

## *The Mark IV Special Coffee Maker*

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

# Hardware Interface

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

# Exercise

- Assume the existence of a low-level API that actually controls the various devices
- Design a set of OO classes to control the coffee maker
- Prepare UML class-model diagram describing your design

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

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

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

```
/** This function returns the status of the boiler switch.  
 * The boiler switch is a float switch that detects if  
 * there is more than 1/2 cup of water in the boiler.*/  
public int getBoilerStatus();  
  
public static final int BOILER_EMPTY = 0;  
public static final int BOILER_NOT_EMPTY = 1;
```

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

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

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

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

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

public static final int WARMER_ON = 0;
public static final int WARMER_OFF = 1;
```

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

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

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

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

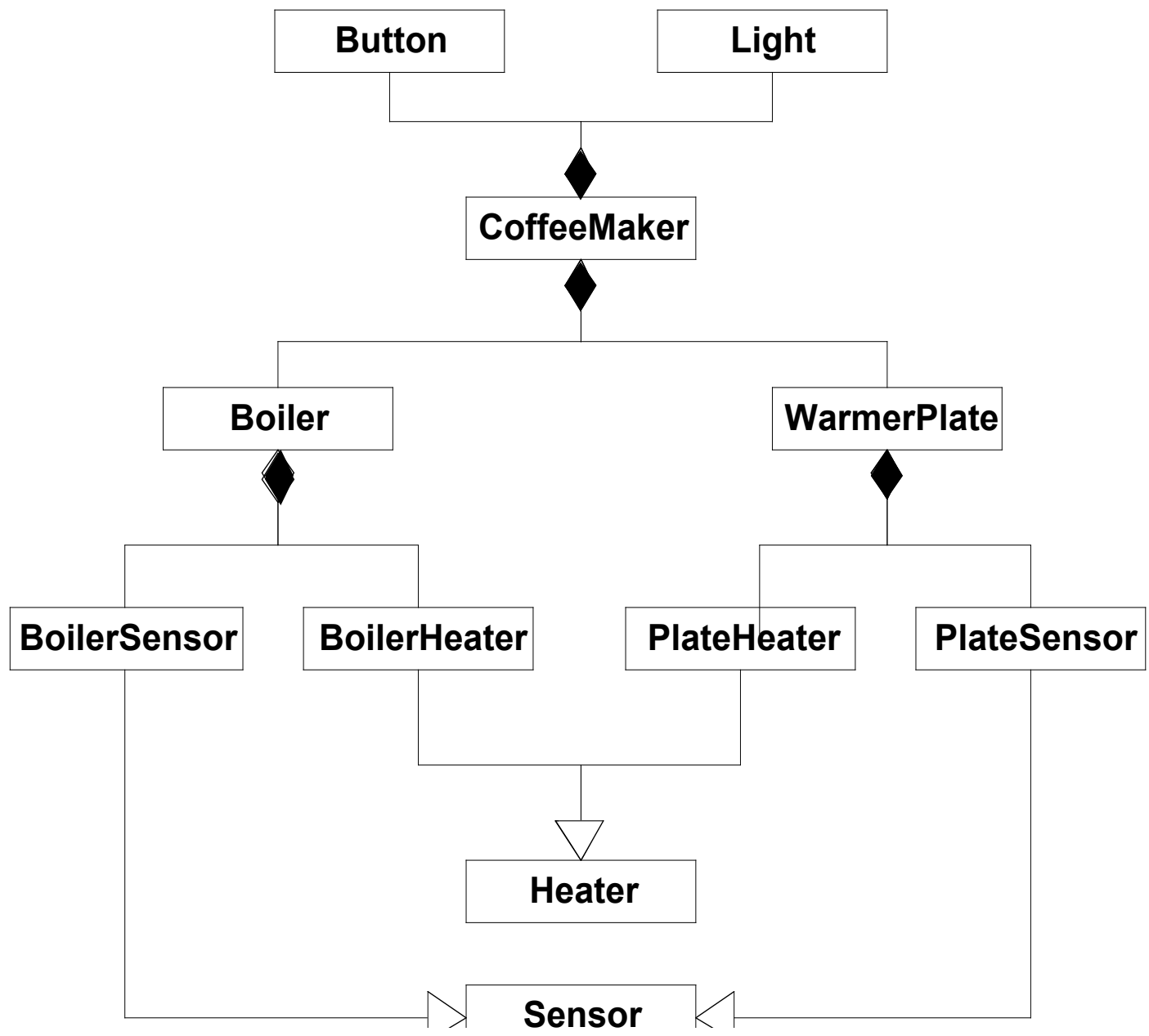
# Traditional Approach

- The traditional OO approach to doing modeling is to search for nouns in the problem statement and model them with classes
- After doing this and analyzing the results, the following list of terms is obtained:

filter holder, receptacle, water strainer, button, warmer plate, warmer-plate sensor, boiler switch, float switch, brew button, momentary switch, state, heating element, boiler, indicator light, brewing cycle, pressure-relief valve, steam pressure, boiler

# Class Model Diagram

- From such a list and our knowledge of the domain, we can construct a high-level class model diagram
- Martin's original version is presented on the following slide



# Problems

- Missing methods
- Vapor classes
- Imaginary abstraction
- God classes
- Missing user interface

# Missing Methods

- Martin didn't include any methods in his diagram, and then he complains about their lack
- What is really going on is an artificial emphasis on structure at the expense of behavior

# Vapor Classes

- The `Light` class exists because of the need to control the hardware light
- But there is no software state
- In fact, the class itself is just an adaptor for the API
- The same observation holds for `Button`, `Boiler`, and `WarmerPlate`

# Imaginary Abstraction

- Martin notices that `Heater` and `Sensor` act as base classes but are not used
  - But isn't this what factoring is all about
- But he notices that there is little if any code to actually factor

# God Classes

- When the previous criticisms are taken into account, the only substantive class remaining is `CoffeeMaker`
- This situation is sometimes called a *God Class* and is generally frowned upon

# Alternative Approach

## Start with Use Cases

- An alternative approach to the above is to write out Use Cases
  - Simple stories illustrating a single execution or pointing out an important obstacle
- What use cases can you come up with?

# Alternative Approach

## Use Cases - 2

- An alternative approach to the above is to write out Use Cases
  - Simple stories illustrating a single execution or pointing out an important obstacle
- What use cases can you come up with?
  - *User pushes brew button*
  - *Containment vessel not ready*
  - *Brewing complete*
  - *Coffee all gone*

# Representation of Use Cases

- Usually a Use Case is textual
  - Either just a short paragraph
  - Or a sequence of Actor-Action-[Object] triples
- UML provides several diagram for denoting Use Cases
  - Collaboration diagram (object diagram with ordered, labeled transitions)
  - Sequence diagram (time ordered message passing)
- Note that a Use Case diagram is a way of expressing context by organizing a set of Use Cases

# CRC Cards

- A CRC card is an informal way of describing what you learn from a set of Use Cases
  - CRC stands for Class-Responsibility-Collaborator
- Each card corresponds to one participant in a Use Case
- Listed on the card are that class's responsibilities and any necessary collaborating classes
- The contents of a card grows as more Use Cases are considered

# Collaboration Diagram Example

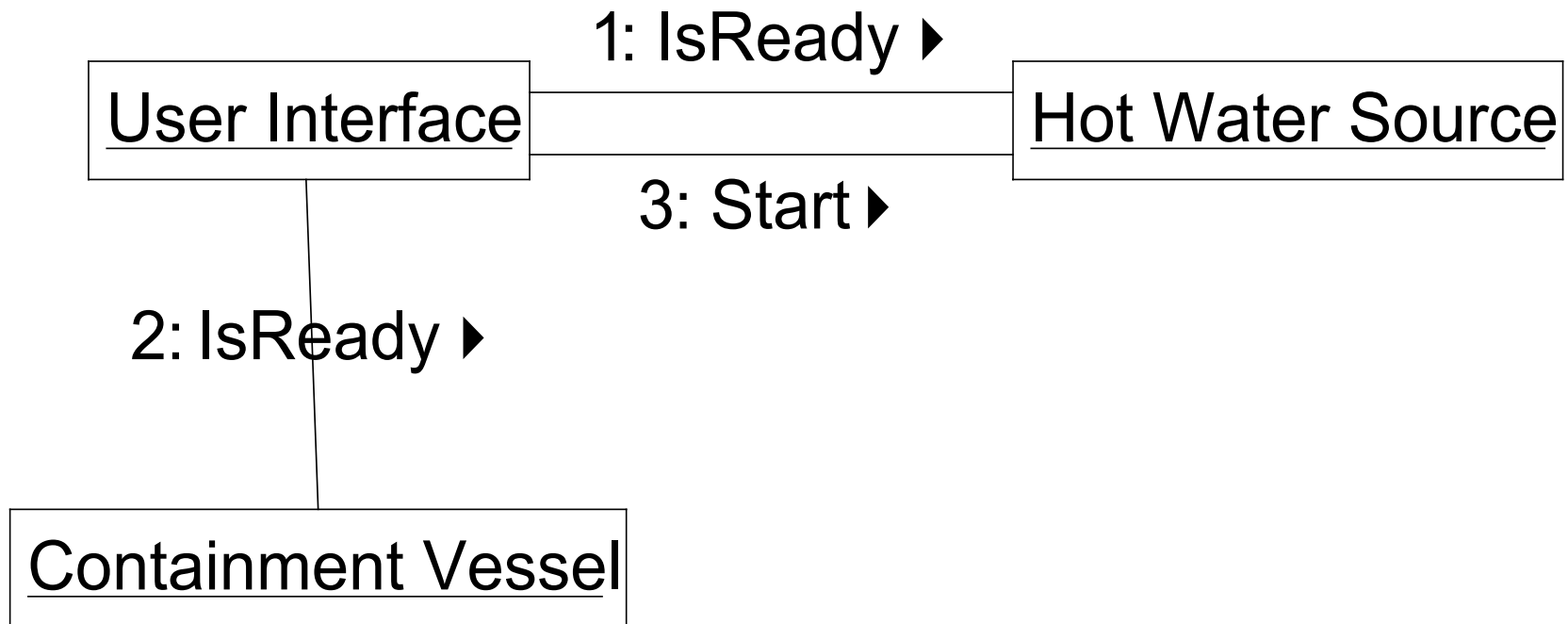
- What must happen in the coffee maker when the user presses the brew button?
- First, list the steps the coffee maker must take
- Then draw a UML collaboration diagram expressing these actions

# Collaboration Diagram - 2

- A collaboration diagram for the first Use Case (*User pushes brew button*) is shown on the next slide
- Note that the rectangles are objects not classes
- Arrows are annotated with numbered method names
  - Arguments may be provided

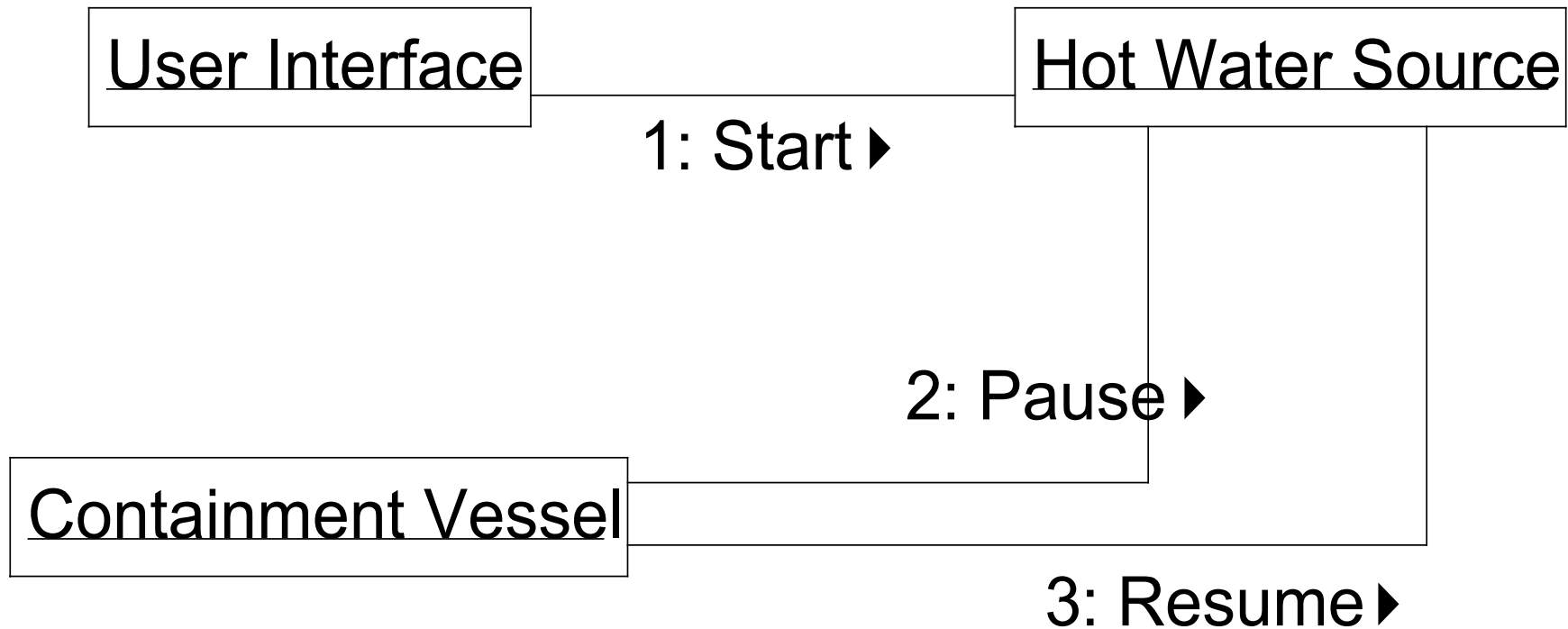
# Collaboration Diagram

## *User Pushes Brew Button*

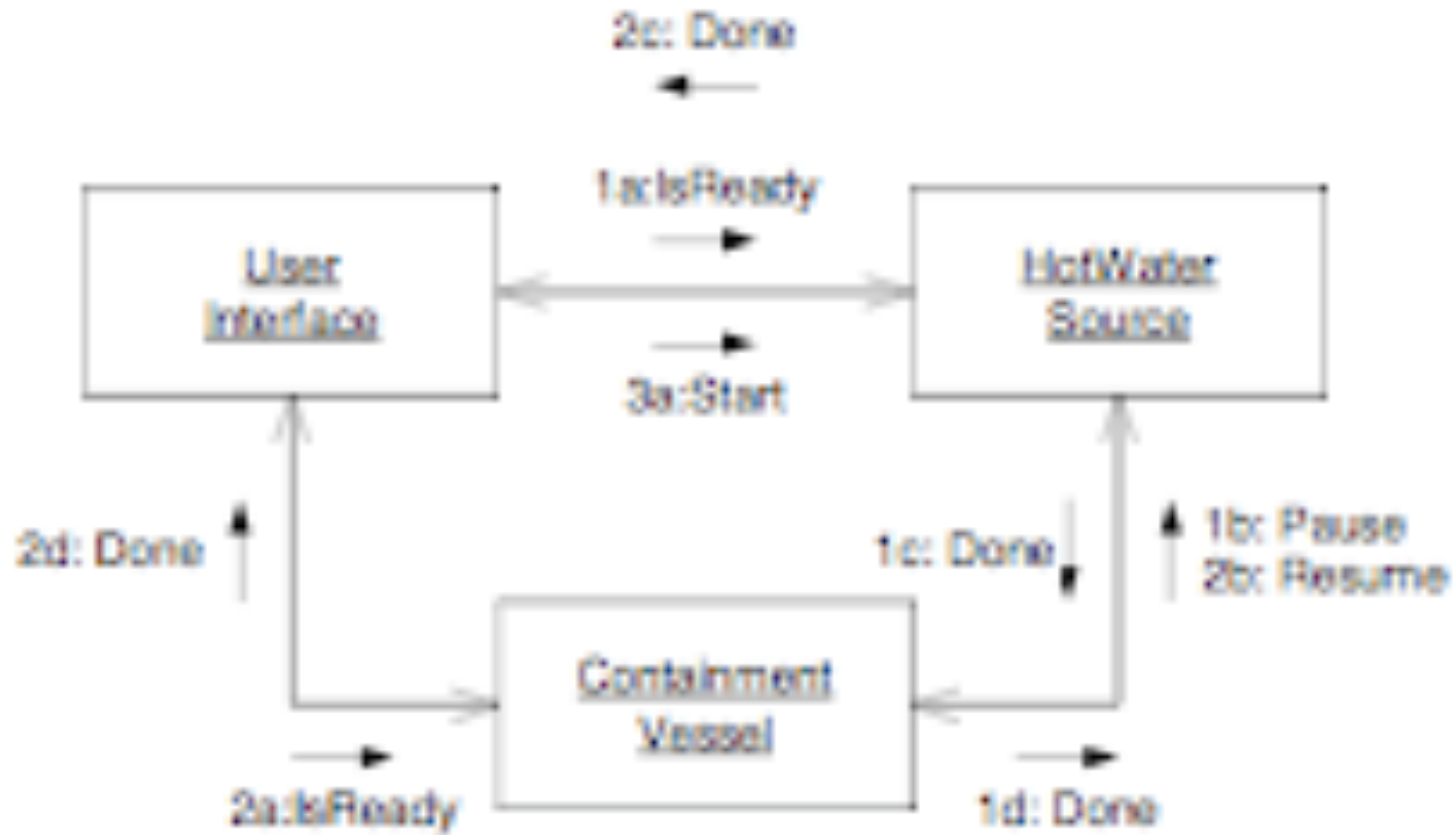


# Collaboration Diagram for Use Case 2

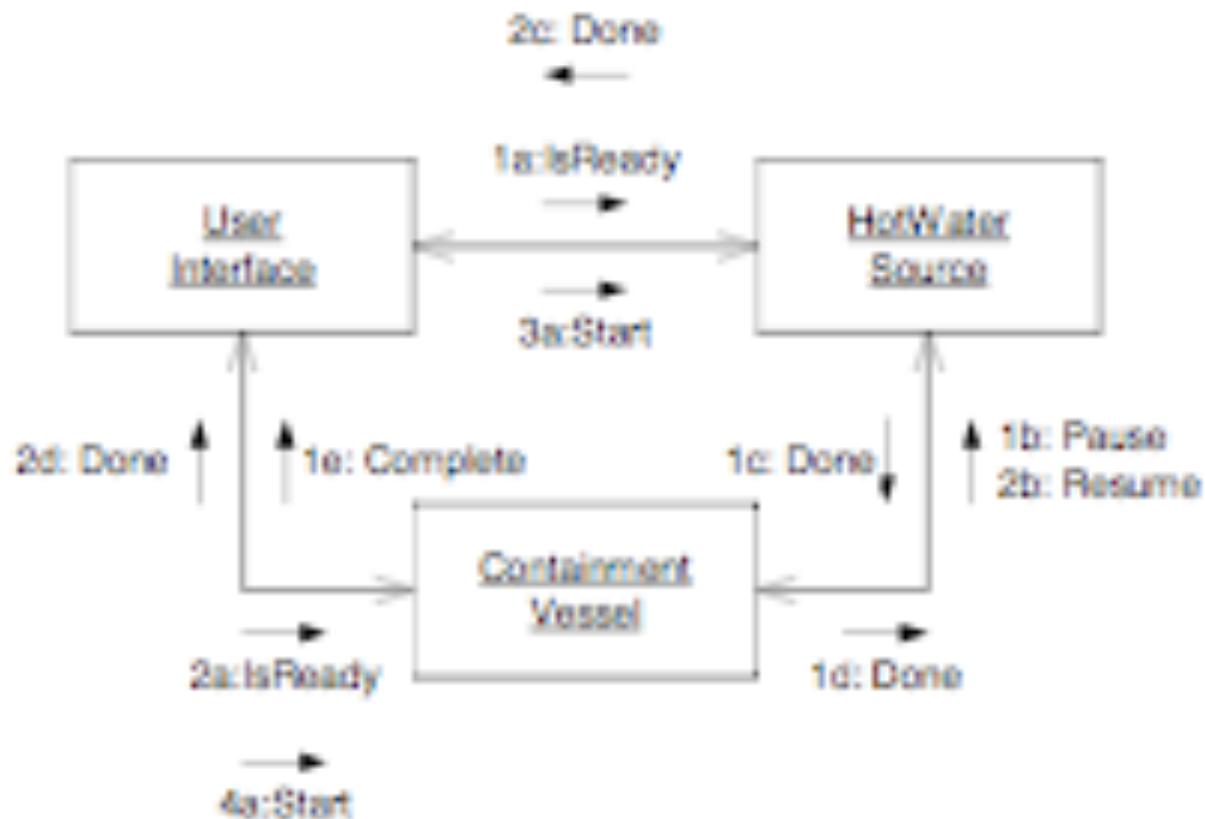
## *Containment Vessel not Ready*



# Collaboration Diagram for Use Case 3 *Brewing Complete*



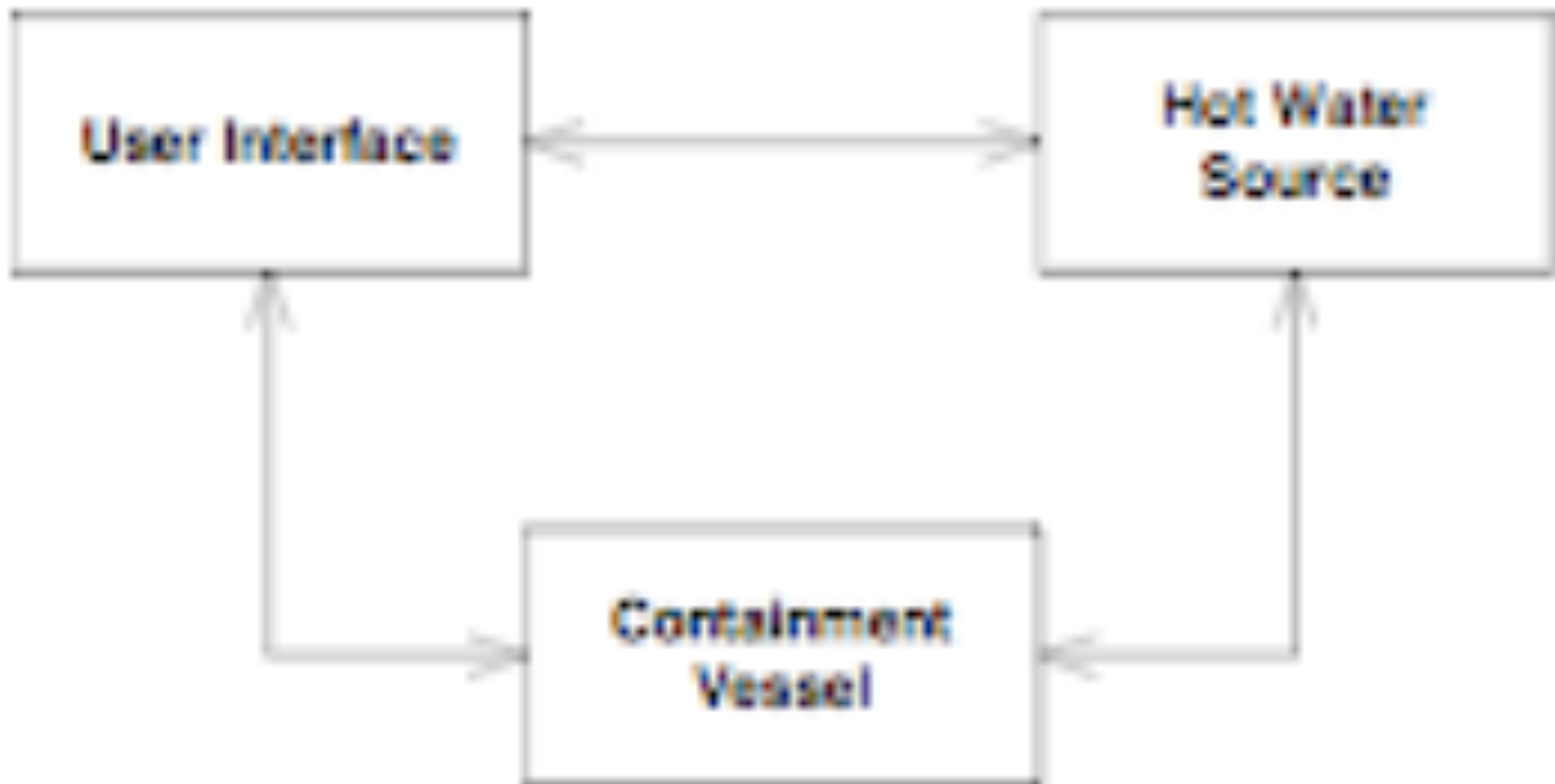
# Collaboration Diagram for Use Case 4 *Coffee All Gone*



# Roles

- Each of the objects in a Collaboration diagram can be thought of as playing a *role* in that particular collaboration
- For example, `ContainmentVessel` sometimes plays the role of `StatusReporter`
- If Collaboration diagrams were drawn for each Use Case, then the sum of the roles for a given object describe that object's overall responsibilities
- That is, the object's features can be synthesized from the roles it plays

# Inferred (Abstract) Classes



# Refinement

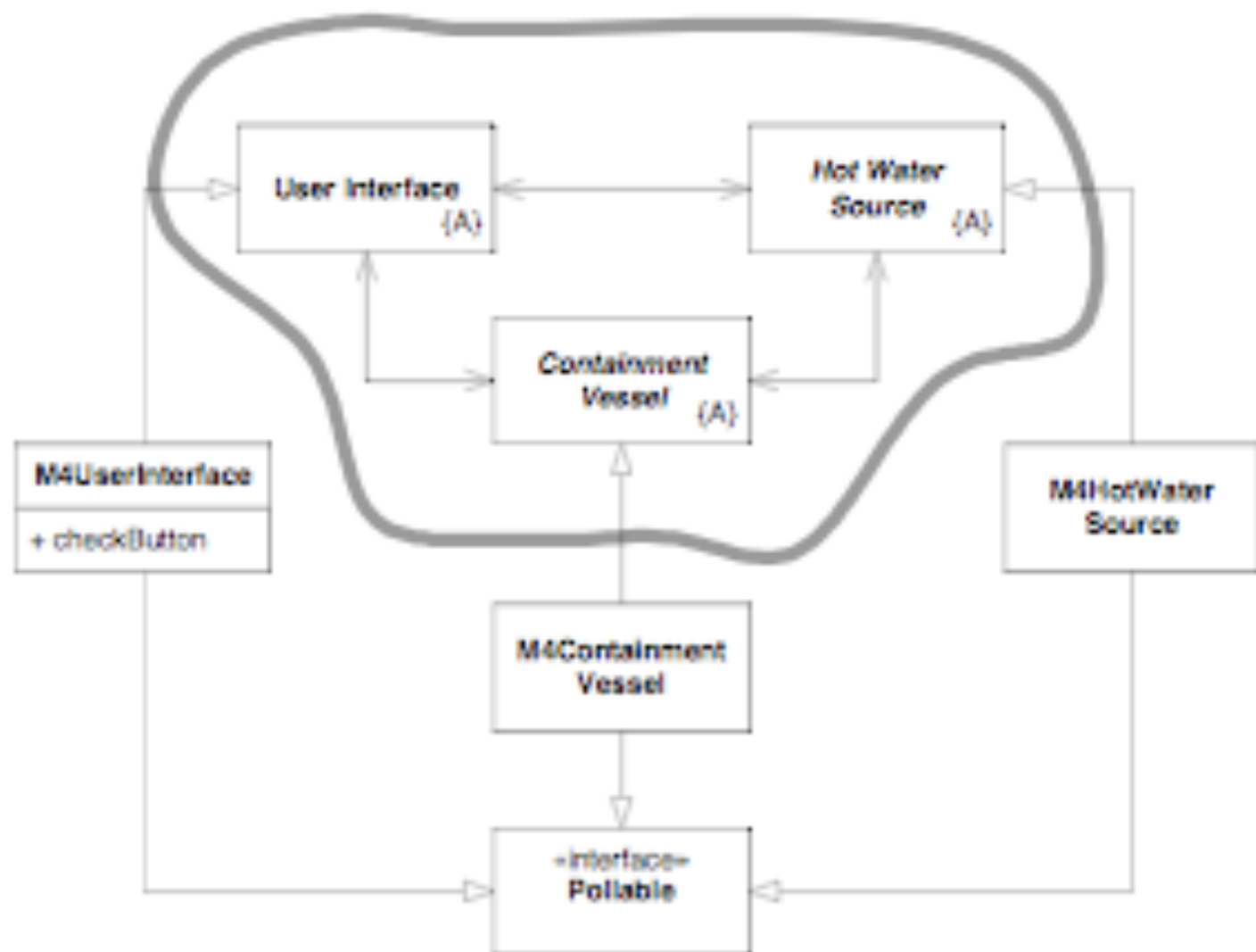
- The class model diagram represents the design of a set of related coffee makers
- We still have to design the Mark IV
- We want to do this in such a way that we don't compromise abstraction
  - *None of the three classes we have created must ever know anything about the Mark IV. This is the Dependency Inversion Principle (DIP). We are not going to allow the high-level coffee making policy of this system to depend upon the low-level implementation. -- Martin*

# Abstraction

- Just like abstract classes can be inferred from a set of related classes, so too can abstract roles be inferred
- If we represent an abstract role with an Interface, then we can make these roles tangible and reusable
- That is, a given class is nothing more than the sum of the implementations of the various roles it plays
- This is a simplified description of so called Role-Based Design

# Solution

- The solution approach is to add specific interfaces for the Mark IV that inherit from (or implement) our abstract classes
- We can also factor out the specific communication mechanism by which the sensors are accessed



# Alternative

- This problem may be too simple for OO?
- A finite-state machine model could be encoded with seven states and 18 transitions
- It would be smaller, but likely not as flexible