SFWRENG / MECHTRON 3K04
Software Development

This Week: Model-Driven Engineering

October 27th, 2025

Dr. Angela Zavaleta Bernuy

McMaster University | Engineering
Computing & Software

Brighter World

Announcements

Announcements

- Midterm marking is happening
- Demo week!
- On Wednesday, we have a guest lecturer
- Next Monday, we have observers

Let’s Review the Midterm ☺

LAQ1

LAQ1. (8 pt)

Angela is building a safety-critical greenhouse in her backyard, so she can grow one of her favourite fruits this winter: Moscato grapes. She has recently installed some sensors and devices to automate the greenhouse:

- A humidity sensor and misting systems to regulate air humidity
- A thermostat and ventilation to regulate a suitable temperature, and
- A door switch and keypad lock to secure the greenhouse at night from raccoons

To grow properly, Moscato grapes need the humidity to stay between 60%−70% and the temperature between 22°C−26°C. At night, the door should automatically lock, and Angela can unlock it in the morning using a keypad.

Complete the following table with the monitored and controlled variable names for the greenhouse system, indicate its type (monitored or controlled), time nature (continuous or discrete), and a one sentence explanation of the variable role. Use the 4-variable model notation.

Variable Name	Type	Time Nature	Explanation

LAQ2

LAQ2. (4 pt)

Angela wants to make sure her Moscato grapes stay safe, so she wants to write some formal specifications to the engineers working on her system. These engineers will work on the humidity control feature of the system, that is, how the program will decide what action to take based on the current humidity level. The system has three possible actions:

- **MIST**: increase humidity
- **IDLE**: keep conditions unchanged
- **VENT**: decrease humidity

The policy is as follows:

- If humidity is below 60%, the system should **MIST**.
- If humidity is between 60% and 70%, it should remain **IDLE**.
- If humidity is above 70%, it should **VENT**.

Write a complete program specification for this behaviour by providing:

(a) A precondition describing the acceptable input range for humidity (i_1).

(b) A postcondition specifying the required output (o) for each case. Use logical notations where appropriate.

LAQ3

LAQ3. (3 pt)

While Angela has her requirements written in formal specification formats, she wants to change some into semi-formal specifications using Gherkin. You have already seen the full specification that defines how the system should **MIST**, **IDLE**, or **VENT** depending on the humidity level. Now, focus only on the venting behaviour (when the humidity is above 70%).

Write a **Gherkin specification** that describes what the system should do in this situation.

SFWRENG / MECHTRON 3K04
Software Development

This Week: Model-Driven Engineering

October 27th, 2025

Dr. Angela Zavaleta Bernuy

McMaster University | Engineering
Computing & Software

Brighter World

Model-Driven Engineering (MDE)

- The idea of model-driven software development originated many years ago (used to be called *case tools*)
- However, only relatively recently has it become a reality
- Models have been in use for decades in engineering and science, so this is a natural progression
  - We have models for molecules, chemical compounds, etc.
  - Now, we have various ways to model software.

Model-Driven Engineering (MDE)

- It goes by other names such as Model-Based Design
- Main idea:
  - Models of the system you want to build, and its environments
  - Transformation of system models through various phases, until we can generate implementations (mechanical/electrical artifacts, code, or other logic implementation)
- Why?
  - Modelling is a form of abstraction:
    - It lets us get rid of some lower-level details in order to get a better understanding of the system
  - Impossible for anyone to understand every line of code in more complex systems:
    - Abstractions are necessary for development and documentation

Model-Driven Engineering (MDE): Benefits

- Many types of **analyses** can be performed on the models **to validate** the initial model
- **Correctness-by-construction** all the way through to implementation
  - Can also generate “proofs” or at least, checks, that correctness-by-construction is achieved
- Can construct **tools to help examine** the models
- Can **regenerate implementations** when changes are introduced
- **Low skill floor** and **high skill ceiling**
  - Easy to get started and allows for high amount of expression
- Facilitates **general reasoning and thinking** about software engineering and development

Correctness-by-Construction (CbC)

- Build programs incrementally based on a formal specification from the start
- Follows a specification-first, refinement-based approach
  - The specification is defined first. i.e. precondition and postcondition
  - Then, the program is successively created using a small set of refinement rules
    - For instance, let's say we are creating a program to find the larger value of an array
    - We first may start with an empty list and create refinement rules for that. Then, we we can define iteration rules.
    - We also need to verify that the loop maintains its properties (keeping the largest value).
  - These rules define side conditions preserving the correctness of the program
- Advantage: Errors are likely to be detected earlier in the design process and can be tracked more easily

Model-Driven Engineering (MDE): Cons

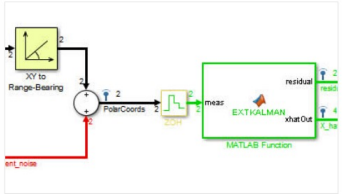
- **Not a lot of tool support** compared to traditional forms of software engineering
- **Difficult** to implement **at scale**:
  - Requires common understanding across practitioners
- Loses benefits with more **simple systems**
  - Can be viewed as redundant by some
- Generally, **developers want to code not model**
  - … until it is too late

Model-Driven Engineering (MDE): Tool Suites

- Most common one in North America and world-wide for automotive and avionics is Matlab/Simulink, developed by The MathWorks
- Others include IBM Rational, and mainly in Europe, SCADE, and ANSYS
- Unified Modeling Language (UML) and its many profiles are used for design and documentation of software
- **Note:** Simulink is really a control theory design environment.
  - The constructs in the language are focused on design of control system rather than requirements
  - It is important to realize and remember this!
- **Modularity:** Up until recently, Simulink designs did not use software engineering principles to achieve effective modularity in the design.
  - In particular, we would be better off if we could design Simulink models so that they are more robust with respect to future changes
    - Recall our discussion on software design, in terms of “information hiding”

Simulink

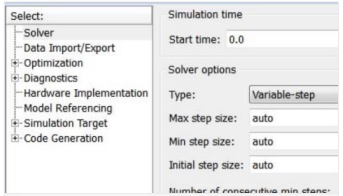
Capabilities



Building the Model

Model hierarchical subsystems with predefined library blocks.

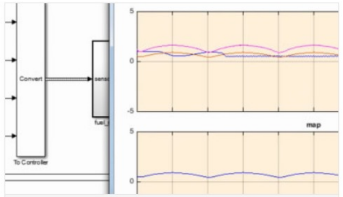
» Learn more



Simulating the Model

Simulate the dynamic behavior of your system and view results as the simulation runs.

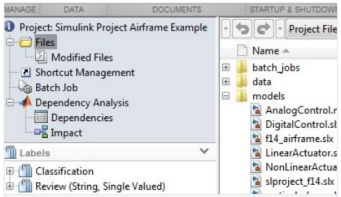
» Learn more



Analyzing Simulation Results

View simulation results and debug the simulation.

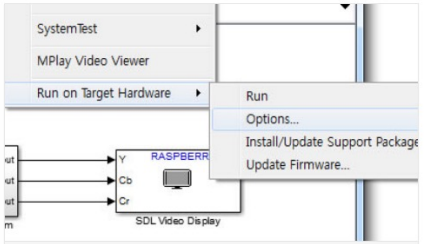
» Learn more



Managing Projects

Develop and share applications as code, executables, or software components.

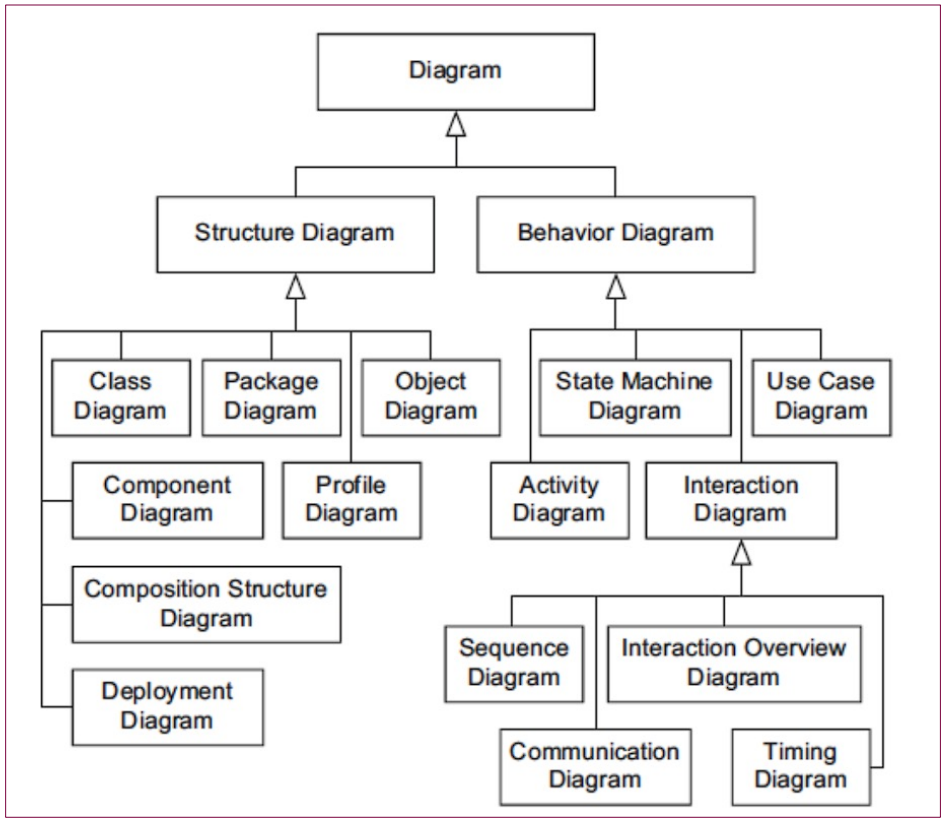
» Learn more



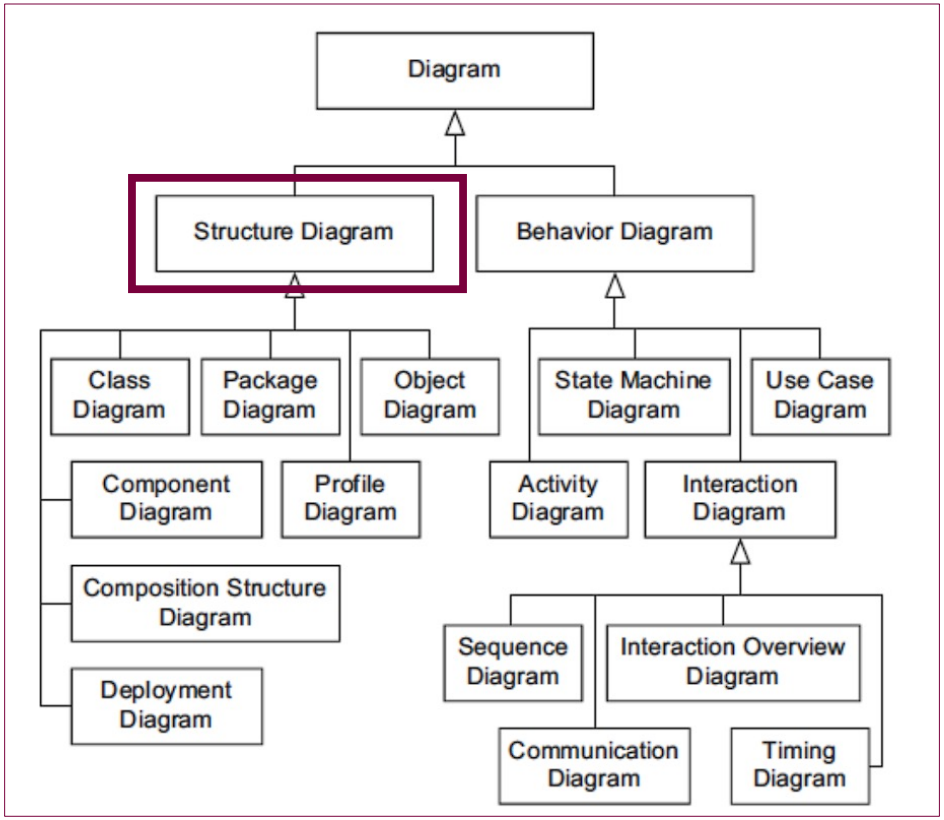
Connecting to Hardware

Connect your model to hardware for real-time testing and embedded system deployment.

Unified Modeling Language (UML)

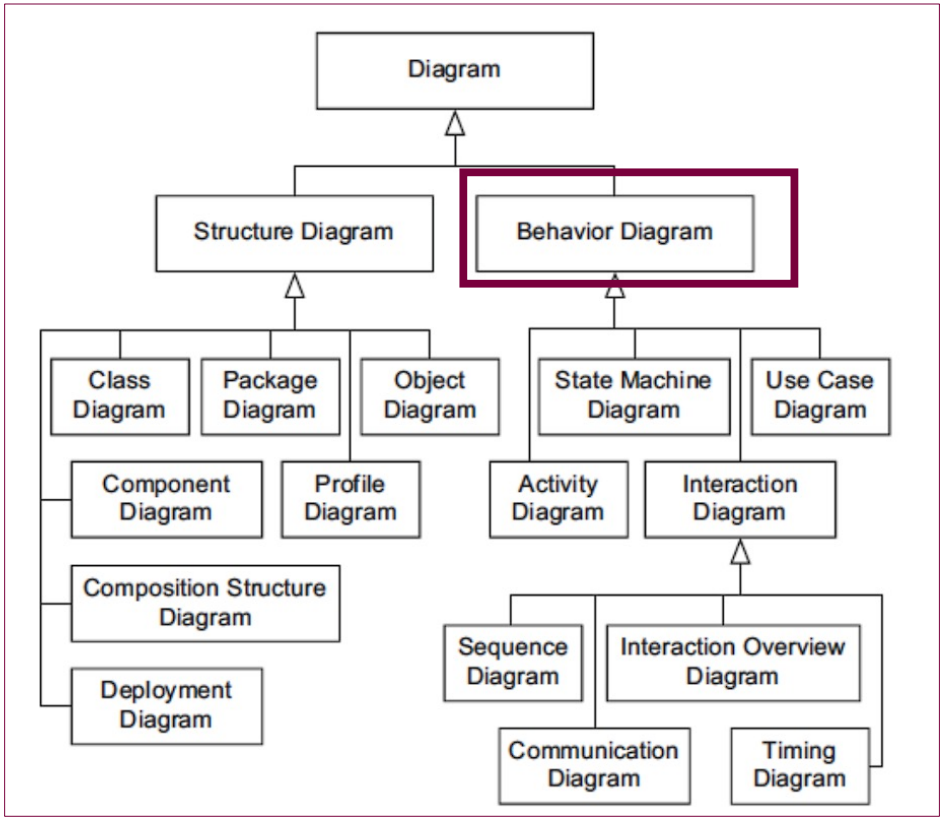


UML: Structure Diagram



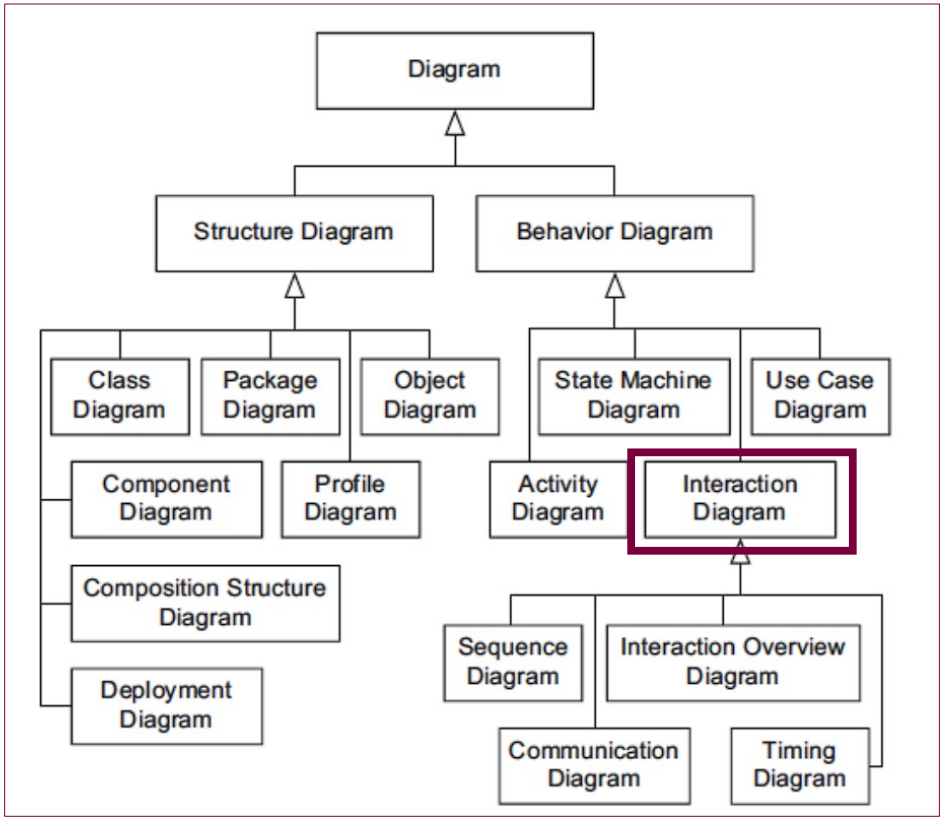
- Used primarily to determine how a system is glued together
- Often used to model system architectures, requirements, and defining domain specific languages (DSL)
- Best for defining modules/components and relationships between modules/components
- *Just like the blueprints of a building: they tell us what parts exist and how they fit together*

UML: Behaviour Diagram



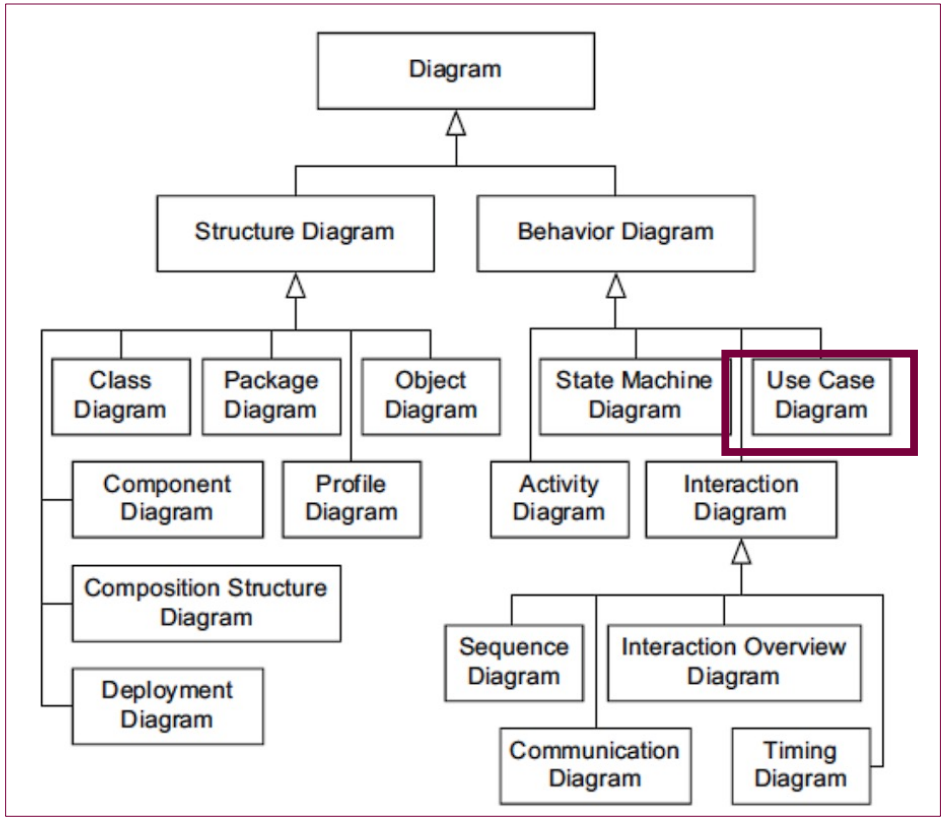
- Used primarily to describe what a system does, sometimes how it does it
- Often used for defining requirements, designs, and implementations
- Best for control theory and system development
- *If structure diagrams are the static blueprints, behaviour diagrams are the animations, they show how the system moves, reacts and communicates*

UML: Interaction Diagram

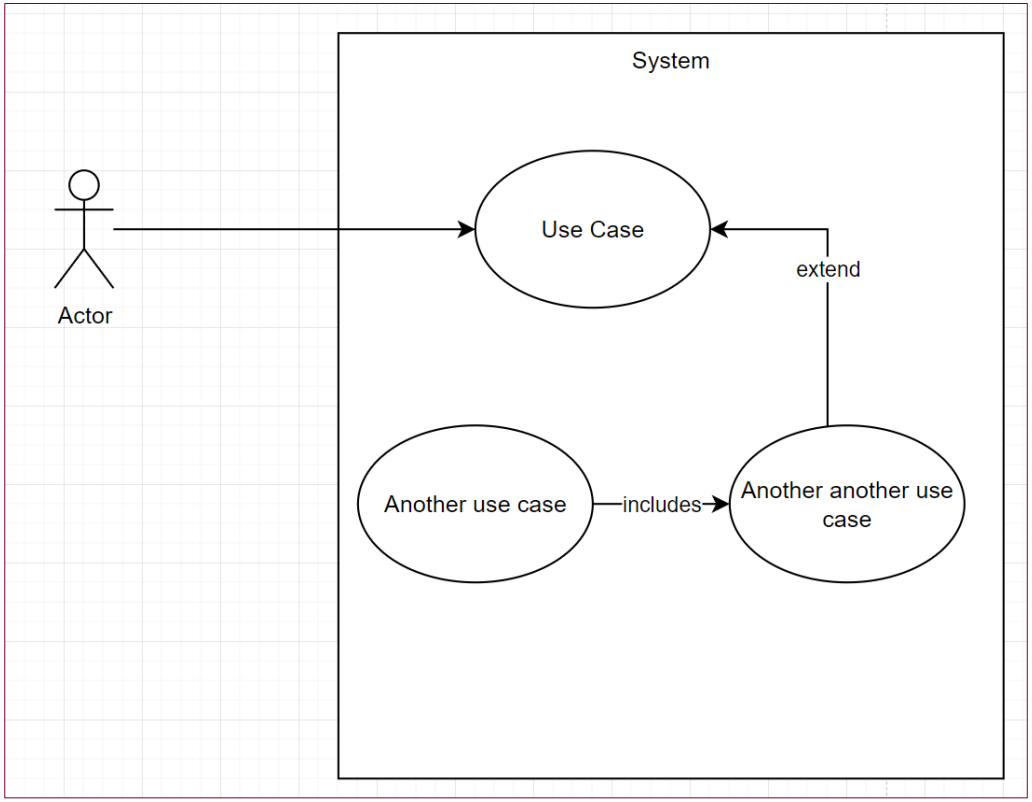


- A subset of Behaviour Diagrams specialized in capturing behavior between objects
- More niche in MDE: if more details are needed about specific behaviors that is when you might use interaction diagrams
- *Interaction diagrams zoom in on communication: they tell the story of how parts of the system exchange messages to make behavior happen. In MDE, these diagrams often feed into simulation and code generation steps.*

UML: Use Case Diagram



UML: Use Case Diagram

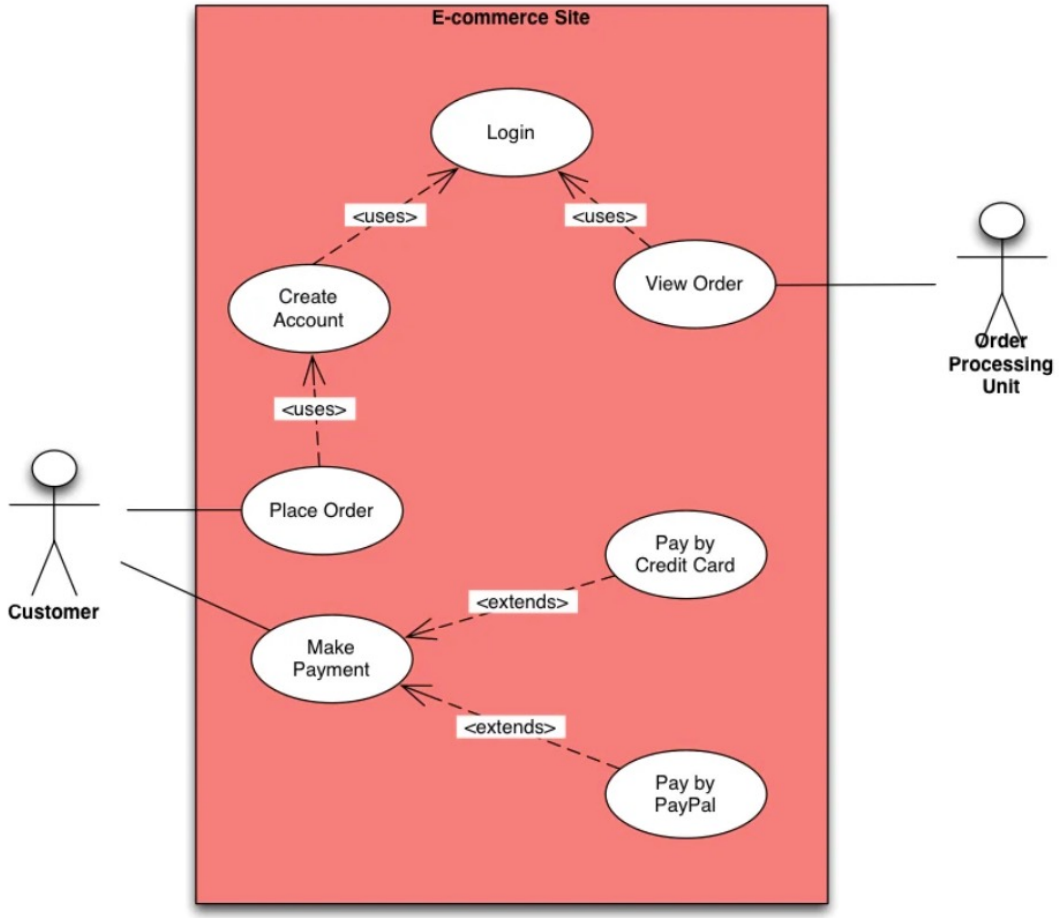


- 3 main components: Actors, System, Use Case
- Most often used for requirements and concepts
- Easiest form of modelling with the least rules
- Answers the question who will use our system, what will they use in the system, and sometimes how will they use the system
- **Includes:** this relations highlights when one use case includes another. The source use case and exist independently of the target.
- **Extends:** this relationship highlight when one use case extends another. This is the opposite of includes. The source use case cannot exist independently of the target use case.

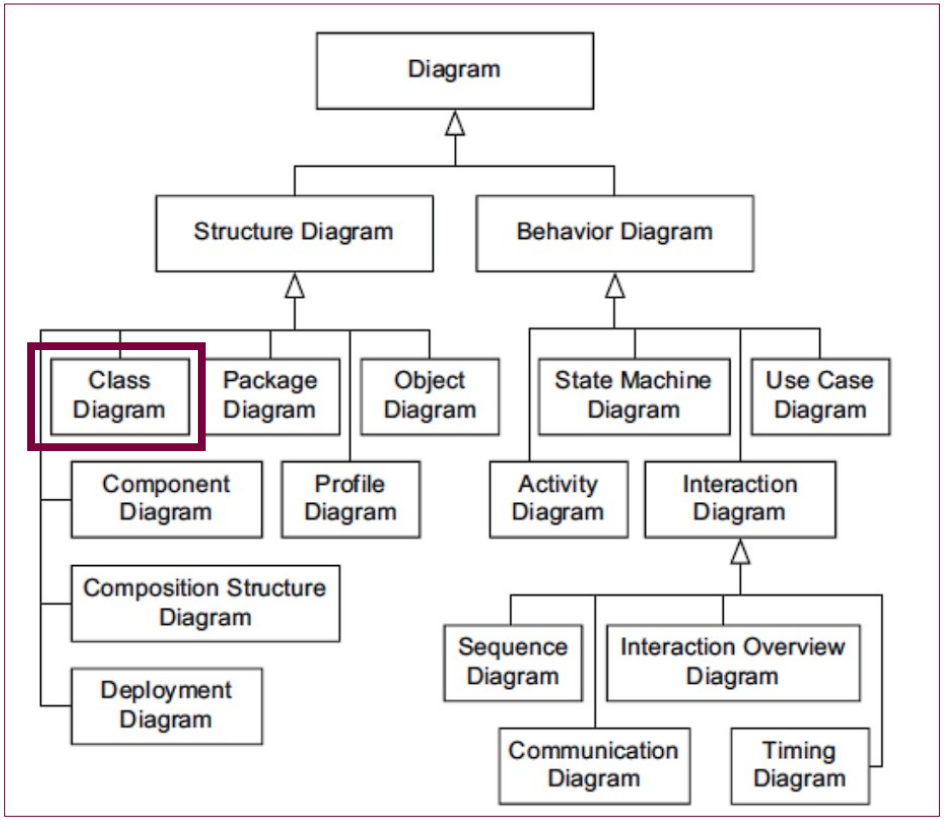
UML: Use Case Diagram – Exercises

In this example, *uses* is just equivalent to *includes*.

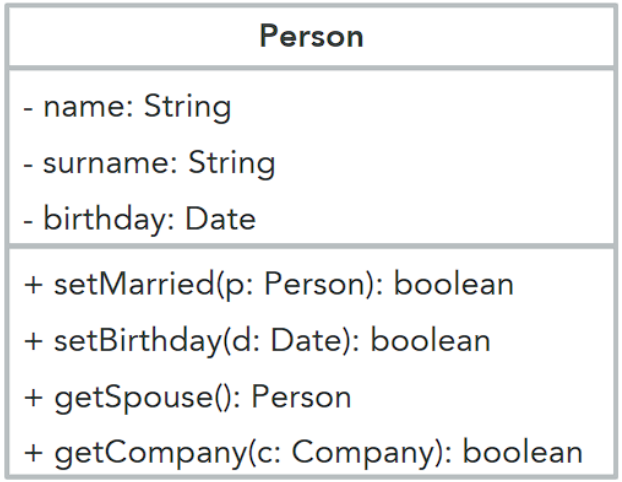
With the person next to you: **Identify** the actors, systems and use cases



UML: Class Diagram

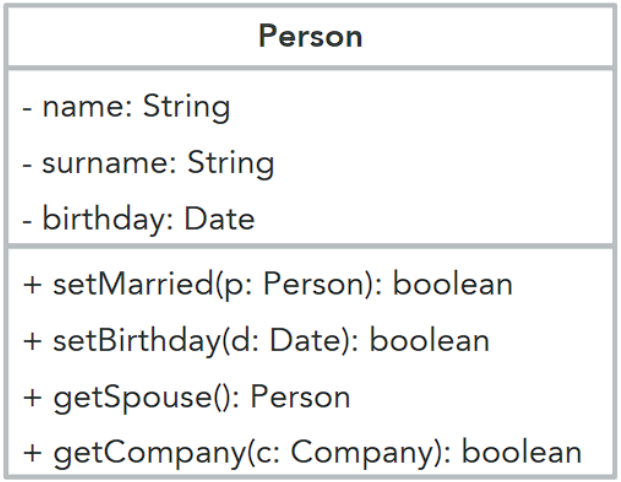


- Class diagram depicts the types of objects in a system and the static relationships between them.
- A class is generally made of 2 parts:
  - Name
  - Properties



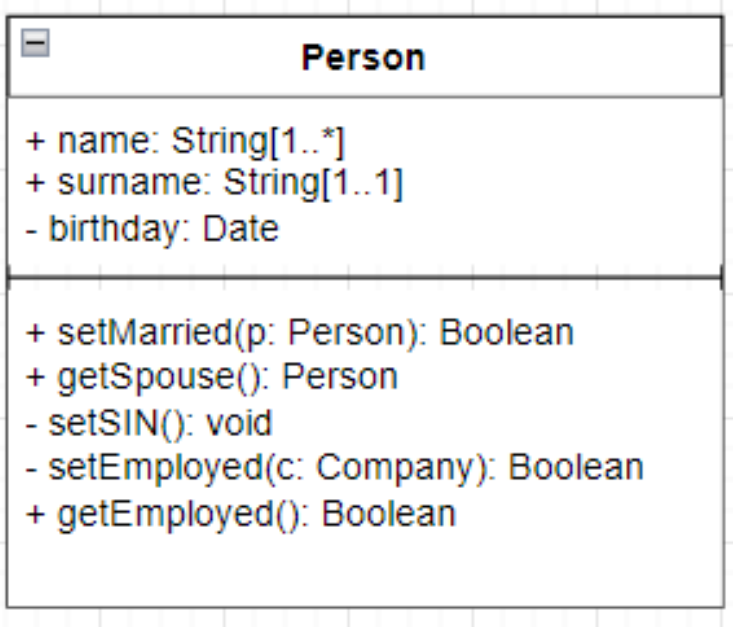
UML: Class Diagram

- Properties can be though of as fields in the class
  - Attributes
  - Methods
- Properties contain information such as private/public, multiplicities, return types and sometimes more.



UML: Class Diagram

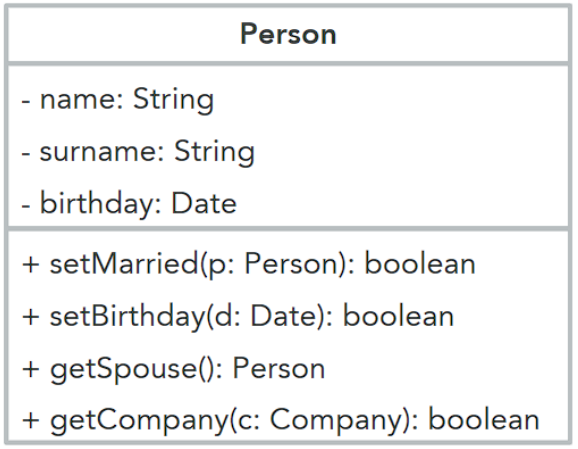
- Visibility:
  - + public: can be accessed by any other class
  - - private: access only within the object permitted
  - # protected: access by objects within the same class and subclasses permitted
  - ~ friendly: access by objects within the same package
- Attribute:
  - Visibility name: type [multiplicity]
- Method:
  - Visibility name (parameters): return type



UML: Class Diagram

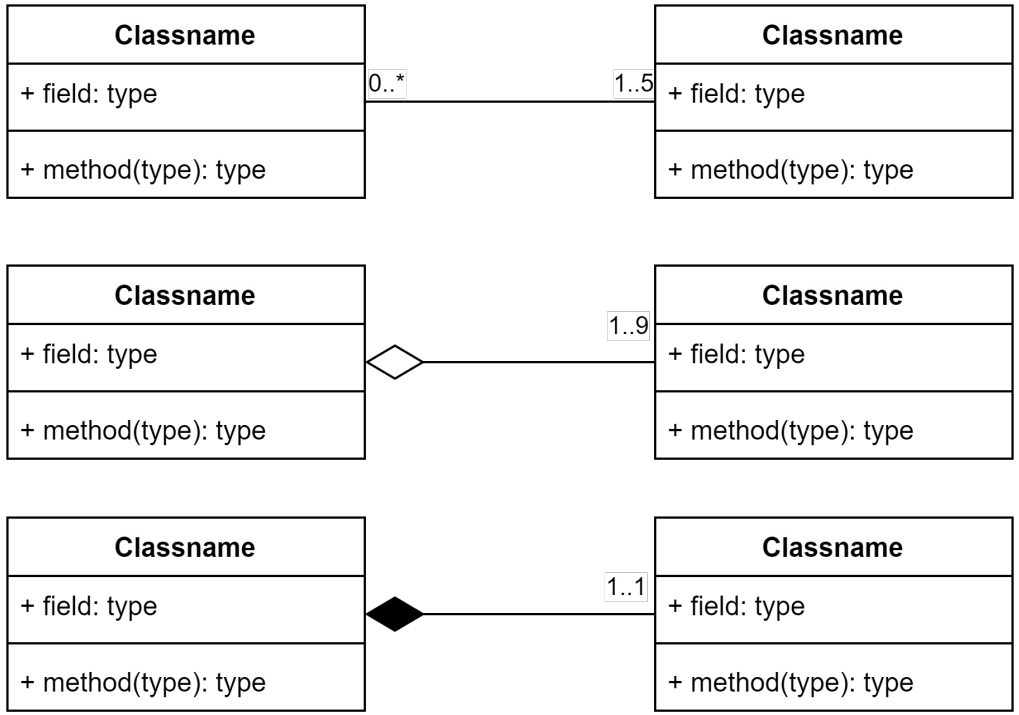
```
public class Person {
    private String name;
    private String surname;
    private Date birthday;

    public boolean getMarried (Person p) {...}
    public boolean setBirthday (Date d) {...}
}
```



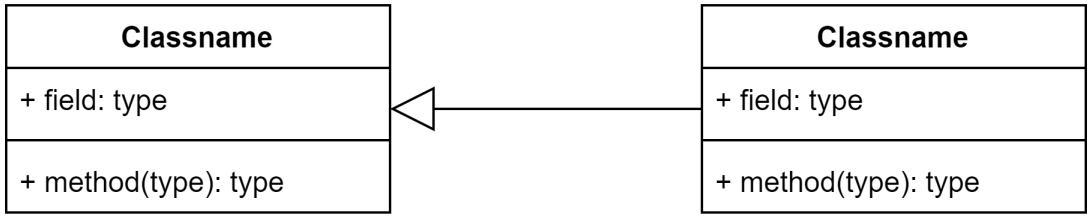
UML: Class Diagram – 5 Types of References

- Association
  - "*knows about*" / "*uses*"
- Aggregation
  - Special type of association
  - "*part of*" / "*has a*"
- Composition
  - "ownership"
  - If the class on the left is deleted, the class on the right is also deleted



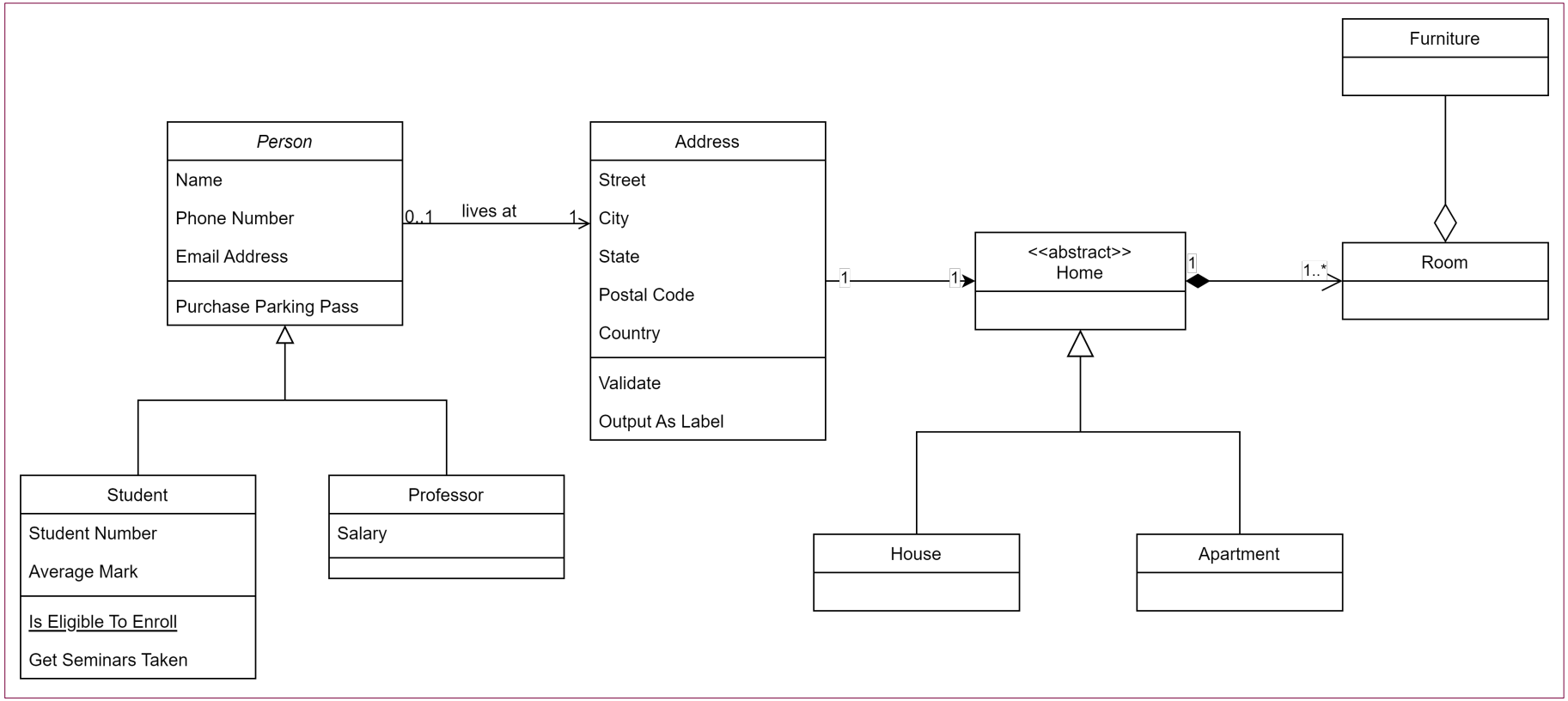
UML: Class Diagram – 5 Types of References

- Inheritance
  - "*is a*"



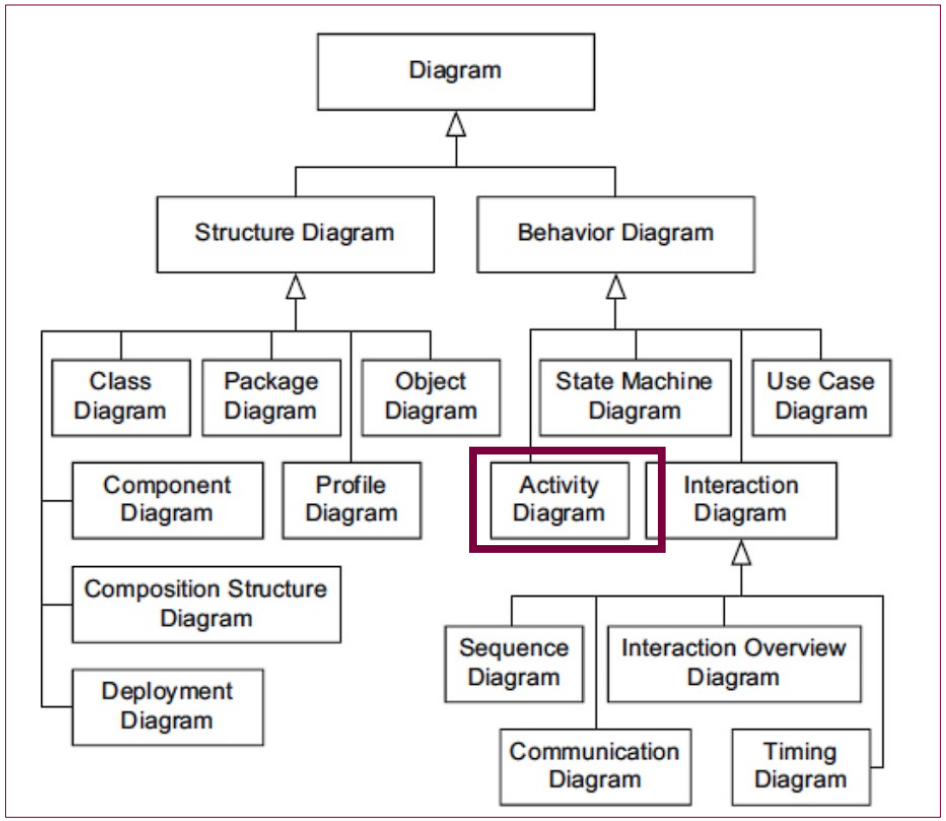
- Interface
  - "*uses*" / "*implements*"

UML: Class Diagram - Exercise

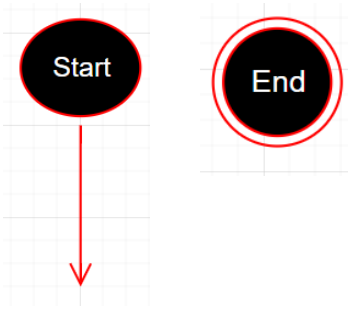


With the person next to you: **Identify** the relationships.

UML: Activity Diagrams



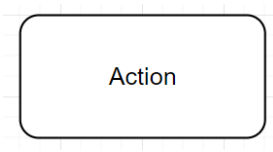
- Models the flow of control or workflow in a system or process
- Composed of a starting point and end point



- 5 main components

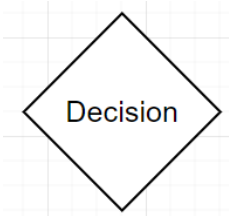
UML: Activity Diagrams – 5 Main Components

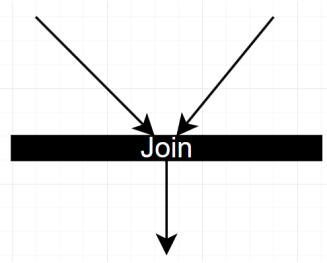
- Actions
  - Individual steps or operations



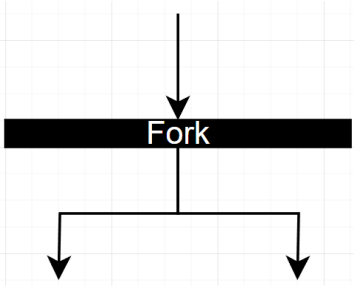
- Joins
  - Opposite of forks
  - Multiple flows merge back into one

- Decisions
  - Branching points where the flow splits based on a condition



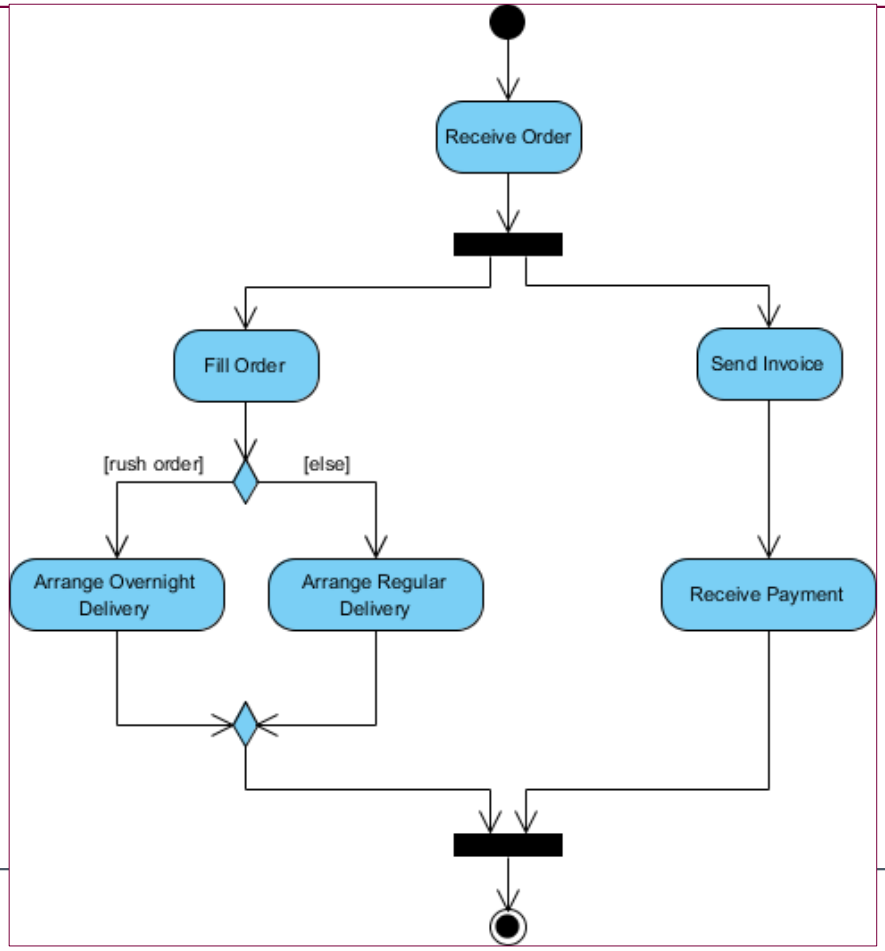


- Forks
  - Parallel execution: a single flow splits into multiple simultaneous paths

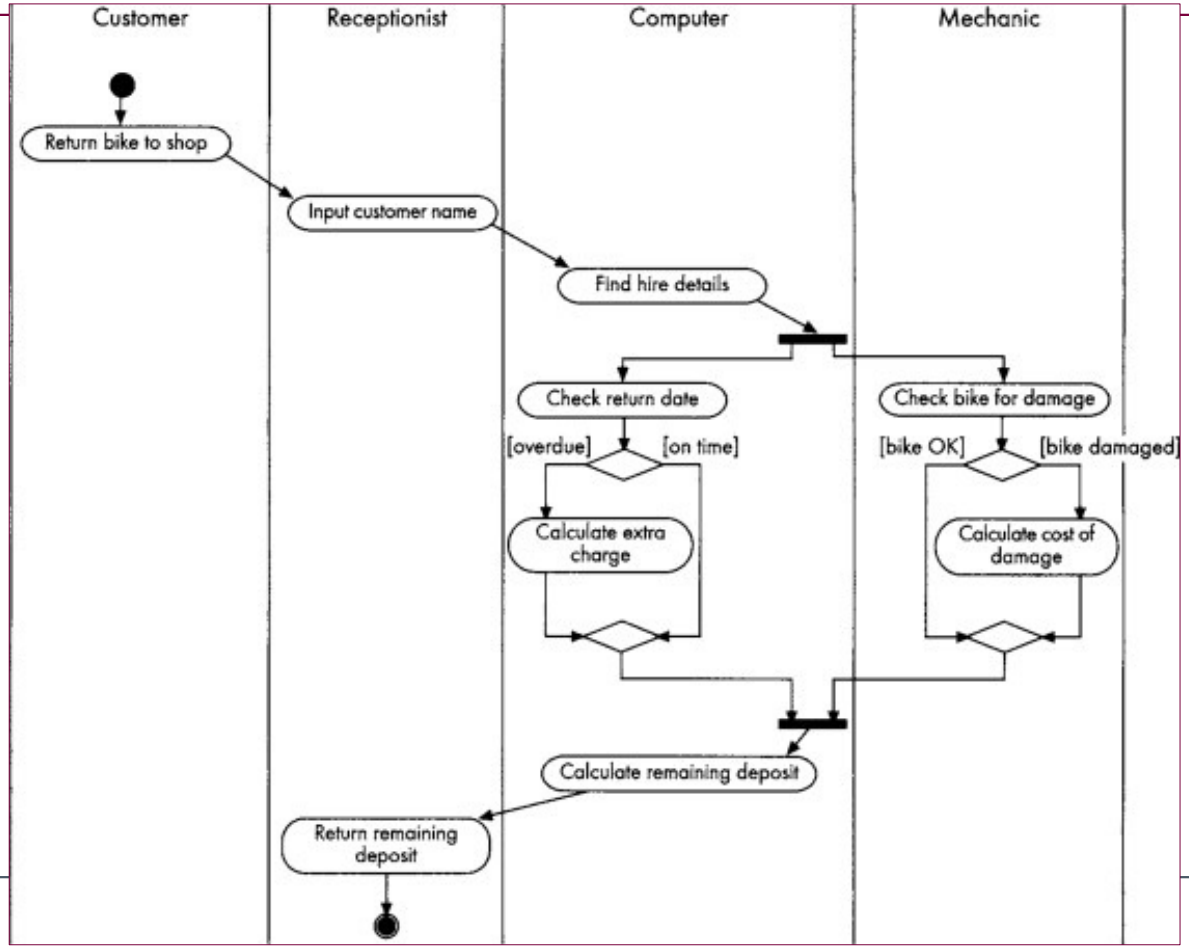


- Edges
  - Arrows that connect actions, decisions, forks, and joins

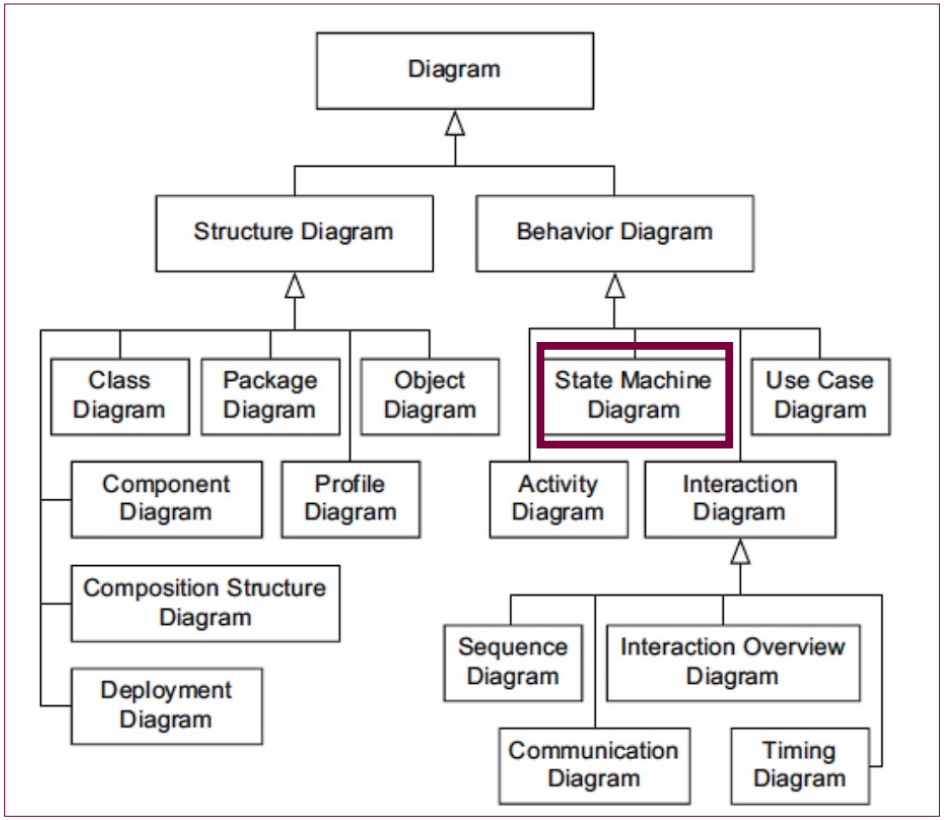
UML: Activity Diagram - Example



UML: Activity Diagram - Example



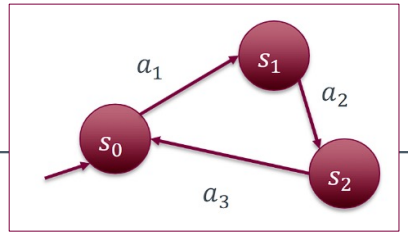
UML: State Machine Diagrams



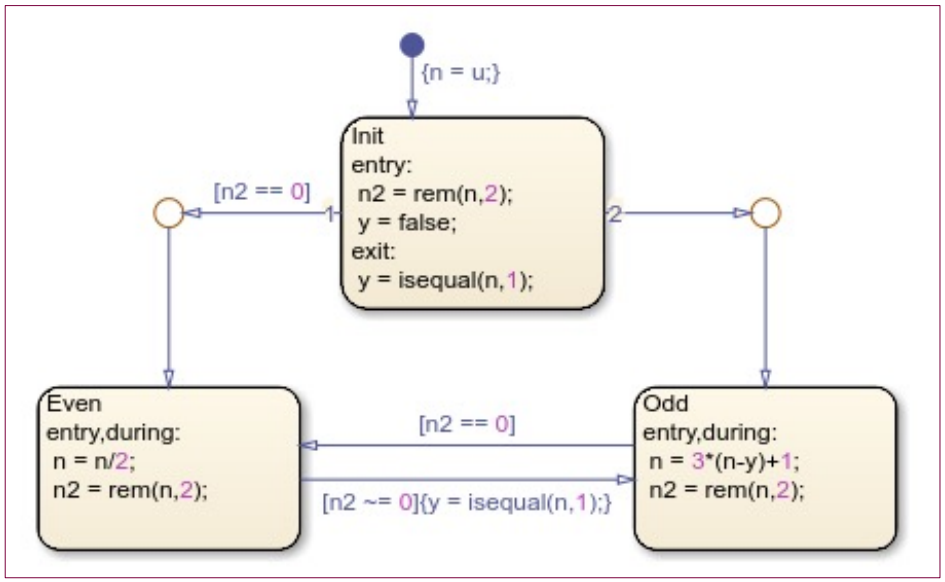
- Models how an object or system changes over time.
- Also called Finite State Machines (FSMs)
- Typically composed of a tuple <S, A, T>
  - S: a finite set of States
  - A: a finite set of Actions
  - T: a transition relations where

$$T \subseteq S \times A \times S \text{ or}$$
$$T : S \times A \rightarrow S$$

- Typically, it will have one initial state where everything starts.

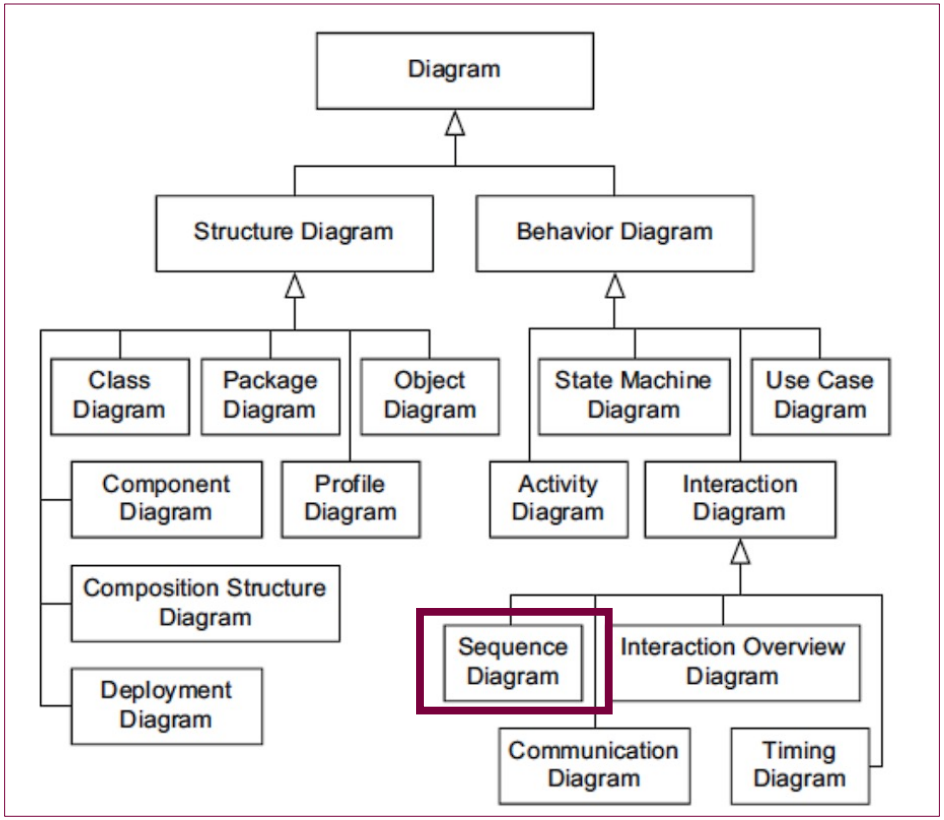


UML: State Machine Diagrams – Simulink Example



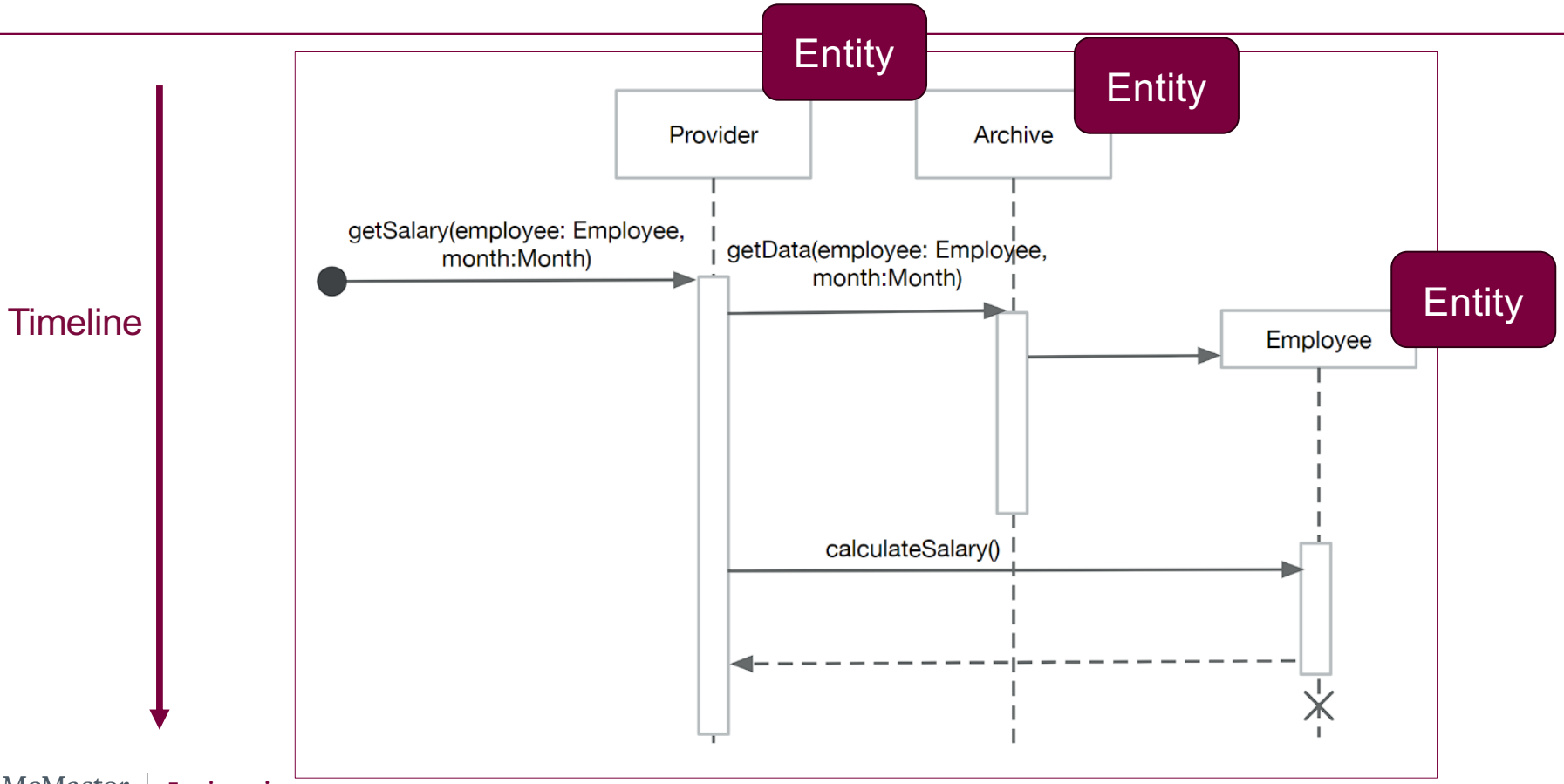
- The initial state in the top state
- Every state has actions to execute
- Transitions can have timing conditions or other Boolean conditions
- Transitions can also have actions to complete during transition (not common or recommended for beginners)

UML: Sequence Diagrams

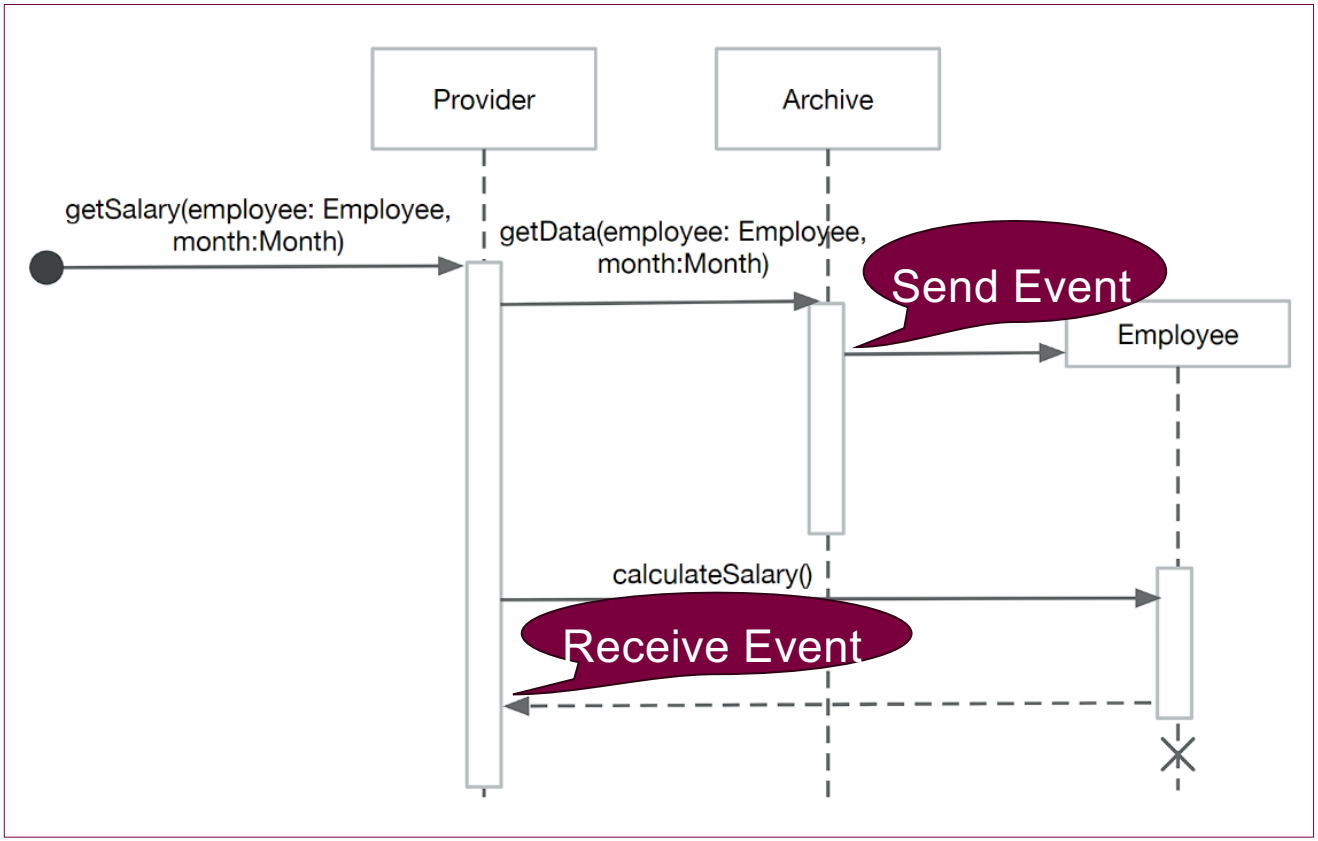


- Specialized for interactions
- Interactions can be everything from component interactions to user interactions
- Models a system by considering:
  - Entities
  - Message exchanges between entities

UML: Sequence Diagrams – Example



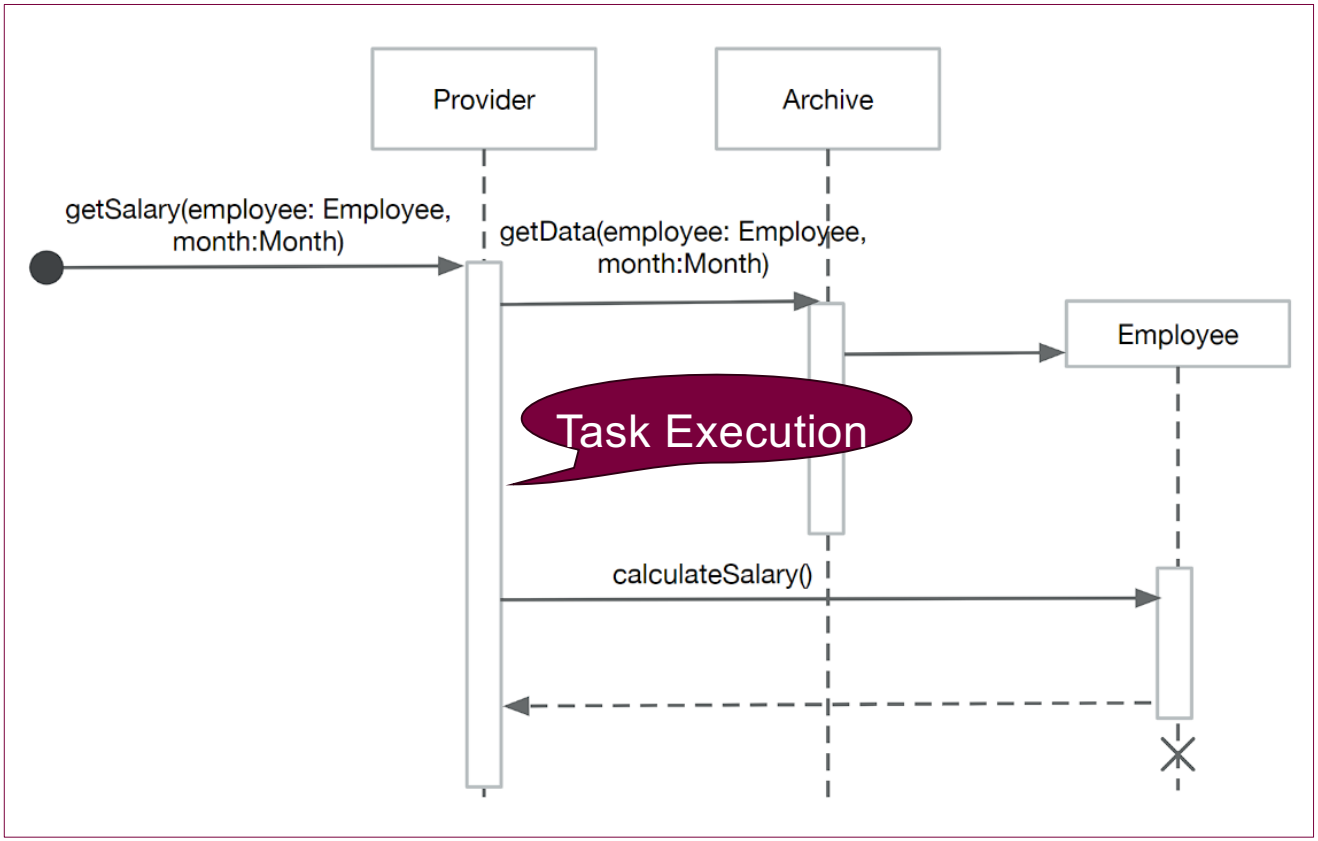
UML: Sequence Diagrams - Example



Timeline

Behaviour Interactions

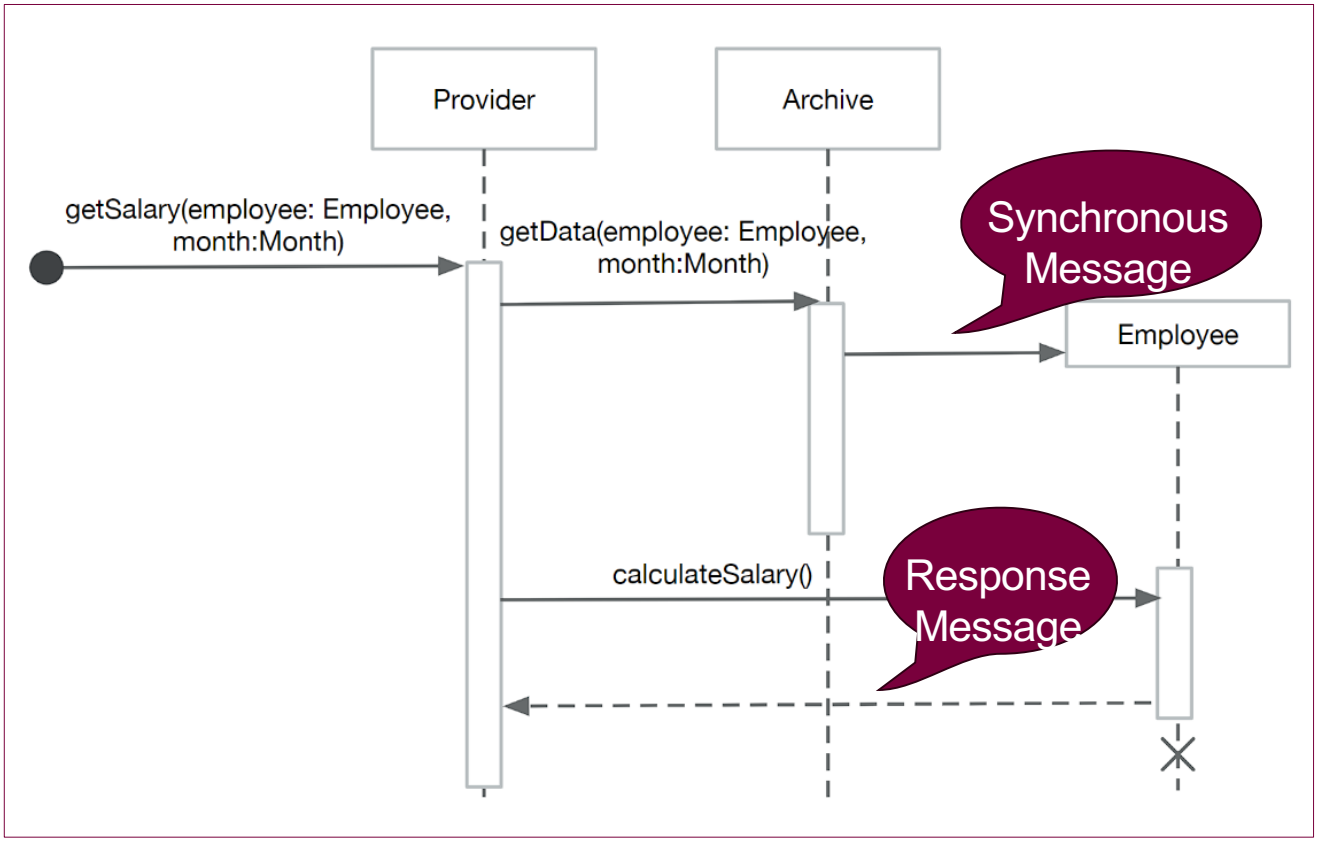
UML: Sequence Diagrams - Example



Timeline

Behaviour Interactions

UML: Sequence Diagrams - Example

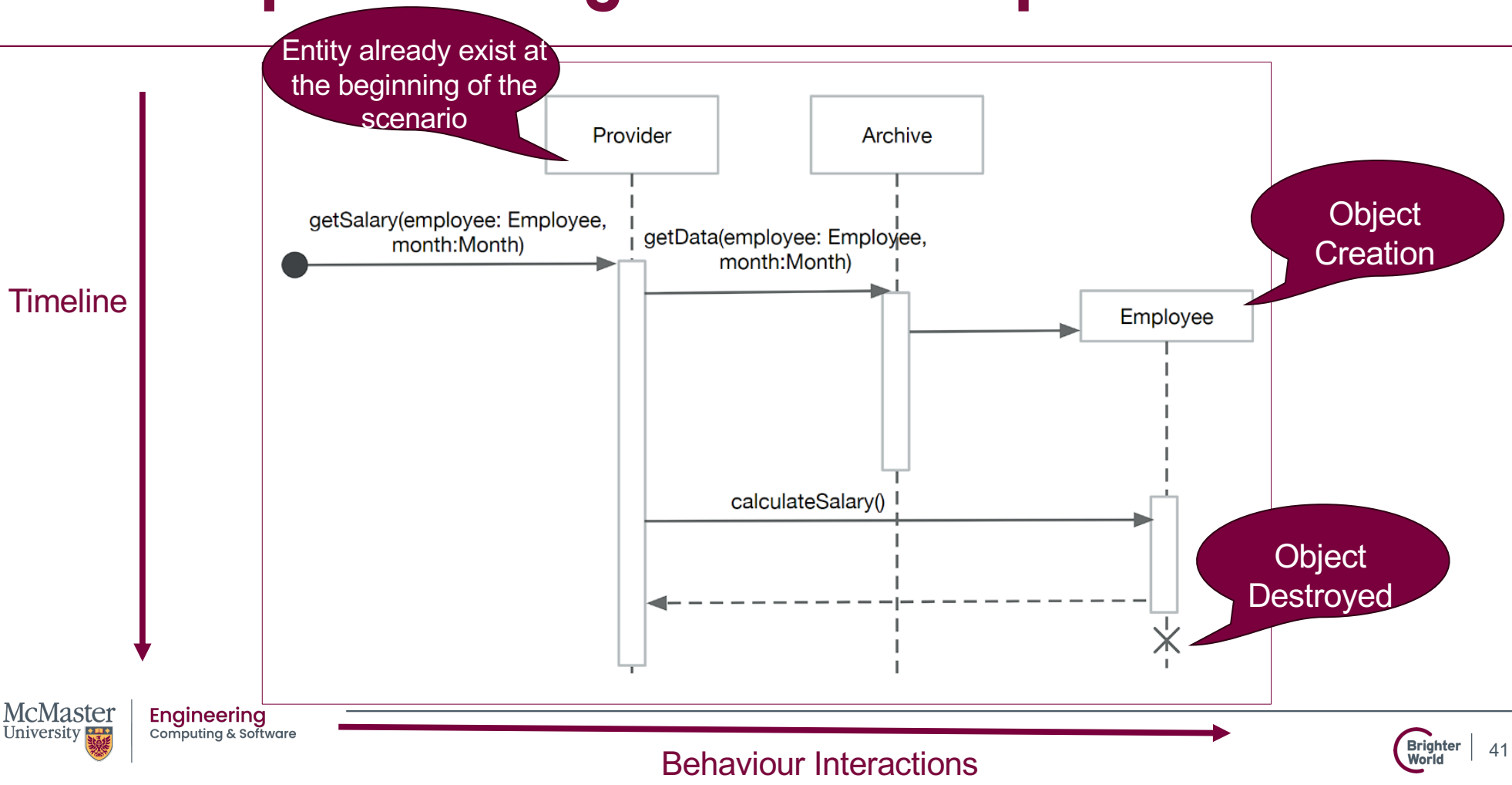


Timeline

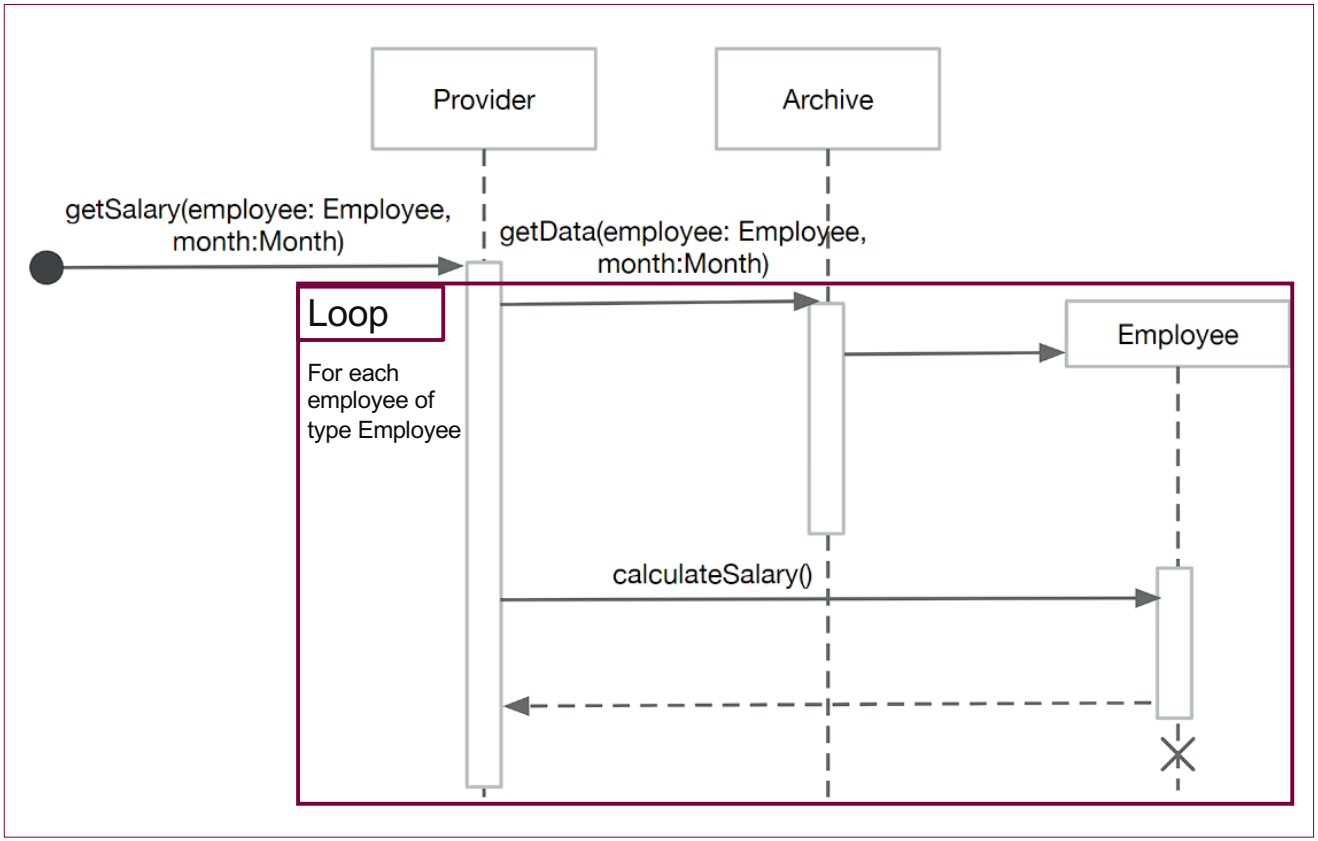
Asynchronous Message

Behaviour Interactions

UML: Sequence Diagrams - Example



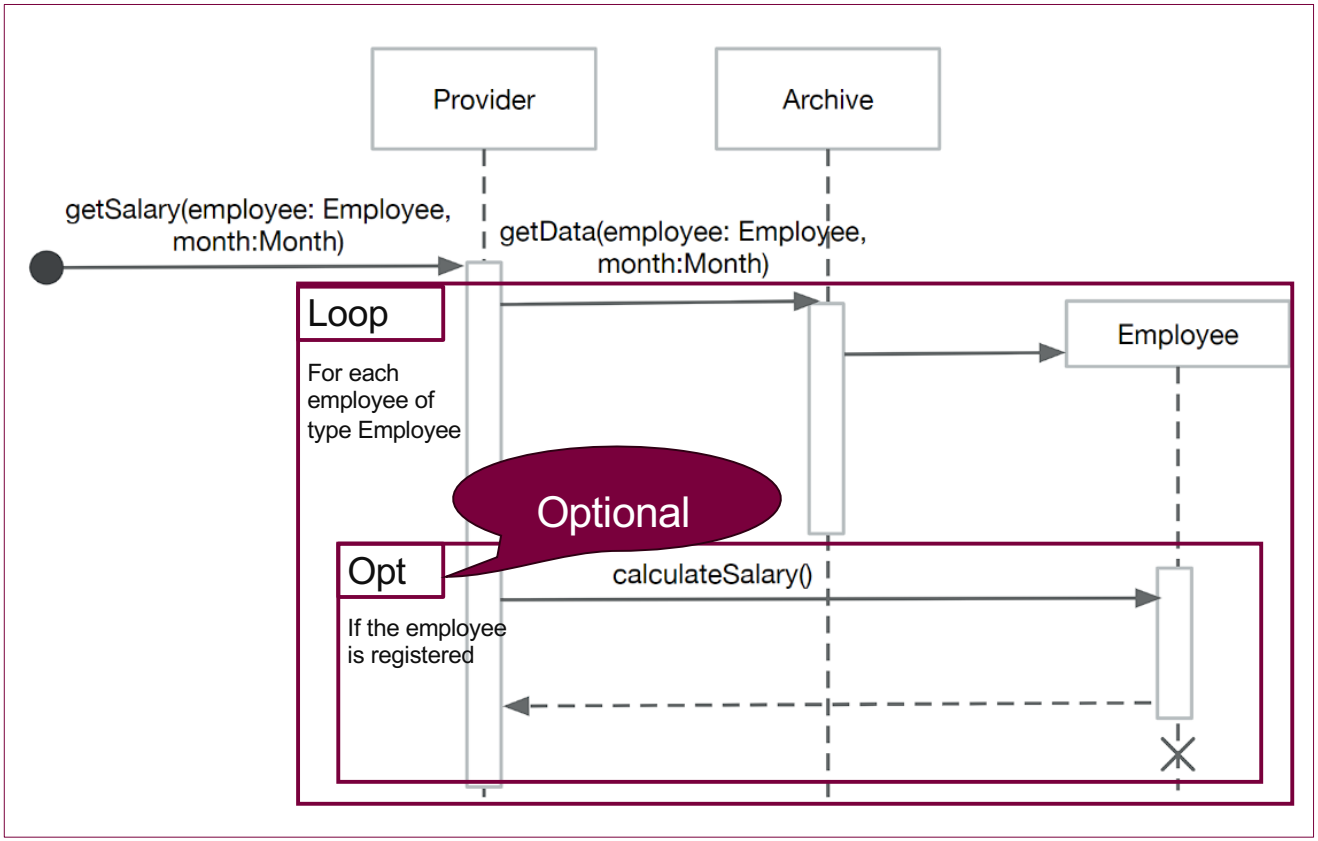
UML: Sequence Diagrams - Example



Timeline

Behaviour Interactions

UML: Sequence Diagrams - Example



Timeline

Behaviour Interactions

Could also have an "Alt" (Alternative). Have a second option at the bottom for those employees who are registered.

UML: Summary – Structure vs. Behaviour

Diagram	Purpose	When to Use	Key Elements
Class Diagram	Defines classes, attributes, methods, and relationships.	When designing the core structure of your system: data, logic, and relationships	Classes, attributes, methods, associations, inheritance, aggregation, compositions
Component/Deployment Diagrams	Show how software components are organized and deployed	When focusing on system architecture or physical deployment	Components, nodes, connections

UML: Summary – Structure vs. Behaviour

Diagram	Purpose	When to Use	Key Elements
Use Case Diagram	Captures system functionality from the user's perspective.	Early in design — to define *requirements* and understand *who interacts with what.*	Actors, use cases, include/extend relationships.
Activity Diagram	Models workflows, control flow, or data flow.	When describing *processes, algorithms,* or *business logic* step-by-step.	Start/end nodes, actions, decisions, forks, joins, edges.
State Machine Diagram	Models how an object changes state in response to events.	When designing *reactive systems*, e.g., embedded systems, UI states, control logic.	States, transitions, events, guard conditions.
Interaction Diagram	Category that includes sequence, communication, and timing diagrams.	When focusing on *communication patterns* between objects.	Entities and exchanged messages.
Sequence Diagram	Models interactions between components over time.	When describing *message flow* or *method calls* that implement a use case.	Lifelines, messages, activation bars, returns.

UML: Does it help modularity?

- Yes!
- Structure Diagrams
  - Recall: it defines the blueprint of the system
  - Architectural modularity: separation of systems into layers/components
- Behaviour Diagrams:
  - Recall: Communication between components
  - Interface modularity: shows how components communicate (communication contracts)

UML: Does it help modularity?

Diagram	Modularity?
Class Diagram	Object-oriented modularity. Encapsulation of data and behaviour.
Use Case Diagram	Functional modularity. Separation of use cases (feature or services)
Activity Diagram	Process modularity Each activity can become an independent function.
State Machine Diagram	Behavioural modularity. Each state machine represents a self-contained behavioural unit.
Sequence Diagram	Interaction boundaries Clarifies which object of module is responsible for which part of the communication flow

Let’s Review!