

---

# Chapter 5 - System Modeling

**MI Qing (Lecturer)**

Telephone: 15210503242

Email: [miqing@bjut.edu.cn](mailto:miqing@bjut.edu.cn)

Office: 410, Information Building

# Topics Covered

---

- ✧ Activity diagram
- ✧ Sequence diagram
- ✧ Class diagram
- ✧ State machine diagram
- ✧ Model-driven engineering

# System Modeling

---

- ✧ System modeling is the process of **developing abstract models of a system, with each model presenting a different view or perspective of that system.**
- ✧ System modeling has now come to mean representing a system using some kind of graphical notation, which is now almost always based on notations in the **Unified Modeling Language (UML).**

# Existing and Planned System Models

---

- ✧ Models of the existing system are used during requirements engineering to **help clarify what the existing system does** and can be used as a basis for discussing its strengths and weaknesses. These then lead to requirements for the new system.
- ✧ Models of the new system are used during requirements engineering to **help explain the proposed requirements to other system stakeholders**. Engineers use these models to **discuss design proposals and to document the system for implementation**.

# System Perspectives

---

- ✧ **An external perspective**: where you model the context or environment of the system.
- ✧ **An interaction perspective**: where you model the interactions between a system and its environment, or between the components of a system.
- ✧ **A structural perspective**: where you model the organization of a system or the structure of the data that is processed by the system.
- ✧ **A behavioral perspective**: where you model the dynamic behavior of the system and how it responds to events.

# Use of Graphical Models

---

- ✧ As a means of **facilitating discussion** about an existing or proposed system.
  - Incomplete and incorrect models are OK as their role is to support discussion.
- ✧ As a way of **documenting an existing system**.
  - Models should be an accurate representation of the system but need not be complete.
- ✧ As a detailed system description that can be **used to generate a system implementation**.
  - Models have to be both correct and complete.

# What is UML

---

- ✧ UML is a standardized modeling language consisting of an integrated set of diagrams, developed to help system and software developers for **specifying, visualizing, constructing, and documenting the artifacts of software systems**.
- Java and C++ are programming languages to implement a system.
  - UML is a modeling language to describe a system or its design.



Booch

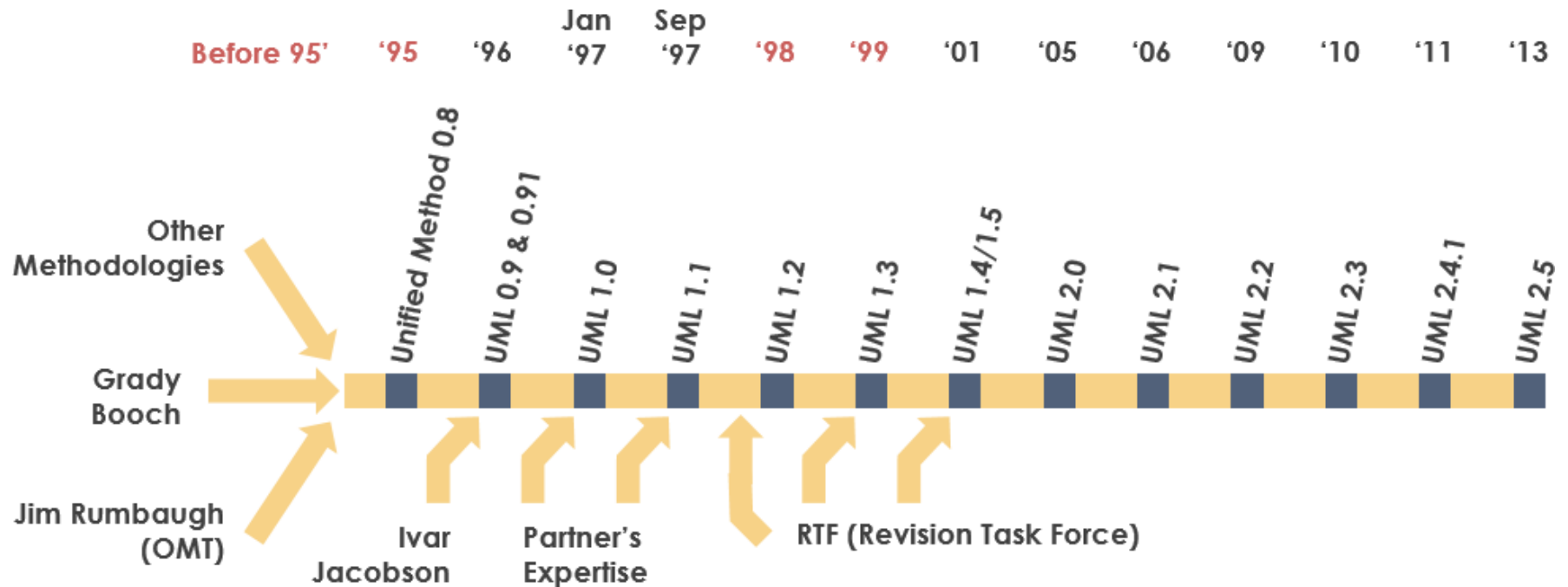


Rumbaugh



Jacobson

# History of UML

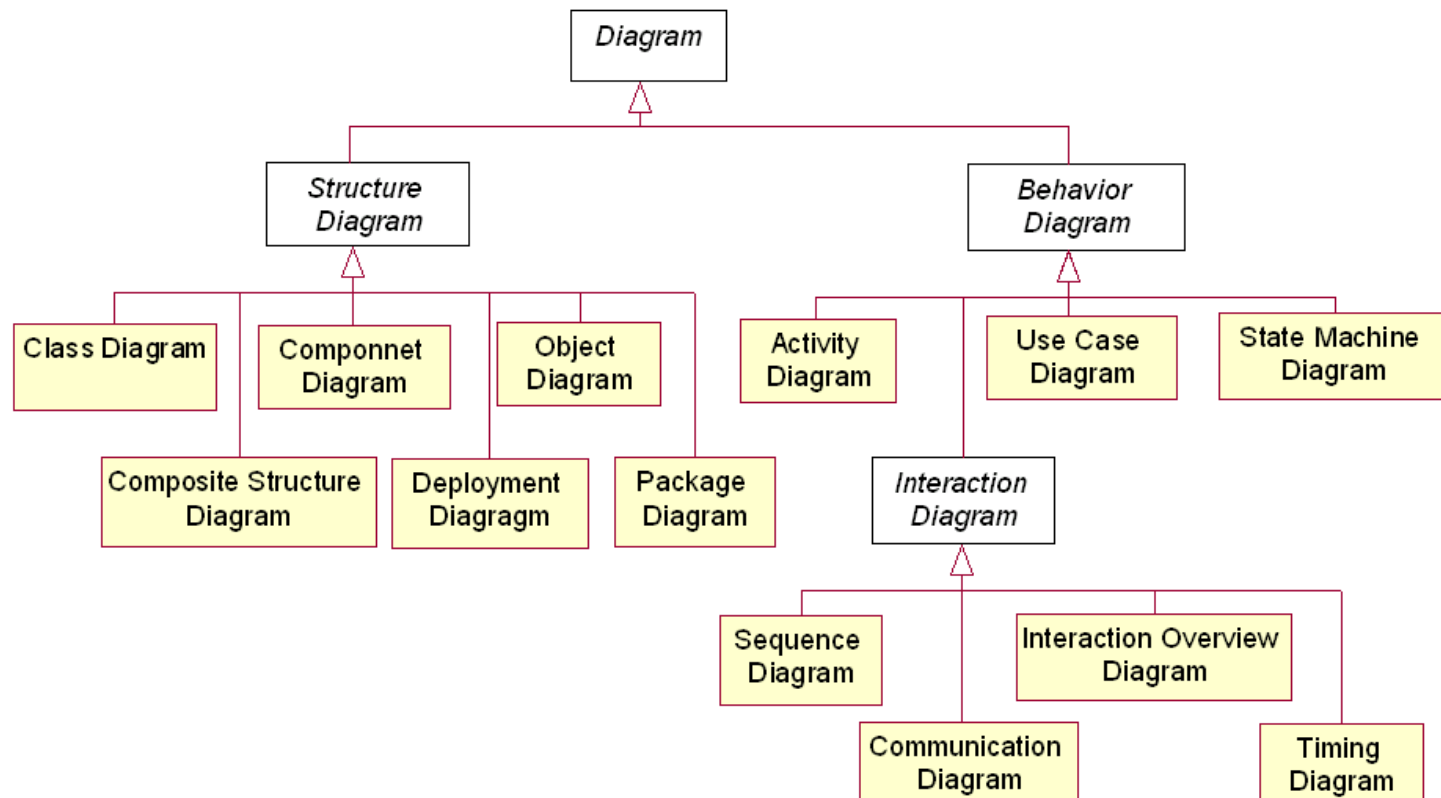


Before '95' - Fragmentation ► '95' - Unification ► '98' - Standardization ► '99' - Industrialization



# Diagrams in UML 2.0

- ✧ UML offers a variety of diagrammatic notations for modeling static and dynamic aspects of an application.



# A Survey by Erickson and Siau

---

- ✧ Most users of the UML thought that five diagram types could represent the essentials of a system.
  - **Activity diagrams**: which show the activities involved in a process or in data processing.
  - **Use case diagrams**: which show the interactions between a system and its environment.
  - **Sequence diagrams**: which show interactions between actors and the system and between system components.
  - **Class diagrams**: which show the object classes in the system and the associations between these classes.
  - **State machine diagrams**: which show how the system reacts to internal and external events.

# UML Modeling and Diagramming tools

---

- ✧ Microsoft Visio
- ✧ Enterprise Architect
- ✧ StarUML
  - <https://staruml.io/>
- ✧ UMLet
  - <http://www.umlet.com/>
- ✧ Draw.io
  - <https://drawio-app.com/uml-diagrams/>

---

# Activity Diagram

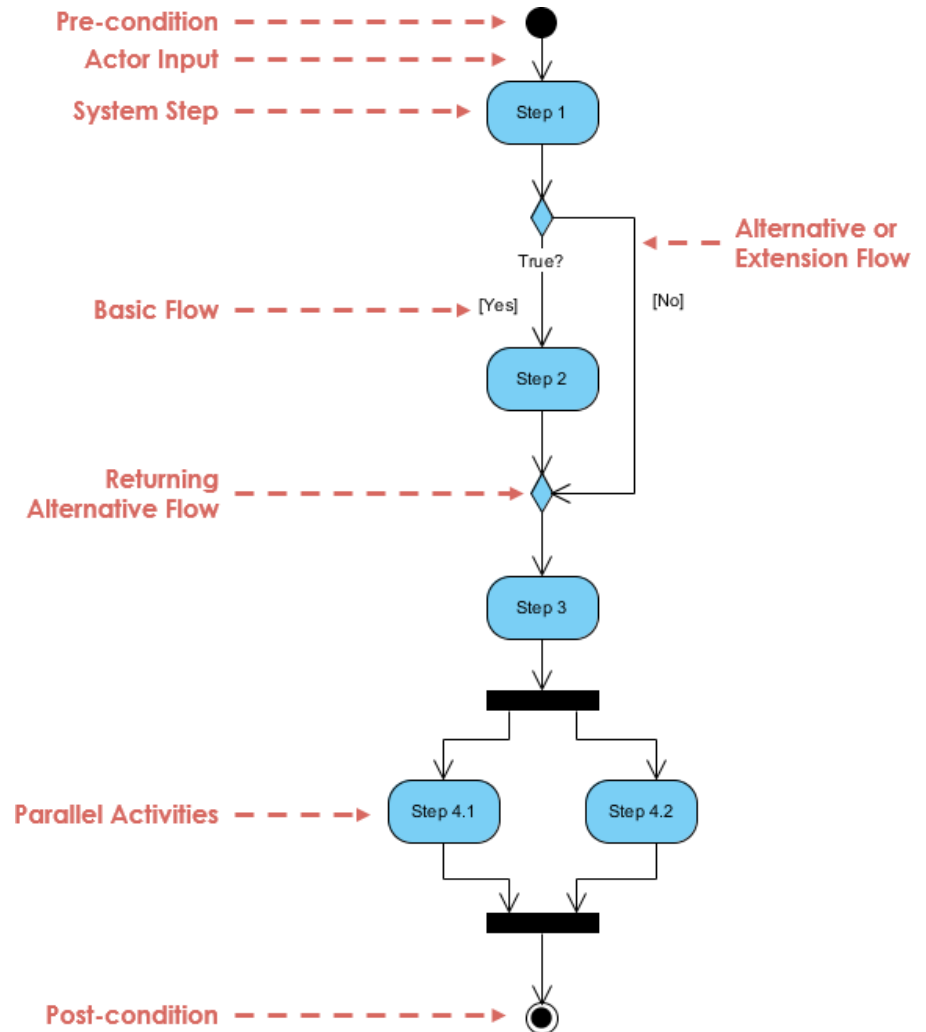
# Activity Diagram

---

- ✧ Activity diagram is an important behavioral diagram in UML diagram to **describe dynamic aspects of the system**.
- ✧ Activity diagram is essentially **an advanced version of flow chart** that modeling the flow from one activity to another activity.
- ✧ Can be used:
  - To model business workflows.
  - To describe a system function that is represented within a use case or between use cases.
  - In operation specifications, to describe the logic of an operation.

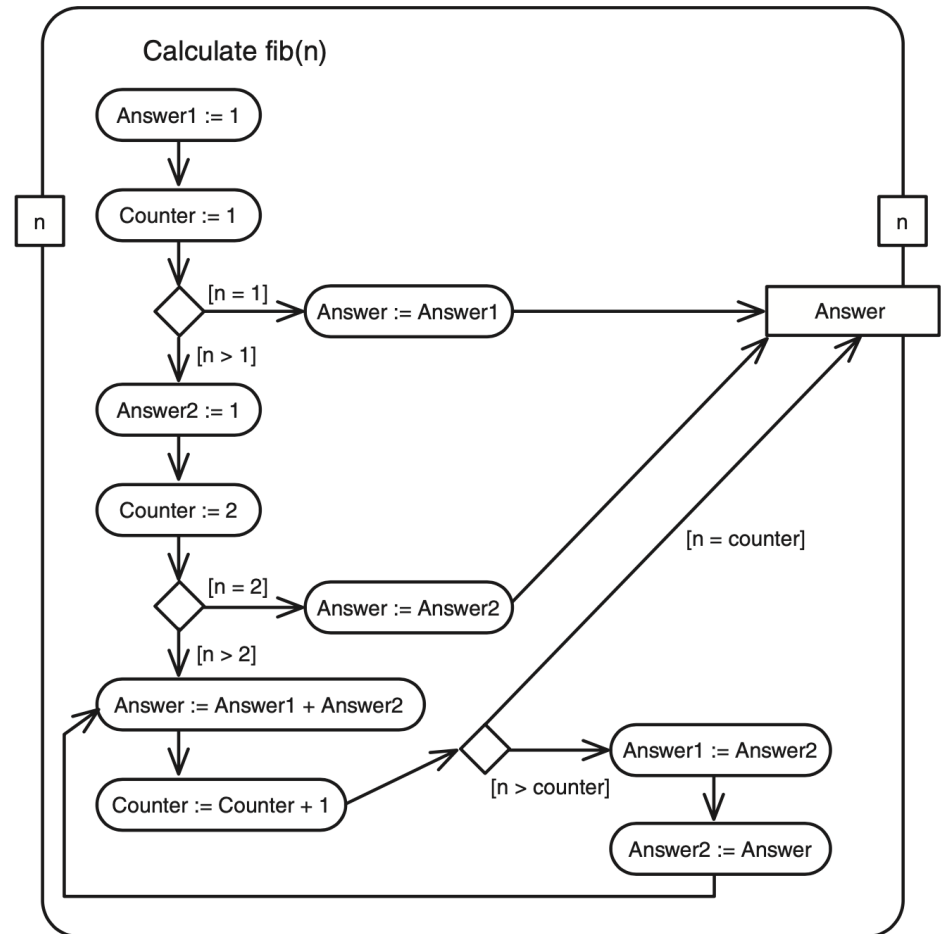
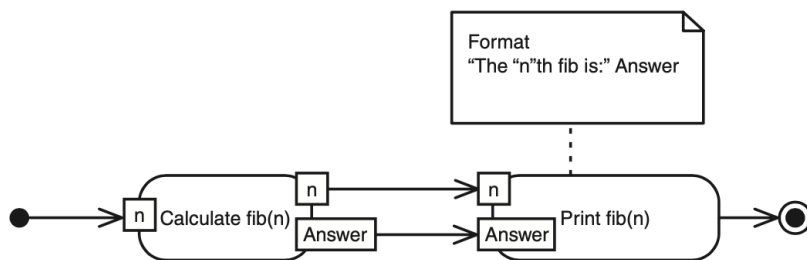
# Key Parts in Activity Diagram

- ✧ Activity and action
- ✧ Control flow
- ✧ Decision node
- ✧ Fork and Join
- ✧ Swimlane



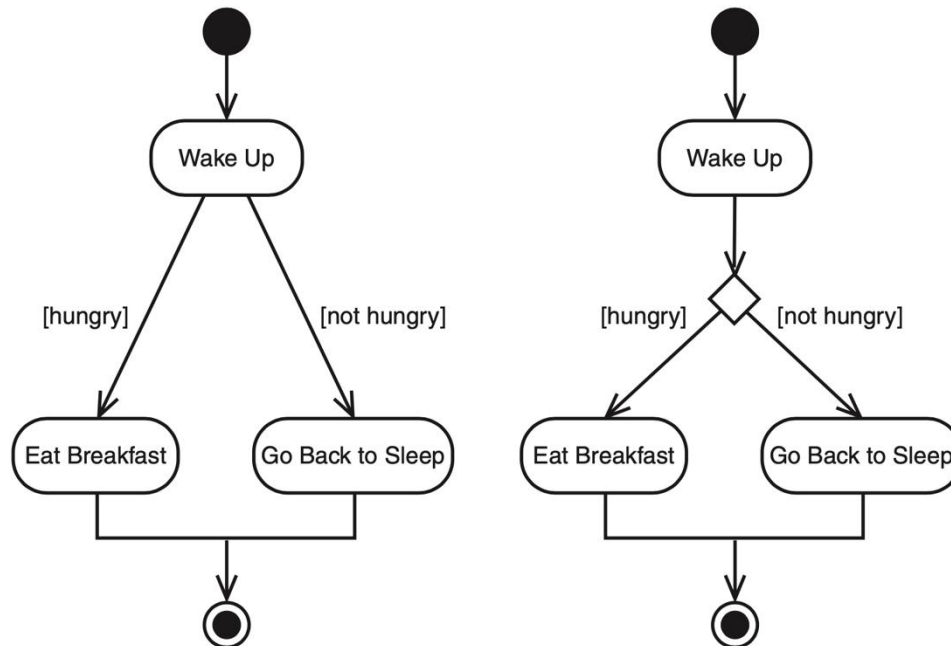
# Activity and Action

- ✧ An activity can consist of a number of actions. The icon for an action is the same as the icon for an activity.
- ✧ Activities consists of action nodes, object nodes, and control nodes linked by activity edges.



# Decision Node

- ✧ Decision node represents a test condition to ensure that the control flow or object flow only goes down one path.
  - A single trigger leads to more than one possible outcome, each with its own **guard condition**.





# Exercise: creating a document

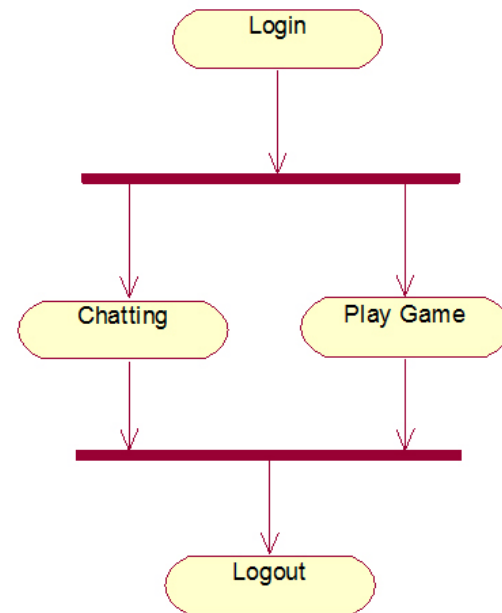
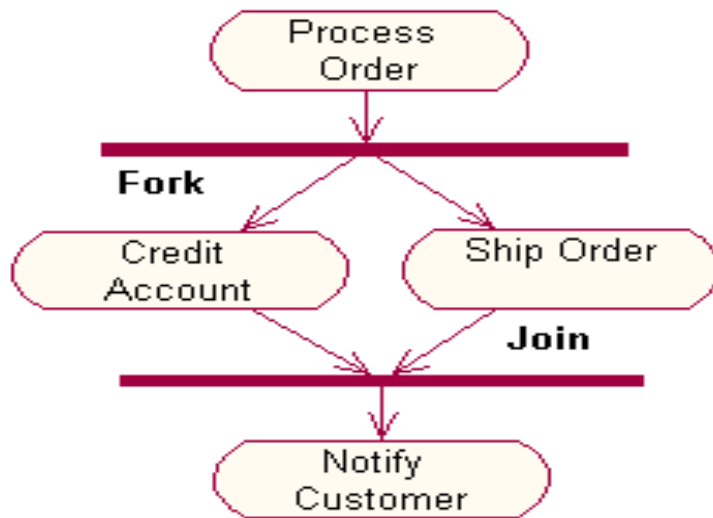
---

1. Open the word processing package.
2. Create a file.
3. Save the file under a unique name within its directory.
4. Type the document.
5. If graphics are necessary, open the graphics package, create the graphics, and paste the graphics into the document.
6. If a spreadsheet is necessary, open the spreadsheet package, create the spreadsheet, and paste the spreadsheet into the document.
7. Save the file.
8. Print a hard copy of the document.
9. Exit the word processing package.

# Fork and Join

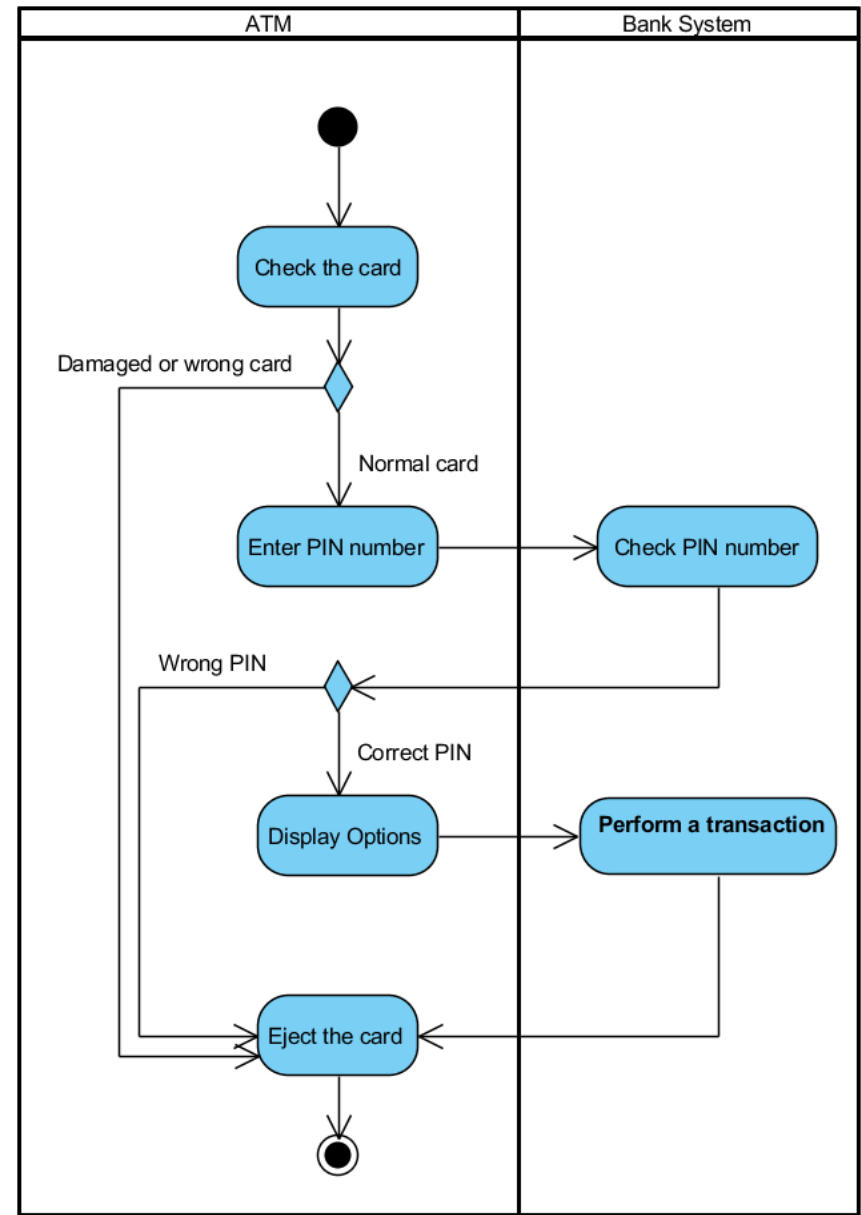
---

- ✧ Fork: Split behavior into a set of parallel or concurrent flows of activities (or actions).
- ✧ Join: Bring back together a set of parallel or concurrent flows of activities (or actions).

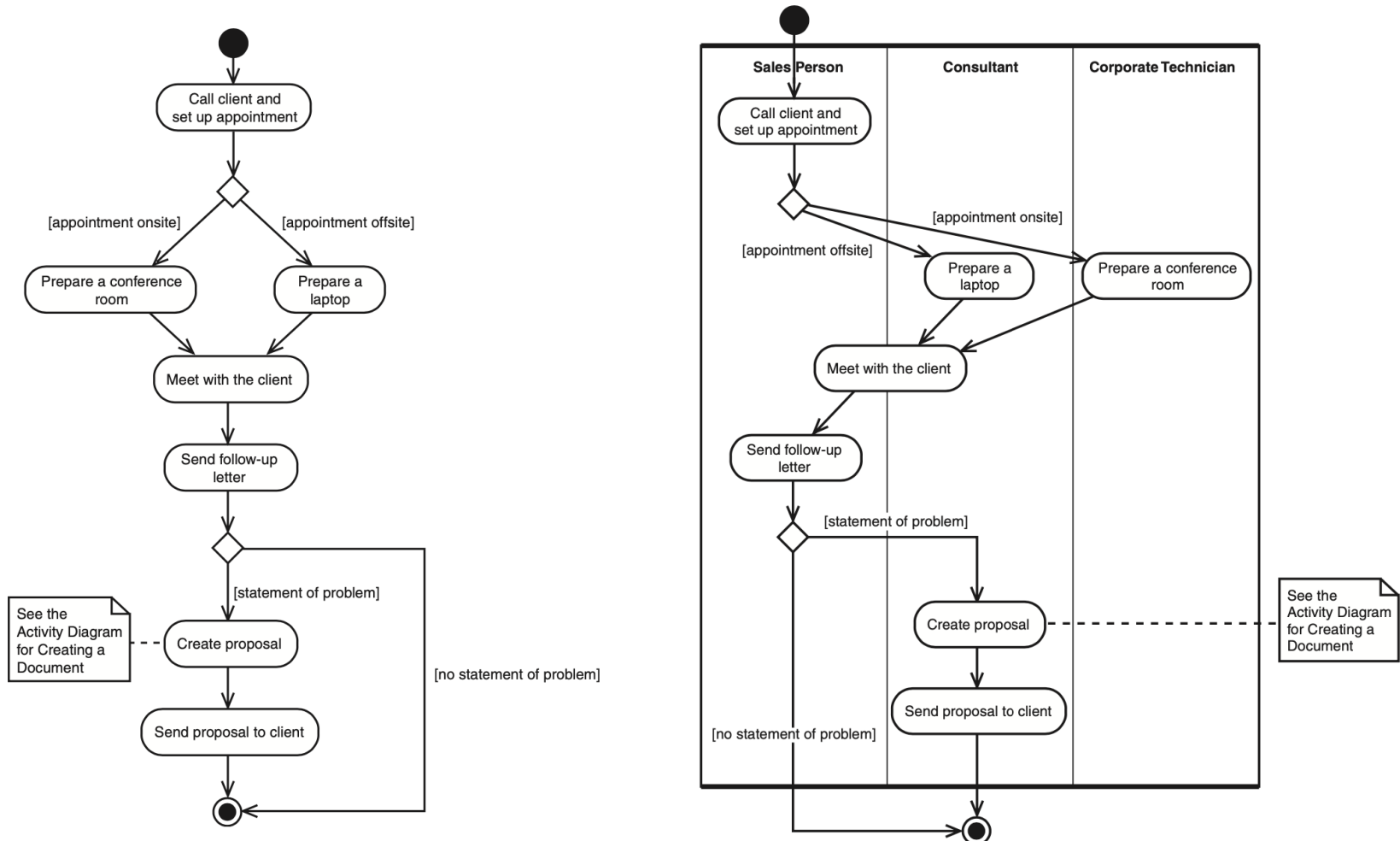


# Swimlane

- ✧ Swimlane is a way to **group activities performed by the same actor on an activity diagram or to group activities in a single thread.**
- ✧ Each swimlane represents one focus of responsibility in the activity diagram.
- ✧ The order of the swimlanes has no significance.

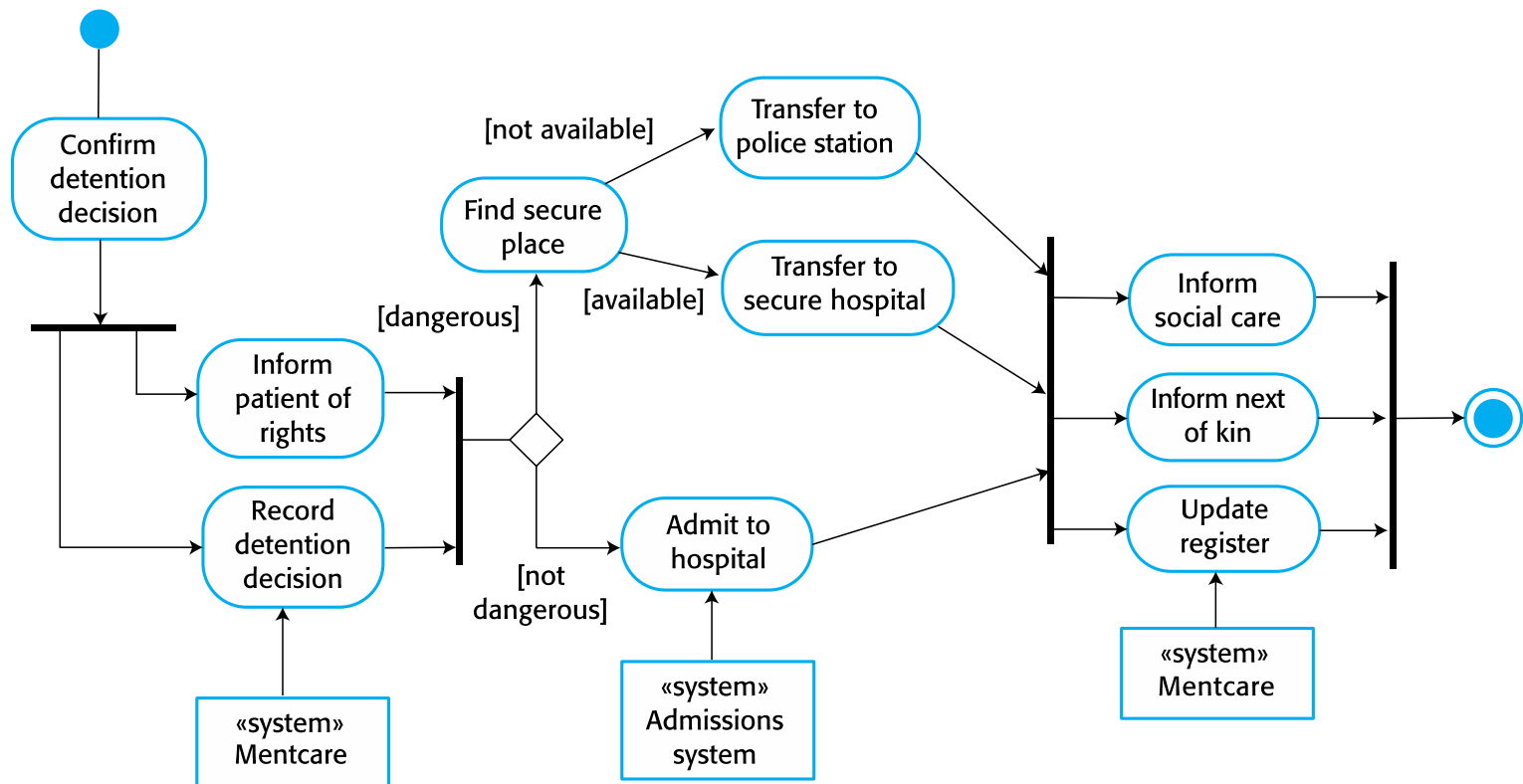


# Example of Swimlane

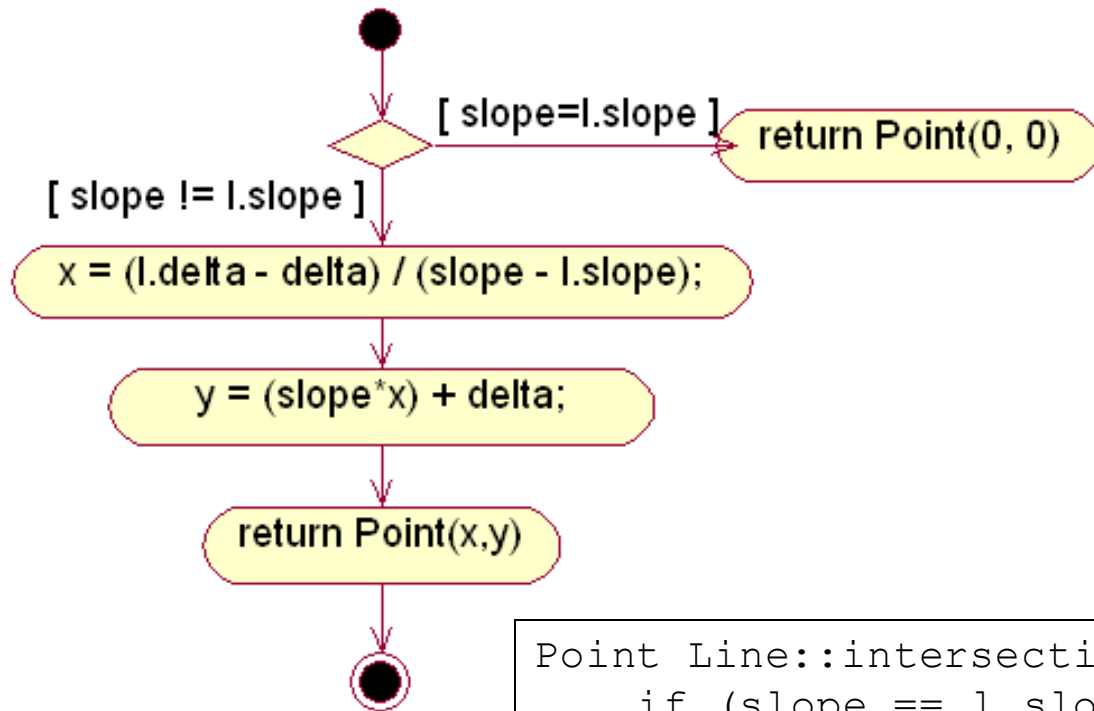


# Example: process model of involuntary detention

- ✧ UML activity diagrams may be used to define business process models.



# Example: intersection of two lines



```
Point Line::intersection (l : Line) {  
    if (slope == l.slope) return Point(0,0);  
    int x = (l.delta - delta) / (slope - l.slope);  
    int y = (slope * x) + delta;  
    return Point(x, y);  
}
```

# Exercise: order processing

---

- ✧ When we receive an order, we check each line item on the order to see if we have the goods in stock. If we do, we assign the goods to the order. If this assignment sends the quantity of those goods in stock below the reorder level, we reorder the goods. While we are doing this, we check to see if the payment is OK. If the payment is OK and we have the goods in stock, we dispatch the order. If the payment is OK but we do not have the goods, we leave the order waiting. If the payment is not OK, we cancel the order.

---

# Sequence Diagram



# Sequence Diagram

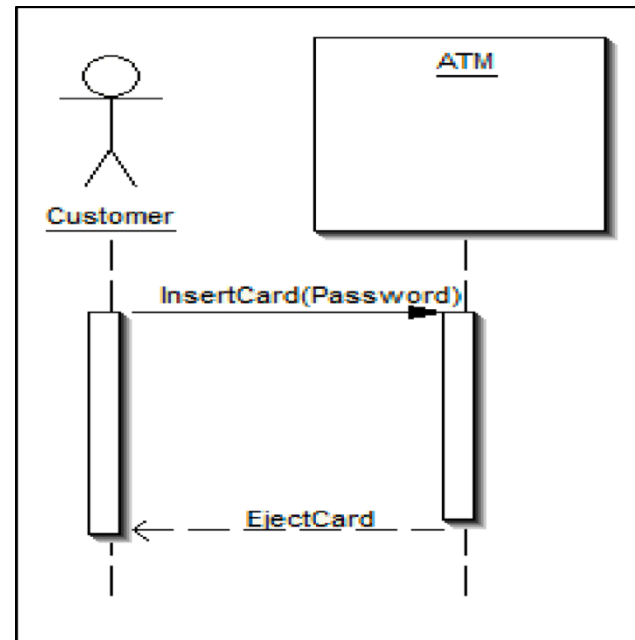
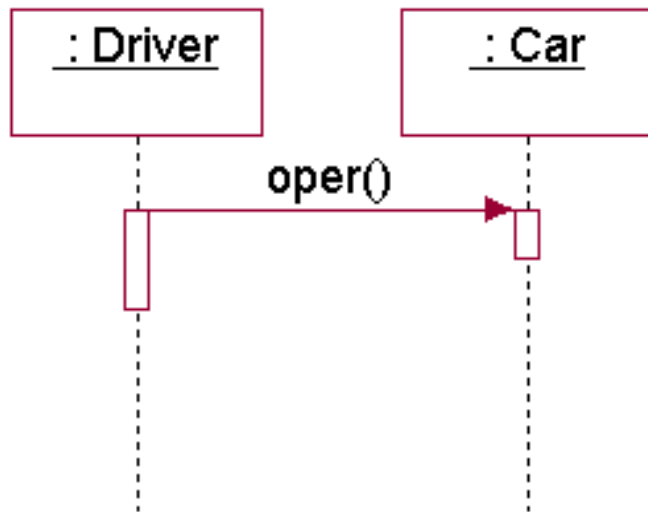
---

- ✧ Sequence diagram is a diagram that shows **object interactions arranged in time sequence**. In particular, it shows the **objects** participating in an interaction and the **sequence of messages** exchanged.
- Describe the flow of messages, events, actions between objects
  - Show concurrent processes and activations
  - Show time sequences that are not easily depicted in other diagrams
  - Typically used during analysis and design to document and understand the logical flow of your system

**Emphasis on time ordering!**

# Sequence Diagram

- ✧ The sequence diagram consists of objects represented as **named rectangles** with the name underlined, messages represented as **annotated arrows**, and time represented as **a vertical progression**.



# Sequence Diagram and Use Cases

---

- ✧ A sequence diagram (SD) shows the sequence of interactions that take place during a particular use case.
- ✧ We draw SDs in order to document **how objects collaborate to produce the functionality of each use case**.
- ✧ The SDs show the **data and messages** which pass across the system boundary and the messages being sent from one object to another in order to achieve the overall functionality of the use case.
- ✧ **A SD can be seen as an expansion of a use case** to the lowest possible level of detail.

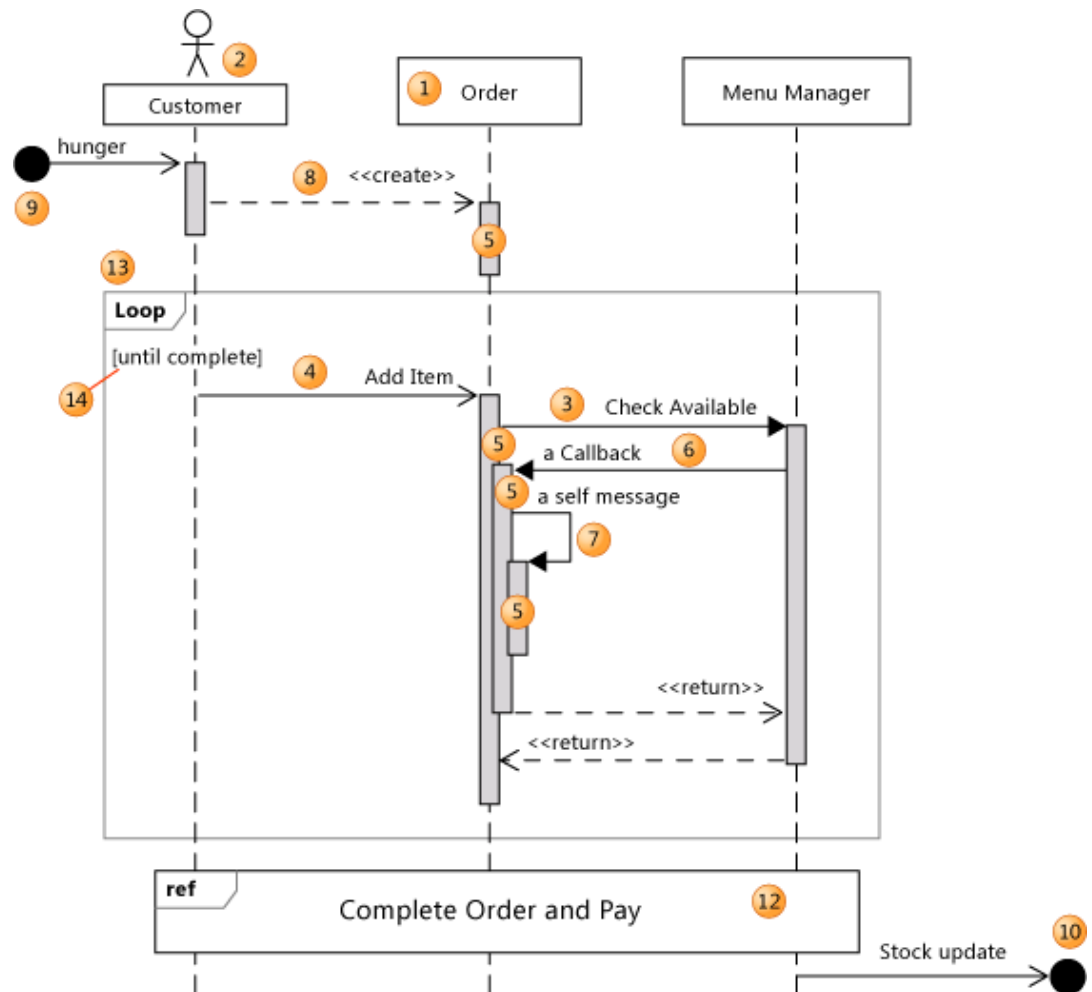
# Key Parts in Sequence Diagram

## ✧ Object Dimension

- The horizontal axis shows the elements that are involved in the interaction

## ✧ Time Dimension

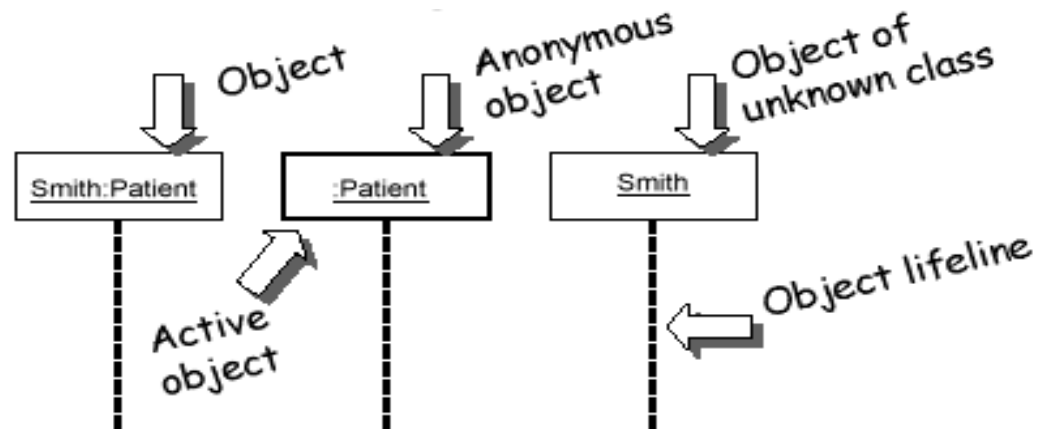
- The vertical axis represents time proceedings (or progressing) down the page



# Object

---

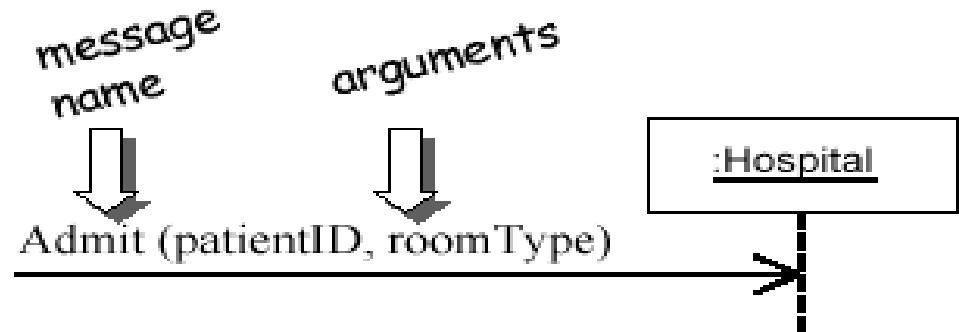
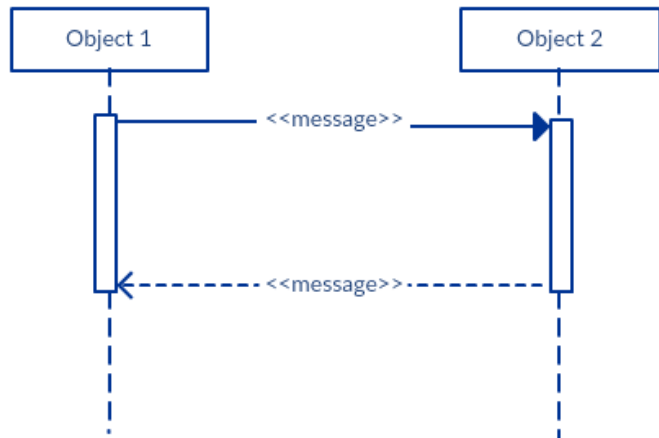
- ✧ The **objects and actors** involved are listed along the top of the diagram, with a dotted line drawn vertically from these. They are arranged in any order that simplifies the diagram.



**Name syntax:** <objectname>:<classname>

# Message

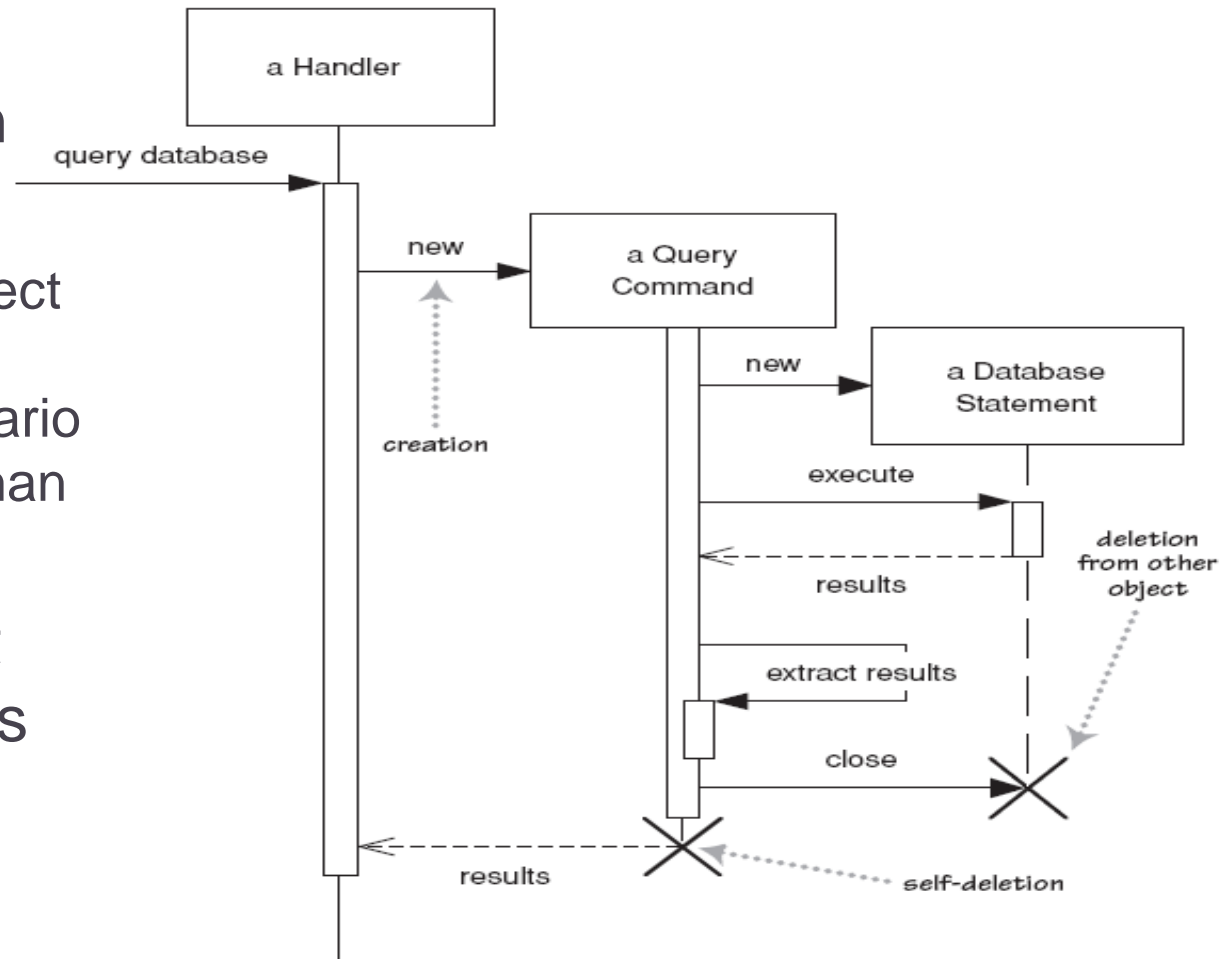
- ✧ An arrow from the Message Caller to the Message Receiver specifies a message in a sequence diagram.
  - Synchronous message
  - Asynchronous message
  - Return message



# Lifeline

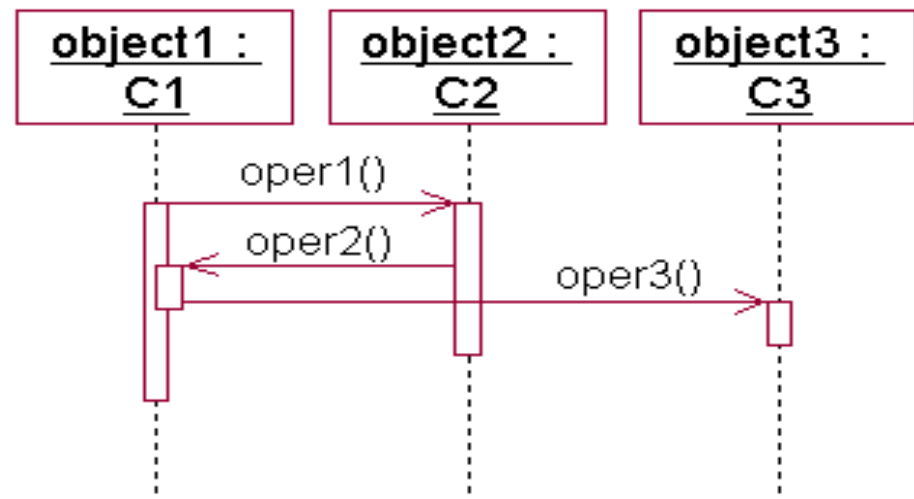
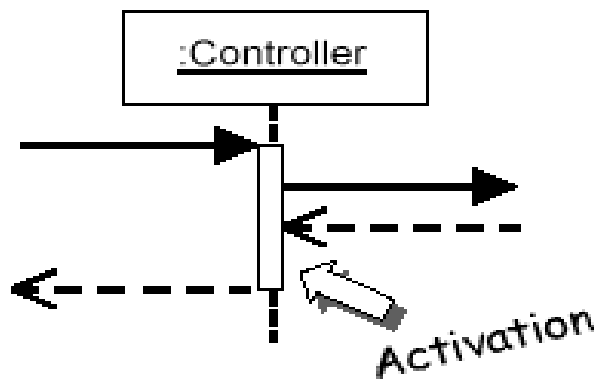
- ✧ **Creation:** arrow with 'new' written above it
  - Note that an object created after the start of the scenario appears lower than the others

- ✧ **Deletion:** an X at bottom of object's lifeline



# Activation

- ✧ Along the lifeline is a narrow rectangle called an activation.
  - Either that object is running its code, or it is on the stack waiting for another object's method to finish
- ✧ The length of the rectangle signifies the activation's duration.

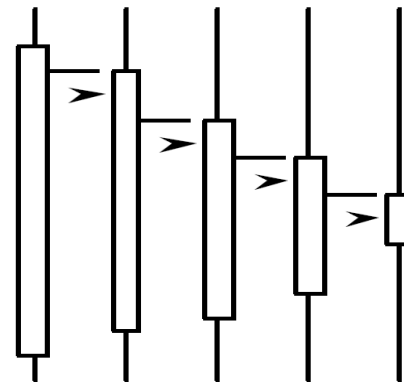
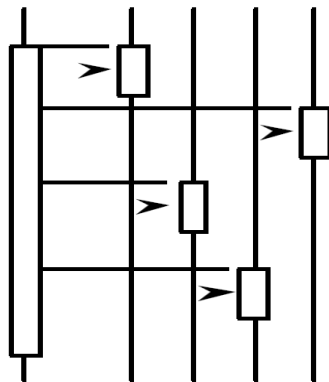




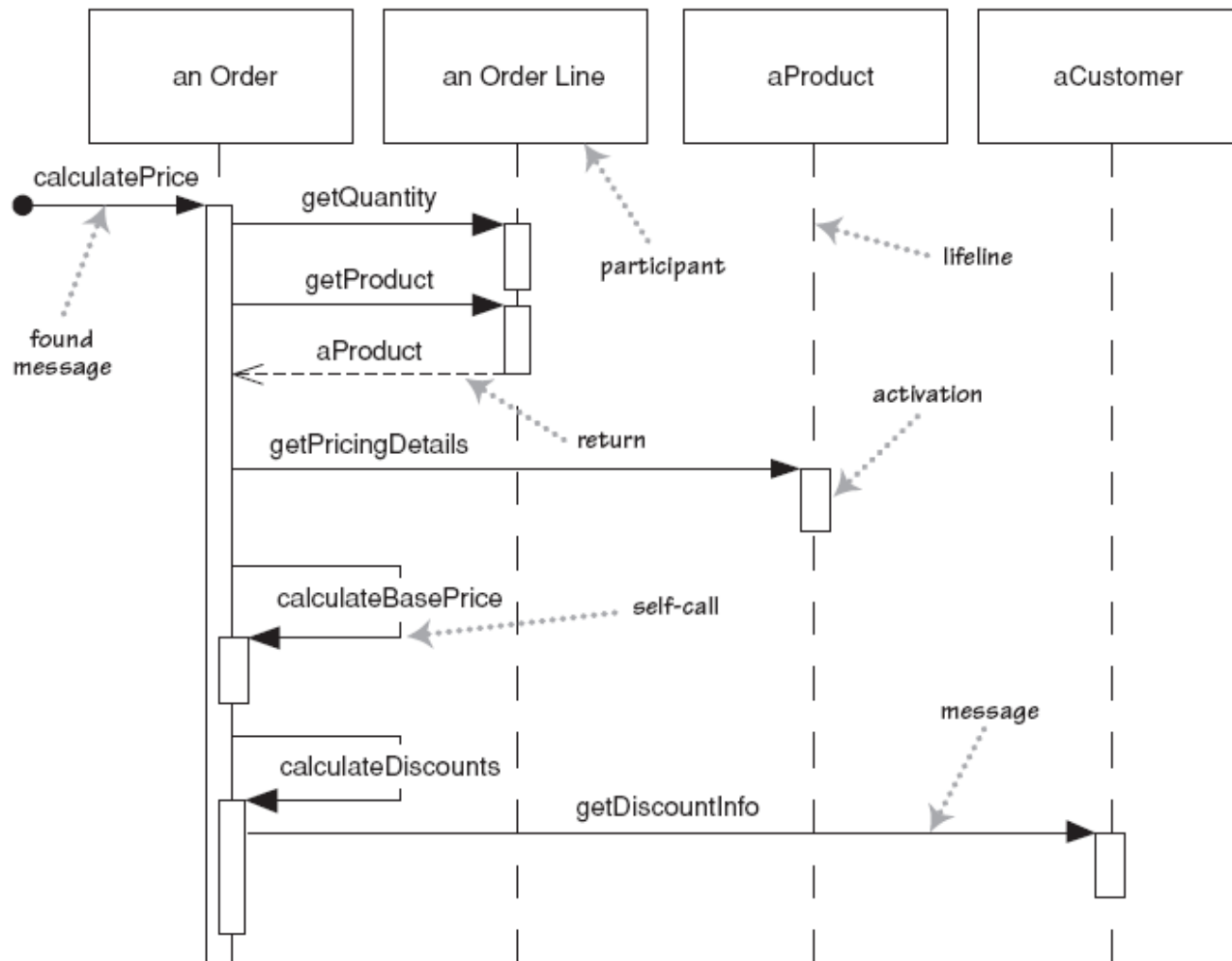
# (De)Centralized System Control

---

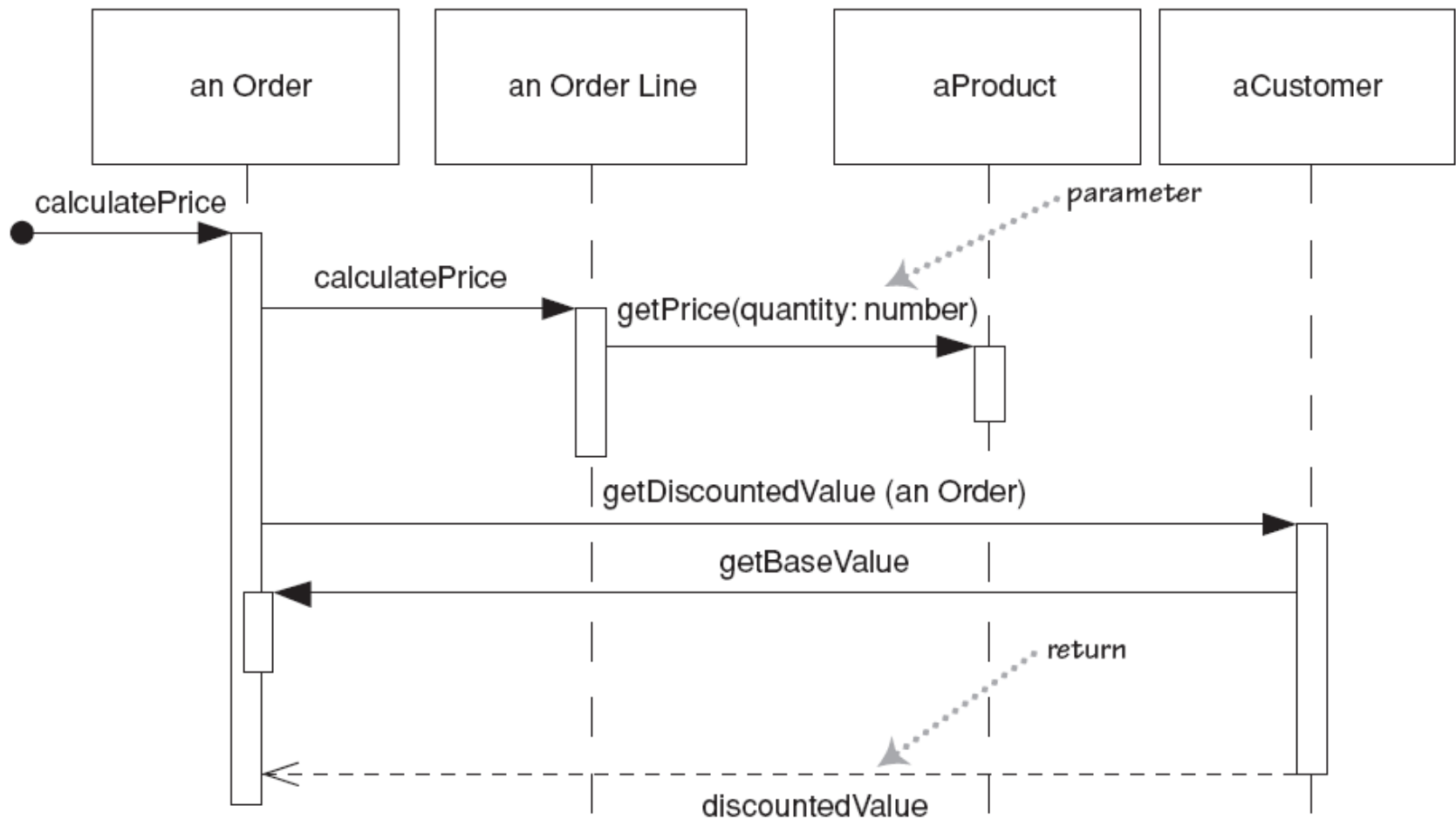
- ✧ What can you say about the control flow of each of the following systems
- Centralized
  - Distributed



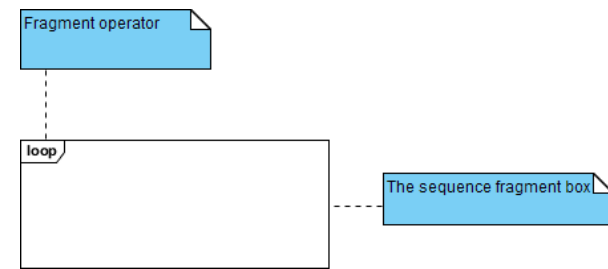
# A Sequence Diagram for Centralized Control



# A Sequence Diagram for Distributed Control



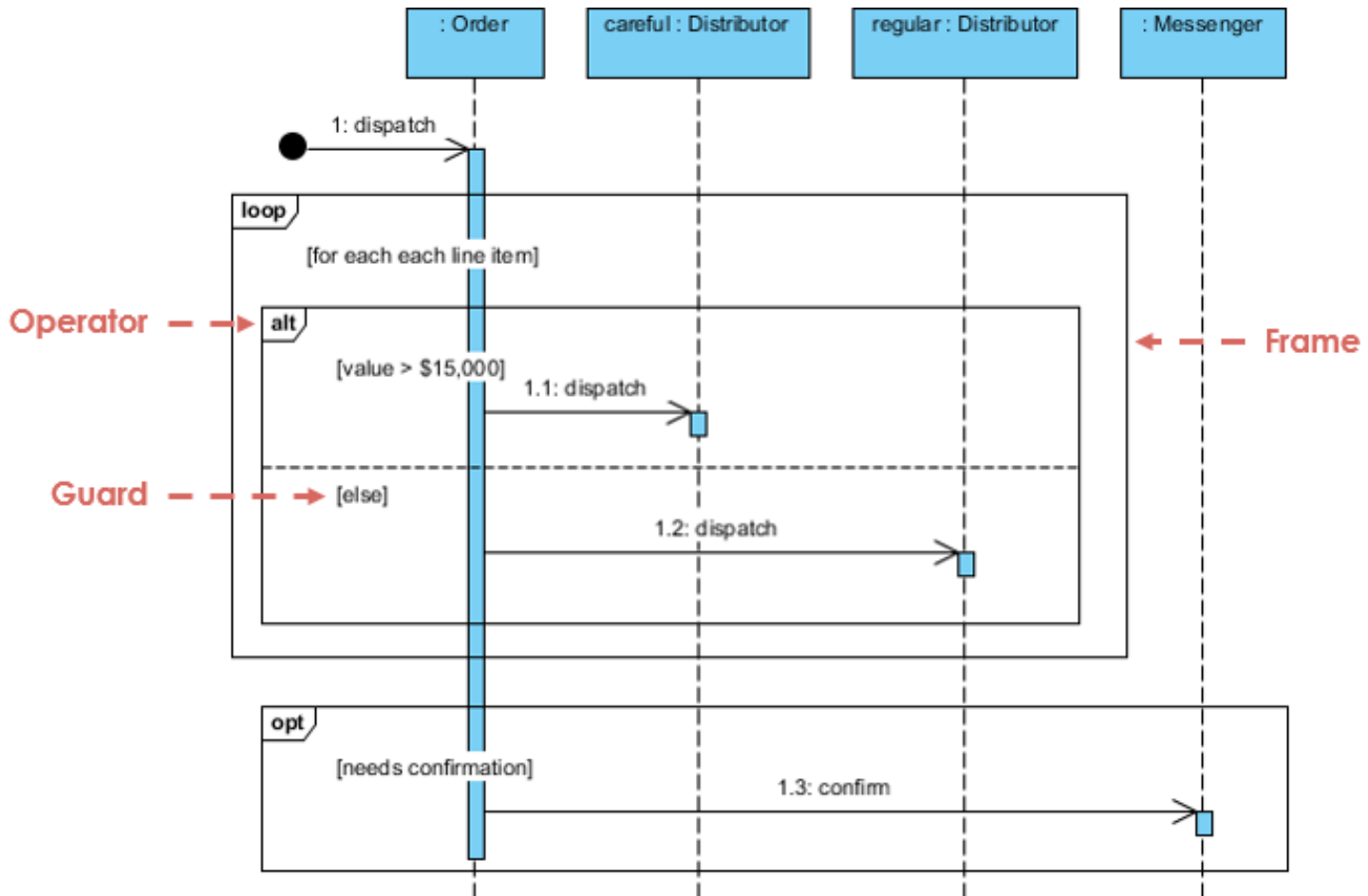
# Sequence Fragment



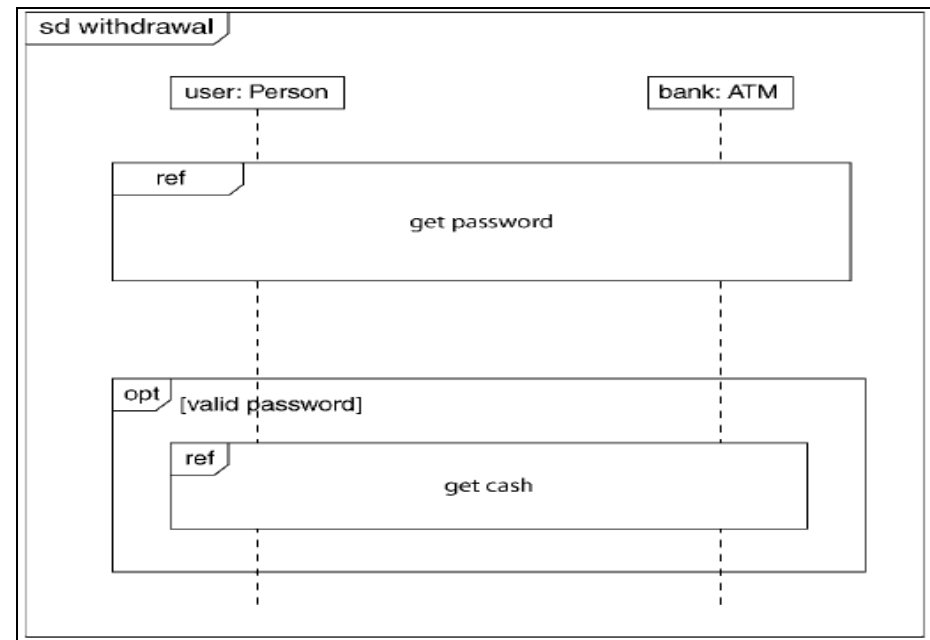
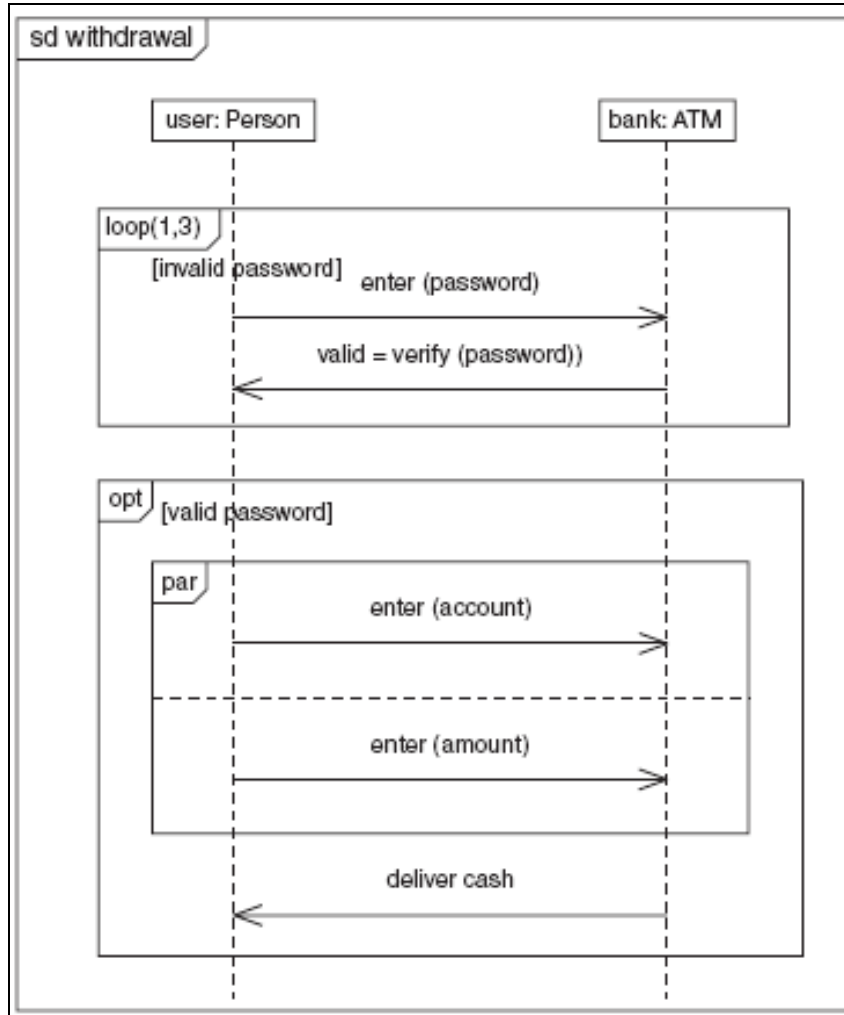
- ✧ A sequence fragment is represented as a box, which encloses a portion of the interactions within a SD.

Operator	Fragment Type
alt	Alternative multiple fragments: only the one whose condition is true will execute.
opt	Optional: the fragment executes only if the supplied condition is true. Equivalent to an alt only with one trace.
par	Parallel: each fragment is run in parallel.
loop	Loop: the fragment may execute multiple times, and the guard indicates the basis of iteration.
ref	Reference: refers to an interaction defined on another diagram. You can define parameters and a return value.
sd	Sequence diagram: used to surround an entire sequence diagram.

# Example: dispatch order



# Example: withdraw cash



# Interaction Models

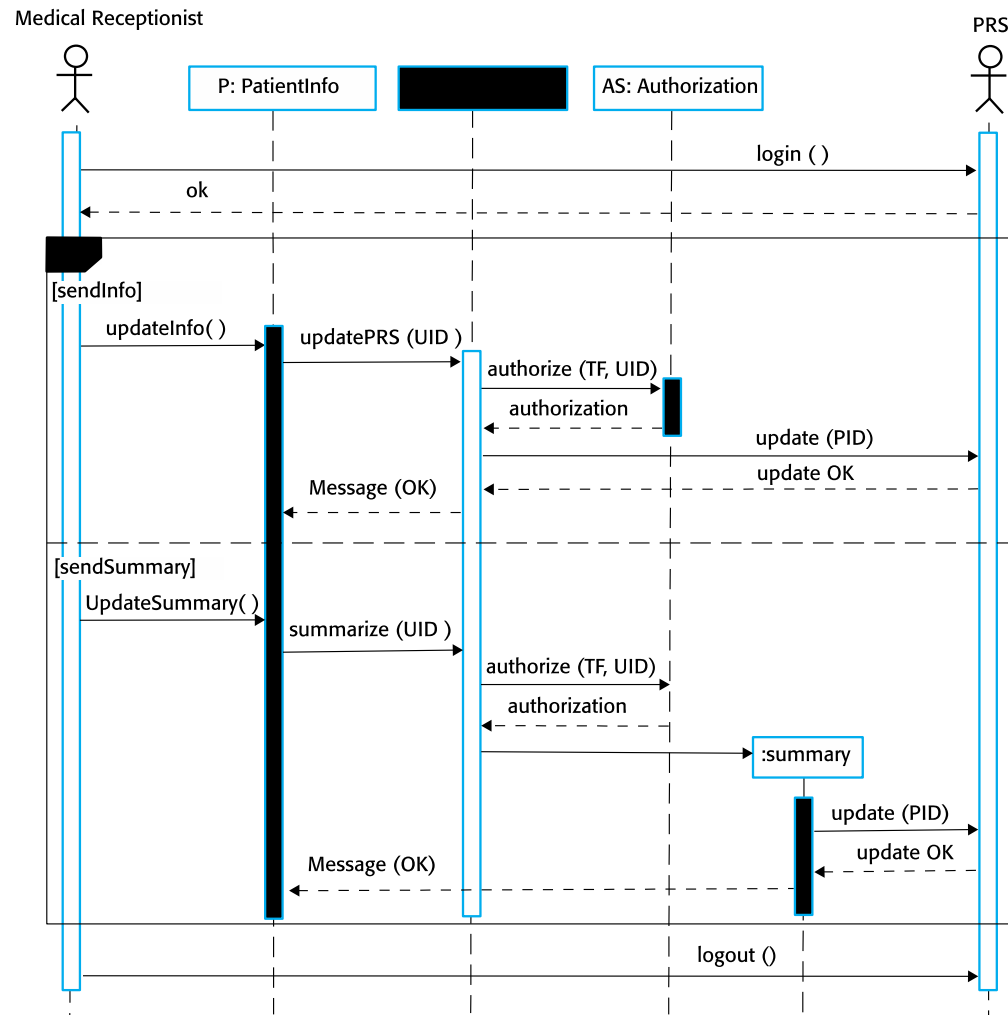
---



## Mentcare system: Transfer data

Actors	Medical receptionist, patient records system (PRS)
Description	A receptionist may transfer data from the Mentcase system to a general patient record database that is maintained by a health authority. The information transferred may either be updated personal information (address, phone number, etc.) or a summary of the patient's diagnosis and treatment.
Data	Patient's personal information, treatment summary
Stimulus	User command issued by medical receptionist
Response	Confirmation that PRS has been updated
Comments	The receptionist must have appropriate security permissions to access the patient information and the PRS.

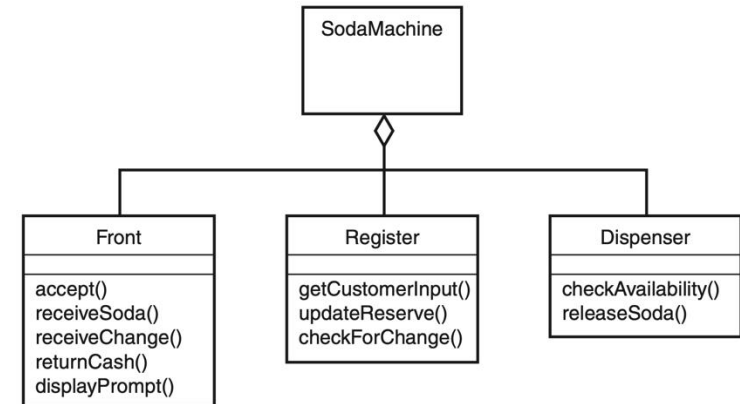
# Interaction Models





# Exercise: buy soda

- ✧ Model the best-case scenario, the sold-out scenario, the incorrect-change scenario, the incorrect-change-and-machine-is-out-of-change scenario.



## The front

- Accepts selections and cash
- Displays prompts like "Out of selection" and "Use correct change"
- Receives change from the register and makes it available to the customer
- Returns cash
- Receives a can of soda from the dispenser and makes it available to the customer

## The register

- Gets the customer's input (that is, the selection and the cash) from the front
- Updates its cash reserve
- Checks for change

## The dispenser

- Checks the availability of a selection
- Releases a can of soda

# Why not just code it?

---

- ✧ Sequence diagrams can be somewhat close to the code level. So why not just code up that algorithm rather than drawing it as a sequence diagram?
  - A good sequence diagram is still a bit above the level of the real code (not EVERY line of code is drawn on diagram)
  - Sequence diagrams are language-agnostic (can be implemented in many different languages)
  - Non-coders can do sequence diagrams
  - Easier to do sequence diagrams as a team
  - Can see many objects/classes at a time on same page

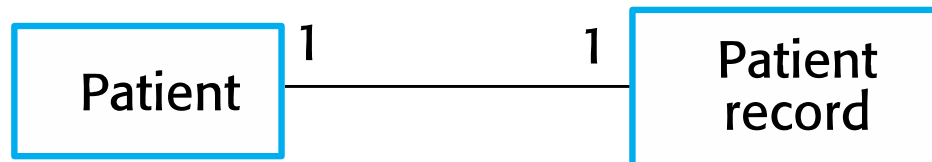
---

# **Class Diagram**

# Class Diagram

---

- ✧ Class diagram is a type of static structure diagram that **describes the structure of a system** by showing the system's classes, their attributes, operations, and the relationships among objects.
- ✧ A class diagram is made up of:
  - A set of classes
  - A set of relationships between classes



# Class Notation

---

✧ Class name (**mandatory**)

✧ Class attributes (**optional**)

- **An attribute is a property of a class.** It describes a range of values that the property may hold in objects of that class.
- A class may have zero or more attributes.

✧ Class operations (**optional**)

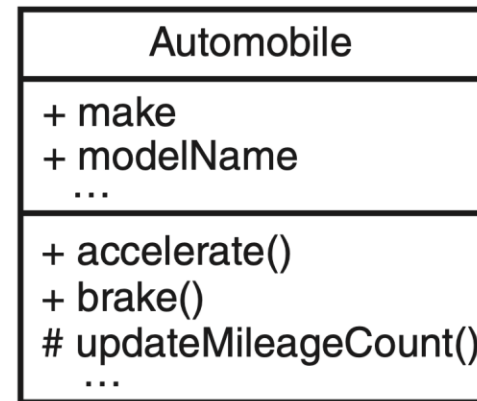
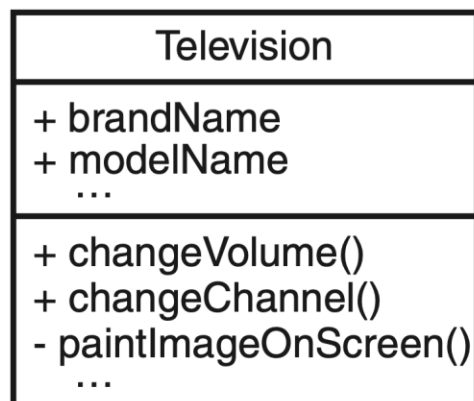
- **An operation is something a class can do,** and hence it is something that you (or another class) can ask the class to do.
- The return type of a method is shown after the colon at the end of the method signature.

WashingMachine
brandName modelName serialNumber capacity
acceptClothes(c:String) acceptDetergent(d:Integer) turnOn():Boolean turnOff():Boolean

# Visibility of Class Attributes and Operations

- ✧ Visibility applies to attributes or operations and specifies the extent to which other classes can use a given class's attributes or operations.

Access Right	public (+)	private (-)	protected (#)
Members of the same class	yes	yes	yes
Members of derived classes	yes	no	yes
Members of any other class	yes	no	no



# Finding Classes

---

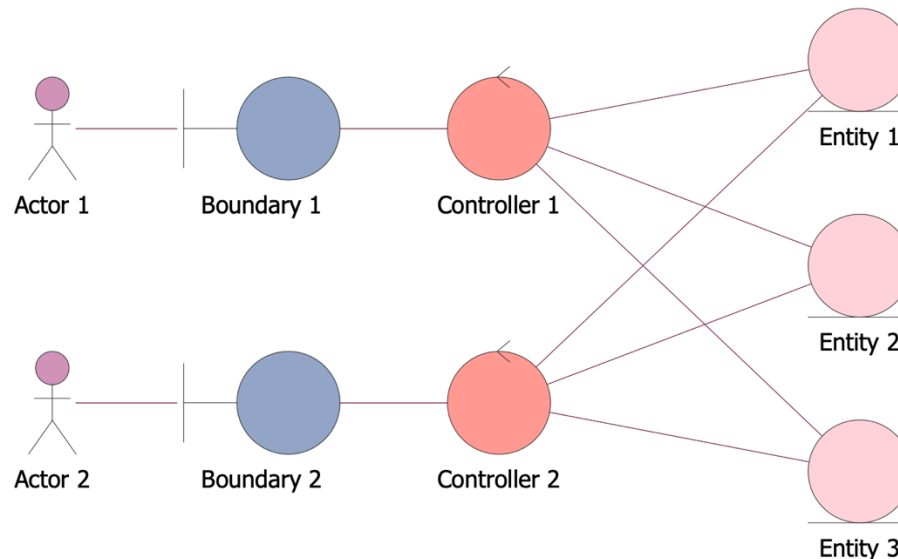
- ✧ In stakeholders' descriptions of the problem:
  - Be alert to the **nouns** because these will become the classes in your model.
  - Be alert to the **verbs** because these will constitute the operations in those classes.
  - The attributes will emerge as **nouns** related to the class nouns.
- ✧ It's better to include many candidate classes at first because:
  - You can always eliminate them later if they turn out not to be useful.
  - Explicitly deciding to discard classes is better than just not thinking about them.

# Selecting Classes

---

✧ Discard classes for concepts which:

- Are beyond the scope of the analysis
- Refer to the system as a whole
- Duplicate other classes
- Are too vague or too specific






# Relationship Notation

---

- ✧ A class may be involved in one or more relationships with other classes.
- ✧ A relationship can be one of the following types:

Dependency 

Aggregation 

Inheritance 

Composition 

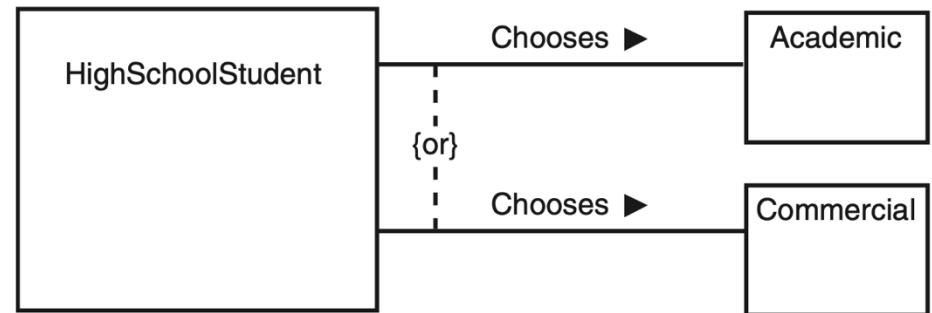
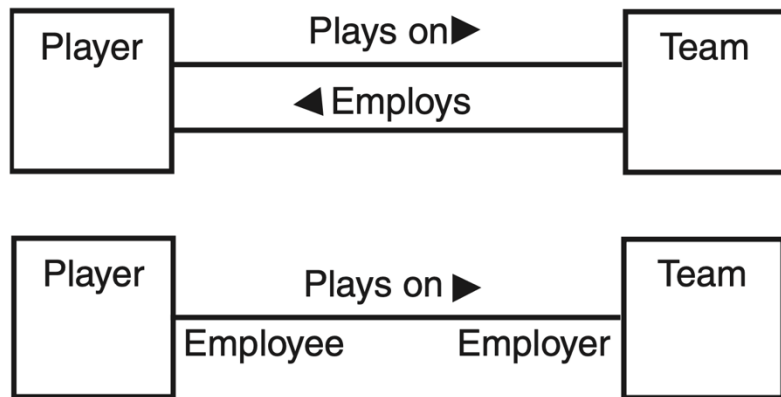
Association 

Directed Association 

Interface Type Implementation 

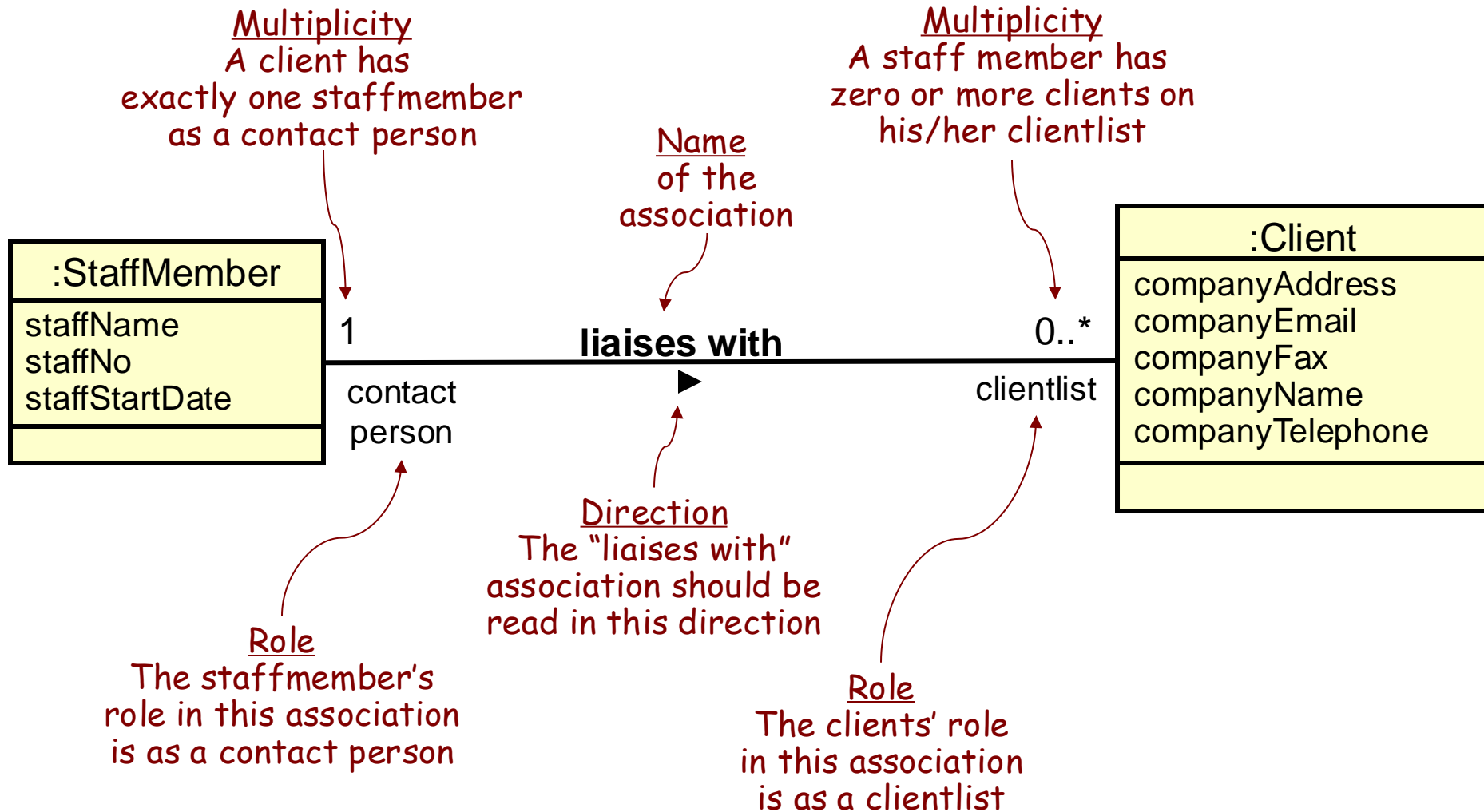
# Association

- ✧ When classes are connected together conceptually, that connection is called an association.
- ✧ An association is visualized as a line connecting the two classes, with the **name of the association** just above the line. When one class associates with another, each one usually plays a **role within that association**.



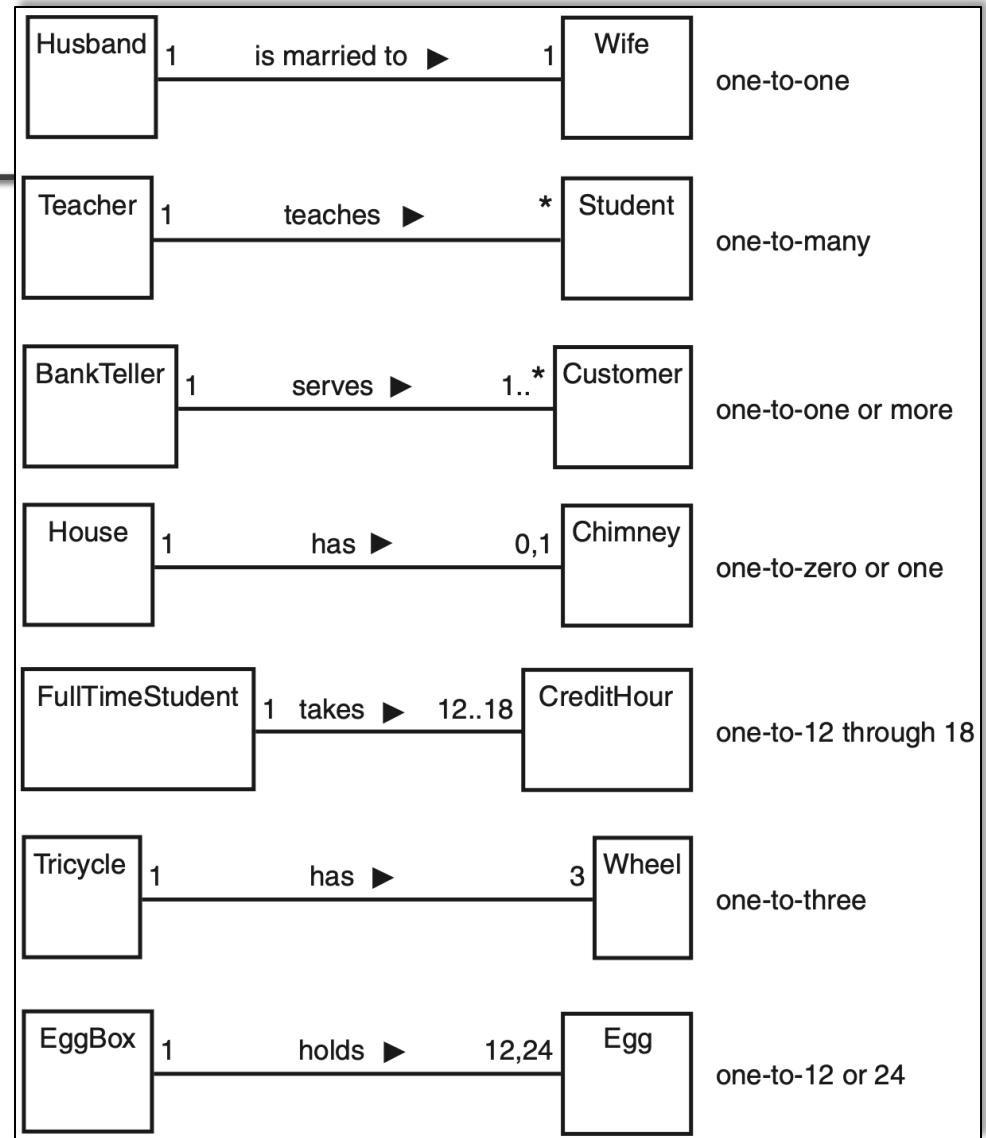
Constraints on Associations

# Association



# Multiplicity

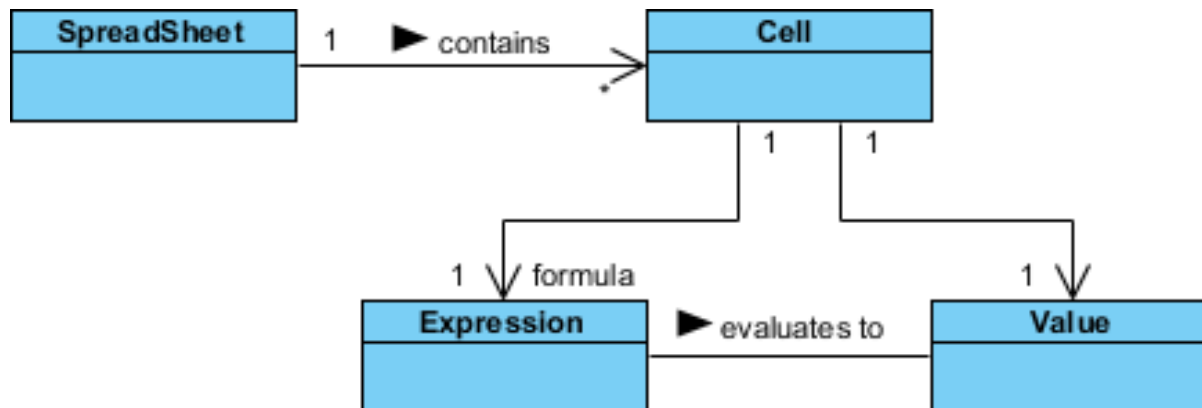
- ✧ Multiplicity denotes the number of objects of one class that can relate to one object of an associated class.
- ✧ When class A is in a one-to-zero or one multiplicity with class B, class B is said to be optional for class A.



# Navigability

---

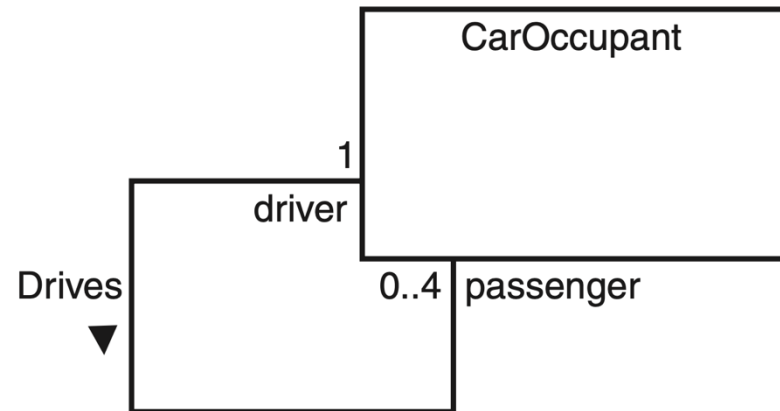
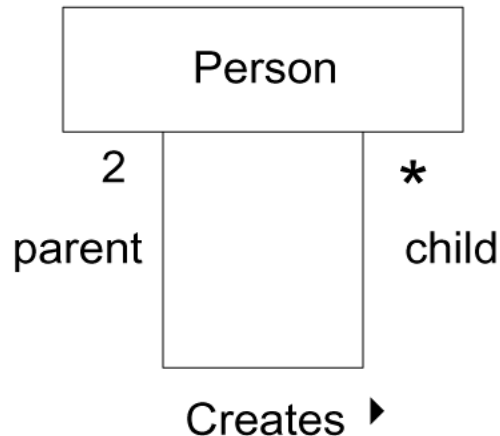
- ✧ The arrows indicate whether, given one instance participating in a relationship, it is possible to determine the instances of the other class that are related to it.
  - Given a spreadsheet, we can locate all of the cells that it contains, but we cannot determine from a cell in what spreadsheet it is contained.



# Reflexive Associations

---

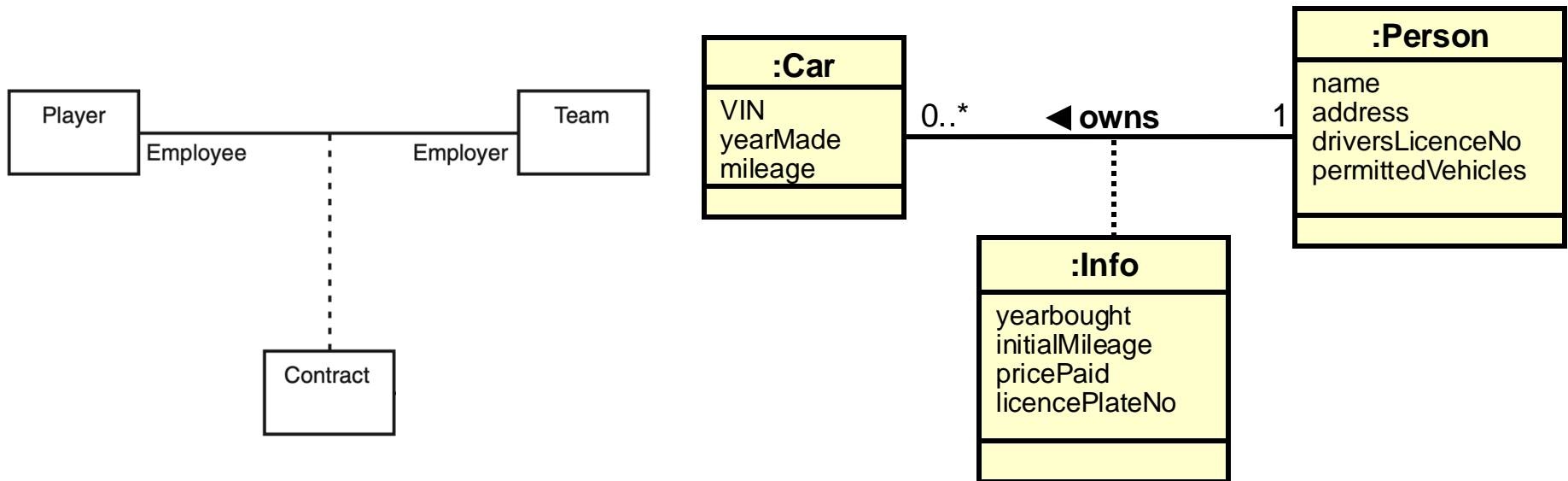
- ✧ Sometimes, a class is in an association with itself. Referred to as a reflexive association, this can happen when a class has objects that play a variety of roles.



# Association Class

✧ An association class models **an association's attributes and operations**.

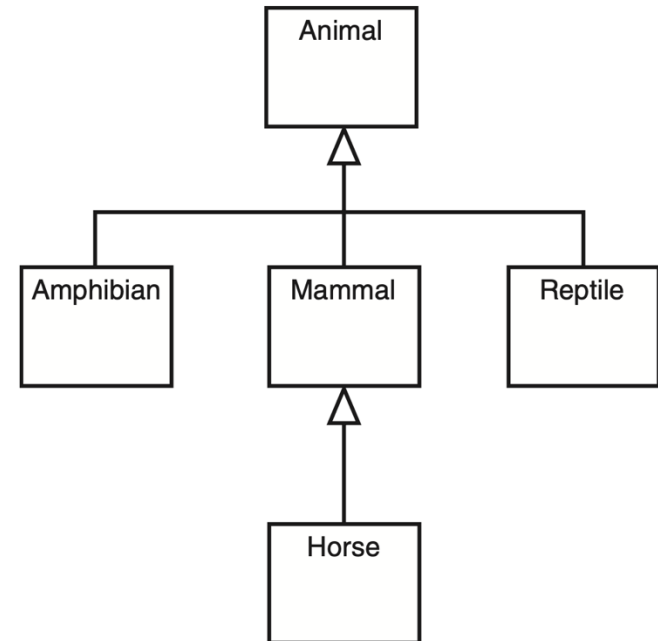
- Because we need to retain information about the association and that information doesn't naturally live in the classes at the ends of the association.



# Inheritance and Generalization

---

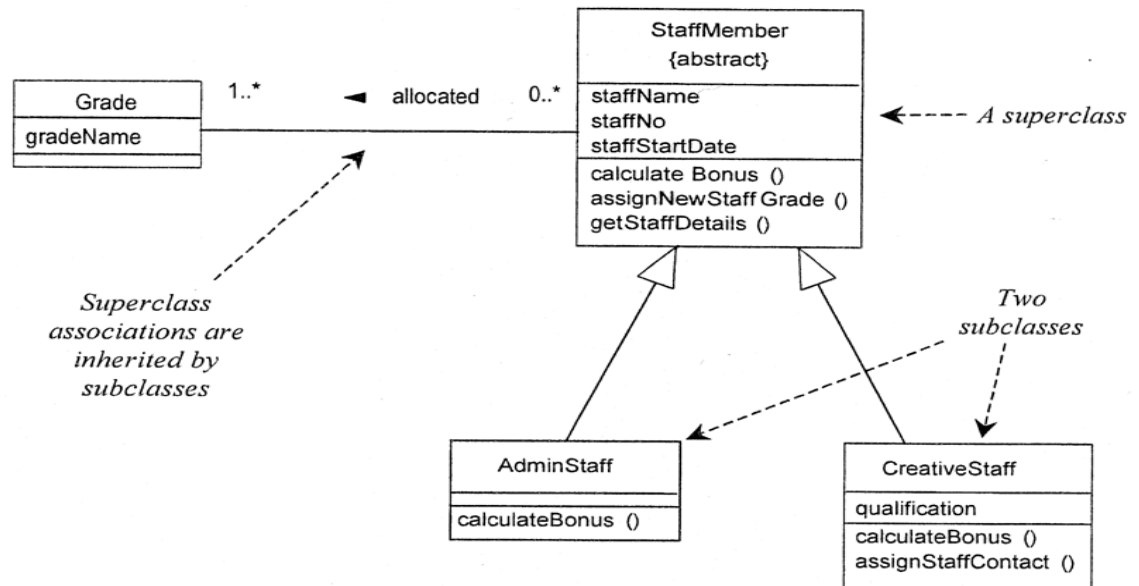
- ✧ In modeling systems, it is often useful to examine the classes in a system to see if there is scope for generalization. If changes are proposed, then you do not have to look at all classes in the system to see if they are affected by the change.
- ✧ In object-oriented languages, such as Java, **generalization is implemented using the class inheritance mechanisms built into the language.**





# Inheritance and Generalization

- ✧ The lower-level classes are subclasses inherit the attributes and operations from their superclasses. These lower-level classes then add more specific attributes and operations.
- ✧ When modeling inheritance, be sure the child class satisfies the “**is a kind of**” relationship with the parent class.
- ✧ An abstract class name is shown in italics (or declared as {abstract}), meaning they have no instances.



# Discovering Inheritance

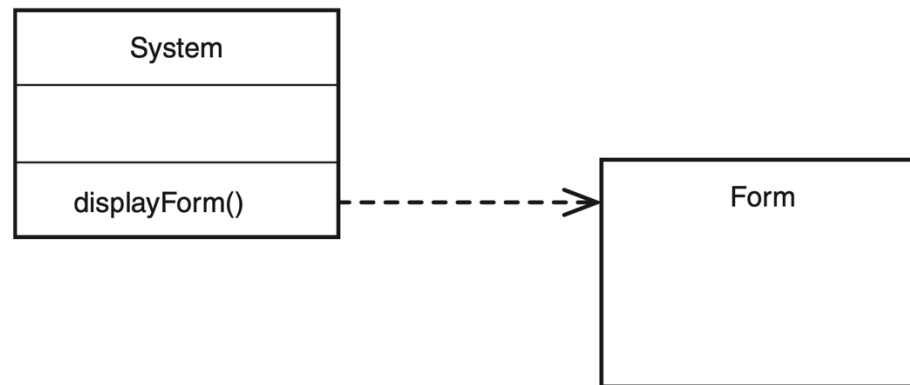
---

- ✧ Look for generalizations in two ways:
  - **Top Down**: you have a class, and discover it can be subdivided.
  - **Bottom Up**: you notice similarities between classes you have identified.
- ✧ But don't generalize just for the sake of it:
  - Be sure that everything about the superclass applies to the subclasses.
  - Be sure that the superclass is useful as a class in its own right.
  - Don't add subclasses or superclasses that are not relevant to your analysis.

# Dependency

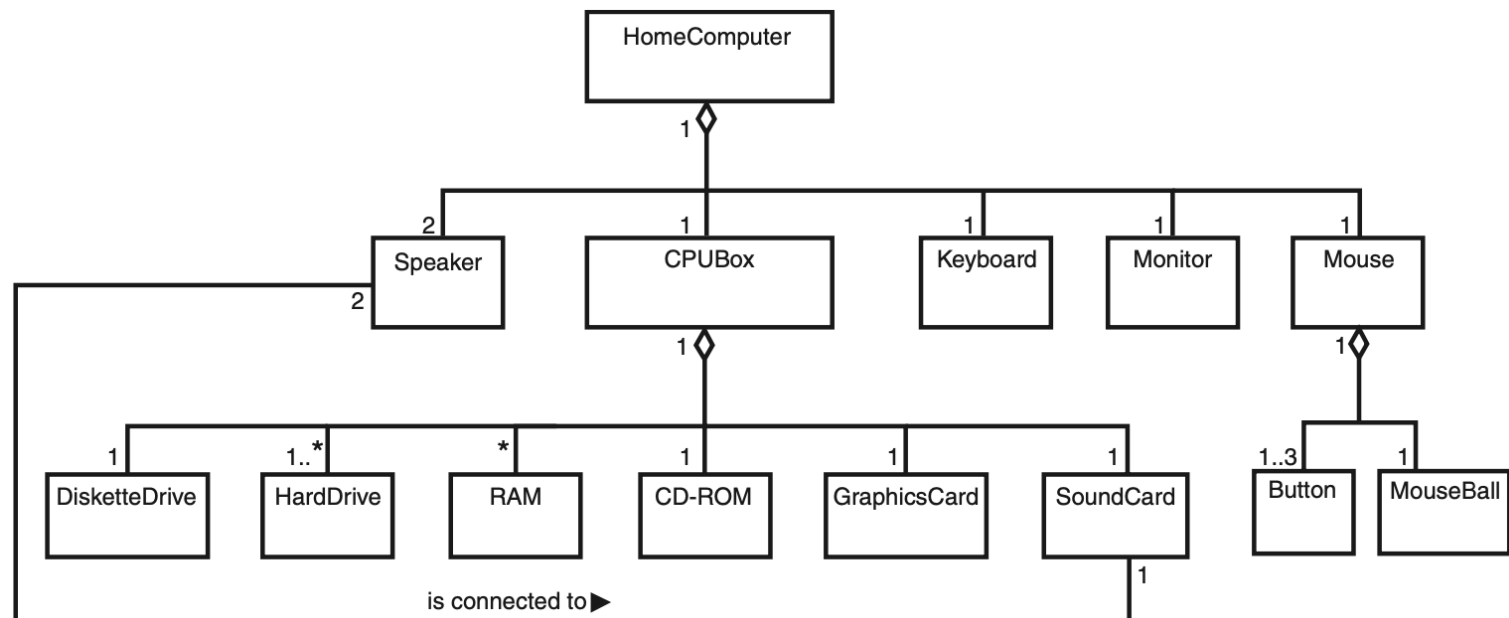
---

- ✧ In a dependency, one class uses another.
  - An association almost always implies that one object has the other object as a field/property/attribute (terminology differs).
  - A dependency typically (but not always) implies that **an object accepts another object as a method parameter, instantiates, or uses another object.**



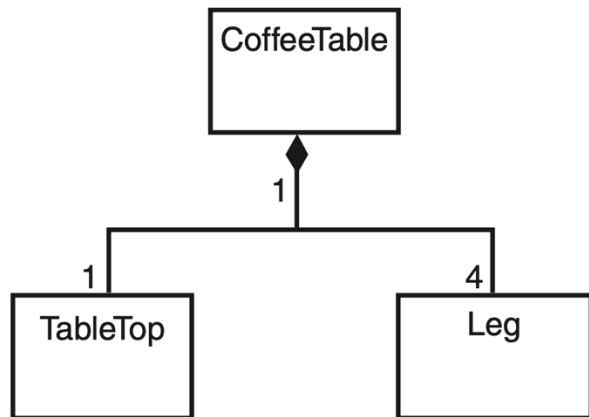
# Aggregation and Composition

- ✧ Sometimes a class consists of a number of component classes. This is a special type of relationship called an aggregation. The components and the class they constitute are in a **part-whole** association.

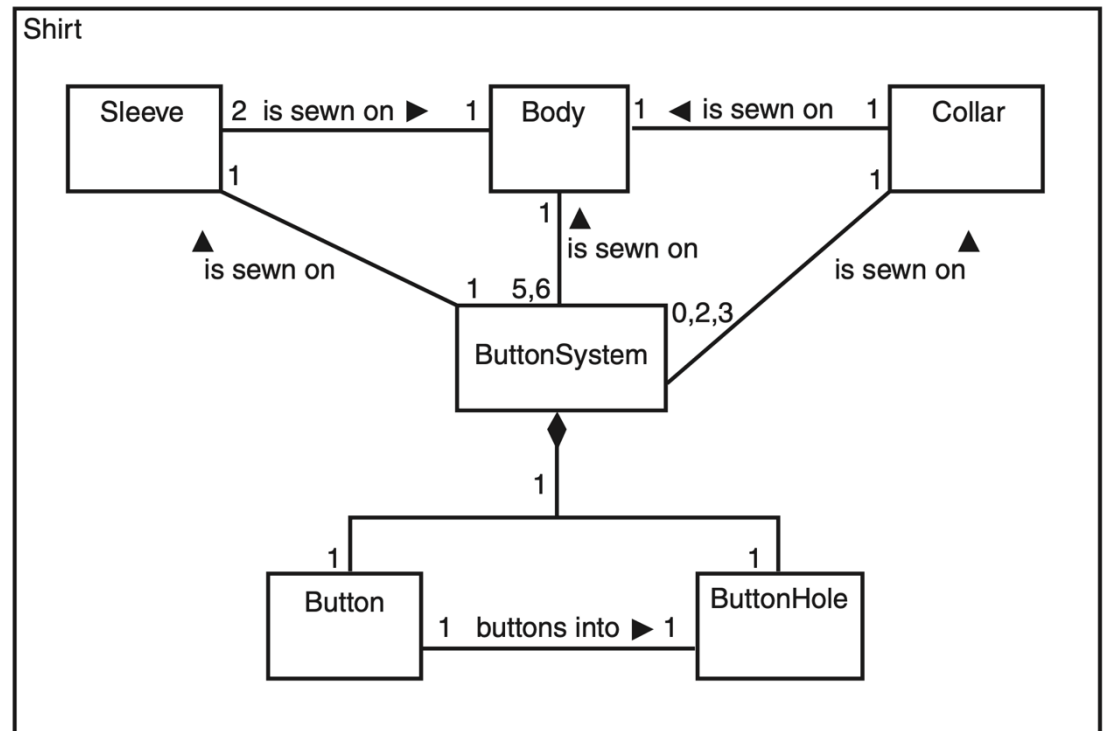


# Aggregation and Composition

- ✧ A composite is a strong type of aggregation. Each component in a composite can belong to just one whole.

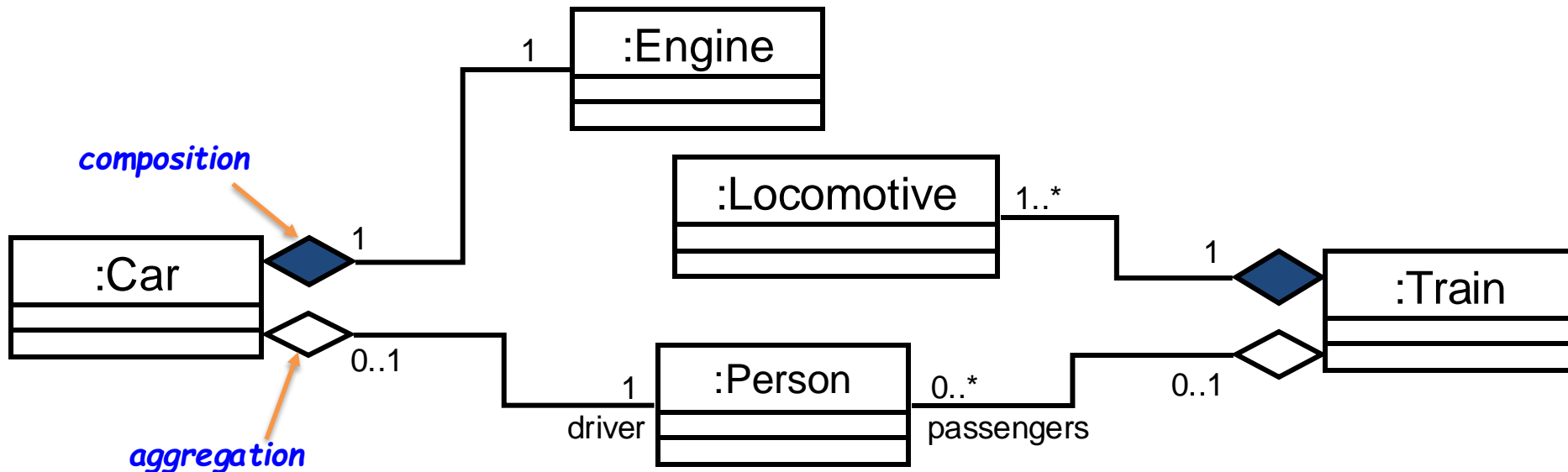


The composite is one way to show the components of a class. If you want to give the sense of showing the class's internal structure, you can go a step further with the **composite structure diagram**.



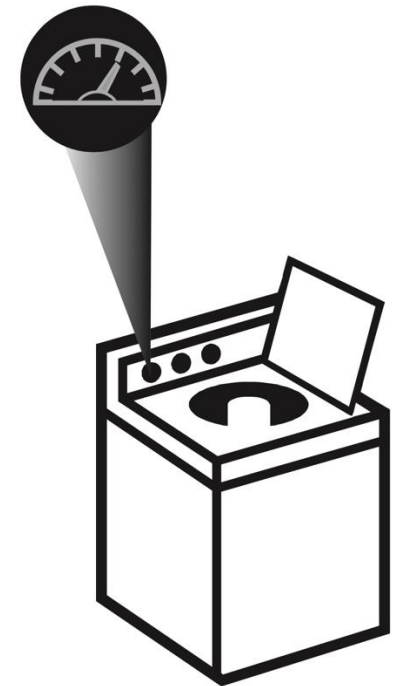
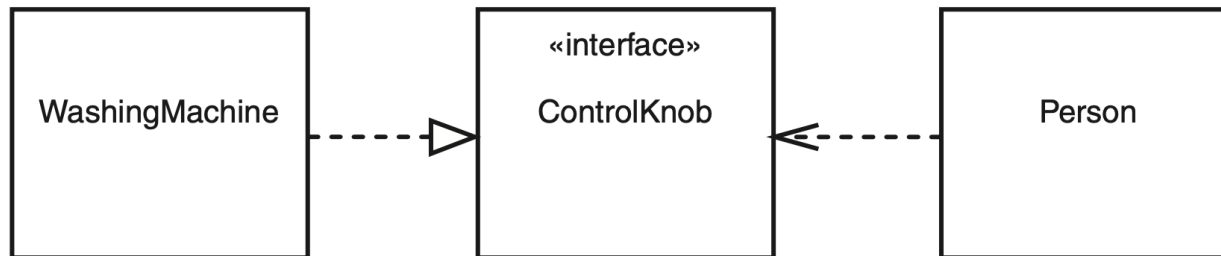
# Aggregation and Composition

- ✧ Aggregation (implies “has-a” or “part-whole”)
- ✧ Composition (implies “ownership”)
  - If the whole is removed from the model, so is the part.
  - The whole is responsible for the disposition of its parts.



# Interface and Realization

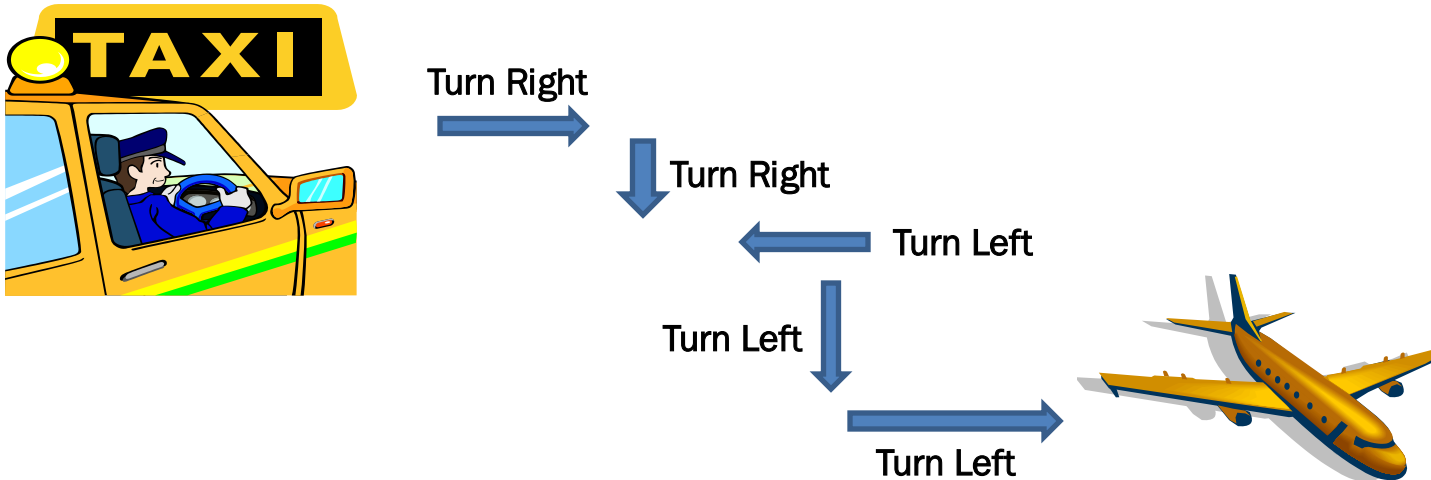
- ✧ An interface is **a set of operations** that specifies some aspect of a class's behavior, and it's a set of operations a class presents to other classes.
- ✧ A class is related to an interface via **realization**. A class can realize more than one interface, and an interface can be realized by more than one class.



# Interface and Realization

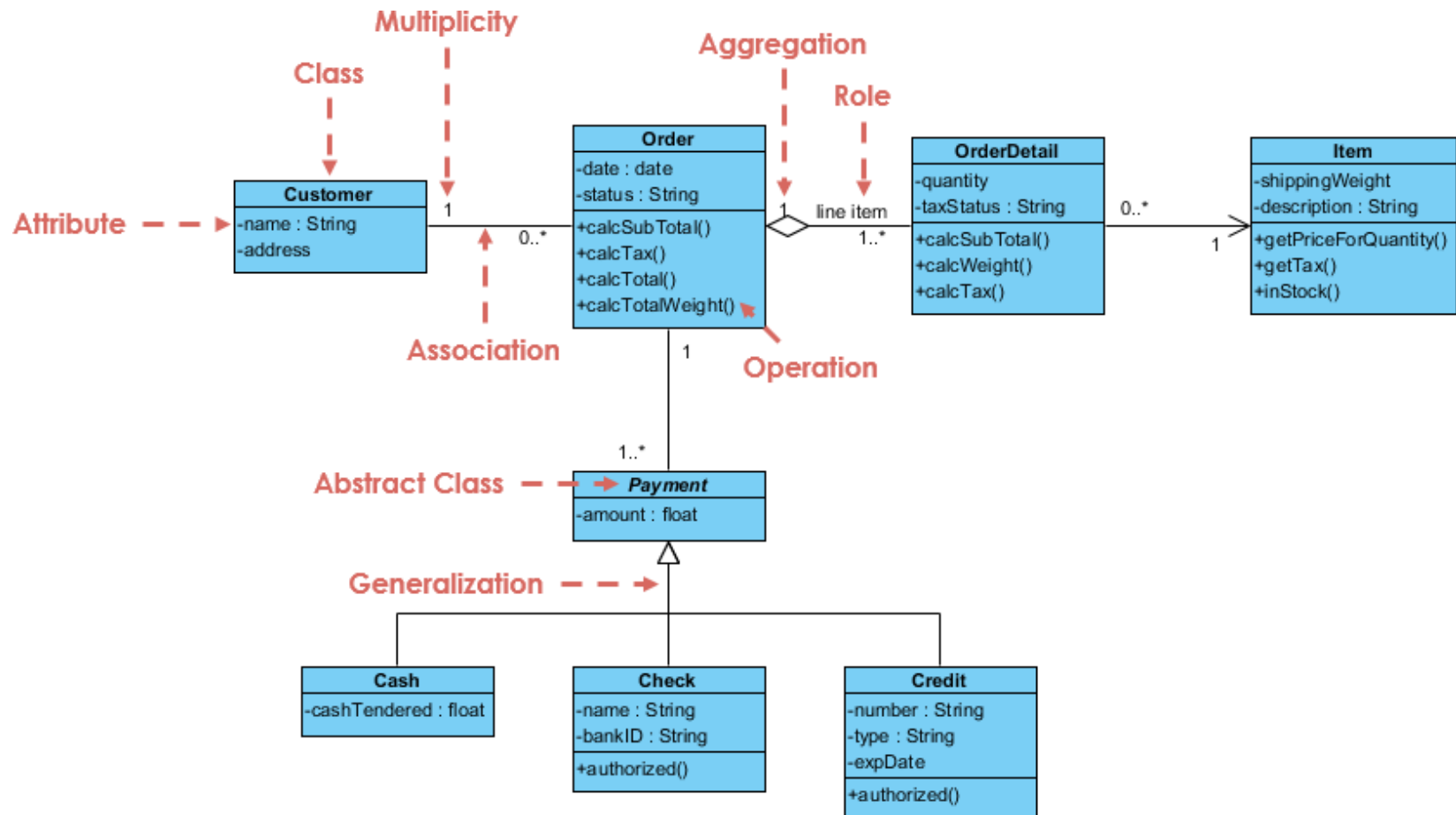
---

- ✧ To develop a good sense of OO thought process:
  - Knowing the difference between the interface and implementation
  - Thinking more abstractly
  - Giving the user the minimal interface possible

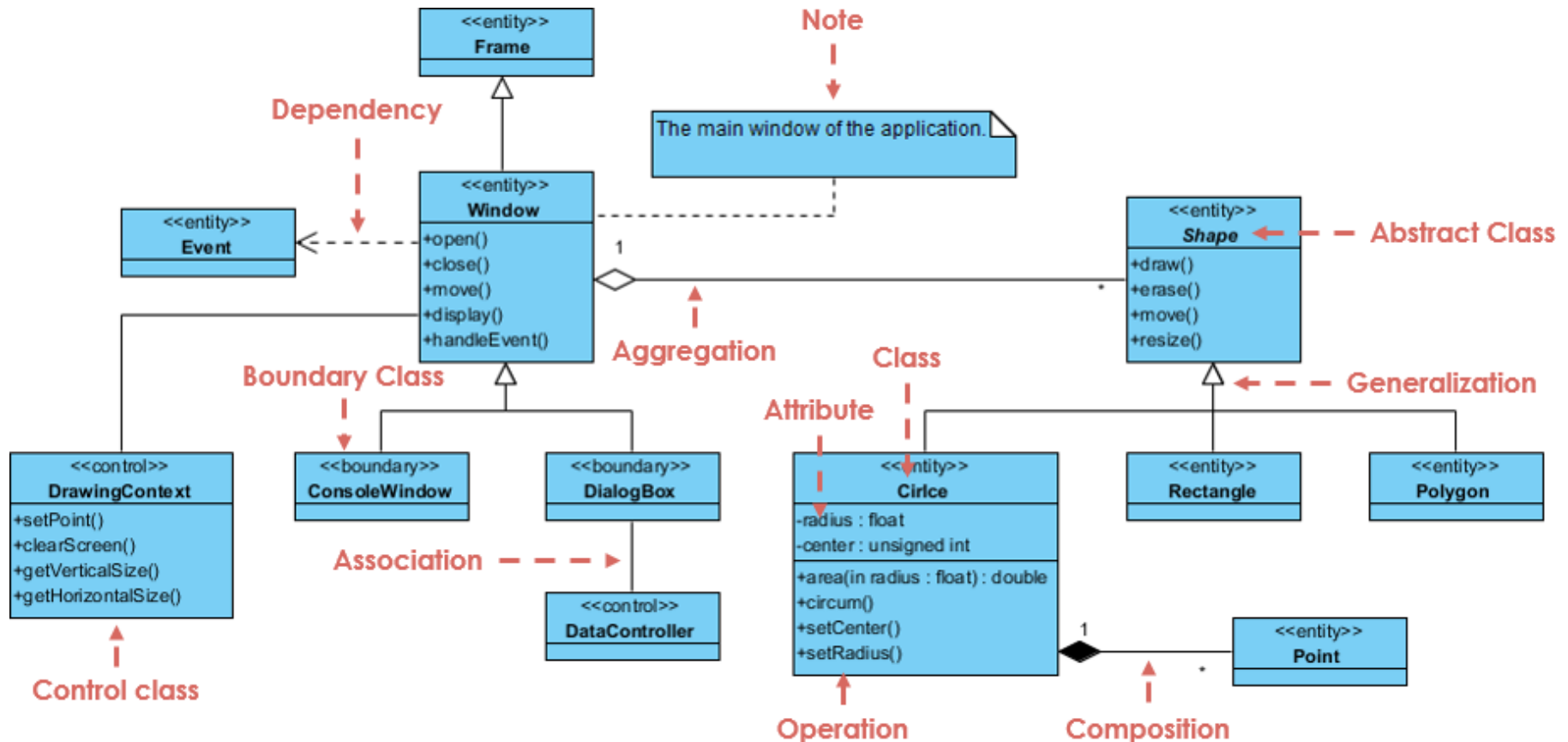




# Example: order system



# Example: GUI



# Exercise

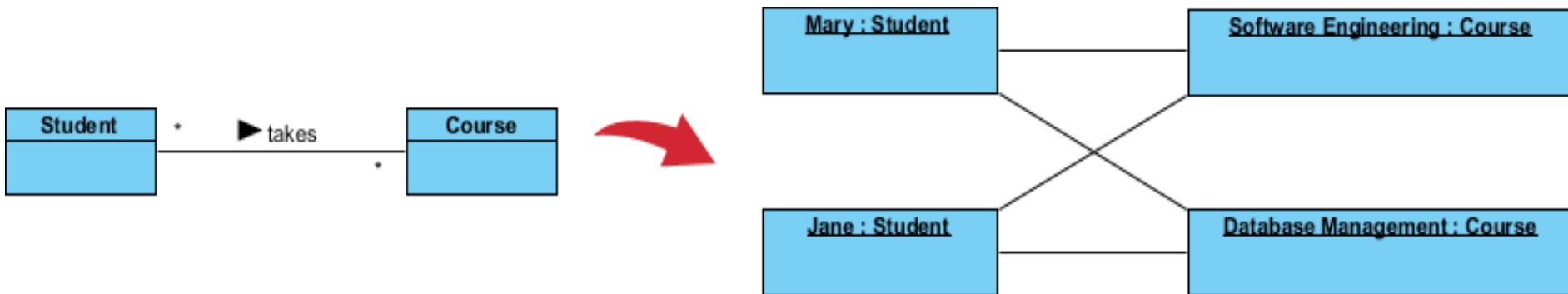
---

- ✧ 假定我们要建立一个学校的信息系统，需要记录以下信息：
- 学校中的每名学生可以学习多门课程，每门课程可以有任意人数的学生选听。
  - 每门课程由一或多位授课教师讲授，每位教师可以不教课程或教多门课程。
  - 教师可以在一或多个系任职，但最多担任一个系的系主任。
  - 学校必须包括至少一个系，每个学校通常招收多名学生。
  - 每个系可开设一或多门课程，每门课可由一或多个系开设。

# Class Diagram and Object Diagram

---

- ✧ A class diagram gives general, definitional information (the properties of a class and its attributes) as well as other classes it associates with.
- ✧ An object diagram gives information about **specific instances of a class** and how they link up at **specific instants in time**.



---

# State Machine Diagram

# State Machine Diagram

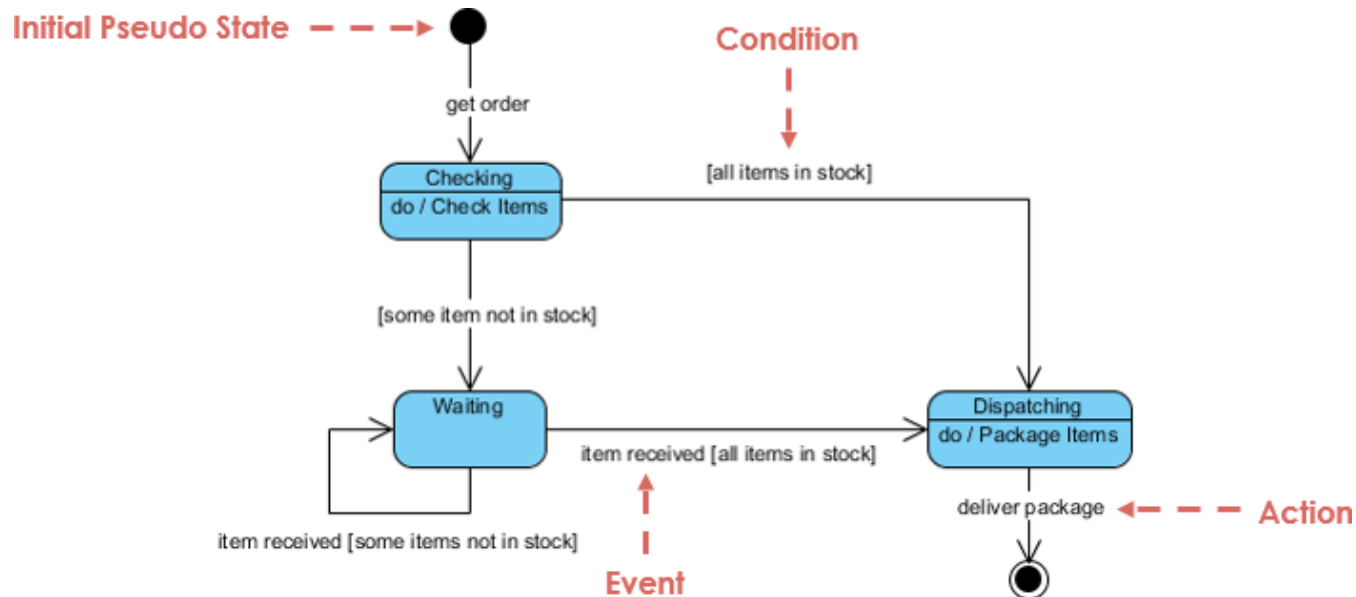
---

- ✧ Objects in a system change their states in response to **events and time**. The state machine diagram captures these state changes.
  - An object responds differently to the same event depending on what state it is in.
- ✧ State machine diagrams are usually applied to objects but can be applied to any element that has behavior to other entities such as: actors, use cases, methods, subsystem, etc.

# Key Parts in State Machine Diagram

✧ A state machine diagram is a graph consisting of:

- States (simple states or composite states)
- Transitions connecting the states



# States

---

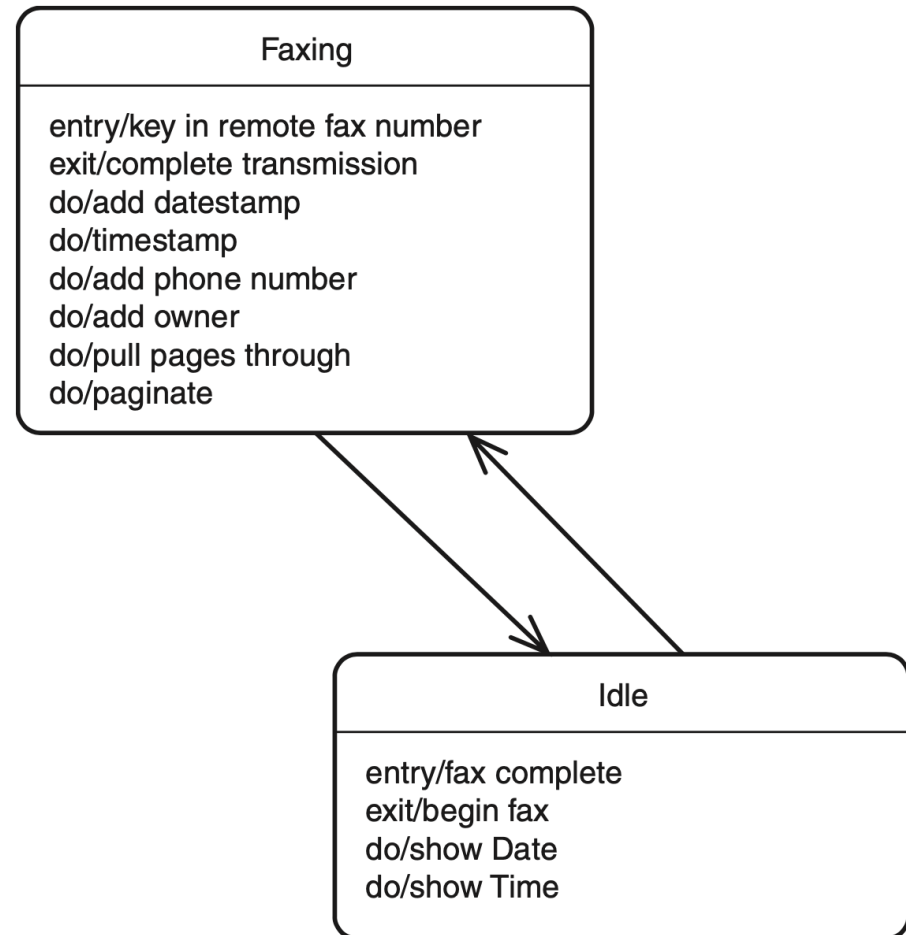
- ✧ All objects have “state”
  - The object either exists or it doesn't
  - If it exists, then it has a value for each of its attributes
  - Each possible assignment of values to attributes is a “state”
- ✧ The state space of most objects is enormous
  - State space size is the product of the range of each attribute
- ✧ Only part of that state space is “interesting”
  - Some states are not reachable
  - Integer and real values usually only vary within some relevant range
  - Often, we're only interested in certain thresholds



# States

---

- ✧ A state icon can be subdivided into areas that show the state's **name and activities**.
- ✧ Three frequently used categories of activities are:
  - entry: what happens when the system enters the state
  - exit: what happens when the system leaves the state
  - do: what happens while the system is in the state



# Transitions and Events

---

## ✧ Transitions

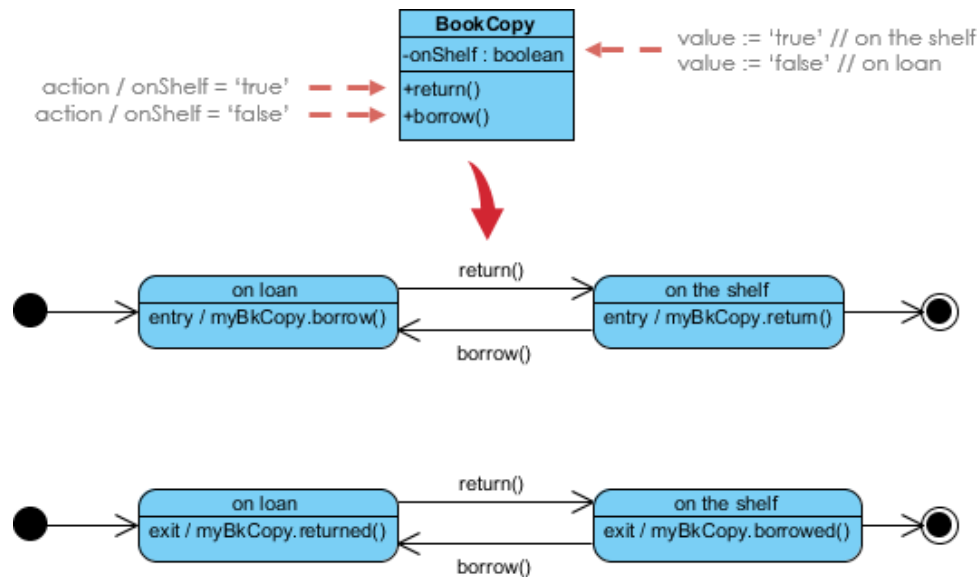
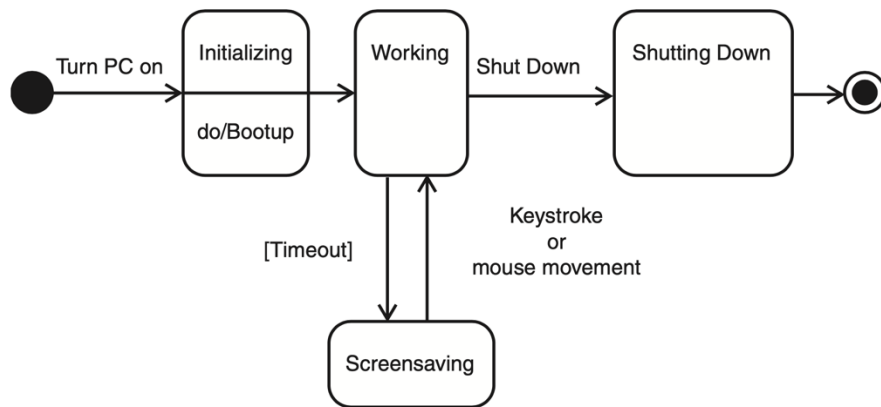
- Are enabled when the state is “on”; disabled otherwise
- Every transition has an event that acts as a trigger
- A transition may also have a condition (or guard)
- A transition may also cause some action to be taken
- When a transition is enabled, **it can fire if the trigger event occurs and its guard is true**
- Syntax: **event [guard] / action**

## ✧ Events

- Occurrence of stimuli that can trigger an object to change its state
- Determine when transitions can fire

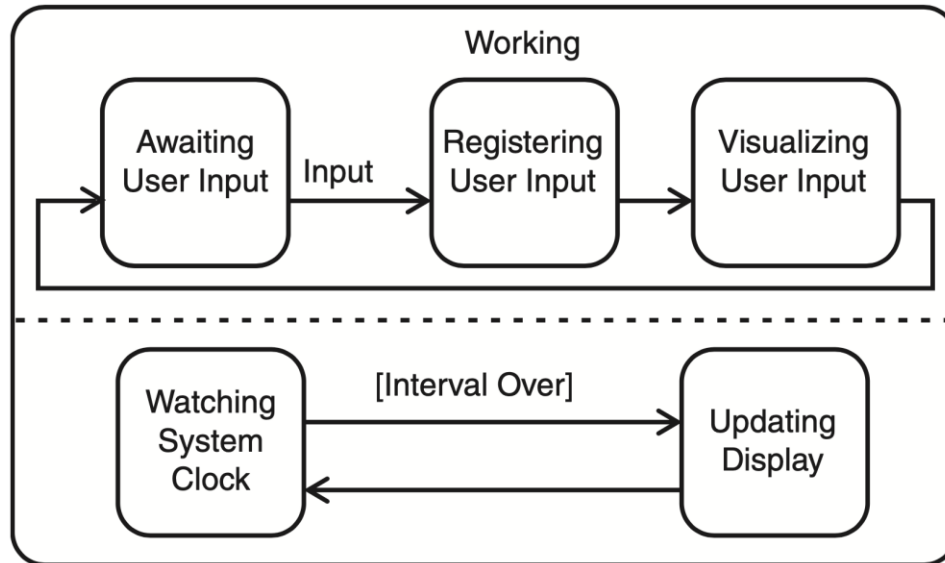
# Transitions and Events

- ✧ Transition lines depict the movement from one state to another.
- ✧ Each transition line is labeled with the event that causes the transition.



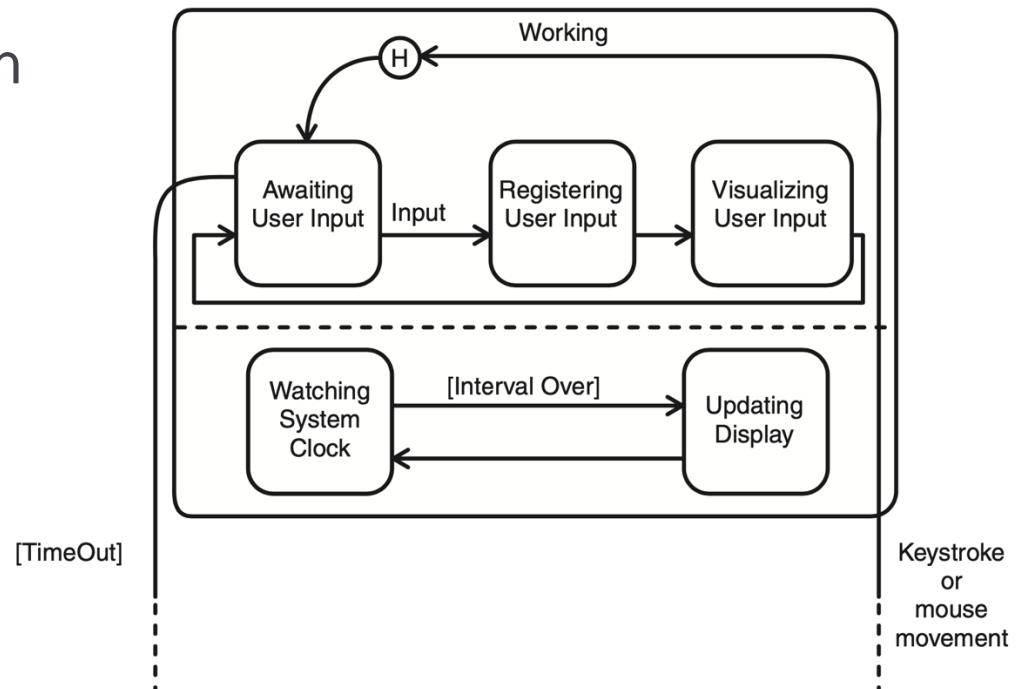
# Substates

- ✧ A state which has substates (nested states) is called a composite state. **Substates may be nested to any level.**
- ✧ Substates may either be sequential (occurring one after the other) or concurrent (occurring at the same time).



# History States

- ✧ Unless otherwise specified, when a transition enters a composite state, the action of the nested state machine starts over again at the initial state (unless the transition targets a substate directly).
- ✧ History states allow the state machine to **re-enter the last substate** that was active prior to leaving the composite state.



# Checking Your State Machine Diagrams

---

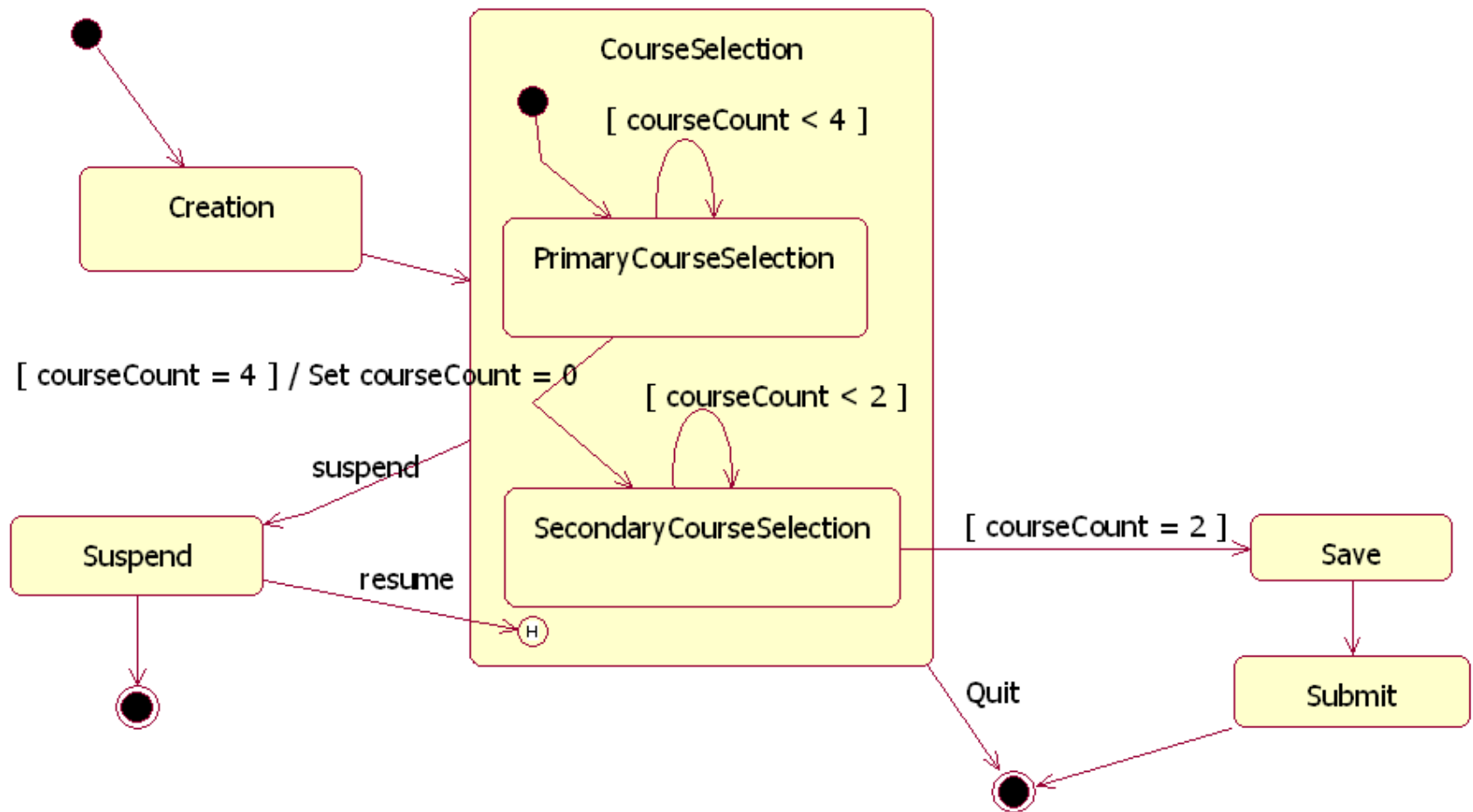
## ✧ Style guidelines:

- Give each state a unique, meaningful name
- Only use composite states when the state behavior is genuinely complex
- Do not show too much detail on a single state machine diagram
- Use guard conditions carefully to ensure your state machine diagram is unambiguous

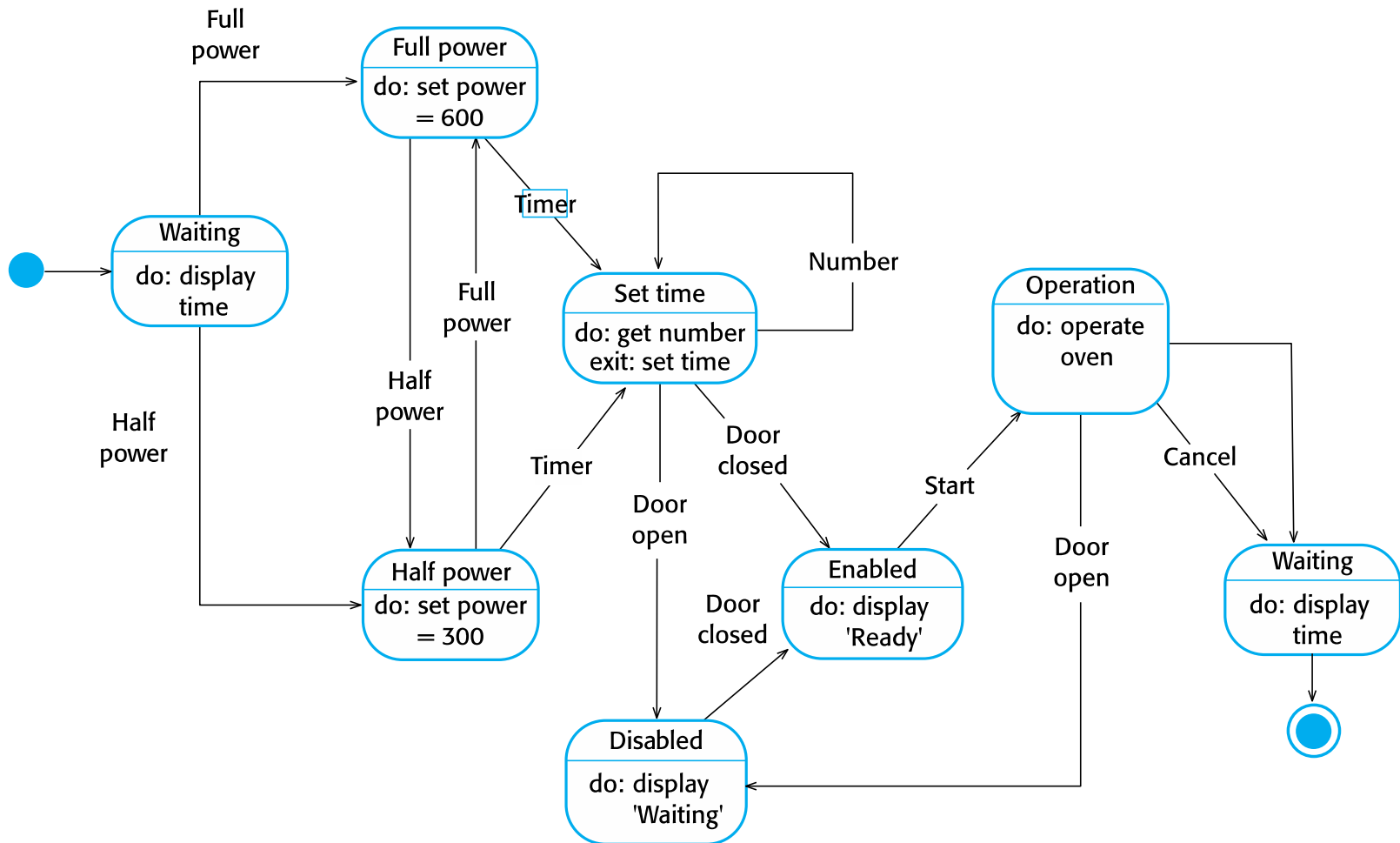
## ✧ You probably should not use state machine diagrams if:

- You find that most transitions are fired “when the state completes”
- Many of the trigger events are sent from the object to itself

# Example 1: course selection

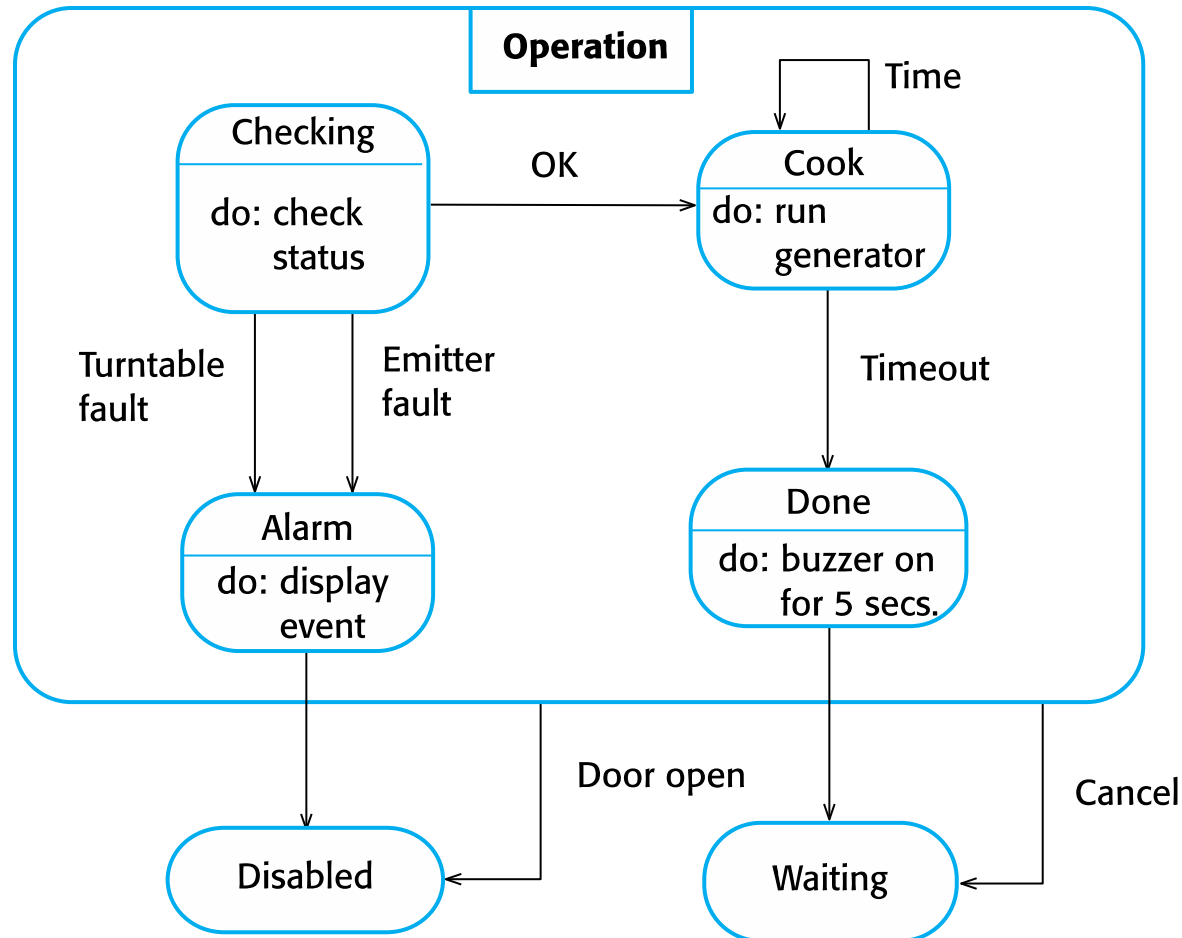


# Example 2: microwave oven





# Example 2: microwave oven



## Example 2: microwave oven

---

State	Description
Waiting	The oven is waiting for input. The display shows the current time.
Half power	The oven power is set to 300 watts. The display shows 'Half power'.
Full power	The oven power is set to 600 watts. The display shows 'Full power'.
Set time	The cooking time is set to the user's input value. The display shows the cooking time selected and is updated as the time is set.
Disabled	Oven operation is disabled for safety. Interior oven light is on. Display shows 'Not ready'.
Enabled	Oven operation is enabled. Interior oven light is off. Display shows 'Ready to cook'.
Operation	Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for five seconds. Oven light is on. Display shows 'Cooking complete' while buzzer is sounding.

# Example 2: microwave oven

---

Stimulus	Description
Half power	The user has pressed the half-power button.
Full power	The user has pressed the full-power button.
Timer	The user has pressed one of the timer buttons.
Number	The user has pressed a numeric key.
Door open	The oven door switch is not closed.
Door closed	The oven door switch is closed.
Start	The user has pressed the Start button.
Cancel	The user has pressed the Cancel button.

# Behavioral Models

---

- ✧ Behavioral models are models of the dynamic behavior of a system as it is executing. They show what happens or what is supposed to happen **when a system responds to a stimulus from its environment.**
- ✧ Data-driven modeling
  - Some data arrives that has to be processed by the system.
  - Modeling with activity diagrams and sequence diagrams.
- ✧ Event-driven modeling
  - Some event happens that triggers system processing.
  - Modeling with state machine diagrams.

---

# Model-Driven Engineering

# Model-Driven Engineering

---

- ✧ Model-driven engineering (MDE) is an approach to software development where **models rather than programs are the principal outputs of the development process.**
- ✧ The programs that execute on a hardware/software platform are then generated automatically from the models.
- ✧ Proponents of MDE argue that this raises the level of abstraction in software engineering so that engineers no longer have to be concerned with programming language details or the specifics of execution platforms.

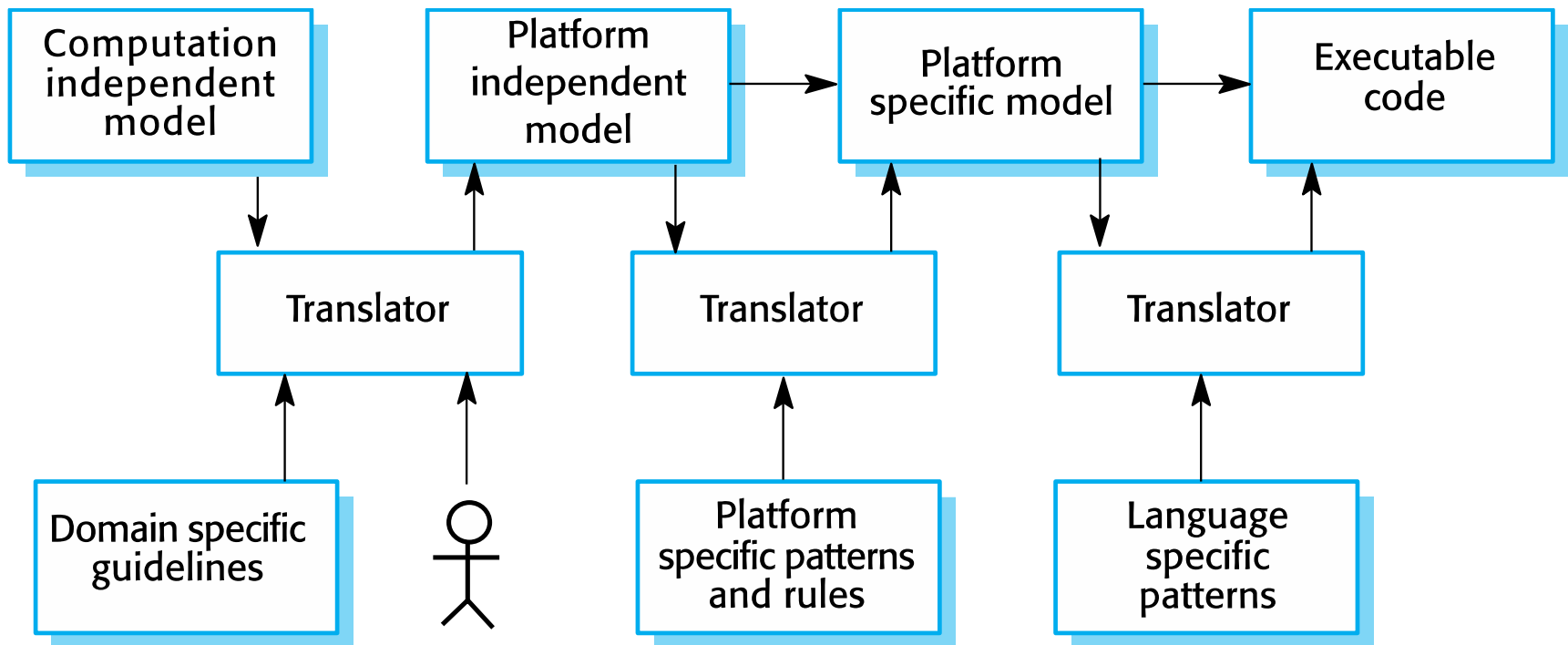
# Types of Model

---

- ✧ A computation independent model (CIM)
  - These model the important domain abstractions used in a system. CIMs are sometimes called domain models.
- ✧ A platform independent model (PIM)
  - These model the operation of the system without reference to its implementation. The PIM is usually described using UML models that show the static system structure and how it responds to external and internal events.
- ✧ Platform specific models (PSM)
  - These are transformations of the platform-independent model with a separate PSM for each application platform. In principle, there may be layers of PSM, with each layer adding some platform-specific detail.

# Automated Model Transformation

---





# Usage of Model-Driven Engineering

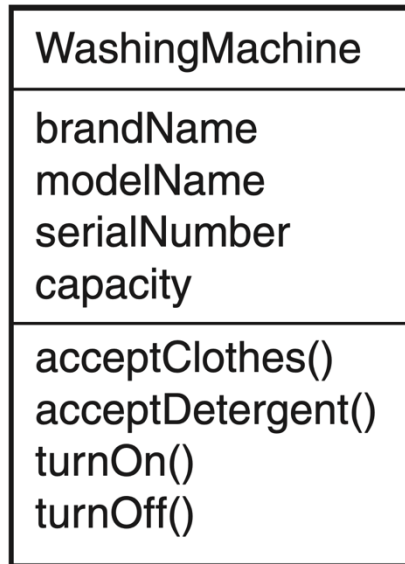
---

- ✧ Model-driven engineering is still at an early stage of development, and it is unclear whether or not it will have a significant effect on software engineering practice.
- ✧ Pros
  - Allows systems to be considered at higher levels of abstraction.
  - Generating code automatically means that it is cheaper to adapt systems to new platforms.
- ✧ Cons
  - Models for abstraction are not necessarily right for implementation.
  - Savings from generating code may be outweighed by the costs of developing translators for new platforms.

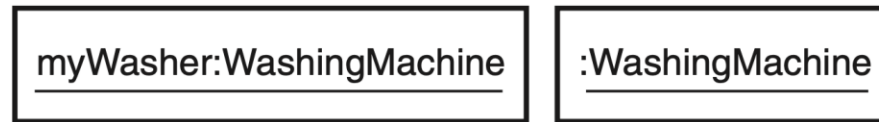
---

# Summary

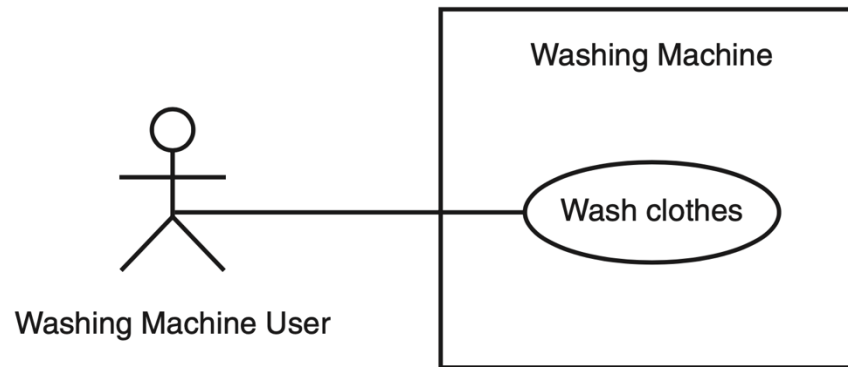
# System modeling



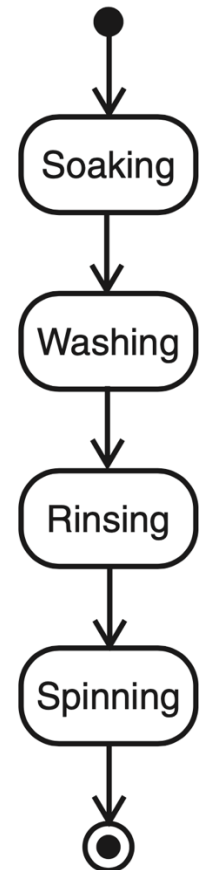
The UML class diagram



The UML object diagram

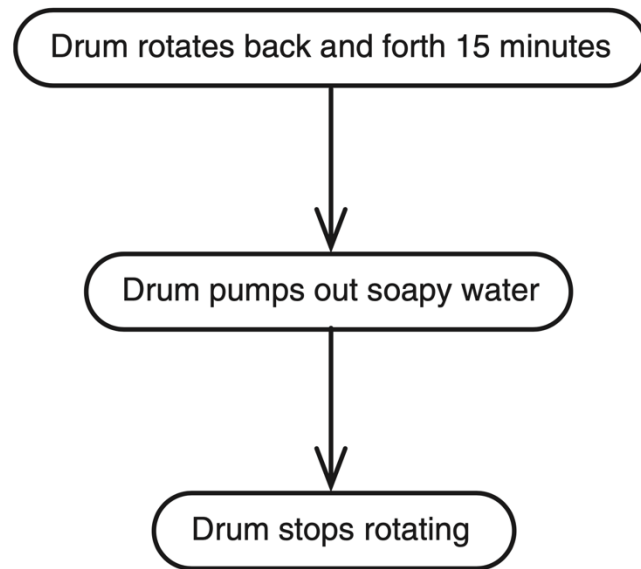


The UML use case diagram

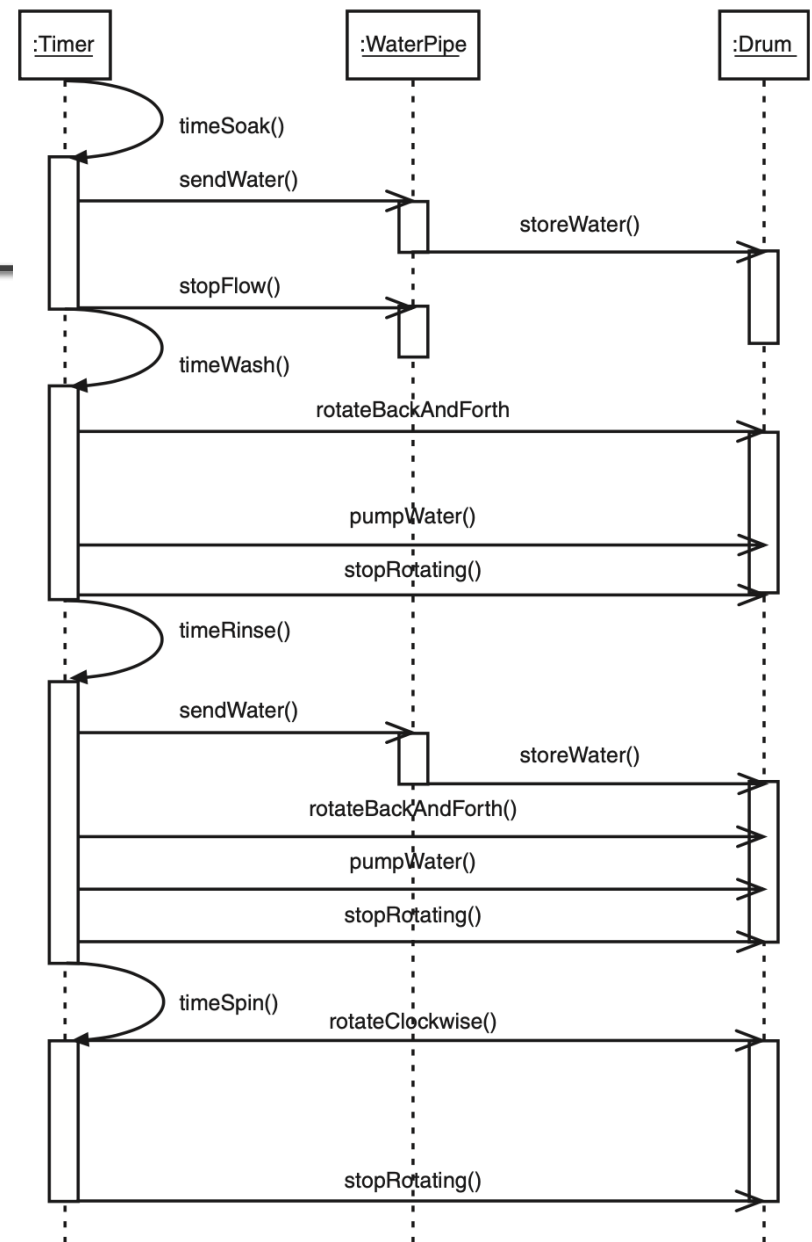


The UML state machine diagram

# System modeling



The UML activity diagram



The UML sequence diagram