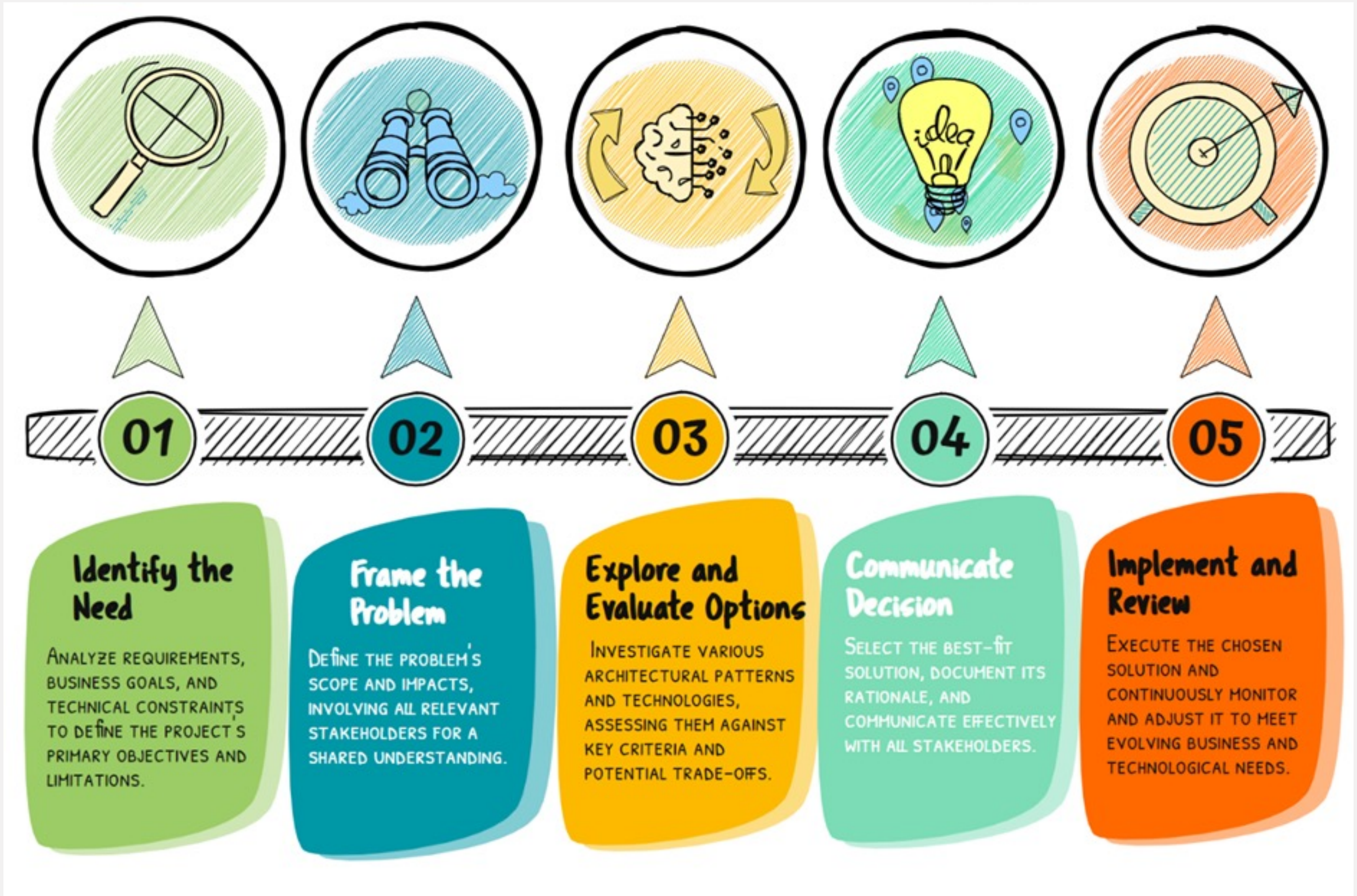
Architectural Decision Process

Lecture-3

Core Concepts of Architectural Decision Process



01 Identify the Need

Analyze requirements, business goals, and technical constraints to define the project's primary objectives and limitations.

02 Frame the Problem

Define the problem's scope and impacts, involving all relevant stakeholders for a shared understanding.

03 Explore and Evaluate Options

Investigate various architectural patterns and technologies, assessing them against key criteria and potential trade-offs.

04 Communicate Decision

Select the best-fit solution, document its rationale, and communicate effectively with all stakeholders.

05 Implement and Review

Execute the chosen solution and continuously monitor and adjust it to meet evolving business and technological needs.

1. Identify the Need

Begin by understanding the need, which includes gathering requirements, understanding business objectives, and recognizing technical challenges. It’s about asking the right questions to determine the problems to be solved, business goals, and technical constraints.

2. Framing the Problem

Define the problem’s scope, identify stakeholders, and understand potential solution impacts. Clear articulation ensures common understanding among all stakeholders

3. Exploring & Evaluating Options

Explore architectural options, considering different patterns, technologies, and approaches. Evaluate these against criteria like scalability, maintainability, cost, and performance. This stage often involves analyzing trade-offs to weigh the pros and cons of each option

4. Communicate Decision

Select the most suitable option based on a comprehensive understanding of trade-offs and alignment with project goals. Document the decision thoroughly for transparency and future reference, and communicate effectively with all involved parties.

5. Implement & Review

Implement the chosen decision and continuously review its impact. Architectural decisions are not static; they should be revisited to ensure they continue to meet the evolving needs of the business and technology.

Principles in Architectural Decision

- **Scalability**

  Essential in today’s rapidly changing digital landscape, scalability ensures systems can handle increasing loads without performance loss. This involves strategic decisions on database scalability and scaling strategies.

- **Maintainability**

  Vital for long-term system success, maintainability focuses on simple, modular design for ease of updates and adherence to coding standards.

- **Security**

  In an era of evolving cyber threats, security is integral. This encompasses data encryption, secure communication, and compliance with standards like GDPR.

Principles in Architectural Decision

- **Reliability & Resilience**

  Systems must be robust, designed for fault tolerance, and maintain functionality under diverse conditions

- **Performance**

  Optimizing system performance is key, requiring careful algorithm selection and resource efficiency.

- **Flexibility and Adaptability**

  Architectures must accommodate new technologies and evolving business requirements.

- **Cost-Efficiency**

  Balancing initial and long-term operational costs with the benefits of architectural choices is critical.

What is UML?

An emerging standard for modeling object-oriented software

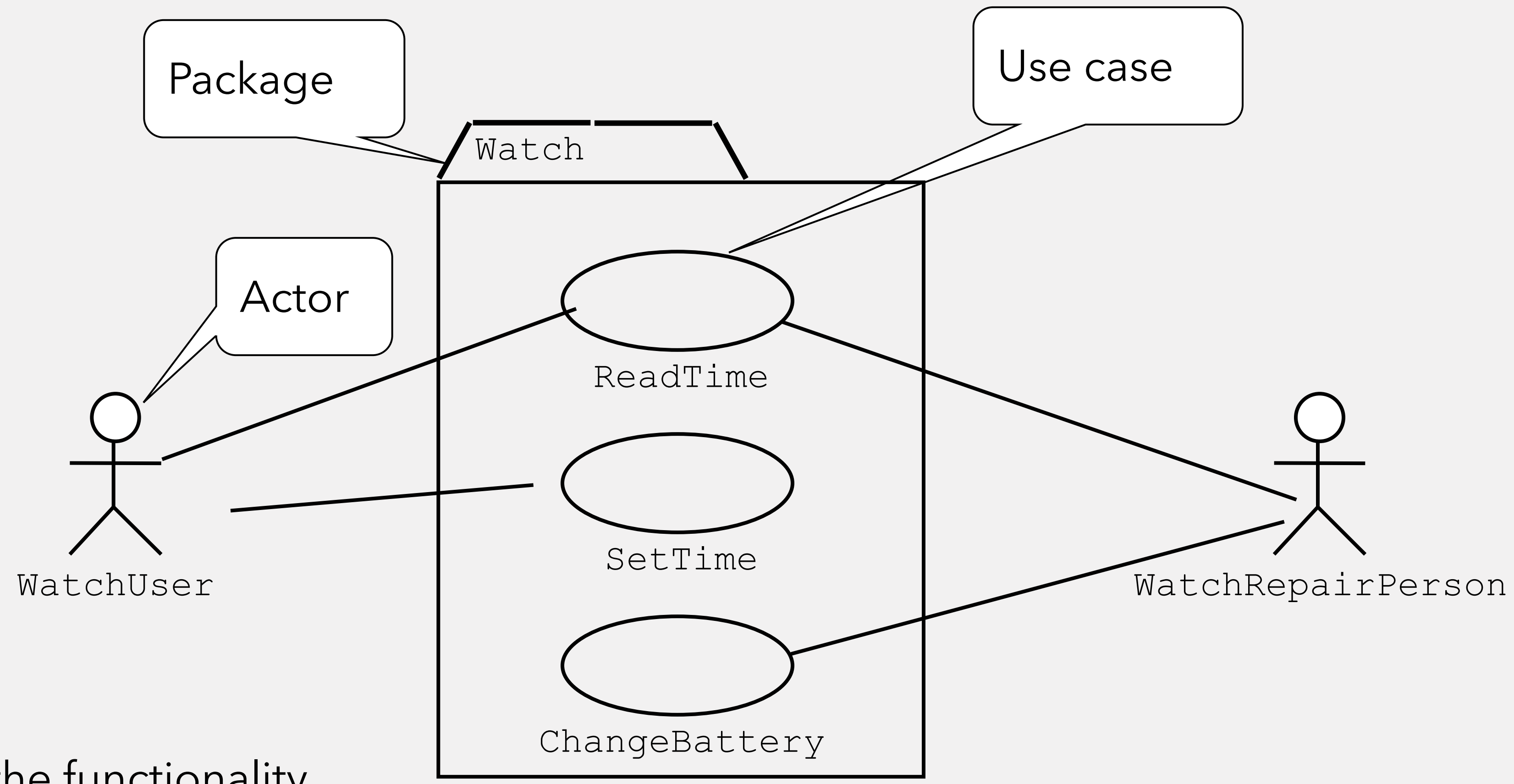
OR

UML stands for Unified Modelling Language. It’s a rich language to model software solutions, application structures, system behavior, and business processes.

Common UML Diagrams

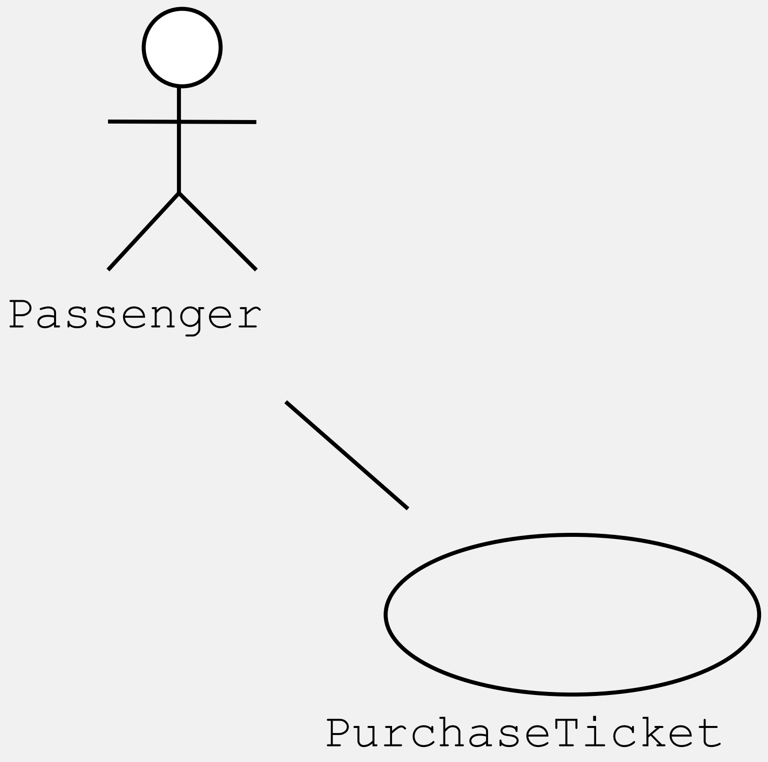
- **Use case Diagrams**
  Describe the functional behavior of the system as seen by the user.
- **Class diagrams**
  Describe the static structure of the system: Objects, Attributes, Associations
- **Sequence diagrams**
  Describe the dynamic behavior between actors and the system and between objects of the system
- **State diagrams**
  Describe the dynamic behavior of an individual object (essentially a finite state automaton)
- **Activity Diagrams**
  Model the dynamic behavior of a system, in particular, the workflow (essentially a flowchart)

Use Case Diagram



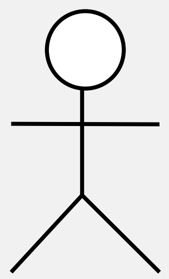
Use case diagrams represent the functionality of the system from user's point of view

Use Case Diagram



- Used during requirements elicitation to represent external behavior
- ***Actors*** represent roles, that is, a type of user of the system
- ***Use cases*** represent a sequence of interaction for a type of functionality
- The use case model is the set of all use cases. It is a complete description of the functionality of the system and its environment

Actors



- An actor models an external entity which communicates with the system:

  User

  External system

  Physical environment

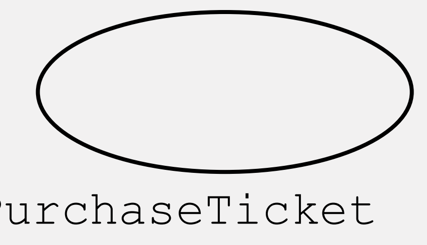
- An actor has a unique name and an optional description.

- Examples:

  Passenger: A person in the train

  GPS satellite: Provides the system with GPS coordinates

Use Case

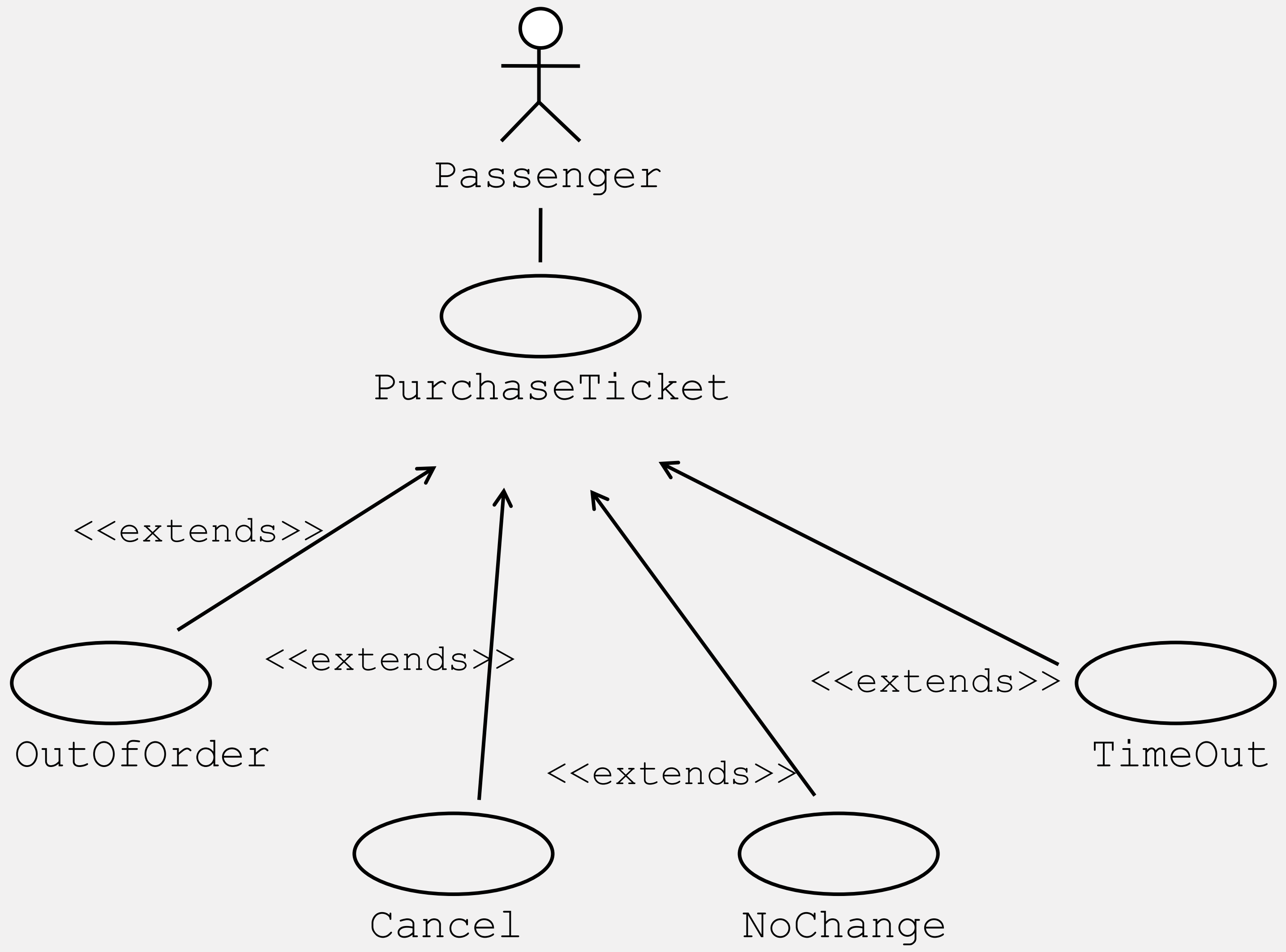


A use case represents a class of functionality provided by the system as an event flow.

A use case consists of:

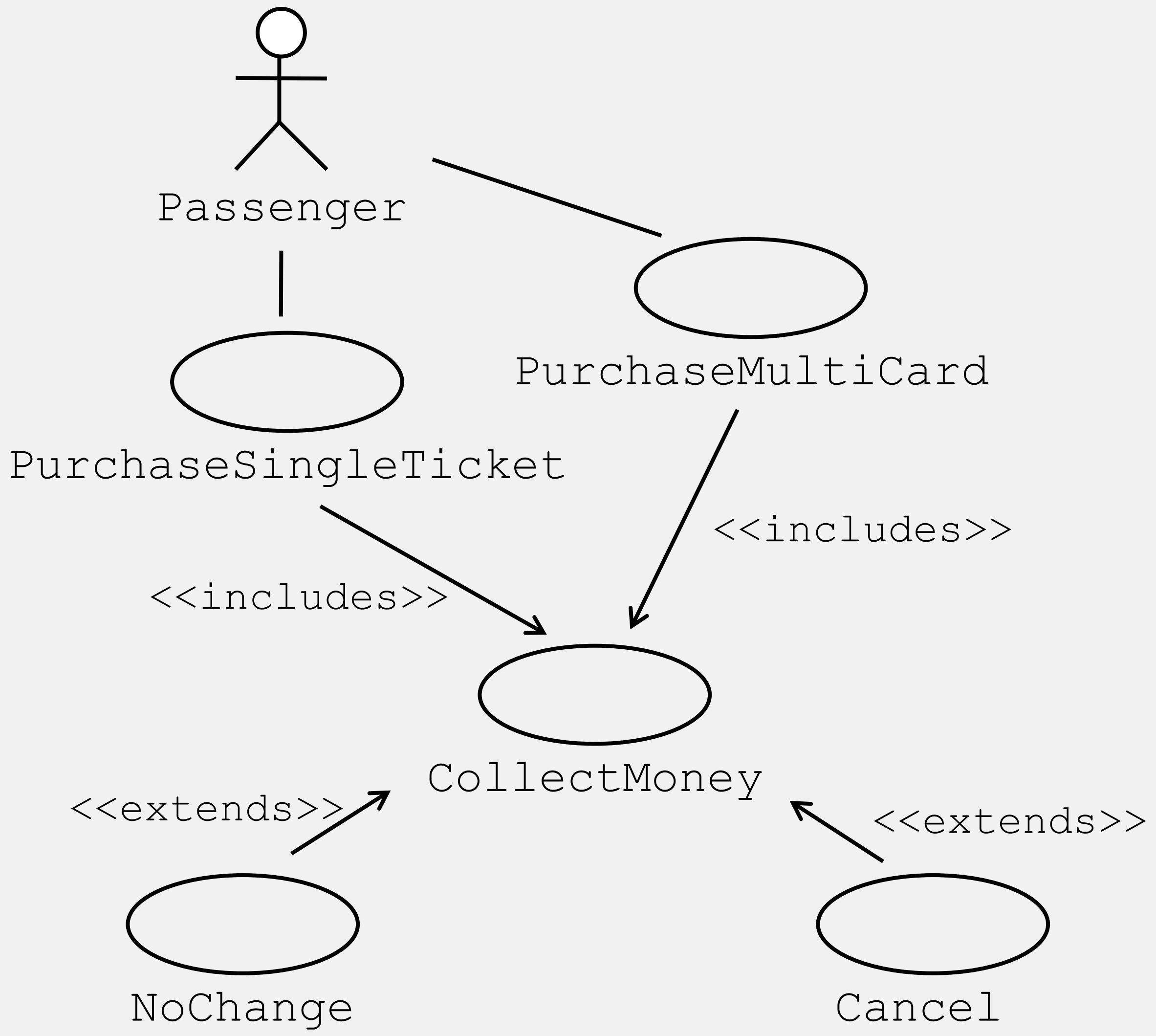
- Unique name
- Participating actors
- Entry conditions
- Flow of events
- Exit conditions
- Special requirements

The <<extends>> Relationship



- <<extends>> relationships represent exceptional or seldom invoked cases.
- The exceptional event flows are factored out of the main event flow for clarity.
- Use cases representing exceptional flows can extend more than one use case.
- The direction of a <<extends>> relationship is to the extended use case

The <<includes>> Relationship

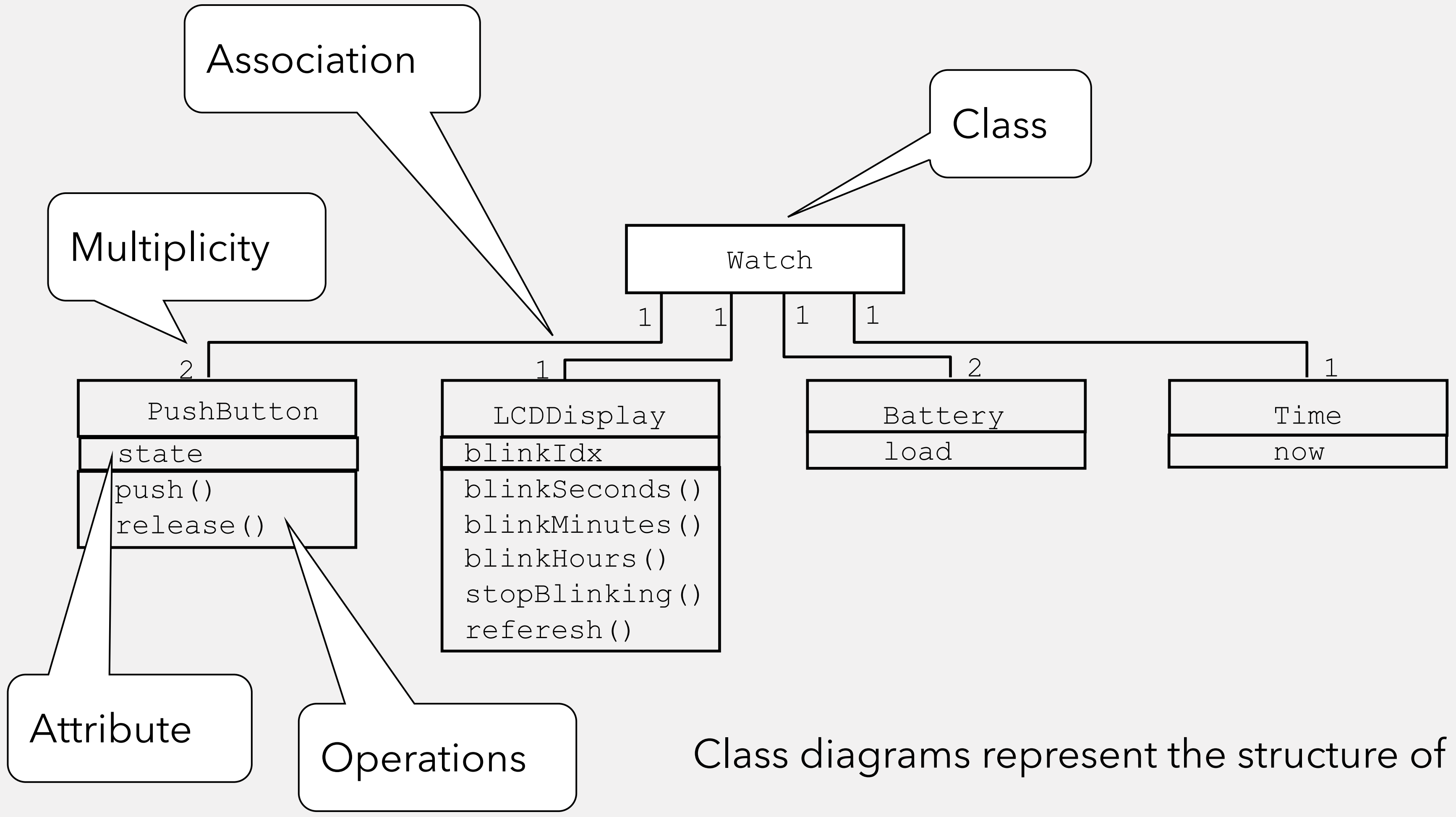


- <<includes>> relationship represents behavior that is factored out of the use case.
- <<includes>> behavior is factored out for reuse, not because it is an exception.
- The direction of a <<includes>> relationship is to the using use case (unlike <<extends>> relationships).

Use Case Diagrams: Summary

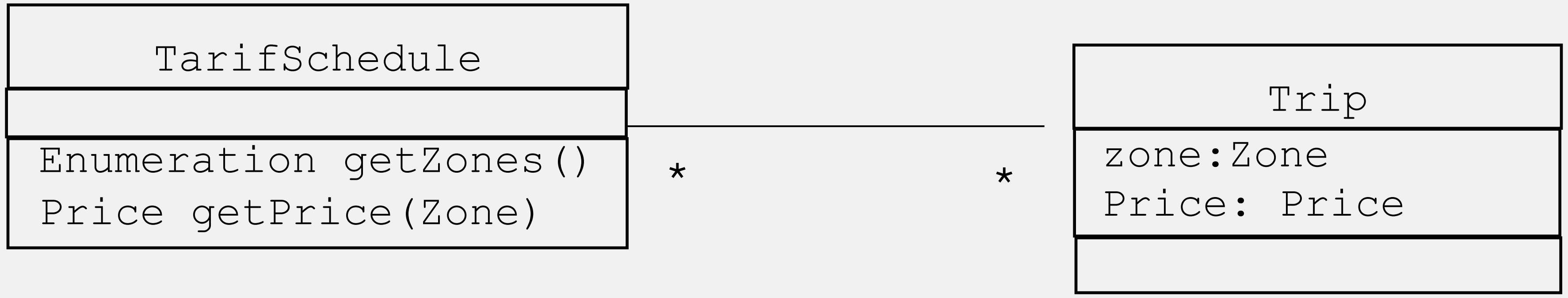
- Use case diagrams represent external behavior
- Use case diagrams are useful as an index into the use cases
- Use case descriptions provide meat of model, not the use case diagrams.
- All use cases need to be described for the model to be useful.

Class Diagram



Class diagrams represent the structure of the system

Class Diagram



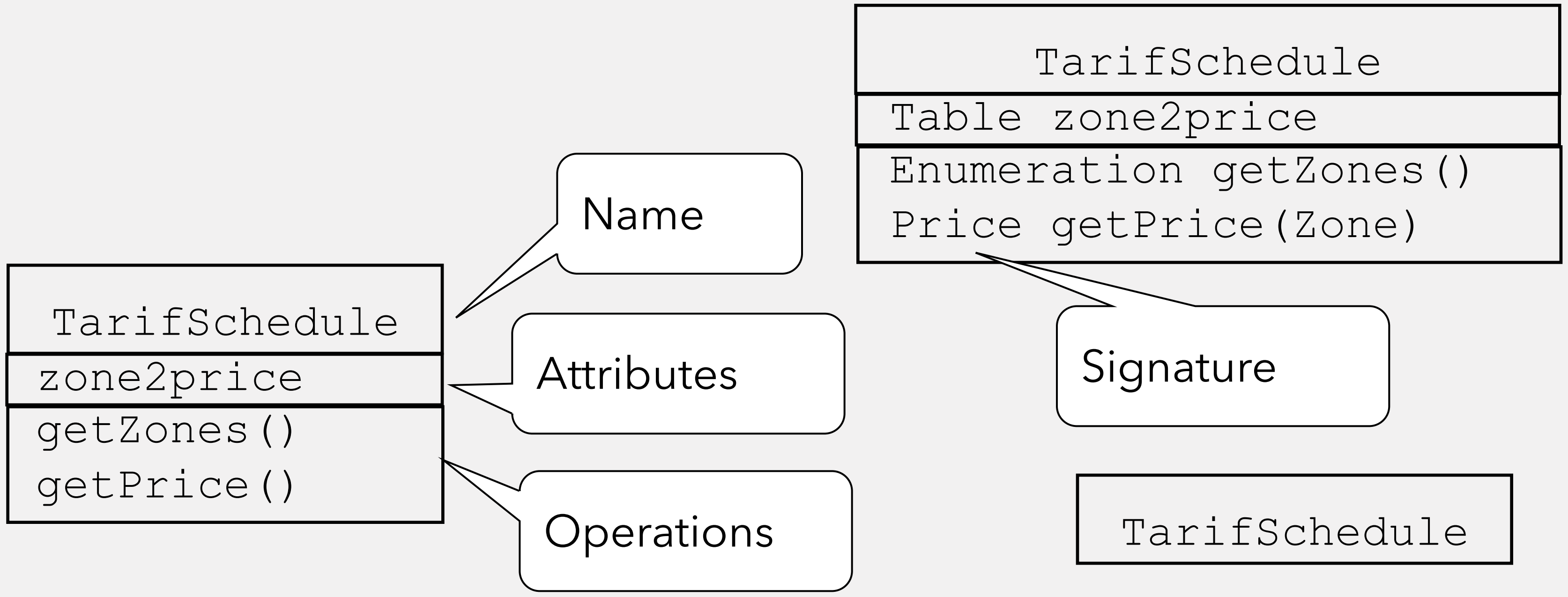
- Class diagrams represent the structure of the system.
- Used

  during requirements analysis to model problem domain concepts

  during system design to model subsystems and interfaces

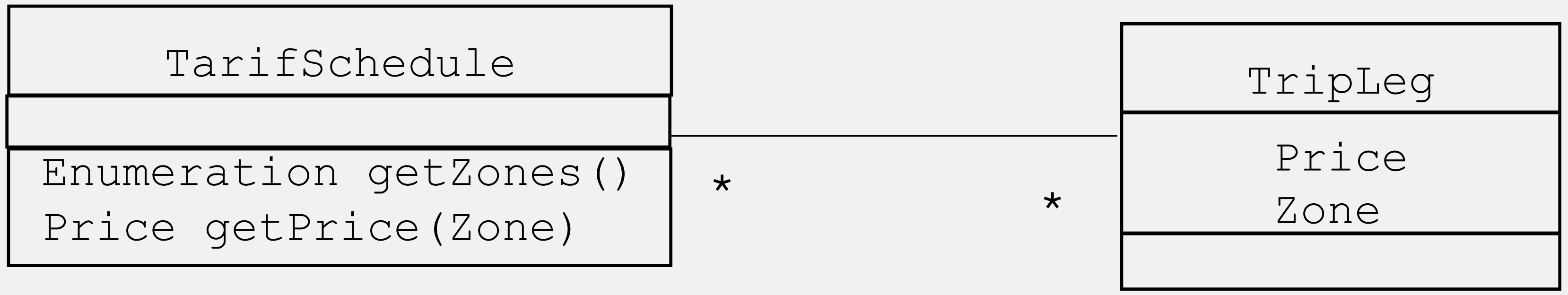
  during object design to model classes.

Classes



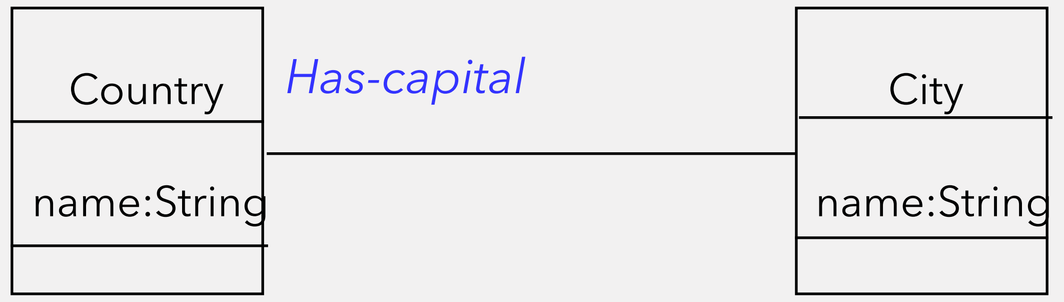
- A ***class*** represent a concept
- A class encapsulates state ***(attributes)*** and behavior ***(operations).***
- Each attribute has a ***type***.
- Each operation has a ***signature***.
- The class name is the only mandatory information.

Associations

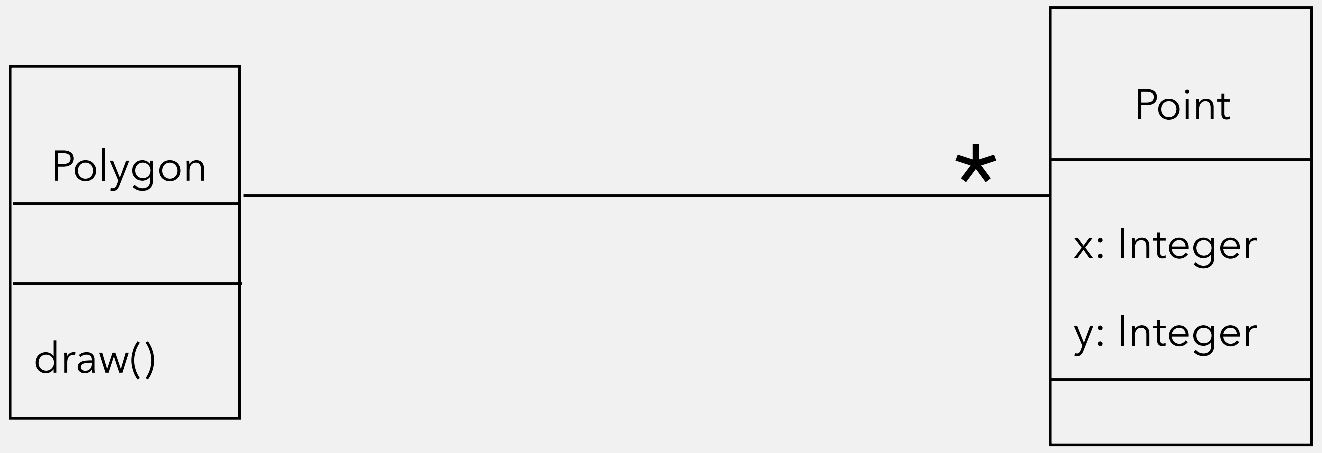


- Associations denote relationships between classes.
- The multiplicity of an association end denotes how many objects the source object can legitimately reference.

1-to-1 and 1-to-many Associations

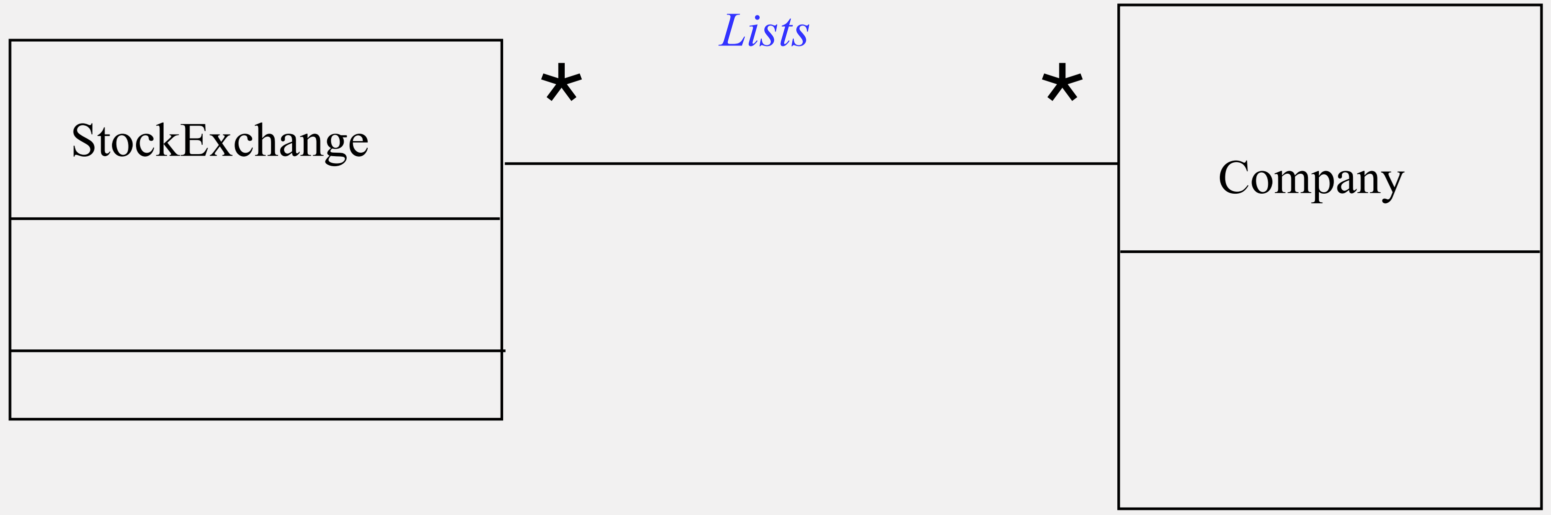


One-to-one association



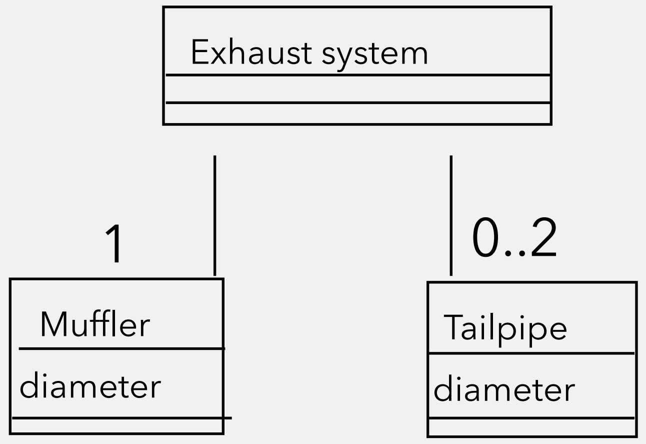
One-to-many association

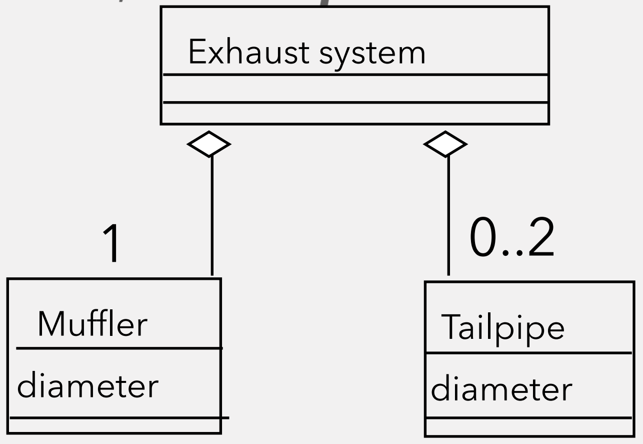
Many-to-Many Associations

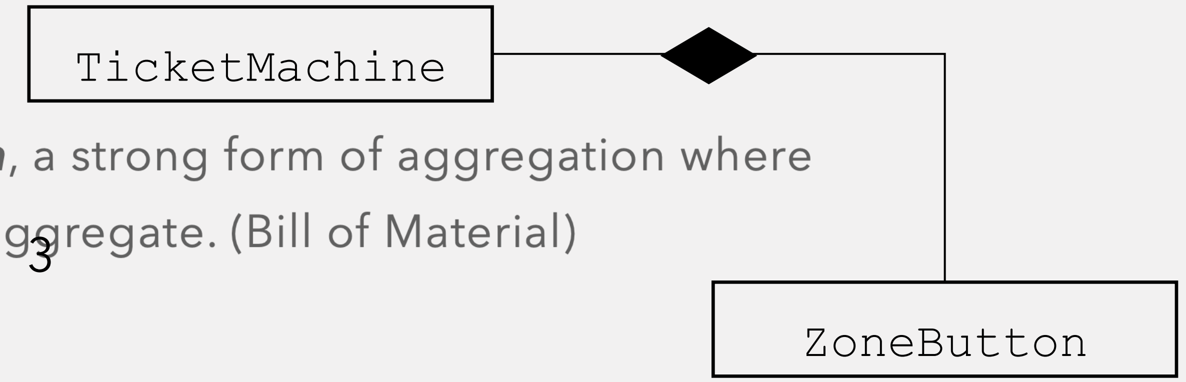


Aggregation

- An ***aggregation*** is a special case of association denoting a "consists of" hierarchy.
- The ***aggregate*** is the parent class, the ***components*** are the children class.

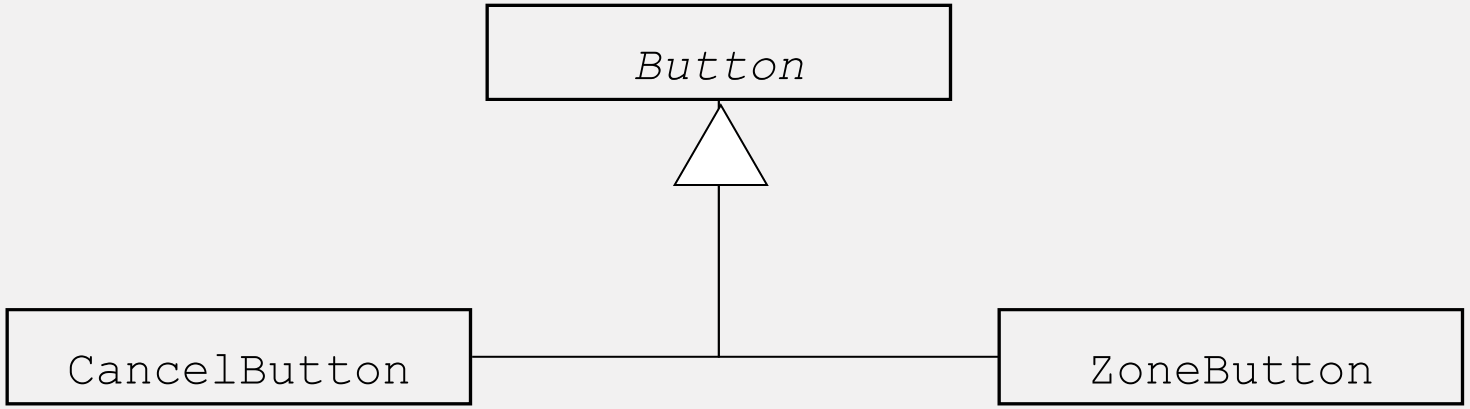






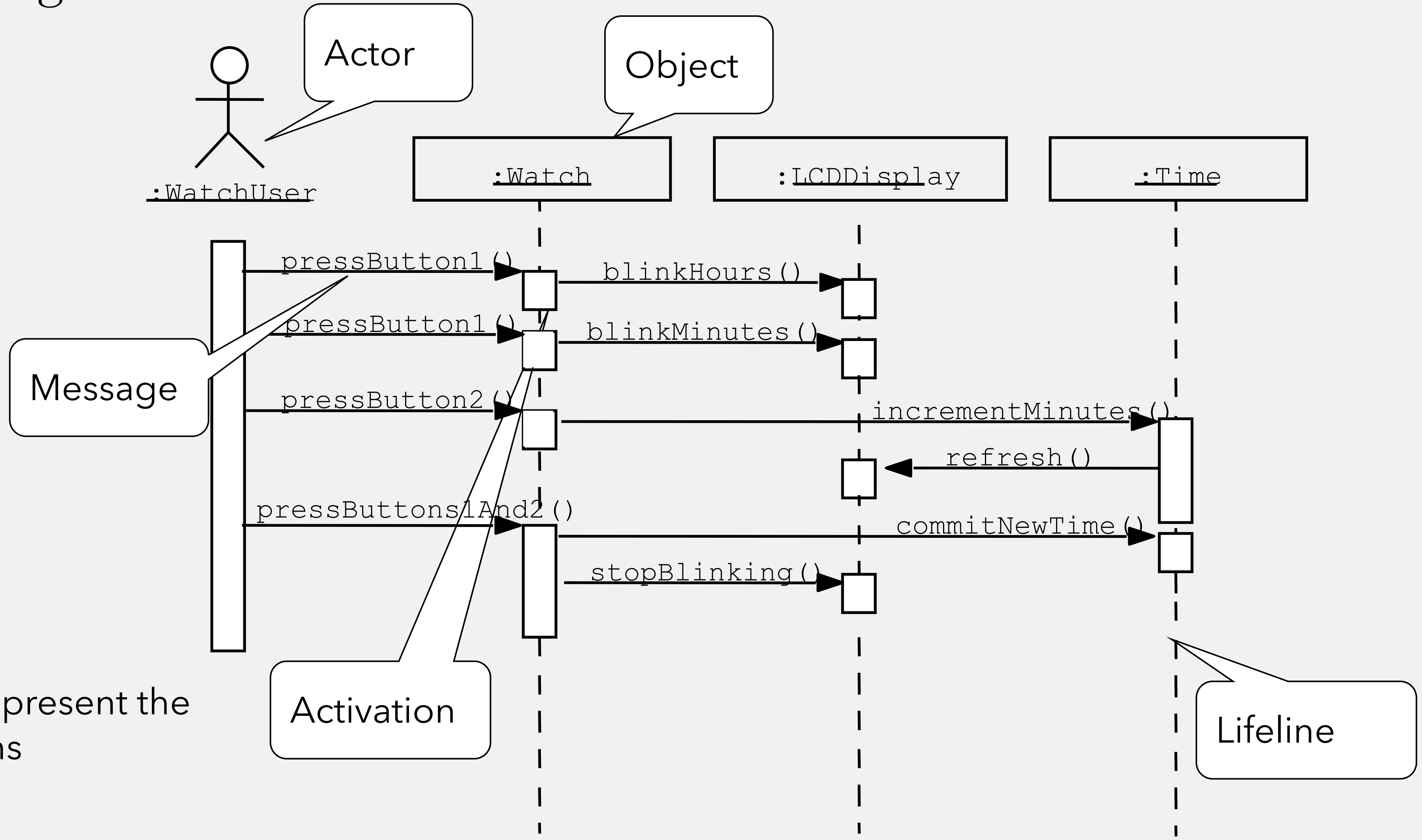
- A solid diamond denotes ***composition***, a strong form of aggregation where components cannot exist without the aggregate. (Bill of Material)

Inheritance



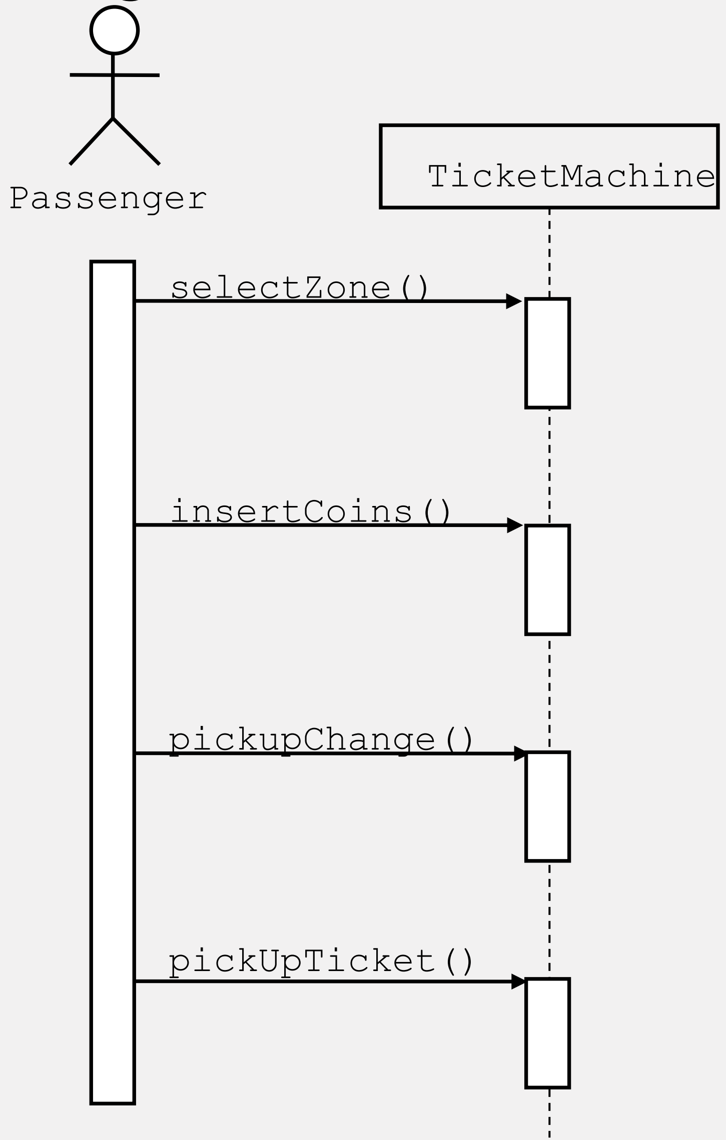
- The **children classes** inherit the attributes and operations of the **parent class**.
- Inheritance simplifies the model by eliminating redundancy.

Sequence Diagram



Sequence diagrams represent the behavior as interactions

Sequence diagrams



- Used during requirements analysis

  To refine use case descriptions

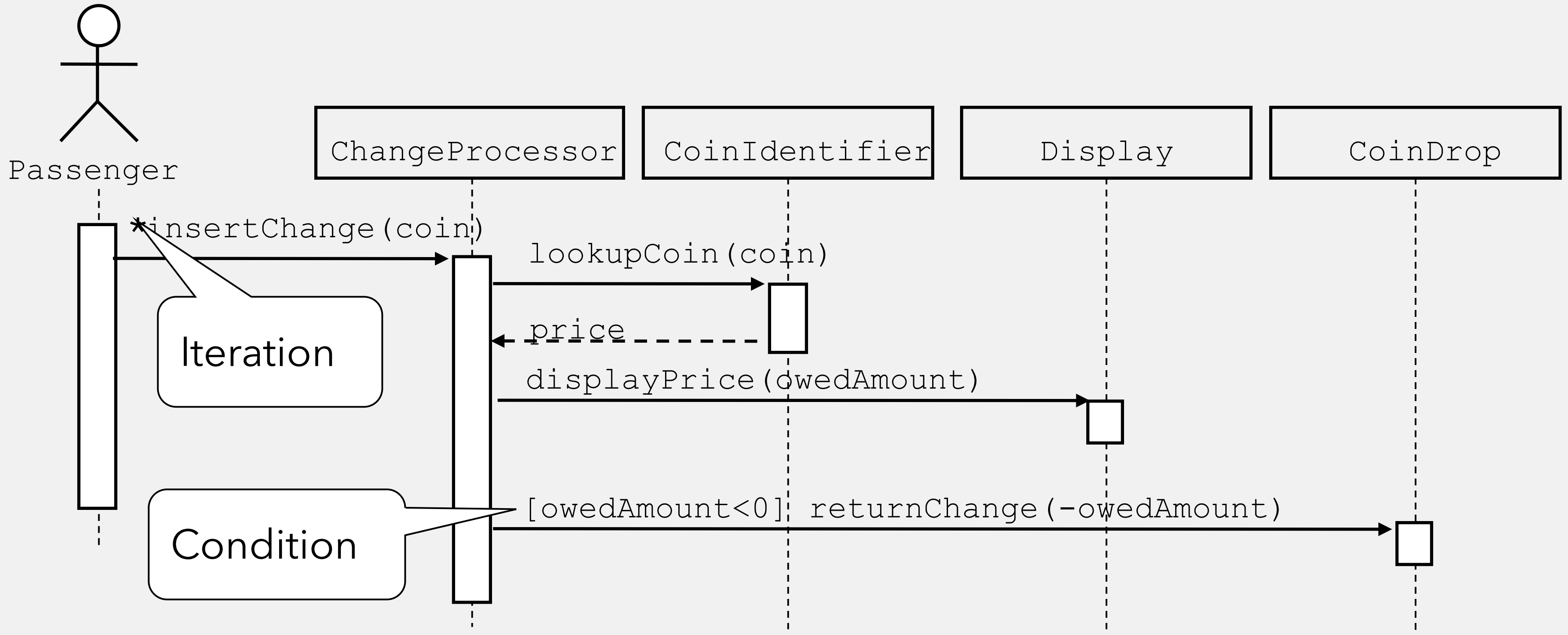
  to find additional objects (“participating objects”)

- Used during system design

  to refine subsystem interfaces

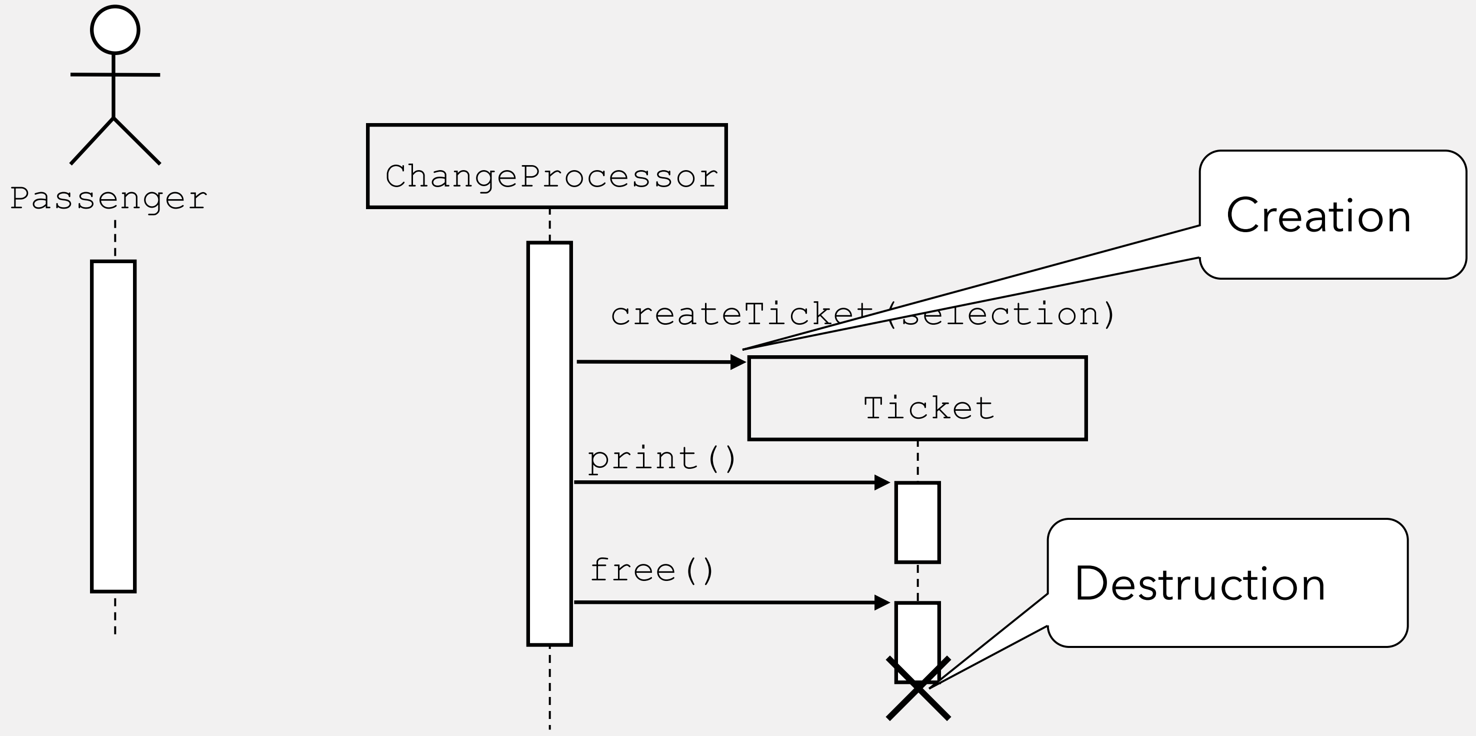
- ***Classes*** are represented by columns
- ***Messages*** are represented by arrows
- ***Activations*** are represented by narrow rectangles
- ***Lifelines*** are represented by dashed lines

Iteration & condition



- Iteration is denoted by a * preceding the message name
- Condition is denoted by boolean expression in [ ] before the message name

Creation and destruction

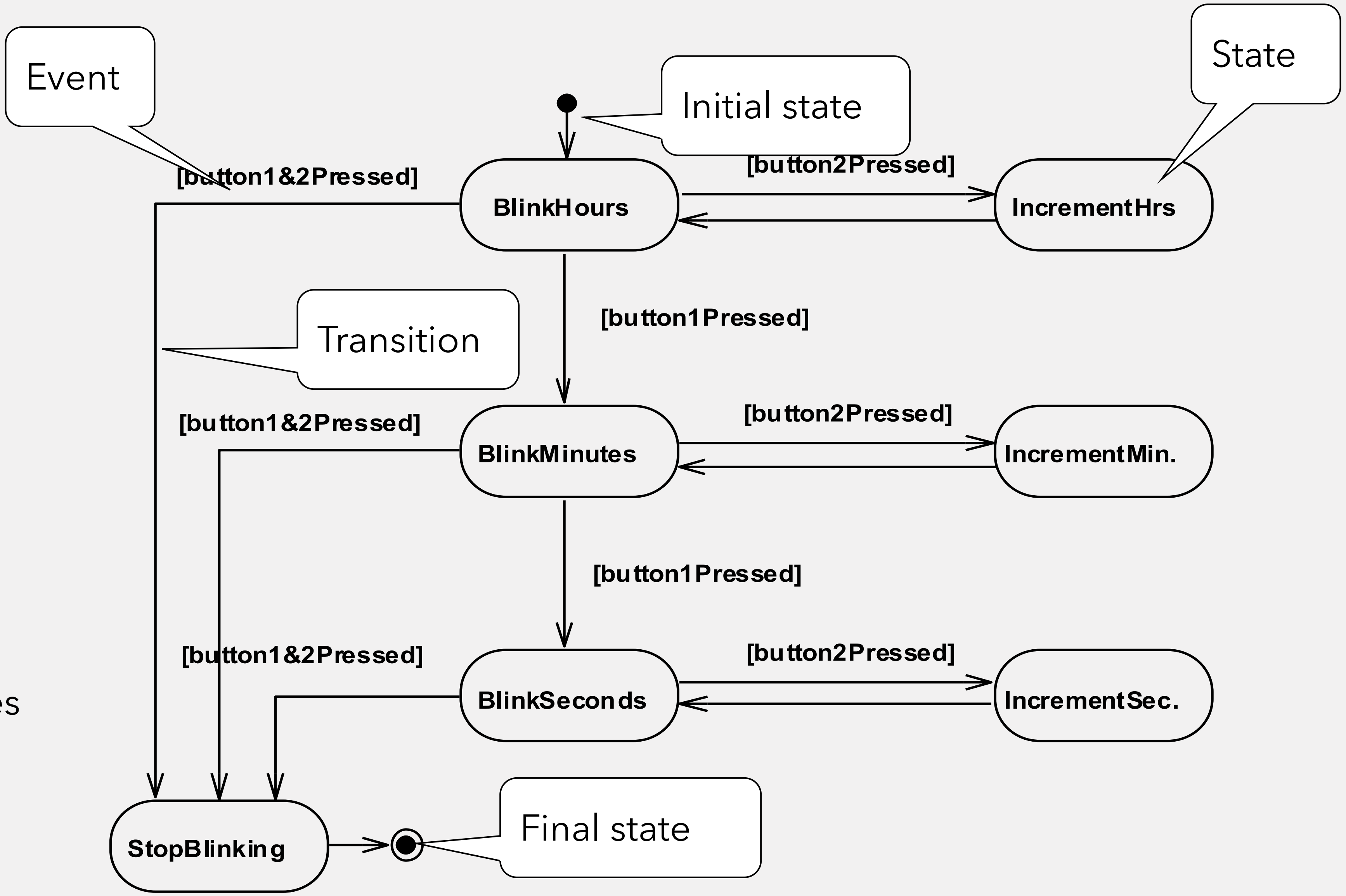


- Creation is denoted by a message arrow pointing to the object.
- Destruction is denoted by an X mark at the end of the destruction activation.
- In garbage collection environments, destruction can be used to denote the end of the useful life of an object.

Sequence Diagram Summary

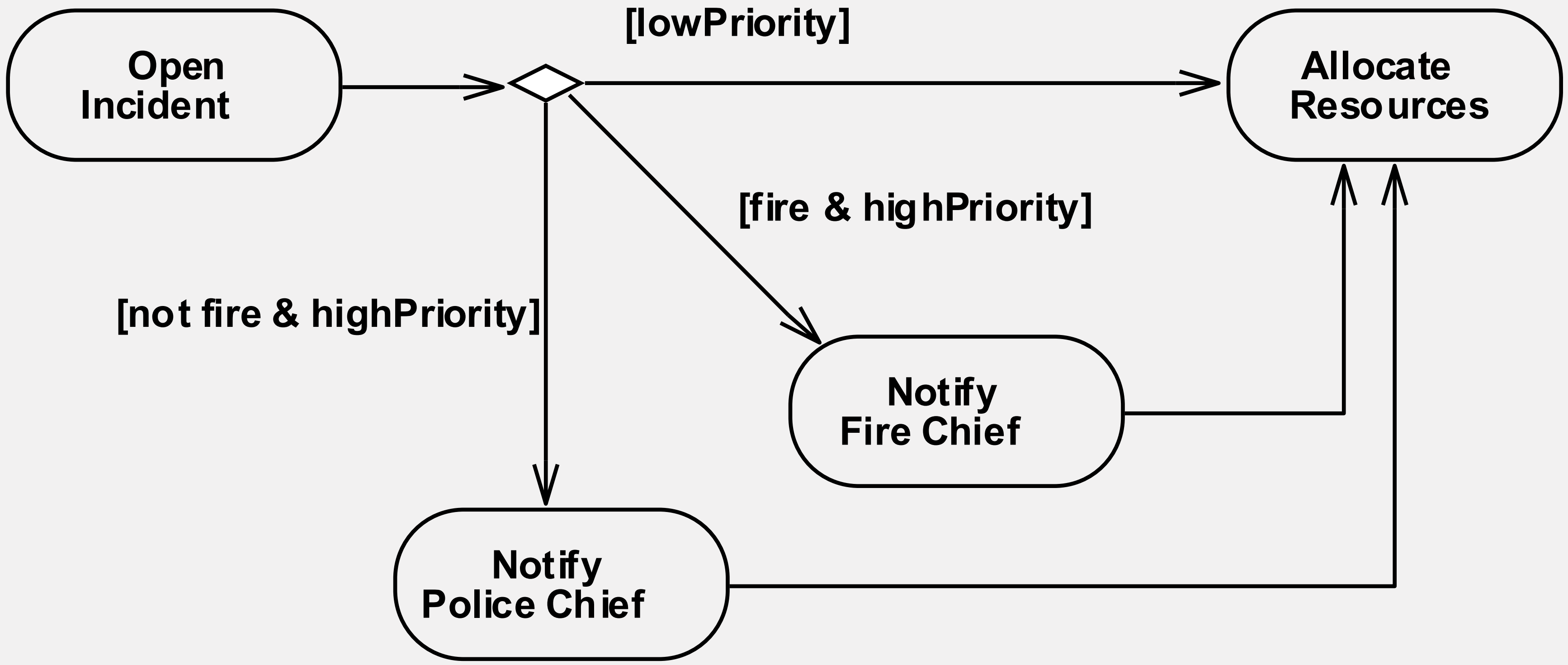
- UML sequence diagram represent behavior in terms of interactions.
- Useful to find missing objects.
- Time consuming to build but worth the investment.
- Complement the class diagrams (which represent structure).

State Diagram



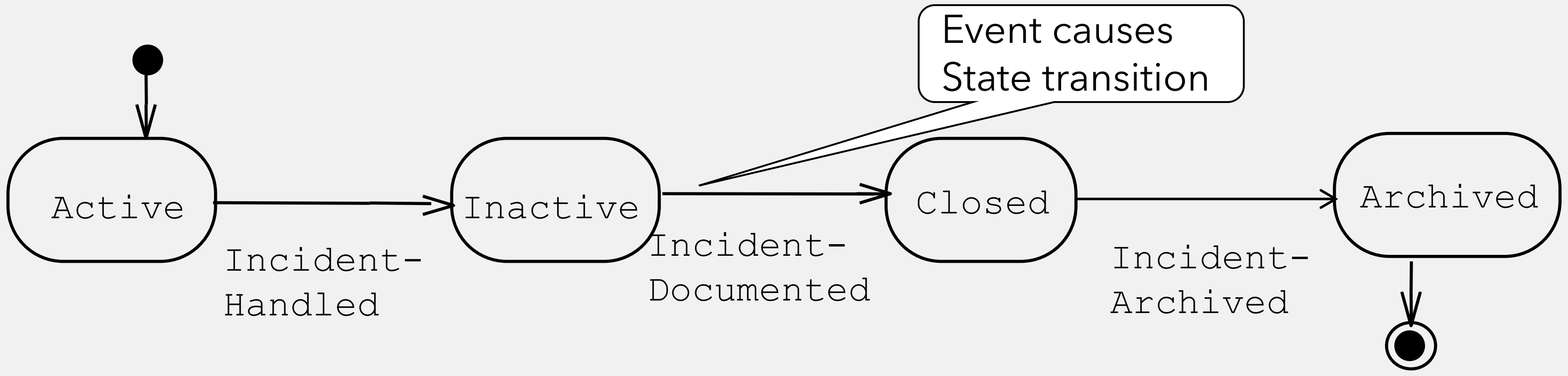
Represent behavior as states and transitions

Activity Diagram

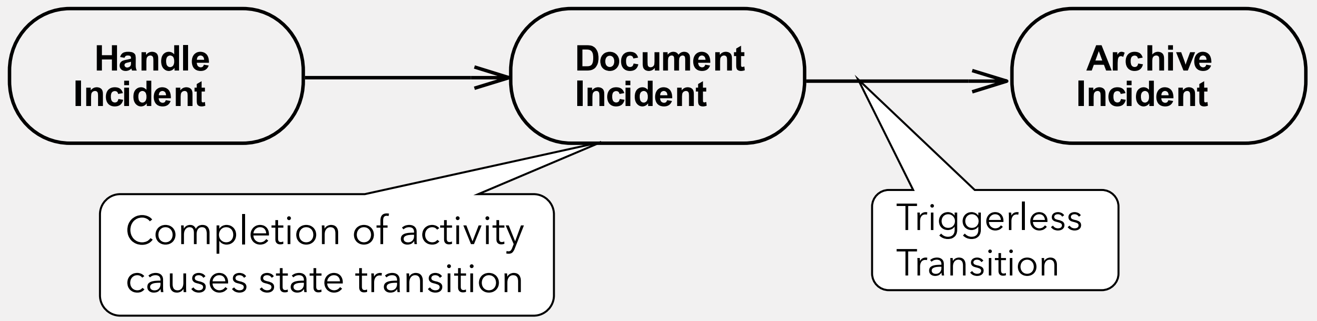


State vs Activity Diagram

Statechart Diagram for Incident
Attribute or Collection of Attributes of object of type Incident

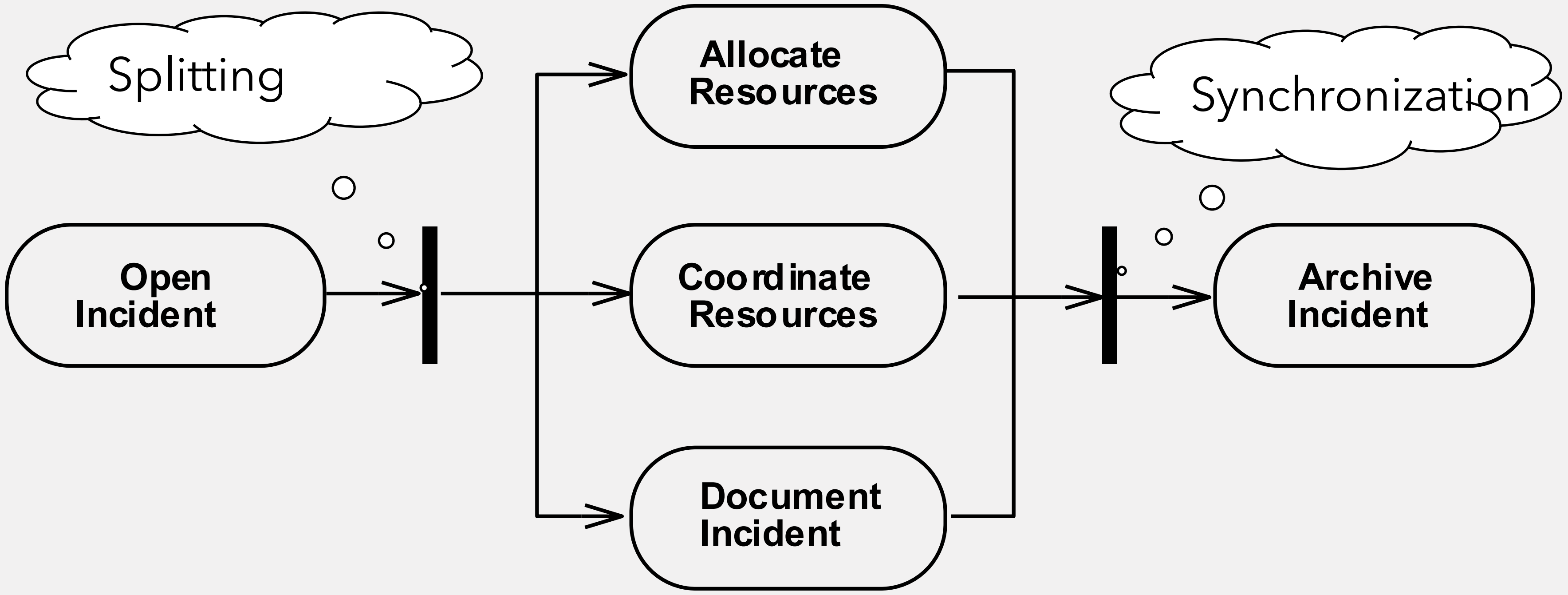


Activity Diagram for Incident
Operation or Collection of Operations)



Activity Diagrams: Modeling Concurrency

- Synchronization of multiple activities
- Splitting the flow of control into multiple threads



UML Summary

- UML provides a wide variety of notations for representing many aspects of software development

  Powerful, but complex language

  Can be misused to generate unreadable models

  Can be misunderstood when using too many exotic features

- For now we concentrate on a few notations:

  Functional model: Use case diagram

  Object model: class diagram

  Dynamic model: sequence diagrams, statechart and activity diagrams