**Chapter 10: Object-Oriented Systems Analysis and Design Using UML**

Table of Contents

[Commonly Used UML Diagrams 4](#_Toc63530164)

[Use Case Modelling 6](#_Toc63530165)

[Activity Diagrams 13](#_Toc63530166)

[Sequence Diagrams 18](#_Toc63530167)

[Class Diagrams 22](#_Toc63530168)

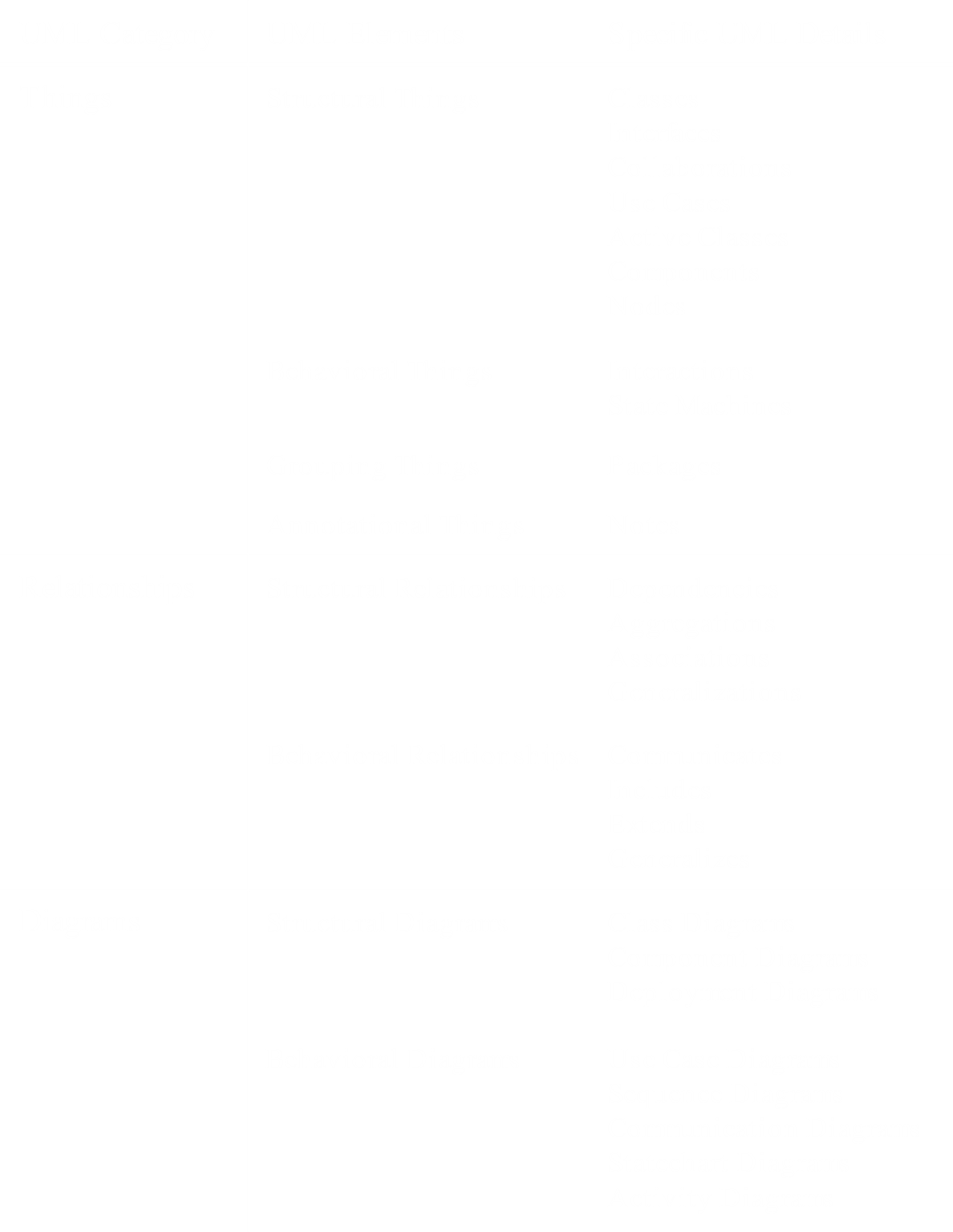
The Unified Modelling Language (UML) is an industry standard for modelling object-oriented systems. We will not be focusing on object-oriented systems for now, since it is assumed that the topic is well understood from previous courses. Instead, we will jump directly into the details of UMLs.

UML is a graphical representation of the system. They are widely used to visualize and document software systems designs. As such, it allows communication between the development team and the business team, since the latter will be unlikely to understand the technical terms commonly used by the former.

UML diagrams consist of three main components:

* Things – These are elements in the diagram. It includes classes, nodes, packages, etc.
* Relationships – These are the involvement of two or more things, such as through aggregation, association, dependencies etc.
* Diagrams – These are collections of components and include class diagrams, use case diagrams, activity diagrams etc.

A complete list of all components and the categories they fall into is given below:



## Commonly Used UML Diagrams

A few common UML diagrams are:

* Use Case Diagrams – These are the starting points of UML modelling. They describe how the system is used.
* Use Case Scenarios – These are descriptions of exceptions to the main behaviour described by the primary use case.
* Activity Diagrams – These show the overall flow of activities.
* Sequence Diagrams – These show the sequence of activities and class relationships. Sequences are not a concern for the other diagrams.
* Class Diagrams – These show classes and relationships. They are similar to ER diagrams.
* Statechart Diagrams – These show state transitions.

An overview of the entire system is given below:



Essentially, each diagram leads to the development of other UML diagrams.

### Use Case Modelling

Use cases describe what the system does, without describing how the system does it. For example, it might show that a facial recognition system is present, but it will not give any details about how that system works.

In a use case, an ‘actor’ using the system initiates an event that begins a related series of interactions in the system. Use cases are thus used to document single transactions or events.

Use case modelling is based on the interactions and relationships of individual use cases. It describes the system’s actions from the point of view of a user, and thus allows us to perform requirement analysis. Someone looking at a use case diagram will be able to point out any features that they think is unnecessary or may make requests for additional features to be included.

There are essentially three types of use case models:

* Textual or Tabular Descriptions – Rather than using a diagram, the different steps of an event are given as a description.
* Diagrams – An event and the steps involved are shown graphically, using a diagram.
* Informal Use Cases – These are also descriptions, but are more like brief overviews instead of detailed descriptions. Informal use cases are less commonly used.

#### Definitions

Before we can begin discussing details of how to create use cases, we need to be familiarized with a few definitions.

* Actors – Actors are things with a behaviour or role. These do not necessarily have to be people. They initiate an event. They are similar to the external entities that we saw in context diagrams.
* Scenarios – These are specific sequences of actions and interactions between actors and the system used to accomplish a particular goal. Essentially, a scenario is a use case instance.

Say a user is ordering something on a website. They first have to put something in the cart, then go to the cart and checkout, provide payment details and finally the object will be delivered to them. These are the series of steps for this scenario.

* Use Cases – A use case is a collection of related scenarios, describing how actors use the system to accomplish a goal.

#### Textual or Tabular Descriptions

**Use Case Title**: Pay for a job posting

**Primary Actor**: Recruiter

**Level**: Actor goal

**Precondition**: The job information has been entered, but is not viewable.

**Minimal Guarantees**: None

**Success Guarantees:** Job is posted; recruiter’s credit card is charged.

**Main Success Scenario**:

1. Recruiter submits credit card number, date and authentication information.
2. System validates credit card.
3. System charges credit card full amount.
4. Job posting is made viewable to Job Seekers.
5. Recruiter is given a unique confirmation number.

**Extensions**:

2a: The card is not of a type accepted by the system.

2a1: The system notifies the user to use a different card.

2b: The card is expired.

2b1: The system notifies the user to use a different card.

2c: The card is expired.

2c1: The system notifies the user to use a different card.

3a: The card has insufficient available credit to post the ad.

3a1: The system charges as much as it can to the current credit card.

3a2: The user is told about the problem and asked to enter a second credit card for the remaining charge. The user case continues as Step 2.

The above is an example for a textual description of a use case. It is for a website where recruiters can pay to put up advertisements for available jobs.

A few major components of the above description are:

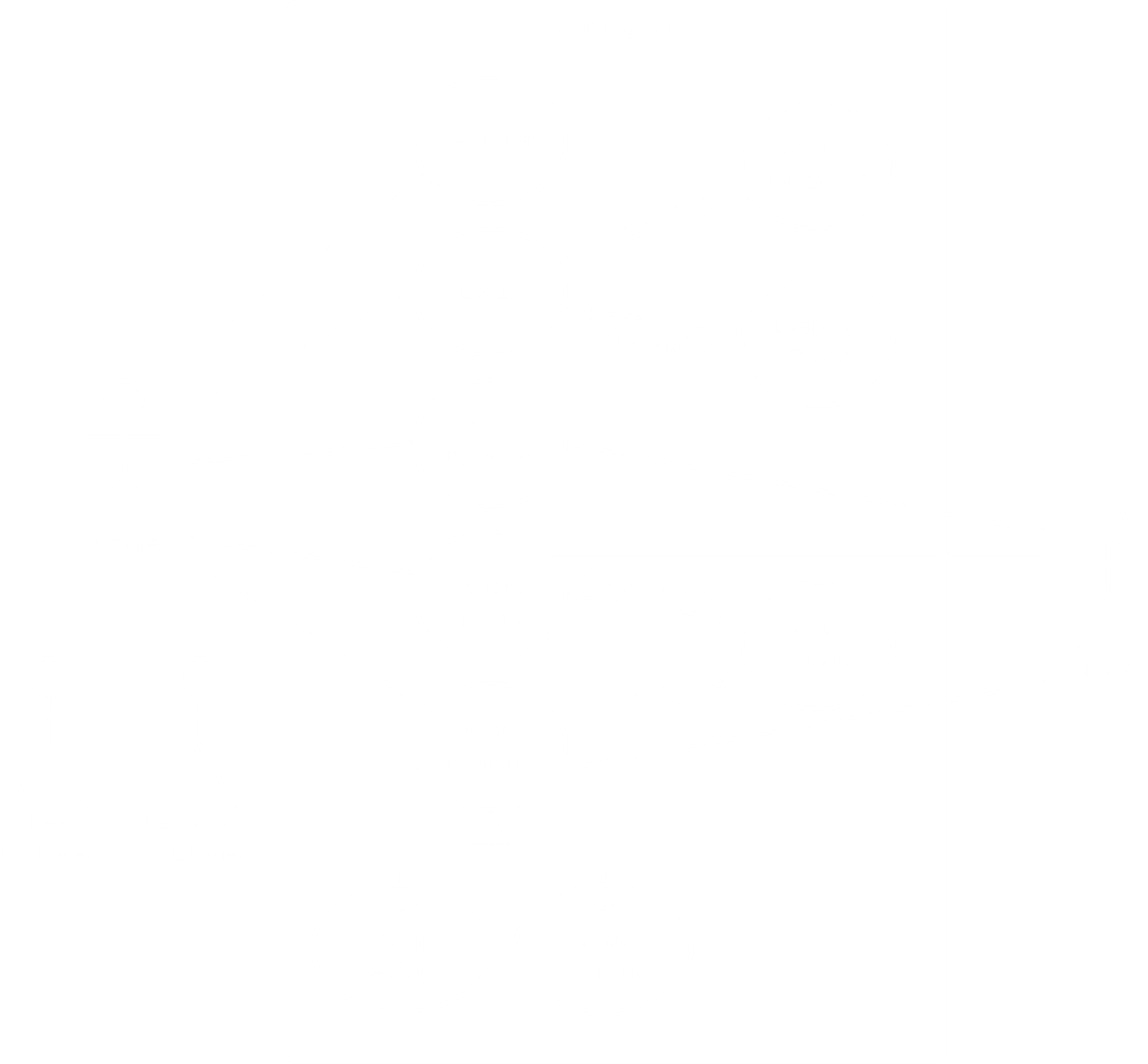
* Use Case Title – This simply describes what the use case is.
* Primary Actor – The primary actor is the actor that initiates the event, as opposed to the secondary actor who reacts to the event. For example, say the accounts department needs to verify the payment made by the recruiter. The accounts department would then be the secondary actor for this use case, since they can only react to what the primary actor has done.
* Preconditions – These are conditions that need to be met before the given event can begin.
* Success Guarantees – These are the outcomes that will result from successful completion of the event.
* Main Success Scenario – The main steps of the event are listed here.
* Extensions – These are errors or exceptions that might occur in each of the steps of the main success scenario. For each error, steps for what will happen when the error occurs must also be listed.

#### Use Case Diagrams

A use case diagram consists of:

* A boundary around the components of the system. This is a rectangle.
* Actors, which are the entities that interact with the system
* The use case, or the representation of the steps that accomplish the given event.
* The relationships between different steps and between actors and the system.

Consider the example below:



This example is for a banking app. The main points to notice here are:

* The name of the system is given at the top. This can be either outside the boundary or inside. Either is fine.
* The primary actor is the customer, and the bank itself is the secondary actor. This is because the customer will perform different actions with the app, and the bank will respond to those actions.
* There are a few different actions available to the user. These are – register, login, check balance, transfer funds and make payments. Each of these are separate use cases and are put in a circle inside the boundary.
* There are solid lines between the actors and the use cases. These indicate association. Notice that there are lines with both the customer and the bank for all of the use cases except login. This is because, in this system, login is the only use case that can be completed without involving the bank. Every other use case involves transactions of some type, which has to involve the bank.
* There are three other types of relationships in the diagram, include relationships, extend relationships and generalizations.
  + Include relationships are between a base use case and an included use case.

For example, logging into the app must involve some form of verification. The login use case cannot be completed without first completing the verification use case. As such, the login use case is the base use case and the verification use case is the included use case.

Include relationships are shown with a dotted arrow from the base use case to the included use case. The word <<include>> is written next to the arrow.

* + Extend relationships are conditional use cases connected to the main base case. Conditional use cases only execute when a certain condition is met.

For example, the relationship between the login use case and the display error use case is an extend relationship, since the display error use case only executes if the login information provided by the customer is incorrect.

Extend relationships are shown with a dotted arrow from the conditional use case to the base use case. The word <<extend>> is written next to the arrow.

* + Generalization relationships are when there are multiple things that can be combined to a generalized thing. Essentially, this deals with base classes and inherited classes, exactly the way it did in object-oriented programming.

For example, the pay from savings account use case and the pay from current account use case can be generalized to the make payment use case using a generalization relationship. Similarly, the new customer actor and the returning customer actor can be generalized to the customer actor using a generalization relationship. Thus, we can have generalization relationships between use cases and between actors.

Generalization relationships are shown using a branched arrow that points from all the different inherited classes to the base class.

### Activity Diagrams

Activity diagrams show the sequence of activities required in a process to achieve a particular goal. Since it is a sequence, the order is important here. Both sequential and parallel activities are shown and any decisions are also included.

An activity diagram is usually created for a particular use case and may show the different possible scenarios.

The different symbols we will be using for activity diagrams are:

* Rectangles with rounded ends, representing activities
* Arrows, representing the flow of activity. Keep in mind that this is not the same as the arrows in data flow diagrams. Here, the arrow simply tells us where we need to go.
* Diamonds, representing branches and merges.

Branches are points where the flow diverges due to a conditional statement. For example, we could ask a question and if the answer is yes, we will have to follow one flow and if we say no, we will have to follow another flow.

Merges are used to merge two or more activities together so that they can be brought onto the same flow.

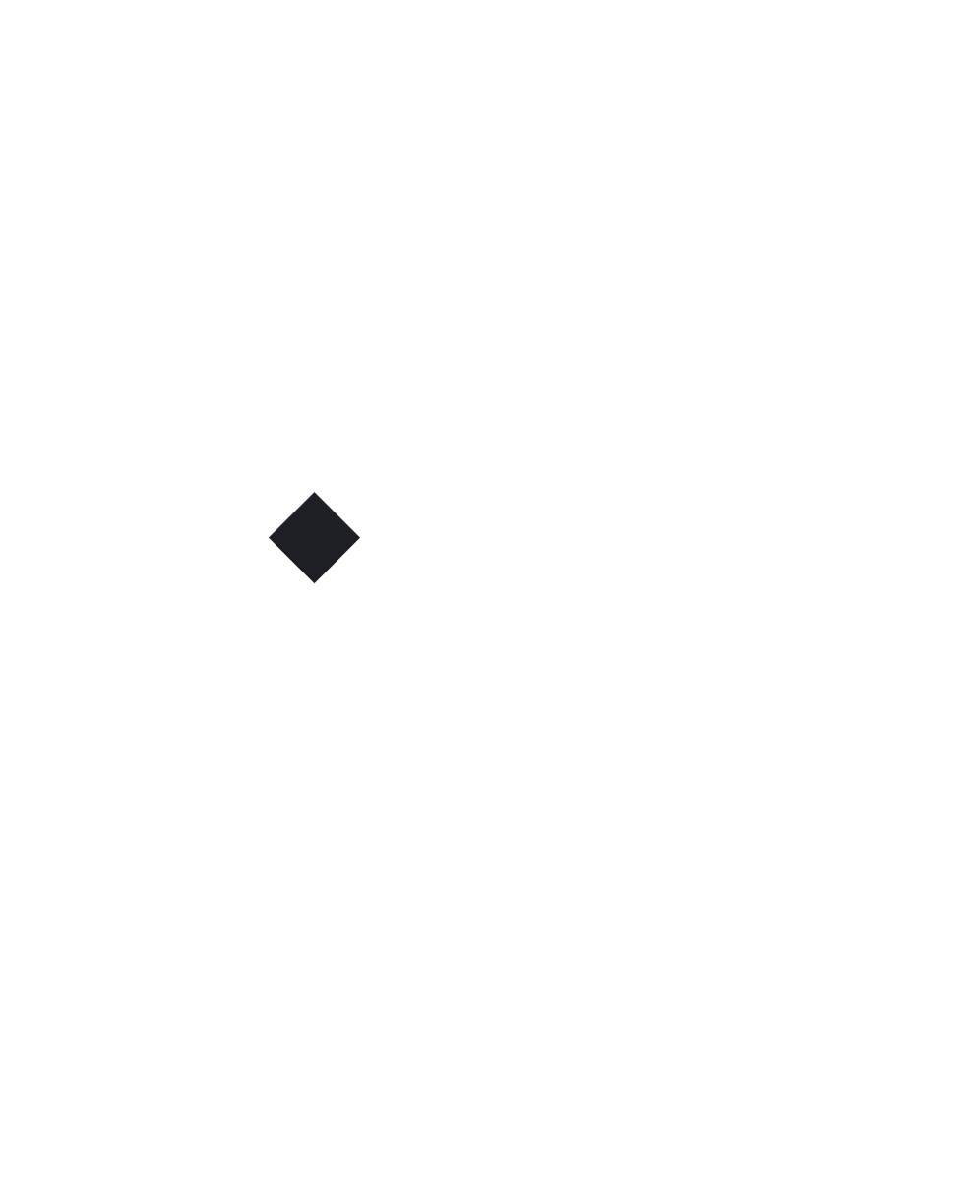
* Long, flat rectangles, representing either joins or forks.

Forks are used to divided a single path from a single activity into multiple paths that go to multiple activities which will work in parallel.

On the other hand, a join does the opposite, taking multiple paths from multiple parallel activities and joining them into one path that goes to one activity.

* Filled-in circles, representing the starting point of the diagram.
* Black circles surrounded by white circles, representing the end of the diagram.
* Swimlanes, which help divided the activities. These are not really symbols, but rather divisions of activities. For example, in the diagram below, there is a vertical line in the middle of the diagram. We could say that every activity on the left of the line is for one group of users and every activity on the right of the line is for another. Thus, each of these groups is a different swimlane.

Swimlanes can also be used to divide up the activities based on location. For example, for an e-commerce website, there are some activities that will take place on the browser, some on the server and some in a database. We can divide each set of activities into a separate group using swimlanes.



Example

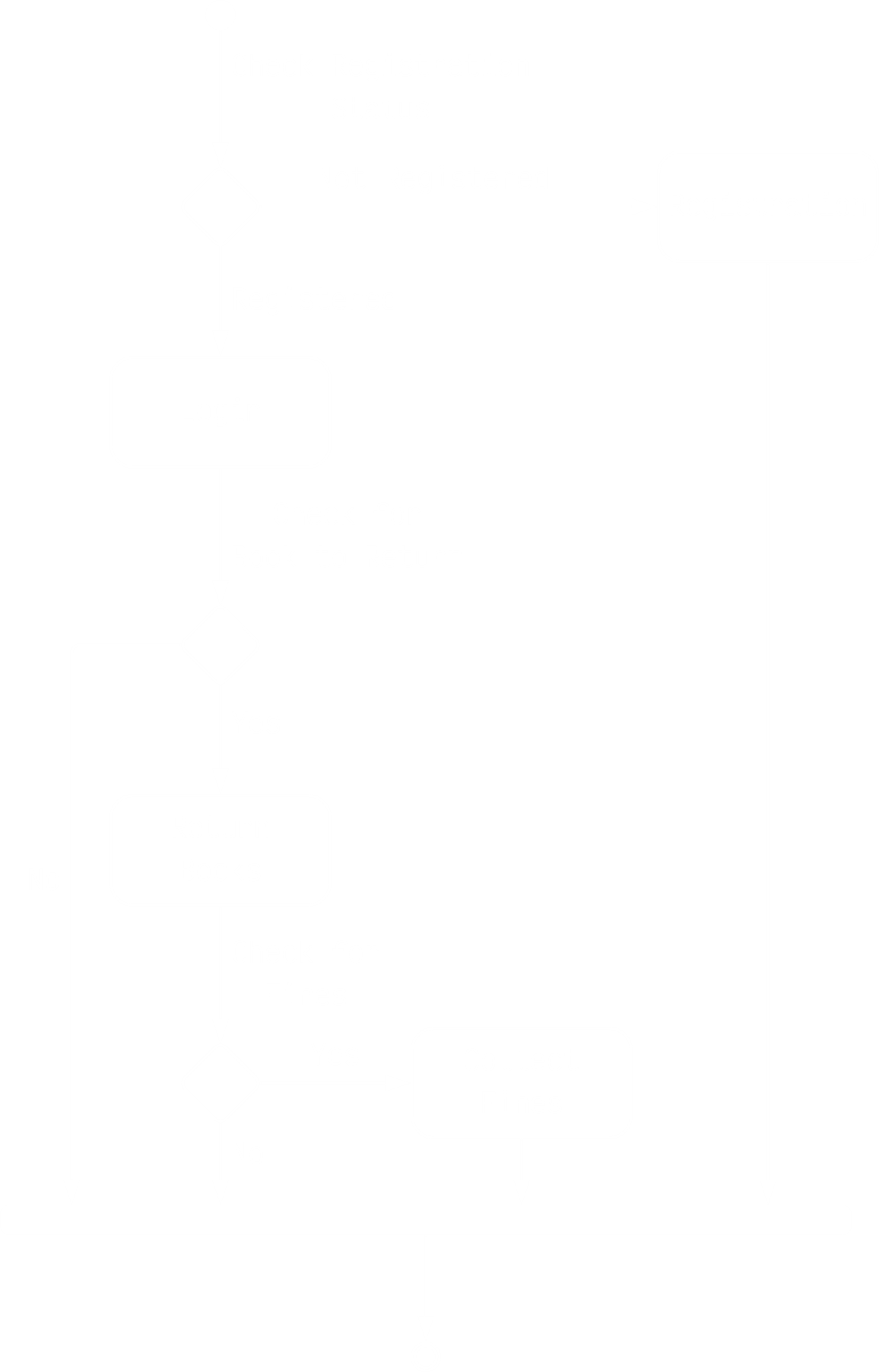
Say we want to make a system for returning books to a library.

1. The first step is to start the activity diagram (*black circle*).
2. Next, we ask (*branch*) whether or not the user is registered.
   1. If they are not, they need to register first (*activity*).
   2. If they are, then they can login (*activity*).
3. Once a user has logged in, we need to ask (*branch*) whether they have any books to return.
   1. If they do not, then we can bring them to the same point as a newly registered user (*merge* or *join*).
   2. If they do, we need to take the books back (*activity*) and also check if they are past the due date (*branch*).
      1. If they are, we need to collect fines (*activity*).
      2. If they are not, we can move on.

In either case, we are again at the same point as a newly registered user or one with no books to return.

1. Finally, we can bring everything onto the same level (*merge or join*).
2. Now, we can end the activity diagram (*black circle with white boundary*).

The activity diagram for this would look somewhat like this:



### Sequence Diagrams

Obviously, sequence diagrams also maintain a sequence. The main purpose of a sequence diagram however, is to show the interactions between classes and object instances. We also show alternatives. Simply put, these are if/else conditions. However, we should only include important alternatives, not each and every one.

Often, sequence diagrams are used to show the processing described in use case scenarios. For example, if we have a use case that allows students to register at a university, the main use case diagram would just tell us that such an activity exists and that students and someone from the administrative staff is involved. The sequence diagram is what would tell us about all the different objects and classes that are involved internally.

Sequence diagrams have two forms, generic and instance form. In the generic form, we deal with the system as a whole, but in the instance form, we only deal with a particular part of the system, such as the registration process. We shall discuss this in more detail a little later.

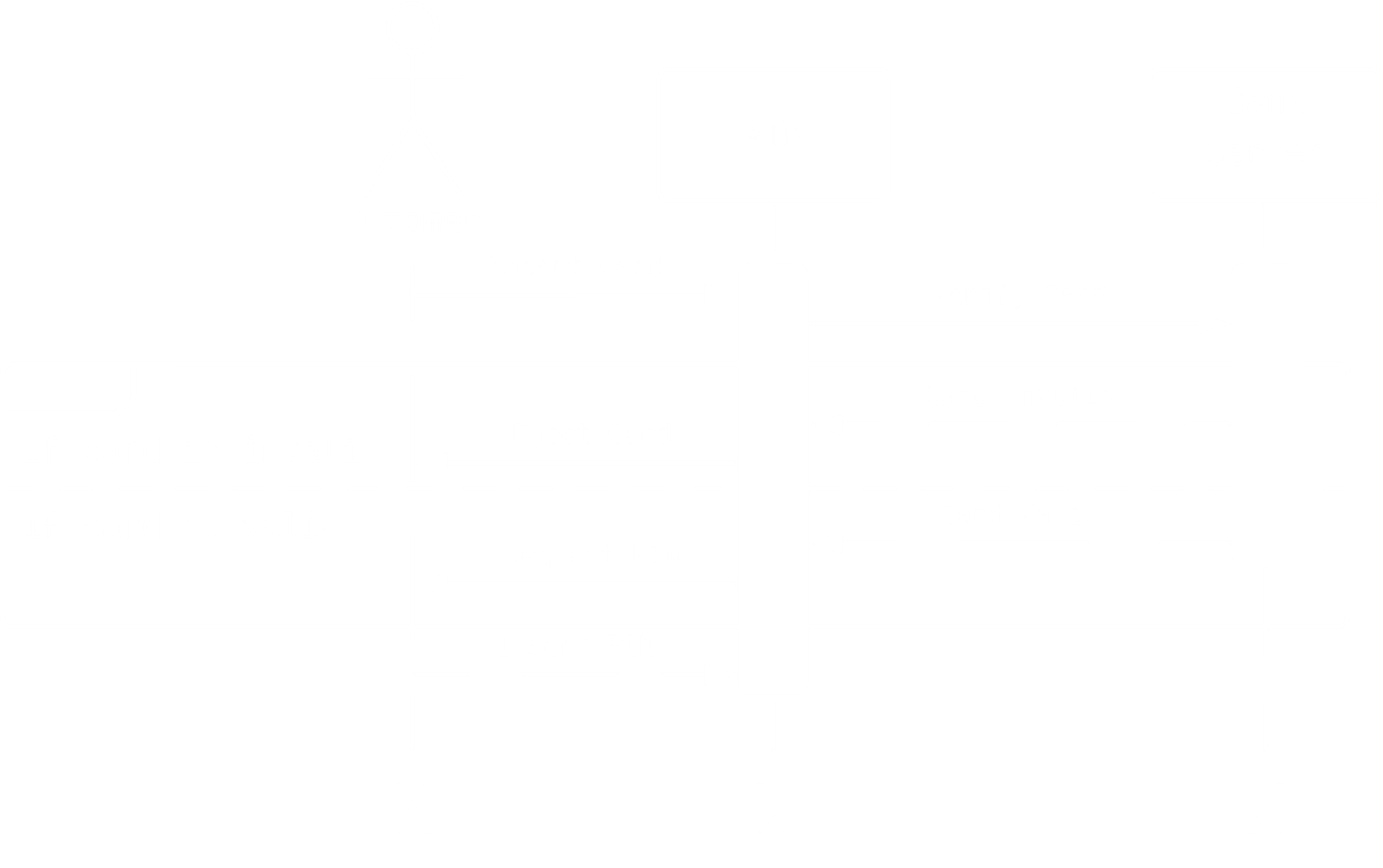
The symbols used in sequence diagrams are:

* Rectangles, used to represent classes and objects of classes. The rectangle that we put at the absolute left of the diagram is the one that starts the diagram.
* Diagrams of stick men representing actors.
* Lifelines, which are dotted lines starting from the class or object that indicate how long those objects exist in the program.
* Activation boxes, which are longer rectangles on top of lifelines that tell us for which part of the lifeline the class or object was active. Note that actors do not get activation boxes.
* Messages, which are solid arrows. These are sent between classes.
* Return messages, which are shown by dotted arrows. Note that a return message is just a response to a particular message or request.
* Asynchronous signals, indicated by the arrow. If one object or class sends a message to another but does not wait for a response before continuing with its work, this message is called an asynchronous signal. The other object will respond at some point. This is opposed to synchronous communication, in which the first object or class needs to wait for a response before continuing.

The timing for a sequence diagram is shown from top to bottom, meaning the first interactions are drawn at the top of the diagram and the last ones are drawn at the bottom.

Example

Consider that we want to represent the system for withdrawing money from an ATM.



In this diagram, notice a few special parts.

We first send a message by inserting our ATM card, and the ATM machine sends a message to the bank server to verify the card.

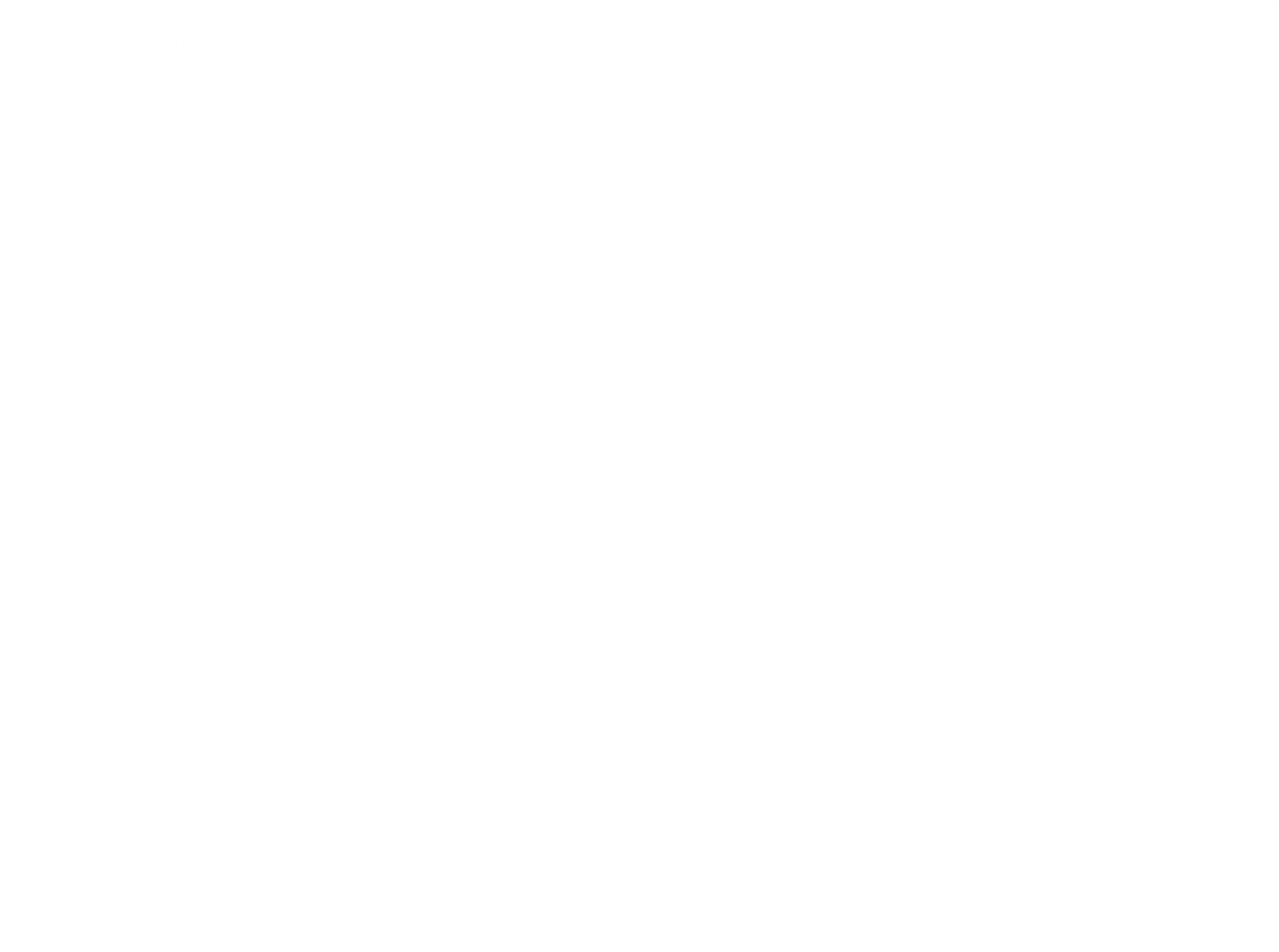
Now, one of two things could happen. The bank server could return a message to the ATM telling it that the card is valid, or it could send a message telling it that the card is invalid. The ATM machine in turn will either request a PIN from us or eject the card. Thus, we have two possible paths. These are called alternatives.

We show alternatives by putting that part of the diagram in a block and dividing the alternatives with dotted lines. We also need to indicated what the conditions are for each of the alternatives to occur.

Another thing to notice is, the ATM machine did not give us a return message in response to the original request we sent by inserting the ATM card. This is because the request for a PIN or the rejection of the card is not a response to anything that the customer sent, but rather a message to the customer. The ATM machine sends us a message by ejecting the card or asking for a PIN. We then send a return message by inserting our PIN.

The activation boxes are drawn once the diagram is complete. Note that this particular diagram is not technically complete yet, since obviously we do not go to ATM machines to make sure we have a valid ATM card and give the machine our PIN.

Example



The above is another example of a sequence diagram. The previous example we saw modelled the system being used. This one is modelled based on the code. Here, we included functions being called and the return types instead of just messages.

### Class Diagrams

Class diagrams are somewhat similar to ERDs in the way that they are implemented. They show the static features of the system, meaning the features that do not change. The different classes in the system and the relationships between them are shown in a class diagram.

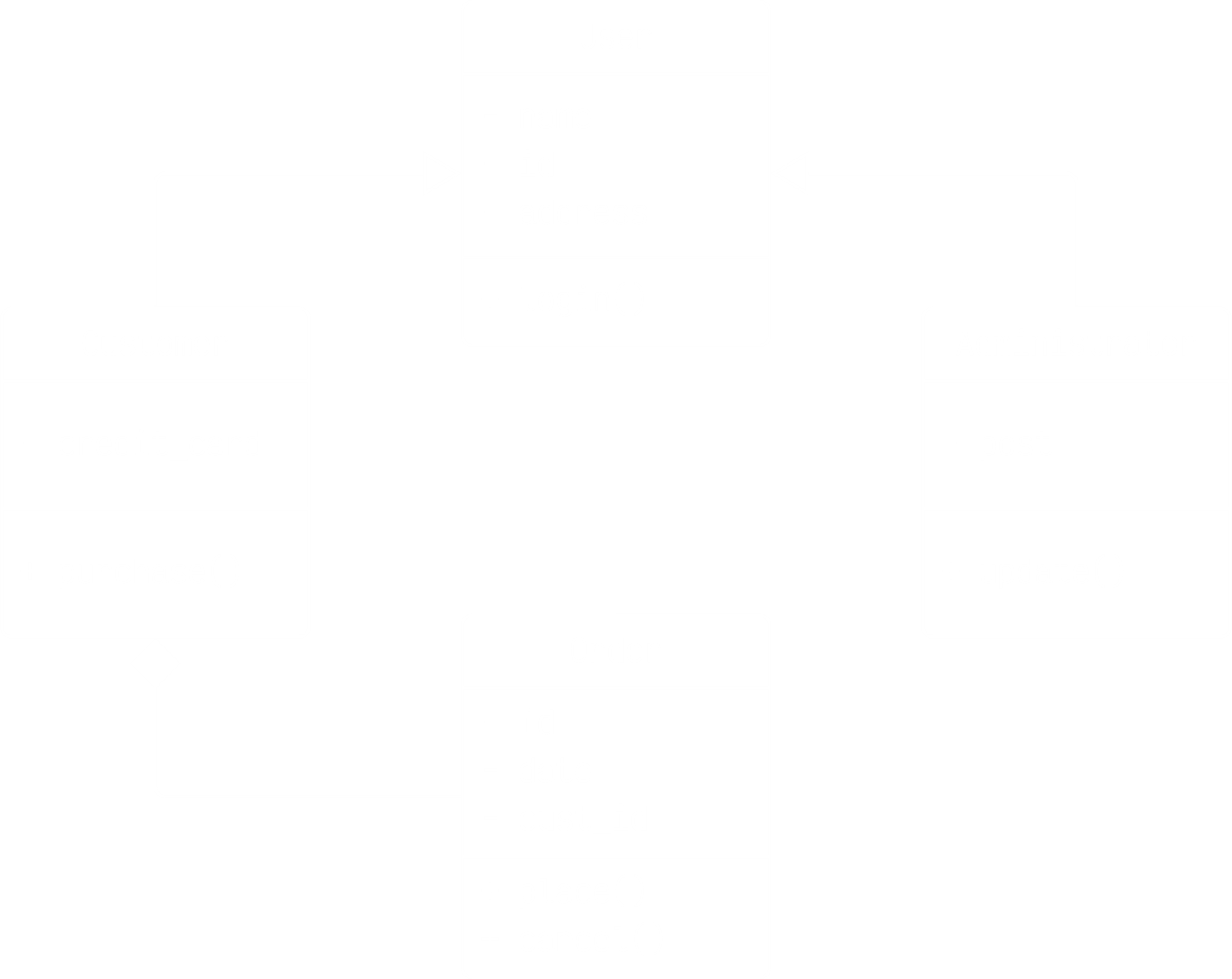
The class diagram does not show any particular processing, but it does show the processing requirements. Additionally, data storage requirements are also shown.

The different elements in a class diagram are:

* Classes
* Attributes – These are the data storage requirements. The three different attributes are private (-), public (+) and protected (#).
* Methods – These are the processing requirements. Both standard and customized methods are included. Standard methods are methods that are general and can be used by any class. Customized methods are created especially for specific classes.
* Relationships – The relationships between classes can include
  + Association - Classes are related, but are independent; represented with a straight line between the classes
  + Aggregation - Child class object can exist without parent class object; represented with empty diamond headed arrow from child to parent
  + Composition - Child class object cannot exist without parent class object; represented with a filled diamond headed arrow from child to parent
  + Generalization – Empty headed arrow from child classes to parent class.

Cardinalities are also shown.

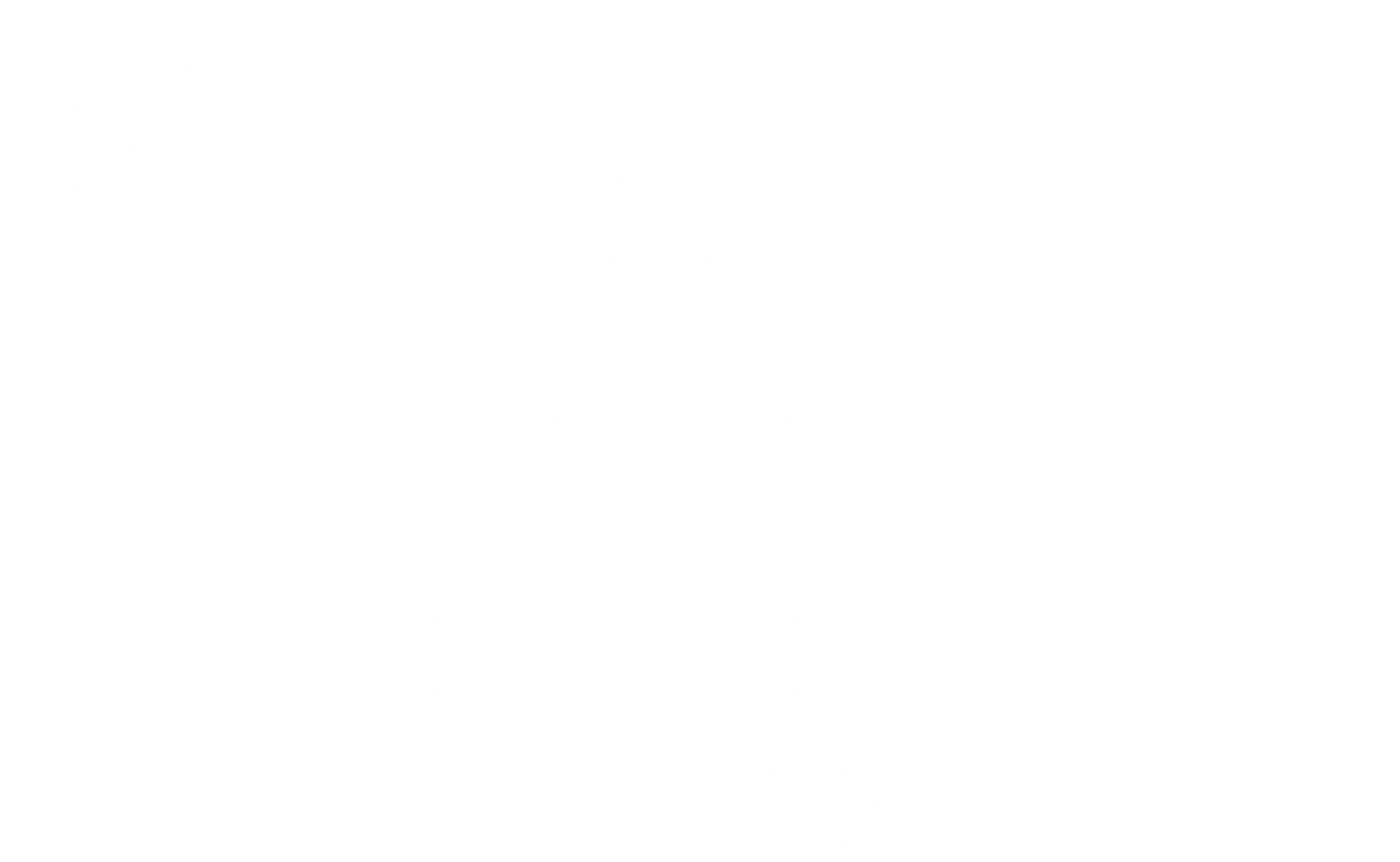
Example



A few points to notice about this example are:

* The Customer and Administrator classes are related to the User class by a generalization relationship. Attributes and methods that are common to both the Customer and Administrator classes are included in the User class.
* The Order class has a composition relationship with the Customer class, since orders cannot exist without being related to some customer. The relationship is a 1-to-many relationship.

Another example which highlights aggregation and composition relationships is shown below.



#### Method Overloading

Method overloading is when the same method is included multiple times in the same class. This can be done as long as the parameters for the method are different for each definition. Thus, depending on which parameters are passed to the method, a different definition will be used.

#### Types of Classes

There are four different types of classes:

1. Entity Classes – These represent real-world items like people and things. These are the entities represented in an ER diagram.
2. Interface Classes – These are also called boundary classes. They provide a means for users to work with the system. Interface classes have two categories, human and system.
   1. Human Interfaces are things like displays, windows, forms, dialogue boxes, etc. Essentially, anything that users use to interact with the system falls into this category.
   2. System Interfaces involve sending and receiving data from other systems. This may be databases within the organization, or completely external organizations. System interfaces are the least stable, since although there is generally an agreed upon format for the data being exchanged, there is no guarantee that the external organization will always follow it.
3. Abstract Classes – These classes cannot be instantiated, i.e. they cannot have any objects. Abstract classes are used to provide a base for generalization relationships. They are denoted in italics.
4. Control Classes – Control classes are used to control the flow of activities. To create reusable classes, there may be many different smaller control classes throughout the system. For example, there could be a general data validation class that can validate user IDs and passwords, but which is not suitably designed to be used directly with login display. Thus, there could be a logon control class that works directly with the login display, takes the information and passes it on to the general data validation class.

An example of a sequence diagram that uses the different classes is shown below. Notice how the shapes of the circles vary slightly depending on the type of class.

