OOAD Design Problem: The Coffee Maker

Jim Weirich

Introduction

This paper describes a design for the Coffee Maker problem from Robert Martin’s "*First Principles of OOD and UML"* course. The problem is also described in Martin’s book "*Designing Object Oriented C++ Applications using the Booch Methodology*".

The Extreme Programming web site is also working through this example, but using XP techniques rather than the up-front design that I have used here. It will be interesting to compare the results.

A browsable Java implementation of this design is available online (as well as its Java docs). There is a demonstration Java applet that lets you play with the coffee maker. Finally, I wrote a small description of the process of writing the Java version.

This is a slightly updated version of the Coffee Maker design. I use this design in as a class room example and it has been revised many times. The original version I first presented on the web is still availble.

Problem Summary

(From *Designing Object Oriented C++ Applications using the Booch Methodology* by Robert Martin).

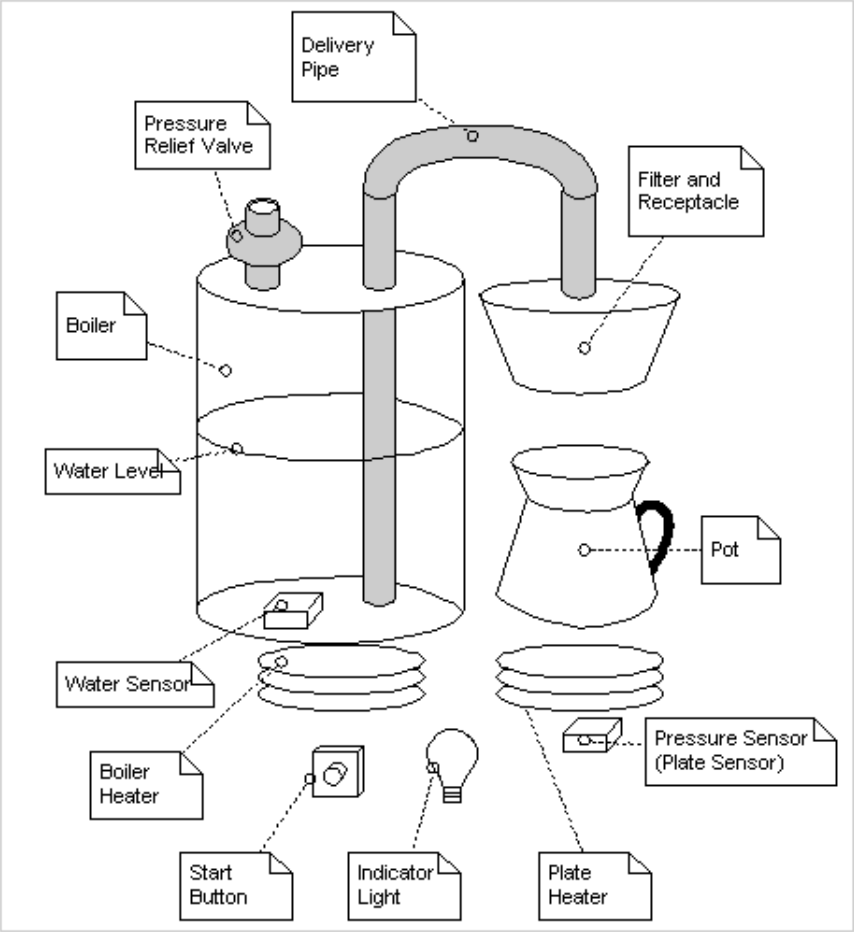

Figure 1 -- Coffee Maker Schematic

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 coffee is 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**, and **potNotEmpty**.
- A sensor for the boiler, which determines if there is water present or not. It has two states: **boilerEmpty** or **boilerNotEmtpy**.
- 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. It can be opened or closed.

A hardware interface is given in Appendix A.

Understanding the Problem

We will start out is a simple UML diagram capturing the relationships of the various physical elements of the problem. Note that this is a description of the physical relationships between the coffee maker components and may have little to do with the final software configuration.

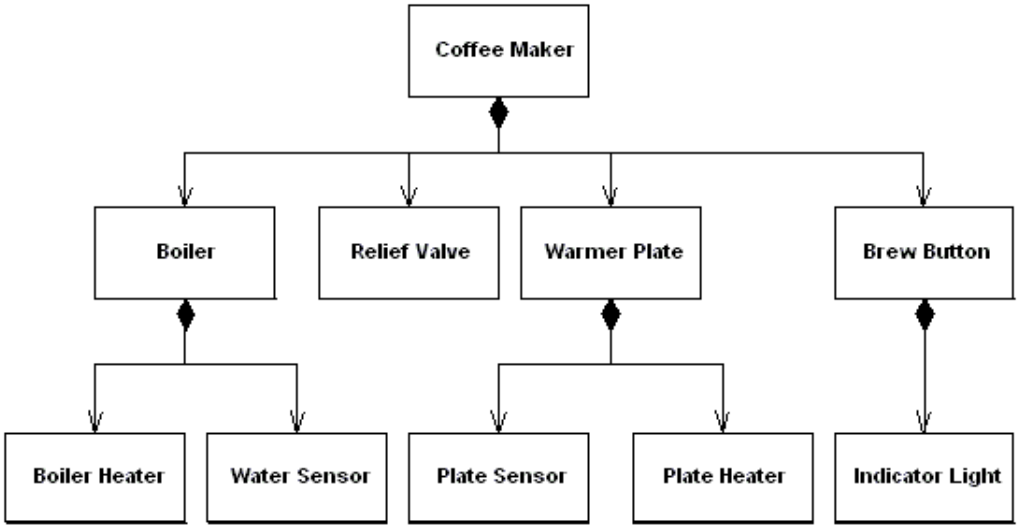

Figure 2 - Physcial Component Identification

Use Cases

Lets look more closely at what the software must accomplish. We will capture these requirements is a set of Use Cases.

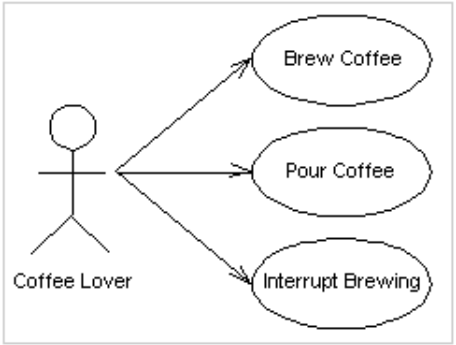

Figure 3 - Use Cases

Use Case: Brew Coffee

Let’s tackle the most straight forward use of the coffee maker first. This would be the normal use case

first.

In "Brew Coffee" the user (Coffee Lover) starts a pot of coffee brewing. The Coffee Lover must fill the coffee maker with the correct ingredients (coffee grounds in filter, water in boiler, pot on warmer). Then he pushes the brew button and waits for the coffee to finish brewing.

Use Case: Brew Coffee
System: Coffee Maker
Actors: Coffee Lover
Precondition: None

Step	Actor (Coffee Lover)	System (Coffee Maker)
1	Puts empty pot on warmer. Fills boiler with water. Puts filter and coffee grounds into filter holder and loads it into the receptacle.	
2	Pushes "Brew Button"	
3		Closes relief valve to allow water to flow to receptacle and turns on boiler heater.
4		When boiler is empty, turns off boiler heater. The indicator light is also turned on.

Use Case: Pour Coffee

In this use case, the coffee lover pours a cup of coffee from the pot. In doing so, he must remove the pot from the warmer and the software responds accordingly.

Use Case: Pour Coffee
System: Coffee Maker
Actors: Coffee Lover
Precondition: The Coffee Maker is not Brewing

Step	Actor (Coffee Lover)	System (Coffee Maker)
1	Lifts pot from warmer.	
2		Detects the pot has been lifted and turns off the plate warmer. If the indicator light is on, then turn it off (see note below)
3	Replaces pot on warmer.	
4		If there is coffee remaining in the pot, restart the plate warmer.

> **NOTE:** When should the indicator light be turned off? The original problem description does not answer that question. At this point we must go back to the system users and get clarifiation on this question. We will assume that they told us to turn off the light as soon as someone lifts the pot of coffee off the coffee maker after it is done brewing. This is reflected in the above use case.

Interrupt Brewing

What happens if the coffee pot is removed from the plate warmer before the coffee is done brewing. The coffee maker should turn off the boiler and open the relief valve (to prevent further water from entering the filter). When the pot is returned, the system should automatically restart the brewing process.

Use Case: Pour Coffee
System: Coffee Maker
Actors: Coffee Lover
Precondition: Brewing in progress

Step	Actor (Coffee Lover)	System (Coffee Maker)
1	Removes the pot from the plate warmer.	
2		Turns off the boile and opens the pressure relief valve.
1	Returns the pot to the plate warmer.	
2		Turns on the boiler and closes the relief valve, continuing the normal brewing cycle.

Design

Our first attempt at a design is shown in Figure 4. This is just a first try and is not a very "pretty" design. Most of the logic is lumped into the middle class creatively called "Controller". A lot of work will be put into refining this class into smaller pieces, but for now it gives us a place to start.

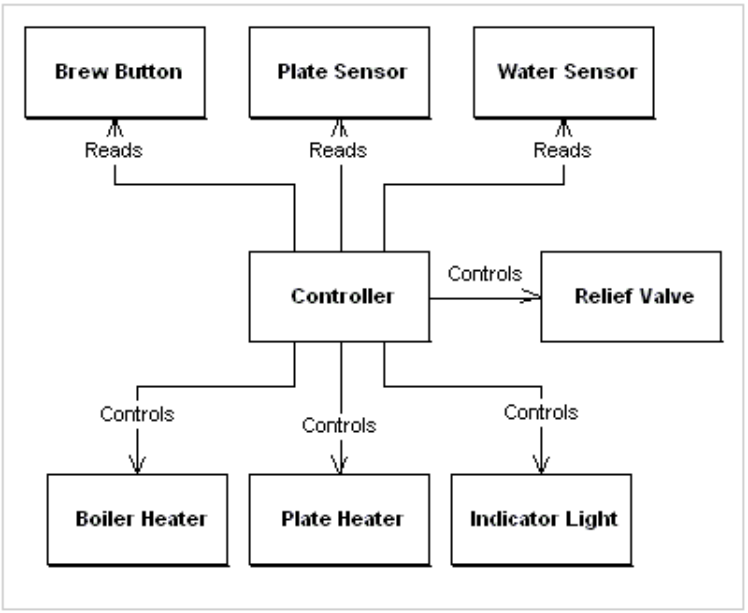

Figure 4 - Initial Design

Some people may be uncomfortable with the divergence of the Analysis and the Design models. Perhaps they expect that the design model will just be a refinement of the analysis. Remember that analysis is an attempt to understand the problem domain. Although we use OO tools to represent our understanding, that understanding has only an indirect influence on the structure of the software solution (in that the software must actually provide a solution). In fact, as we refine our software design, we will find that it moves even farther away from the analysis model. Solutions oriented classes, which do not appear in analysis, will be added to the design. Classes that came from the analysis model may mutate or even disappear entirely.

Output Hardware

When refining a design, I like to start with the easy stuff, and concrete objects are about as easy as they come. Focusing on the **PlateHeater**, **BoilerHeater**, **IndicatorLight** and **ReliefValve**, we find that the first three classes can be modeled using on/off commands and a simple query to detect the current state of the device. We can create a common base class called **OnOffDevice** and derive the two heaters and the light from it.

ReliefValve posses an interesting question. It seems its natural interface tends toward an open/close command interface rather than the on/off of the other three classes. Yet, it would be nice if it conformed to the same interface as the other devices. With the same interface, you could use the valve object in places where an on/off device might be expected).

Consider the following lines of code ...

```
valve.Open(); // the value is now open
```

```
valve.On(); // is the valve open or closed?
```

I definitely prefer the open/closed idiom for a valve. I think that choosing an on/off interface for it will lead to confusion to future user. If I ever need to use a valve where an on/off device is expected, then a simple adapter will solve that problem. Figure 5 shows the current design for the output devices.

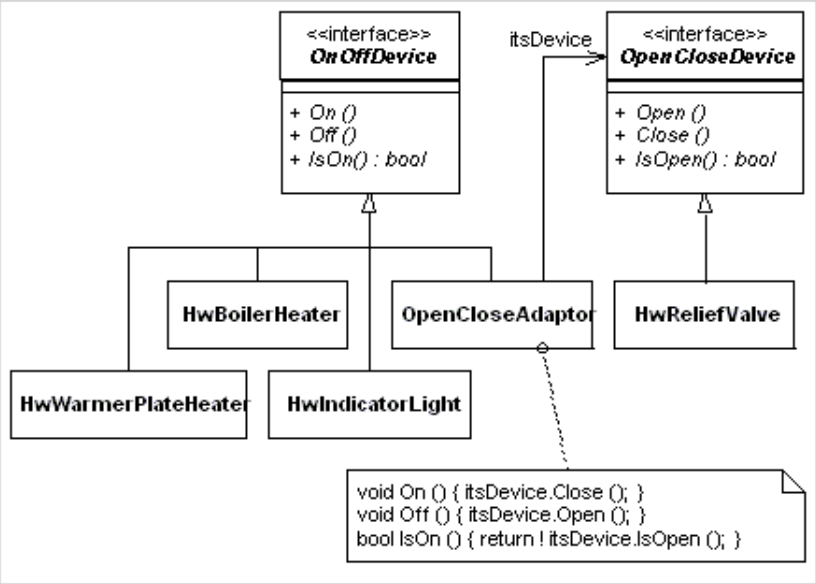

Figure 5 - Output Devices

> **NOTE:** *It will be a convention that the classes prefixed with "Hw" will interface directly with the hardware interface functions described in Appendix A.*

Input Hardware

With the output hardware out of the way, let’s turn our attention to the input side of the problem. Here there are three classes, two sensors and momentary push button. Filling in the obvious choices for member functions for each of the devices gives the following class diagram.

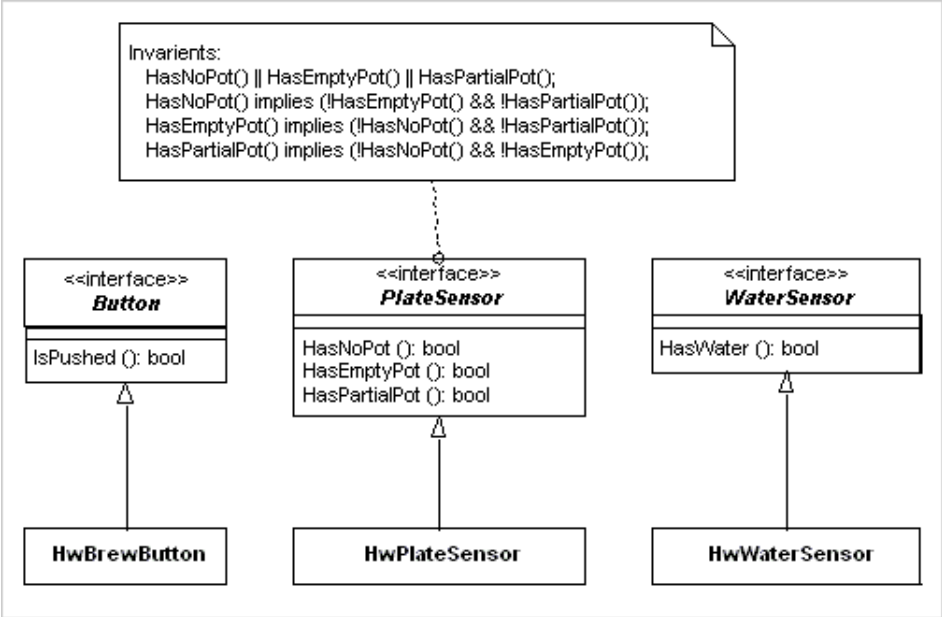

Figure 6 -- Input Hardware

Notice that we provide abstract interfaces for all the hardware. The rest of the design will only reference the abstract interfaces, allowing us to substitute other kinds of buttons and sensors as we need them.

Names

I want to spend just a little time talking about names. Names are very important and the choosing of names should be given careful consideration. The name of the method in the **WaterSensor** started out to be **IsBoilerEmpty ()**. It bothered me that a generic sounding name like **WaterSensor** had a method that referenced a boiler. It implied that the water sensor could only be used with boilers, an implication that I don't believe is true. I considered calling the method **IsWaterAvailable ()** or perhaps **IsWaterGone ()**, both of which sounded too awkward to me. A simple **IsWet ()** (or its negative **IsDry ()**) is getting closer. I finally settled on **HasWater ()**, which describes the test exactly without over specifying the class. Almost all my boolean query functions use the word "Is" in them, but the "Has" seems close enough to avoid confusion.

The **BrewButton** class was another problem. The name "**BrewButton"** seems over specific. A plain button class would seem much more reusable (especially in designs that are not involved in the brewing of coffee). However, if you read our hardware interface specification (see the appendix), you will note that the brew button has a particular function to be called. I decided that the button concept was important enough to rate its own abstraction (**Button**) and that **BrewButton** will be just one implementation of that interface. By the way, my first name for the query function was **IsDepressed()**, but that sounded too dismal. I changed it to **IsPressed()**.

The **PlateSensor** class was particularly difficult, in not only choosing the names of the methods, but also exactly what the methods should be. With two state sensors, I definitely prefer encoding the query as a TRUE/FALSE question. In other words, I prefer ...

```
If (sensor.HasWater()) { ? }
```

Over

```
If (sensor.State() == WaterSensor::HasWater) { ? }
```

However, **PlateSensor** is a three-state sensor. How many query functions do I need? My first attempt at this suggested three query functions (**IsPotOnWarmer (), IsPotEmpty (), IsPotNotEmpty ()**), only one of which would be true at any time. One problem with these names is that they are not really testing what they say. **IsPotEmpty ()** doesn't really check if the pot is empty (what if a coffee drinker just poured the last few drops from the pot into his cup, but hasn't yet returned the pot to the warmer? We have no way of knowing if the pot is empty or not). I played around with just two queries (**IsPotOnWarmer ()** and **IsPotEmpty()**) where the third query must be inferred from the state of the first two, but that seemed unsatisfactory.

The final names came from a realization that the *pot* is the wrong focus for the query. We are not asking about the state of the pot, but the state of the *warmer*. This gives us the queries: **HasNoPot (), HasEmptyPot (), HasPartialPot ()**. Note the appearance of the "Has" form of a query.

I give a list of invariants that specify that only one and only one of the query functions will be true at any instant in time.

A Common Sensor Class

There is a great temptation to combine the two sensor classes together as a single sensor. After all, we were able to combine the heaters and lights into on/off devices. Unfortunately, the two sensors provide very different interfaces. If we did manage to combine them, what would be the common methods in the base class? One common suggestion is to give both classes a query called **GetState ()**, that would return on of the following enums: **warmerEmpty**, **potEmpty**, **potNotEmpty, boilerEmpty** or **boilerNotEmtpy**. That leads to the structure in Figure 7. Unfortunately, Figure 7 is unusable. Code that tried to use polymorphic sensors would have to handle every possible sensor state. Some have suggested doing away with the enum and just use numbers (0, 1, and 2) for the states. We would have to assign arbitrary meanings to each of the states. Even so, some sensors would return two different states and some would return three. You still couldn't write code that could use either sensor without the exact concrete type.

So the combined sensor idea in Figure 7 in *not* a good idea.

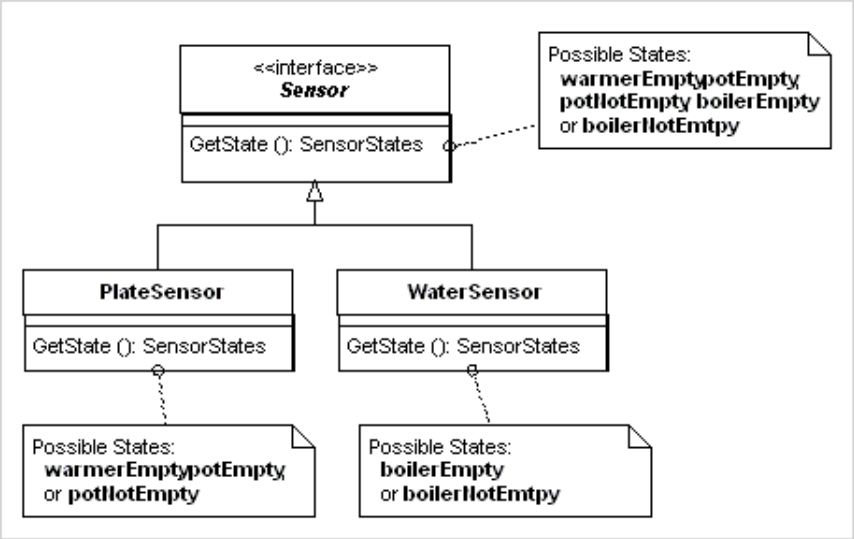

Figure 7 -- An Attempt to Unify Sensor Types (*not recommended*)

More on Input Hardware

We are not done with the input hardware yet, but I want to think about the "controller" part of the program before refining the input hardware any further.

The Brew Controller

The controller software manages all the output devices based on data from the input hardware. Sounds like a good candidate for a state diagram. Figure 8 is a first draft of the state transitions for our controller.

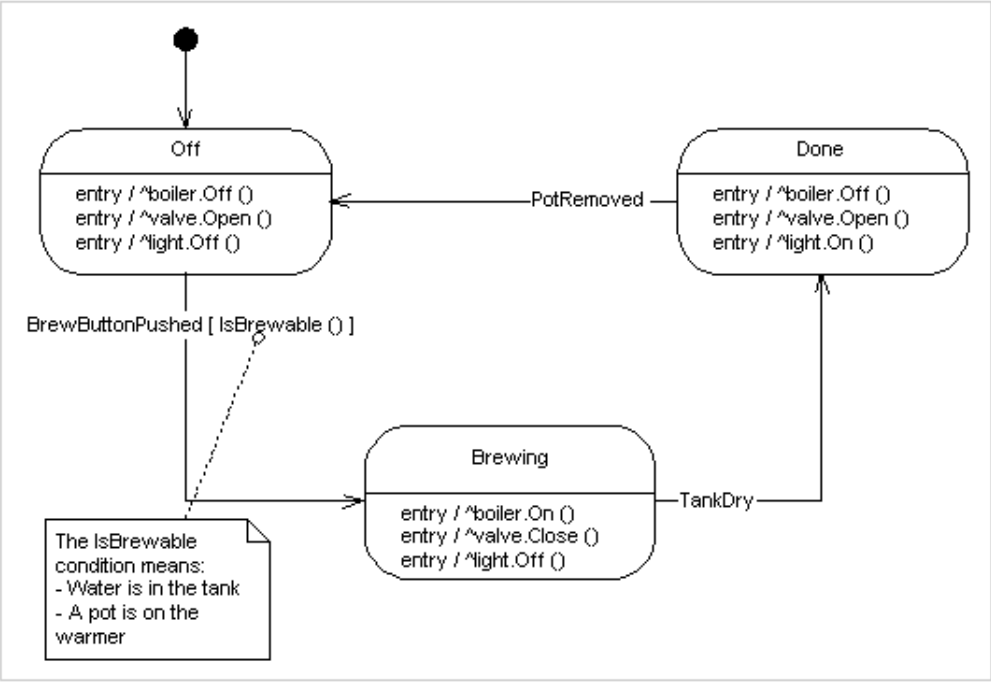

Figure 8 -- Basic Brew Controller State Diagram

The brew controller starts in the *Off* state. When the button is pressed, the controller enters the *Brewing* state until the water tank is empty, then it goes into *Done* state.

Is the *Done* state really needed? It seems to be. After the brewing is done, the indicator light must be turned on. How long should the light be left on. This is address in the note in use case descriptions. We leave the light on until the user removes the pot. This is captured in our state diagram. While in the *Done* state, a *PotRemoved* event will return us to the original *Off* state.

The Brew Controller, Try #2

The above Brew controller state machine is incomplete. The use cases specify that the user is able to remove the pot during brewing, and that the brewer process is halted until the pot is returned. Revising the state diagram in Figure 8 gives us the following result.

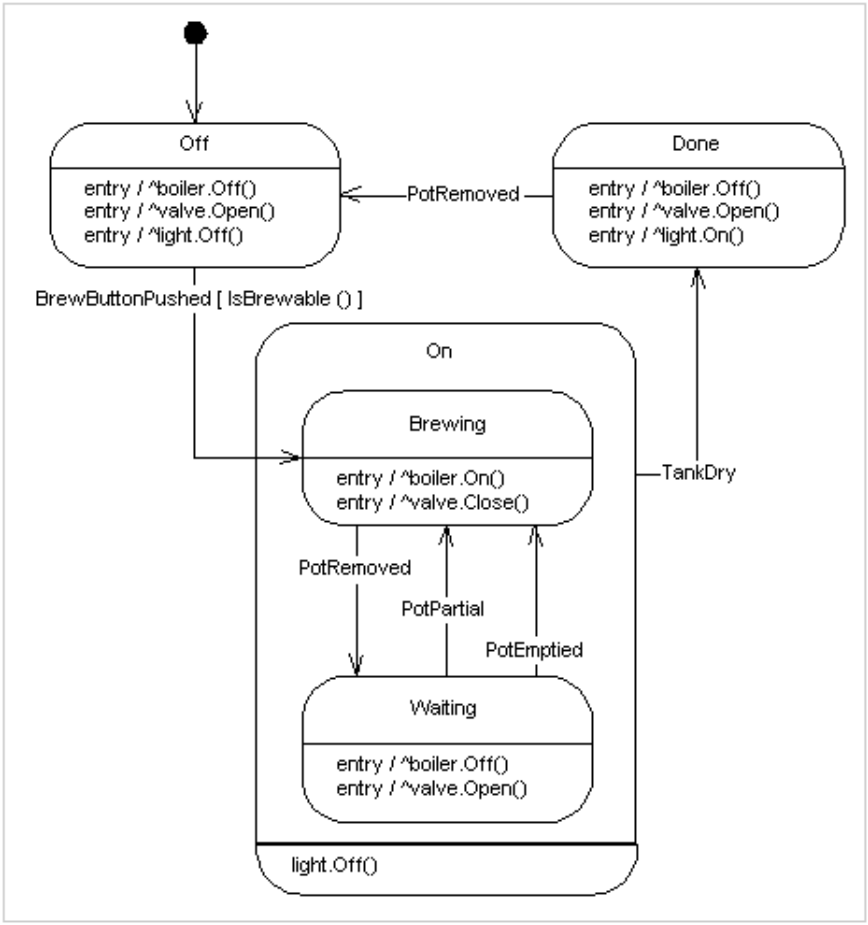

Figure 9 -- Brew Controller with "Pause" States

The *Brewing* state has been broken down into two substates, *Brewing* and *Waiting*. The waiting state is triggered by the removal of the pot. Returning the pot returns the brew controller back to the brewing state.

The Brew Controller, Try #3

Although, the brew controller illustrated in Figure 9 meets the requirements of the use cases, we are unhappy with it for design reasons. It is overly specific about the types of signals, specifying each event in terms of the "BrewButton" and other very application specific names. Can we isolate the basic control algorithm from such simple things as changing from a BrewButton to a Timer Trigger to start the brew cycle?

Figure 10 is an attempt to produce a generic controller. This controller will control anything that can be started and stopped, with a separate *Done* state, and the ability to pause while running. It also has a user definable **IsReady()** query that can be used to prevent starting in unsafe or desirable conditions. It controls a single device (that it turns on and off) and an indicator light (that is also turned on and off). All application specific names are gone.

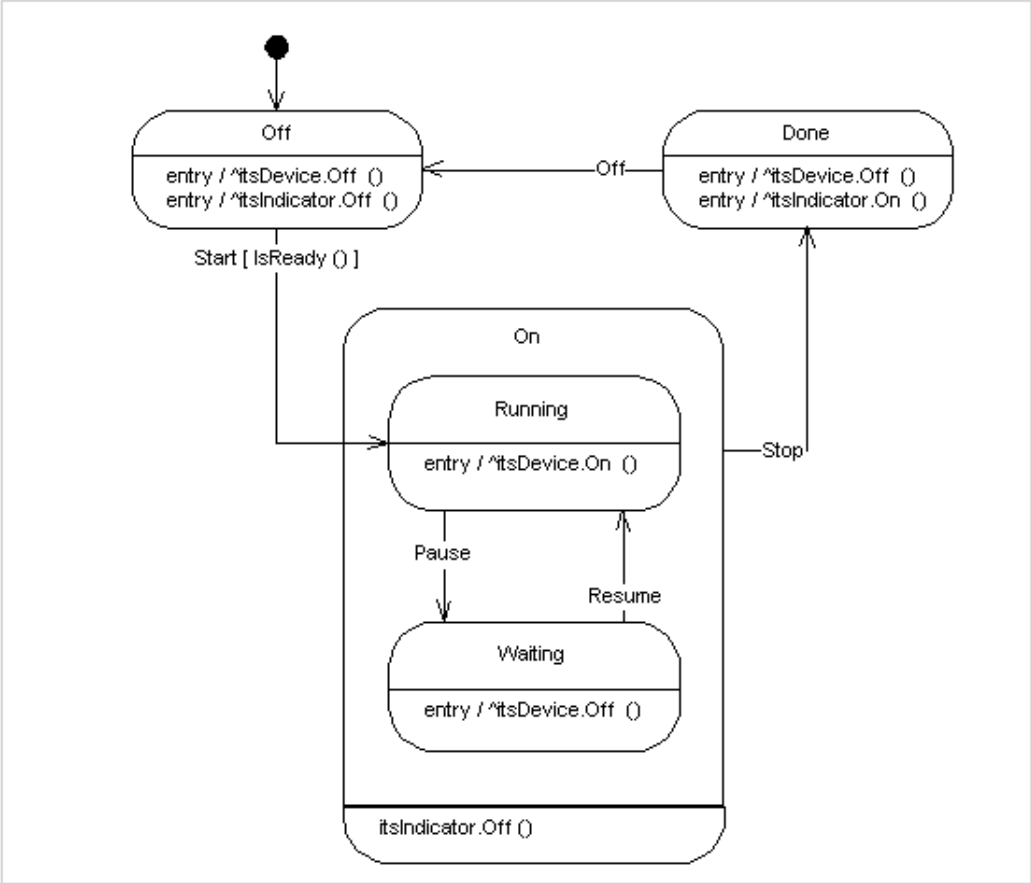

Figure 10 -- Start/Stop Controller States

Specializing a Start/Stop Controller

Since the Start/Stop controller doesn’t know about our application events, how do we get it to respond to button pushes and water sensors? Figure 11 gives a clue. Create a BrewController class that inherits the Start/Stop machine behavior and maps the application events into the pure *Start*, *Stop*, etc.

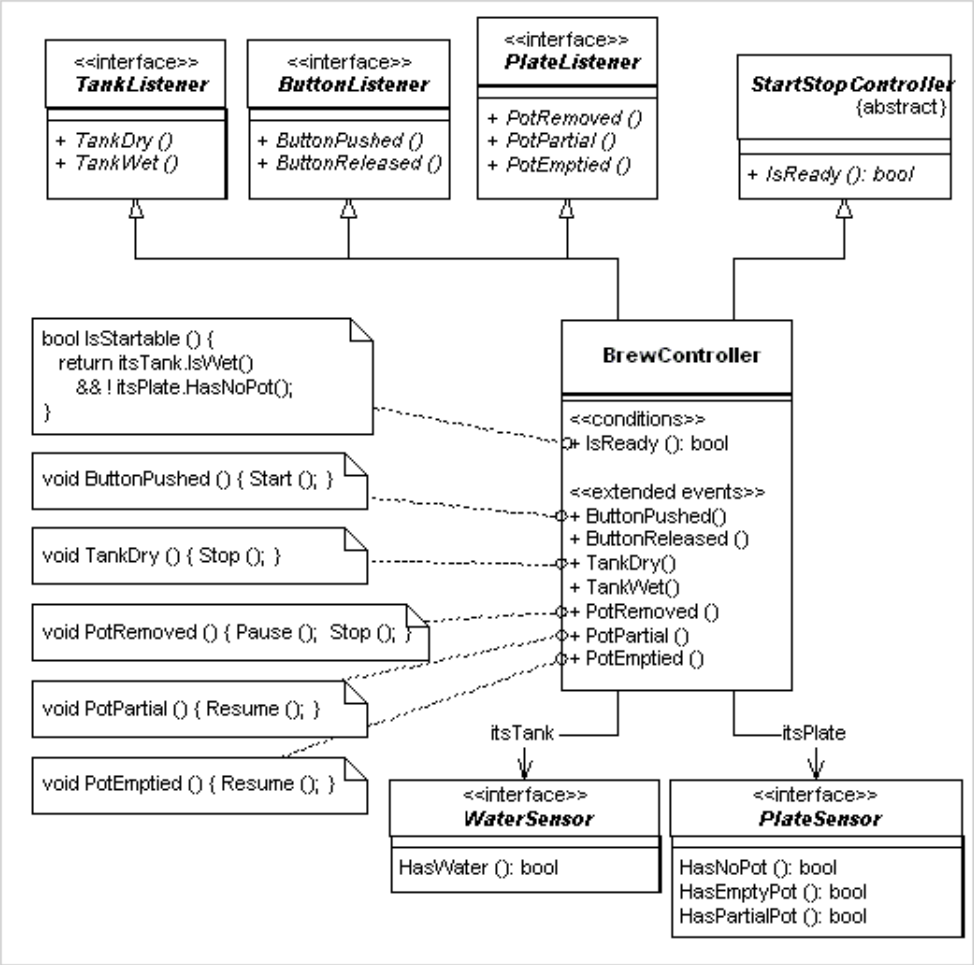

Figure 11 -- Specializing the Start/Stop Controller

Notice that the Brew controller depends on the various application specific listeners, but the Start/Stop controller itself is completely independent of them. Each application event is translated into a start/stop like event.

The *PotRemoved* is translated into both a *Pause* and a *Stop* event. Since the irrelavent events are ignored by the state machine, this is no problem.

The Plate Controller

After dealing with the Brew Controller, the Plate controller seems almost trivial. Here is its state machine.

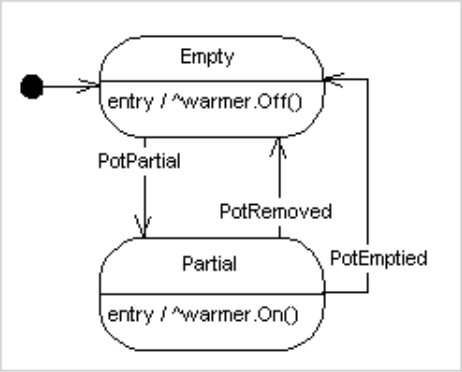

Figure 12 -- Plate Controller States

Can we make a generic version of the Plate controller as we did for the brew controller. Well, technically yes. But the plate controller is so simple to implement, there is little advantage to doing more that the straight forward implementation shown in Figure 13.

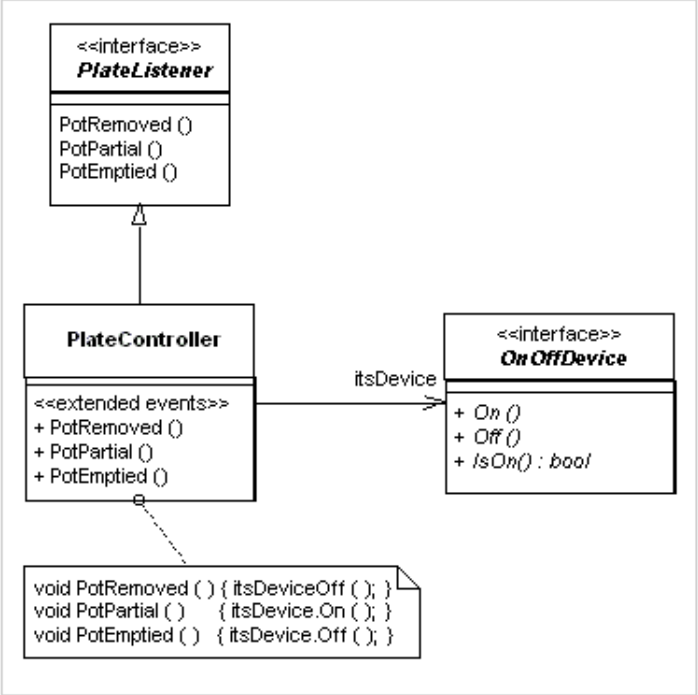

Figure 13 -- Plate Controller

Controlled Devices

The controllers talk to generic On/Off devices. This removes the explicit dependencies on particular kinds of devices. The **PlateController** may always control a heater, but there is no need for our software to depend on that fact.

But what about the **BrewController**? It must control both a boiler heater (which is On/Off) and a relief valve (which is not). We can combine both the heater and the valve into a single special purpose On/Off device we will call a **BrewDevice** (see Figure 14).

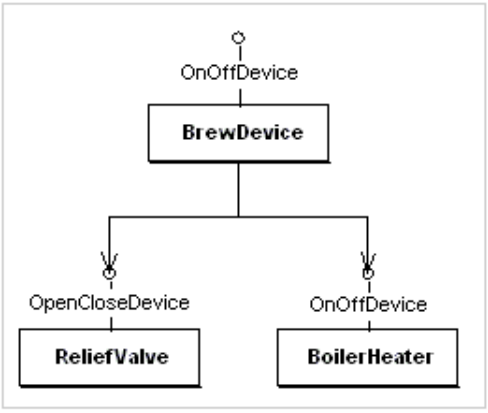

Figure 14 -- Brew Device

The **BrewDevice** works fine, but it is a specialized object that is created only to combine the relief valve and the boiler heater. A little more work will yield a more general solution. Figure 15 shows a general way of combining several On/Off devices using a composite pattern.

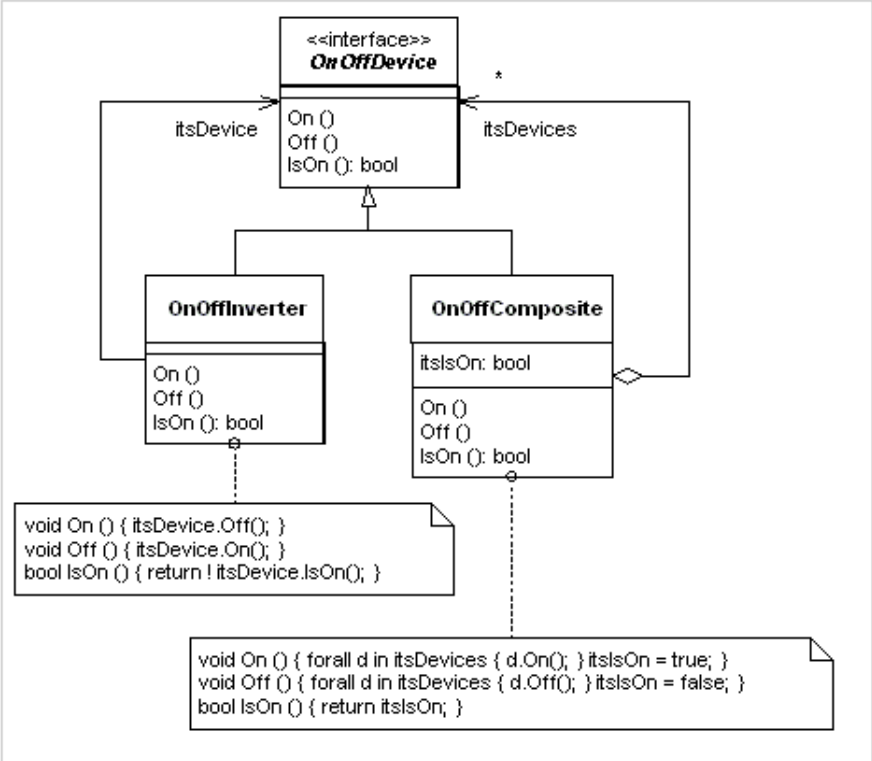

Figure 15 -- Composite On/Off Device

Figure 16 shows a brew device constructed out of the composite pattern of Figure 15 and the Open/Close device adapter from Figure 5.

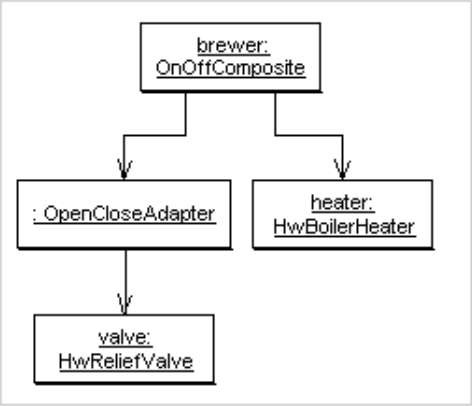

Figure 16 -- Brew Device Using Composite

The following sequence diagram gives the details of the controller turning on the composite object and the propagation and translation of that message to the individual hardware devices.

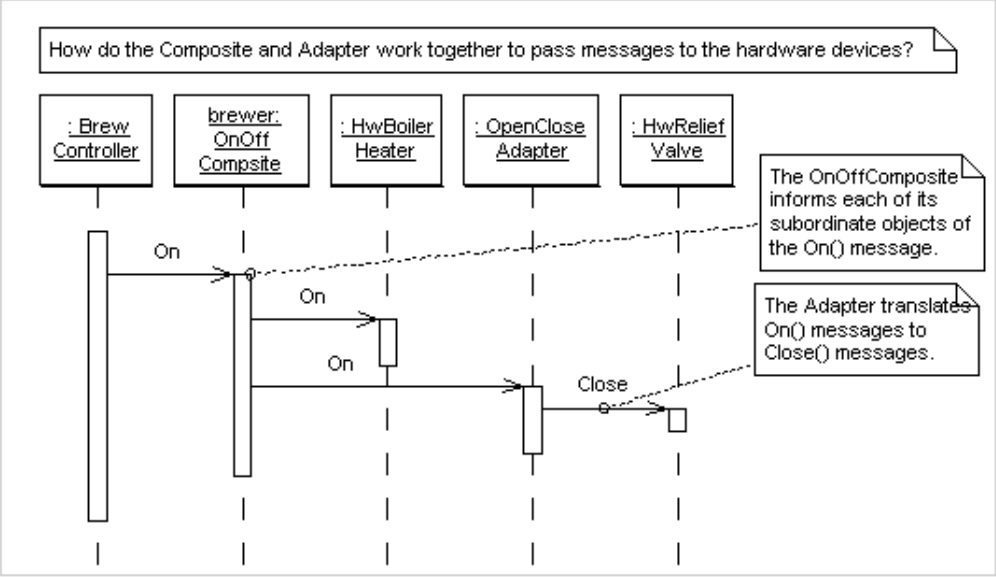

Figure 17 -- Composite Sequence Diagram

Controller State Machines

We have omitted the details of implementing the state machines for each of the controllers. It is not because the topic is trivial or uninteresting, but because of space and time constraints. These state machines are simple and may be coded by hand. An alternative is to use the State Pattern from the GOF book and use Robert Martin's SMC (State Machine Compiler) to generate most of the grunt work.

Managing Events

We still haven't addressed the issue of generating events needed by teh brew and plate controllers. The individual controllers don't care how the events are created, so long as each event is delivered to the

proper controller(s). Since our input devices are passive, we need another piece of software that will watch the input devices and generate the expected events.

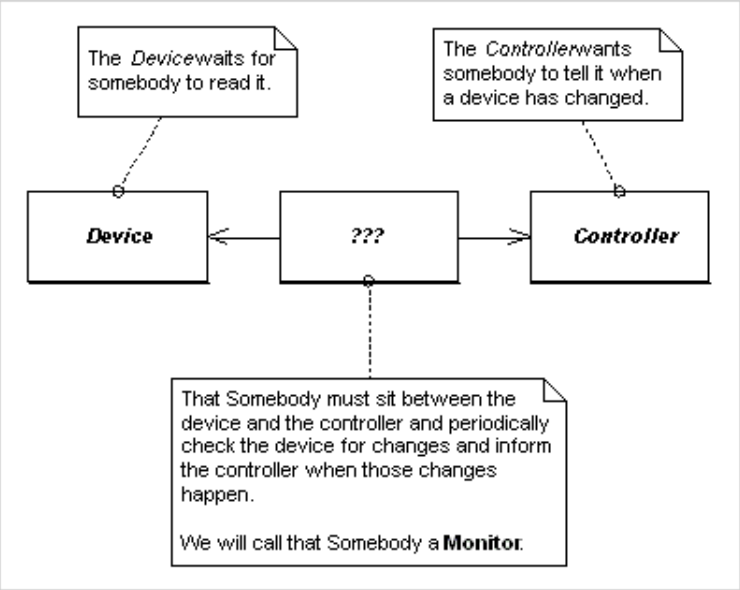

Figure 18 -- What Connects Devices to Listeners?

We will call the object that generates events by listening to passive input device a *Monitor*. Monitors should be able to notify a list of (appropriate) listeners of any change in the state of the input device it is monitoring.

When and how often should monitors read their input devices? There are several overall approaches to this problem.

1. Poll each device as fast as possible, looking for changes.
2. Use a timer to poll the devices at particular intervals.
3. Tie each device to a hardware interrupt, so that when the state changes, the software is immediately notified of the change.

Each of these approaches has advantages and disadvantages. We would like to design the software so that the choice between the polling vs. interrupts vs. timer is as localized as possible. We do this by giving each monitor a *Trigger()* method that is called by some type of scheduler. The scheduler takes care of the decision to poll or use interrupts. In turn the monitors only read their inputs when triggered. Figure 19 shows the details of this.

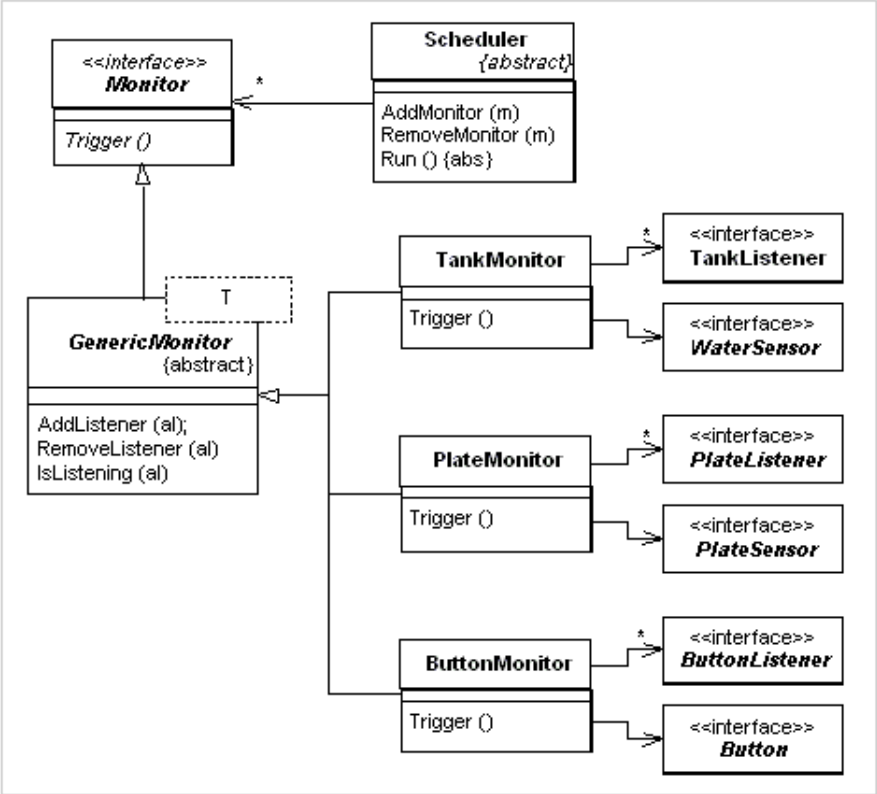

Figure 19 -- Monitors

Figure 20 describes how the scheduler and monitor interact to inform the controller of changes in the monitored device.

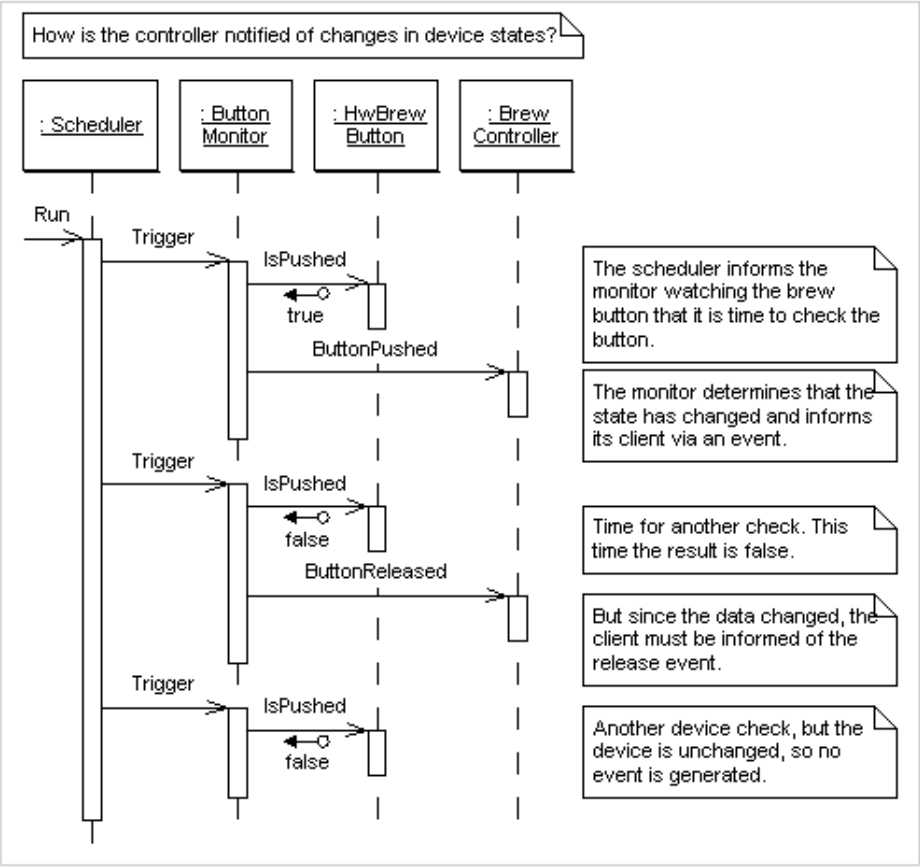

Figure 20 -- Monitor Sequence Diagram

Putting it all Together

We've gone through quite a number of class sorted through numerous interfaces and protocols. So, how does it fit together?

Figure 21 shows the objects that will be created and how they are interconnected. Seen in this form, it is not so formidable. We can see the three monitors connected to their respective hardware inputs being driven by a polling scheduler. The monitors feed into the various controllers that, in turn, drive the hardware output devices. A composite and adapter are used to combine two devices into a single object for the **BrewController** to use.

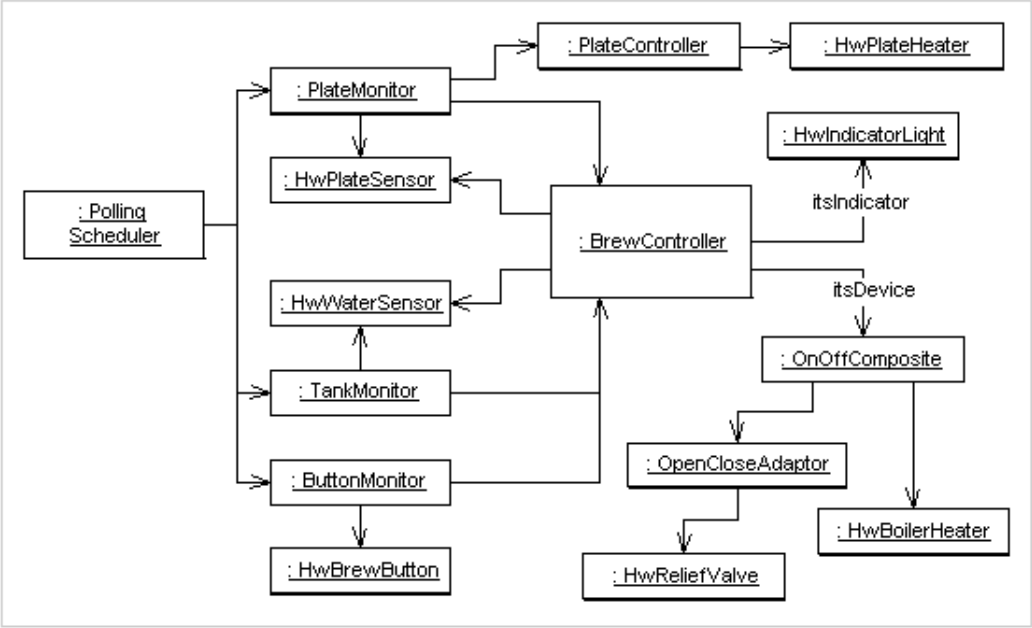

Figure 21 -- System Object Diagram

Building the System

Somewhere, someone needs to know the concrete types so that the system can be put together. The objects in this design are rather static. Once created, they stay in existence. In is entirely possible that the main program (or some high level configuration function) could build the initial system from concrete objects and kick start it into running. The main program might look something like the following snippet of code.

```
int main ()
{
    // Create the devices

    auto_ptr<Button> brewbutton (new HwBrewButton);
    auto_ptr<PlateSensor> plate (new HwPlateSensor);
    auto_ptr<WaterSensor> tank (new HwWaterSensor);

    auto_ptr<OnOffDevice> warmer (new HwPlateHeater);
    auto_ptr<OnOffDevice> light (new HwIndicatorLight);
    auto_ptr<OnOffDevice> heater (new HwBoilerHeater);
    auto_ptr<OpenCloseDevice> valve (new HwReliefValve);

    // Create controllers, adapters, etc., and establish the required
    // connections. Finally, kick start the system by starting the
    // scheduler.

    sch->Run();
    return 0;
}
```

NOTE: *We are using auto_ptr so that the objects will be automatically deleted when the*

function exits.

Now suppose we wish to create a test version of our software, where everything is exactly the same as the real version, except the hardware is simulated on in a debugging environment. That means we have classes like **SimulatedBoilerHeater**, **SimulatedBrewButton**, that need to be instantiated instead of the "Hw" versions.

We can duplicate the main program and change those seven lines of code, but there is a alternative that relies on less duplicated code. We can create an abstract factory interface that has a creation method for each of the hardware devices we wish to control at creation time. We then derive two (or more!) concrete classes that implement the abstract factory interface (see Figure 21).

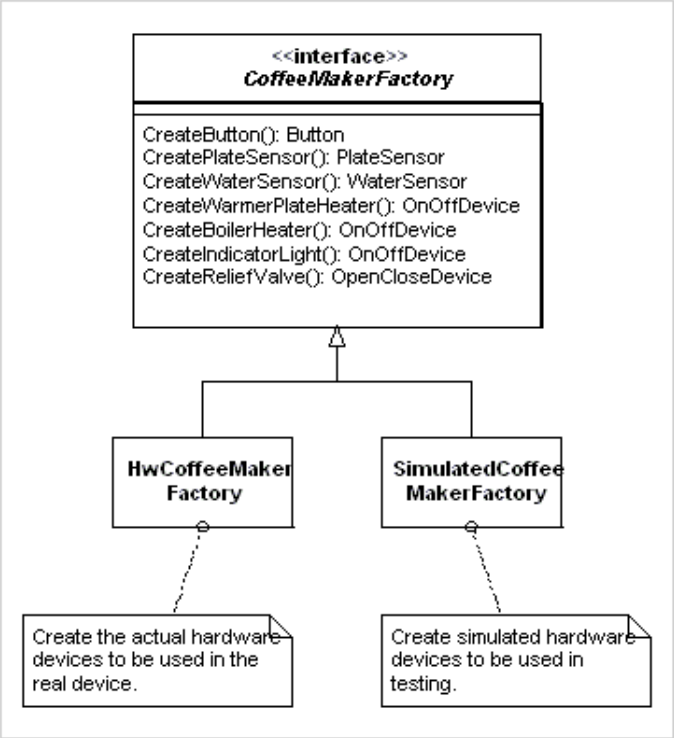

Figure 22 -- Coffee Maker Abstract Factory

Now, instead of a seven line change, we can make a single line change to control which factory is created.

```
int main ()
{
    // Create the factory

    auto_ptr<CoffeeMakerFactory> f (new SimulatedCoffeeMakerFactory);

    // Create the devices

    auto_ptr<Button> brewbutton (f->CreateButton());
    auto_ptr<PlateSensor> plate (f->CreatePlateSensor());
    auto_ptr<WaterSensor> tank (f->CreateWaterSensor());

    auto_ptr<OnOffDevice> warmer (f->CreateWarmerHeater());
```

```
    auto_ptr<OnOffDevice> light (f->CreateIndicatorLight());
    auto_ptr<OnOffDevice> heater (f->CreateBoilerHeater());
    auto_ptr<OpenCloseDevice> valve (f->CreateReliefValve());

    // ? and so on ?
}
```

We could even select between the two sets of hardware devices at run time.

```
    // Choose which factory to use
    auto_ptr<CoffeeMakerFactory> f;
    if (doSimulatedRun)
    {
        f.reset (new SimulatedCoffeeMakerFactory);
    }
    else
    {
        f.reset (new HwCoffeeMakerFactory);
    }
```

For seven lines of code, I'm not sure an Abstract Factory Pattern is warranted. If there were hundreds of lines of code concerned with object creation, and these lines of code were scattered throughout the application, then the Abstract Factory Pattern would have a clear advantage.

End Notes

Software engineering is about tradeoffs: speed vs. size, flexibility vs. complexity to name a few. In this design, I tended to choose the flexible over the simple, mainly to show how to build flexibility into the design. Is there a cost for this flexibility? Yes, I believe there is. The cost is a more classes with more levels of indirection. Is the flexibility worth the cost? In this small teaching example, the answer is debatable. However, as the project grows in size and complexity, the importance of a decoupled design grows as well.

Appendix A - Hardware Interface

The hardware-engineering group has graciously provided the following C header file for the hardware interface library. All functions do the obvious function implied by their name. It is worth noting that the **GetBrewButtonStatus** function will only return the "pushed" status once for each push of the button (this avoids problems with the user leaning on the button and sending many "pushed" messages).

```
/* Coffee Maker Low Level Hardware Interface */
enum WarmerPlateStatus { potNotEmpty, potEmpty, warmerEmpty };
enum WarmerPlateStatus GetWarmerPlateStatus ();

enum BoilerStatus { boilerEmpty, boilerNotEmpty };
enum BoilerStatus GetBoilerStatus ();

enum BrewButtonStatus { brewButtonPushed, brewButtonNotPushed };
enum BrewButtonStatus GetBrewButtonStatus ();

enum BoilerState { boilerOn, boilerOff };
```

```
void SetBoilerState (enum BoilerHeaterState s);

enum WarmerState { warmerOn, warmerOff };
void SetWarmerState (enum WarmerState s);

enum IndicatorState { indicatorOn, indicatorOff };
void SetIndicatorState (enum IndicatorState s);
```

Appendix B - Summary Class Diagrams

See also: Figure 21 -- System Object Diagram

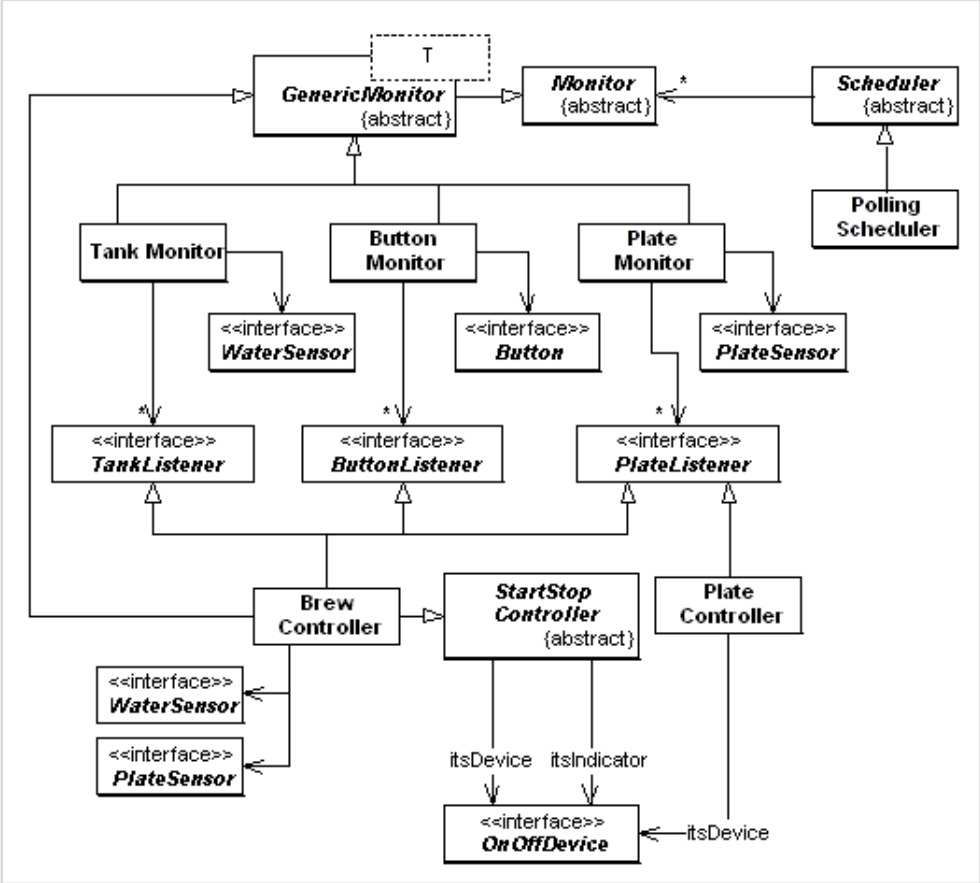

Figure B1 -- Overall Class Structure

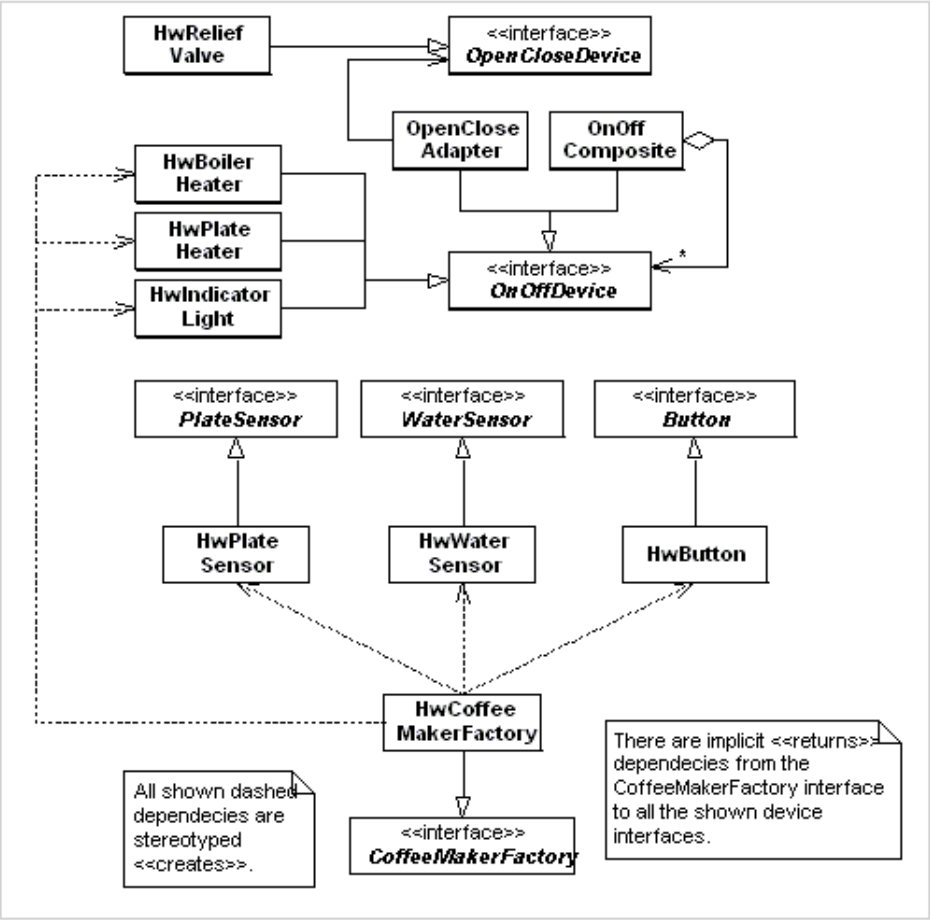

Figure B2 -- Hardware Devices

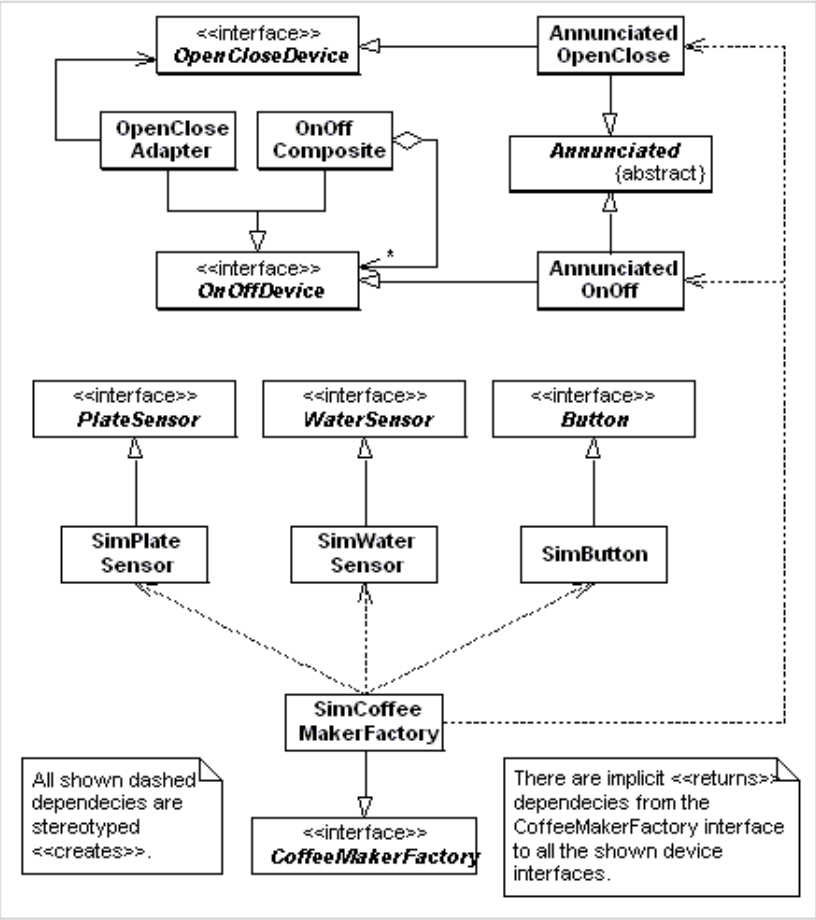

Figure B3 -- Simulated Hardware Devices

Jim Weirich / jweirich@one.net
Last modified: Tue Dec 28 19:56:52 1999