**Object-Oriented Design with UML**

Table of Contents

[Abstraction 3](#_Toc83393286)

[Multiple Inheritance 4](#_Toc83393287)

[Object Oriented Design 5](#_Toc83393288)

[UML Basics 6](#_Toc83393289)

[UML Diagram Categories 7](#_Toc83393290)

[Use Case Diagrams 8](#_Toc83393291)

[Actors 8](#_Toc83393292)

[Use Cases 9](#_Toc83393293)

[<<extends>> and <<includes>> Relationships 10](#_Toc83393294)

[Class Diagrams 12](#_Toc83393295)

[Classes and Instances 12](#_Toc83393296)

[Class Notation 13](#_Toc83393297)

[Relationships 14](#_Toc83393298)

[Dependencies 14](#_Toc83393299)

[Generalizations 14](#_Toc83393300)

[Association 15](#_Toc83393301)

[Aggregations and Compositions 16](#_Toc83393302)

[Multiplicities 17](#_Toc83393303)

[Object Diagram 19](#_Toc83393304)

[Package Diagram 21](#_Toc83393305)

[Sequence Diagram 22](#_Toc83393306)

[Message Arrows 23](#_Toc83393307)

[Activity Diagrams 25](#_Toc83393308)

[State Diagram 27](#_Toc83393309)

[Component Diagrams 29](#_Toc83393310)

[Deployment Diagram 31](#_Toc83393311)

## Abstraction

An **abstract** model is one that includes the most important details of a system while ignoring the less important ones.

Abstraction allows us to **manage complexity** by concentrating on essential characteristics that make an entity different from others.

For example, if we are talking about an order processing system that has a salesman involved, we should not be concerned about which exact salesman will be present. The major point is that some salesman is there.

## Multiple Inheritance

A single class can inherit from **multiple classes**. However, we should not use this unless absolutely necessary, because it can quickly get messy.

## Object Oriented Design

We use **object-oriented design** because it makes managing the system easy. If we need to **add** new features, having the existing system following an object-oriented design will make the work easy.

Going back to the order processing system example, following an object-oriented design, we may have a Customer class involved. If we later decide to have multiple groups of customers, we can just create new classes that inherit from the Customer class. We do not have to modify, or even think about, the entire system.

## UML Basics

The process of **modelling** involves creating an abstraction of reality. As mentioned before, we ignore irrelevant details and only include relevant ones. What exactly is relevant depends on the **purpose** of the model.

Software is becoming increasing **complex**. For large systems, there are sometimes millions of lines of code involved involving hundreds of developers. Sadly, code is sometimes difficult to understand for someone who did not write it. To deal with this and create a **simple representation** of the complex system, we perform modelling.

The **Unified Modelling Language** (UML) is a modelling language used to express and design documents and software. It is particularly useful for object-oriented design. It works independently to the implementation language.

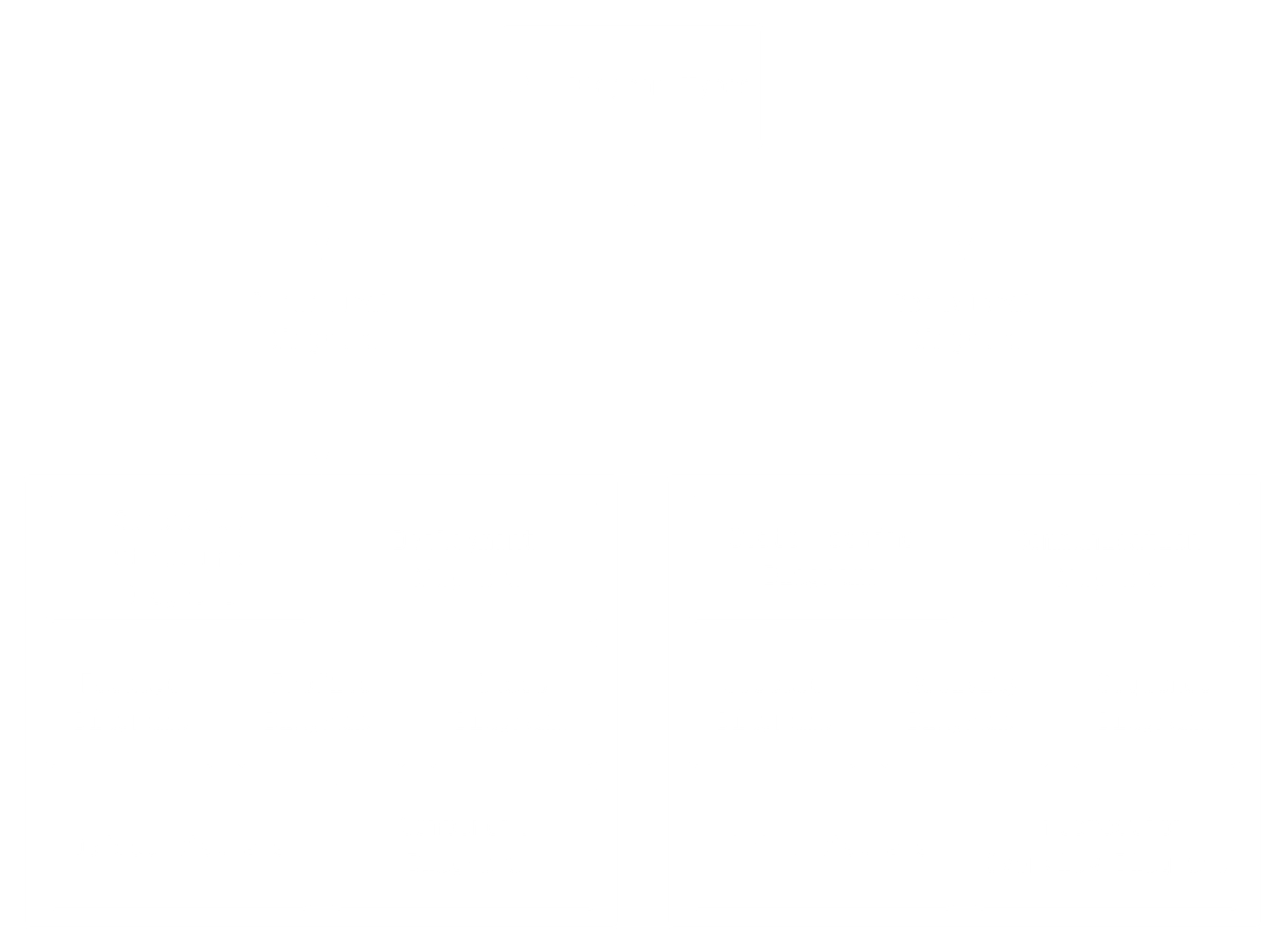
The industry looks for techniques to **automate** the production of software to **improve quality** and **reduce cost** and **time-to-market**. Businesses also seek techniques to manage the **complexity** of systems. UML was designed to do exactly this.

* It has an **open standard**, **graphical notation** for specifying, visualizing, constructing and documenting software systems.
* It **increases understanding** of the product for customers and developers.
* It supports **diverse application areas**.
* It is supported in many **software packages**.

## UML Diagram Categories

There are mainly two categories of UML diagrams:

1. Structure Diagrams
2. Behavioural Diagrams



## Use Case Diagrams

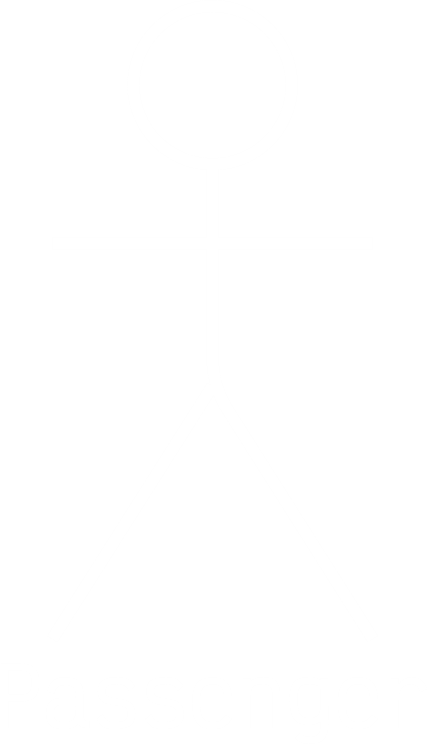
**Use Case Diagrams** are used during requirements elicitation to represent **external behaviour**.

It mainly consists of two things, actors and use cases. **Actors** are users of the system. These are not necessarily humans. They can be other systems as well. **Use cases** are sequences of interactions for a type of functionality.

The advantages of using use case diagrams are:

* Use case diagrams give a graphical overview of the **actors** involved in the system, different **functions** used by the actors and how functions **interact**.
* They make it easier to **determine new requirements**. New use cases often generate new requirements as the system is analysed and the design formed.
* They make it easier to **communicate** with clients, since they are so simple.
* They allow us to **generate test cases**. Test cases are usually created for each use case.

### Actors



An **actor** models an **external entity** which communicates with the system. This could be a user, an external system or a physical environment. An actor has a **unique name** and an optional **description**.

For example, a passenger on a train is an actor for the train system as is a GPS satellite that provides a system with GPS coordinates.

### Use Cases



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

* A **unique name**
* Participating **actors**
* **Entry** and **exit conditions**
* **Flow** of events
* **Special Requirements**

For example, take the PurchaseTicket use case.

Name: Purchase Ticket

Participating Actor: Passenger

Entry Condition: Passenger standing in front of ticket distributor.

Passenger has sufficient money to purchase a ticket.

Exit Condition: Passenger has a ticket.

Event Flow: 1. Passenger selects the number of zones to travel.

2. Distributor displays the amount due.

3. Passenger inserts money of at least due amount.

4. Distributor returns change.

5. Distributor issues ticket.

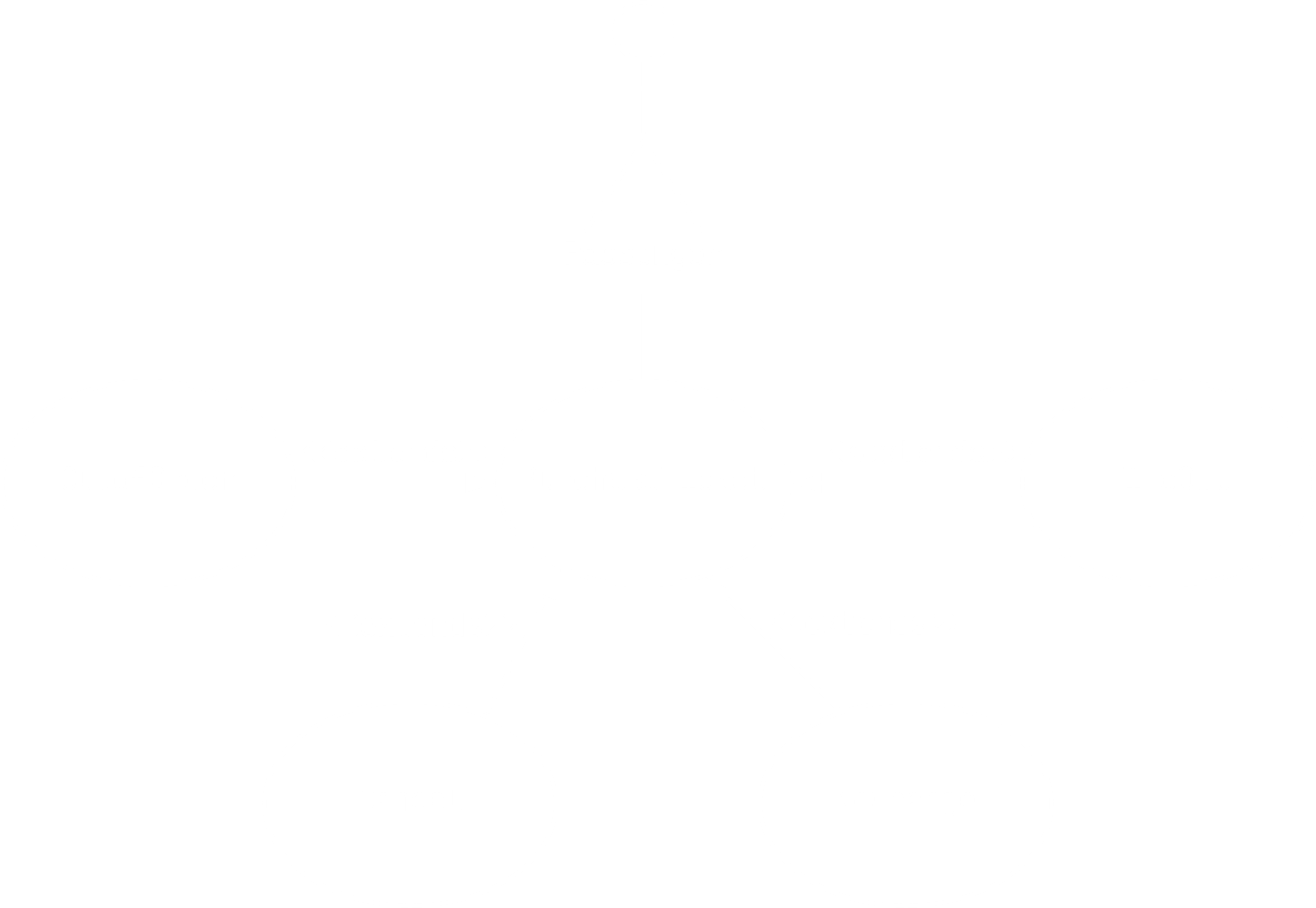
Exceptional Cases: 1. Insufficient tickets.

### <<extends>> and <<includes>> Relationships

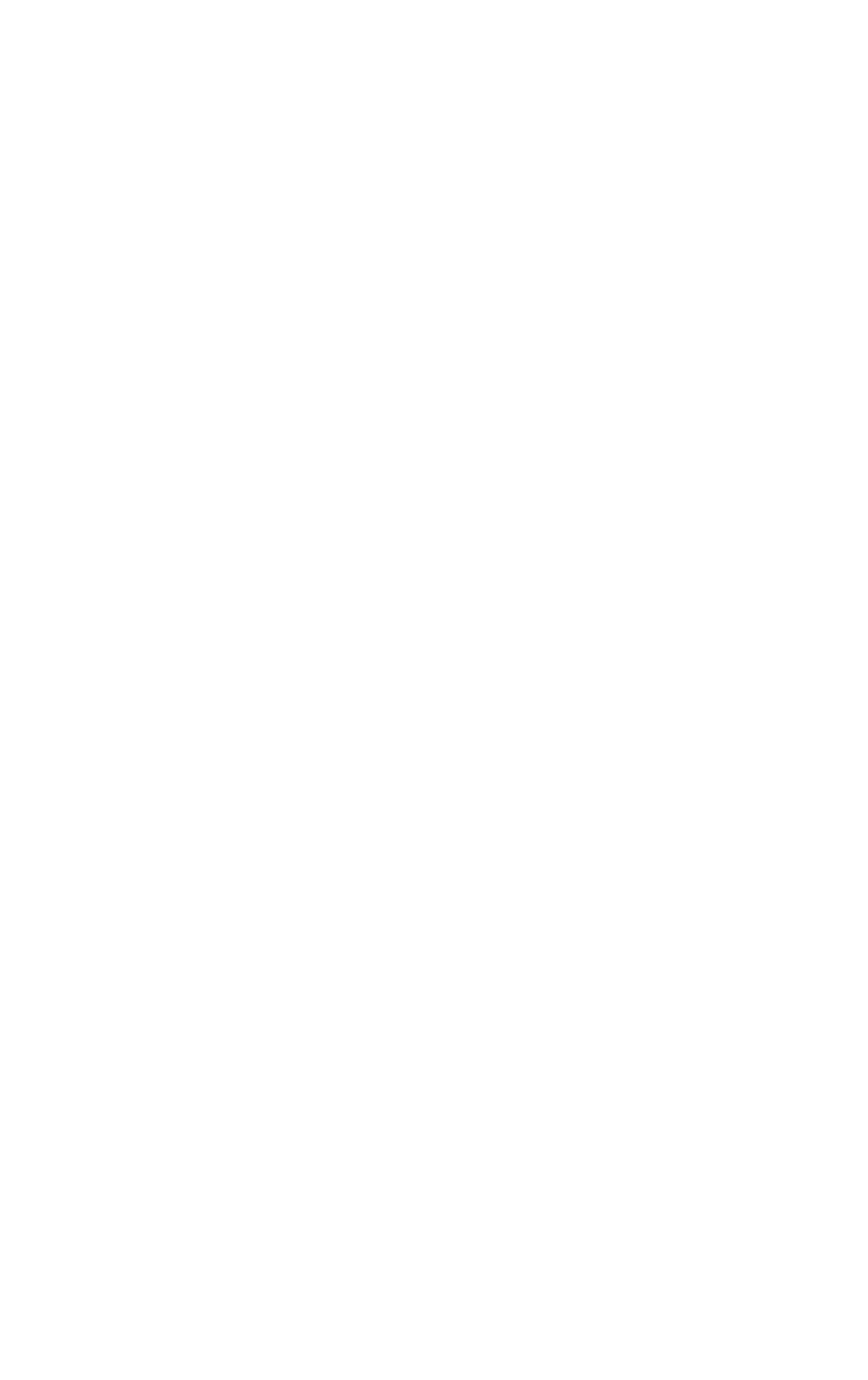
Use cases may have special relationships between them. These fall into one of two categories:

1. <<extends>> relationships
2. <<includes>> relationships

An **<<extends>>** relationship is one that is **seldom put to use**. These tend to deal with exceptional cases, so they are factored out of the main event flow for simplicity. <<extends>> relationships are represented by a directional arrow from the extended use case to the main use case.



An **<<includes>>** relationship is one that the main use case **cannot function without**. It is absolutely necessary. The only reason it is being factored out is so that it can be reused elsewhere. An <<includes>> relationship is represented by a directional arrow from the main use case to the extended use case.



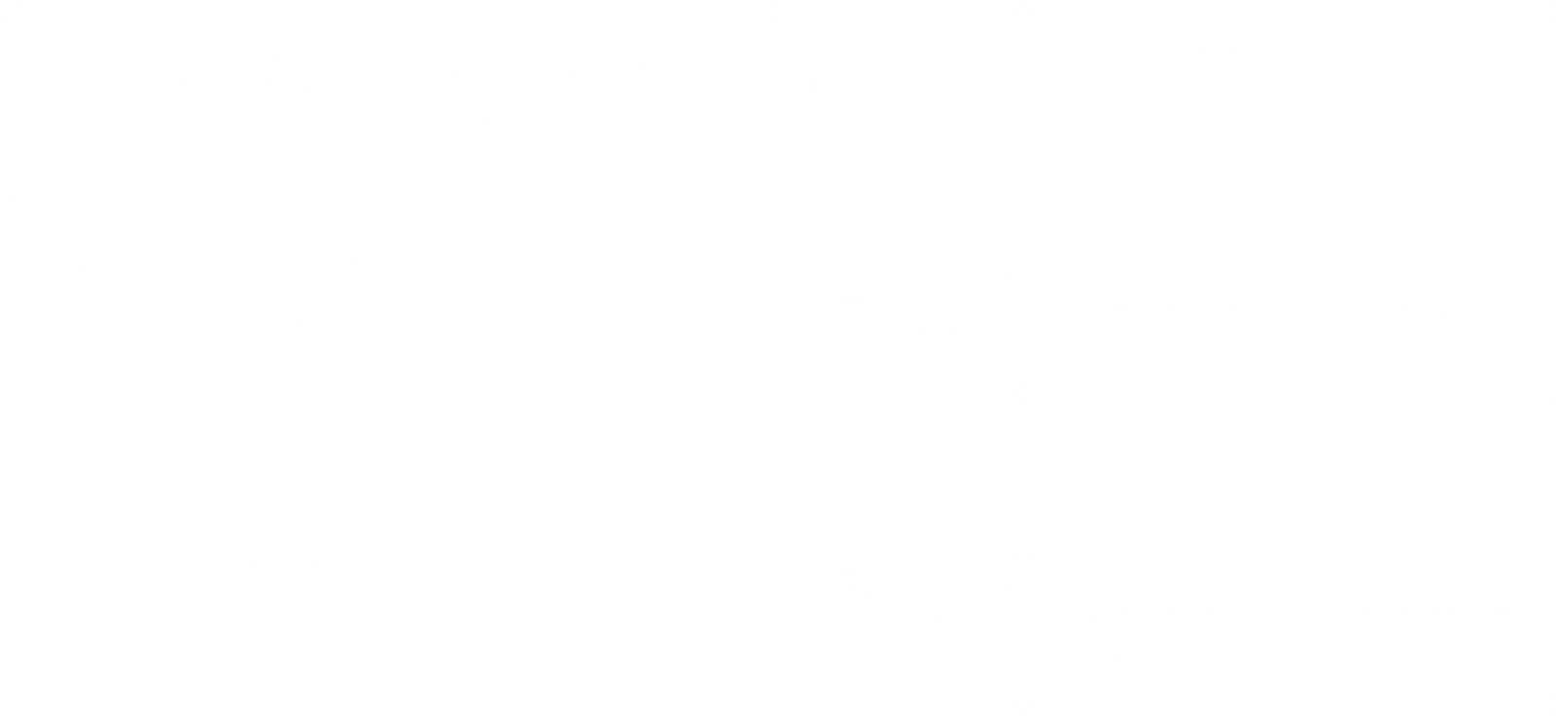
## Class Diagrams

**Class diagrams** give an overview of a system by showing its **classes** and the **relationships** among them. Class diagrams are **static**. They display what **interacts** but not what **happens** when they interact.

Class diagrams also show **attributes** and **operations** of each class. They are a good way to describe the overall **architecture** of the system components.

### Classes and Instances

A **class** represents a concept. It encapsulates **states** and **behaviours**. Each attribute has a type and each behaviour has a signature. The class **name** is the only mandatory information.



An **instance** represents a phenomenon. The name of an instance is underlined and may contain the class of the instance. The attributes are represented with their values.



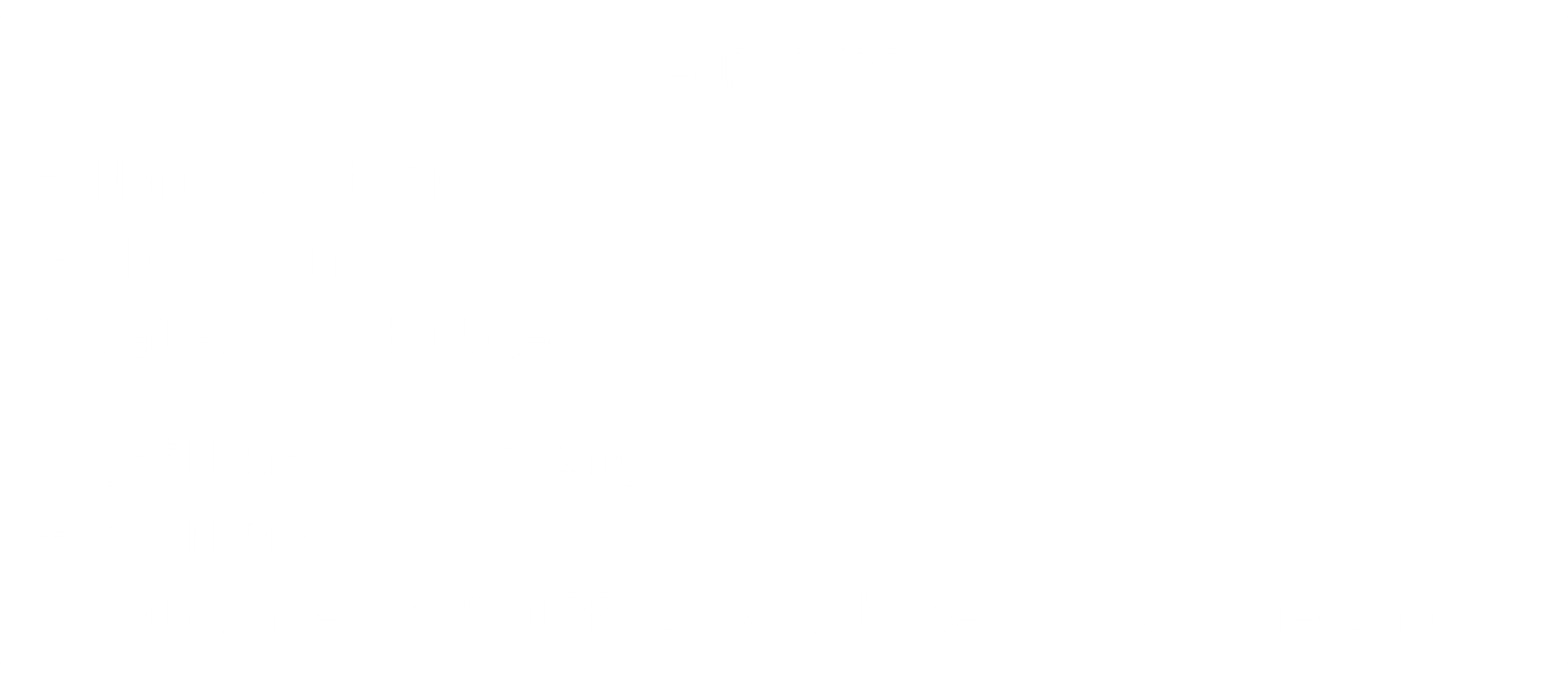
### Class Notation

**Classes** are denoted with **rectangles** divided into three parts:

1. The class **name**
2. The **attributes**
3. The **operations**

Each of these has a modifier:

* Private: -
* Public: +
* Protected: #
* Static: Underlined
* Abstract: Name in italics



## Relationships

In UML, **object interconnections**, both logical and physical, are modelled as **relationships**.

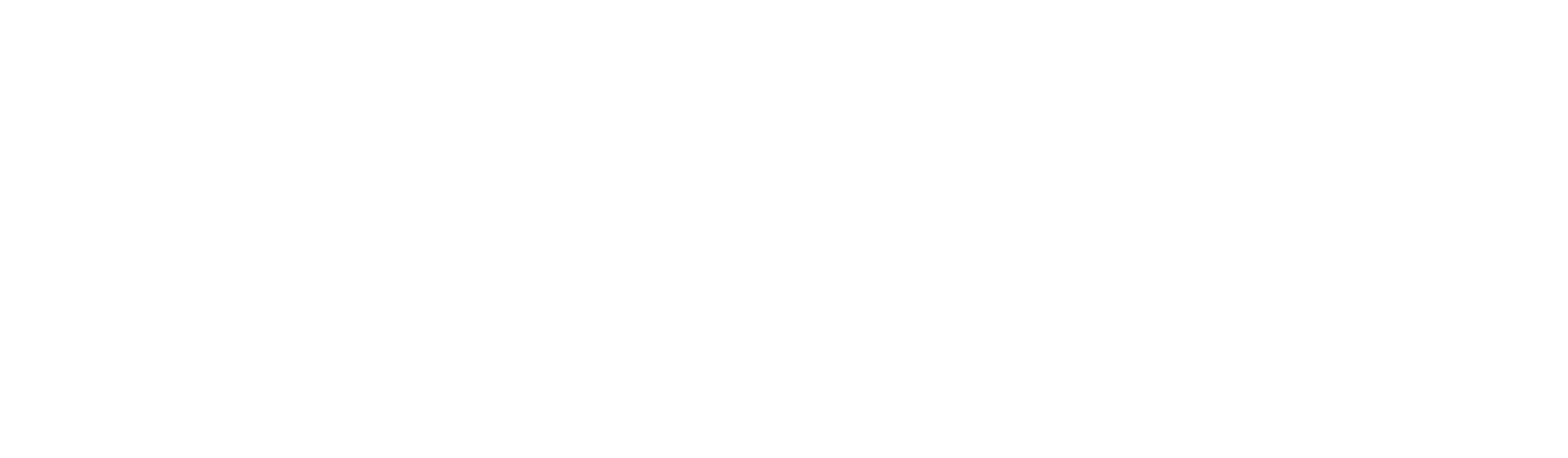
There are three kinds of relationships:

1. Dependencies
2. Generalizations
3. Associations

### Dependencies

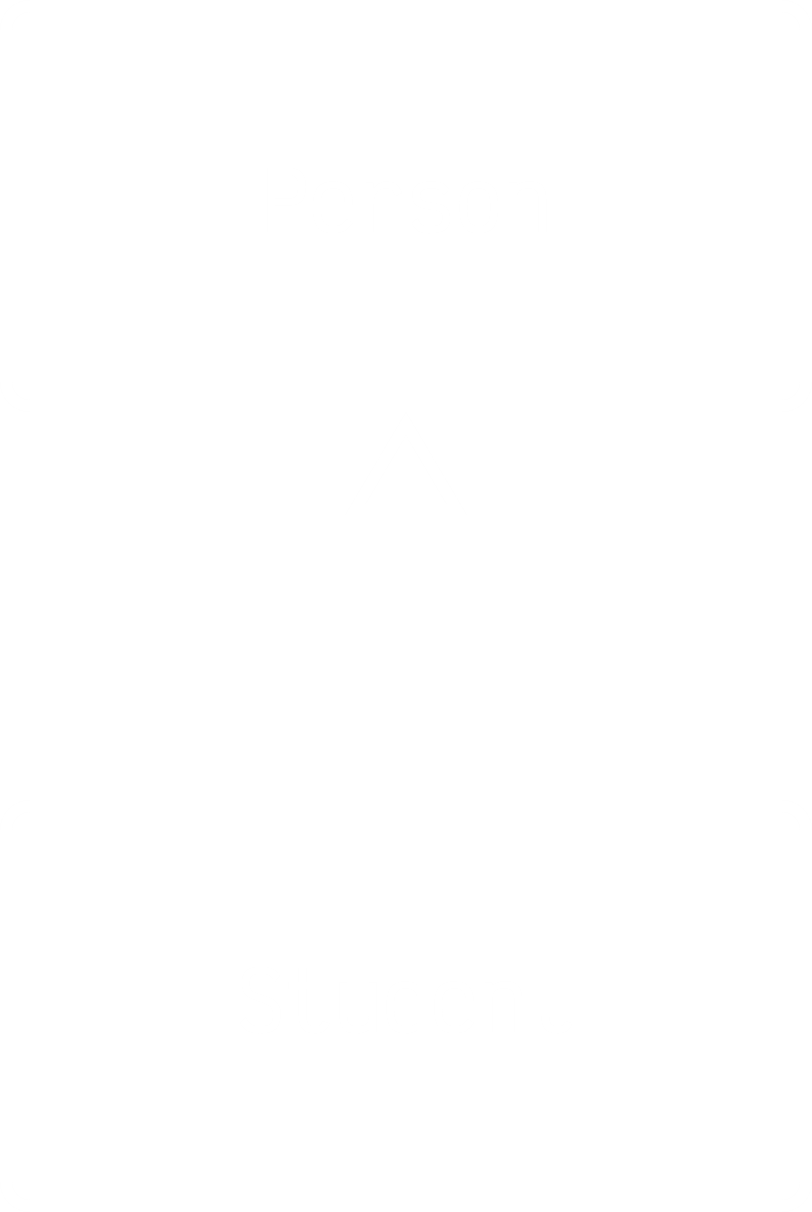
A **dependency** is when one element requires another element to be able to function.

In the diagram below, CourseSchedule has a Course object as a parameter in two different functions, so CourseSchedule is dependent on Course.

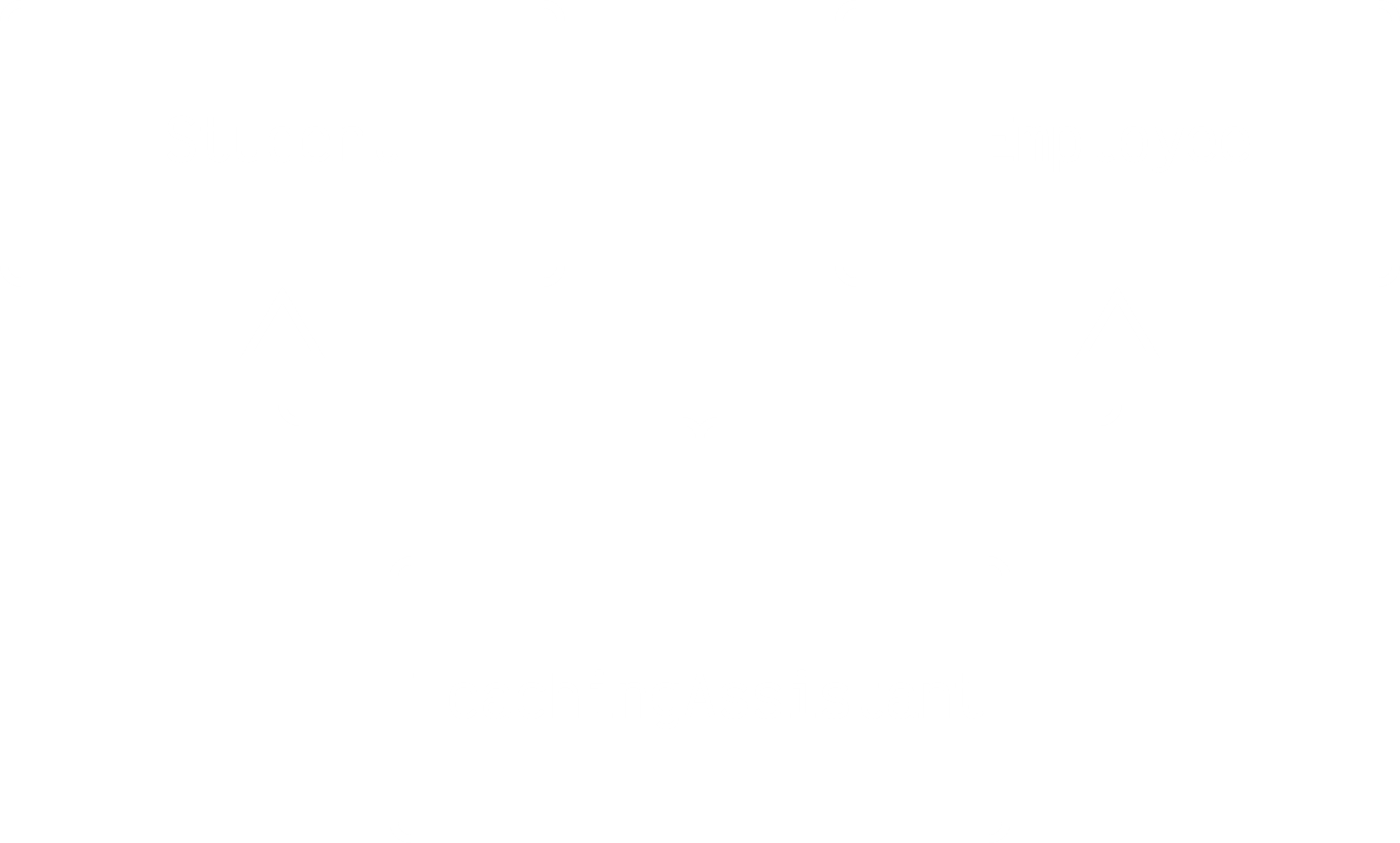


### Generalizations

A **generalization** connects a **subclass** to its **superclass**. It indicates an **inheritance** of attributes and behaviours from the superclass to the subclass and indicates a specialization in the subclass of the more general superclass.

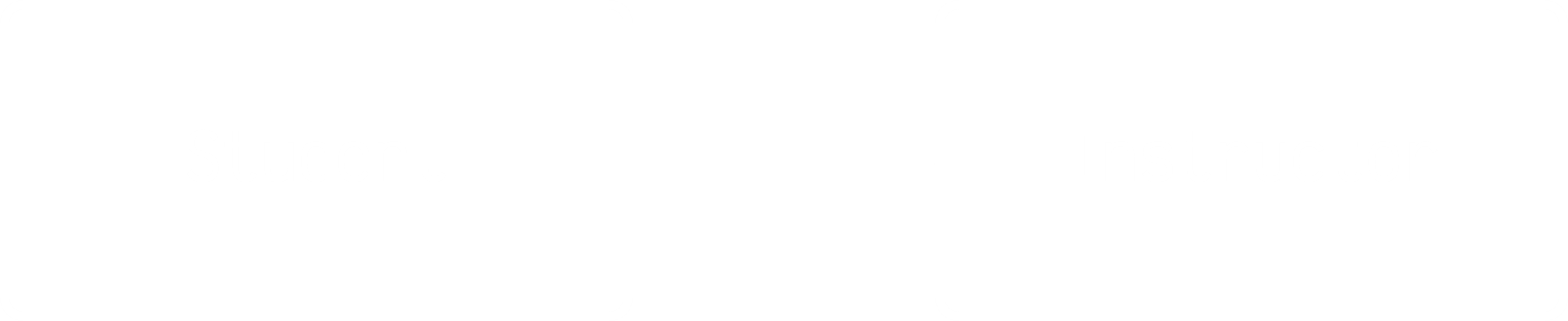


UML allows **multiple inheritance**, even though some programming languages do not.

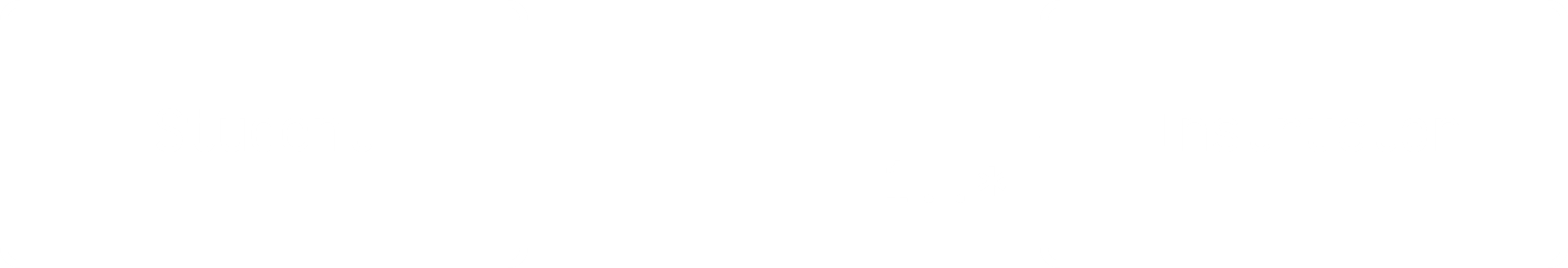


### Association

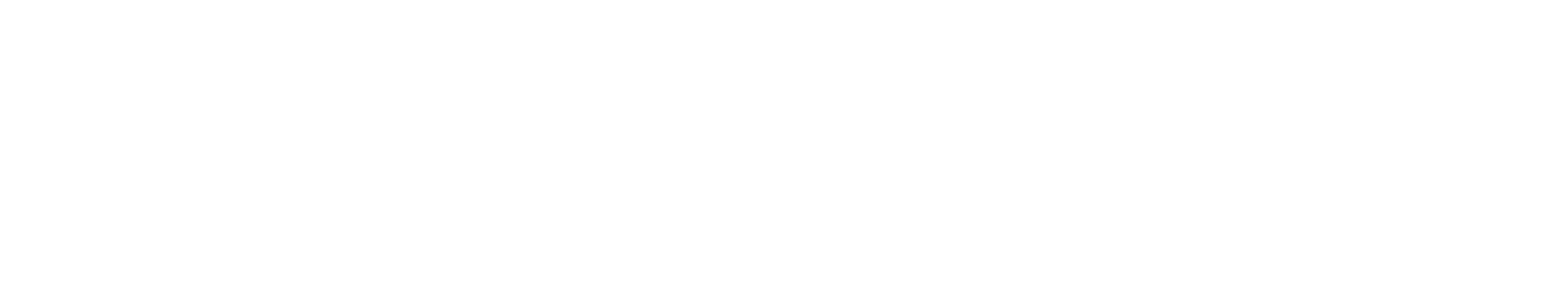
An **association** denotes two classes in a model that communicate with each other in some way.



We can indicate the **multiplicity** of an association by adding the multiplicities to the line denoting an association.



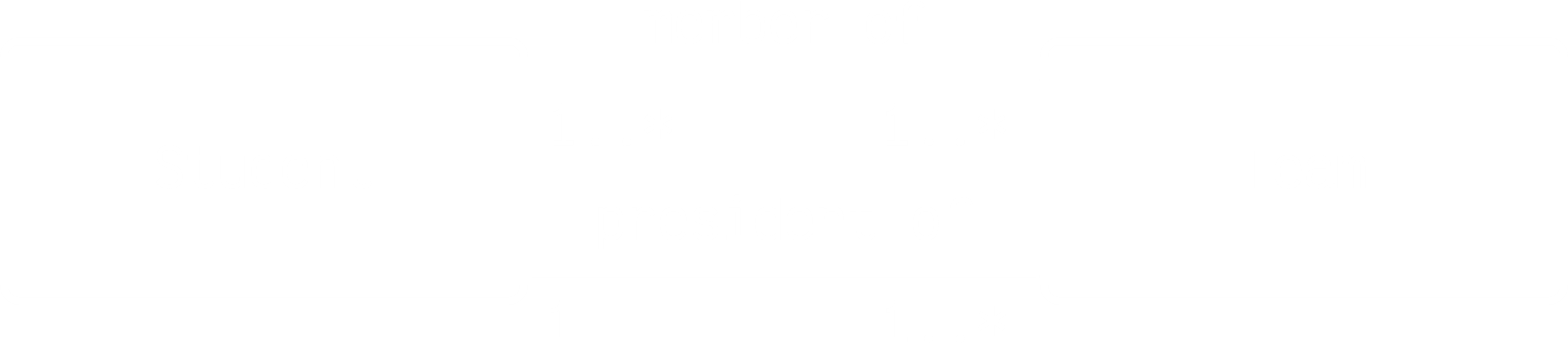
We can also indicate the behaviour or **role** of an object in the association using a **role name**.



The association can have a **name**.



We can even have **multiple associations** between classes.

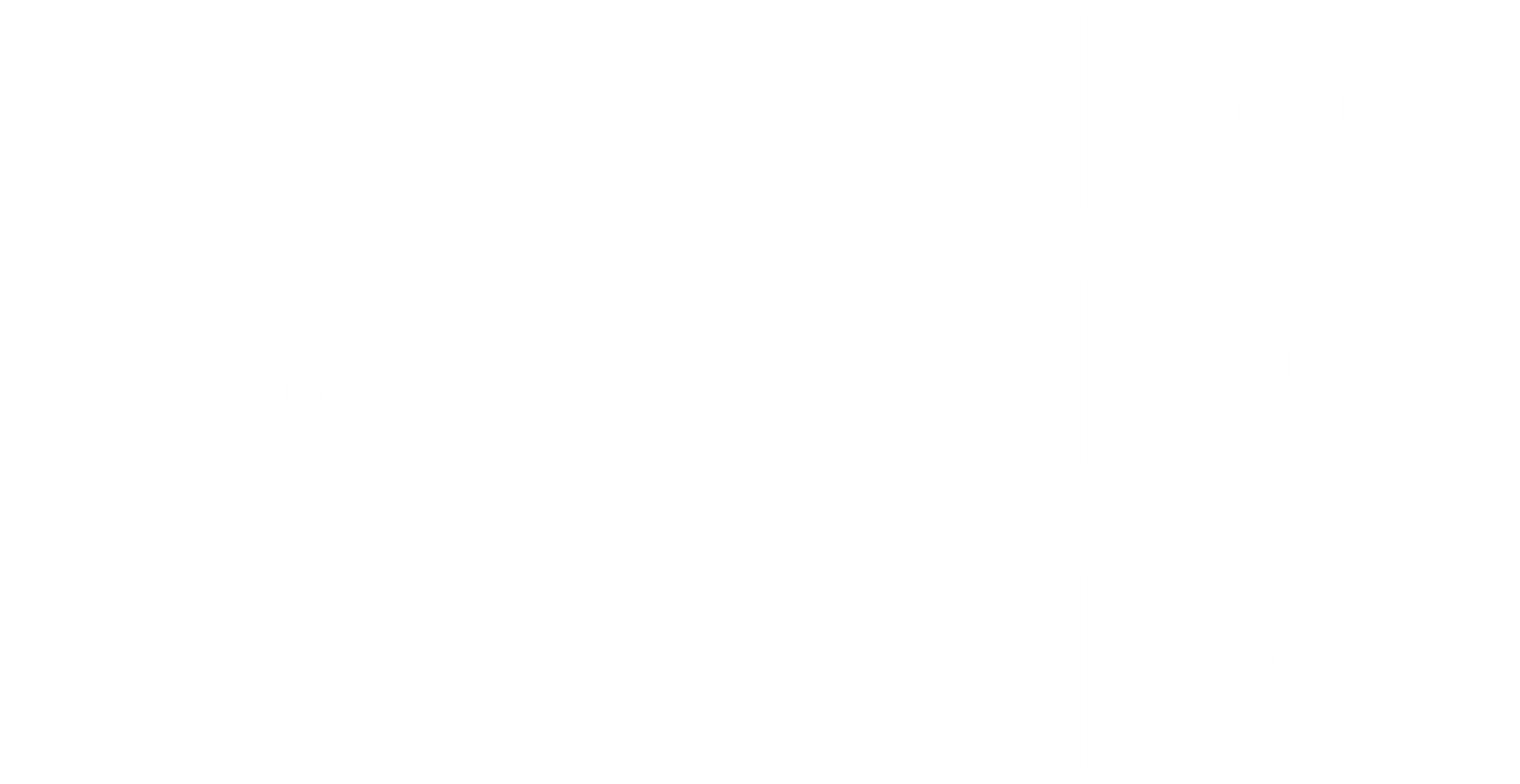


### Aggregations and Compositions

An **aggregation** is a special association which indicates that a **constituent part** can exist independently from the **aggregated whole**.



A **composition** indicates an **ownership**, where the **part** lives and dies with the **whole**.

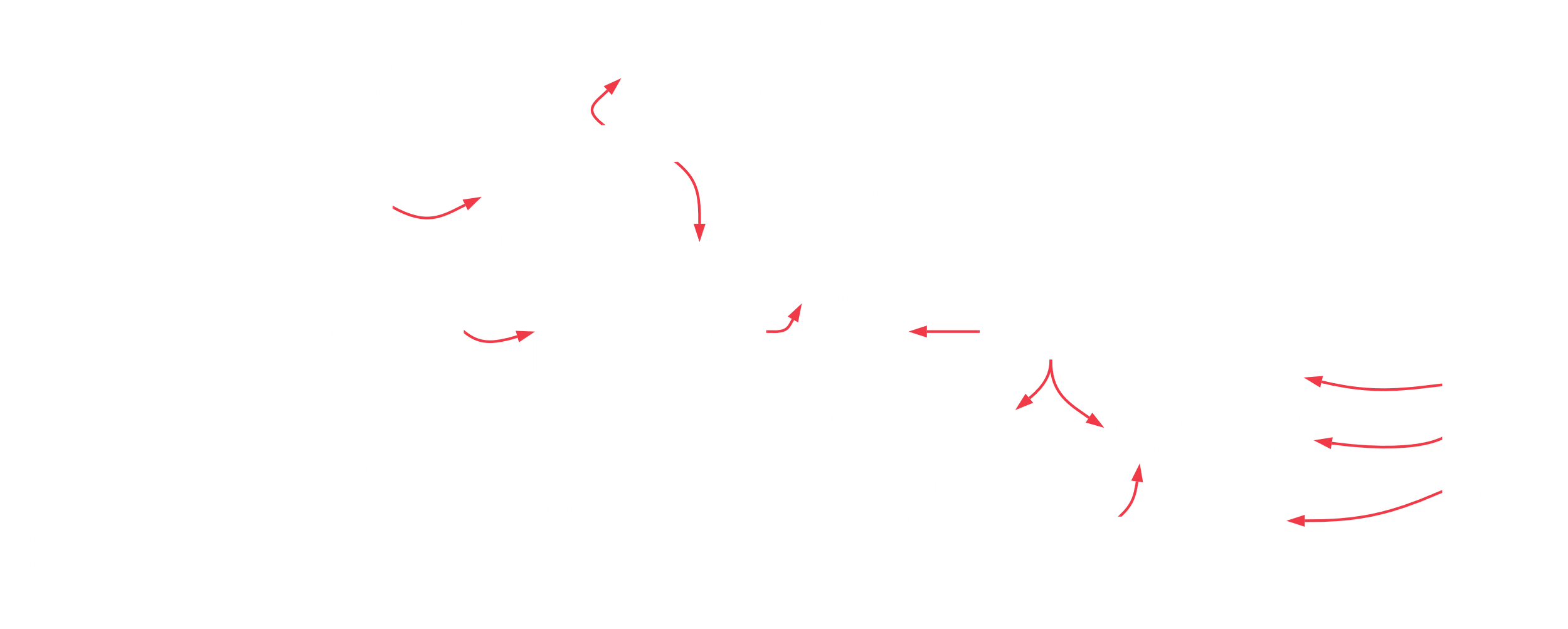


### Multiplicities

**Multiplicities** in associations can be of four types:

|  |  |  |  |
| --- | --- | --- | --- |
| **Multiplicities** | | | **Meaning** |
|  | | | Zero or one instance. The notation indicates to instances. |
|  | or |  | No limit on the number of instances (including none). |
|  | | | Exactly one instance. |
|  | | | At least one instance. |

Example



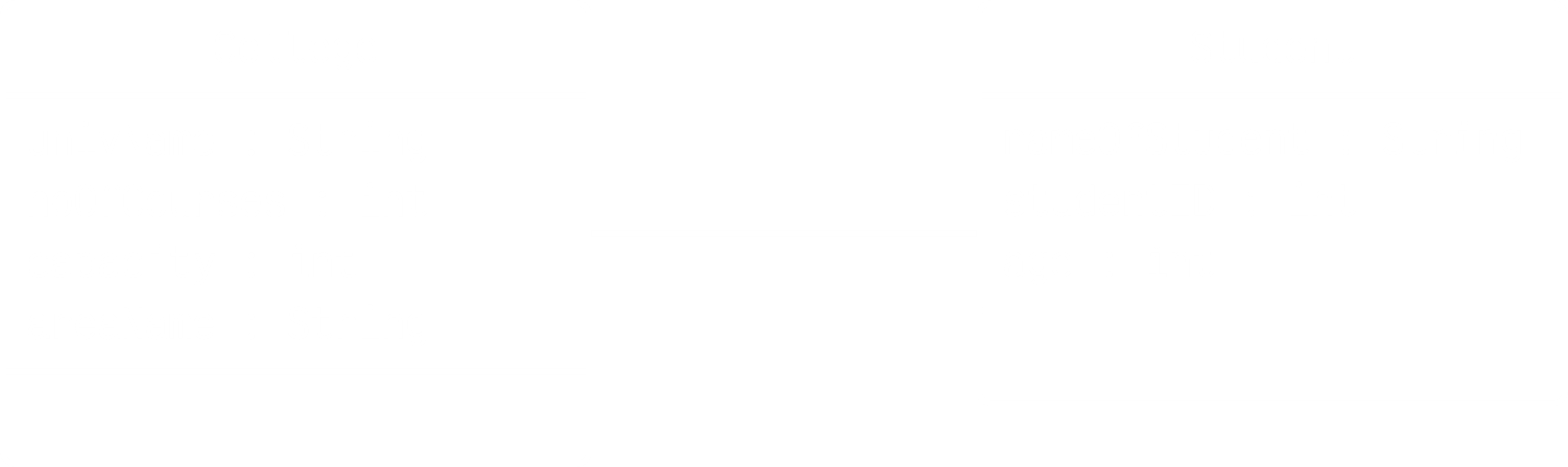
## Object Diagram

In a live application, we do not directly use classes, but rather **instances** or **objects** of the classes. A pictorial representation of the relationships between these objects at any point of time is called an **object diagram**.

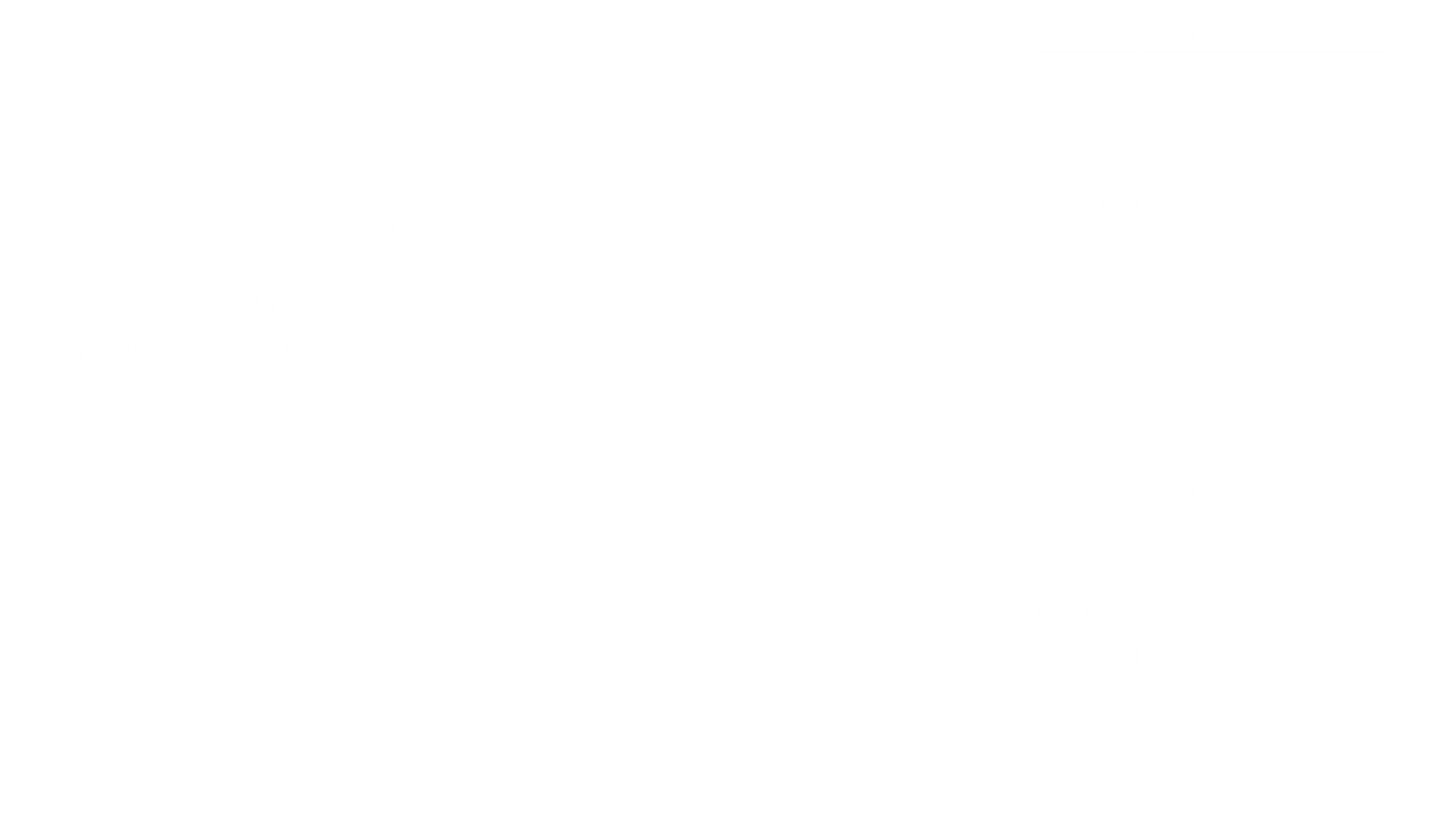
An object diagram is similar to a class diagram and uses similar notations for relationships. It reflects the picture of how classes interact with each other at **runtime** and, in the actual system, how objects created at runtime are related to classes.

The only difference is that a class diagram shows a class with **attributes** and **methods** without any values, while in the object diagram, they do have **values**.

For example, consider the following class diagram:



For this class diagram, we may get the following object diagram:



The object diagram also shows the **name** of the instantiated objects, separated from the class name by a ‘:’ and **underlined** to indicate instantiation.

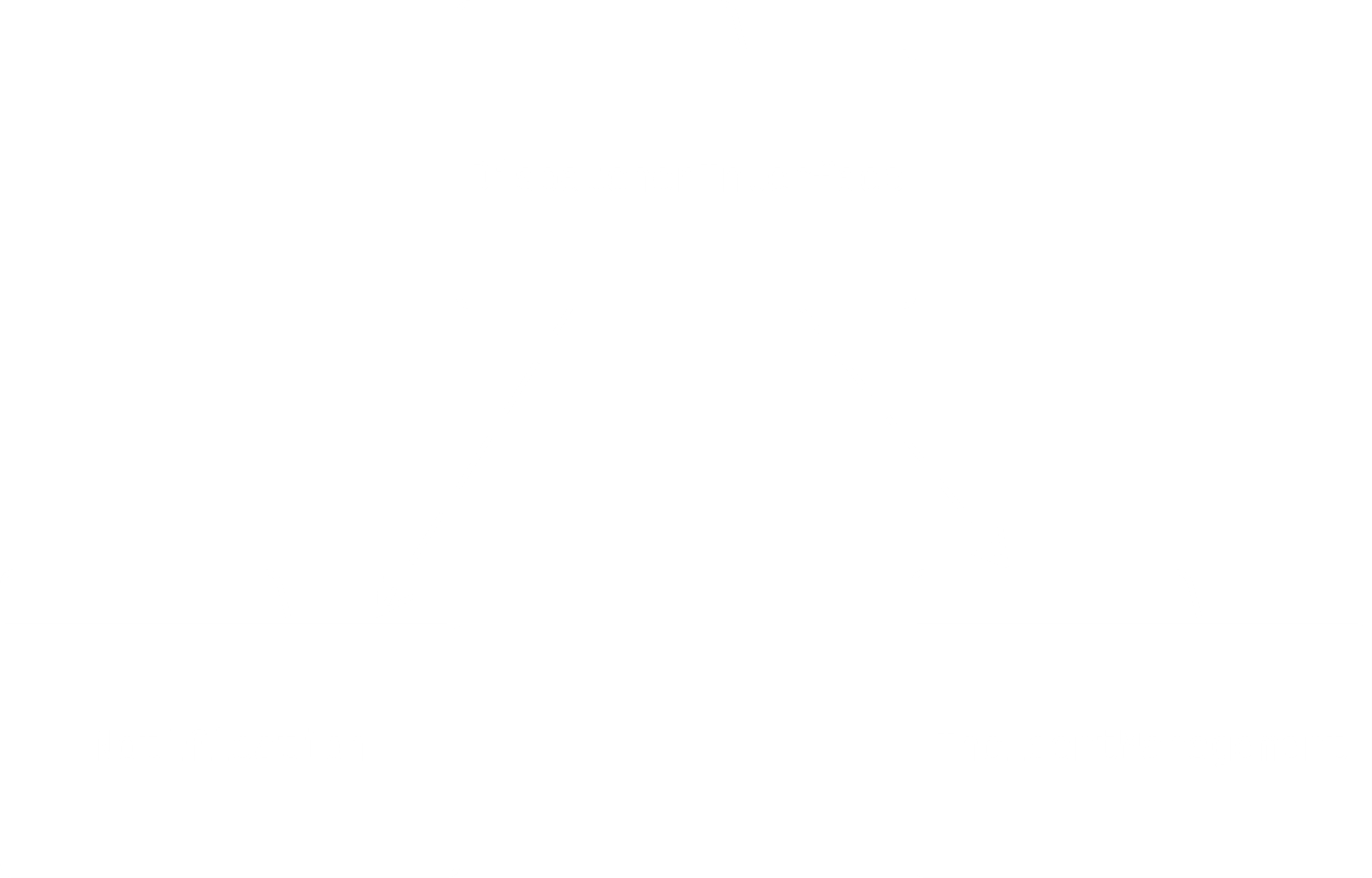
We should use object diagrams as a means of **debugging** the functionality of the system and **checking** whether the system has been designed as per the **requirements** and behaves how the business functionality needs the system to respond.

We should avoid representing all the objects of the system though, since things can quickly become complex.

## Package Diagram

To organize complex class diagrams, we can **group classes** into **packages**. Thus, a package is a collection of logically related UML elements.

Packages appear as **rectangles** with **small tabs** at the top. The package **name** is on the tab or inside the rectangle. **Dotted arrows** show dependencies, where if one package changes, it would force the other package to change as well.



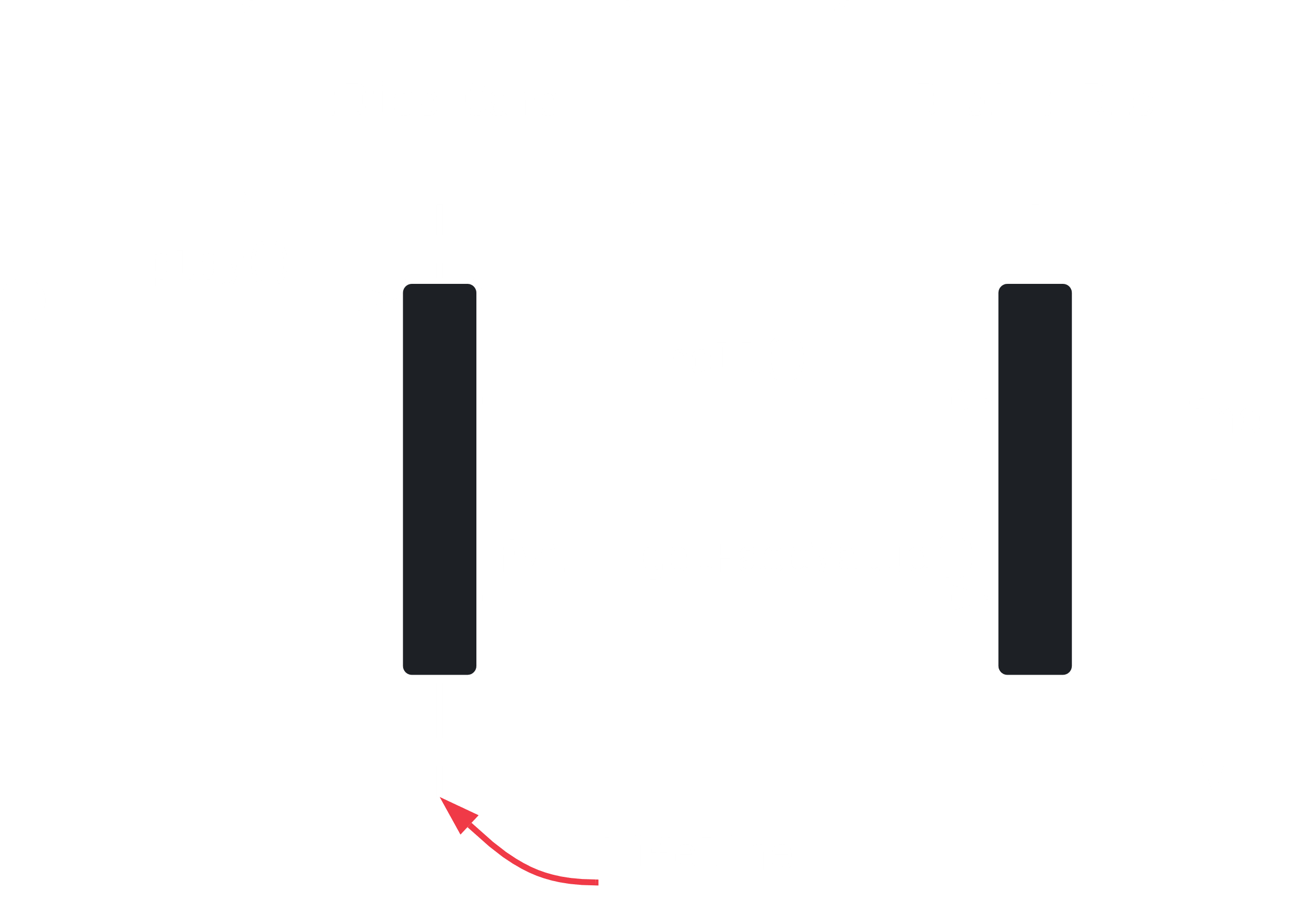
## Sequence Diagram

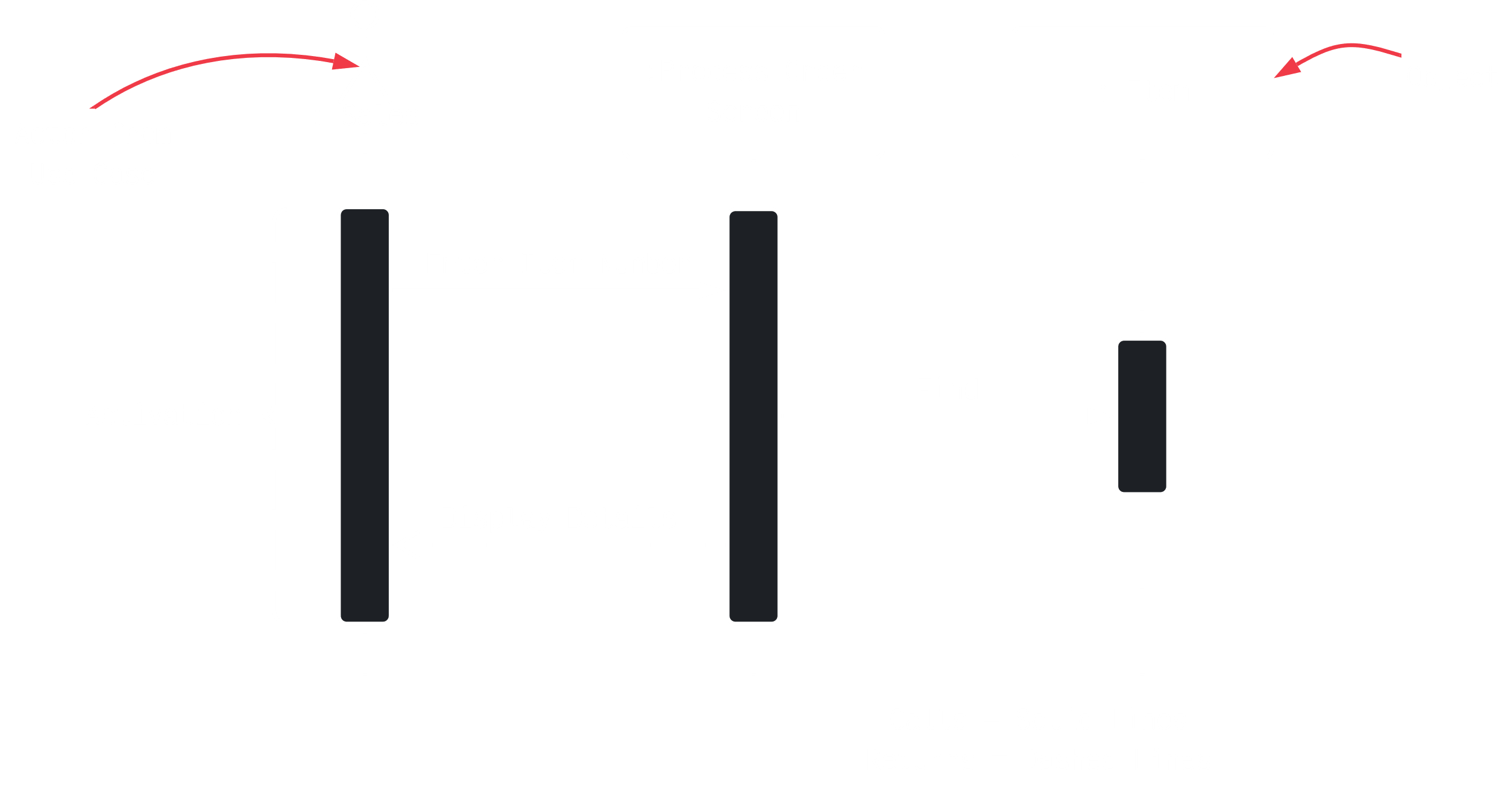
A **sequence diagram** captures the **behaviour** of a single scenario. It is very useful for the implementation state.

A sequence diagram shows a number of example objects, represented as rectangles with their class names, and the messages that are exchanged between these objects, represented as directional arrows. It depicts the sequence of **actions** that occur in the system.

A sequence diagram is **two-dimensional** in nature. On the horizontal axis, we have the life of the objects it represents, while on the vertical axis, it shows the sequence of the creation or invocations of these objects.

The different parts of a sequence diagram can be seen in the diagrams below:





**Lifelines** show the duration for which an object is alive, while the **activation bars** show the duration for which an object is actively taking part in the system, meaning an instance of the object exists.

### Message Arrows

A message has the following format:

attribute = message\_name(arguments) : return\_type

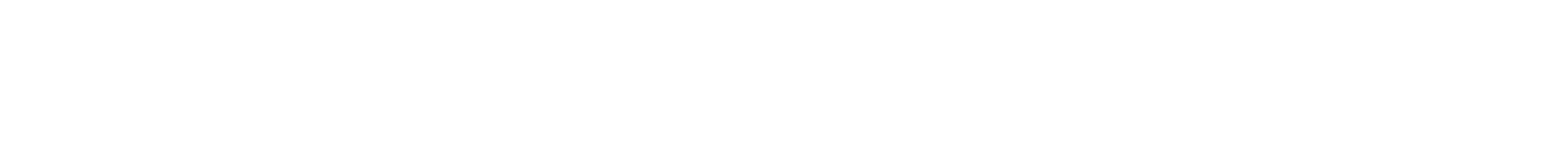
All parts except the message\_name are optional.

Message Arrows can actually be of several types:

1. **Synchronous** – These are messages where the sender waits for response before carrying on with other messages.



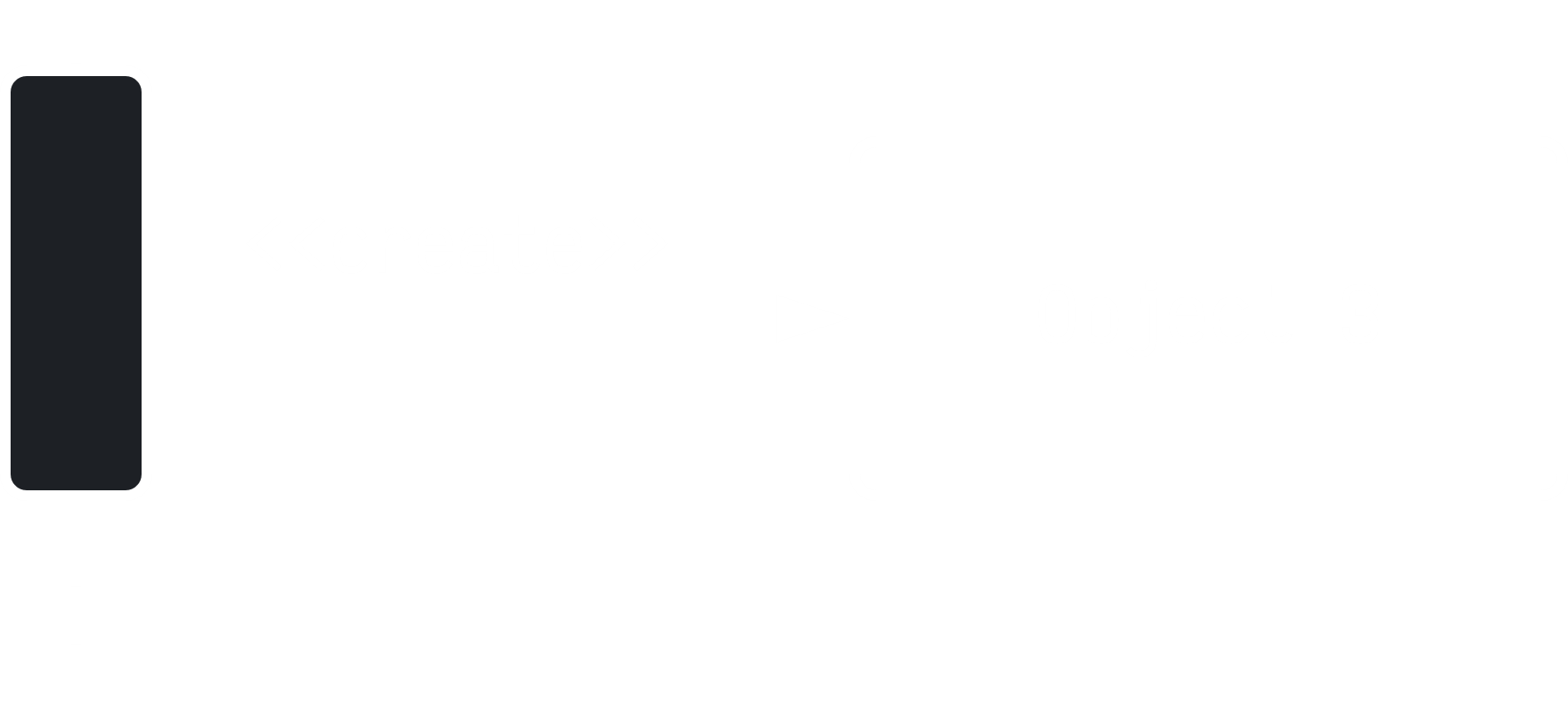
1. **Asynchronous** – These are messages where the sender does not wait for a response.



1. **Return** – These messages indicate that the message’s receiver is done processing the message and is returning control to the message caller.



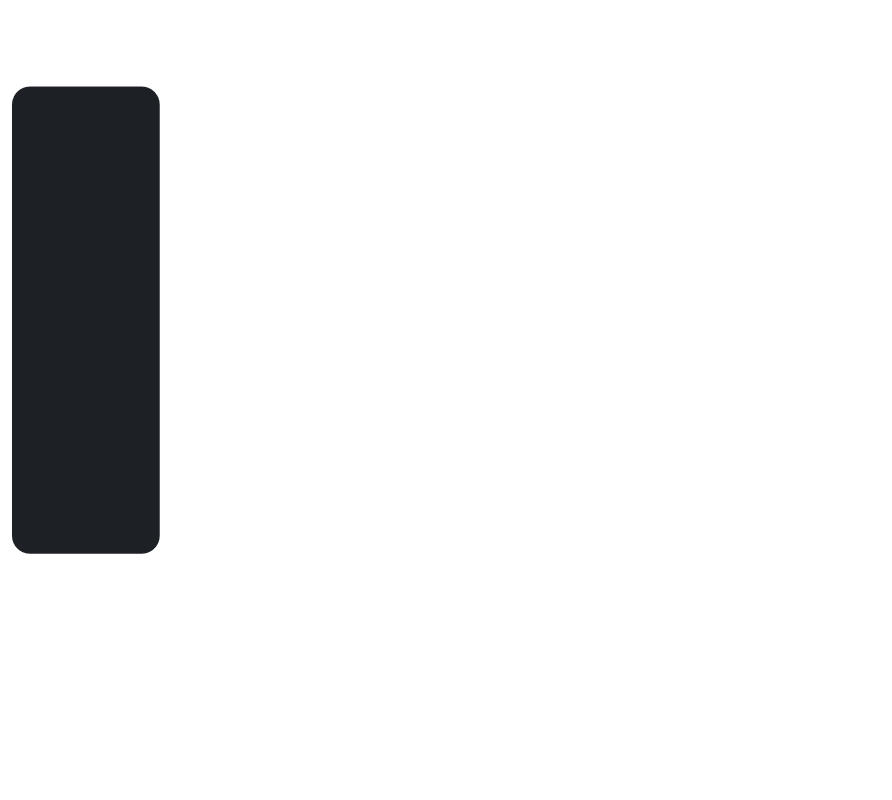
1. **Participant Creation** – Objects do not always live for the entire duration of the sequence. These messages are used to create objects.



1. **Participant Destruction** – These messages are used to destroy objects.

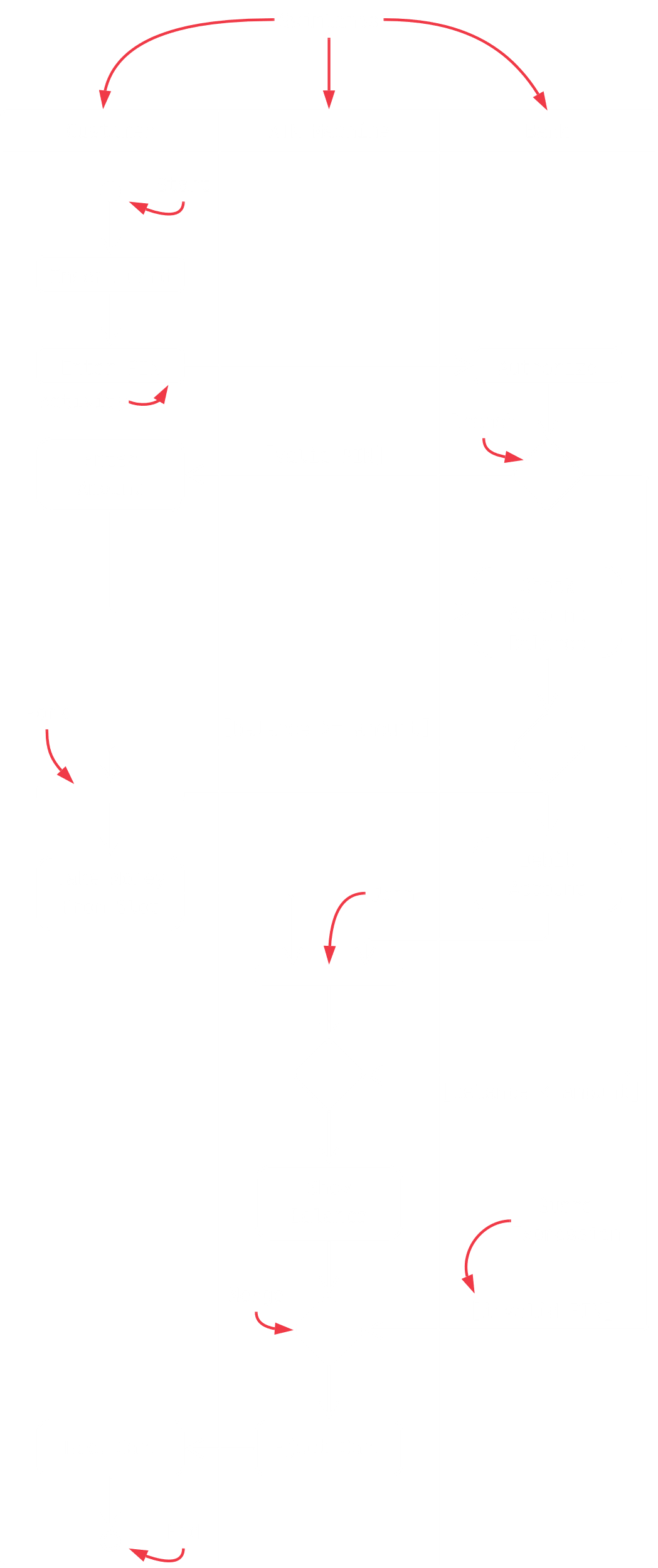


1. **Reflexive** – These are messages that an object sends to itself.



## Activity Diagrams

An **activity diagram** is just a fancy flowchart. It displays the **flow of activities** involved in a single process.

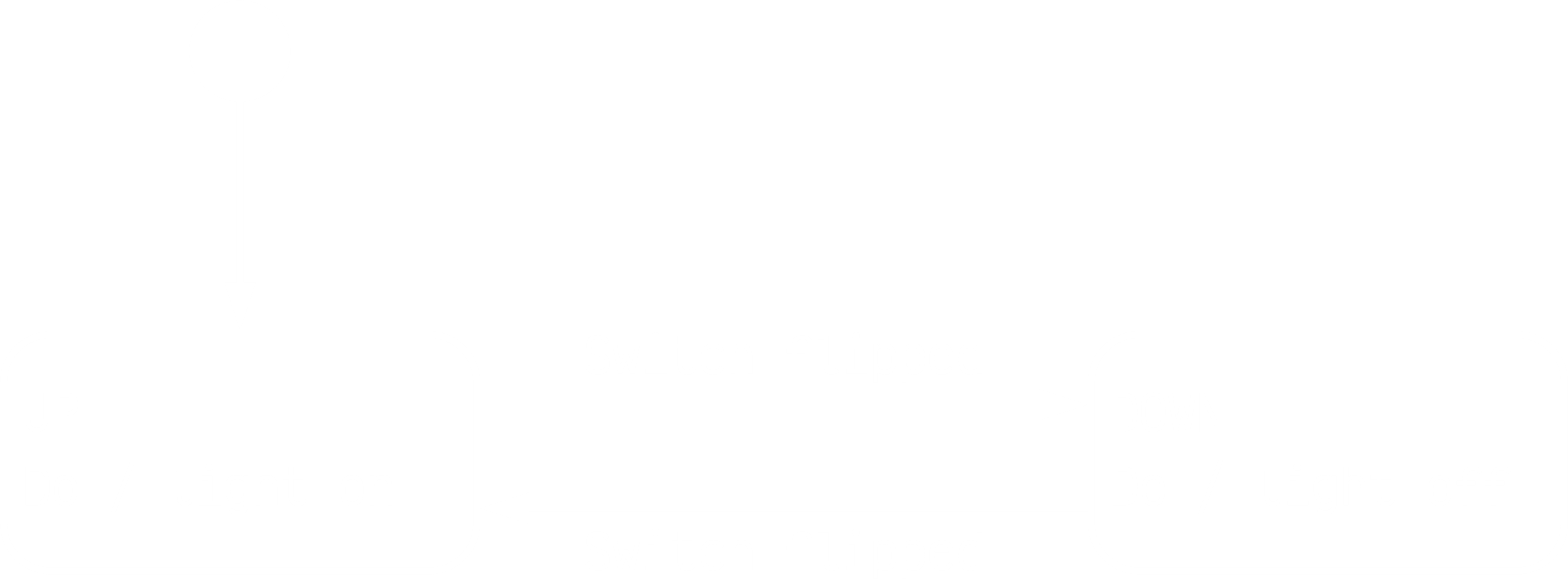


The parts of an activity diagram are:

* **States** – These describe what is being processed. They are indicated by boxes with rounded corners.
* **Swim Lanes** – These indicate which object is responsible for which activity.
* **Branch** – These are for transitions that branch. They are indicated by a diamond.
* **Fork** – These are for transitions that fork into parallel activities. They are indicated by solid bars.
* **Start** and **End**

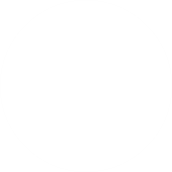
## State Diagram

**State diagrams** are used to describe the **behaviour** of the system. They describe all possible **states** of an **object** as an event occurs.



A state diagram includes the following parts:

* **Initial State** – This is the starting point of a state diagram. It is denoted as a black circle.



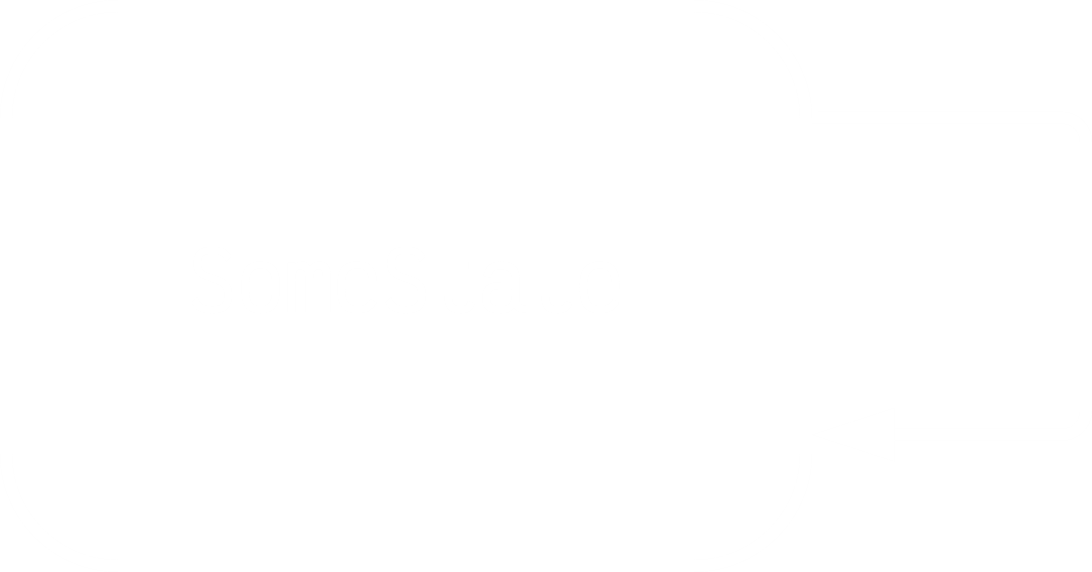
* **State** – These represent the state of an object at an instance of time. It is denoted as a rounded rectangle.



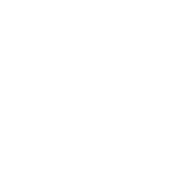
* **Transition** – These are arrows that indicate an object is transitioning from one state to another. The trigger event is shown above the arrow.



* **Self-Transitions** – Sometimes, an object may be required to perform some action when it recognizes an event, but it ends up in the same state.

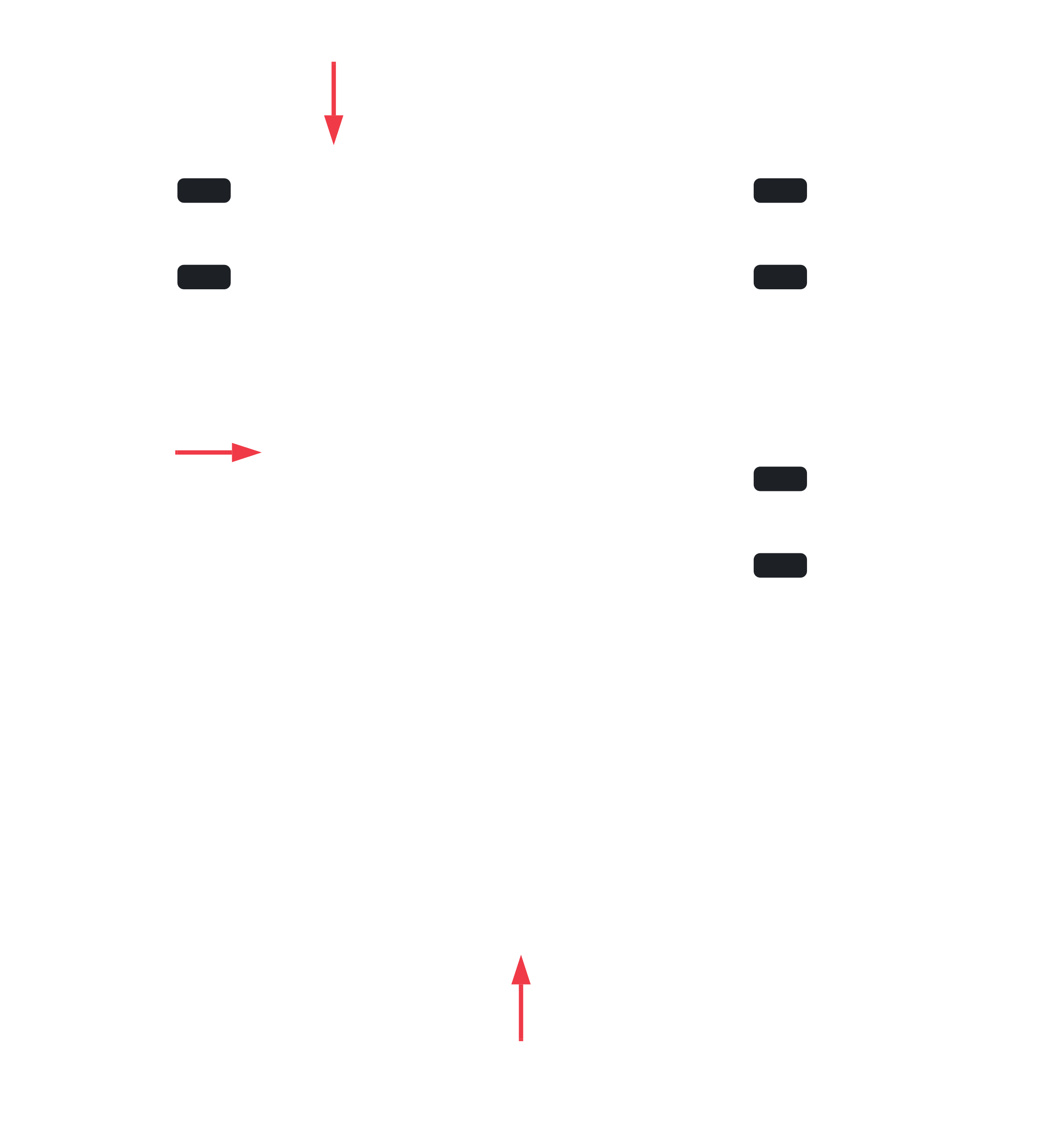


* **Final State** – This is the end of the state diagram. It is indicated by a bull’s eye symbol.



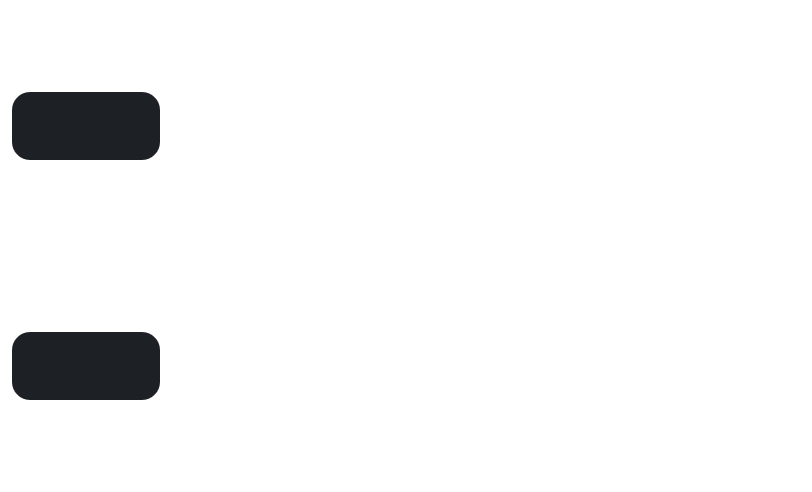
## Component Diagrams

A **component diagram** displays the structural relationship of the components in a software system. These are mostly used when working with complex systems with many components. Components communicate with each other using **interfaces**. The interfaces are linked using **connectors**.

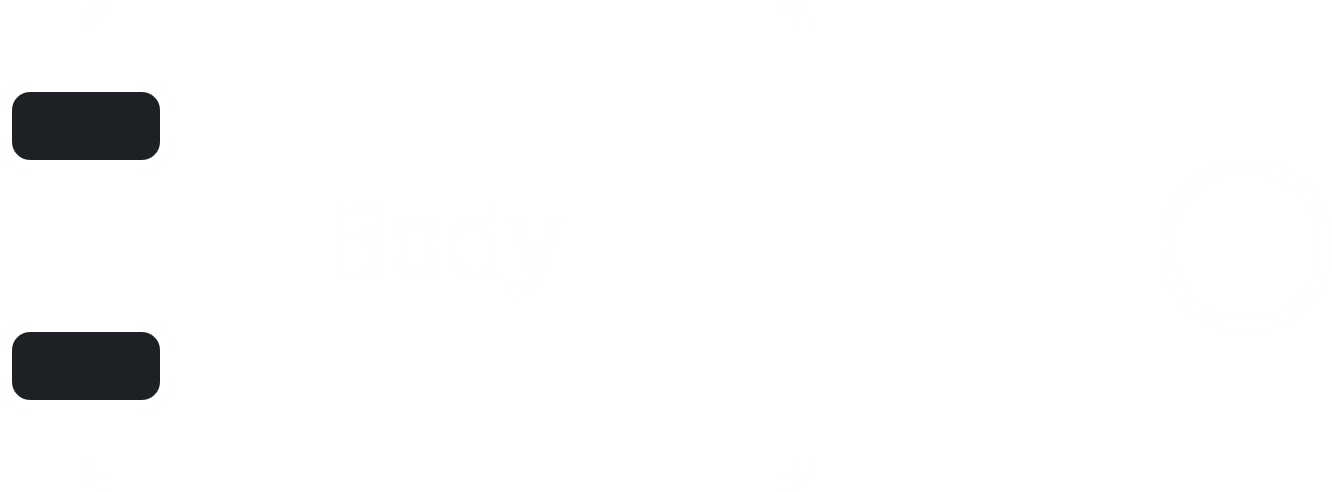


A component diagram has the following parts:

* **Component** – These are shown using rectangles with two bars to the upper left.



* **Dependencies** – These are shown with dashed arrows.
* **Interfaces** – These are shown as a circle with a solid line connected to the component.

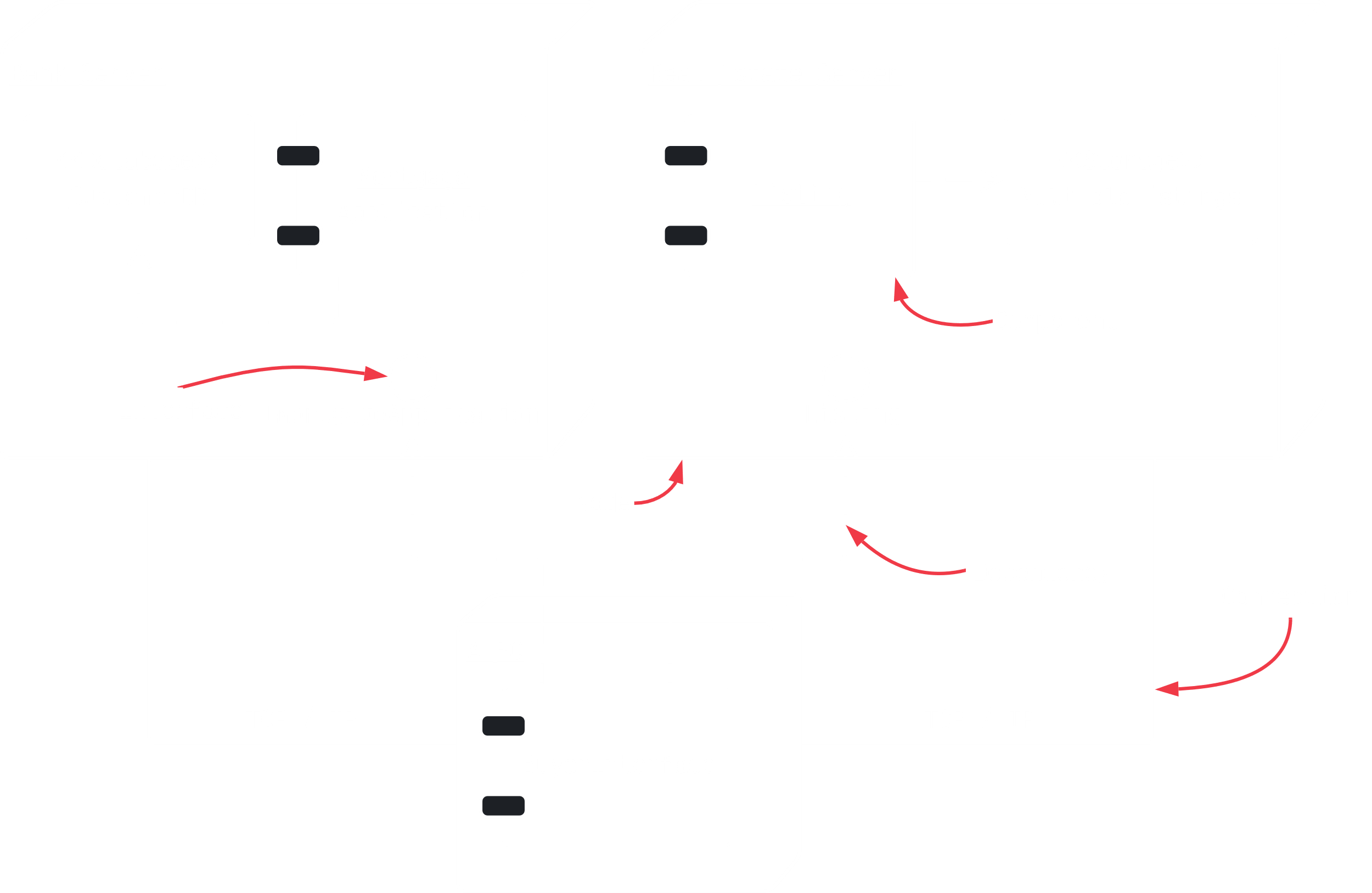


A component diagram is very similar to a package diagram.

## Deployment Diagram

A **deployment diagram** shows the physical architecture of the hardware and software of the **deployed system**. There are **nodes**, which typically contain components, packages or some kind of computational unit, such as a machine or device. The **physical relationship** between software and hardware in the delivered system can explain how the system interacts with the external environment.

Deployment diagrams are usually useful when the software is deployed across **multiple machines**, each with its own, unique configuration.



In UML, a deployment diagram models the **physical deployment** of artifacts on nodes. For example, for a website, we would show hardware components as nodes, what software components (artifacts) run on each node and how the different pieces are connected.

The nodes appear as **boxes** and the artifacts allocated to each node appear as **rectangles** within the boxes. Nodes can have **subnodes**, which appear as nested boxes. A single node may represent multiple physical nodes.

There are two types of nodes:

1. Device Node
2. Execution Environment Node