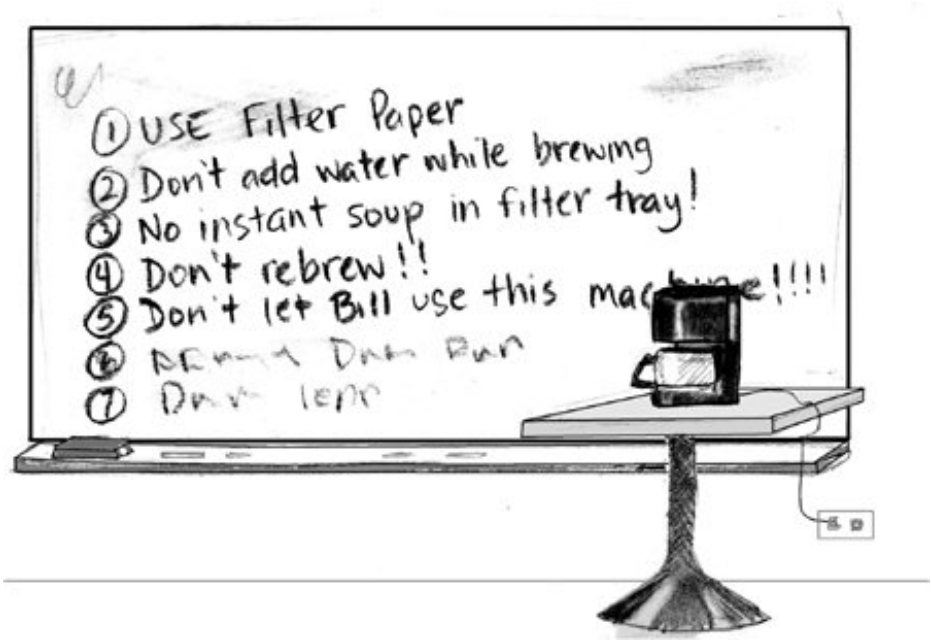
Chapter 20. Heuristics and Coffee



Angela Brooks

Over the past dozen years, I have taught object-oriented design to professional software developers. My courses are divided into morning lectures and afternoon exercises. For the exercises, I divide the class 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 whiteboard, and we critique their designs.

I have taught these courses hundreds of times and have noticed a group of design mistakes 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 the chapter 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: design the software that controls a simple coffee maker. Here is the specification I give them.[1]

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

Specification

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 turns on only if coffee is 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 water is present. It has two states: `boilerEmpty` or `boilerNotEmpty`.
- The Brew button. This momentary button 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. The value 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 20-1. If this code looks strange to you, keep in mind that it was written by hardware engineers.

Listing 20-1. **`CoffeeMakerAPI.cs`**

```
namespace CoffeeMaker
{
  public enum WarmerPlateStatus
  {
    WARMER_EMPTY,
    POT_EMPTY,
    POT_NOT_EMPTY
  };

  public enum BoilerStatus
  {
    EMPTY,NOT_EMPTY
  };

  public enum BrewButtonStatus
  {
    PUSHED,NOT_PUSHED
  };

  public enum BoilerState
  {
    ON,OFF
  };

  public enum WarmerState
  {
    ON,OFF
  };

  public enum IndicatorState
  {
    ON,OFF
  };

  public enum ReliefValveState
  {
    OPEN, CLOSED
  };

  public interface CoffeeMakerAPI
  {
    /*
     * 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.
     */
    WarmerPlateStatus GetWarmerPlateStatus();

    /*
     * 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.
     */

    BoilerStatus GetBoilerStatus();

    /*
     * 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
     * NOT_PUSHED.
     *
     * Thus, even if this function is polled at a very slow
     * rate, it will still detect when the brew button is
     * pushed.
     */

    BrewButtonStatus GetBrewButtonStatus();

    /*
     * This function turns the heating element in the boiler
     * on or off.
     */

    void SetBoilerState(BoilerState s);

    /*
     * This function turns the heating element in the warmer
     * plate on or off.
     */

    void SetWarmerState(WarmerState s);

    /*
     * 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.
     */

    void SetIndicatorState(IndicatorState s);

    /*
     * 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.
     */
    void SetReliefValveState(ReliefValveState s);
}
```

```
}
```

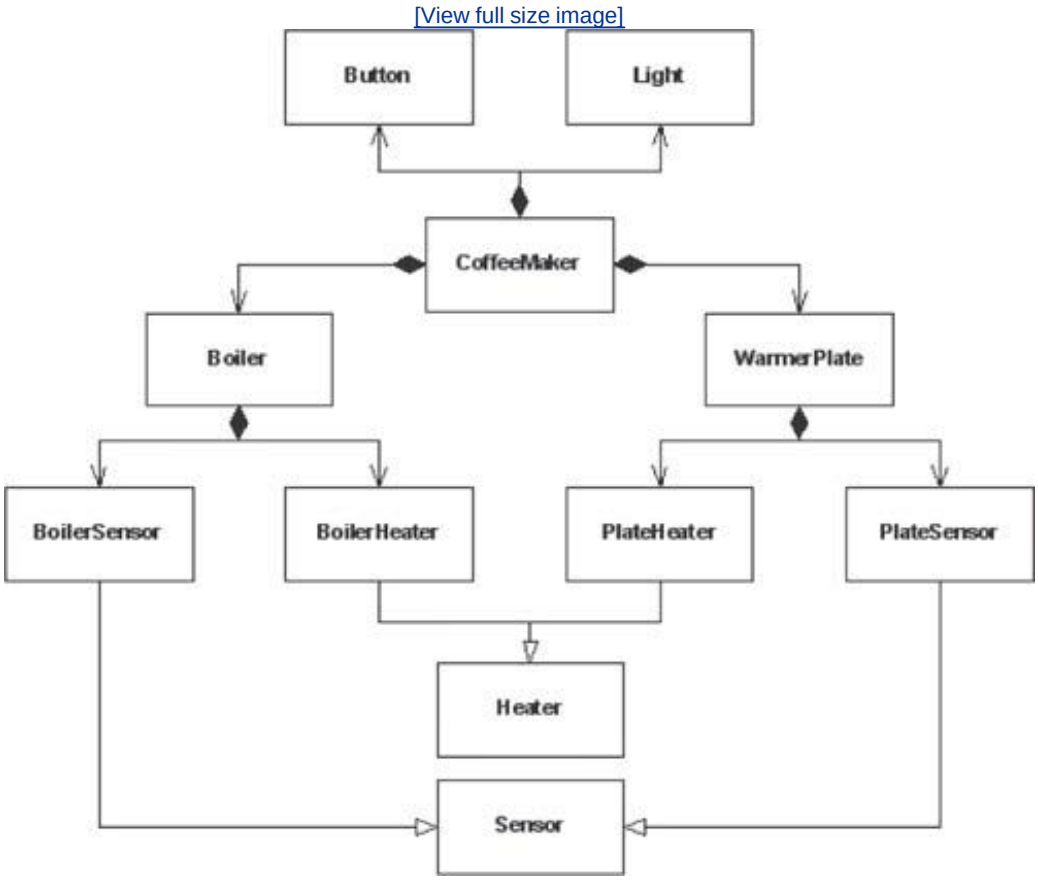
If you want a challenge, stop reading here and try to design this software yourself. Remember that you are designing the software for a simple, embedded real-time system. What I expect of my students is a set of class diagrams, sequence diagrams, and state machines.

A Common but Hideous Solution

By far the most common solution that my students present is the one in Figure 20-1. In this diagram, the central `CoffeeMaker` class is 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, two base classes, `Sensor` and `Heater`, act as parents to the `Boiler` and `WarmerPlate` elements, respectively.

Figure 20-1. Hyperconcrete coffee maker

[View full size image]



It is difficult for beginners to appreciate just how hideous this structure is. Quite a few rather serious errors are lurking in this diagram. Many of them would not be noticed until you 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 20-1 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 they create diagrams without methods, designers may be partitioning the software on something other than behavior. Partitionings that are not based on 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

If we consider the methods we might put in the class `Light` we can see how poorly partitioned this particular design is. Clearly, the `Light` object 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 20-2.

Listing 20-2. `Light.cs`

```
public class Light {
  public void On() {
    CoffeeMaker.api.SetIndicatorState(IndicatorState.ON);
  }

  public void Off() {
    CoffeeMaker.api.SetIndicatorState(IndicatorState.OFF);
  }
}
```

Class `Light` has some peculiarities. 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`. Apparently, the `Light` class is nothing more than a call translator and is not 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 20-1, to mere placeholders without much reason to exist. For this reason, I call them *vapor classes*.

Imaginary Abstraction

Note the `Sensor` and `Heater` base classes in Figure 20-1. The previous section should have convinced you that their derivatives were mere vapor, but what about the base classes themselves? On the surface, they seem to make a lot of sense. But, there doesn't seem to be any place for their derivatives.

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, *Who uses them?*

No class in the system 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 its 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 20-3. A class with nothing but abstract functions and that no other class uses is officially useless.

Listing 20-3. `Heater.cs`

```
public interface Heater {
 void TurnOn();
 void TurnOff();
}
```

The `Sensor` class (Listing 20-4) 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 20-4. **`Sensor.cs`**

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

What 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, 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 the abode of gods in disguise. Figure 20-1 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 implement those classes, we find that only one of those classes, `CoffeeMaker`, has any interesting behavior; the rest are all imaginary abstractions or vapor classes.

An Improved 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 (or any) problem is to step back and separate its details from its essential nature. Forget about boilers, valves, heaters, sensors, and all the little details; concentrate on the underlying problem. What is that problem? The problem is: How do you make coffee?

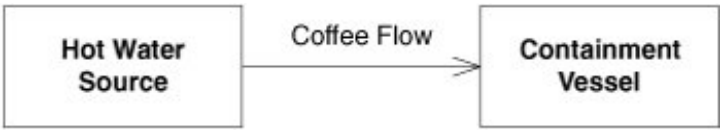
How *do* you make coffee? The simplest, most common solution to this problem is to pour hot water over coffee grounds and to collect the resulting infusion in some kind of vessel. Where do we get the hot water from? Let's call it a `HotWaterSource`. Where do we collect the coffee? Let's call it a `ContainmentVessel`.[2]

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

Are these two abstractions 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 for letting us know whether any coffee was left in the vessel.

How would you capture the previous discussion in a UML diagram? Figure 20-2 shows one possible schema. `HotWaterSource` and `ContainmentVessel` are both represented as classes and are associated by the flow of coffee.

Figure 20-2. Crossed wires



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 `Containment-Vessel` 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 20-3. Note that the `ContainmentVessel` is sending the `Start` message to the `HotWaterSource`. This means that the association in Figure 20-2 is backward. `HotWaterSource` does not depend on the `ContainmentVessel` at all. Rather, the `ContainmentVessel` depends on the `HotWaterSource`.

Figure 20-3. Starting the flow of hot water

The lesson here is simply this: Associations are the pathways through which messages are sent between objects. Associations 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

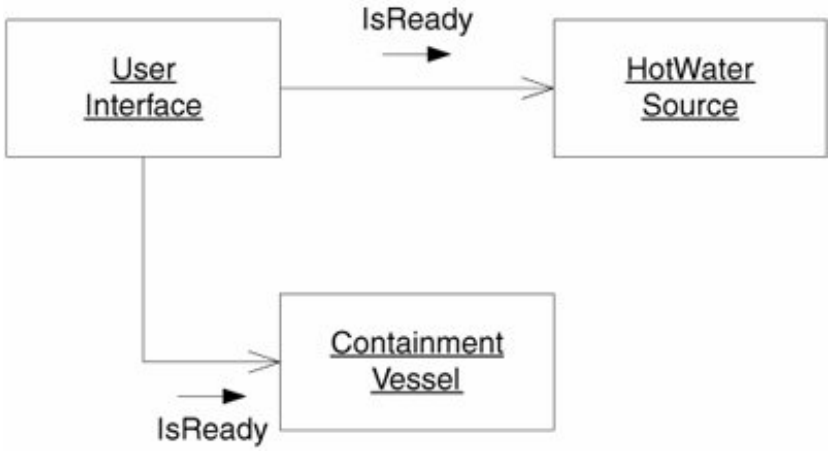
OK, given these three classes, how do their instances communicate? Let's look at several use cases to see whether we can find out what the behavior of these classes is.

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 `UserInterface` object first sends a message to the `HotWaterSource` and the `ContainmentVessel` to see whether they are ready. This is shown in Figure 20-4.

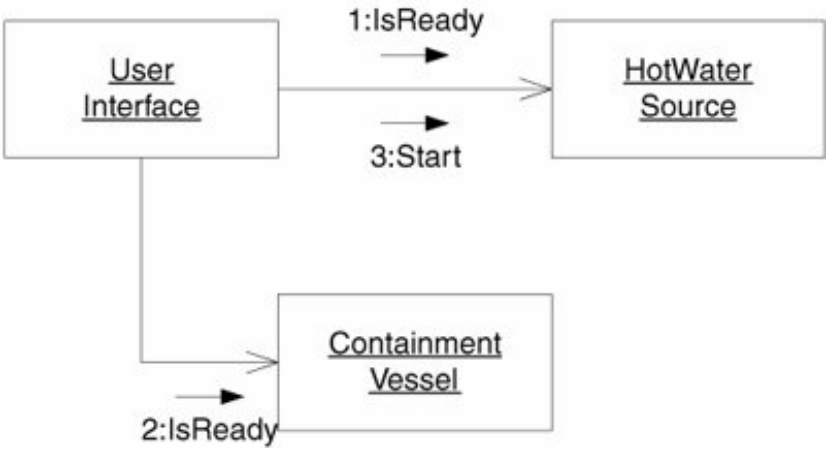
Figure 20-4. Brew button pressed, checking for ready



If either of these queries returns `false`, we refuse to start brewing coffee. The `UserInterface` object can take care of letting the user know that his or her 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. The `UserInterface` object should probably 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 20-5 shows the completed scenario.

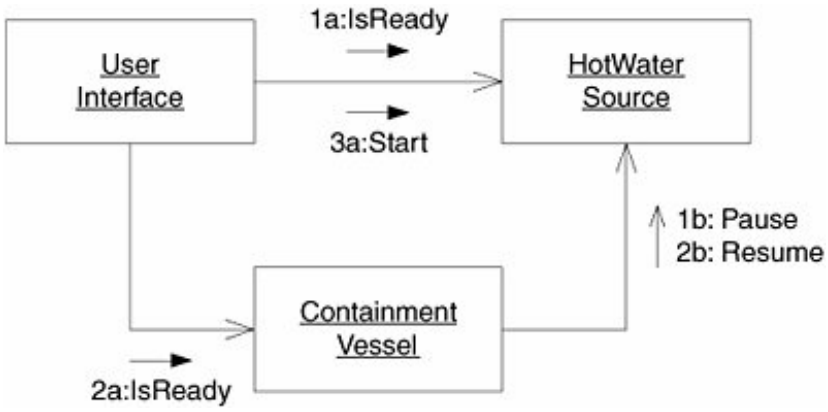
Figure 20-5. Brew button pressed, complete



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 `ContainmentVessel` 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 20-6 adds the new methods.

Figure 20-6. Pausing and resuming the flow of hot water



Use case 3: Brewing complete

At some point, we will be done brewing coffee and will 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 difficult to

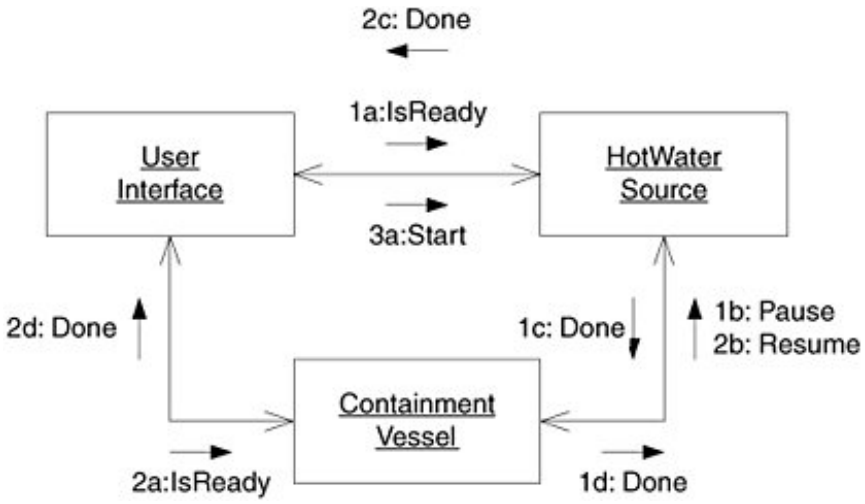
envision a coffee maker in which the `ContainmentVessel` would be the one to detect that brewing was done. For example, what if our coffee maker was plumbed into the water mains and therefore had an infinite supply of water? What if an intense microwave generator heated the water as it flowed through the pipes into a thermally isolated vessel?[3] What if that vessel had a spigot from which users got their coffee? In this case, a sensor in the vessel would know that it was full and that hot water should be shut off.

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

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 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? Does the `ContainmentVessel` need to do or to keep track of anything special once the brewing is complete? In the Mark IV, it's going to detect an empty pot being put back on the plate, signaling that the user has poured the last of the coffee. This causes the Mark IV to turn the light *off*. 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 20-7 shows the new messages. Note that I've shown that either `HotWaterSource` or `ContainmentVesslel` can send the `Done` message.

Figure 20-7. Detecting when brewing is complete

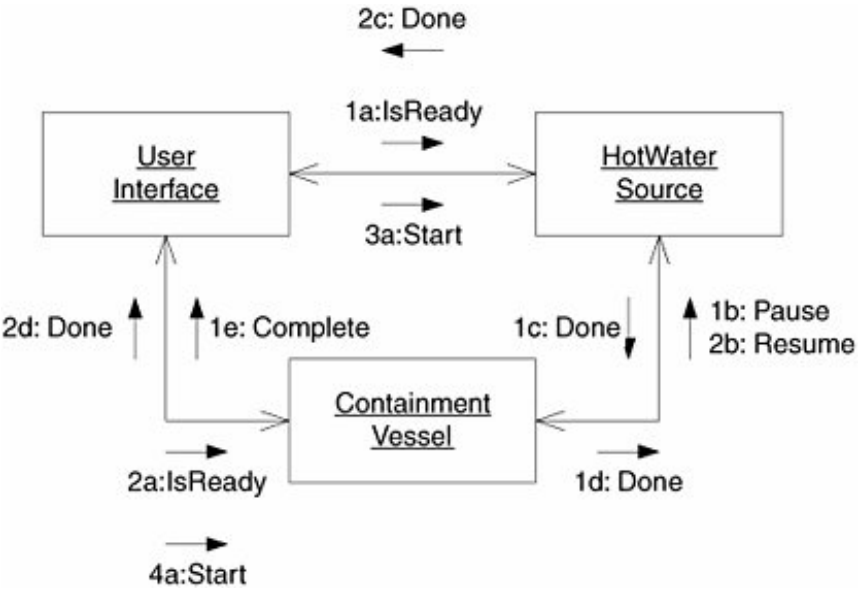


Use case 4: Coffee all gone

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

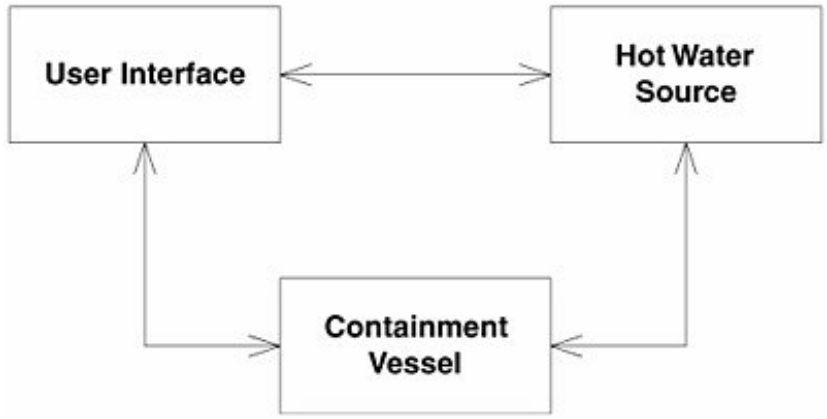
`Complete` message to the `UserInterface`. Figure 20-8 shows the completed collaboration diagram.

Figure 20-8. Coffee all gone



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 20-9.

Figure 20-9. Class diagram



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 does there appear to be any vapor classes.

So far, so good, but how do we implement the Mark IV in this structure? Do we simply implement the methods of these three classes to invoke the `CoffeeMakerAPI`? This 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 on the low-level implementation.

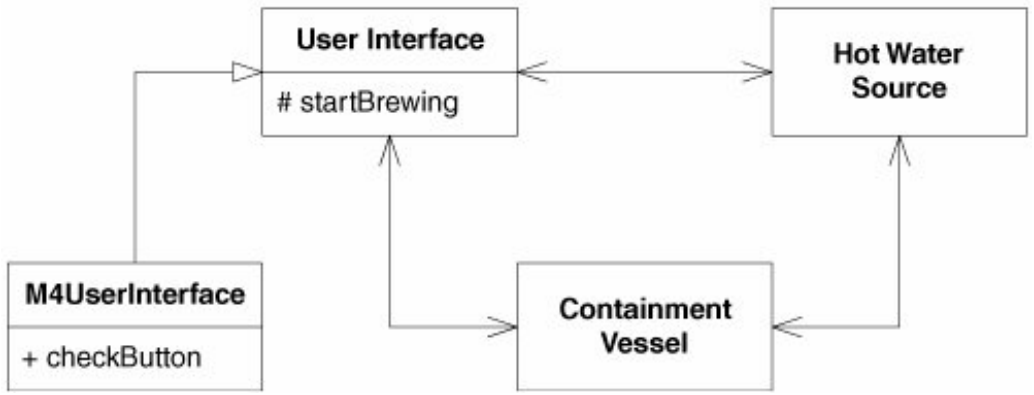
OK, then, how will we create the Mark IV implementation? Let's look at all the use cases again. But this time, let's look at them from the Mark IV point of view.

Use case 1: User pushes Brew button

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 DIP and put the call in a derivative of `UserInterface`. See Figure 20-10 for details.

Figure 20-10. Detecting the Brew button



We've derived `M4UserInterface` from `UserInterface`, and we've put a `Check-Button()` method in `M4UserInterface`. When this function is called, it will call the `CoffeeMakerAPI.GetBrewButtonStatus()` function. If the button has been pressed, the fuction will invoke the protected `StartBrewing()` method of `UserInterface`. Listings 20-5 and 20-6 show how this would be coded.

Listing 20-5. **M4UserInterface.cs**

```
public class M4UserInterface : UserInterface
{
  private void CheckButton()
  {
    BrewButtonStatus status =
      CoffeeMaker.api.GetBrewButtonStatus();
    if (status == BrewButtonStatus.PUSHED)
    {
      StartBrewing();
    }
  }
}
```

Listing 20-6. **UserInterface.cs**

```
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();
    }
  }
}
```

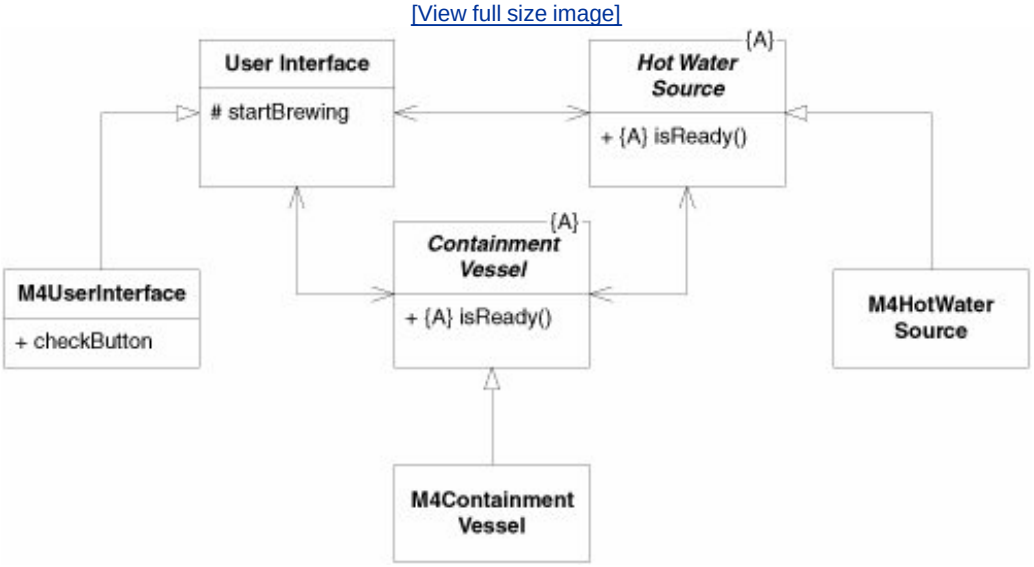
You might be wondering why I created the protected `StartBrewing()` method at all. Why didn't I simply call the `Start()` functions from `M4UserInterface`? The reason is simple but significant. The `IsReady()` tests and the consequential calls to the `Start()` methods of the `HotWaterSource` and the `ContainmentVessel` are highlevel 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. This is yet another example of the Single-Responsibility Principle (SRP). 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, 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 only 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 20-11 shows the new structure, and Listings 20-7 and 20-8 show the implementation of the two derivatives.

Figure 20-11. Implementing the **`isReady`** methods

[View full size image]



Listing 20-7. **`M4HotWaterSource.cs`**

```
public class M4HotWaterSource : HotWaterSource
{
  public override bool IsReady()
  {
    BoilerStatus status =
      CoffeeMaker.api.GetBoilerStatus();
    return status == BoilerStatus.NOT_EMPTY;
  }
}
```

Listing 20-8. **`M4ContainmentVessel.cs`**

```
public class M4ContainmentVessel : ContainmentVessel
{
  public override bool IsReady()
  {
    WarmerPlateStatus status =
      CoffeeMaker.api.GetWarmerPlateStatus();
    return status == WarmerPlateStatus.POT_EMPTY;
  }
}
```

Implementing the `Start()` functions

The `Start()` method of `HotWaterSource` is simply 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 `CoffeeMaker.api.XXX` structures I was writing, so I did a little refactoring at the same time. The result is in Listing 20-9.

Listing 20-9. **`M4HotWaterSource.cs`**

```
public class M4HotWaterSource : HotWaterSource
{
  private CoffeeMakerAPI api;

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

  public override bool IsReady()
  {
    BoilerStatus status = api.GetBoilerStatus();
    return status == BoilerStatus.NOT_EMPTY;
  }

  public override void Start()
  {
    api.SetReliefValveState(ReliefValveState.CLOSED);
    api.SetBoilerState(BoilerState.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 20-10 shows the code.

Listing 20-10. **M4ContainmentVessell.cs**

```
public class M4ContainmentVessel : ContainmentVessel
{
  private CoffeeMakerAPI api;
  private bool isBrewing = false;

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

  public override bool IsReady()
  {
    WarmerPlateStatus status = api.GetWarmerPlateStatus();
    return status == WarmerPlateStatus.POT_EMPTY;
  }

  public override void Start()
  {
    isBrewing = true;
  }
}
```

Calling `M4UserInterface.CheckButton`

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 that 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 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.

These teams' mistakewhich I eventually point out to them after letting them sweat a bitis 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 platform that does not support threading. This means that we're going to have to poll. How can we make this work?

Consider the `Pollable` interface in Listing 20-11. This interface has nothing but a `Poll()` method. What if `M4UserInterface` implemented this interface? What if the `Main()` program hung in a hard loop,

calling this method over and over again? Then the flow of control would continuously be reentering `M4UserInterface`, and we could detect the Brew button.

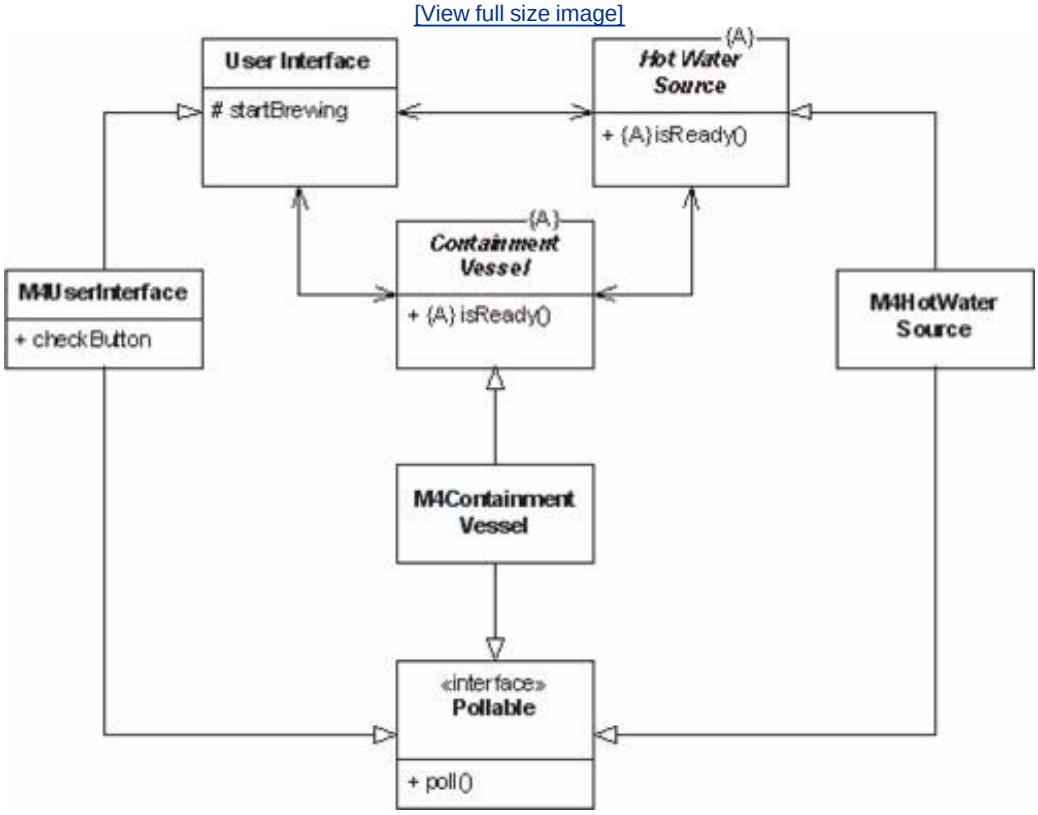
Listing 20-11. `Pollable.cs`

```
public interface Pollable
{
  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 20-12, we can derive all the M4 derivatives from `Pollable` and call them all from `Main()`.

Figure 20-12. Pollable coffee maker

[View full size image]



Listing 20-12 shows what the `Main` function might look like. It is placed in a class called `M4CoffeeMaker`. 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 20-12. **M4CoffeeMaker.cs**

```
public static void Main(string[] args)
{
  CoffeeMakerAPI api = new M4CoffeeMakerAPI();
  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(hws,ui);

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

It should now 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 20-13 shows what `M4UserInterface` looks like now.

Listing 20-13. **M4UserInterface.cs**

```
public class M4UserInterface : UserInterface
                             , Pollable
{
  private CoffeeMakerAPI api;

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

  public void Poll()
  {
    BrewButtonStatus status = api.GetBrewButtonStatus();
    if (status == BrewButtonStatus.PUSHED)
    {
      StartBrewing();
    }
  }
```

```
}
```

Completing the coffee maker

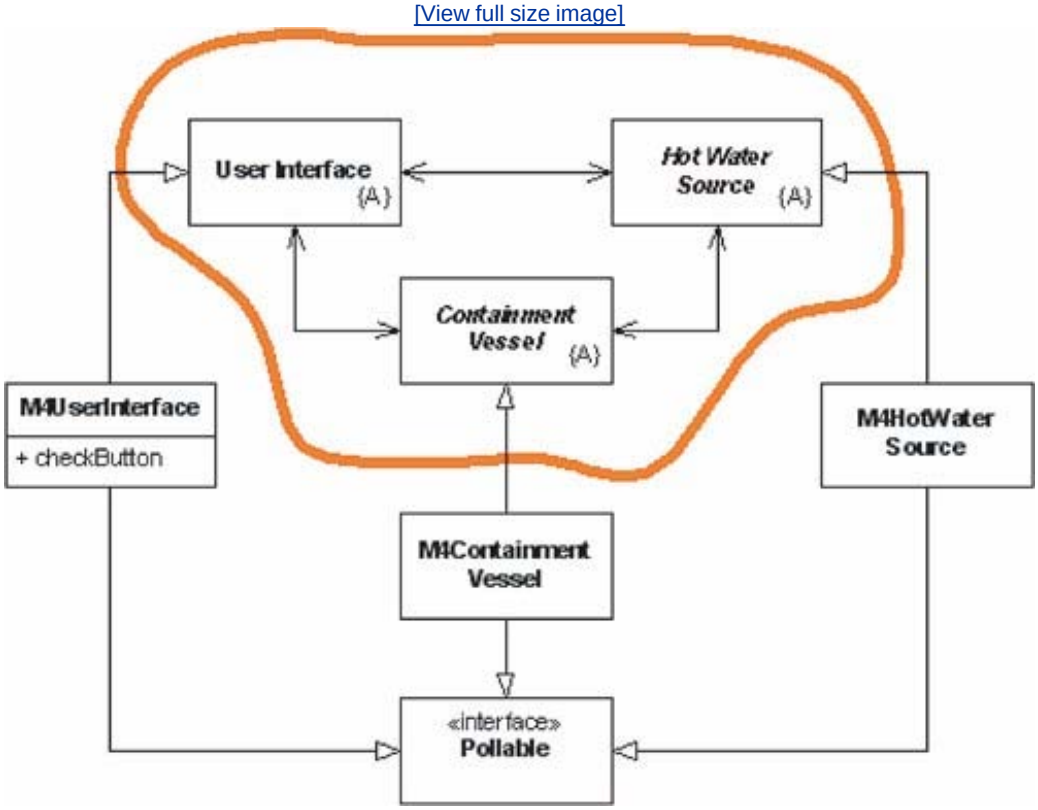
The reasoning used in the previous sections can be repeated for each of the other components of the coffee maker. The result is shown in Listings 20-14 through 20-21.

The Benefits of This Design

Despite the trivial nature of the problem, this design shows some very nice characteristics. Figure 20-13 shows the structure. I have drawn a line around the three abstract classes. These classes hold the high-level policy of the coffee maker. Note that all dependencies that cross the line point inward. Nothing inside the line depends on anything outside. Thus, the abstractions are completely separated from the details.

Figure 20-13. Coffee maker components

[View full size image]



The abstract classes know nothing of buttons, lights, valves, sensors, or any other of the detailed

elements of the coffee maker. By the same token, the derivatives are dominated by those details.

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.

The Roots of This Design

I did not simply sit down one day and develop this design in a nice straightfoward manner. Indeed, in 1993, my first design for the coffee maker looked much more like Figure 20-1. 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 was created, test first, using the unit tests in Listing 20-22. I created the code, based on the structure in Figure 20-13, 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 of the test cases. However, I felt that such an analysis would have been overkill for this book.

OOverkill

This example has certain pedagogical advantages. It is small and 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 an FSM, we'd find that it had 7 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.

[5] [Martin1995], p. 65

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. In this example, however, the reverse is more likely to be true.

Listing 20-14. **UserInterface.cs**

```
using System;

namespace CoffeeMaker
{
  public abstract class UserInterface
  {
    private HotWaterSource hws;
    private ContainmentVessel cv;
    protected bool 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 20-15. **M4UserInterface.cs**

```
using CoffeeMaker;

namespace M4CoffeeMaker
{
  public class M4UserInterface : UserInterface
                               , Pollable
  {
    private CoffeeMakerAPI api;

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

    public void Poll()
    {
      BrewButtonStatus buttonStatus = api.GetBrewButtonStatus();
      if (buttonStatus == BrewButtonStatus.PUSHED)
      {
        StartBrewing();
      }
    }
    public override void Done()
    {
      api.SetIndicatorState(IndicatorState.ON);
    }

    public override void CompleteCycle()
    {
```

```
      api.SetIndicatorState(IndicatorState.OFF);
    }
  }
}
```

Listing 20-16. **HotWaterSource.cs**

```
namespace CoffeeMaker
{
  public abstract class HotWaterSource
  {
    private UserInterface ui;
    private ContainmentVessel cv;
    protected bool 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 bool IsReady();
    public abstract void StartBrewing();
    public abstract void Pause();
    public abstract void Resume();
```

```
  }
}
```

Listing 20-17. **M4HotWaterSource.cs**

```
using System;
using CoffeeMaker;

namespace M4CoffeeMaker
{
  public class M4HotWaterSource : HotWaterSource
                                , Pollable
  {
    private CoffeeMakerAPI api;

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

    public override bool IsReady()
    {
      BoilerStatus boilerStatus = api.GetBoilerStatus();
      return boilerStatus == BoilerStatus.NOT_EMPTY;
    }

    public override void StartBrewing()
    {
      api.SetReliefValveState(ReliefValveState.CLOSED);
      api.SetBoilerState(BoilerState.ON);
    }

    public void Poll()
    {
      BoilerStatus boilerStatus = api.GetBoilerStatus();
      if (isBrewing)
      {
        if (boilerStatus == BoilerStatus.EMPTY)
        {
          api.SetBoilerState(BoilerState.OFF);
          api.SetReliefValveState(ReliefValveState.CLOSED);
          DeclareDone();
        }
      }
    }

    public override void Pause()
    {
```

```
      api.SetBoilerState(BoilerState.OFF);
      api.SetReliefValveState(ReliefValveState.OPEN);
    }

    public override void Resume()
    {
      api.SetBoilerState(BoilerState.ON);
      api.SetReliefValveState(ReliefValveState.CLOSED);
    }
  }
}
```

Listing 20-18. **ContainmentVessel.cs**

```
using System;

namespace CoffeeMaker
{
  public abstract class ContainmentVessel
  {
    private UserInterface ui;
    private HotWaterSource hws;
    protected bool isBrewing;
    protected bool 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;
    }
```

```
    protected void DeclareComplete()
    {
      isComplete = true;
      ui.Complete();
    }
    protected void ContainerAvailable()
    {
      hws.Resume();
    }

    protected void ContainerUnavailable()
    {
      hws.Pause();
    }

    public abstract bool IsReady();
  }
}
```

Listing 20-19. **M4ContainmentVessel.cs**

```
using CoffeeMaker;

namespace M4CoffeeMaker
{
  public class M4ContainmentVessel : ContainmentVessel
                                   , Pollable
  {
    private CoffeeMakerAPI api;
    private WarmerPlateStatus lastPotStatus;

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

    public override bool IsReady()
    {
      WarmerPlateStatus plateStatus =
                        api.GetWarmerPlateStatus();
      return plateStatus == WarmerPlateStatus.POT_EMPTY;
    }

    public void Poll()
    {
      WarmerPlateStatus potStatus = api.GetWarmerPlateStatus();
      if (potStatus != lastPotStatus)
```

```
      {
        if (isBrewing)
        {
          HandleBrewingEvent(potStatus);
        }
        else if (isComplete == false)
        {
          HandleIncompleteEvent(potStatus);
        }
        lastPotStatus = potStatus;
      }
    }

    private void
    HandleBrewingEvent(WarmerPlateStatus potStatus)
    {
      if (potStatus == WarmerPlateStatus.POT_NOT_EMPTY)
      {
        ContainerAvailable();
        api.SetWarmerState(WarmerState.ON);
      }
      else if (potStatus == WarmerPlateStatus.WARMER_EMPTY)
      {
        ContainerUnavailable();
        api.SetWarmerState(WarmerState.OFF);
      }
      else
      { // potStatus == POT_EMPTY
        ContainerAvailable();
        api.SetWarmerState(WarmerState.OFF);
      }
    }

    private void
    HandleIncompleteEvent(WarmerPlateStatus potStatus)
    {
      if (potStatus == WarmerPlateStatus.POT_NOT_EMPTY)
      {
        api.SetWarmerState(WarmerState.ON);
      }
      else if (potStatus == WarmerPlateStatus.WARMER_EMPTY)
      {
        api.SetWarmerState(WarmerState.OFF);
      }
      else
      { // potStatus == POT_EMPTY
        api.SetWarmerState(WarmerState.OFF);
        DeclareComplete();
      }
    }
  }
}
```

Listing 20-20. **Pollable.cs**

```
using System;

namespace M4CoffeeMaker
{
  public interface Pollable
  {
    void Poll();
  }
}
```

Listing 20-21. **CoffeeMaker.cs**

```
using CoffeeMaker;

namespace M4CoffeeMaker
{
  public class M4CoffeeMaker
  {
    public static void Main(string[] args)
    {
      CoffeeMakerAPI api = new M4CoffeeMakerAPI();
      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 20-22. **TestCoffeeMaker.cs**

```
using M4CoffeeMaker;
using NUnit.Framework;

namespace CoffeeMaker.Test
{
  internal class CoffeeMakerStub : CoffeeMakerAPI
  {
    public bool buttonPressed;
    public bool lightOn;
    public bool boilerOn;
    public bool valveClosed;
    public bool plateOn;
    public bool boilerEmpty;
    public bool potPresent;
    public bool potNotEmpty;

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

    public WarmerPlateStatus GetWarmerPlateStatus()
    {
      if (!potPresent)
        return WarmerPlateStatus.WARMER_EMPTY;
      else if (potNotEmpty)
        return WarmerPlateStatus.POT_NOT_EMPTY;
      else
        return WarmerPlateStatus.POT_EMPTY;
    }

    public BoilerStatus GetBoilerStatus()
    {
      return boilerEmpty ?
             BoilerStatus.EMPTY : BoilerStatus.NOT_EMPTY;
    }

    public BrewButtonStatus GetBrewButtonStatus()
    {
      if (buttonPressed)
      {
        buttonPressed = false;
        return BrewButtonStatus.PUSHED;
      }
```

```
    else
    {
      return BrewButtonStatus.NOT_PUSHED;
    }
  }

  public void SetBoilerState(BoilerState boilerState)
  {
    boilerOn = boilerState == BoilerState.ON;
  }

  public void SetWarmerState(WarmerState warmerState)
  {
    plateOn = warmerState == WarmerState.ON;
  }
  public void
  SetIndicatorState(IndicatorState indicatorState)
  {
    lightOn = indicatorState == IndicatorState.ON;
  }

  public void
  SetReliefValveState(ReliefValveState reliefValveState)
  {
    valveClosed = reliefValveState == ReliefValveState.CLOSED;
  }
}

[TestFixture]
public class TestCoffeeMaker
{
  private M4UserInterface ui;
  private M4HotWaterSource hws;
  private M4ContainmentVessel cv;
  private CoffeeMakerStub api;

  [SetUp]
  public void SetUp()
  {
    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();
  }

  [Test]
  public void InitialConditions()
  {
    Poll();
    Assert.IsFalse(api.boilerOn);
    Assert.IsFalse(api.lightOn);
    Assert.IsFalse(api.plateOn);
    Assert.IsTrue(api.valveClosed);
  }

  [Test]
  public void StartNoPot()
  {
    Poll();
    api.buttonPressed = true;
    api.potPresent = false;
    Poll();
    Assert.IsFalse(api.boilerOn);
    Assert.IsFalse(api.lightOn);
    Assert.IsFalse(api.plateOn);
    Assert.IsTrue(api.valveClosed);
  }

  [Test]
  public void StartNoWater()
  {
    Poll();
    api.buttonPressed = true;
    api.boilerEmpty = true;
    Poll();
    Assert.IsFalse(api.boilerOn);
    Assert.IsFalse(api.lightOn);
    Assert.IsFalse(api.plateOn);
    Assert.IsTrue(api.valveClosed);
  }

  [Test]
  public void GoodStart()
  {
    NormalStart();
    Assert.IsTrue(api.boilerOn);
    Assert.IsFalse(api.lightOn);
    Assert.IsFalse(api.plateOn);
    Assert.IsTrue(api.valveClosed);
  }

  private void NormalStart()
  {
    Poll();
```

```
    api.boilerEmpty = false;
    api.buttonPressed = true;
    Poll();
  }

  [Test]
  public void StartedPotNotEmpty()
  {
    NormalStart();
    api.potNotEmpty = true;
    Poll();
    Assert.IsTrue(api.boilerOn);
    Assert.IsFalse(api.lightOn);
    Assert.IsTrue(api.plateOn);
    Assert.IsTrue(api.valveClosed);
  }

  [Test]
  public void PotRemovedAndReplacedWhileEmpty()
  {
    NormalStart();
    api.potPresent = false;
    Poll();
    Assert.IsFalse(api.boilerOn);
    Assert.IsFalse(api.lightOn);
    Assert.IsFalse(api.plateOn);
    Assert.IsFalse(api.valveClosed);

    api.potPresent = true;
    Poll();
    Assert.IsTrue(api.boilerOn);
    Assert.IsFalse(api.lightOn);
    Assert.IsFalse(api.plateOn);
    Assert.IsTrue(api.valveClosed);
  }

  [Test]
  public void PotRemovedWhileNotEmptyAndReplacedEmpty()
  {
    NormalFill();
    api.potPresent = false;
    Poll();
    Assert.IsFalse(api.boilerOn);
    Assert.IsFalse(api.lightOn);
    Assert.IsFalse(api.plateOn);
    Assert.IsFalse(api.valveClosed);

    api.potPresent = true;
    api.potNotEmpty = false;
    Poll();
    Assert.IsTrue(api.boilerOn);
    Assert.IsFalse(api.lightOn);
```

```
    Assert.IsFalse(api.plateOn);
    Assert.IsTrue(api.valveClosed);
  }

  private void NormalFill()
  {
    NormalStart();
    api.potNotEmpty = true;
    Poll();
  }

  [Test]
  public void PotRemovedWhileNotEmptyAndReplacedNotEmpty()
  {
    NormalFill();
    api.potPresent = false;
    Poll();
    api.potPresent = true;
    Poll();
    Assert.IsTrue(api.boilerOn);
    Assert.IsFalse(api.lightOn);
    Assert.IsTrue(api.plateOn);
    Assert.IsTrue(api.valveClosed);
  }
  [Test]
  public void BoilerEmptyPotNotEmpty()
  {
    NormalBrew();
    Assert.IsFalse(api.boilerOn);
    Assert.IsTrue(api.lightOn);
    Assert.IsTrue(api.plateOn);
    Assert.IsTrue(api.valveClosed);
  }

  private void NormalBrew()
  {
    NormalFill();
    api.boilerEmpty = true;
    Poll();
  }


  [Test]
  public void BoilerEmptiesWhilePotRemoved()
  {
    NormalFill();
    api.potPresent = false;
    Poll();
    api.boilerEmpty = true;
    Poll();
    Assert.IsFalse(api.boilerOn);
    Assert.IsTrue(api.lightOn);
```

```
      Assert.IsFalse(api.plateOn);
      Assert.IsTrue(api.valveClosed);

      api.potPresent = true;
      Poll();
      Assert.IsFalse(api.boilerOn);
      Assert.IsTrue(api.lightOn);
      Assert.IsTrue(api.plateOn);
      Assert.IsTrue(api.valveClosed);
    }

    [Test]
    public void EmptyPotReturnedAfter()
    {
      NormalBrew  ();
      api  .
      potNotEmpty  = false;
      Poll  ();
      Assert.IsFalse(api.boilerOn);
      Assert.IsFalse(api.lightOn);
      Assert.IsFalse(api.plateOn);
      Assert.IsTrue(api.valveClosed);
    }
  }
}
```

PREV NEXT