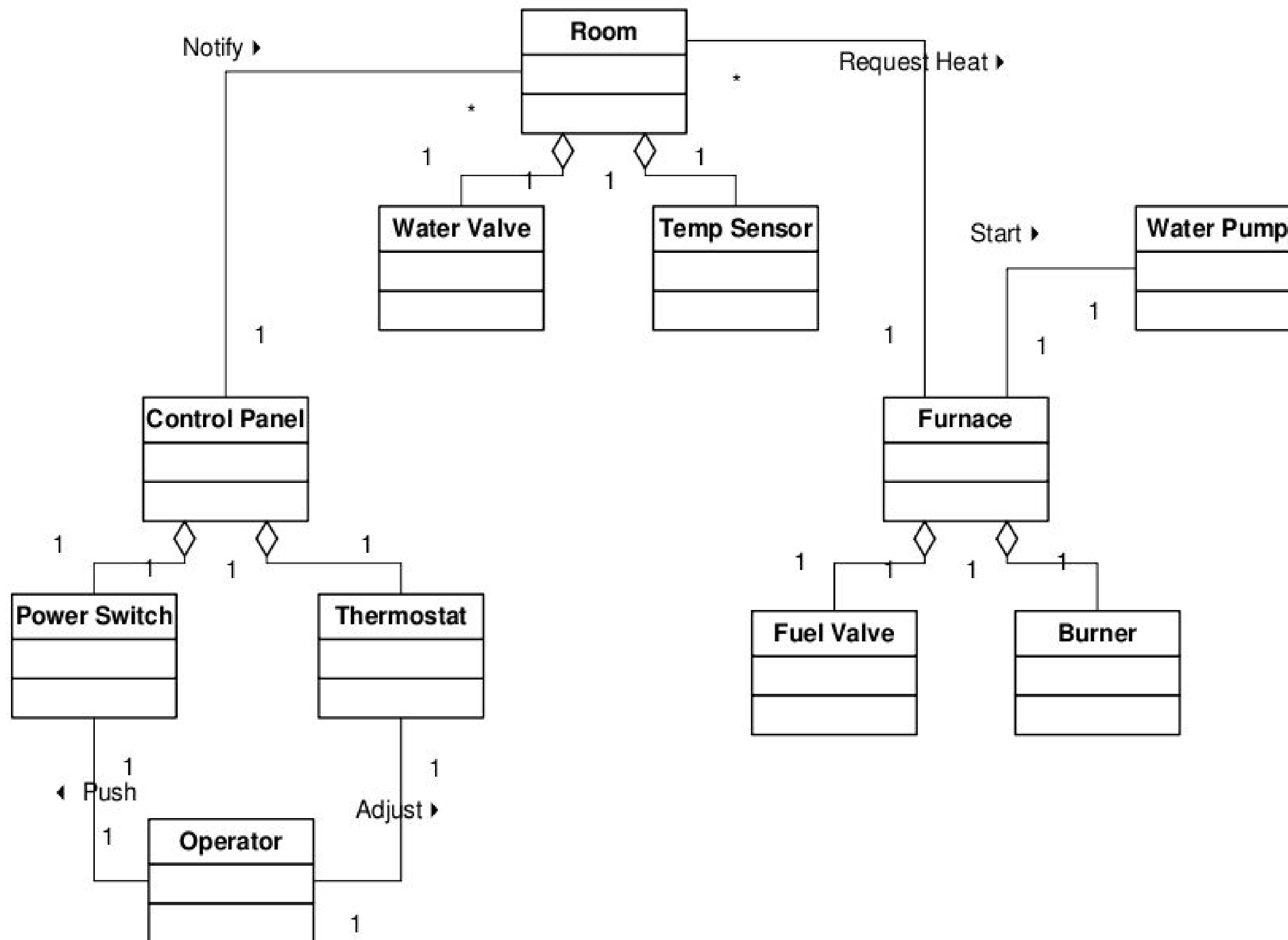
Dynamic Modeling

CSCE 740 - Lecture 19 - 11/04/2015

Class Diagram

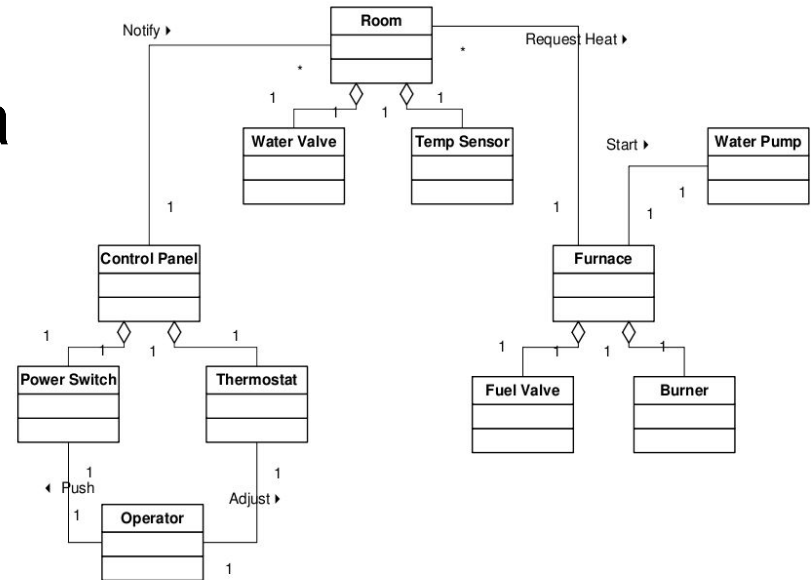


Overview

- 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.
- Today, we will discuss using sequence diagrams to examine dynamic behavior.

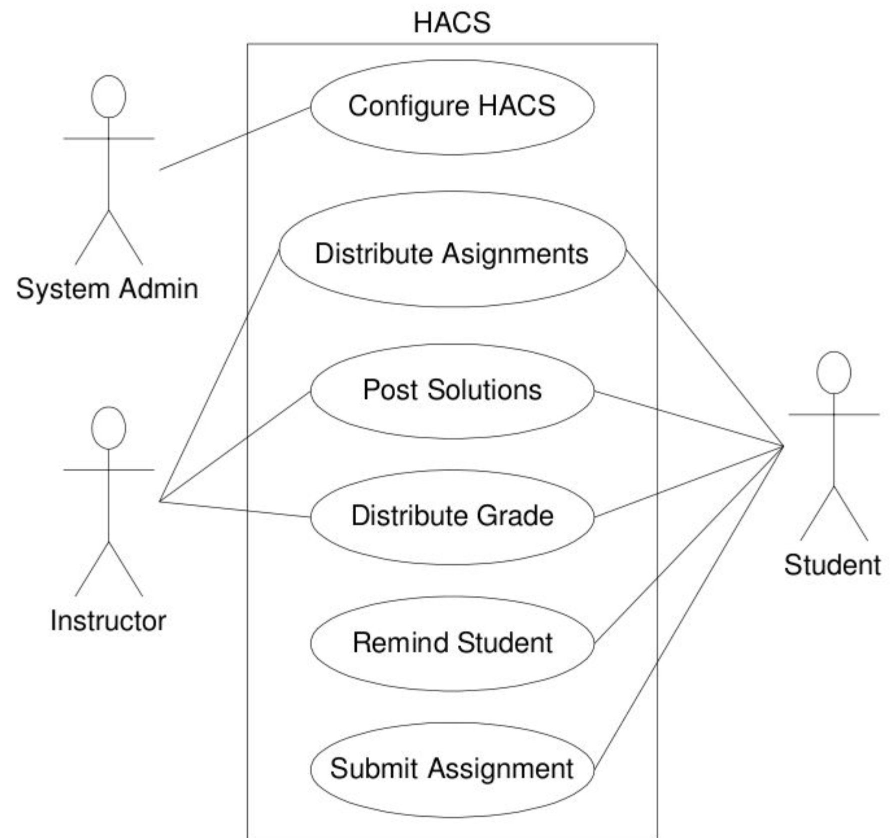
Why Model Dynamic Behavior?

- 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.**



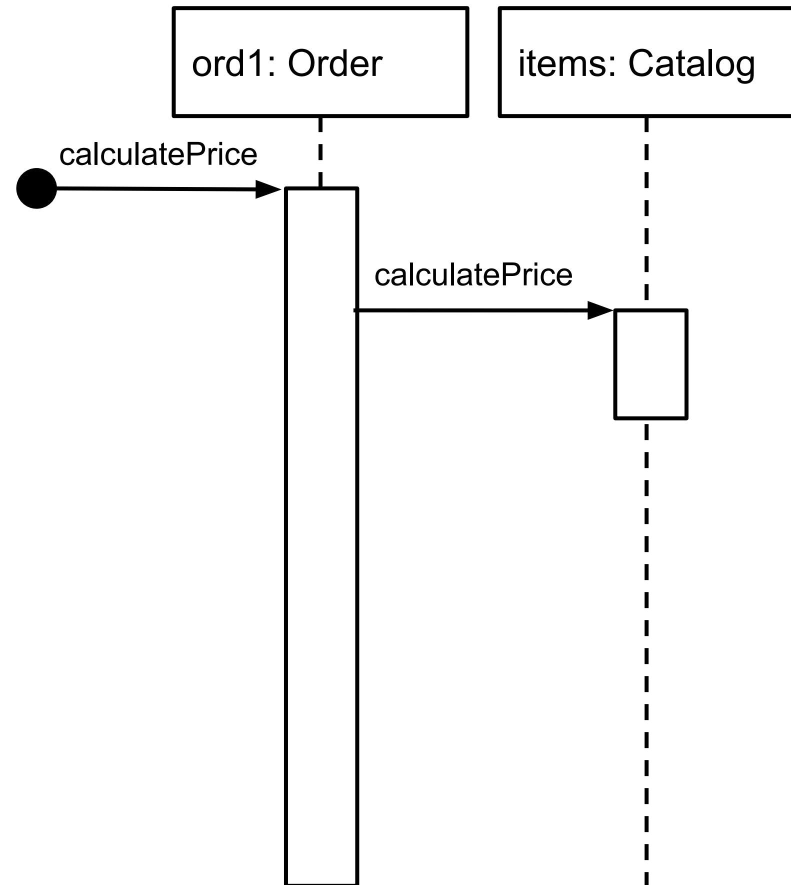
Start With The Use-Cases

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

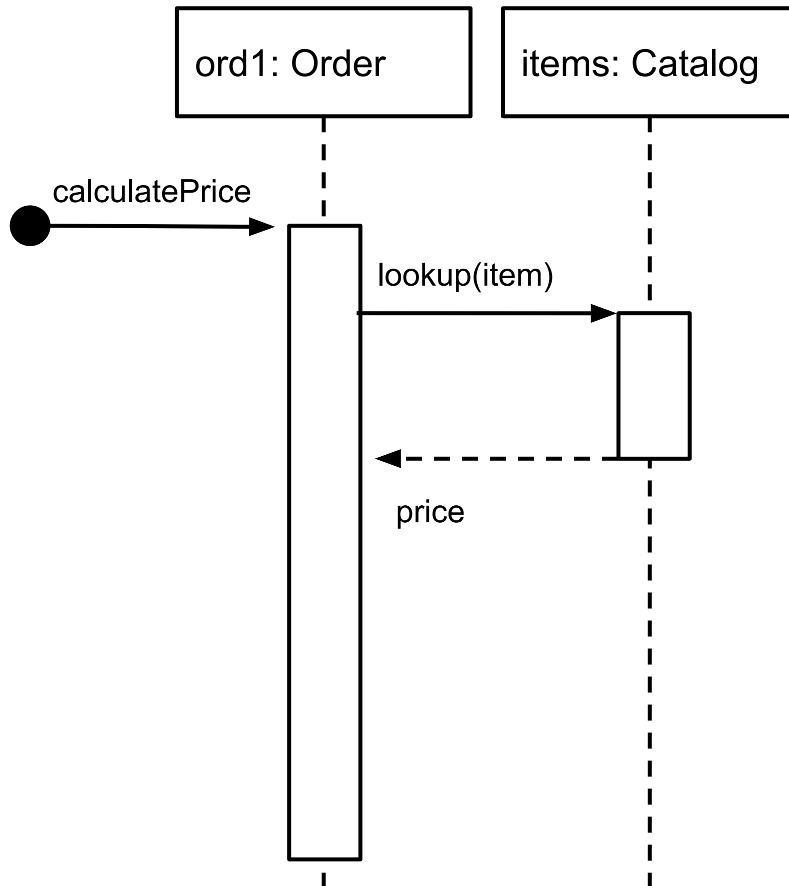


Sequence Diagrams

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

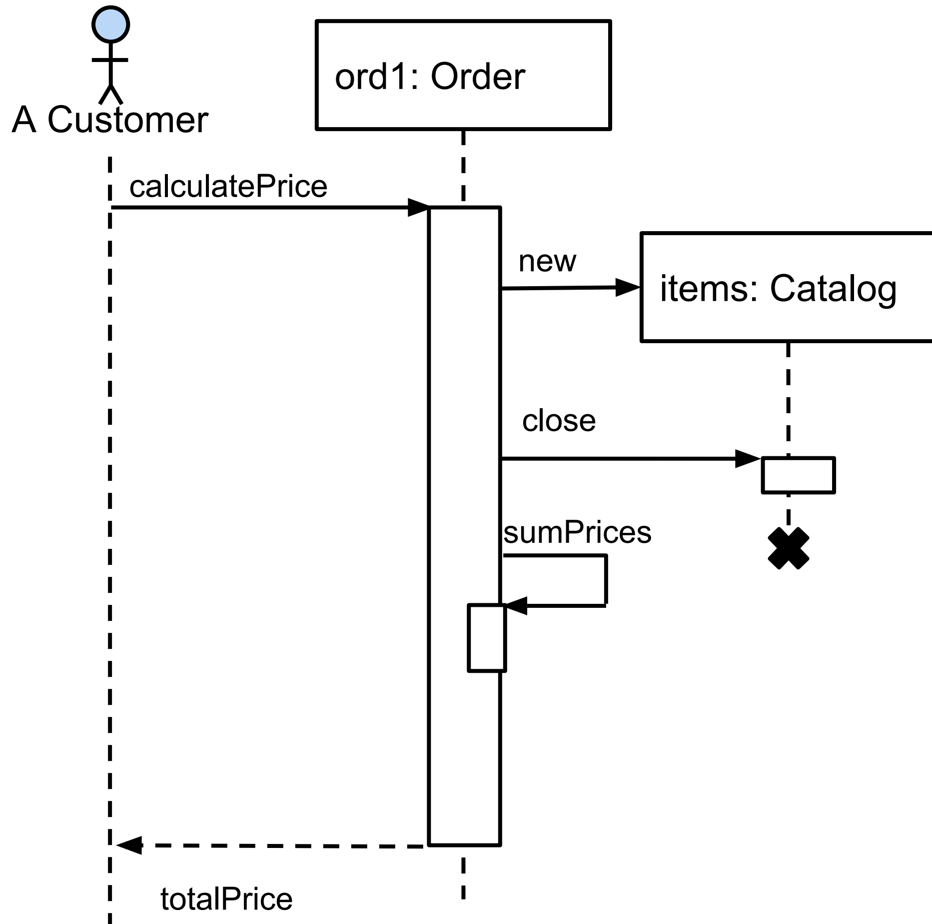


Sequence Diagram Syntax



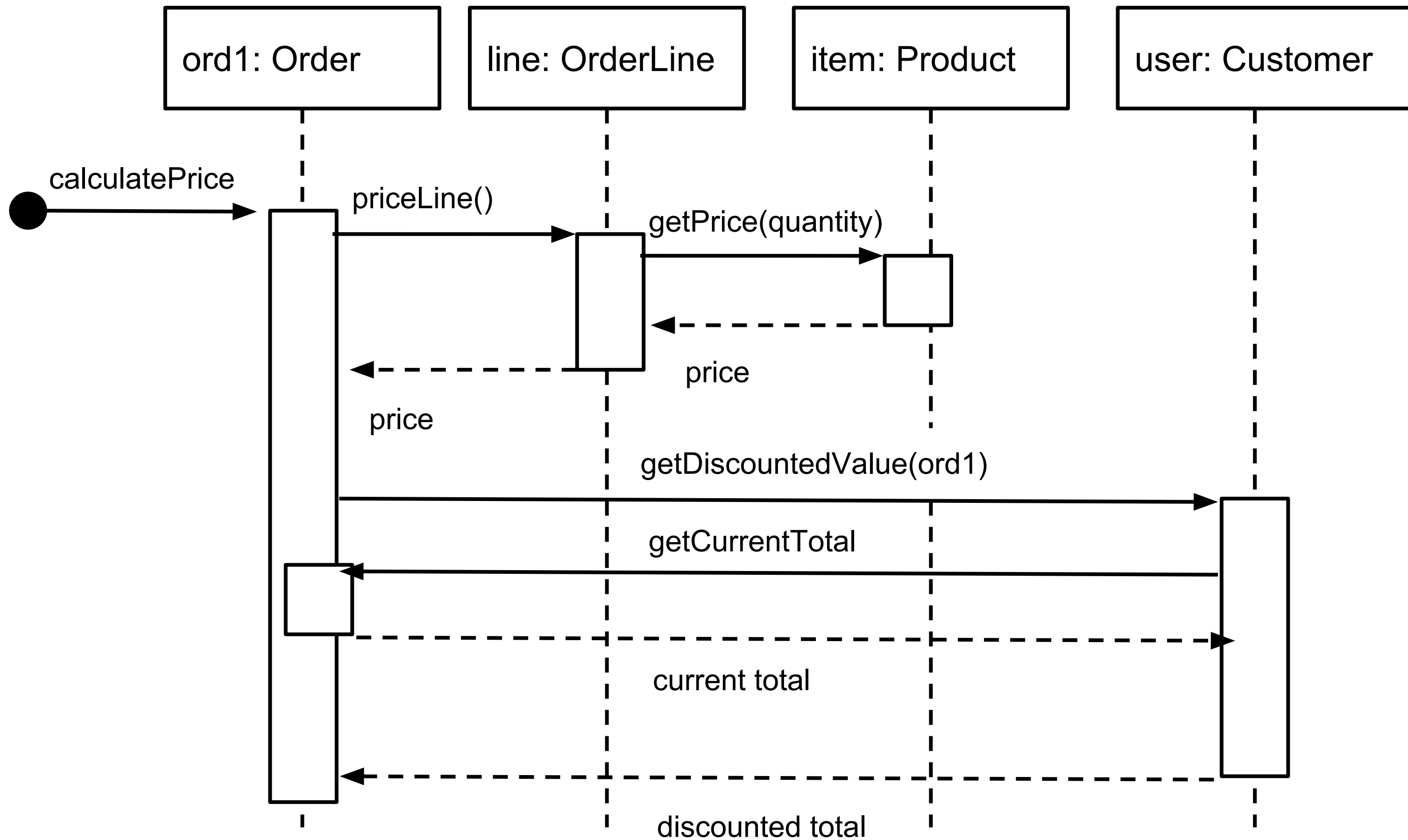
- Naming: *name : Class* or, informally, “A Class”.
- Lifeline: dashed line indicates life of the object.
- Found Message: Commands from an unmodeled source.
- Activation Box: A method is being executed.
- Message: One object calls a method offered by another object.
- Return: Information that the object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



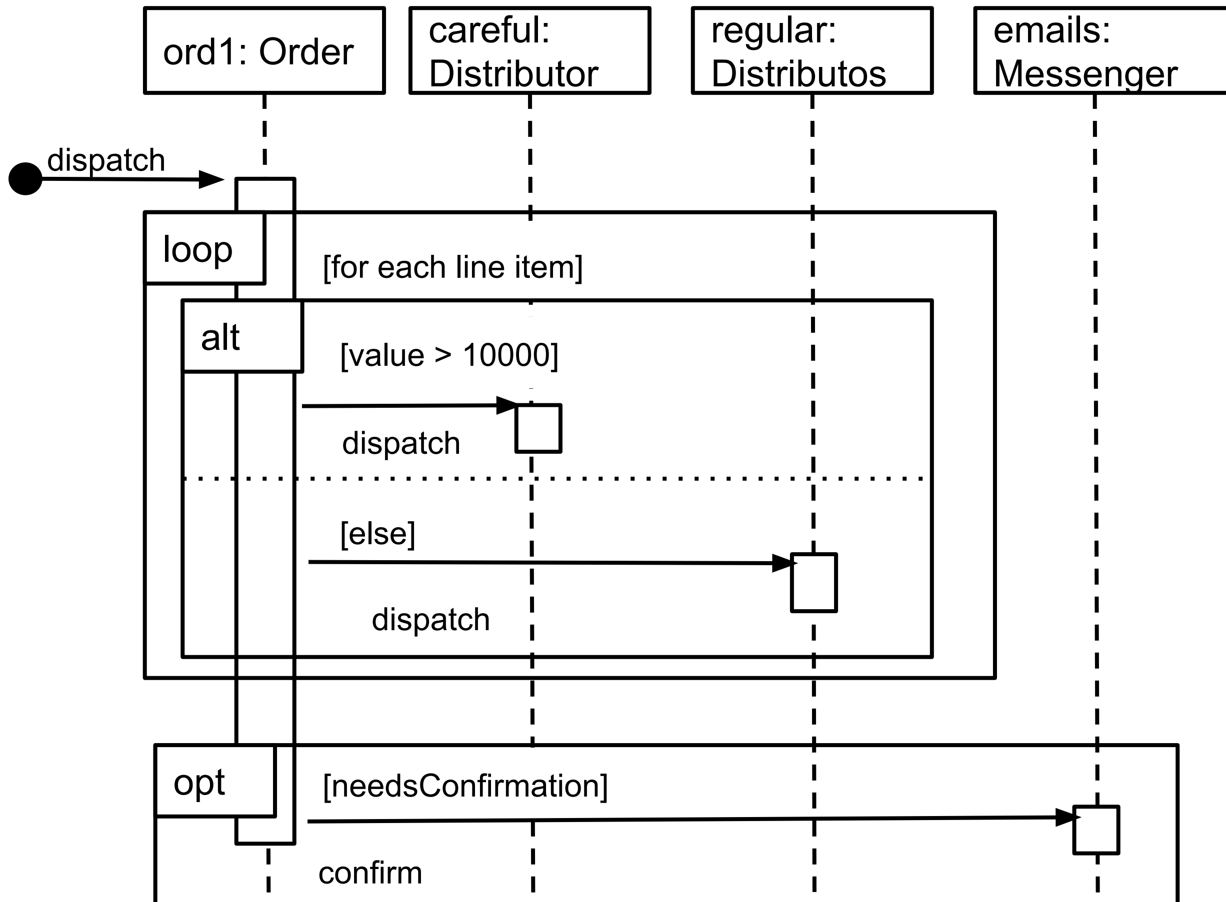
Conditional Behavior

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



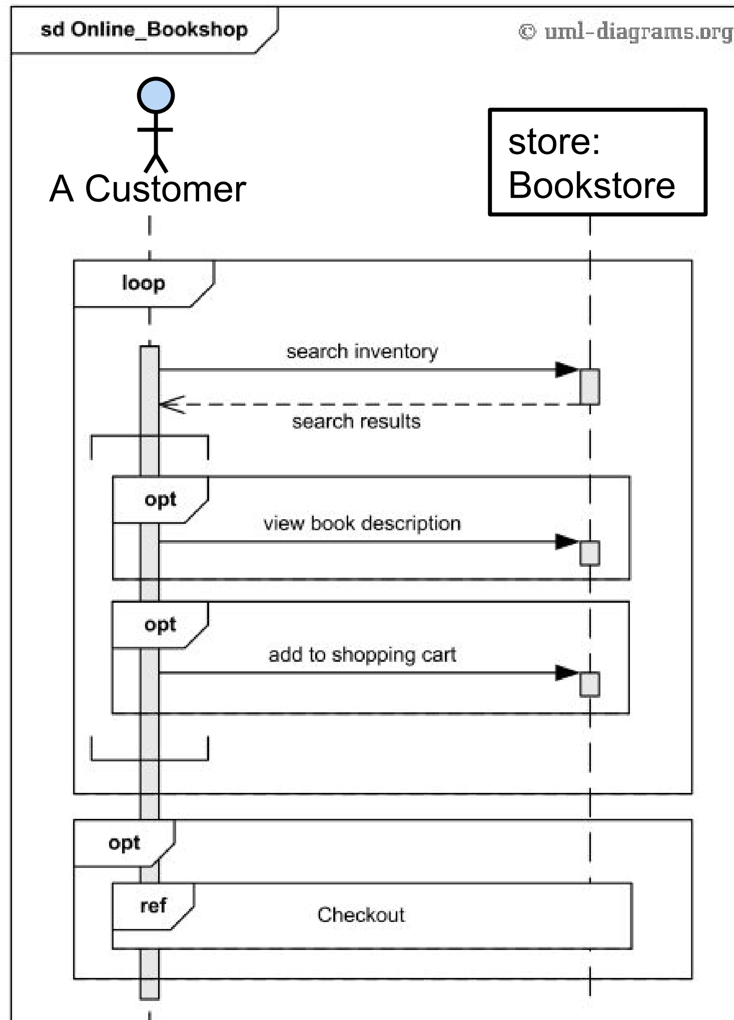

```
procedure dispatch
    foreach(line item)
        if (item.value > 10000)
            careful.dispatch
        else
            regular.dispatch

    if (needsConfirmation)
        emails.confirm
```

Frame Operators

- **alt:** Alternative paths, only one of which will execute.
- **opt:** Optional set of interactions.
- **loop:** Set of interactions may execute 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.

Online Bookstore Example



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

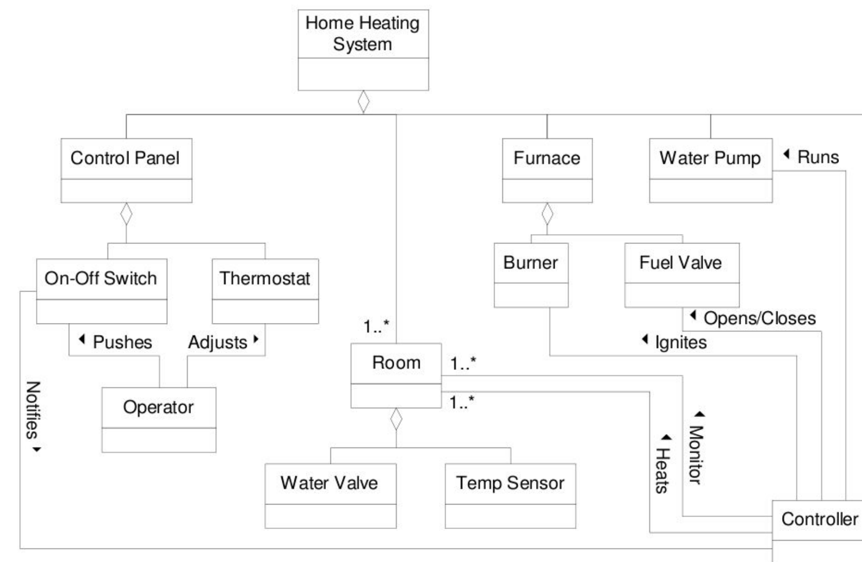
Use Case: Power Up

Actors: Home Owner

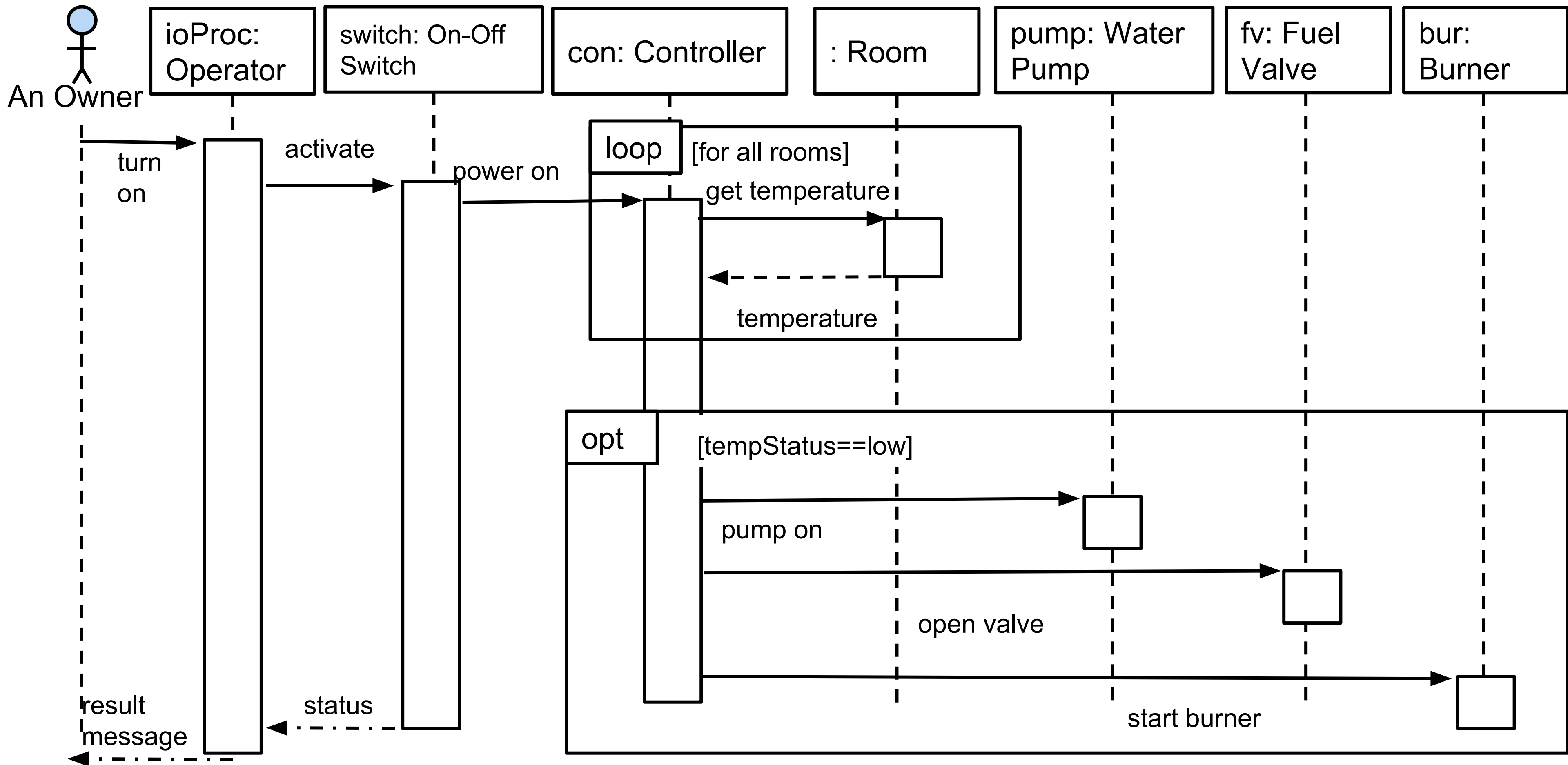
1. The Home Owner moves the power switch to the “on” position.
2. The system responds with a “system ready” status message if it starts successfully.

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

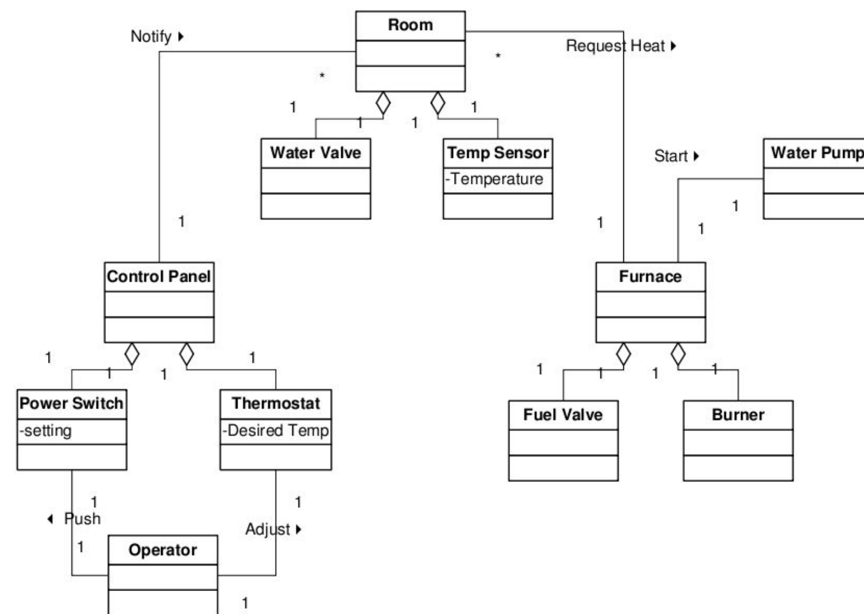
Use Case: Power Up

Actors: Home Owner

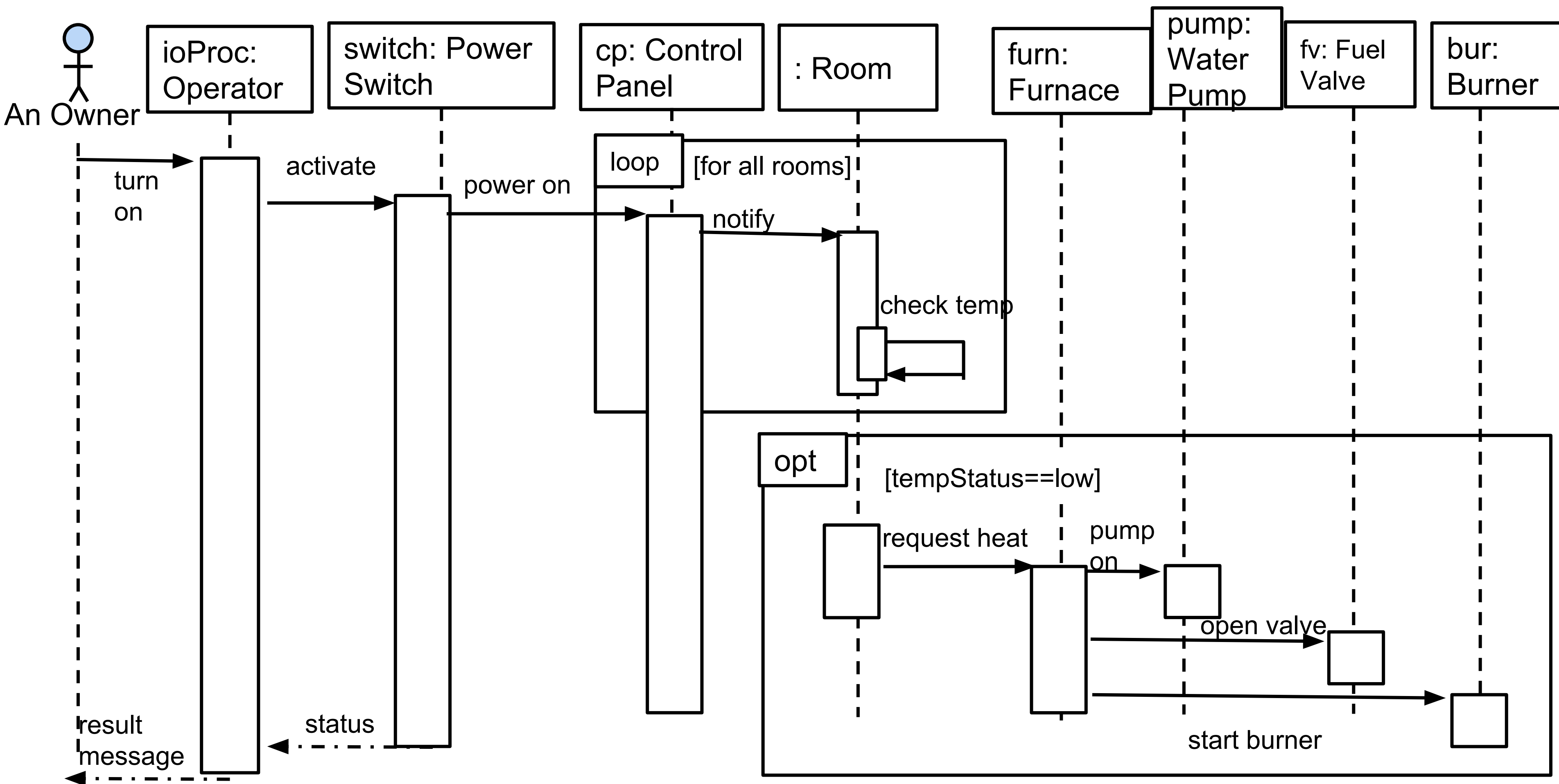
1. The Home Owner moves the power switch to the “on” position.
2. The system responds with a “system ready” status message if it starts successfully.

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

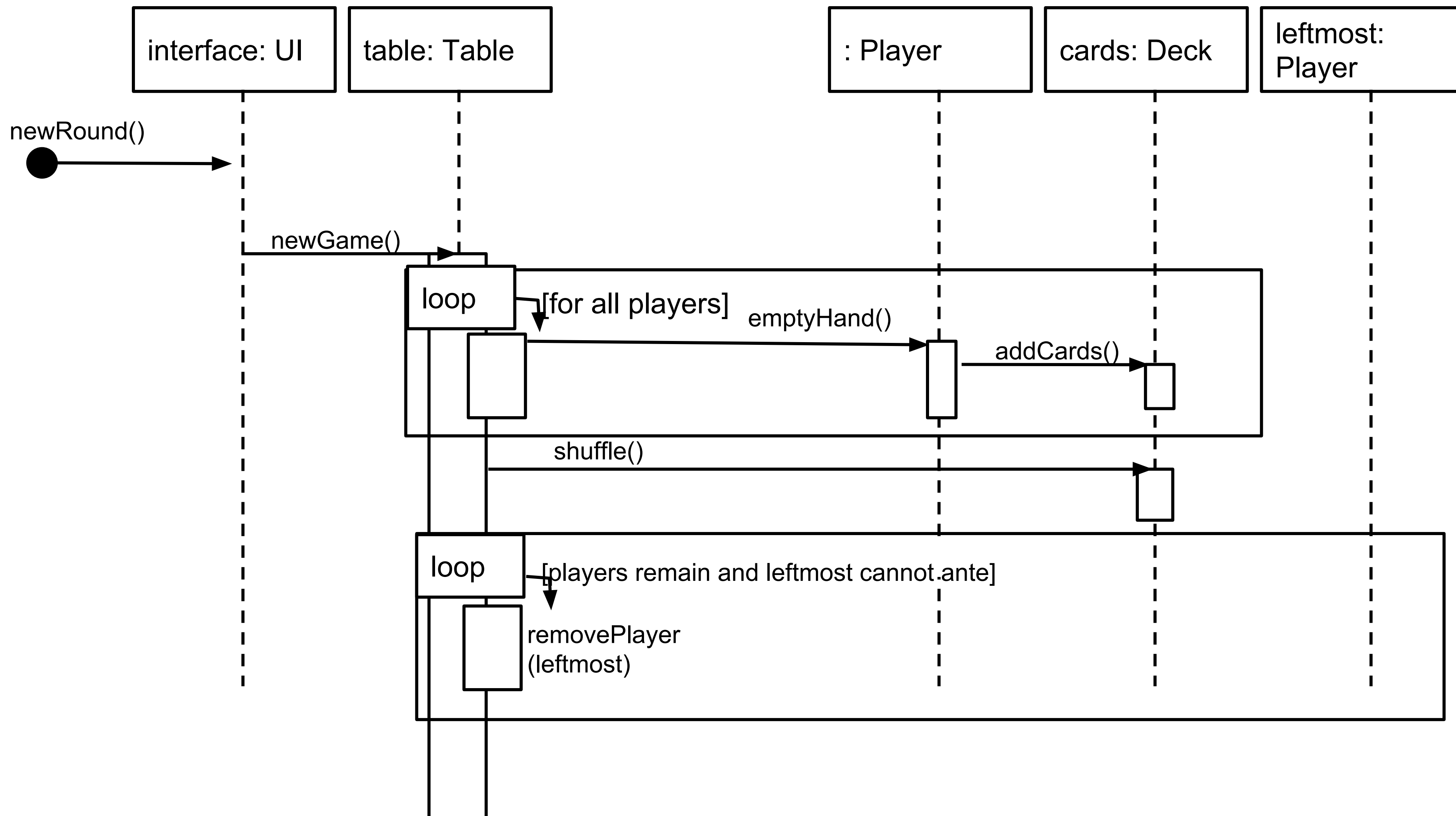


Example - Poker Hand

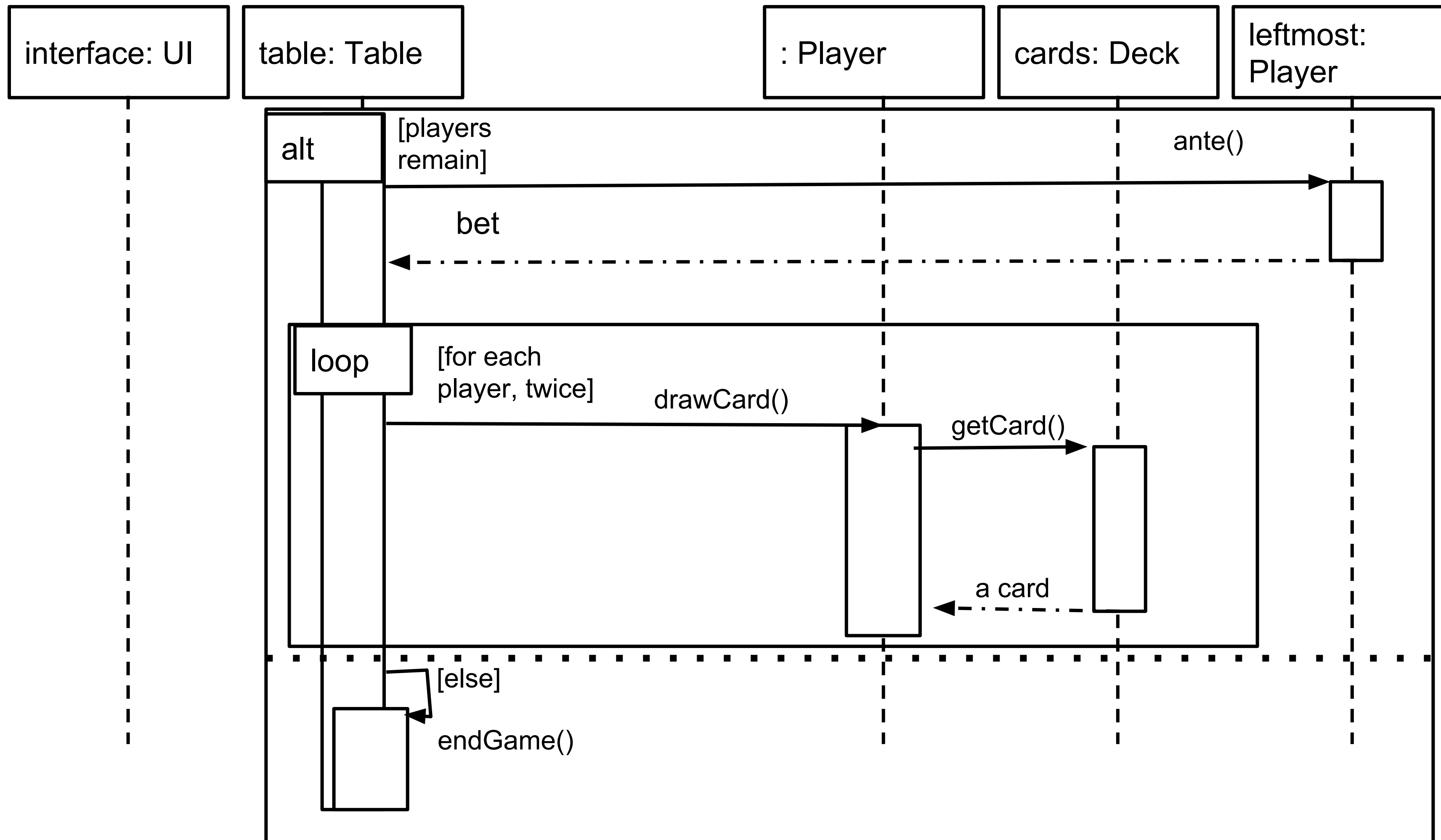
Starting a New Game Round

- 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.

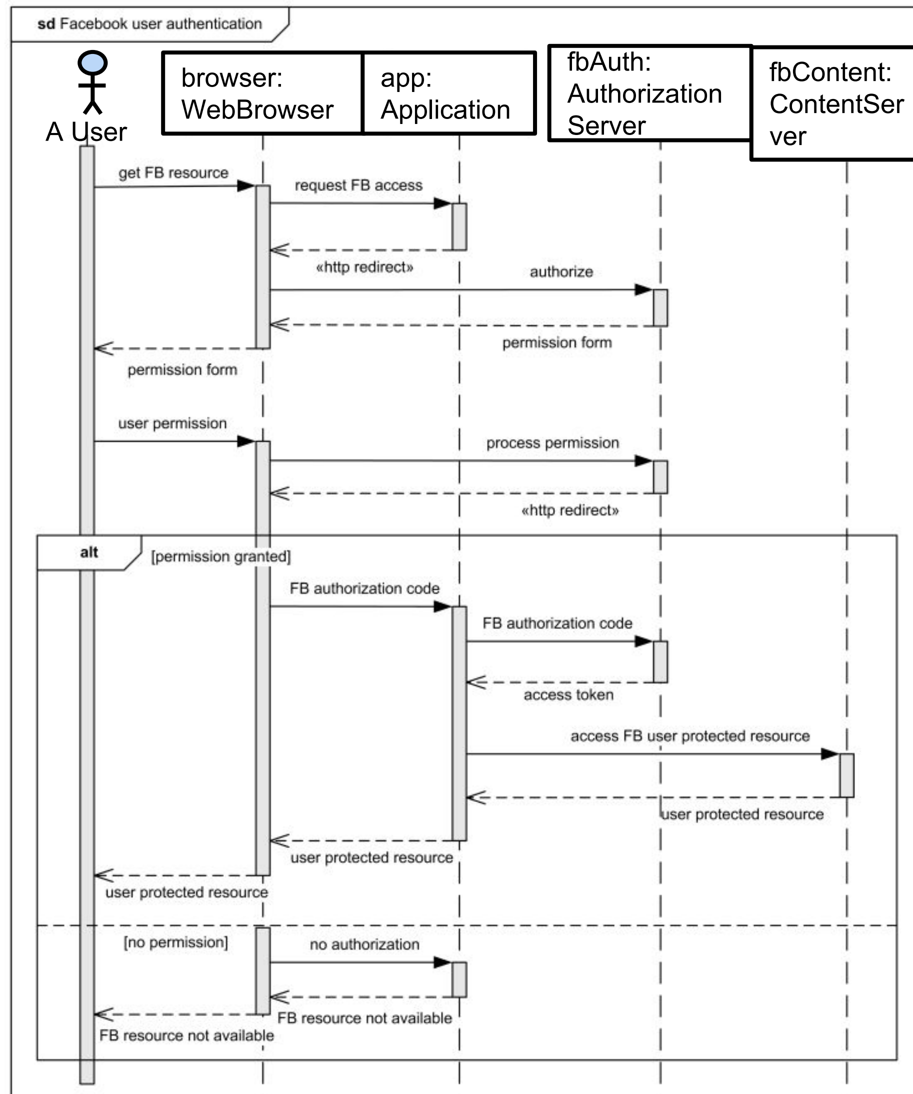
Example - Poker Hand



Example - Poker Hand



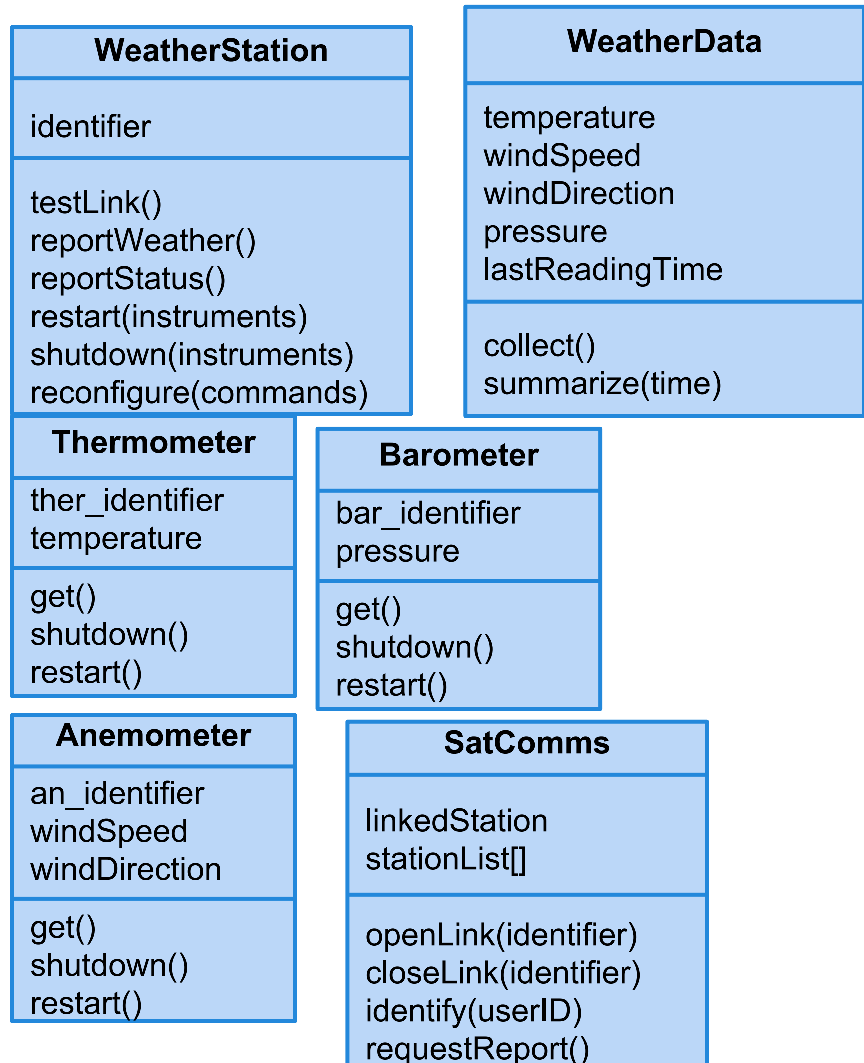
Facebook Web User Authentication



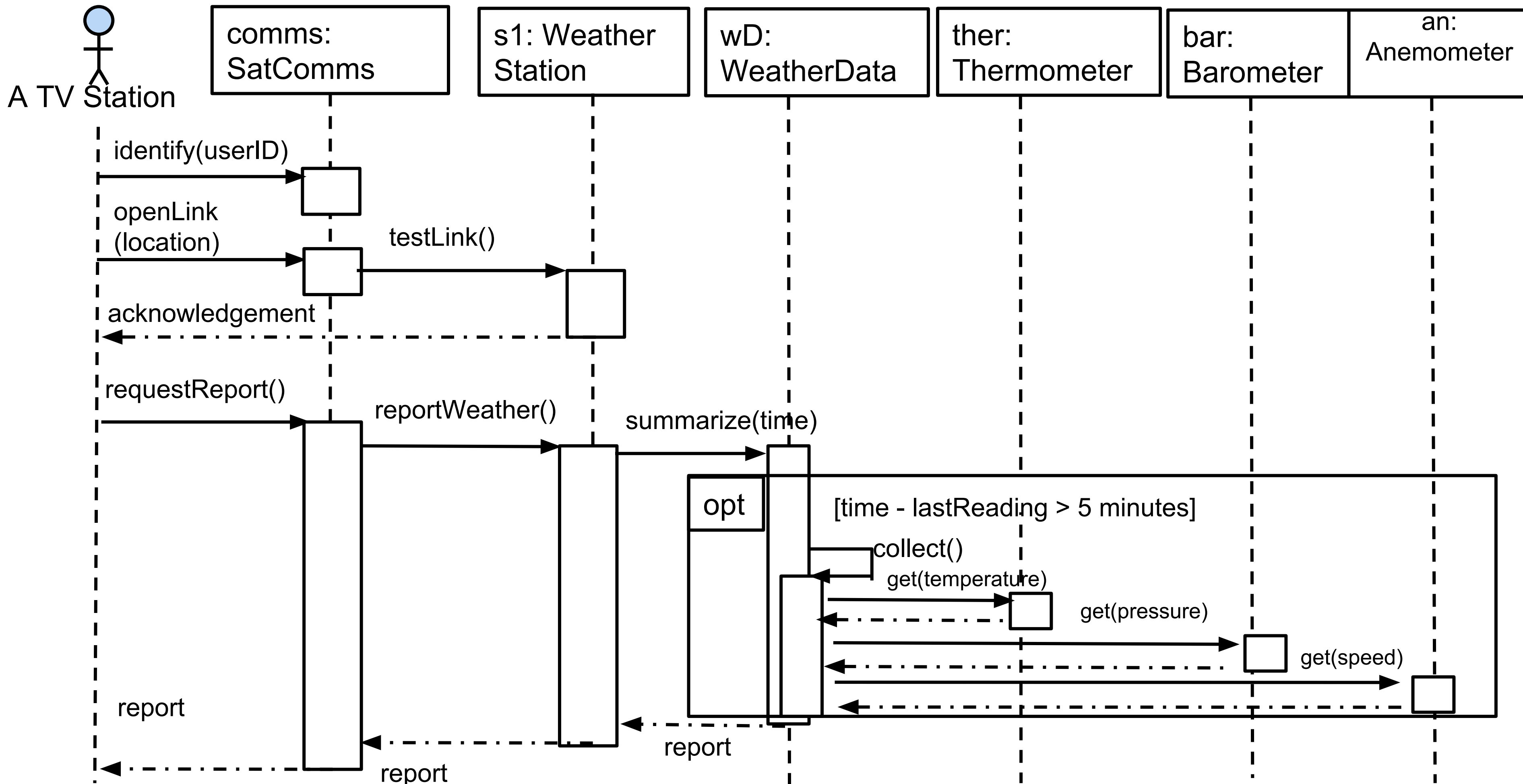
Sequence Diagram Activity

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



When to Use Sequence Diagrams

Use sequence diagrams when you want to look at the interaction between objects during a single use-case of the system.

- Diagrams show what information is passed between objects…
- and, more importantly, in what order.
- Allows for concurrency, process creation, and process destruction.
- Clarity is the goal - use comments.

Why Not Just Code It?

Sequence diagrams can be close to the code level, so why not take the class diagram and start coding it?

- A sequence of steps != code for those steps.
- Sequence diagrams are language-agnostic.
- Non-coders can still draw sequence diagrams.
- Easier to come up with sequence diagrams as a team.
- Can see many objects/classes at a time on same page.

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.

Next Time

- Testing Fundamentals.
- Reading:
  - Sommerville, ch. 8
- Homework due Friday.
  - Any questions?

backup slides

Modeling Dynamic Behavior

Several model formats used to examine dynamic behavior, at differing levels of detail:

- Use Case Diagrams
  - Externally visible behavior of system.
- Activity Diagrams
  - Flow of events during execution.
- Sequence Diagrams
  - Interactions between objects.
- State Diagrams
  - Detailed behavior of a single object.

Low Detail

High Detail

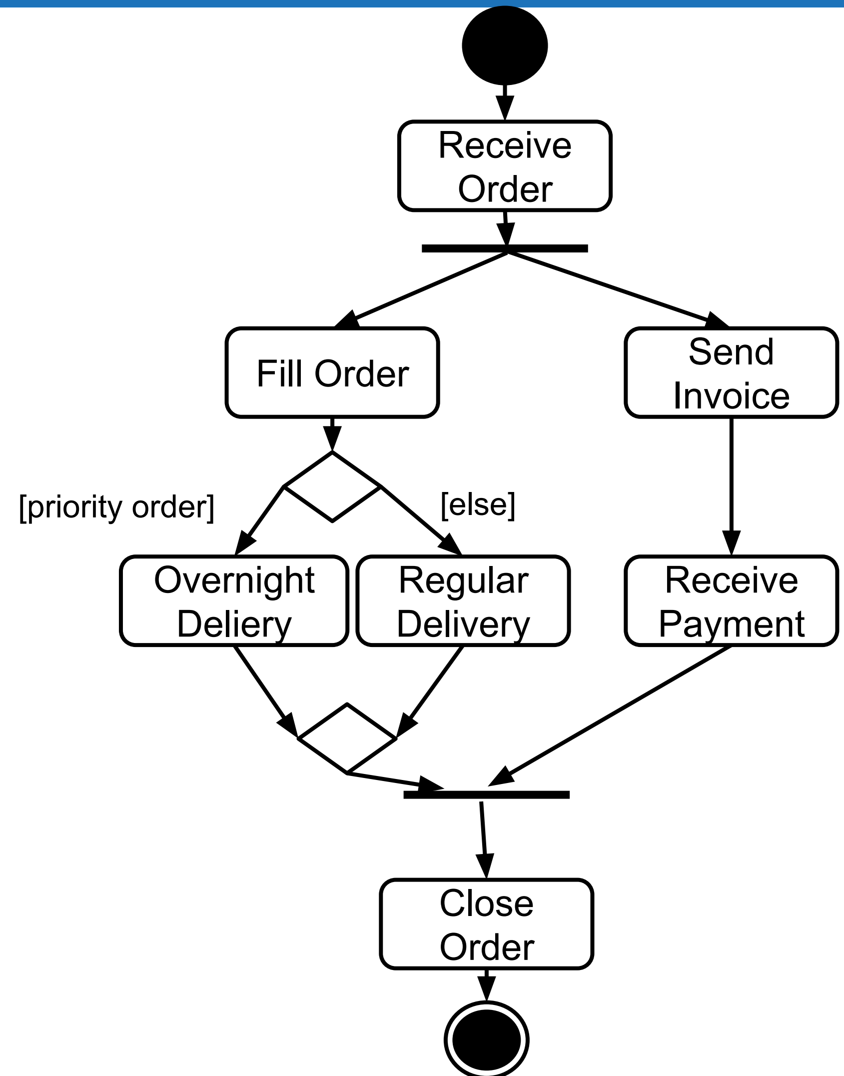
Activity Diagrams

Show how system and user actions are connected together:

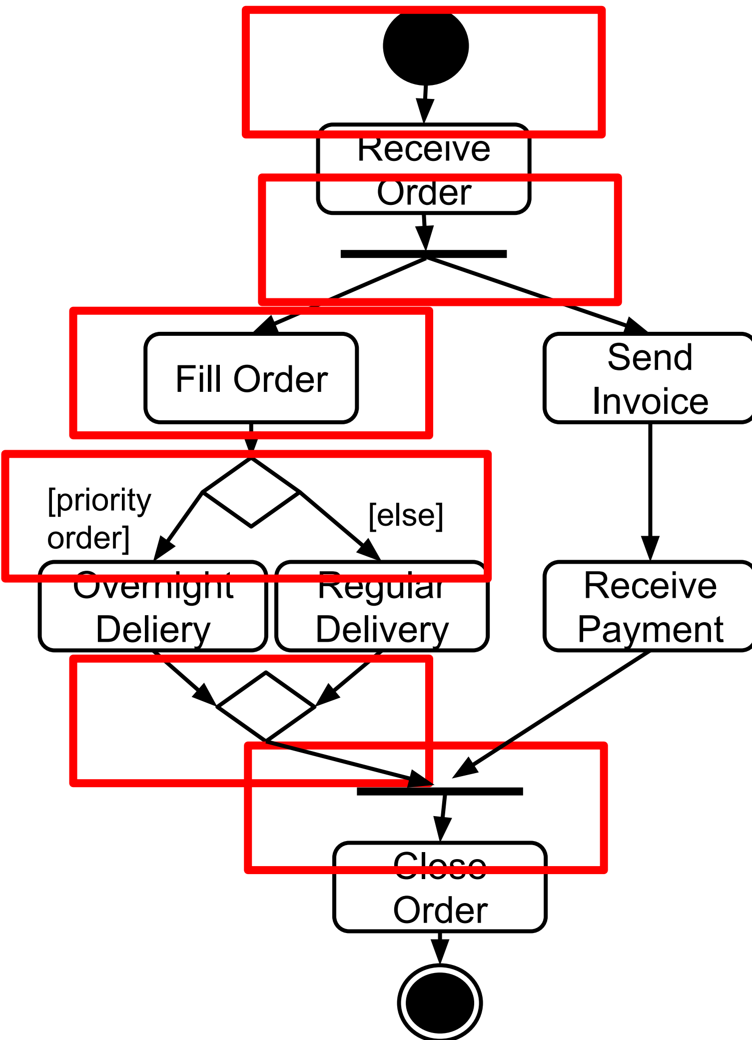
- Shows order of processing.
- Captures parallelism.

Allows analysis of:

- Processing
- Synchronization
- Conditional selection of activities

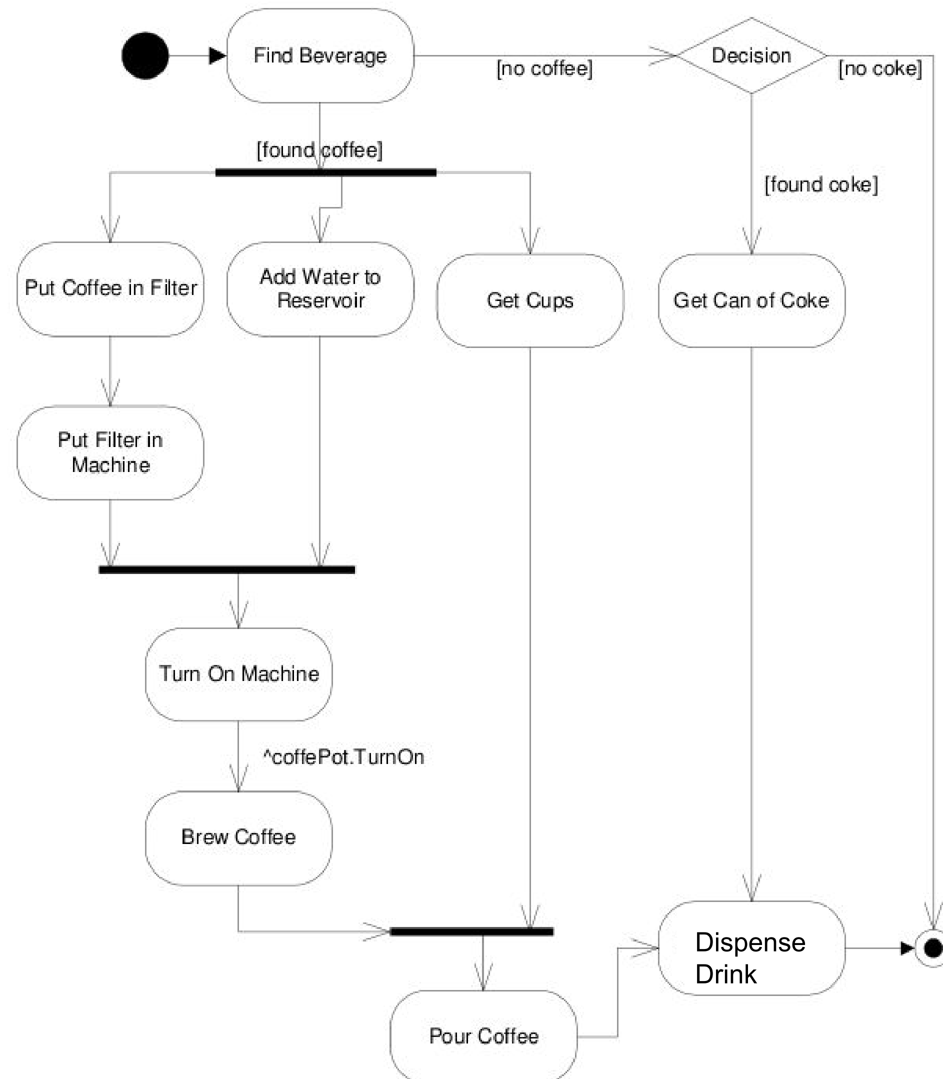


Activity Diagram Syntax



- Initial Node: Where execution begins.
- Action: Something that the system does.
- Fork: Split into concurrent activities.
- Join: Combine event flow back into one stream.
- Decision: Perform different activities based on result.
- Merge: Resume execution after a decision split.

Drink Dispenser Example

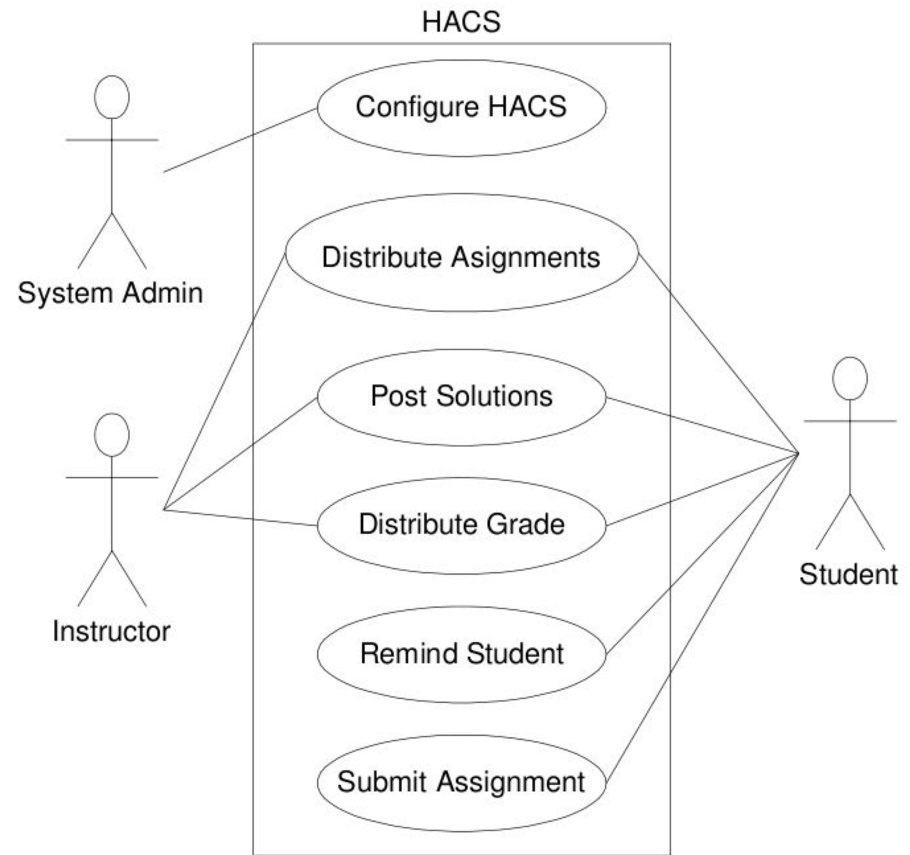


HACS Example

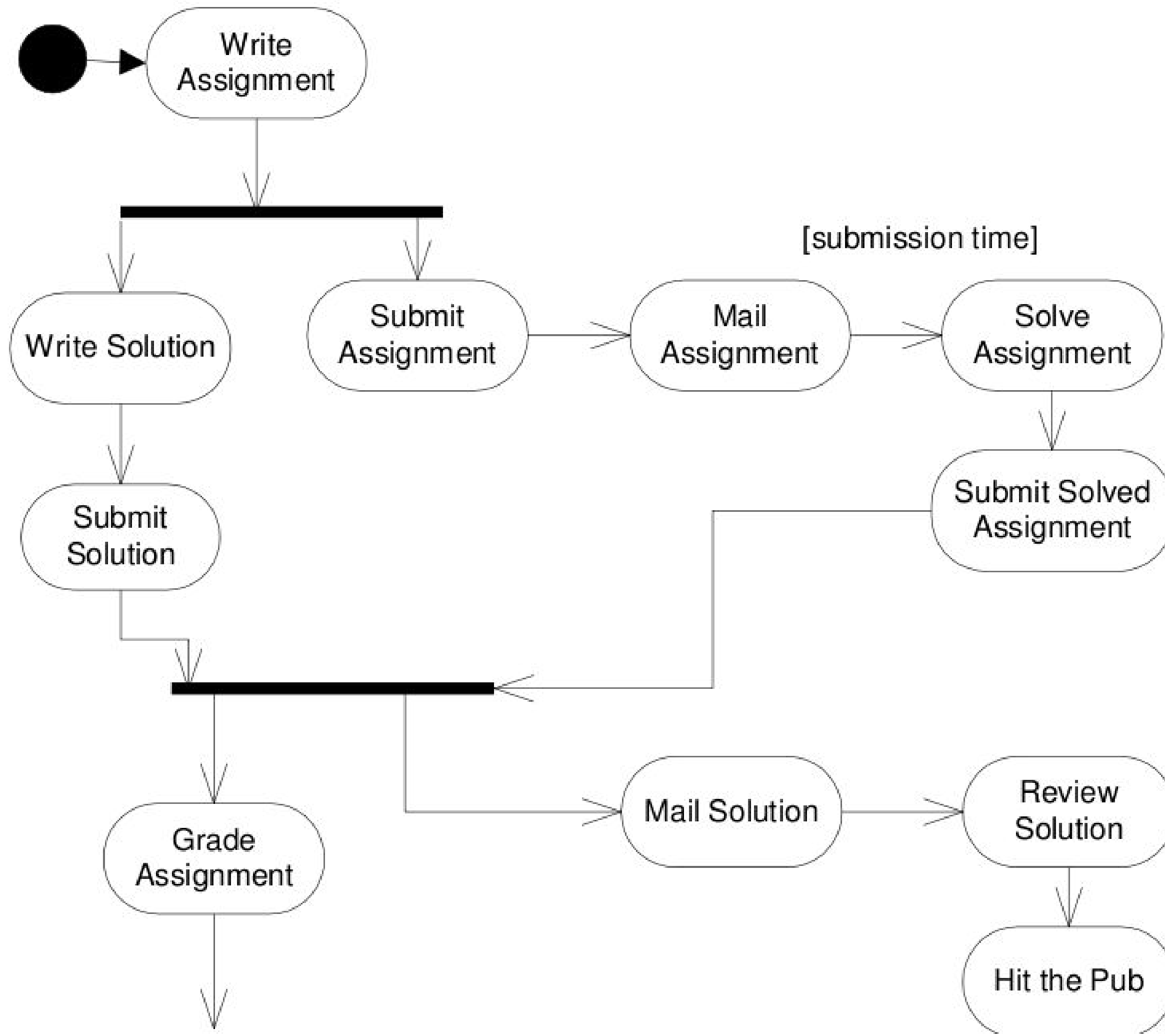
Use Case: Distribute Assignments

Actors: Instructor, Student

Description: The Instructor completes an assignment and submits it to the system. The Instructor will also submit the delivery date, due date, and the class the assignment is assigned for. The system will, at the due date, mail the assignment to the student.

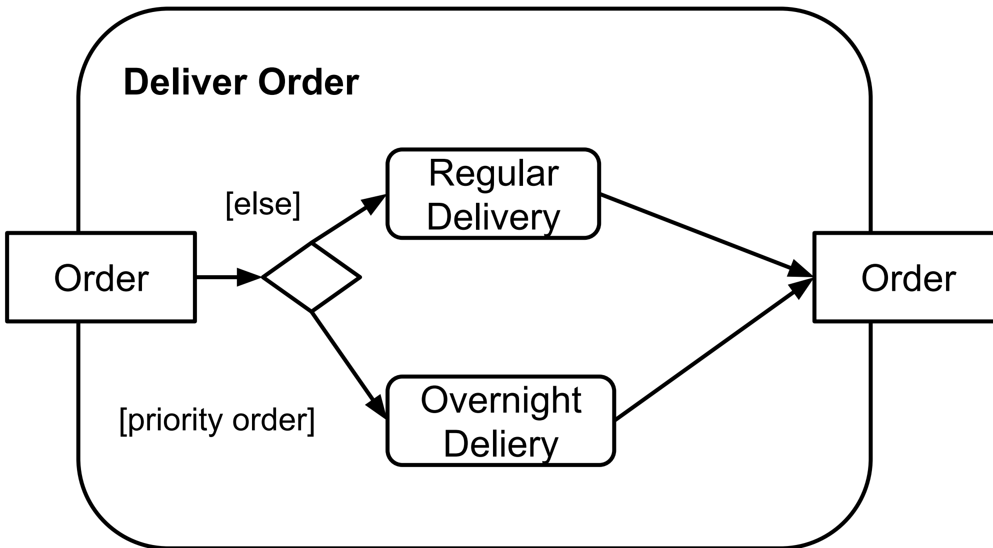


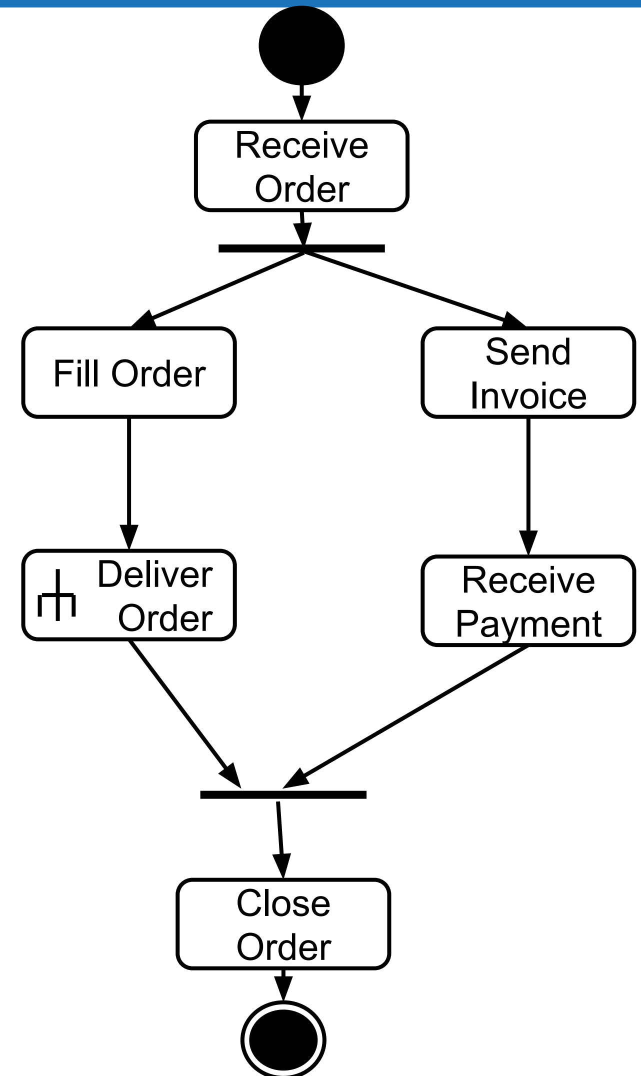
HACS Example



Subactivities

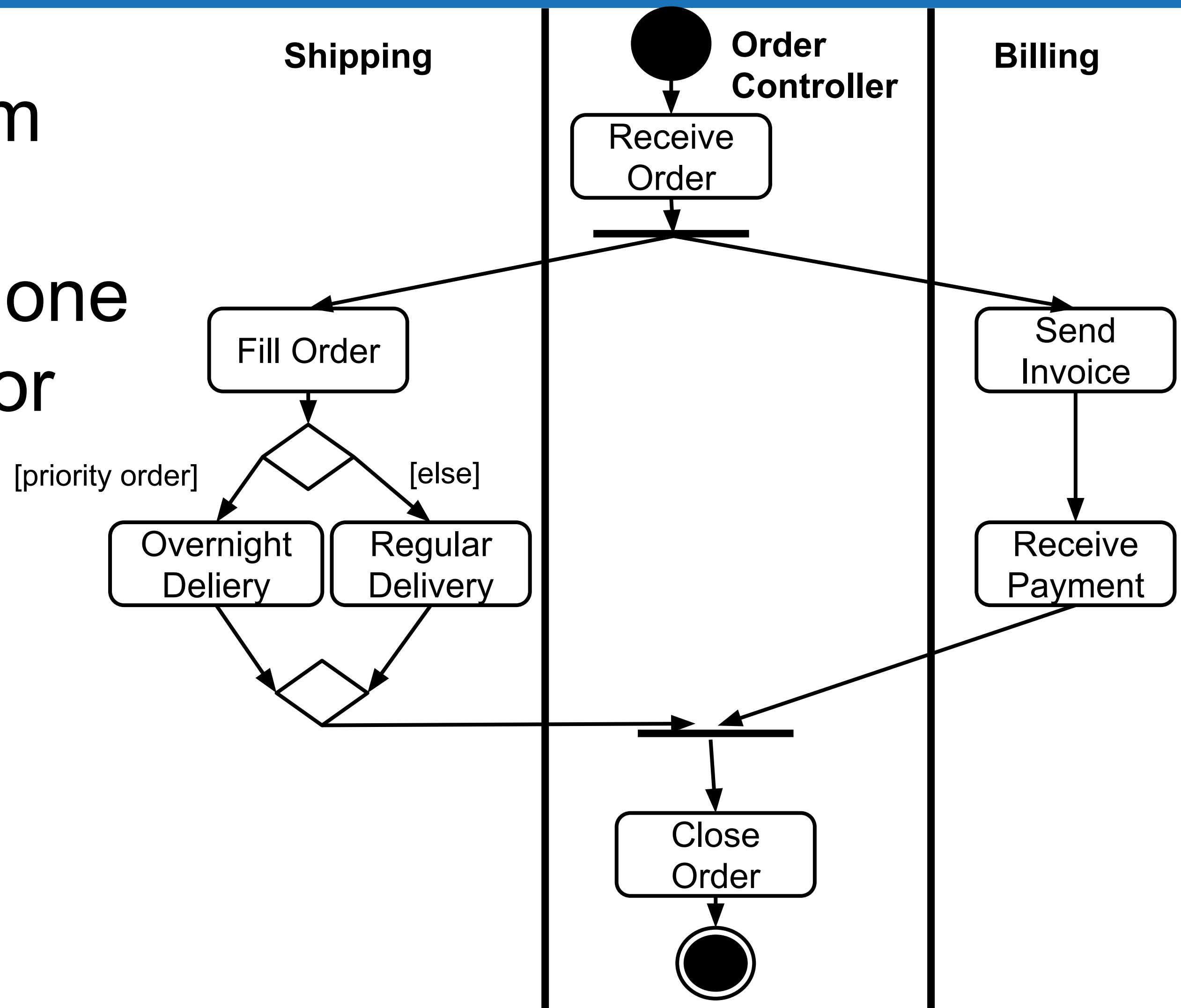
Describe complex actions using subactivity diagrams.



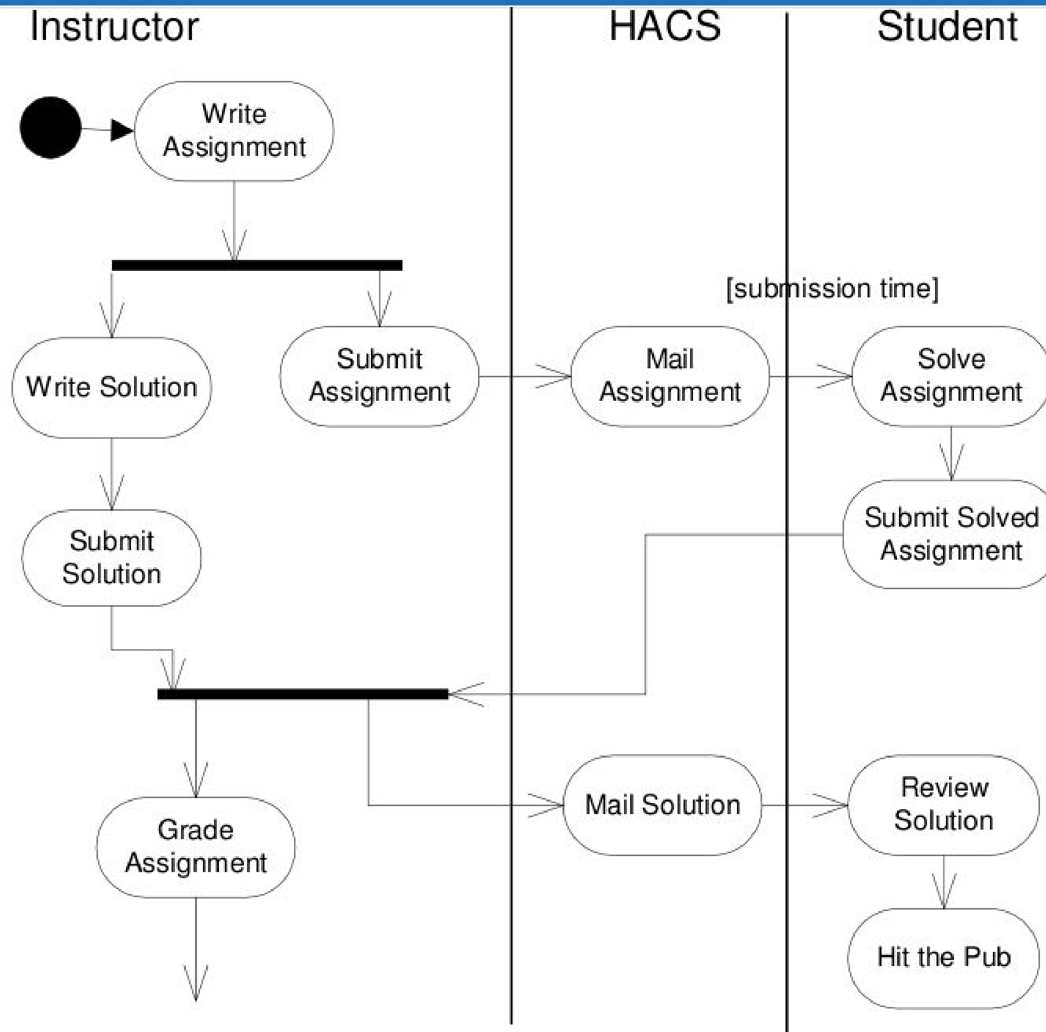


Partitions

Divide diagram into actions performed by one entity (object or subsystem) or actor.



HACS Partitions Example



Problems with Activity Diagrams

- They are glorified flowcharts.
  - Very easy to make a traditional data-flow oriented design, but not well-suited to OO design.
- Hard to isolate behavior of a single object.
  - Do not show how objects collaborate to perform an action.
  - Do not show how individual objects behave over time.
- However...
  - They can be very powerful when you know how to use them correctly.

When to use Activity Diagrams

Useful when:

- Analyzing a use case (or collection of use cases).
    - Great stepping stone from use cases to dynamic design models.
- Understanding flow of information or control through a system.
- Working with parallel applications.

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

Should NOT always be avoided, but must be used 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.

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 code.

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 a specific hardware architecture, requires certain CPU/RAM/GPU, or requires special software, then make sure those 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.