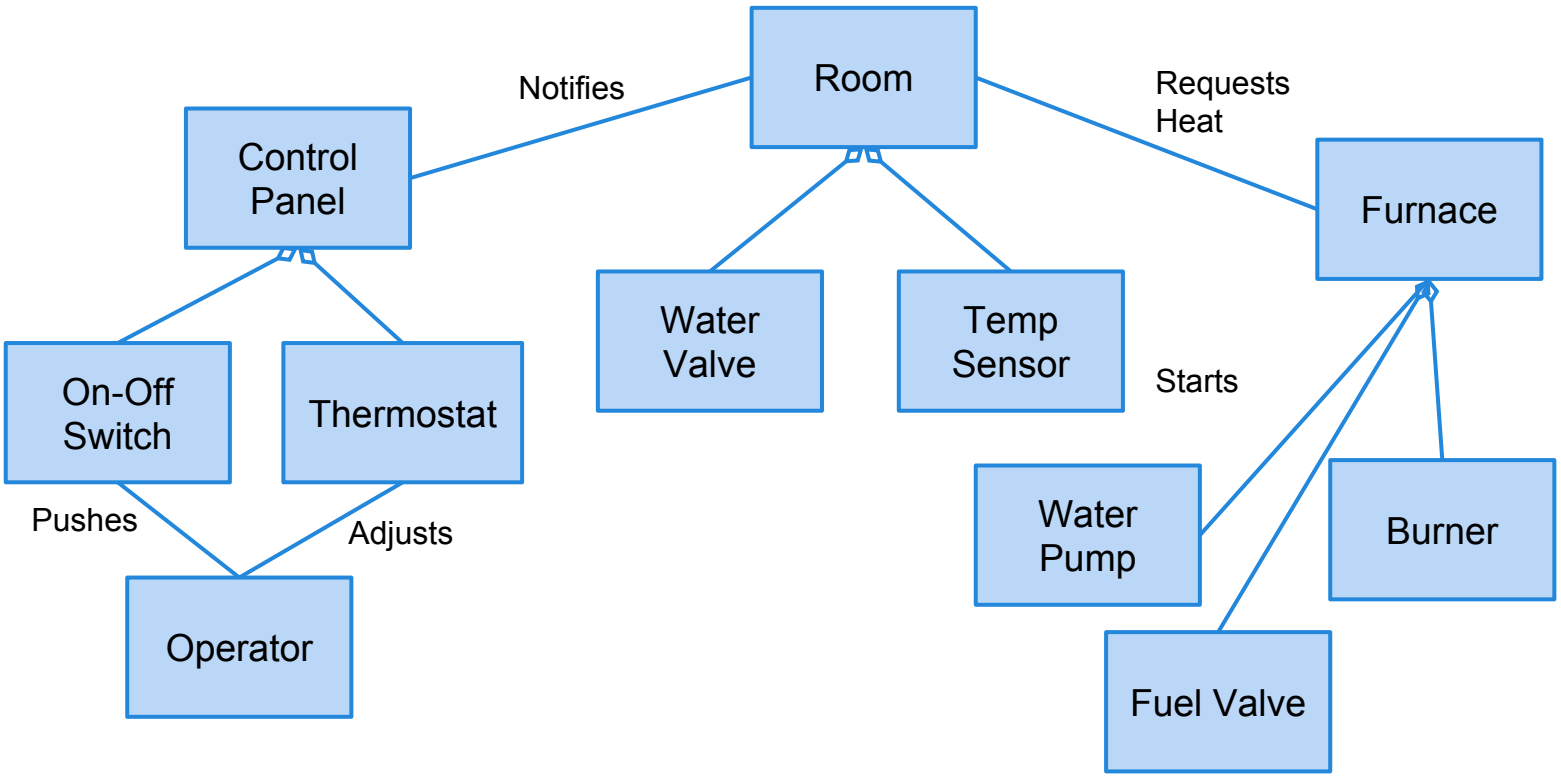
From Design to Implementation

CSCE 247 - Lecture 21 - 04/10/2019

Class Diagram



What Does This Transition Mean?

- Move away from the conceptual model and start thinking about the implementation.
  - Understanding both the static and dynamic design of your system.
  - Making final decisions on algorithms, data structures, etc.
- Refine (revise) your model so it is clear *what to build.*
- Make decisions on *how to build it.*

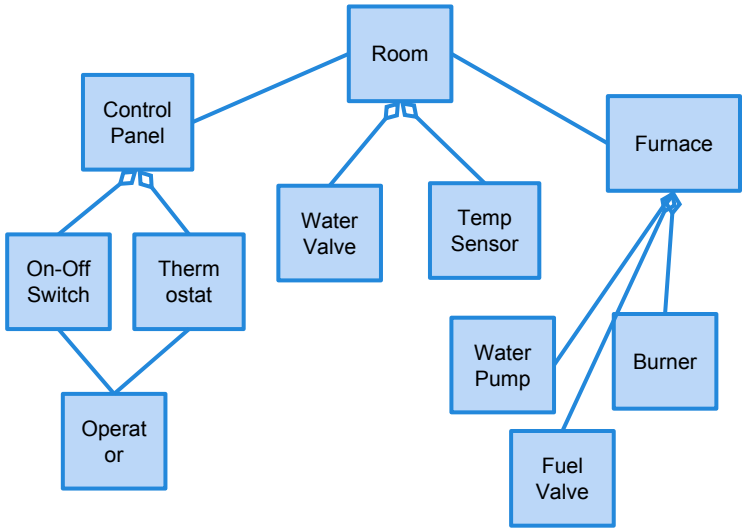
Modeling Dynamic Behavior

Overview

- Static models describe the structure of the classes (attributes, operations) and their relationships.
- Dynamic models describe how objects interact and change state, including the ordering of interactions.
- To properly implement a system, we should understand both the static and dynamic behavior.

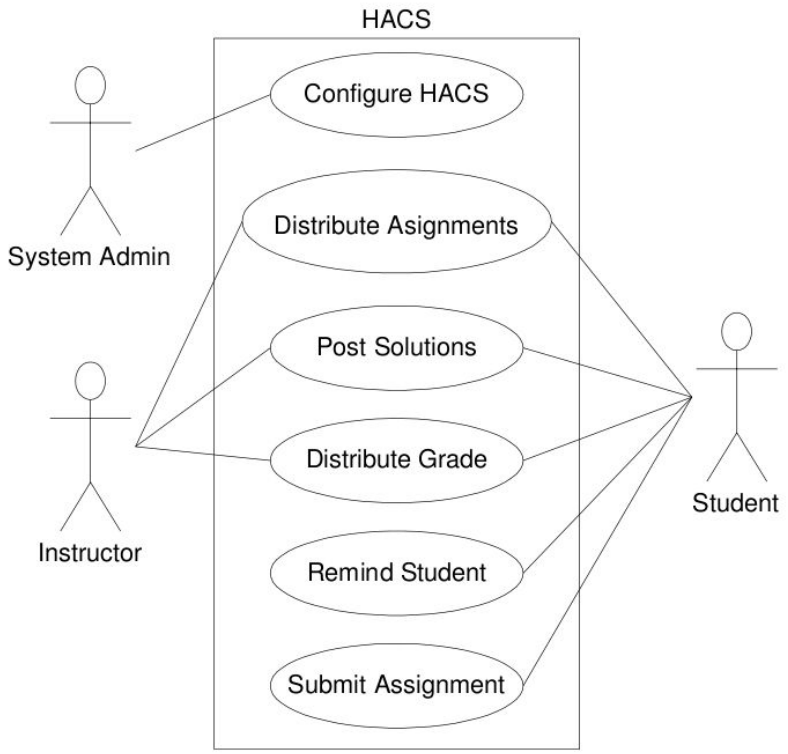
Why Model Dynamic Behavior?

- Static models tells us that Rooms request heat from a Furnace.
  - But not when
  - Or how
  - Or how often
- … and that a Furnace can start a Water Pump
  - But not under what circumstances
- Dynamic models add **context.**



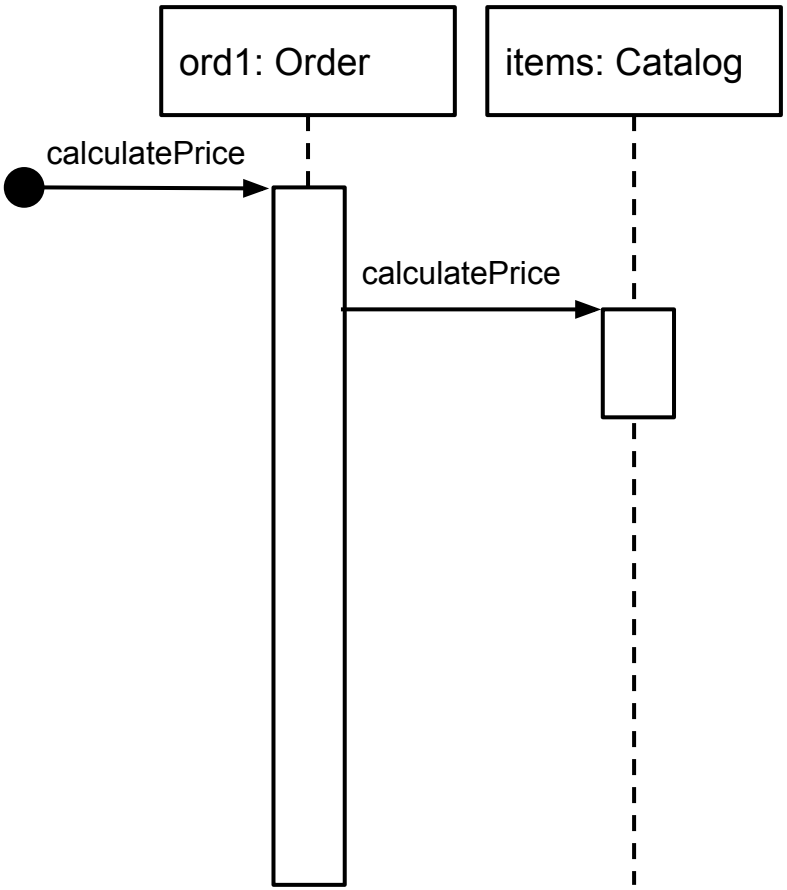
Start With The Use-Cases

- Use-cases describe functions the system can accomplish.
- Functions can be decomposed into series of actions performed internally by system classes.

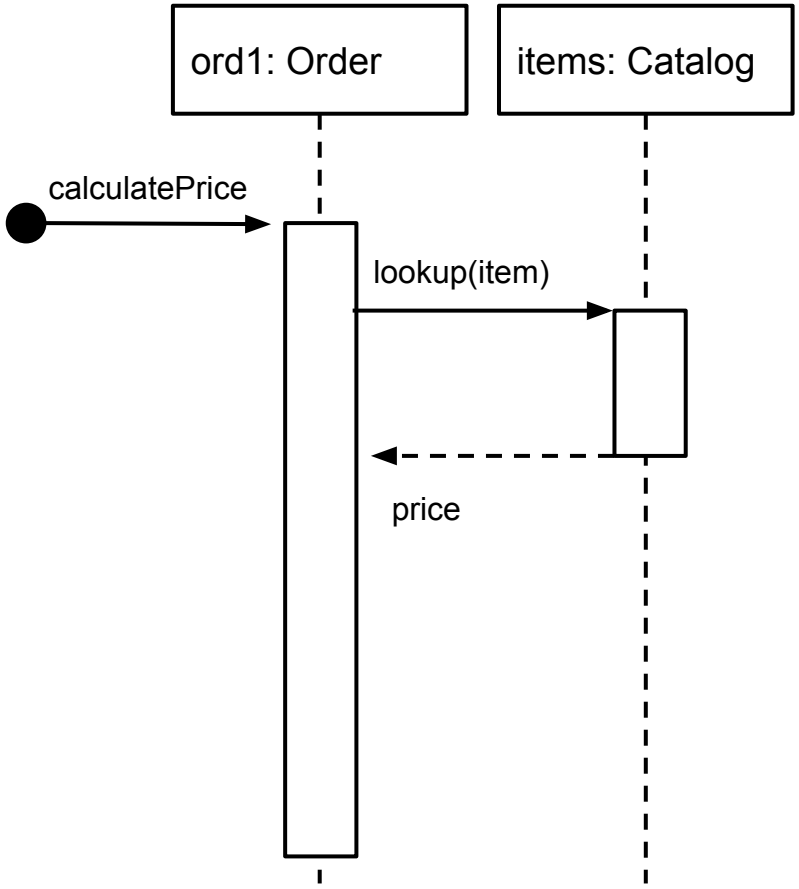


Sequence Diagrams

- Capture how the system fulfills a use case.
  - Sequence of interactions between objects within the system.
- Highlight the order and sequencing of interactions.

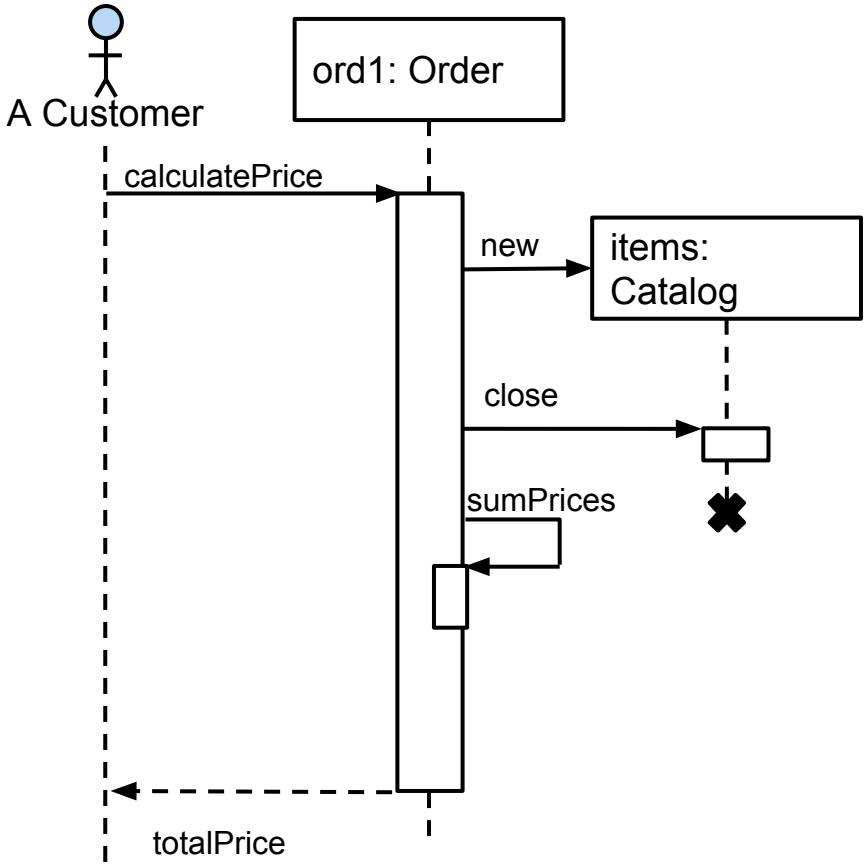


Sequence Diagram Syntax



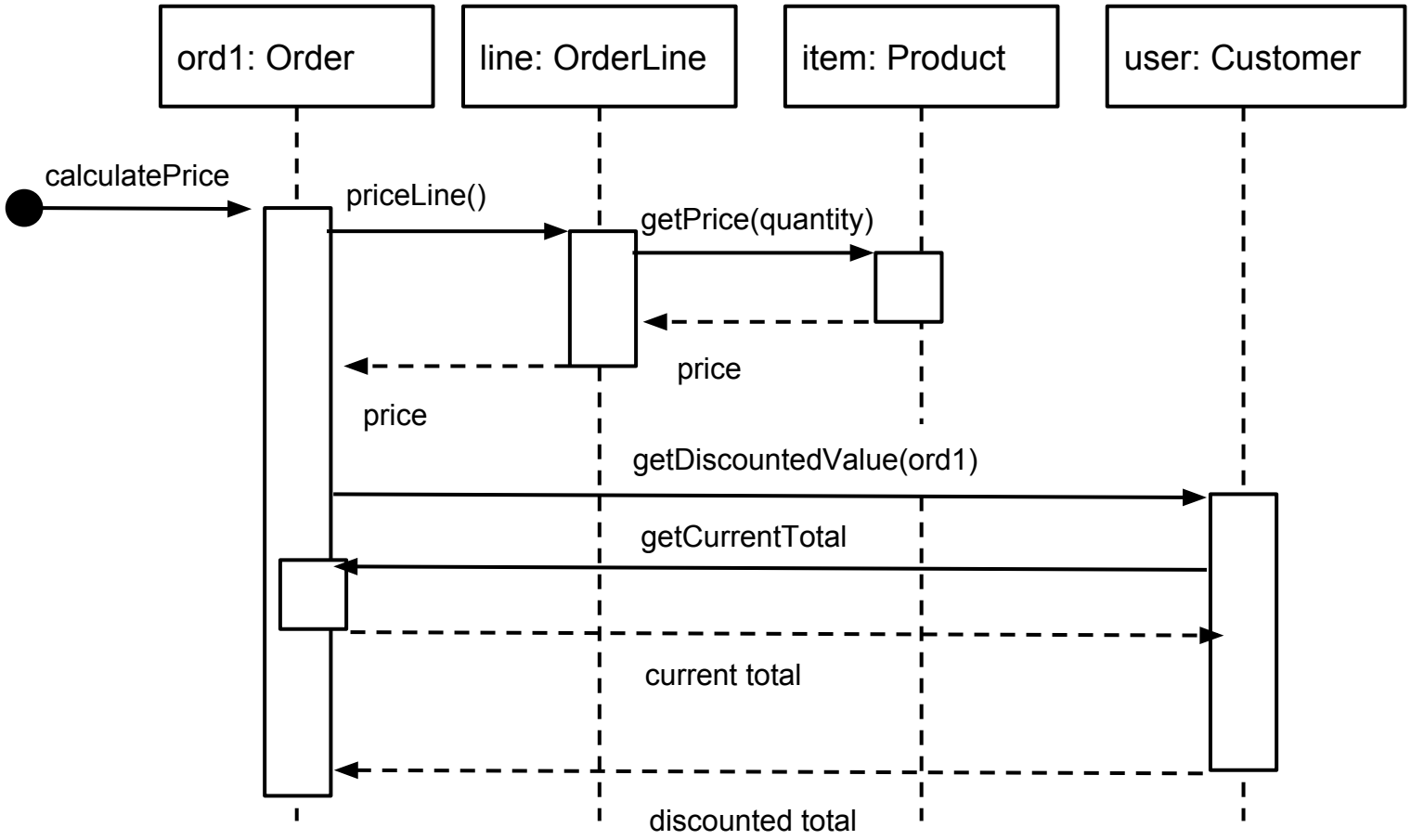
- Naming: *name : Class* or, informally, “A Class”.
- Lifeline: dashed line indicates life of the object.
- Found Message: Commands from an unmodeled source.
- Activation Box: A method is being executed.
- Message: One object calls a method offered by another object.
- Return: Information that the object returns to the calling object.

Sequence Diagram Syntax (2)



- Actors: external users/systems can be modeled as objects
- New: When an object is created, a “new” message should point to the box naming the new object.
- Close: When an object is destroyed, end its lifeline with an X.
- Self-Call: Objects can call their own methods.

Ordering Example



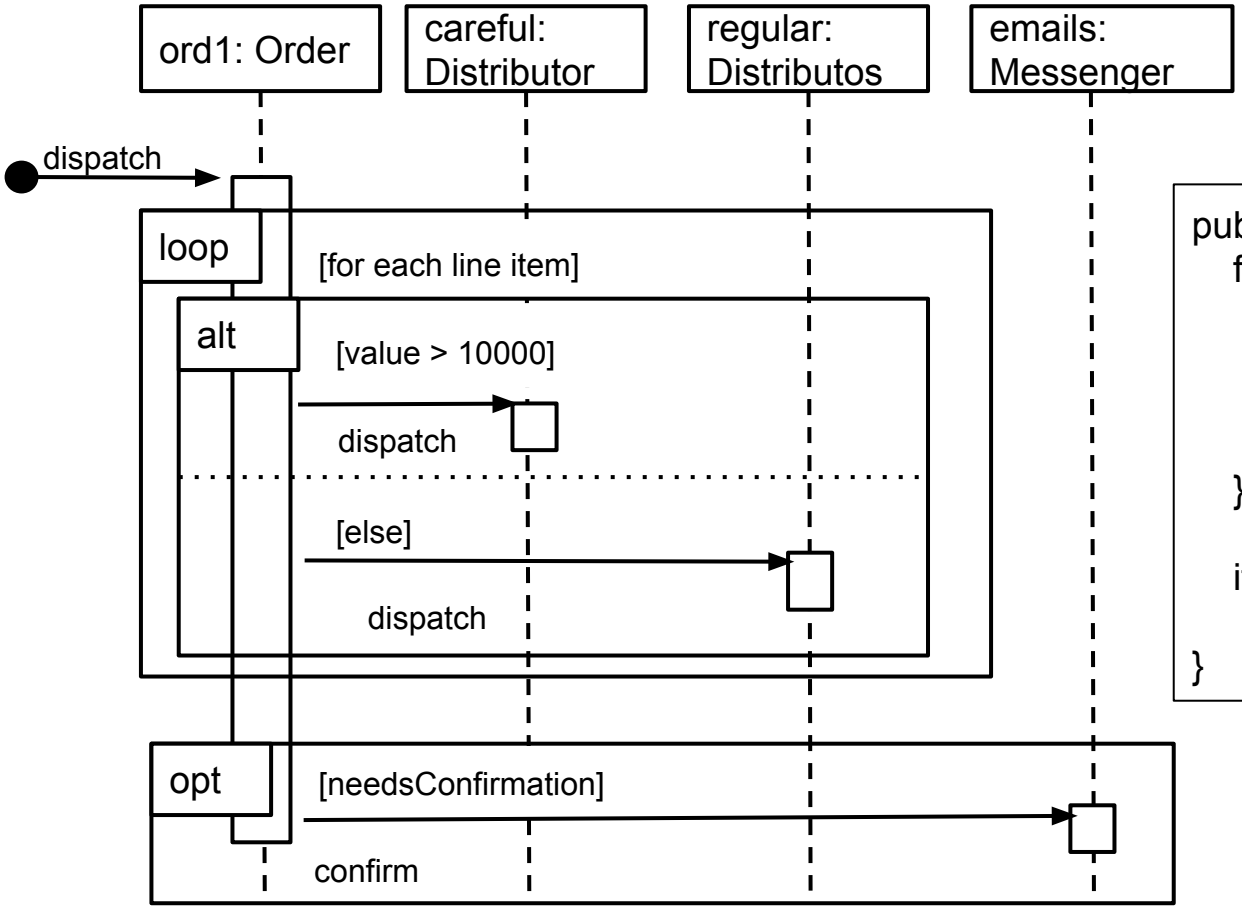
Conditional Behavior

When capturing complex scenarios, you will commonly encounter conditional behavior:

- The user does something, if this is X, do this… If Y, do this… If Z, do something else…
- For each item, do this...

Use “frames” to highlight branches in the diagram.

Loops and Conditions




```
public void dispatch(){
    for(lineitem in items){
        if (item.value > 10000)
            careful.dispatch
        else
            regular.dispatch
    }

    if (needsConfirmation)
        Emails.confirm
}
```

Frame Operators

- **alt:** Alternative paths, only one will execute.
- **opt:** Optional set of interactions.
- **loop:** Set of interactions executes multiple times.
- **par:** Each indicated set of interactions will execute in parallel.
- **region:** Critical region, only one thread can execute this interaction sequence at once.
- **neg:** This set of interactions can never legally happen.
- **ref:** Used to refer to a set of interactions depicted on another diagram.

Home Heating Use Case

Use Case: Power Up

Actors: Home Owner

Description:

1. The Home Owner moves the power switch to the “on” position.
2. The system responds with a “system ready” status message if it starts successfully.

Class Diagram - v1

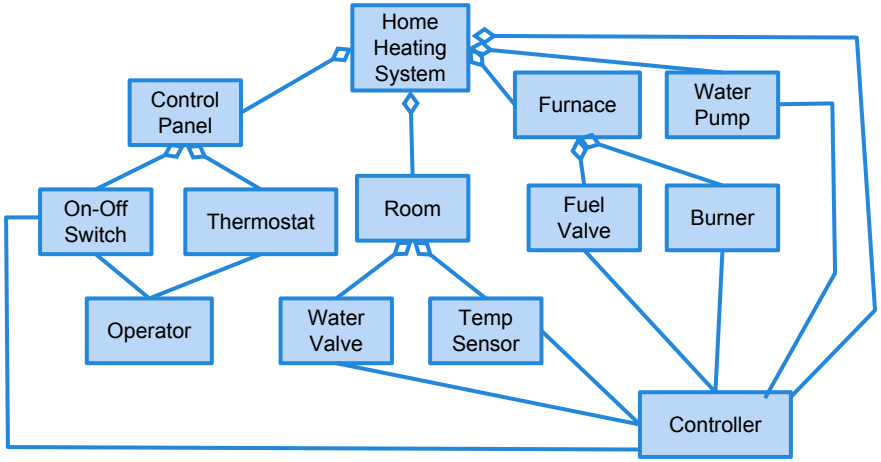
Use Case: Power Up

Actors: Home Owner

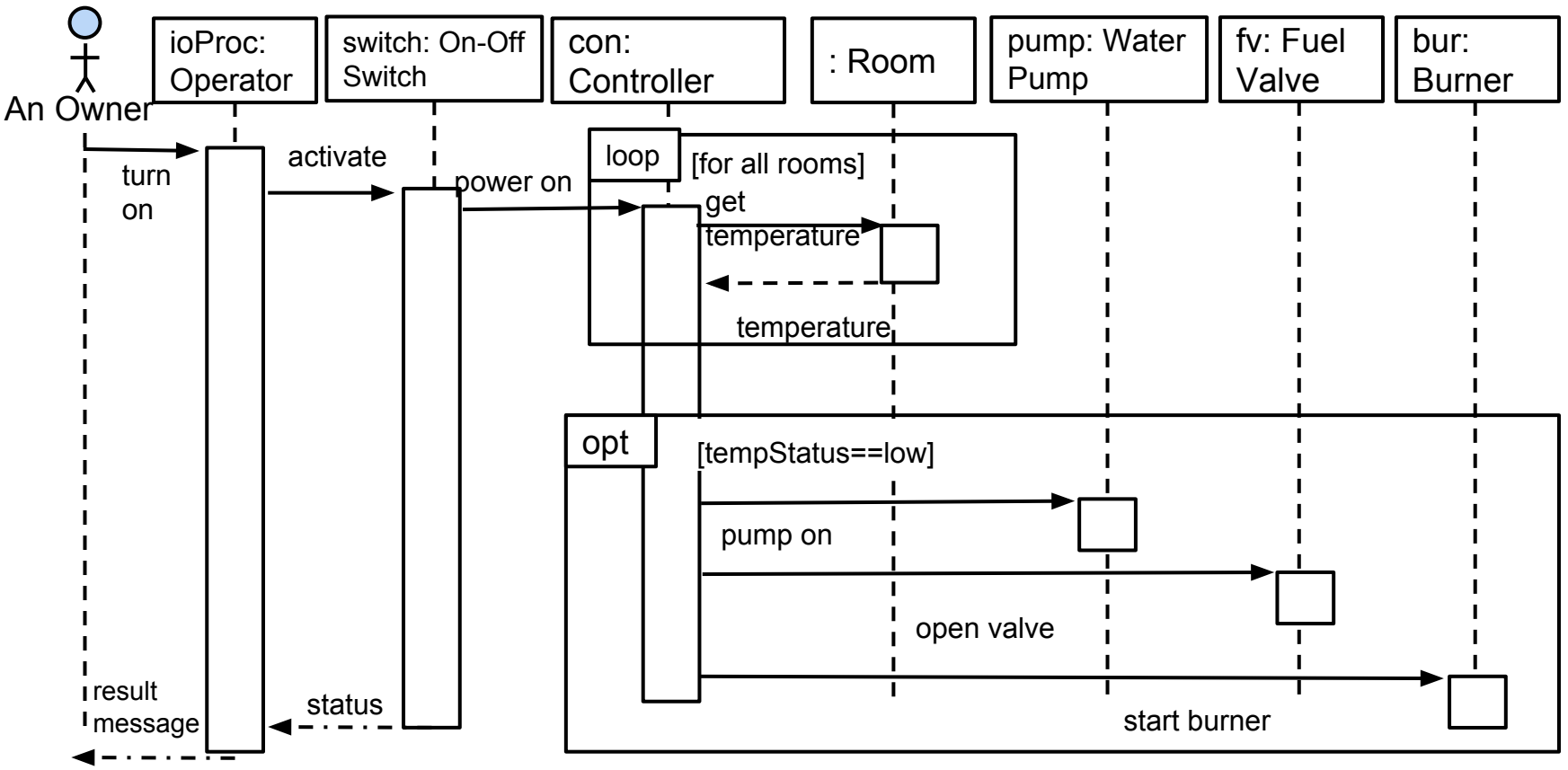
1. The Home Owner moves the power switch to the “on” position.
2. The system responds with a “system ready” status message if it starts successfully.

Related Requirement:

An operator class processes input signals. When the power is turned on, each room is temperature checked. If a room is below the desired temperature, the valve for the room is opened, the water pump started, the fuel valve opened, and the burner ignited.



Sequence Diagram - v1



Class Diagram - v2

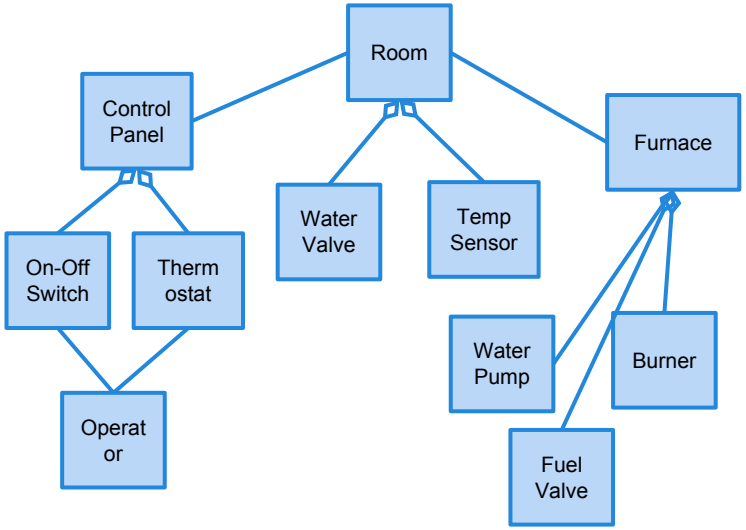
Use Case: Power Up

Actors: Home Owner

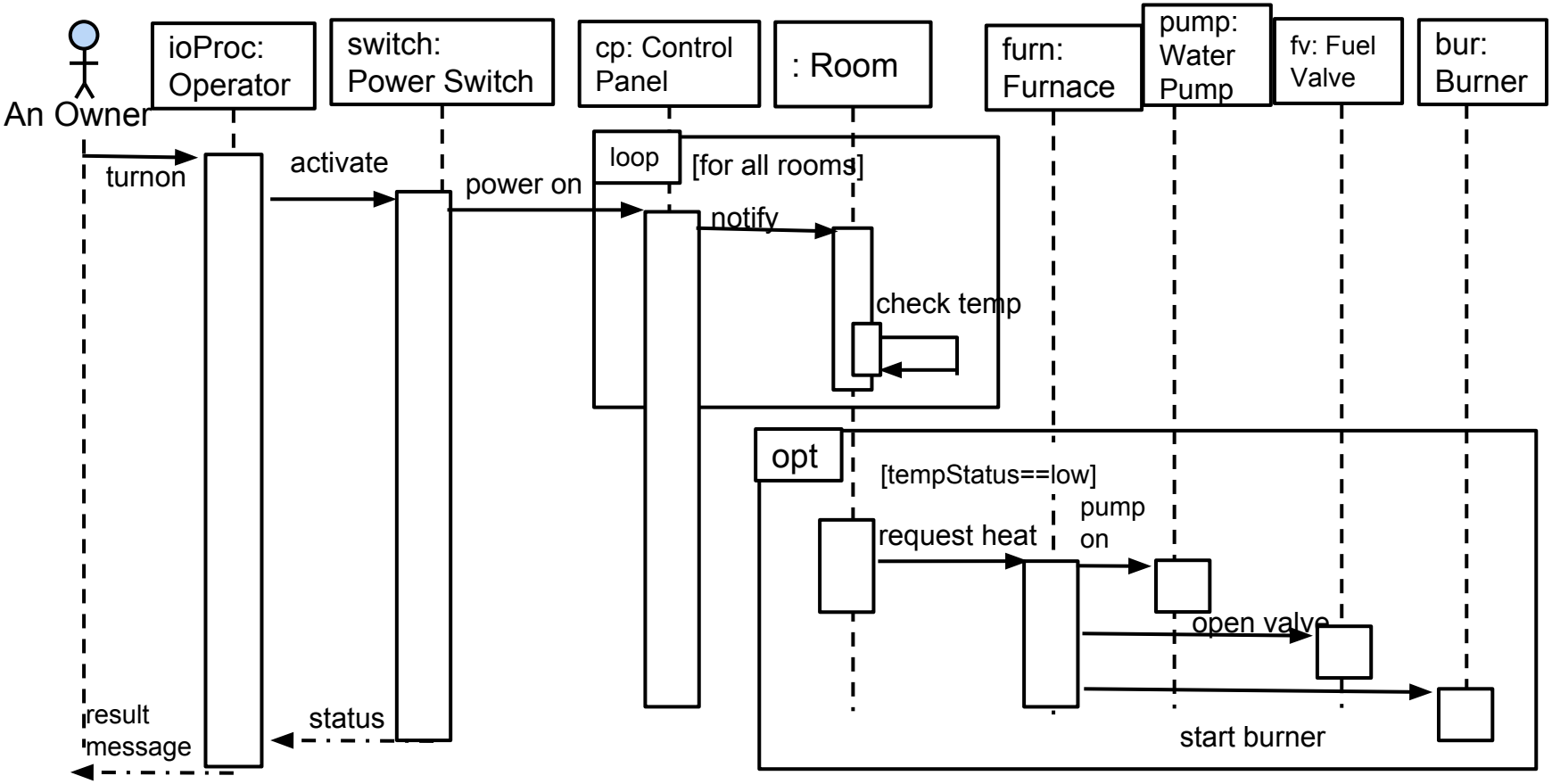
1. The Home Owner moves the power switch to the “on” position.
2. The system responds with a “system ready” status message if it starts successfully.

Related Requirement:

An operator class processes input signals. When the power is turned on, each room is temperature checked. If a room is below the desired temperature, the valve for the room is opened, the water pump started, the fuel valve opened, and the burner ignited.



Sequence Diagram - v2

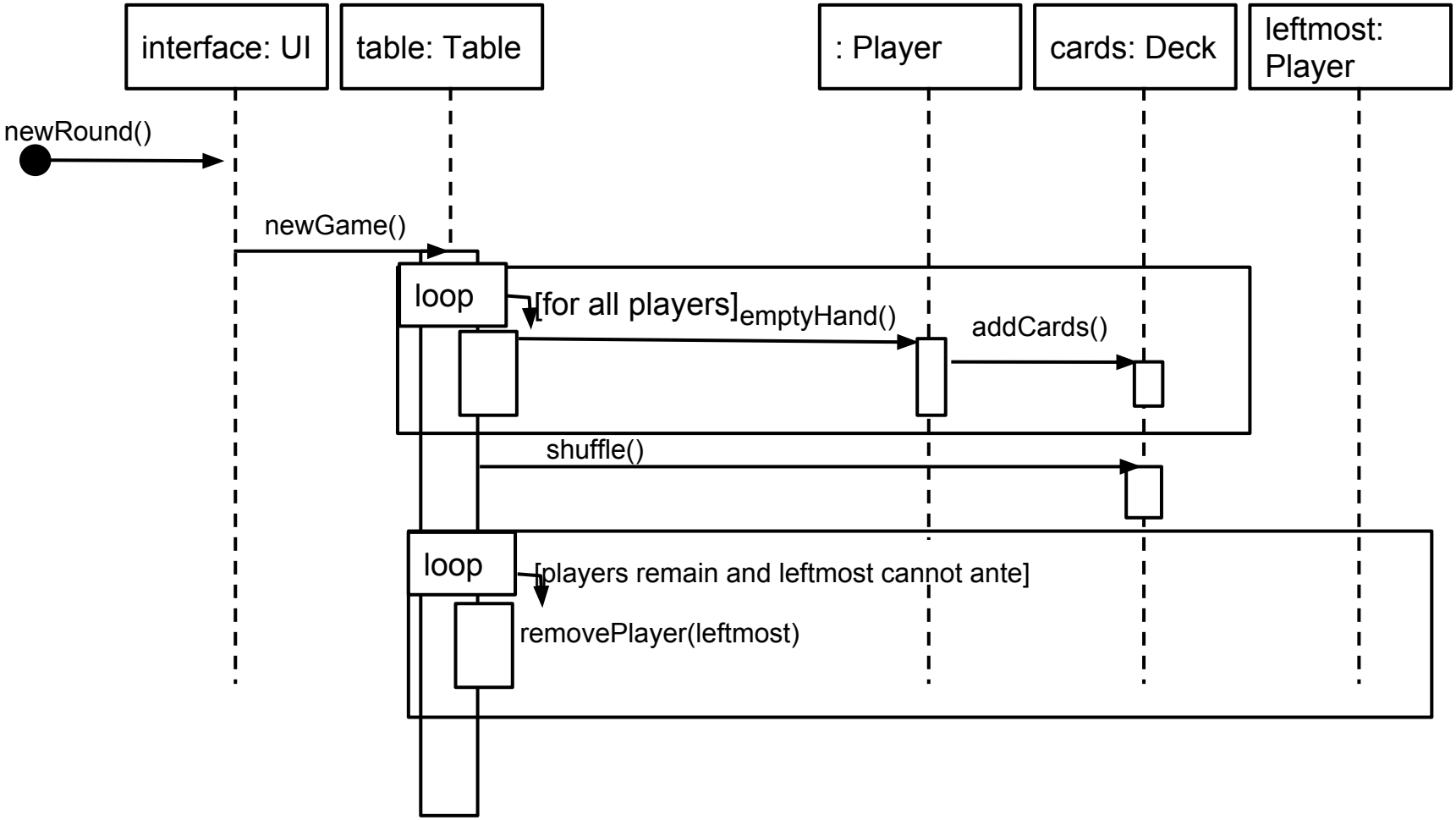


Example - Poker Hand

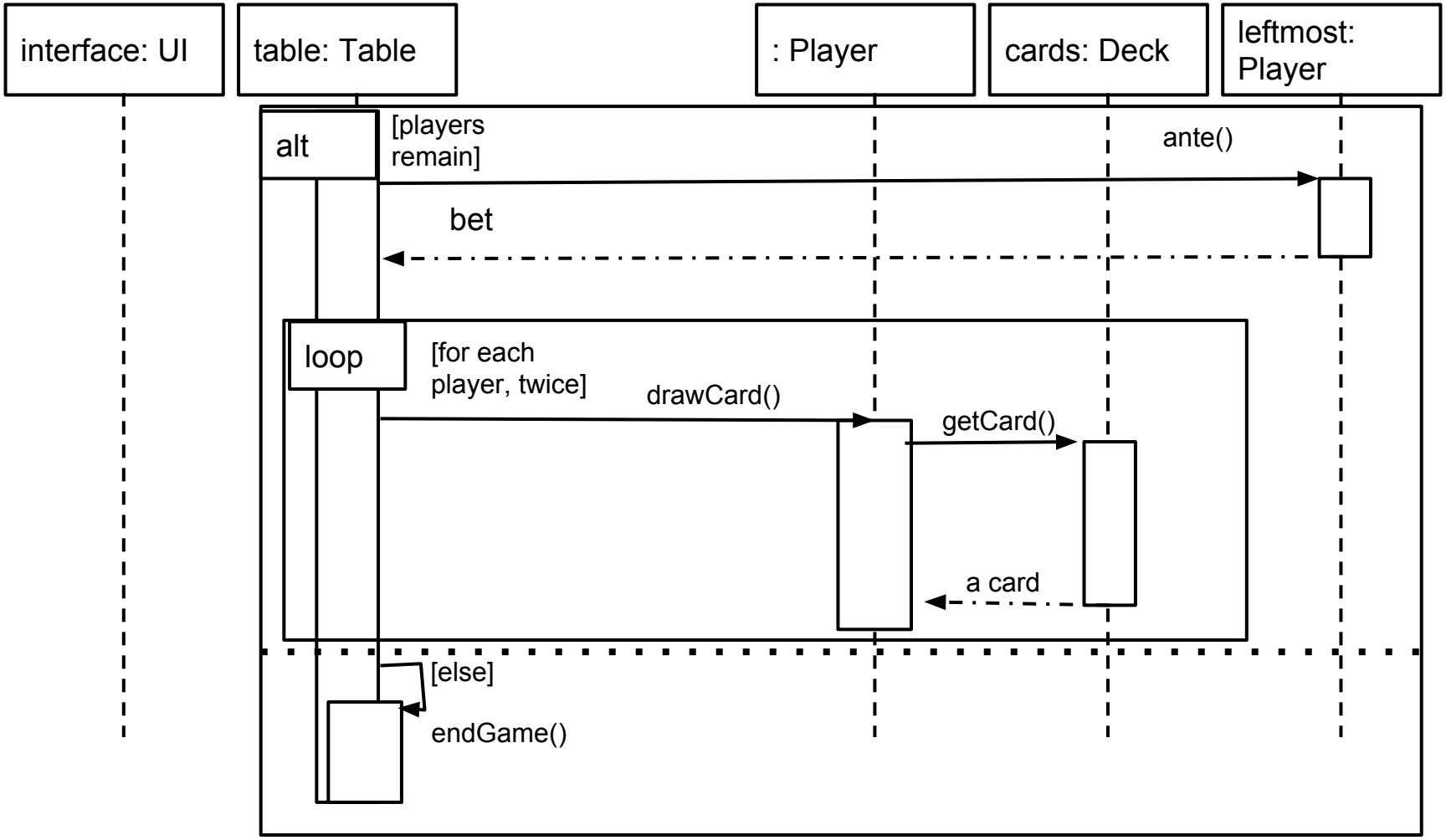
Starting a New Game Round

- The scenario begins when a request for a new round is sent to the UI.
- All players' hands are emptied into the deck, which is then shuffled.
- The player left of the dealer supplies an ante bet of the proper amount.
- Next each player is dealt a hand of two cards from the deck in a round-robin fashion; one card to each player, then the second card.
- If the player left of the dealer doesn't have enough money to ante, he/she is removed from the game, and the next player supplies the ante.
- If that player also cannot afford the ante, this cycle continues until such a player is found or all players are removed.

Example - Poker Hand



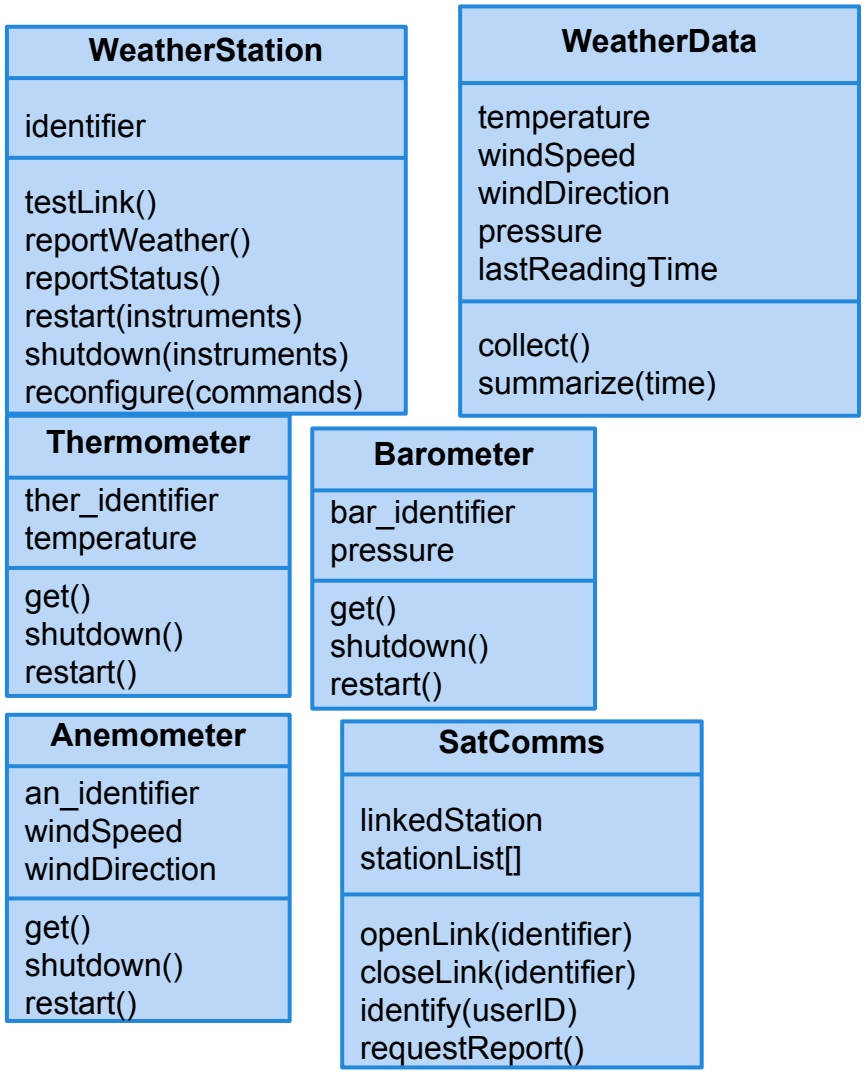
Example - Poker Hand



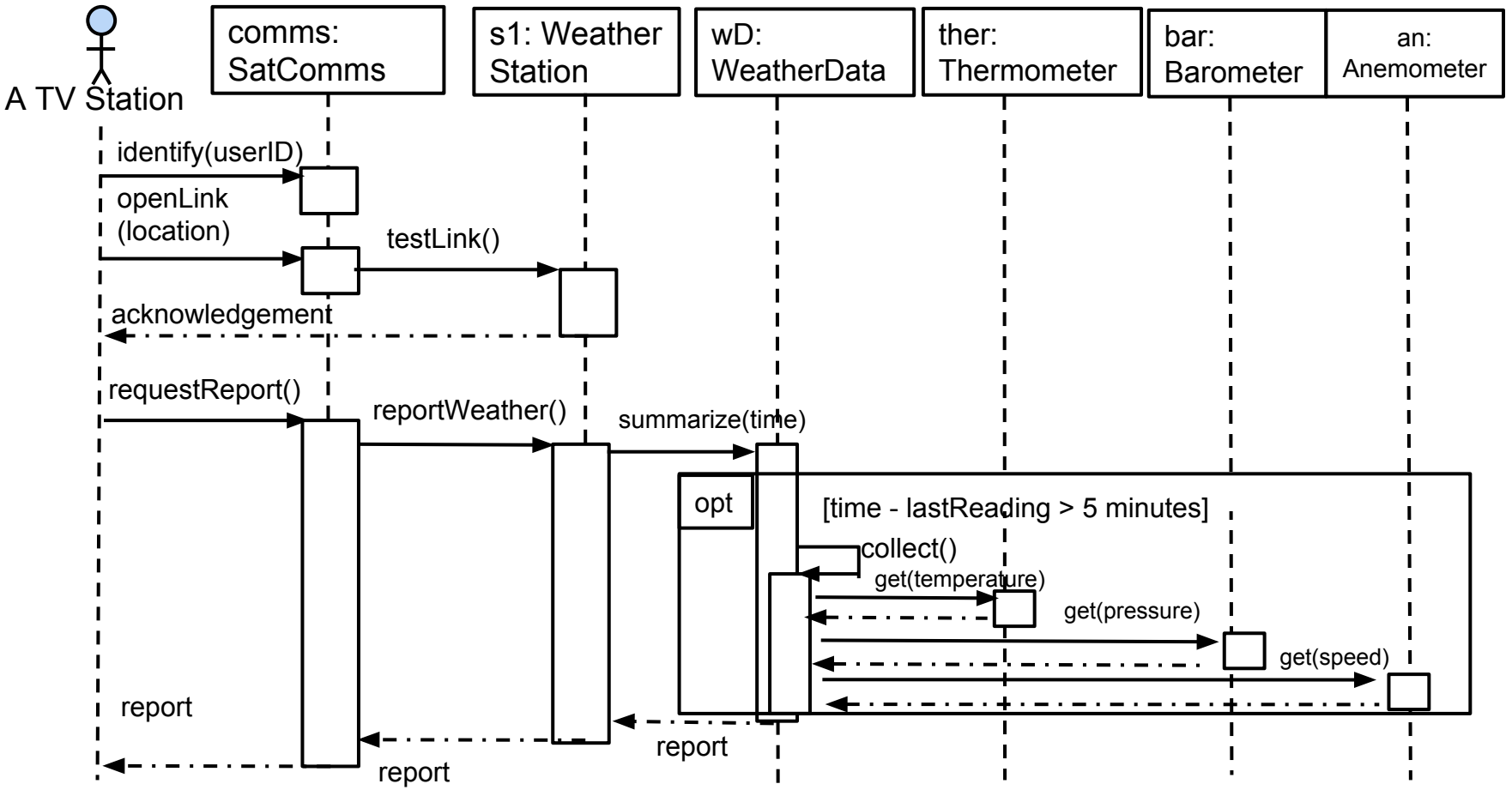
Sequence Diagram Activity

A local television station opens a communication link to the SatComms module, identifies itself, and requests a link to the WeatherStation instance for a particular location. It then requests a weather report from the WeatherStation. The WeatherStation requests a summary from that location's WeatherData instance. If new readings have not been take in the last five minutes, then the WeatherData will gather new values for its attributes. WeatherData will then return its summary.

Draw a sequence diagram for this scenario using these classes.



Activity Solution



Preparing for Implementation

Choosing Data Structures

Design documents detail *what is being stored*, but not *how to store it.*

Choice of data structure matters:

- Storage and operation costs
- Suitability to problem (and what data is being stored)
- Many guidelines out there - key is to think through the problem and your priorities (ease-of-use vs efficiency)

Choosing Algorithms

Design gives you *what a method should do*, implementation concerns *how to code it to do that.*

Many ways to solve a problem, think carefully about choice.

- Good design may suggest certain realization.
- Be prepared to trade efficiency for maintainability or understandability.

Error-Prone Constructs

Use these, but use them with great care.

- Floating-point numbers
  - Inherently imprecise. The imprecision may lead to invalid comparisons.
- Pointers
  - Pointers referring to the wrong memory areas can corrupt data.
  - Aliasing can make programs difficult to understand and change.

Error-Prone Constructs

- Dynamic memory allocation
  - Run-time allocation can cause memory overflow and garbage collection issues.
- Parallelism
  - Can result in subtle timing errors because of unforeseen interaction between parallel processes.
- Recursion
  - Errors in recursion can cause memory overflow.
- Interrupts
  - Can cause a critical operation to be terminated and make a program difficult to understand.

Implementation Structure (Packages)

Packages are groupings of classes that provide structure and organization to the source code.

Allows developers to:

- Determine what types are related and dependent
- Find classes that provide certain functionality
- Know that the class names will not conflict
- Share data freely between package members

What does this do?

```
#include <stdio.h>
main(t,_,a)
char *a;
{
return!0<t?t<3?main(-79,-13,a+main(-87,1-_,main(-86,0,a+1)+a)):
1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13? main(2,_+1,"%s %d
%d\n"):9:16:t<0?t<-72?main(_,t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l+,/n{n+,/+#n+,/#
\ ;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l \
q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n'){)#}w'){){nl]'/+#n';d}rw' i;#
\ ){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw' \
iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlw]!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w! nr'/ ') }+}{rl#'{n'
')# \ }'+}##(!!/")
:t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1)
:0<t?main(2,2,"%s"):*a=='/'||main(0,main(-61,*a,
"!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);
}
```

What about this?

```
int m,u,e=0; float
l,_,I;main(){for(;e<1863;putchar((+
+e>923&&952>
e?60-m:u)["\n)ed.fsg@eum(rezneuM
drahnreB"]))for(u=_=l=0;(m=e%81)
<80&&I*l+_*_<6&&20>++u;_=2*l*_+e/81
*.09-1,l=I)I=l*l-_*_-2+m/27.;}
```

Writing Good Code - Style Guide

- Variable Naming:
  - Use camel case
    - variableName
    - ClassOrEnumOrInterfaceName
  - Names should be easily readable
    - Descriptive, favor long name over abbreviation.
- Brackets - pick one:

```
try{
   // do stuff
}
```

```
try
{
   // do stuff
}
```

Writing Good Code - Style Guide

- Indentation
  - DO indent, but use spaces instead of tabs!
  - How many? Four spaces is common. Pick a number and stick to it.

- Fully qualify import statements

Bad:

```
import java.util.*;
import org.apache.foo.*;
```

Good:

```
import
java.util.ArrayList;
import org.apache.foo.Bar;
```

Writing Good Code - Documentation

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param  url  an absolute URL giving the base
 * @param  name the location of the image, relati
 * @return      the image at the specified URL
 * @throws    MalformedURLException if image U
 * @see         Image
 */
public Image getImage(URL
      try {
         return getImage(new URL(url, name));
      } catch (MalformedURLException e) {
         return null;
      }
}
```

Method Description

describe a parameter.

<variable name>
on>

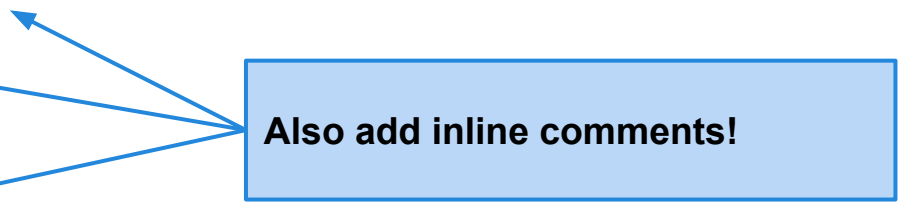
@ return - describes what is being returned

@see - cross-reference another class.

@throws - describe the exceptions being thrown and why they are thrown

Writing Good Code - Documentation

```
/**
 * <<Description>>
 *
 * @param  url  an absolute URL giving the base location of the image
 * @param  name the location of the image, relative to the url argument
 * @return      the image at the specified URL
 * @see         Image
 */
public Image getImage(URL url, String name) {
    // The URL might be malformed, so make sure we check for that.
    try {
        // Try to retrieve the image from the URL.
        return getImage(new URL(url, name));
    } catch (MalformedURLException e) {
        // If we can't get the image, return a null value
        return null;
    }
}
```



Code Reuse

Most modern software is constructed, in part, by reusing existing components or systems.

- When developing software, consider how to make use of existing code.
- Possible at many levels of development.
- Be careful - many problems and costs associated with reuse.

Code Reuse Levels

1. Abstraction Level
   Use knowledge from similar projects in your system design (design/architectural patterns)
2. Object Level
   Import individual objects and functions from libraries and use them in your project.
3. Component Level
   Incorporate collections of objects and adapt them to your needs.
4. System Level
   Reuse complete applications, wired together with scripting.

Costs of Code Reuse

- The time spent looking for software to reuse and addressing whether it fits your needs can be high.
- Buying and licensing software for reuse can be expensive.
- Cost of adapting and configuring the reusable components to fit your requirements can be more expensive than coding yourself.
- Integrating reused systems with each other and with your new code can result in new defects.

Host-Target Development

Most software is developed on one type of computer (the host) and deployed on different types of computers (targets).

- For embedded systems, the target is **very** different from the host.
- For desktop applications, still need to consider a wide variety of target environments.

Target Support Issues

- The hardware and software requirements of a component.
  - If a component is designed for specific hardware architecture, requires certain CPU/RAM/GPU or special software, make sure assumptions are clearly stated.
- The availability requirements of the system.
  - Components may be deployed on multiple platforms. Make sure an alternative implementation of the component is available if one fails.
- Component Communications
  - If distributed components must communicate, try to install them on a single system or ensure geographically close servers exist.

Managing Change

Change happens all the time, so managing change is essential.

- When teams work together, their work must not conflict.
  - Changes must be coordinated. Otherwise, one programmer may overwrite the other's work.
  - Everybody must have access to the most up-to-date versions of all project components.
- If something is broken, we should be able to go back to the working version.

Configuration Management

The process of managing a changing system.
Three fundamental activities:

1. Version Management
   Different versions of system components are tracked. Coordinates development by several programmers. Prevents overwriting of code.
2. System Integration
   Support is provided to help developers define what versions of a component are used to create a system build. Supports automated builds by linking components.
3. Problem Tracking
   Allow users to report bugs and other problems, and allow developers to see who is working on these problems.

Summary

- Move away from the conceptual model and start thinking about the implementation

- Refine (revise) your model so it is clear *what to build*
- Make decisions on *how to build it*

We Have Learned

- Dynamic modeling allows us to design how the system acts during execution.
  - Sequence diagrams allow modeling of detailed object interactions.
- These provide context to the static structural diagrams.
- In preparing for implementation, consider trade-offs in choosing algorithms and language structures, and in code reuse.

Next Time

- Structure-based Testing
- Reading:
  - Sommerville, ch. 8
- Assignment 4 - Due April 21
  - Use the lessons from today in implementing the assignment.