# The Mark IV Special Coffee Maker

## 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.

[GetBoilerStatus, SetBoilerState]

* The heating element for the **warmer plate**. It can be turned on or off.

[GetWarmerPlateStatus, SetWarmerState]

* 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.

[GetBrewButtonStatus, SetIndicatorState]

* 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.

[SetReliefValveState]

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.

## 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

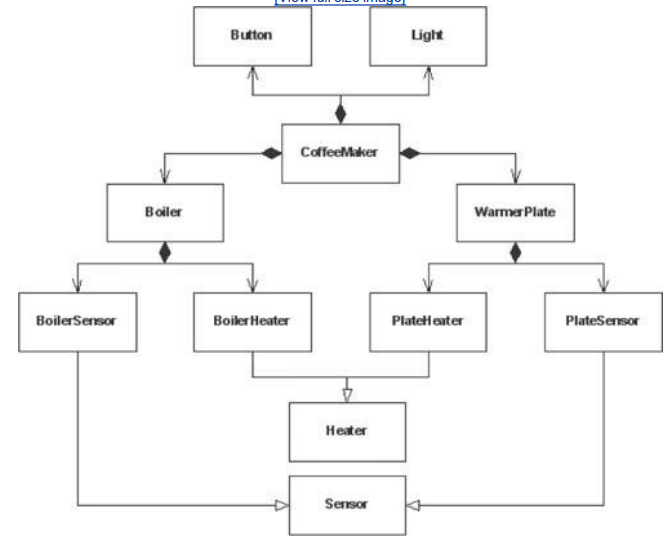
* a Boiler, The Boiler contains
  + a BoilerSensor: **Sensor** and
  + a BoilerHeater: **Heater**.
* a WarmerPlate, The WarmerPlate contains
  + a PlateSensor: **Sensor** and
  + a PlateHeater: **Heater**.
* a Button, and
* a Light.

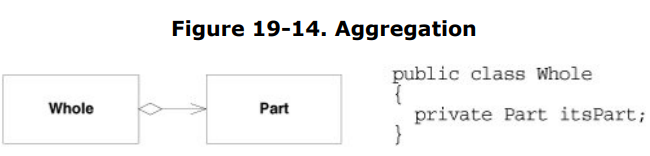
Finally, two base classes, **Sensor** and **Heater**, act as parents to the Boiler and WarmerPlate elements, respectively.

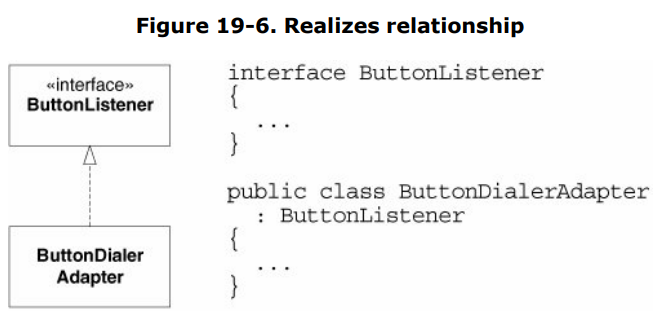
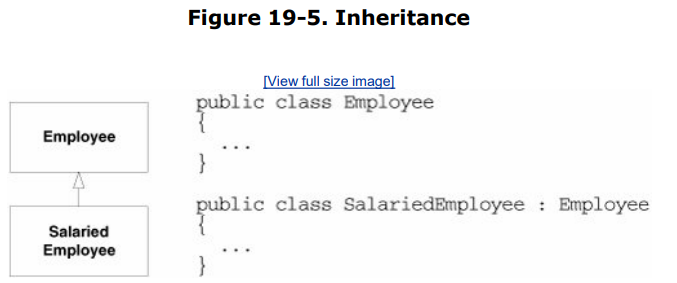
**The biggest problem** that Figure 20-1 exhibits is a complete lack of methods. We are writing a program here, and programs are about behavior! **Where is the behavior in this diagram?**

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**.

Figure 20-1. **Hyperconcrete** coffee maker



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.**

Class Light, 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. For this reason, I call them vapor classes.

Note the Sensor and Heater base classes. **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?

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.

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**.

## An Improved Solution

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**.

*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?*

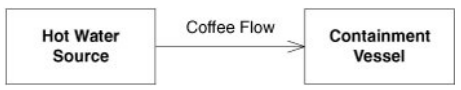
If we think about the Mark IV unit, we could imagine **the boiler, valve, and boiler sensor** playing the role of the **HotWaterSource**.

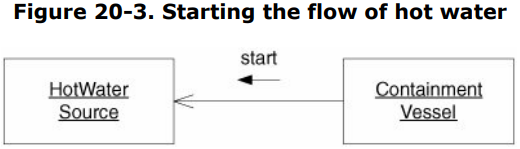
The **HotWaterSource** would be **responsible for** heating the water and delivering it over the coffee grounds to drip into the **ContainmentVessel**.

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.**

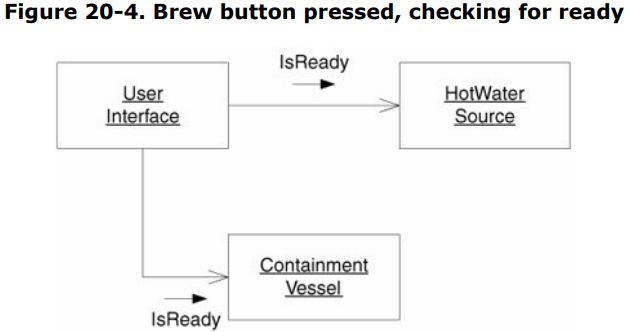
The lesson here is simply this**: Associations are the pathways through which messages are sent between objects.** Associations have nothing to do with the flow of physical objects. The fact that hot water flows from the boiler to the pot does not mean that there should be an association from the HotWaterSource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

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

*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

**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.**

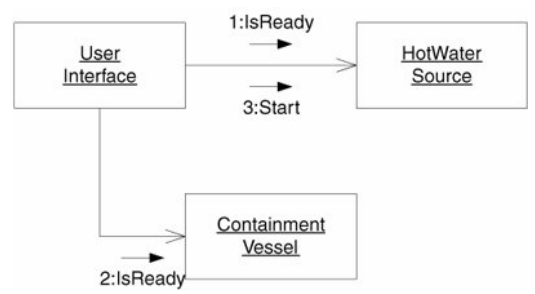
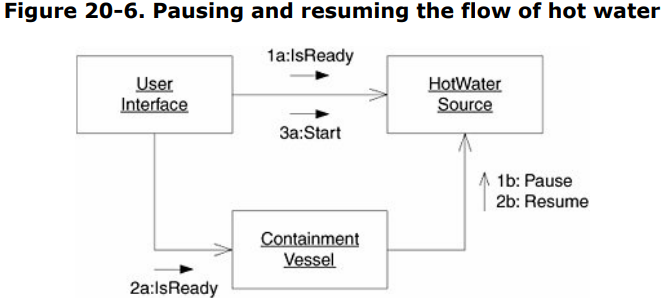
**The UserInterface object should probably send a Start message to the HotWaterSource.** The HotWaterSource will then start doing whatever it needs to do to get hot water flowing. In the case of the Mark IV, it will close the valve and turn on the boiler. Figure 20-5 shows the completed scenario.

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.



### Use case 3: Brewing complete

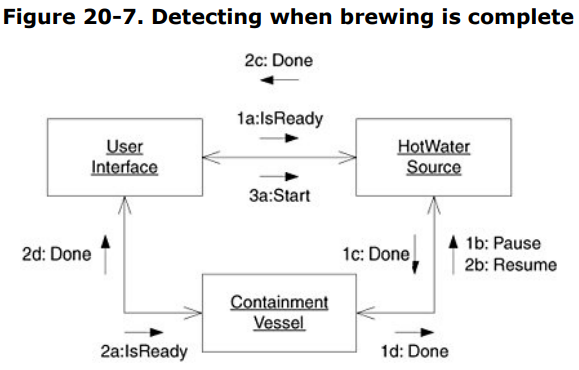
At some point, we will be done brewing coffee and will have to turn off the flow of hot water.

*Which one of our objects knows when brewing is complete?* 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.

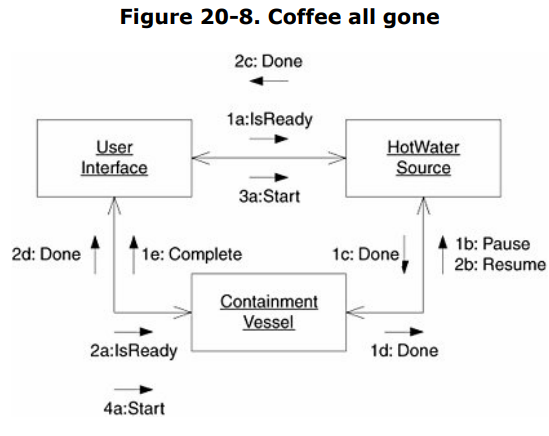
It should also be clear that the HotWaterSource needs to know that brewing is over, because it'll need to stop the flow of hot water. In the Mark IV, it'll shut down the boiler and open the valve.

*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.**

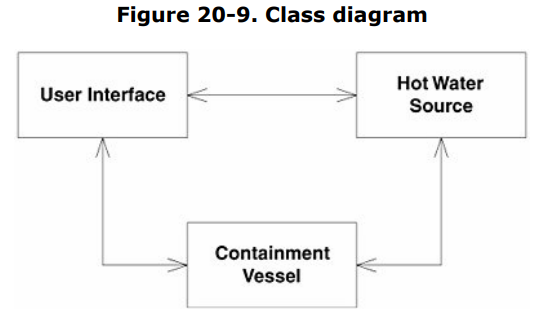


### Use case 4: Coffee all gone

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.



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.

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 **no**t appear to be a **god object** anywhere. Nor does there appear to be **any vapor classes**.

## Implementing the Abstract Model

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!

**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.

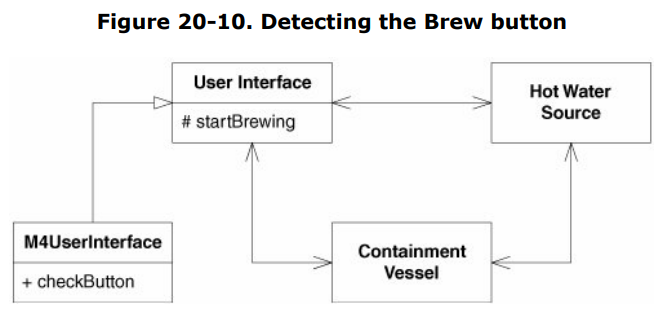
OK, then, how will we create the Mark IV implementation? Let's look at all the use cases again. But this time, let's look at them from the Mark IV point of view.

### 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.

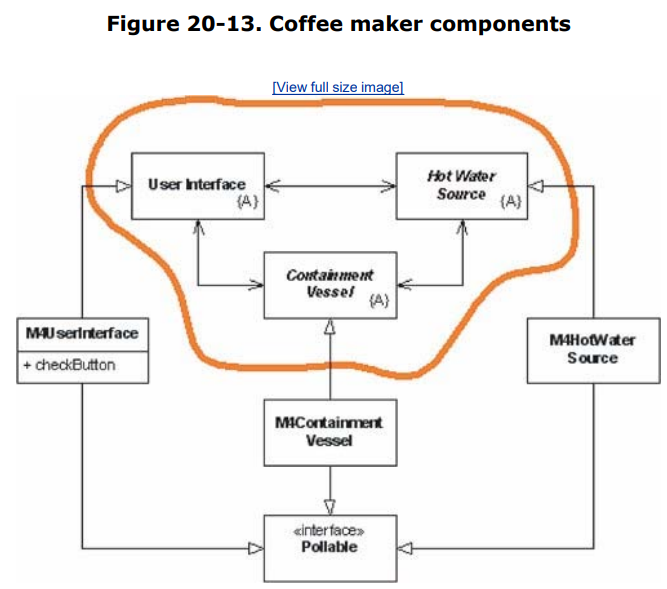
*Where should it call this function?* We've already decreed that the **UserInterface class itself cannot know about the CoffeeMakerAPI.** So where does this call go? **We'll apply DIP and put the call in a derivative of UserInterface.** See Figure 20-10 for details.

**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 function will invoke the protected StartBrewing() method of UserInterface.



*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.**

*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.



## 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.

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.