UserInterface.java

```
  public void completeCycle() {
    api.setIndicatorState(api.INDICATOR_OFF);
  }
}
```

**Listing 11–16** HotWaterSource.java

```
public abstract class HotWaterSource {
  private UserInterface ui;
  private ContainmentVessel cv;
  protected boolean isBrewing;

  public HotWaterSource() {
    isBrewing = false;
  }

  public void init(UserInterface ui, ContainmentVessel cv) {
    this.ui = ui;
    this.cv = cv;
  }

  public void start() {
    isBrewing = true;
    startBrewing();
  }

  public void done() {
    isBrewing = false;
  }

  protected void declareDone() {
    ui.done();
    cv.done();
    isBrewing = false;
  }

  public abstract boolean isReady();
  public abstract void startBrewing();
  public abstract void pause();
  public abstract void resume();

}
```

**Listing 11–17** M4HotWaterSource.java

```
public class M4HotWaterSource extends HotWaterSource
                              implements Pollable {
  private CoffeeMakerAPI api;

  public M4HotWaterSource(CoffeeMakerAPI api) {
    this.api = api;
  }

  public boolean isReady() {
    int boilerStatus = api.getBoilerStatus();
    return boilerStatus == api.BOILER_NOT_EMPTY;
```

**Listing 11–17 (Continued)** M4HotWaterSource.java

```
  }

  public void startBrewing() {
    api.setReliefValveState(api.VALVE_CLOSED);
    api.setBoilerState(api.BOILER_ON);
  }

  public void poll() {
    int boilerStatus = api.getBoilerStatus();
    if (isBrewing) {
      if (boilerStatus == api.BOILER_EMPTY) {
          api.setBoilerState(api.BOILER_OFF);
          api.setReliefValveState(api.VALVE_CLOSED);
          declareDone();
      }
    }
  }

  public void pause() {
    api.setBoilerState(api.BOILER_OFF);
    api.setReliefValveState(api.VALVE_OPEN);
  }

  public void resume() {
    api.setBoilerState(api.BOILER_ON);
    api.setReliefValveState(api.VALVE_CLOSED);
  }
}
```

**Listing 11–18** ContainmentVessel.java

```
public abstract class ContainmentVessel {
  private UserInterface ui;
  private HotWaterSource hws;
  protected boolean isBrewing;
  protected boolean isComplete;

  public ContainmentVessel() {
    isBrewing = false;
    isComplete = true;
  }

  public void init(UserInterface ui, HotWaterSource hws) {
    this.ui = ui;
    this.hws = hws;
  }

  public void start() {
    isBrewing = true;
    isComplete = false;
  }

  public void done() {
    isBrewing = false;
  }
```

**Listing 11–18 (Continued)** ContainmentVessel.java

```
  protected void declareComplete() {
    isComplete = true;
    ui.complete();
  }

  protected void containerAvailable() {
    hws.resume();
  }

  protected void containerUnavailable() {
    hws.pause();
  }

  public abstract boolean isReady();

}
```

**Listing 11–19** M4ContainmentVessel.java

```
public class M4ContainmentVessel extends ContainmentVessel
  implements Pollable {
  private CoffeeMakerAPI api;
  private int lastPotStatus;

  public M4ContainmentVessel(CoffeeMakerAPI api) {
    this.api = api;
    lastPotStatus = api.POT_EMPTY;
  }

  public boolean isReady() {
    int plateStatus = api.getWarmerPlateStatus();
    return plateStatus == api.POT_EMPTY;
  }

  public void poll() {
    int potStatus = api.getWarmerPlateStatus();
    if (potStatus != lastPotStatus) {
      if (isBrewing) {
        handleBrewingEvent(potStatus);
      } else if (isComplete == false) {
        handleIncompleteEvent(potStatus);
      }
      lastPotStatus = potStatus;
    }
  }

  private void handleBrewingEvent(int potStatus) {
    if (potStatus == api.POT_NOT_EMPTY) {
      containerAvailable();
      api.setWarmerState(api.WARMER_ON);
    } else if (potStatus == api.WARMER_EMPTY) {
      containerUnavailable();
      api.setWarmerState(api.WARMER_OFF);
    } else { // potStatus == api.POT_EMPTY
      containerAvailable();
      api.setWarmerState(api.WARMER_OFF);
```

**Listing 11–19 (Continued)** M4ContainmentVessel.java

```
    }
  }

  private void handleIncompleteEvent(int potStatus) {
    if (potStatus == api.POT_NOT_EMPTY) {
      api.setWarmerState(api.WARMER_ON);
    } else if (potStatus == api.WARMER_EMPTY) {
      api.setWarmerState(api.WARMER_OFF);
    } else { // potStatus == api.POT_EMPTY
      api.setWarmerState(api.WARMER_OFF);
      declareComplete();
    }
  }
}
```

**Listing 11–20** Pollable.java

```
public interface Pollable {
  public void poll();
}
```

**Listing 11–21** CoffeeMaker.java

```
public class CoffeeMaker {
  public static void main(String[] args) {
    CoffeeMakerAPI api = new M4CoffeeMakerAPIImplementation();

    M4UserInterface ui = new M4UserInterface(api);
    M4HotWaterSource hws = new M4HotWaterSource(api);
    M4ContainmentVessel cv = new M4ContainmentVessel(api);

    ui.init(hws,cv);
    hws.init(ui,cv);
    cv.init(ui,hws);

    while(true) {
      ui.poll();
      hws.poll();
      cv.poll();
    }
  }
}
```

**Listing 11–22** TestCoffeeMaker.java

```
import junit.framework.TestCase;
import junit.swingui.TestRunner;

class CoffeeMakerStub implements CoffeeMakerAPI {
  public boolean buttonPressed;
  public boolean lightOn;
  public boolean boilerOn;
  public boolean valveClosed;
  public boolean plateOn;
  public boolean boilerEmpty;
  public boolean potPresent;
  public boolean potNotEmpty;
```

**Listing 11–22 (Continued)** TestCoffeeMaker.java

```
  public CoffeeMakerStub() {
    buttonPressed = false;
    lightOn = false;
    boilerOn = false;
    valveClosed = true;
    plateOn = false;
    boilerEmpty = true;
    potPresent = true;
    potNotEmpty = false;
  }

  public int getWarmerPlateStatus() {
    if (!potPresent)
      return WARMER_EMPTY;
    else if (potNotEmpty)
      return POT_NOT_EMPTY;
    else
      return POT_EMPTY;
  }

  public int getBoilerStatus() {
    return boilerEmpty ? BOILER_EMPTY : BOILER_NOT_EMPTY;
  }

  public int getBrewButtonStatus() {
    if (buttonPressed) {
      buttonPressed = false;
      return BREW_BUTTON_PUSHED;
    } else {
      return BREW_BUTTON_NOT_PUSHED;
    }
  }

  public void setBoilerState(int boilerStatus) {
    boilerOn = boilerStatus == BOILER_ON;
  }

  public void setWarmerState(int warmerState) {
    plateOn = warmerState == WARMER_ON;
  }

  public void setIndicatorState(int indicatorState) {
    lightOn = indicatorState == INDICATOR_ON;
  }

  public void setReliefValveState(int reliefValveState) {
    valveClosed = reliefValveState == VALVE_CLOSED;
  }
}

public class TestCoffeeMaker extends TestCase {
  public static void main(String[] args) {
    TestRunner.main(new String[]{"TestCoffeeMaker"});
  }
```

**Listing 11–22 (Continued)** TestCoffeeMaker.java

```
  public TestCoffeeMaker(String name) {
    super(name);
  }

  private M4UserInterface ui;
  private M4HotWaterSource hws;
  private M4ContainmentVessel cv;
  private CoffeeMakerStub api;

  public void setUp() throws Exception {
    api = new CoffeeMakerStub();
    ui = new M4UserInterface(api);
    hws = new M4HotWaterSource(api);
    cv = new M4ContainmentVessel(api);
    ui.init(hws, cv);
    hws.init(ui, cv);
    cv.init(ui, hws);
  }

  private void poll() {
    ui.poll();
    hws.poll();
    cv.poll();
  }

  public void tearDown() throws Exception {
  }

  public void testInitialConditions() throws Exception {
    poll();
    assert(api.boilerOn == false);
    assert(api.lightOn == false);
    assert(api.plateOn == false);
    assert(api.valveClosed == true);
  }

  public void testStartNoPot() throws Exception {
    poll();
    api.buttonPressed = true;
    api.potPresent = false;
    poll();
    assert(api.boilerOn == false);
    assert(api.lightOn == false);
    assert(api.plateOn == false);
    assert(api.valveClosed == true);
  }

  public void testStartNoWater() throws Exception {
    poll();
    api.buttonPressed = true;
    api.boilerEmpty = true;
    poll();
    assert(api.boilerOn == false);
    assert(api.lightOn == false);
    assert(api.plateOn == false);
    assert(api.valveClosed == true);
```

**Listing 11–22 (Continued)** TestCoffeeMaker.java

```
  }

  public void testGoodStart() throws Exception {
    normalStart();
    assert(api.boilerOn == true);
    assert(api.lightOn == false);
    assert(api.plateOn == false);
    assert(api.valveClosed == true);
  }

  private void normalStart() {
    poll();
    api.boilerEmpty = false;
    api.buttonPressed = true;
    poll();
  }

  public void testStartedPotNotEmpty() throws Exception {
    normalStart();
    api.potNotEmpty = true;
    poll();
    assert(api.boilerOn == true);
    assert(api.lightOn == false);
    assert(api.plateOn == true);
    assert(api.valveClosed == true);
  }

  public void testPotRemovedAndReplacedWhileEmpty()
  throws Exception {
    normalStart();
    api.potPresent = false;
    poll();
    assert(api.boilerOn == false);
    assert(api.lightOn == false);
    assert(api.plateOn == false);
    assert(api.valveClosed == false);
    api.potPresent = true;
    poll();
    assert(api.boilerOn == true);
    assert(api.lightOn == false);
    assert(api.plateOn == false);
    assert(api.valveClosed == true);
  }

  public void testPotRemovedWhileNotEmptyAndReplacedEmpty()
  throws Exception {
    normalFill();
    api.potPresent = false;
    poll();
    assert(api.boilerOn == false);
    assert(api.lightOn == false);
    assert(api.plateOn == false);
    assert(api.valveClosed == false);
    api.potPresent = true;
    api.potNotEmpty = false;
    poll();
```

**Listing 11–22 (Continued)** TestCoffeeMaker.java

```
    assert(api.boilerOn == true);
    assert(api.lightOn == false);
    assert(api.plateOn == false);
    assert(api.valveClosed == true);
  }

  private void normalFill() {
    normalStart();
    api.potNotEmpty = true;
    poll();
  }

  public void testPotRemovedWhileNotEmptyAndReplacedNotEmpty()
  throws Exception {
    normalFill();
    api.potPresent = false;
    poll();
    api.potPresent = true;
    poll();
    assert(api.boilerOn == true);
    assert(api.lightOn == false);
    assert(api.plateOn == true);
    assert(api.valveClosed == true);
  }

  public void testBoilerEmptyPotNotEmpty() throws Exception {
    normalBrew();
    assert(api.boilerOn == false);
    assert(api.lightOn == true);
    assert(api.plateOn == true);
    assert(api.valveClosed == true);
  }

  private void normalBrew() {
    normalFill();
    api.boilerEmpty = true;
    poll();
  }

  public void testBoilerEmptiesWhilePotRemoved()
  throws Exception {
    normalFill();
    api.potPresent = false;
    poll();
    api.boilerEmpty = true;
    poll();
    assert(api.boilerOn == false);
    assert(api.lightOn == true);
    assert(api.plateOn == false);
    assert(api.valveClosed == true);
    api.potPresent = true;
    poll();
    assert(api.boilerOn == false);
    assert(api.lightOn == true);
    assert(api.plateOn == true);
    assert(api.valveClosed == true);
```

**Listing 11–22 (Continued)** TestCoffeeMaker.java

```
  }

  public void testEmptyPotReturnedAfter() throws Exception {
    normalBrew();
    api.potNotEmpty = false;
    poll();
    assert(api.boilerOn == false);
    assert(api.lightOn == false);
    assert(api.plateOn == false);
    assert(api.valveClosed == true);
  }
}
```

## OOverkill

This example has certain pedagogical advantages. It is small, easy to understand, and shows how the principles of OOD can be used to manage dependencies and separate concerns. On the other hand, its very smallness means that the benefits of that separation probably do not outweigh the costs.

If we were to write the Mark IV coffee maker as a finite state machine, we'd find that it had seven states and 18 transitions.[5] We could encode this into 18 lines of SMC code. A simple main loop that polls the sensors would be another ten lines or so, and the action functions that the FSM would invoke would be another couple of dozen. In short, we could write the whole program in less than a page of code.

If we don't count the tests, the OO solution of the coffee maker is *five* pages of code. There is no way that we can justify this disparity. In larger applications the benefits of dependency management and the separation of concerns clearly outweigh the costs of OOD. However, in this example the reverse is true.

5. [Martin1995], p. 65.

# Notes

**[Martin1995]:** Robert C. Martin, *Designing Object Oriented C++ Applications using the Booch Method*. Upper Saddle River, NJ.: Prentice Hall, 1995.

**[Beck2002]:** Kent Beck, *Test-Driven Development*. Reading, Mass.: Addison-Wesley, 2002.