Object Oriented Analysis and Design

The Unified Process in Software Engineering

Unified process (UP) is an architecture centric, use case driven, iterative and incremental development process. UP is also referred to as the unified softwa

The Unified Process is an attempt to draw on the best features and characteristics of traditional software process models, but characterize them in a way that imple

Phases of the Unified Process

.Inception
.Elaboration
.Conception
.Transition
Production

The **inception** phase of the UP encompasses both customer communication and pl

The **elaboration** phase encompasses the communication and modeling activities c

Phases of the Unified Process

continuation

The **construction** phase of the UP is identical to the construction activity defined for the generic software process. Using the architectural model as input, the const

The **transition** phase of the UP encompasses the latter stages of the generic construction activity and the first part of the generic deployment (delivery and feedback

The **production** phase of the UP coincides with the deployment activity of the generic process. During this phase, the ongoing use of the software is monitored, sup

Modelling Concepst

Modeling is the process of representing a model which includes its construction and working. This model is similar to a real system, which helps the analyst predic

Simulation of a system is the operation of a model in terms of time or space, which helps analyze the performance of an existing or a proposed system. In other wo

Developing Simulation Models

Simulation models consist of the following components: system entities, input variables, performance measures, and functional relationships. Following are the steps to perform simulation analysis.

Step 1 − Identify the problem with an existing system or set requirements of a proposed system.

Step 2 − Design the problem while taking care of the existing system factors and limitations.

Step 3 − Collect and start processing the system data, observing its performance results.

Step 4 − Develop the model using network diagrams and verify it using various verification techniques.

Step 5 − Validate the model by comparing its performance under various conditions with the real system.

Step 6 − Create a document of the model for future use, which includes objectives, assumptions, input variables, and the process in detail.

Step 7 − Select an appropriate experimental design as per requirement.

Step 8 − Induce experimental conditions on the model and observe the result.

Performing Simulation Analysis

Following are the steps to perform simulation analysis.

Step 1 − Problem a problem statement.

Step 2 − Choose input variables and create entities for the simulation process. There

Step 3 − Create constraints on the decision variables by assigning it to the simulation

Step 4 − Determine the output variables.

Step 5 − Collect data from the real-life system to input into the simulation.

Step 6 − Develop a flow chart showing the process of the simulation process.

Step 7 − Choose an appropriate simulation software to run the model.

Step 8 − Verify the simulation model by comparing its result with the real-time syste

Step 9 − Perform an experiment on the model by changing the variable values to fin

Step 10 − Finally, apply these results into the real-time system.

Modelling & Simulation – Advantages

Easy to understand − Allows to understand how the system really operates without working on real-time system.

Easy to test − Allows to make changes into the system and their effect on the output can be observed.

Easy to upgrade − Allows to determine the system requirements by applying different configurations.

Easy to identifying constraints − Allows to perform bottleneck analysis that causes delay in the work process, i.e. machine, etc.

Easy to diagnose problems − Certain systems are so complex that it is not easy to understand their interactions.

Modelling & Simulation – Disadvantages

Designing a model is a skill which requires domain knowledge, training and exp

Operations working on real-time systems using random number, hence difficult to

Simulation requires manpower and it is a time-consuming process.

Simulation results are difficult to maintain, etc.

Simulation process is expensive. It requires experts to understand.

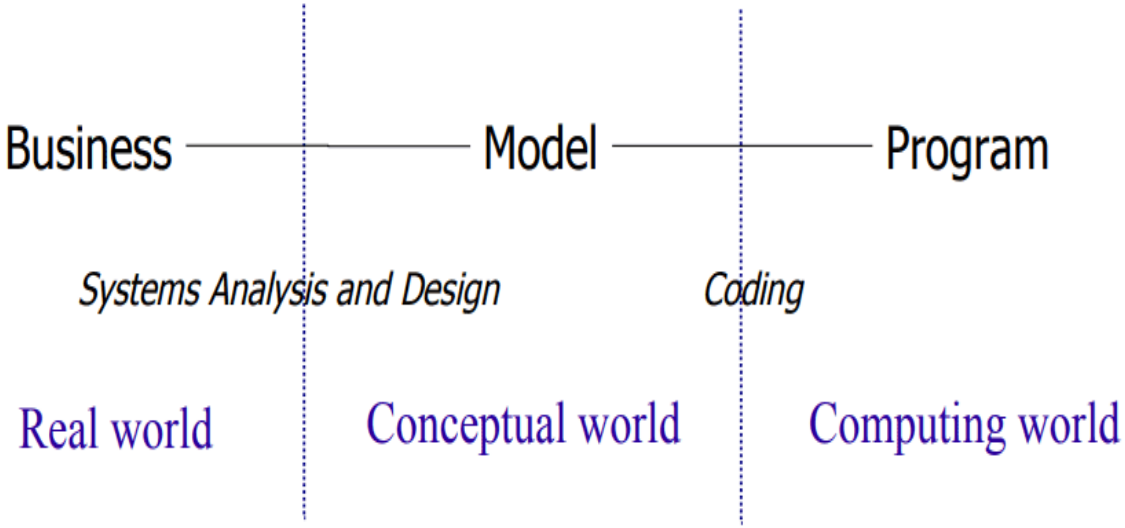
Simulation process is expensive in time. However, Modelling & Simulation allows

Modeling concepts

What is a Model?

"A model captures a **view** of a physical system. It is an **abstraction** of the physical system, with a certain **purpose**. This purpose determines what is to be included in the model and what is irrelevant. Thus the model completely describes those aspects of the physical system that are **relevant** to the purpose of the model, at the relevant **level of detail**"

Where we apply OOAD?



Why use a model?

- A model is quicker and easier to build
- A model can be used in a simulation
- A model can evolve as we learn
- We can choose which details to include in a model
- A model can represent real or imaginary things from
- any domain
- A model allows us to talk, or reason, about the real
- thing without **actually building it**
- Much of software development involves creating and
- refining models, rather than writing lines of code

Why use a model?

 describes what the software should do
 represents people, things and concepts important to understand what is going on
 shows connections and interactions among these people, things and concepts
 shows the business situation in enough detail to evaluate possible designs
 must be organized so as to be useful for designing the software

What is a Diagram?

A diagram is a graphical representation of a set of elements in the model of the system

Models vs Diagrams

- A diagram illustrates some aspect of a system
- A model provides a complete view of a system at a particular stage and from a particular perspective

Why use a diagram?

Natural language is often too ambiguous to be used for modeling

Communication + Ambiguity = Confusion!

A large object with one trunk and four legs

UML (Unified Modeling language) diagrams

UML 2 defines 13 types of diagrams

We can divide all the types of UML diagrams to 3 kinds:

*** Structural:**

Class Diagram, Object Diagram
Component Diagram, Package Diagram
Composite Structure Diagram, Deployment Diagram

*** Behavioural:**

Use Case Diagram
Activity Diagram, State Machine Diagram

*** Interactional:**

Sequence Diagram, Communication Diagram
Timing Diagram, Interaction Overview Diagram

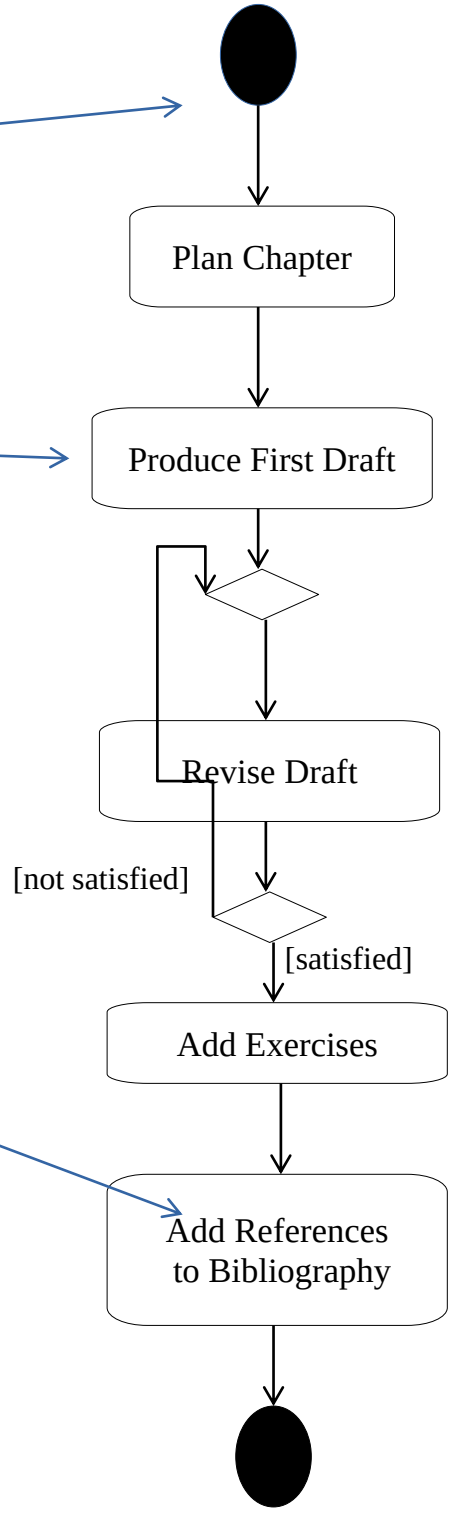
A UML diagram usually consists of:

icons

symbols

paths

strings



What models/diagrams are good?

- **Accurate**
  unambiguous, following rules or standards
- **Concise**
  showing *only* what needs to be shown
- **Complete**
  showing all that needs to be shown
- **Consistent**
  internally and among each other
- **Hierarchical**
  breaking the system down into different levels of details

Developing models

During the life of a project using an iterative lifecycle, models change along the dimensions of:

- abstraction — they become more concrete
- formality — they become more formally specified
- level of detail — additional details are added

Modeling as a Design Technique

Object Modeling Technique (OMT) is real world based modeling approach for software modeling and designing. It was developed basically as a method to deve

Object Modeling Technique is easy to draw and use. It is used in many applications like telecommunication, transportation, compilers etc. It is also used in many r

Purpose of Object Modeling Technique:

- To test physical entity before construction of them.
- To make communication easier with the customers.
- To present information in an alternative way i.e. visualization.
- To reduce the complexity of software.
- To solve the real world problems.

Purpose of Object Modeling Technique:

- To test physical entity before construction of them.
- To make communication easier with the customers.
- To present information in an alternative way i.e. visualization.
- To reduce the complexity of software.
- To solve the real world problems.

Object Modeling Technique's Models:

There are **three** main types of models that has been proposed by OMT:

Object Model:

Object Model encompasses the principles of abstraction, encapsulation, modularity, hierarchy, typing, concurrency and persistence.

Dynamic Model:

Dynamic Model involves states, events and state diagram (transition diagram) on the model. Main concepts related with Dynamic Model

Functional Model:

Functional Model focuses on the how data is flowing, where data is stored and different processes. Main concepts involved in Functional Model are data, data flo

Phases of Object Modeling Technique:

OMT has the following phases:

Analysis:

This the first phase of the object modeling technique. This phase involves the preparation of precise and correct modelling of the real world proble

System Design:

This is the second phase of the object modeling technique and it comes after the analysis phase. It determines all system architecture, concurrent

Object Design:

Object design is the third phase of the object modelling technique and after system design is over, this phase comes. Object design

Implementation:

This is the last phase of the object modeling technique. It is all about converting prepared design into the software. Design phase is

05.08.21

Software Development Methodologies

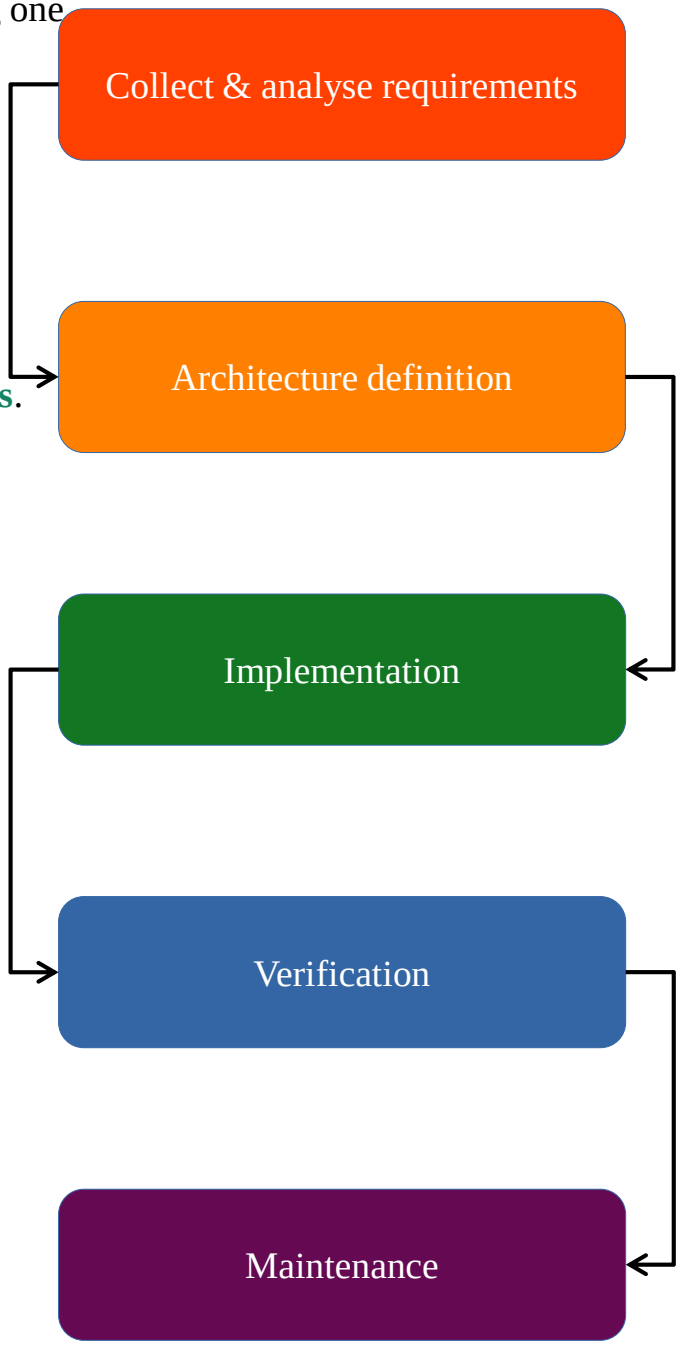
Waterfall vs Agile

There are different ways to develop a software system. Sure, you can just sit down and and start typing the source code. This wor
None of this systems can precisely describe every step of the software development process!

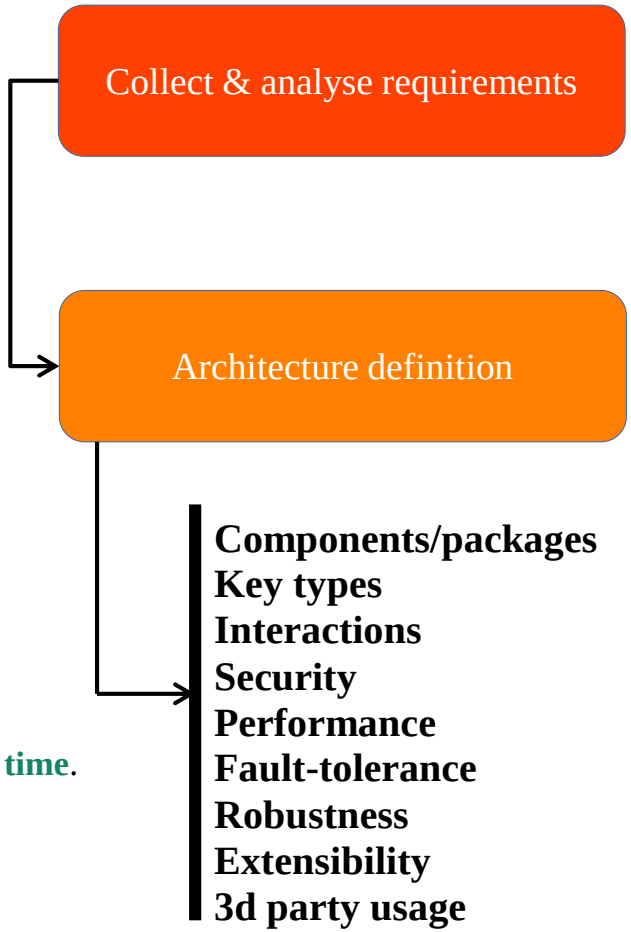
We definitely need them to synchronize and organize our development-related activities.
Activities are including not only coding, but also design, product-management, budgeting, testing, documentation,release, and ma

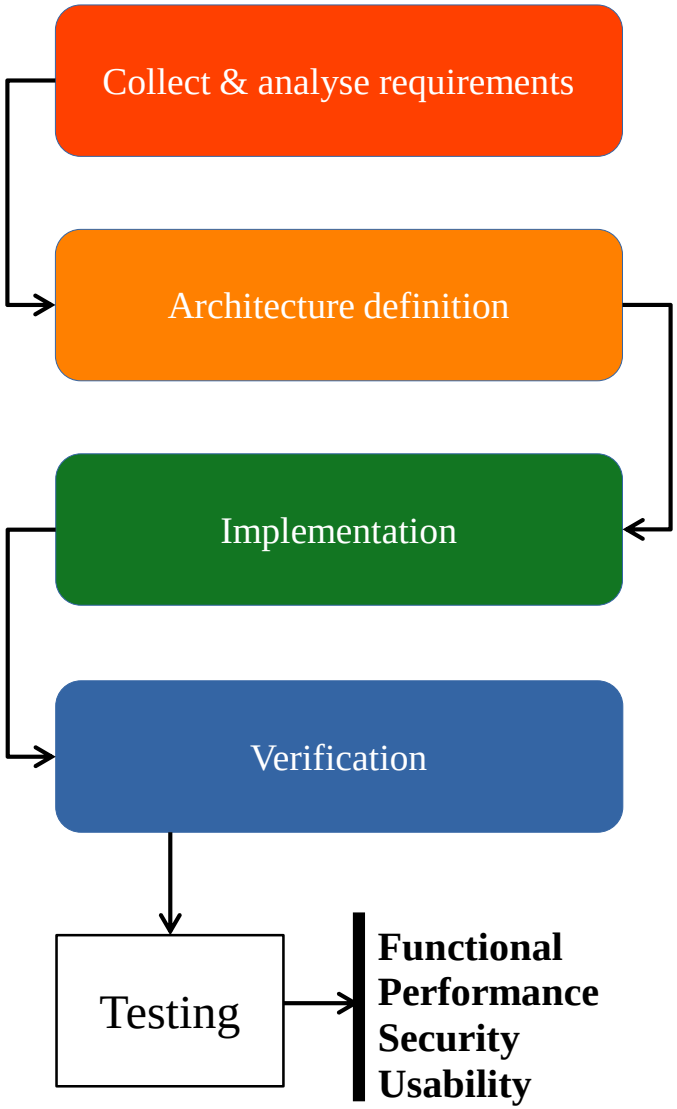
The Waterfall Model

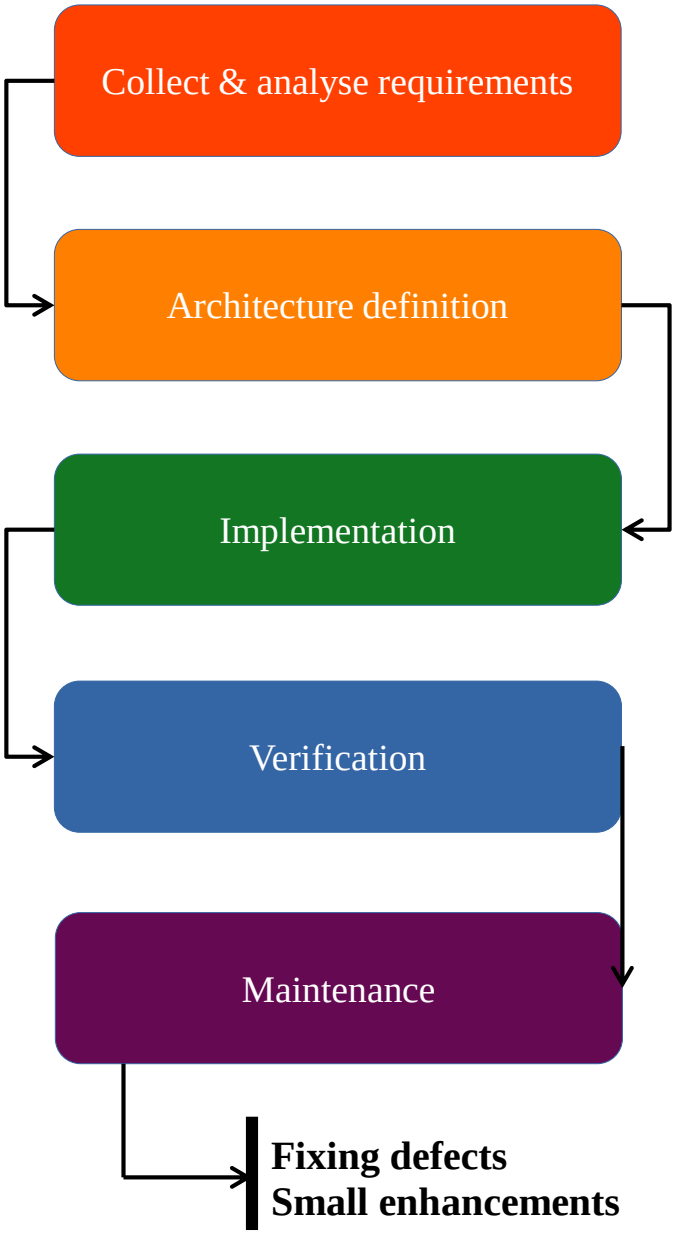
The **Waterfall** is a linear model. It defines development steps or phases. You start executing one step, complete it, and then start the next one.
This approach gives us a steady downward order.
Hence the name Waterfall.
The development process flows in cascades.
Each development phase requires the previous one to be completed,
So, each phase builds upon the previous one
Let's talk a bit about these phases.
First, we **collect and analyze the requirements.**
The expected functionality of the future application must be **clarified with the stakeholders**.
All the details must be **documented thoroughly**.
The very first phase is probably the most important one.
When done right, the Waterfall model will produce the expected outcome.
After collecting and analyzing the requirements, we can proceed to the next phase.
Here's where we define the overall design of our software.
Defining the architecture is like creating the blueprint for a building.
Thus, the design should be as clear and detailed as possible.
The team should be able to implement the product based on this plan.
We should address questions like "What packages or components will form our system?"
"What are the fundamental types of each component?"
"How do these types interact with each other to achieve the required functionality?"
"Is our software secure?"
"How about performance?"
"How does our software respond to errors?"
"How do we handle edge cases?"
"Should we extend our system in the future?"
"What third-party components do we use?"



And the list can grow or shrink depending on the requirements we defined in the previous phase.
The implementation comes next.
The software development phase is usually divided into smaller units.
Each unit is then implemented and tested by developers.
Once the development phase is completed, the product undergoes the verification phase.
During the step, the software is evaluated based on predefined criteria.
We must check whether the product provides the functionality we agreed on.
Tests are executed to ensure that the software works
as expected. We test for functional performance, security, and usability issues. The detected problems are recorded and fixed.
The process goes on until all severe bugs are fixed.
The verification phase may also bring to surface deeper bugs and critical issues that may affect the planned release.
Now, when the testing phase completes and the given version is delivered, the software enters the maintenance phase.
By definition, the maintenance phase is about fixing smaller bugs, but more often than not, it may also include functional enhancements.
The client may come up with new requirements that involve substantial changes.
You may feel tempted to squeeze in just one more patch in the maintenance phase.
This is usually a bad idea.
Instead, you should set up a new Waterfall project and repeat all the steps.
The Waterfall model is used for **life control**, **medical** and **military** systems.
This model requires us to clarify all the requirements and create a detailed plan upfront.
The Waterfall is a perfect choice if all the requirements are **precisely defined and won't change over time**.
Now, the Waterfall has received some criticism for its inability to respond to changes. Due to its linear structure,
new requirements can't be considered at later phases of the development process.
If the client changes their mind frequently or our design misses essential aspects, we're going to hit problems during development or testing.
In such cases, we should follow a different approach.



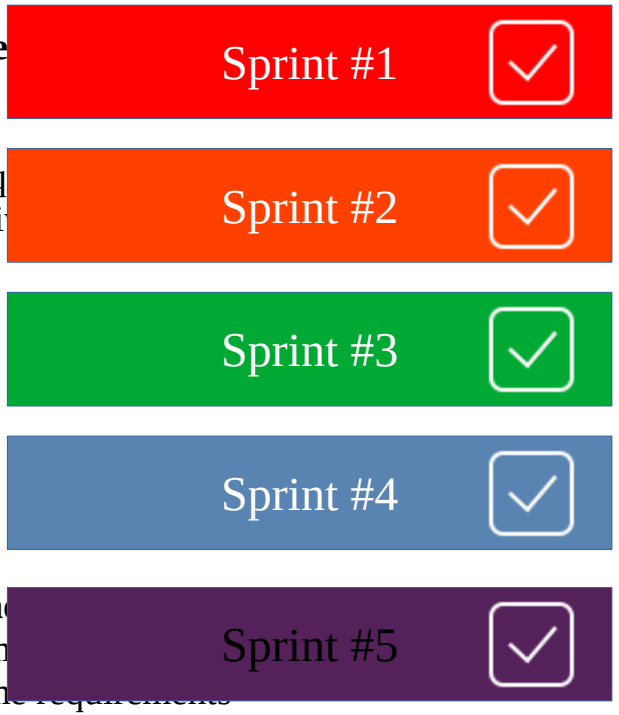




Avoid making substantial changes during the maintenance phase!

The Agile Model

Agile is a relatively new approach to software project management.
It all began with the **Agile Manifesto in 2001**.
This manifesto was an **attempt to end the proliferation of methodologies that had deve** values
*** Individuals and interactions over the processes and tools.**
This does not mean that we won't use processes and tools in agile projects.
We still need tools and processes, but they shouldn't prevent us from implementing the req
Instead of enforcing people to follow a rigid process, we implement a process that's adapti
The second principle:
*** Working software over comprehensive documentation.**
This doesn't mean that agile projects don't use documentation at all.
We should create documentation where it provides a value.
There is no need to create extensive documentation just for the sake of it.
The third principle:
*** Customer collaboration over contract negotiation.**
Don't get this wrong either.
Agile projects also require contracts to manage customer expectations about costs an
Yet, unlike for plan driven projects, there is a **spirit of partnership** between the developm
Due to the somewhat uncertain nature of agile projects, both parties acknowledge that som
and details may need to be redefined or clarified further as the project progresses.
It goes without saying that this kind of partnership requires collaboration and trust.



The next one:
*** Responding to change over following a plan.**
Agile is different from plan-driven approaches, and provides more flexibility compared to the Waterfall model.
The major difference is that Agile welcomes changes even at the later phases of the development cycle.

Some planning is also required for agile projects, but we don't try to come up with a detailed plan
for the entire project before starting any development
activities. As a consequence, we're not blocked until all the requirements are defined and each and
every question gets answered.

Now, let's talk about how an agile approach solves the problem we saw with the Waterfall.
The main idea behind Agile is that we can provide functional software iteratively instead of delivering
the entire project all at once.
The work is broken up into shorter chunks called **sprints**.
The sprint is usually two to four weeks long.
At the end of each sprint, the team should deliver a version that's an improvement over the previous
sprint's outcome.

This interactive approach provides an opportunity to frequently review the product that's being developed.
Stakeholders have a chance to evaluate the software and provide their feedback early on, rather than
waiting for the final product to be delivered.

These frequent checkpoints are super useful as they ensure that the project evolves in the right direction.
Unlike the Waterfall, agile methodologies do not separate testing from development.
Testing is tightly integrated with the development, and the entire team owns the responsibility for the quality of the product.

Also, involving the business users in the development process stands at the core of agile approaches.
There is a strong relationship between the project team and the stakeholders and business users.

This model works best in situations where the requirements can't be defined upfront.
Agile is a good fit for software projects that are depending on many uncertain factors, and changes
are to be expected.
One of the big benefits of this collaborative model is that it usually leads to higher customer satisfaction,
and team members will likely be more motivated by engaging customers directly.
Note that **Agile is not a methodology, but rather a way of thinking** defined by the Agile Manifesto
values and principles.

Scrum and Kanban are Agile frameworks

Scrum official site: https://www.scrum.org/resources/what-is-scrum

A brief introductin to kanban: https://www.atlassian.com/agile/kanban

---

Agile values:

Individuals and interactions over processes and tools

BUT: We sill need tools and processes!

Working software over comprehensive documentation

BUT: Don’t use "agility" as an excuse for a lack of documentation!
Document when it provides a value

Customer collaboration over contract negotiation

BUT: Contracts can’t be avoided!

Responding to change over following a plane

Some planning is also required for agile projects
BUT: Don’t create a detailed plan in advance!
Do only as much planning as needed

Waterfall or Agile?

Waterfall is usually perceived as rigid and bureaucratic compared to agile methodologies.
However, both have their place.
There are cases when a plan-driven methodology won't work.

If the level of uncertainty is high and not all questions can be answered right away, then you should probably choose an agile methodology. For other projects,a Waterfall-like approach will be a better fit.

Let me give you some examples.
When developing a weapons control system, the requirements should be clarified in advance and need to be stable.
Changing the requirements midway would increase the cost considerably and these kinds of projects are expensive anyway.
A Waterfall approach makes perfect sense in this case.

Here's another - actually real example - from UpWork: "Looking to create the next big social media platform for iOS and Android."
Now, coming up with a detailed plan based on assumptions wouldn't make sense in this case.
The description is vague and the customer won't be able to describe precisely what they want.
This high level of uncertainty calls for an agile approach - or rather just skip this project. :)
Creating the next big social network will require multiple iterations.

So, to sum it up, you **should use the Waterfall when you know what the final product should be and clients can't change the scope of the project.**

Agile methodologies should be used when there's a lot of uncertainty involved, the requirements are vague or rapidly changing, and clients can't precisely describe what the end product should do or look like.

Waterfall is best suited for projects with clear requirements and fixed scope

Choose **Agile** if requirements are unstable and may change frequently

Objects

While structured programming relies on actions, **object-oriented programming is organized around objects**.
An object represents a thing. Just like in real life, objects can be simple or complex.
A golf ball is an object, but so is Falcon Heavy.
The way we define an object depends on the level of detail we need.
With the launch of the Falcon Heavy, they also sent the Tesla Roadster with Star Man in the driver's seat toward Mars orbit.

Objects may contain or refer to other objects.
This can also happen in the object-oriented world.
We can describe objects using their **properties**.
They can have **attributes** like name, color, weight and velocity.
A golf ball can be completely white, colored, or it may even glow in the dark.
It has a weight and the price.
Some properties like its position, speed and acceleration can change, while other attributes - its color, for example, will stay the same.
All these properties describe an object in the real world.
This approach works also in an object-oriented language.
Objects have their **identity**, their own **state**.
Changing the state of one object doesn't change the state of other objects.
If we hit a golf ball, it won't affect all the other balls.
Their state is independent.
Each has its private identity.

Besides properties and identity, an object has its own behavior.
The behavior of an object is what it can do.
"The black dog barks."
In this sentence, we identify one object:
the dog. "Bark" is the **behavior** or the action performed by the dog object.
And "black" is its color, one of its attributes.
We can identify the object easily since it's the noun in the sentence.
The verb is the **behavior**, and the adjective is the **property**.
We describe an object using its properties, identity and behavior.
Quite straightforward, but how can we make this work in our code?
For that, we need to introduce a new concept: the class.

The Class

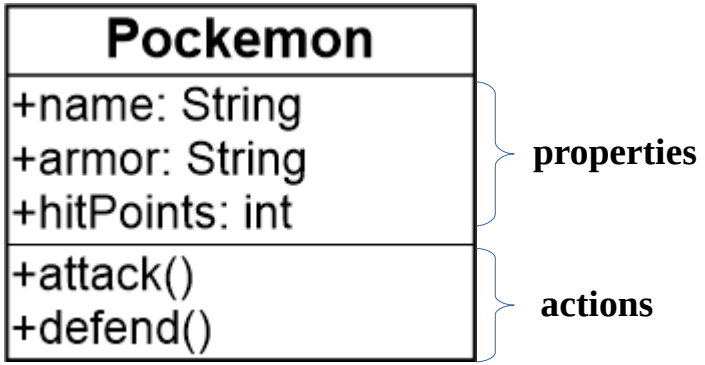
Building an object-oriented system starts by identifying the potential objects, their attributes and responsibilities. We need to have a class before we can create an object.
The class is the blueprint of an object.
You can think of a class as a plan, a description of what an object will be.
Let's say you want to use a Pokémon in your program.
A class called Pokémon would provide a blueprint for what the Pokémon looks like, and what it can do.
The class tells us that each object has a name, armor level and hit points.
It doesn't say what the name or the armor level is.
Pokémon can attack and it can defend itself.
These two actions define its behavior.
We created a class by giving it a name and declaring its properties and actions.
We call these actions methods.
Methods are blocks of code that can be called to execute certain operations.
They may take input parameters and can also return values.
Methods are like functions in structured programming languages.
The methods are basically functions embedded in a class.
Given this class, we can create objects based on it.
Upon object creation, we provide the values for the attributes declared in the class.
Each object will have a name, armor level, and hit points.
The attack and defend behaviors are provided by the class.
Given this blueprint, we can create as many instances as we need.
Another big benefit of classes is that we can package them into libraries or frameworks.
This lets us reuse them in other programs as well.
All modern object-oriented programming languages provide such built-in frameworks and libraries.
We do not have to reinvent the wheel by implementing functionality that's already available.
Instead, we can focus on creating and using the objects in our programs.
Now, each program has different requirements.
The premade classes
will rarely cover all our needs.
You'll often find yourself creating your own classes.
We have now covered the object and the class.
Next, we'll take a look at the core object-orientation principles.

Pockemon
+name: String +armor: String +hitPoints: int
+attack() +defend()

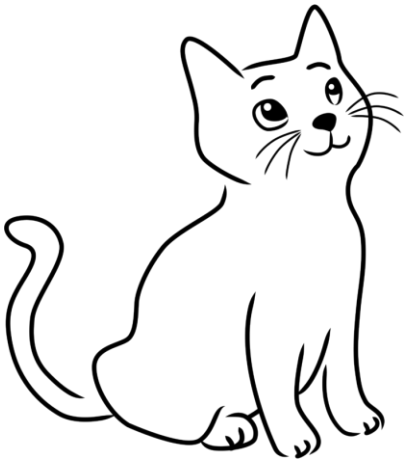
Abstraction

Abstraction is a way of **describing complex problems in simple terms by ignoring some details**. Eliminating the minutiae helps us focus on the bigger picture.
We can dig deeper once we have a broader understanding. How does abstraction work?
If I say cat, you know instantly what I'm talking about. I don't need to say I was thinking of a male, Persian cat; if it's a kitten or if it's big or small.
You understand that I was talking about a cat.
We are naturally good at generalizing things.
We can skip the irrelevant details and people will still understand us. That's because our brains are wired to understand abstract ideas like cat, house, or car. Abstraction works the same way in the object-oriented world.
When we start defining a class, **we focus on the essential qualities and discard unimportant ones**.
Back to our Pokémon example.
We started with the most important attributes and methods.
We don't need details like age, weight or height, so we can ignore them.
These attributes are unessential in our current application. So, that's how abstraction works.
We focus on what's important and ignore all the details we don't need.

Just a pockemon



Just a cat

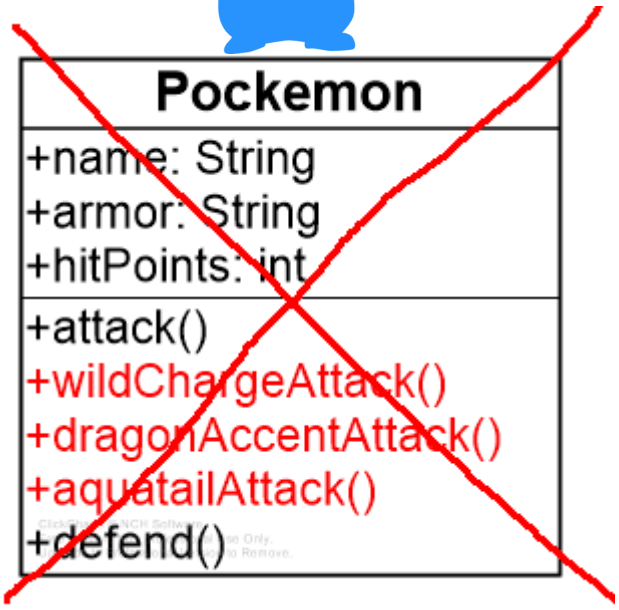


Encapsulation and Data Hiding

The next fundamental idea to keep in mind when dealing with object-oriented programming and classes is called encapsulation. We encapsulate something to protect it and keep its parts together.
Think of how medicine is enclosed in a shell called capsule. In object-orientation, this translates to packing together our properties and methods in a class.
Encapsulation also means hiding the gears and the levers.
Here's an example.
We can use a phone without understanding electronics.
We don't need to know how the touchscreen, the camera or the logic board works.
Similarly, we don't want to expose the inner workings of our class.
An object should only reveal the essential features.
This concept is called data hiding.
By hiding the internal details, the object is protected from external interference.
Basically, we restrict clients from modifying the object in ways we did not originally plan, whether it's intentional or accidental.
Additionally, we prevent other parts of the system from relying on properties or behavior that may change.
If you replace your phone's battery, it won't affect the way you use your phone.
That's because you only interact with the touchscreen.
Changes in the inner workings of your phone don't matter to you.
Our classes shouldn't be any different either.
If we expose unnecessary details, any changes to those attributes or methods may affect other parts of the system.
Whereas if we restrict access to that data or behavior, we don't have to worry about the ripple effects of our changes.
Data hiding is not about selfishly keeping stuff for ourselves, rather, it's about protecting our classes from unwanted external dependencies. As a rule of thumb: **Expose only as much of your class properties and methods as needed for normal usage.**
Data hiding plays an essential role in keeping the dependencies between objects to a minimum. A tightly-coupled system, with most of the objects depending on each other is an obvious sign of a bad design.
Updating or maintaining such a system is a pain.
Any tiny modification will cascade down and require you to change other parts of the system, too.
It's like a never ending nightmare.
Object-oriented programming principles are here to make our lives easier.
Next up is the idea of inheritance.

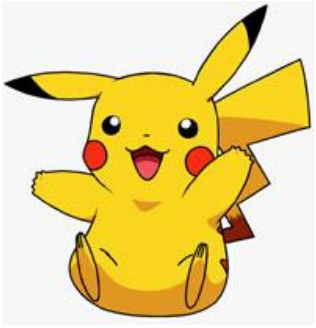
Inheritance

Inheritance is a key concept in object-oriented programming. Without inheritance, we'd end up writing similar code over and over again.
Inheritance means code reuse, that is, reusing an existing class implementation in new classes.
Let us start with an example.
We modeled the Pokémon class with the main properties and behavior in mind.
Given this class, we were able to create our Pokémon instances.
Now, what if we need new types like Electric, Water, or Flying Pokémon?
We will need new classes since these new types have special abilities. For properties, the Pokémon class has a name, armor, and hit points, and it can attack and defend itself.
The Electric, Water, and Flying Pokémon have all these properties and they are also able to attack and defend themselves.
Additionally, they have specialized functionality. An Electric Pokémon has the ability to "Wild Charge."
This attack is only available to electric type Pokémon.
"Aqua Tail" is a damaging Water Pokémon move and Flying Pokémon can perform the "Dragon Ascent" attack.
As you've probably realized, the three new classes are almost identical to the Pokémon class.
The only difference is the special attack type. We could add all these new behaviors to the Pokémon class.
If we did that, we'd end up in a class that **has too many responsibilities.**
Suddenly, all Pokémon objects could swim, and fly, and discharge electricity.
We definitely don't want that.
Our classes should be simple.
They need to have a well-defined purpose.
Object-orientation is about granularity and separation of concerns.
Each class should focus on a set of specific functionalities and do that well.
Creating one-size-fits-all monolithic classes is a major mistake in object-oriented software development.
So how about keeping these classes separate?
That sounds better, but now we keep repeating the same code for common Functionality. There must be a better way.

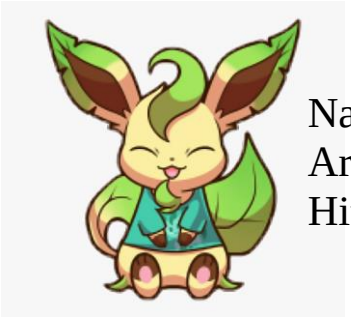


Pockemon
+name: String +armor: String +hitPoints: int
+attack() +defend()

superclass
(parent class)



Name: Pickachu
Armor: 100
Hit points: 1000



Name: Kawaii
Armor: 75
Hit points: 960

Name: Armaldo
Armor: 200
Hit points: 5000

subclasses
(child classes)

Indeed, object-oriented programming languages have a solution for this kind of problem: inheritance.
A class can inherit all attributes and behavior from another class.
In our case, we let Electric Pokʼmon, Water Pokʼmon, and Flying Pokʼmon inherit from the Pokʼmon class.
The data and the behavior from the Pokʼmon class are now available to all these classes, without us having to write a single line of code.
Now we can add specialized behavior or attributes to the classes that inherit from Pokʼmon.
If we enhance or modify the Pokʼmon class, all the other classes will automatically receive those changes.
In object-oriented terms,
Pokʼmon is a parent or a superclass, whereas the Electric Pokʼmon, Water Pokʼmon, and Flying Pokʼmon are subclasses or child classes.
Inheritance is a powerful idea that can save us from a lot of extra typing. And finally, it paves the road to another handy feature called Polymorphism.

Pockemon

+name: String
+armor: String
+hitPoints: int

+attack()
+defend()

ElecrricPockemon

+WildChargeAttack()

FlyingPockemon

+DragonAscentAttack()

WaterPockemon

+AquaTailAttack()

Polymorphism

Here's another term you will often hear when it comes to object orientation: **Polymorphism**. The word has Greek origins, and it consists of two words: "**polys**," which means many, much, and "**morfé**," meaning form, shape.

If you look up the word polymorphism, you will find the following definition: **the condition of occurring in several different forms.**

But how does this apply to programming?

To understand how polymorphism works, we have to revisit the idea of inheritance.

Here's our Pokémon superclass and its subclasses.

The Electric, Water, and Flying Pokémon all inherit the data and the behavior of their superclass.

So they have a name, armor, hit points and they can attack and defend themselves.

The Water Pokémon inherits the attack behavior from the Pokémon superclass.

Now, what if we need Water Pokémon types to cause more damage than basic Pokémon?

For that, we need to provide the specialized implementation of the attack behavior.

This is what we call method overriding. By overriding a method of the superclass, we tell that we want a different behavior in our subclass than the one we inherited.

Method overriding is straightforward.

We reimplemented the method with the same name and parameters as the one defined in the parent class, and provide our own behavior. By doing so, the Water Pokémon objects will have a different attack behavior than their Pokémon superclass.

Calling the attack method on the Electric and Flying Pokémon objects will use the logic implemented in their superclass, since we did not override the attack method in their case.

So, that's method overriding. Polymorphism lets us work with objects created from any of these classes.

We don't need to know whether it's a Water-, Flying- or Electric Pokémon instance to call any of the common methods defined in the superclass.

We could create an army of mixed Pokémon and tell them to attack at once.

Each of them will execute the right attack method without us having to know their exact type.

All we know is that there are all instances of the Pokémon type or one of its subclasses.

```java
public class Pockemon {

    public String name;
    public String armor;
    public int hotPoints;

    public void attack() {

        

    }

    public void defend() {

        

    }

}
```

```java
public class WaterPockemon extends Pockemon {

    @Override
    public void attack() {
        
    }

}
```

Polymorphism is about working freely with instances of many different classes that share a common super class.
It's probably easier to grasp it when using it in a real program.
So, let me show you an example.
I'm going to use Swift and Xcode
on a Mac. We'll implement the Pokʼmon class and all the subclasses.
This is an overly simplified example,
but it's good enough to show you how polymorphism works in a real program.
So, here's our Pokʼmon class.
The attack method just displays a message to the console. The Electric-, Water- and Flying Pokʼmon classes inherit from the Pokʼmon class.
And this is how we do it
in Swift: we put the name of the superclass after the name of the child class separated by a colon.
I override the attack method in the Water Pokʼmon class. To specify that I'm overriding a method in the superclass, I use the override keyword.
For this example, we're simply displaying a different message.
Next, I'll create some Pokʼmon instances: one Pokʼmon object, a Water Pokemon, then an Electric- and a Flying Pokʼmon, too. Then, I define a list with these objects.
Now, I can traverse this list and call the attack method on the objects in the list.
We don't need to know the class they were instantiated from.
Now, if I ran the demo, we'll see that the attack method produced the same output in the console for all the objects, but one. That object was of type Water Pokʼmon - the one which overrides the attack method.

Some Questions:

Which of the following represents an object?

1.Blueprint
2.Instance
3.Property
4.Behavior

What describes an object?

1.Visibility, mutability and actions
2.Name, category and rules
3.Noun, adjective and verb
4.Properties, state and behavior

What do we need to create an object?

1.Inheritance
2.A framework
3.A class
4.Abstraction

Which statement is true about classes?

1.A class must inherit from another class
2.We can create custom classes
3.A class can only have operations
4.A class is a collection of properties

Which object-orientation concept matches the following description?
"Describe complex problems by focusing on the essential qualities."

1.Polymorphism
2.Inheritance
3.Abstraction
4.Encapsulation

Can we skip some details when defining our classes?

1.Yes, that's totally fine when those details are unessential
2.No, we can't skip any detail when designing software
3.Yes, but only if they already defined elsewhere
4.Yes, if we're using the agile methodology

Why should we hide the internal details of our classes?

1.To make sure that we're relying on abstraction
2.To reduce the size of the generated binary
3.To ensure that our classes can be easily subclassed
4.To protect the object from uncontrolled changes

What's the object-oriented term for packing together properties and methods in a class?

1.Inheritance
2.Encapsulation
3.Abstraction
4.Polymorphism

What does it indicate if any modification requires changes in other, unrelated parts of the system?

1.A modular system
2.A layered architecture
3.A tightly-coupled system
4.An object-oriented system

Which object-orientation concept can be used to reuse code?

1.Inheritance
2.Abstraction
3.Polymorphism
4.Data hiding

How can we override a method of a class?

1.We re-implement the method in a superclass and provide a different behavior
2.We change the original implementation and provide a different behavior
3.We re-implement the method in a subclass and provide a different behavior
4.We implement a method with a different name and provide a different behavior

Which of the following describes polymorphism?

1.Reusing the properties and the behavior from a superclass without coding
2.Working with instances of different classes without knowing their exact type
3.A way of describing complex problems by focusing on the essential qualities

What is Wireframing?

Similar to an architectural blueprint, a **wireframe** is a two-dimensional skeletal outline of a webpage or app. Wireframes provide a clear ov

Wireframes can be drawn by hand or created digitally, depending on how much detail is required.

Wireframing is a practice most commonly used by UX designers. This process allows all stakeholders to agree on where the information wi

When does wireframing take place?

The wireframing process tends to take place during the exploratory phase of the product life cycle. During this phase, the designers are testi

Armed with the valuable insights gathered from the user feedback, designers can build on the next, more detailed iteration of the product's

What is the purpose of wireframing?

Wireframes serve three key purposes: They keep the concept user-focused, they clarify and define website features, and they are quick and

Wireframes keep the concept user-focused

Wireframes are effectively used as communication devices; they facilitate feedback from the users, instigate conversations with the stakeho

Wireframing is the perfect way for the designers to gauge how the user would interact with the interface. By using devices such as Lorem Ip

These insights help the designer to understand what feels intuitive for the user, and create products that are comfortable and easy to use.

Wireframes clarify and define website features

When communicating your ideas to clients, they may not have the technical lexicon to keep up with terms like “hero image” or “call to acti

It also enables all stakeholders to gauge how much space will need to be allocated for each feature, connect the site’s information architectu

Seeing the features on a wireframe will also allow you to visualize how they all work together—and may even prompt you to decide to rem

Wireframes are quick and cheap to create

The best part about wireframes? They’re incredibly cheap and easy to create. In fact, if you have a pen and paper to hand, you can quickly s

Often, when a product seems too polished, the user is less likely to be honest about their first impressions. But by exposing the very core of

What are the different types of wireframes?

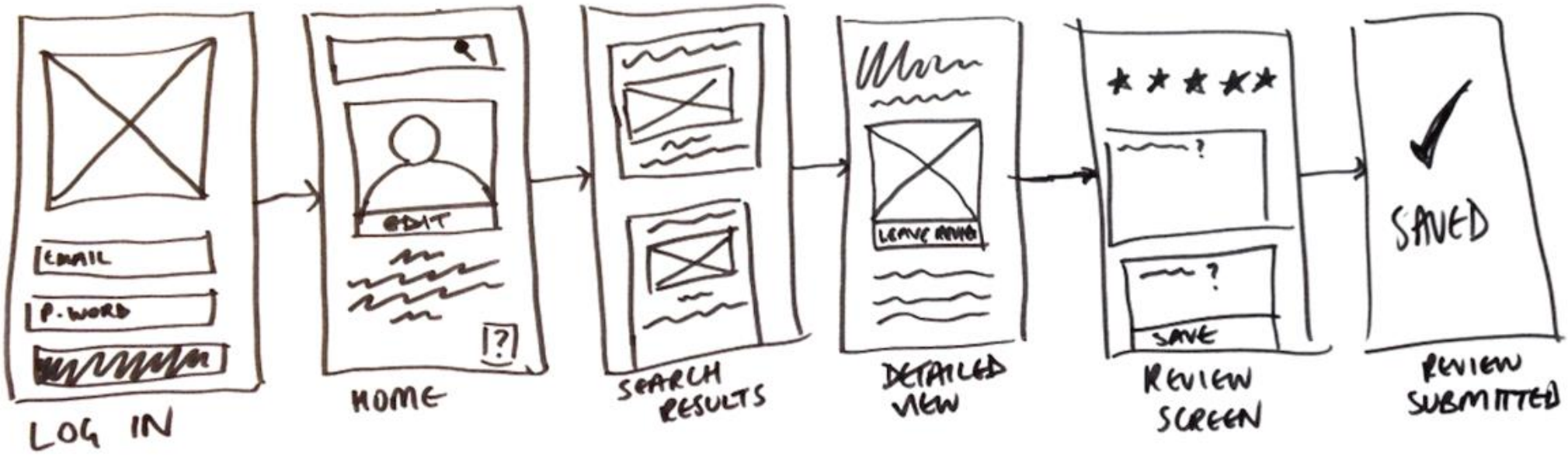
There are three main types of wireframes: **low-fidelity** wireframes, **mid-fidelity** wireframes, and **high-fidelity** wireframes. The most signif

Low-fidelity wireframes

Low-fidelity wireframes are basic visual representations of the webpage and usually serve as the design's starting point. As such, they tend

Low-fidelity wireframes omit any detail that could potentially be a distraction and include only simplistic images, block shapes, and mock c

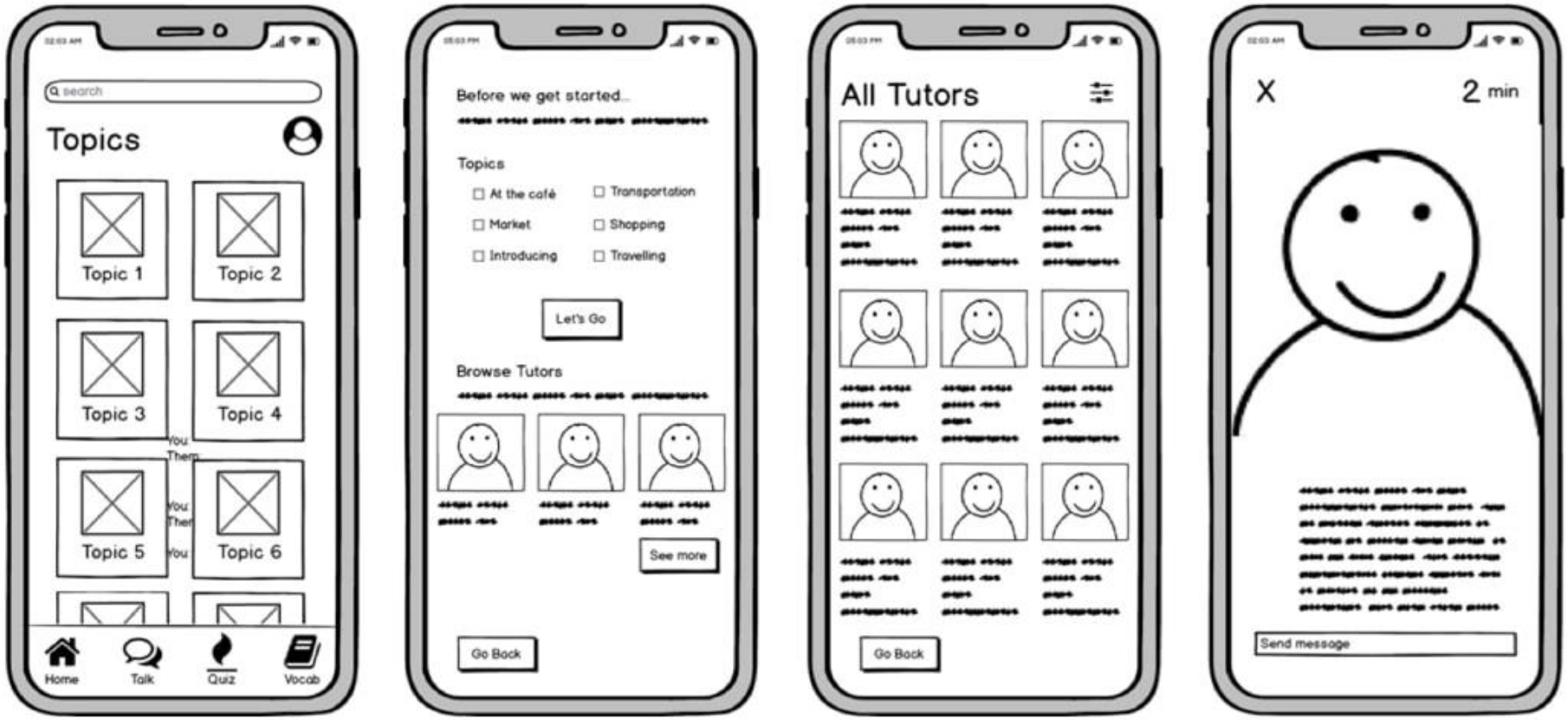
Low fidelity wireframes are useful for starting conversations, deciding on navigation layout, and mapping the user flow. In short, low-fideli



Mid-fidelity wireframes

The most commonly used wireframe of the three, mid-fidelity wireframes feature more accurate representations of the layout. While they st

Varying text weights might also be used to separate headings and body content. Though still black and white, designers can use different sh



High-fidelity wireframes

Finally, high-fidelity wireframes boast pixel-specific layouts. Where a low-fidelity wireframe may include pseudo-Latin text fillers and grey

This added detail makes high-fidelity wireframes ideal for exploring and documenting complex concepts such as menu systems or interacti

High-fidelity wireframes should be saved for the latter stages of the product's design cycle.

What is included in a wireframe?

As we touched on earlier, how many features are included in a wireframe depends largely on whether the wireframe is low, mid or high fide

High-fidelity wireframes may also include navigation systems, contact information, and footers. Typography and imagery should not be par

Wireframes are traditionally created in greyscale, so designers often play around with shading—using lighter shades of grey to represent lig

As wireframes are two-dimensional, it's important to bear in mind that they don't do well with showing interactive features of the interface

What tools are used to create wireframes?

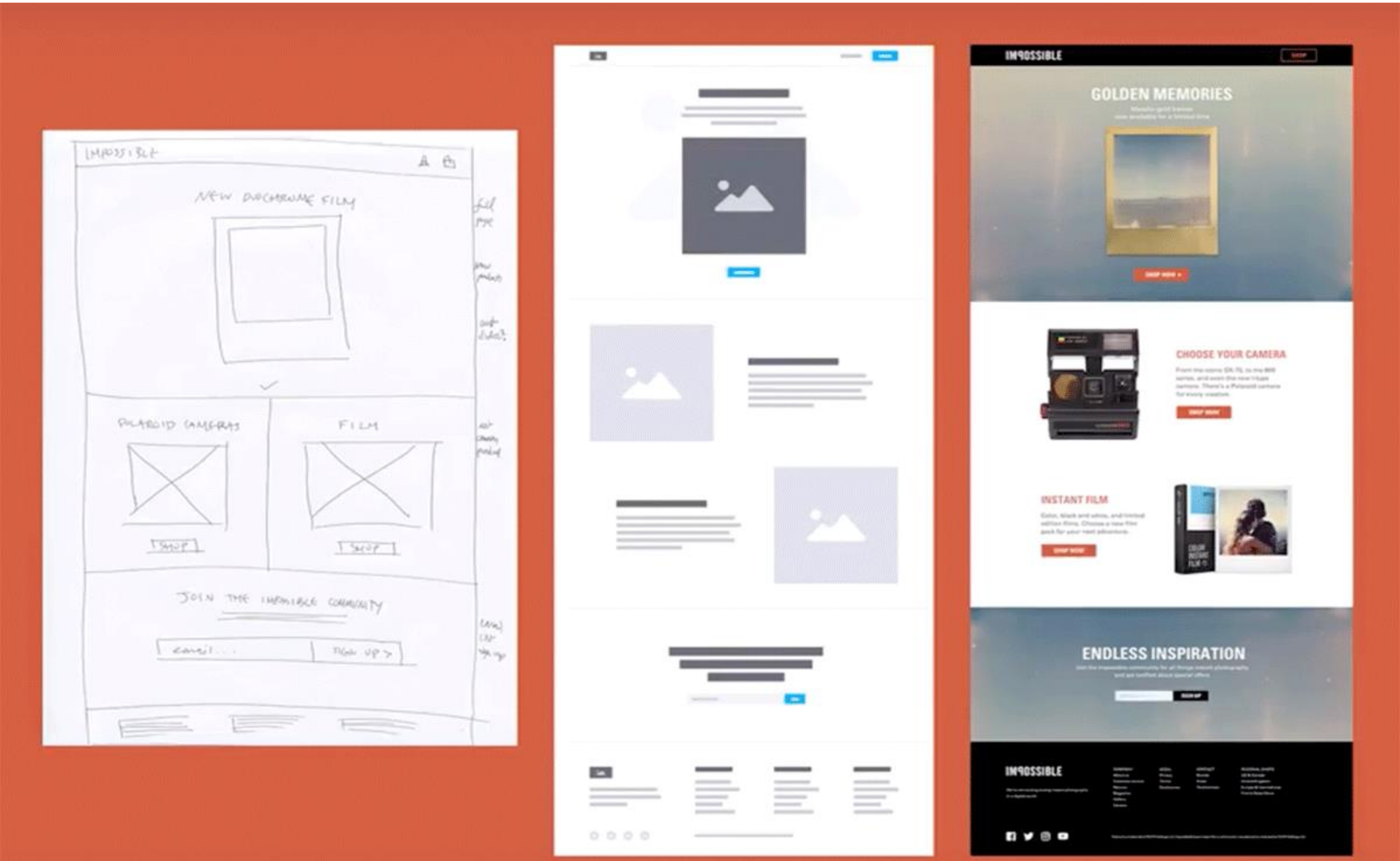
In today's technology-abundant landscape, designers have a myriad of advanced wireframing tools and programs at their fingertips. Built-in

One of the best-known wireframing tools on the scene is **Sketch**, which uses a combination of art boards and vector design shapes to enable

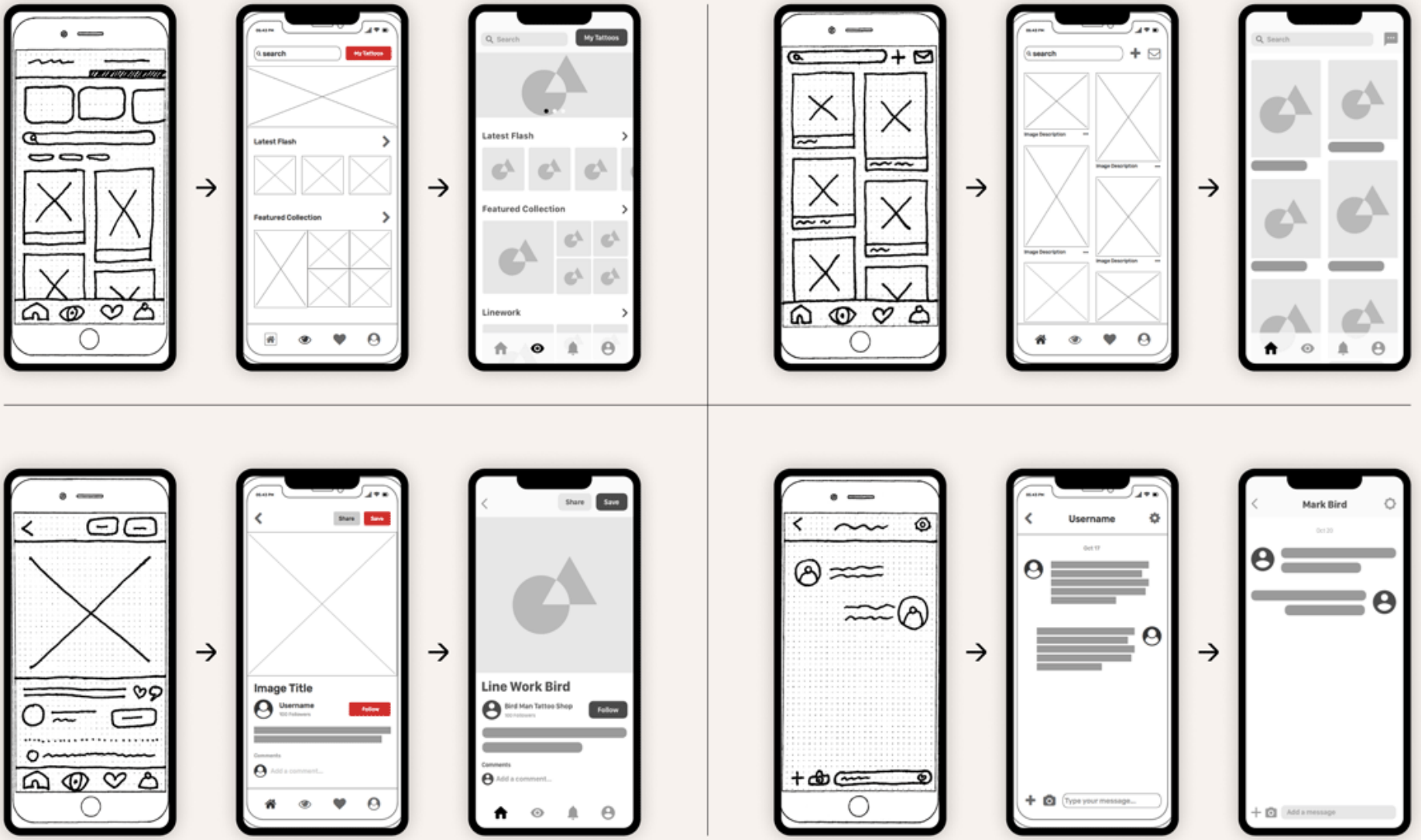
Need something more professional than a paper wireframe but aren't striving for pixel perfection? Opt for the equally popular **Balsamiq**, a

Examples of wireframes

To give you an idea of the variety of ways wireframes can be created (and to provide you with some well-needed inspiration for your own v




ADD A NOTE
Title
SAVE
UPLOAD
TAKE QUIZ
STUDY BOARD
TAKE NOTES
How do deny?
Topic 1
Topic 2
Topic 3
Topic 4
STABILO BOSS



This is also a video that shows you the transition from low-fidelity wireframe to high-fidelity wireframe: **https://www.youtube.com/watch?v=TlMWH_5BmvY**

Prototype

In software development, a prototype is a rudimentary working model of a product or information system, usually built for demonstration p

The word prototype comes from the Latin words proto, meaning original, and typus, meaning form or model. In a non-technical context, a p

Fundamental Object-Oriented Analysis and Design Concepts

Building an object-oriented application requires some preliminary steps. These steps are similar regardless of the development methodology.

First, we need to **collect the requirements**. During the requirements collection phase, we answer the following questions:

What's the problem we're trying to solve?

What does our app or framework need to do to accomplish that functionality?

The requirements collection step involves a lot of brainstorming and discussion.

Once we come to an agreement, we need to document our ideas.

The requirements need to be **as clear as possible**. Only write down the decisions that **underline what the system is going to do.**

Vague thoughts will lead to conflicts later on.

Once the requirements are clear, we come up with a **description of the software system**. We should describe the app f**rom the user's perspective**.

Depending on the project, we may pick an Agile or a Waterfall methodology. For Agile projects, it is completely fine if we don't provide an accurate description. We can still fill the gaps or refine our thoughts later on.

The point here is to gain as much clarity as needed to start the next step.

The step of describing the app may include the creation of visual mockups, wireframes, or even prototypes.

If it helps in communicating your thoughts to the client, then do it.

I've used wireframes and nonfunctional prototypes for most of my projects.

These prototypes proved to be extremely useful, especially if the client was not familiar with the platform or they had no specific expectations.

Let's say a customer asks you to create an iOS version of their Android app.
A prototype will help the client understand that the iOS version will look and behave differently.
By communicating our vision precisely, we avoid surprises and misleading expectations.
Next comes the third phase. During this step, we aim to identify the things that form our system.
These are the potential players that have a specific, well-defined role in our application.
Picking the essential entities won't be challenging if we did a good job during the previous two steps.
We'll realize that we need a class that represents, say, an item that has a name, a price and some other attributes, or a class responsible for securely communicating with the server.
Another class may manage your local persistence, and so on.
In the final phase, we describe the behavior of our system in a formal way. This last step is about creating visual representations of our classes, their attributes and behavior.
We also model the interaction between the objects. We rely on the Unified Modeling Language, or UML for short.
UML is a form of graphical notation that provides a set of standard diagrams. These diagrams let us describe object-oriented systems in a standard way.

Step 1: collect requirements

- Identify the problems we want to solve
- Clarify the functionality required to solve the problems
- Document important decisions

Step 2: describe the system

- Describe the system from the user's point of view
- Create wireframes and prototypes if needed

Step 3: identify the classes

- Object-oriented software design

Step 4: create diagrams

- Create diagrams in the UML language that describes classes (class diagrams), and interactions between th

Collecting Requirements

The initial step of building a software system is crucial. It's often called requirement-collection phase or requirements analysis. But regardle
software design. Requirement means "a thing that is needed or wanted." And that's exactly what we need to focus on during this initial step:
we must clarify what's needed or wanted in our application.
The **features of the system** are the so-called **functional requirements**.

Functional requirements represent what the app needs to provide feature-wise, how it should react to a particular input, or what the expecte
Let's say you are about to develop an app for runners.

Functional requirements

- Represent the features
- Define how to react to an input
- Determine the expected behavior

You should answer questions like the following:
Should the actual speed always be visible on the main screen?
Do we allow imperial or metric units?
Should we make this configurable by the user or automatically adjust the units based on the phone's settings instead?
We'll also usually have **non-functional requirements**.
These are the requirements that are **not directly related to a feature or behavior of the software system, but are important nonetheless**.
Think of **performance requirements-**
you don't want to ruin the user experience with an unresponsive app.
You also may need to address **legal requirements**.
Does the app collect sensitive user data?
Does it allow users to browse the Internet?
Documentation and support are other non-functional requirements. Your software may need to adhere to certain standards or regulations. **Nonfunctional requirements are equally important.**
Ignoring them may cause serious legal issues and all sorts of other problems.
Now, how do we handle this?
There are different ways to gather the requirements.
The easiest way is just to write them down.

Here's an example of a functional requirements of some application:

- *The app must store travel expenses organized by trip*.
- *Each trip must have a home currency.*
- *The default currency is fetched from the phone's settings.*
- *User settings must override the default home currency.*
- *Expenses can be entered in any of the supported currencies*.
- *The app must automatically convert the amounts to the home currency.*

Nonfunctional requirements:

- The app must run on iOS 9 and newer versions.
- The app must avoid unnecessary network roundtrips to reduce data roaming fees and preserve battery.
- The app must include the support email and the link to the app's website.

These are short, concise phrases in the form:

"**The app or system** must do **this or that**."

You don't want to write lengthy descriptions. And feel free to adapt this format to your needs.
You should eventually capture your requirements digitally, but at early stages, **pen and paper or a whiteboard are also fine.**
Just make sure you save them somehow - by taking a photo, for example.

There are also more formal **ways**, **tools**, and **systems** that support the requirements collection step.

Ways to collect requirements:

- write them down
- Use tools/systems

To summarize, the requirements collection step boils down to this: We need to formulate what our software must do and which are the constraints and boundaries we need to consider.
If we are using a **Waterfall** approach, we need to clarify **all the requirements in advance**. For **agile** projects
it's perfectly acceptable if **we continue without having all the answers.** We may even miss some of the questions.
Agile lets us revisit and refine the requirements as we iterate through the software development process.

Mapping Requirements to Technical Descriptions

Once we have gathered the requirements, we can feed them to the next step of the software design process.
This is where we provide short, accurate descriptions of our systems functionality from the user's perspective.
One way of documenting our system's features is through **use cases**.

A use case needs a **title**, something like "Create a new trip," "Add expense," or "Convert currencies."
Note that each use case should represent a distinct functionality.
Next, we define the **actor** who's using this functionality. We call it an actor since it can represent a user who's interacting with the app, but also a non-human entity like another system.

Use Case:
Title: short and descriptive text
Actor: user
Scenario: explains how the software works

Then we describe the **details** of this specific use case; this is called the **scenario**. Here, we should write one or more sentences that explain

Example:

Create New Trip
Actor: Mobile User

- **"The user can initiate the creation of a new trip from the main screen."**
- **"The title is mandatory."**
- **"All the other settings are optional."**
- **"Optionally, the user can write a short description and set a start and end date for the trip."**
- **"The app assigns a default home currency based on the phone's settings, users can override the default home currency with any of the supported currencies."**
- **"The app allows setting the budget for the trip."**
- **"The setting is optional."**
- **"Also, the user can assign a custom thumbnail to a trip."**
- **"The user can save the trip or cancel the trip creation process."**

You can write this as a paragraph or as a bulleted list.
The format doesn't really matter, but it's important to avoid technical terms.
Again, this description should be understood by all stakeholders, including the end users.
The format of the use case document may vary from company to company.
Some may include additional details, but that won't change the essence of it.

The use case document aims to provide a clear and human-friendly description, what a specific part of a software does and how the actors interact with it.

And it is a textual description.

User Story

Very brief description of a feature

User stories are another common way of describing certain features or parts of our application.
User stories are shorter than use case descriptions, usually only one to two sentences long.
They typically follow this format:

As a <type of user> I want <some goal> so that <some reason>

Examples:

- **As a user, I want to add notes to my expenses so that I can identify them later on.**
- **As a power user, I want to retrieve the app's database file so that I can inspect it on any computer.**

If you can't describe the user story in one or two sentences, you may need to split it into multiple smaller user stories.
These larger user stories are known as **epics**.

Epic

Describe a bigger chunk of a functionality. Should be split into smaller user stories.

Epics cover a bigger chunk of functionality like in the following case:
"As a traveler, I want to track my expenses while abroad so that I don't exceed my budget."

This epic could be split into many other stories, including these:

"As a user, I want to create new trips so that I can track each of my trips individually."
"As a business traveler, I want to tag my business trips so that I can separate them from my private travels."

User stories are often written on sticky notes or index cards.
You will see them arranged on walls or tables during meetings and discussions.
And like use case descriptions, user stories don't capture the feature details. They serve as discussion starters instead.

User stories are about **communication** and you will usually see them in **Agile** projects,
Whereas **use case descriptions** are preferable when employing **Waterfall** methodologies.

Use user stories to ignite discussions instead of describing details

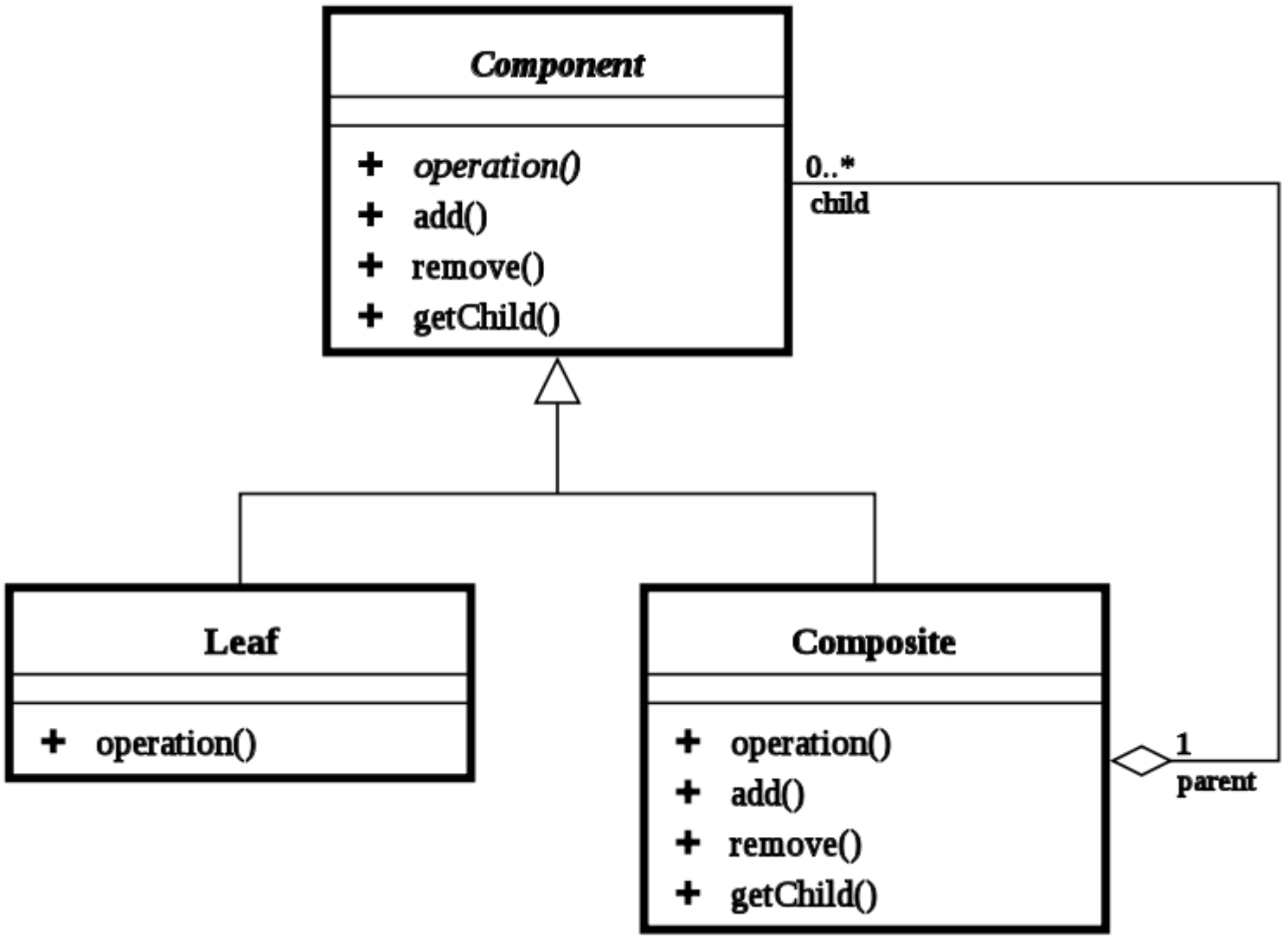
A Common Descriptive Language

The first two steps of the object-oriented analysis don't require any special tool or design language.
We only need text editing software.
Even a piece of paper or a whiteboard would be sufficient to collect the requirements and jot down the use cases or user stories.
The next steps require us to depict the classes that form our system, how they behave and what attributes they need.
We also need to visualize how the objects interact with each other.
The development community faced this very same problem.
The lack of a commonly accepted design language led to the proliferation of different non-standard approaches.
We could also try to come up with a way to draw everything from classes to object interactions.
But luckily we don't have to.
The Unified Modeling Language is a common design language that was released in 1997.
UML provides a set of standard diagram types that can be used to describe both the structure and the behavior of software systems.

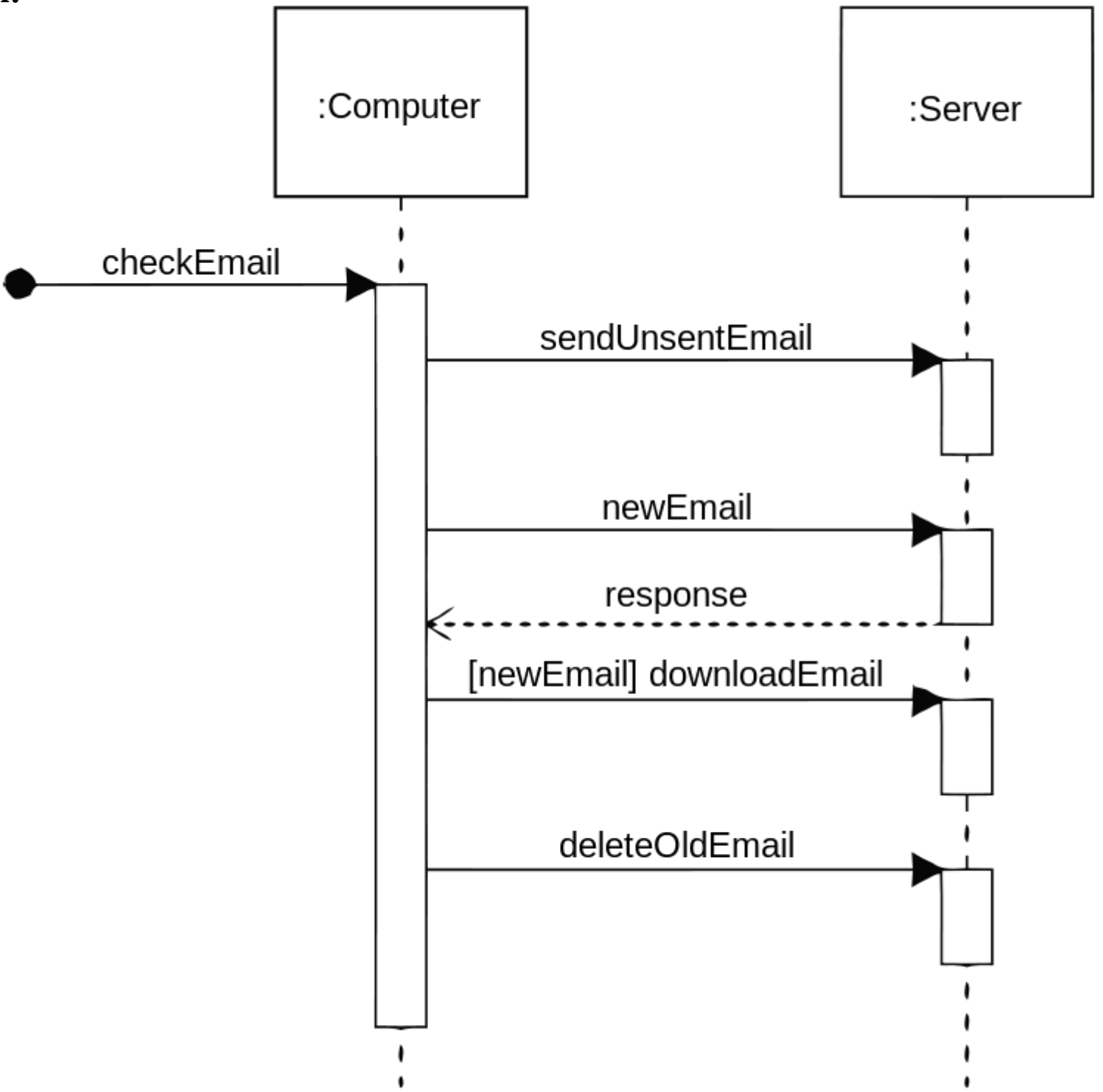
UML diagrams

Standard graphical notation used to describe object-oriented systems

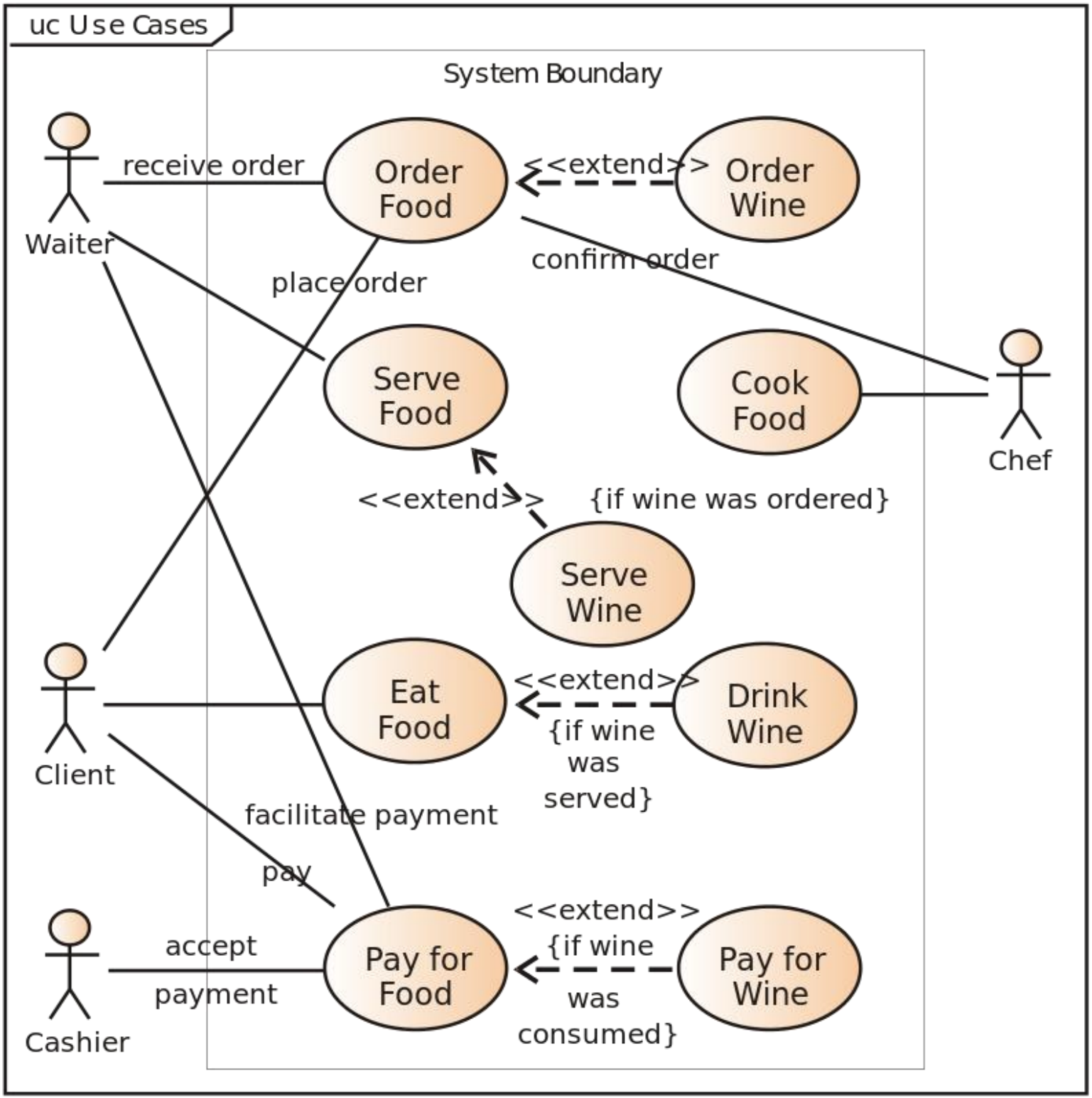
Class diagram:



Sequence diagram:



Use Case diagram:



What's UML?

Understanding a software system just by looking at its source code can be very time consuming. And communicating ideas about software d
The Unified Modeling Language-in short, UML-was introduced to solve this problem. UML is not a textual programming language, but rat

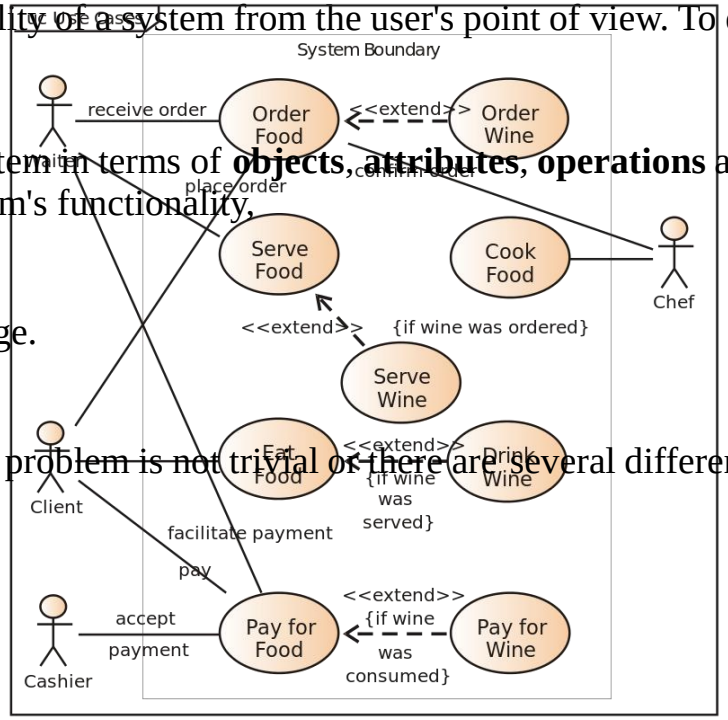
We can use these diagrams to describe the objects that form a system and their interactions.

Unified Modeling Language

A graphical notation used to communicate the design of software systems

UML has many diagram types. We'll be discussing the most common ones:
The **use case diagram** describes the functional model of a system - that is, the functionality of a system from the user's point of view. To de

Use Case diagram:



We'll talk about the **class diagram**, which can be used to describe the structure of a system in terms of **objects**, **attributes**, **operations** and
UML let us model dynamic behavior, too. The **behavioral diagrams** describe the system's functionality, focusing on what happens and the **interactions between objects.**
We'll talk about the actual diagrams shortly.
The best part about UML is that it's independent of any particular programming language.
We can start coding object-oriented software based on UML diagrams.
If those diagrams are detailed enough, they can be converted to source code.
Software developers often find themselves in situations where the solution to a specific problem is not trivial or there are several different

It may be tempting to open up your IDE and just start coding.
The next thing you know, hours have disappeared and you are desperately searching StackOverflow for the answer.
However, it's hard to find a solution if we couldn't first formulate the question.
We need to figure out what to implement before writing a single line of code. That's when UML comes in handy. Whenever something is u

The benefits of this approach are twofold.
First, by thinking about classes, objects and interactions, we gain a deeper understanding of what
should be implemented without being distracted by crashing IDEs or strange compiler error messages.
Secondly, a design helps us communicate our ideas with other developers effectively. We can use UML
diagrams as a starting point for discussions and improvements without having to delve into source code.
Although checking the actual code is useful in many situations,
it will often distract us from answering the real questions, and turn the design discussion into a code
inspection.
Another frequent use of UML is drawing diagrams from existing code.
This technique is called reverse engineering, and it helps uncover the dirty little secrets of undocumented software systems.
We can use UML to create a detailed blueprint of a system. Detailed UML blueprints are usually required for software developing using a V
Although UML is excellent at modeling object-oriented systems, the fact that it's platform and programming-language independent make it
UML has been used in multidisciplinary areas, including scientific research, transportation, banking
and defense.

Use Case diagrams

Use Case Diagram captures the system's functionality and requirements by using **actors** and **use cases**. Use Cases model the services, tasks,

Why Use Case Diagram?

A Use Case consists of **use cases**, persons, or various things that are invoking the features called as actors and the elements that are responsi

When to apply use case diagrams?

A use case diagram doesn't go into a lot of detail—for example, don't expect it to model the order in which steps are performed. Instead, a p

UML is the modeling toolkit that you can use to build your diagrams. Use cases are represented with a labeled oval shape. Stick figures repr

UML use case diagrams are ideal for:

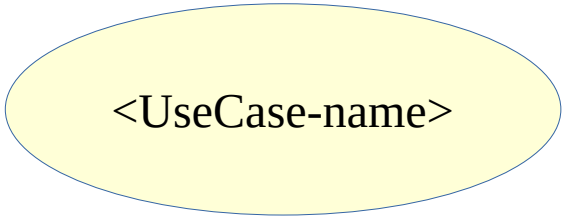
- **Representing the goals of system-user interactions**
- **Defining and organizing functional requirements in a system**
- **Specifying the context and requirements of a system**
- **Modeling the basic flow of events in a use case**

Use-case diagram notations

Following are the common notations used in a use case diagram:

Use-case:

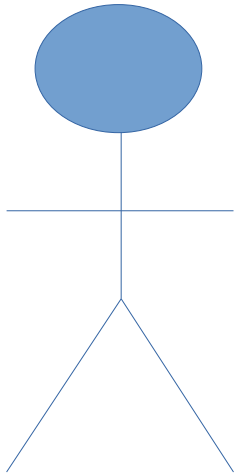
Use cases are used to represent high-level functionalities and how the user will handle the system. A use case represents a distinct functiona



UML UseCase notation

Actor:

It is used inside use case diagrams. The actor is an entity that interacts with the system. An actor can be human or non-human. A user is the



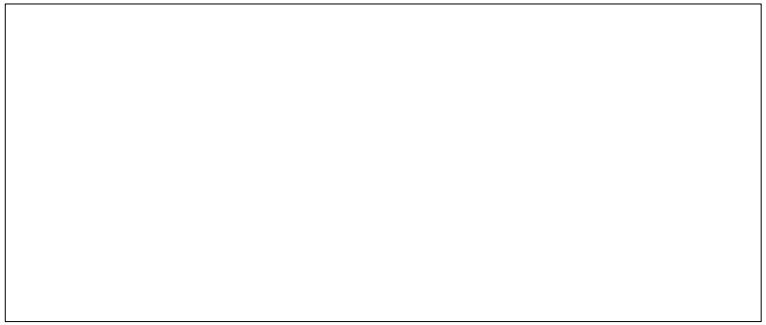
<Actor-name>

Association:

A line between actors and use cases. In complex diagrams, it is important to know which actors are associated with which use cases.

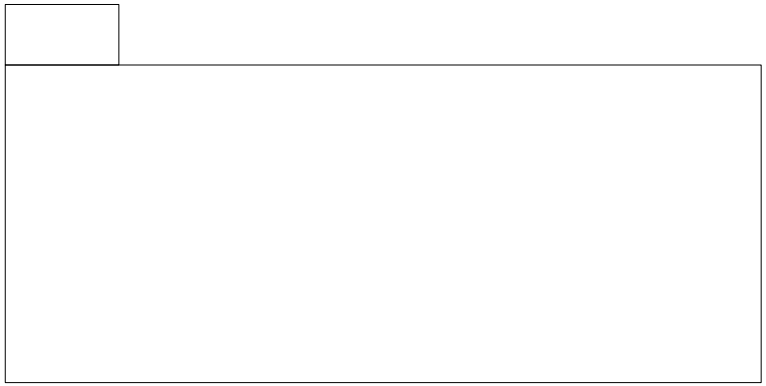
System boundary boxes:

A box that sets a system scope to use cases. All use cases outside the box would be considered outside the scope of that system. For example



Packages:

A UML shape that allows you to put different elements into groups. Just as with component diagrams, these groupings are represented as fil



How to draw a use-case diagram?

To draw a use case diagram in UML first one need to analyse the entire system carefully. You have to find out every single function that is p

A use case is nothing but a core functionality of any working system. After organizing the use cases, we have to enlist the various actors or t

After the actors and use cases are enlisted, then you have to explore the relationship of a particular actor with the use case or a system. One

Following rules must be followed while drawing use-case for any system:

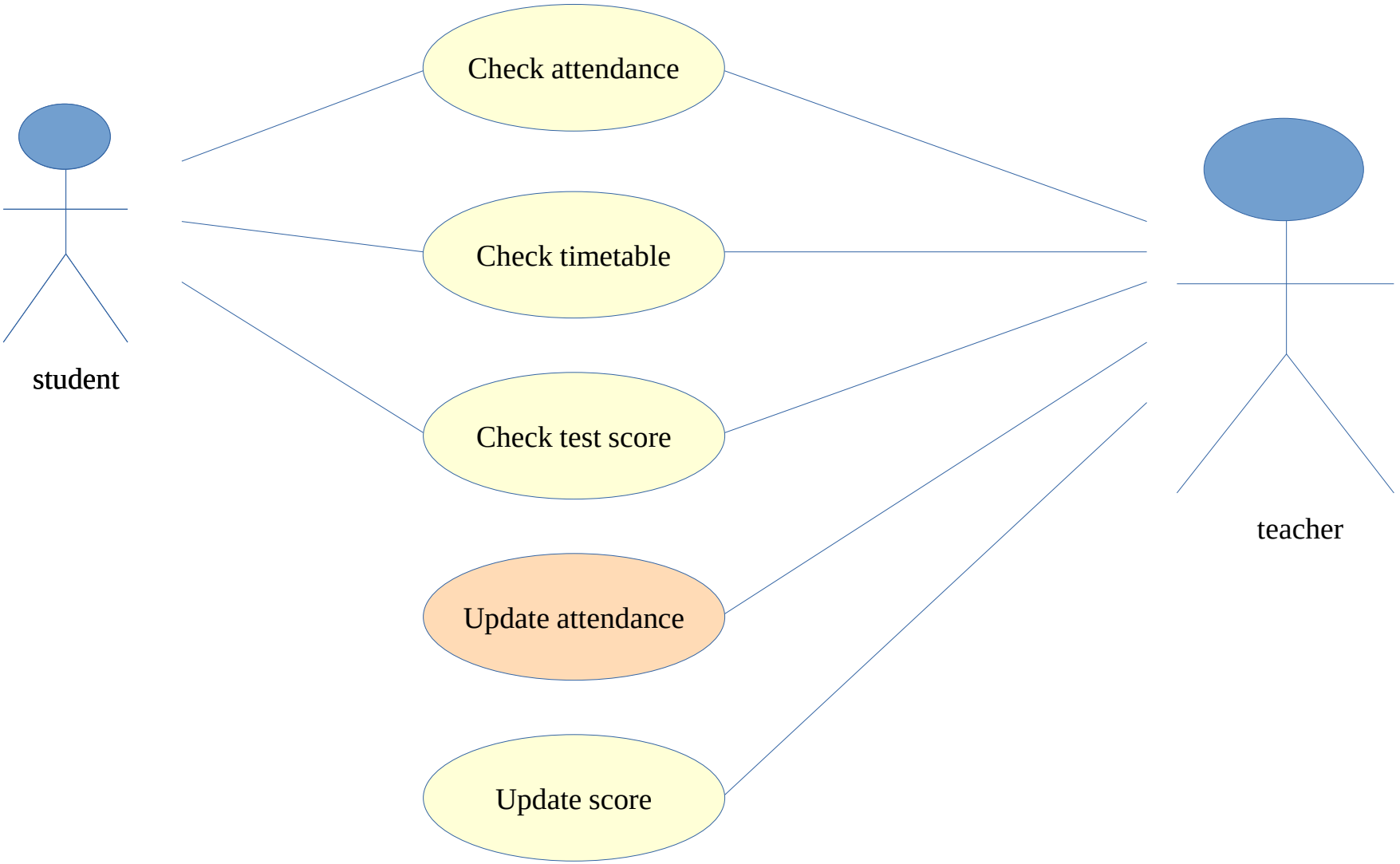
1.The name of an actor or a use case must be meaningful and relevant to the system.
2.Interaction of an actor with the use case must be defined clearly and in an understandable way.
3.Annotations must be used wherever they are required.
4.If a use case or an actor has multiple relationships, then only significant interactions must be displayed.

Tips for drawing a use-case diagram

1.A use case diagram should be as simple as possible.
2.A use case diagram should be complete.
3.A use case diagram should represent all interactions with the use case.
4.If there are too many use cases or actors, then only the essential use cases should be represented.
5.A use case diagram should describe at least a single module of a system.
6.If the use case diagram is large, then it should be generalized.

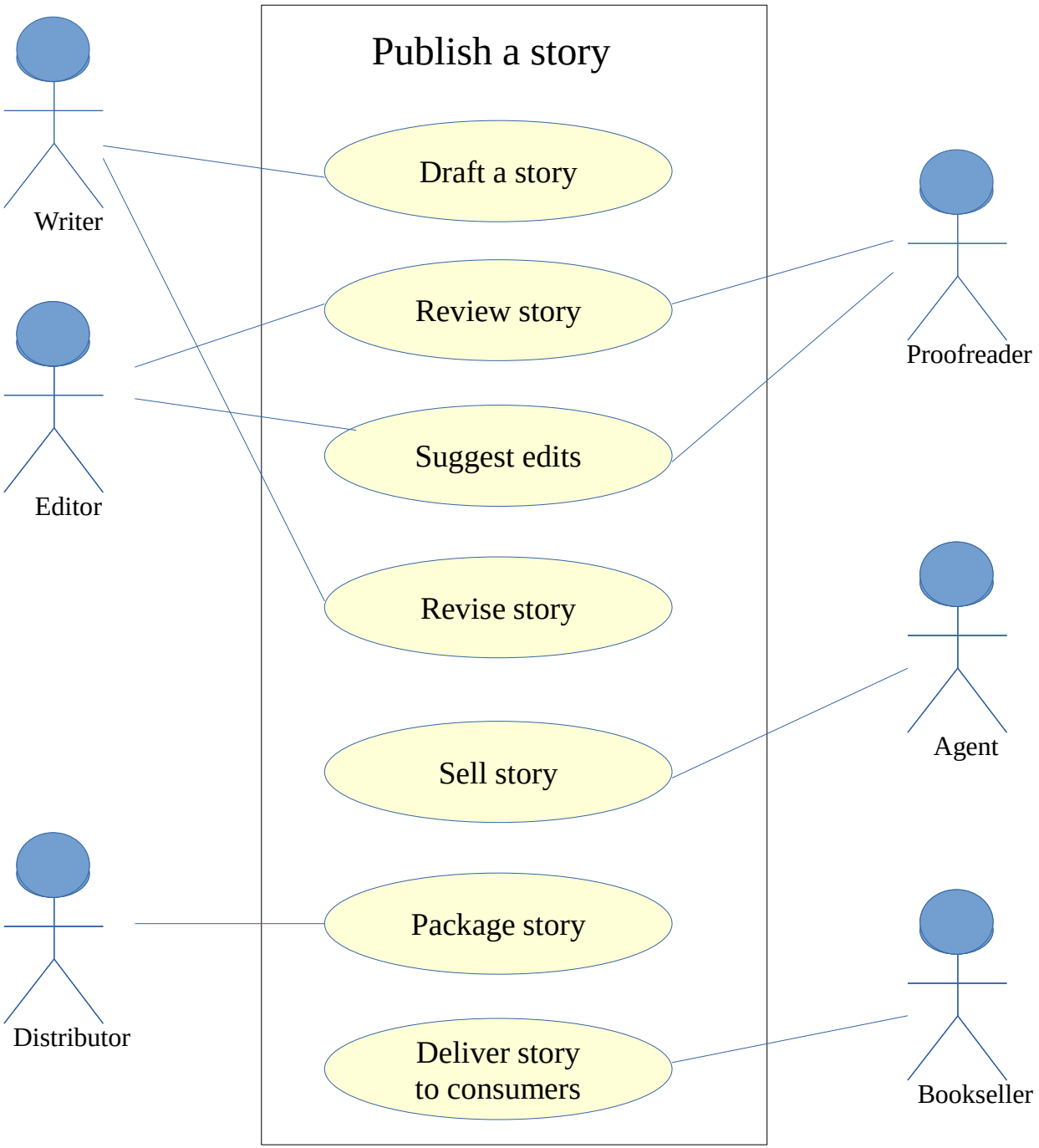
An examples of a use-case diagram

Following use case diagram represents the working of the student management system:



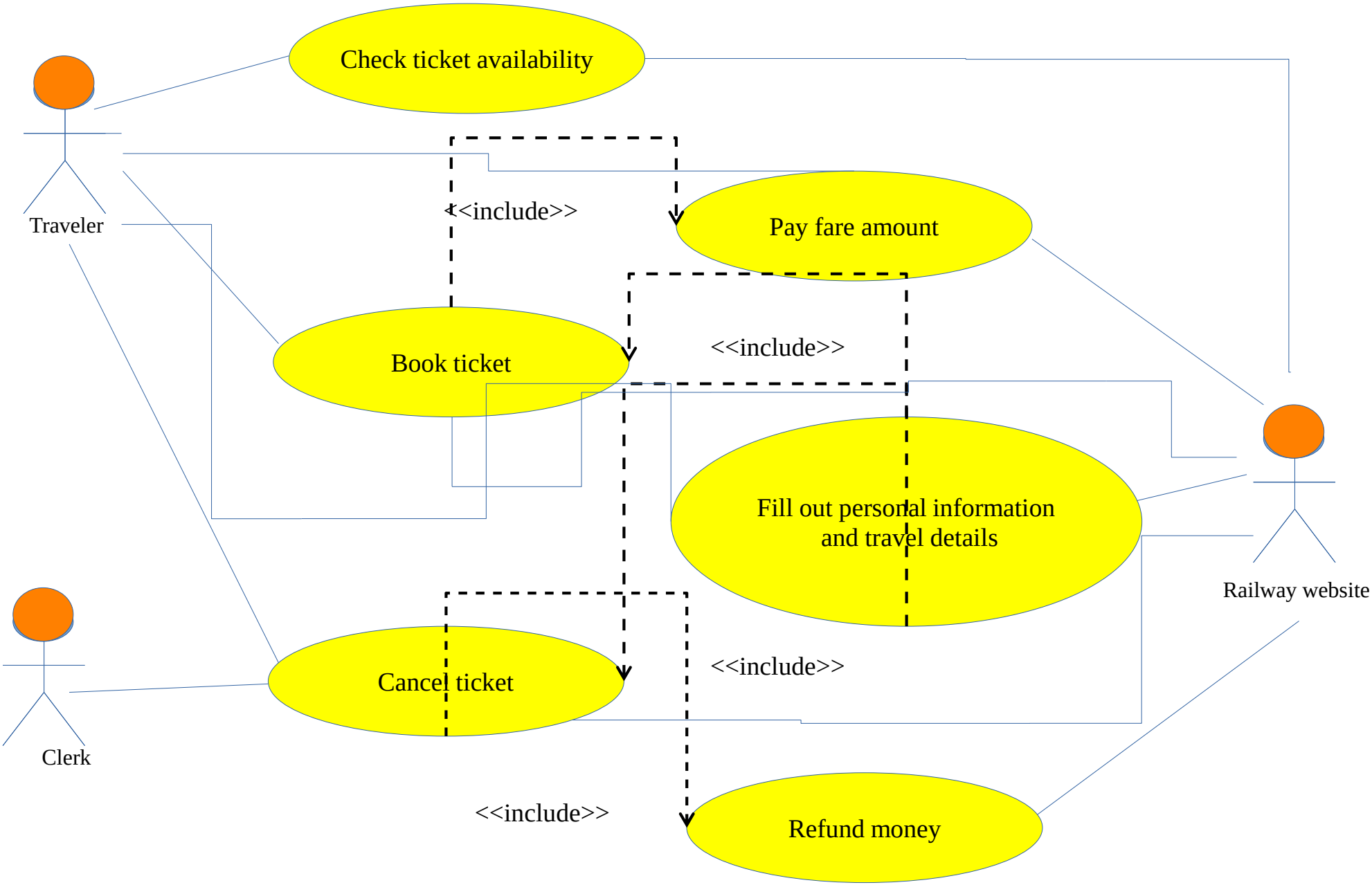
Book publishing use case diagram example

This use case diagram is a visual representation of the process required to write and publish a book. Whether you're an author, an agent, or a



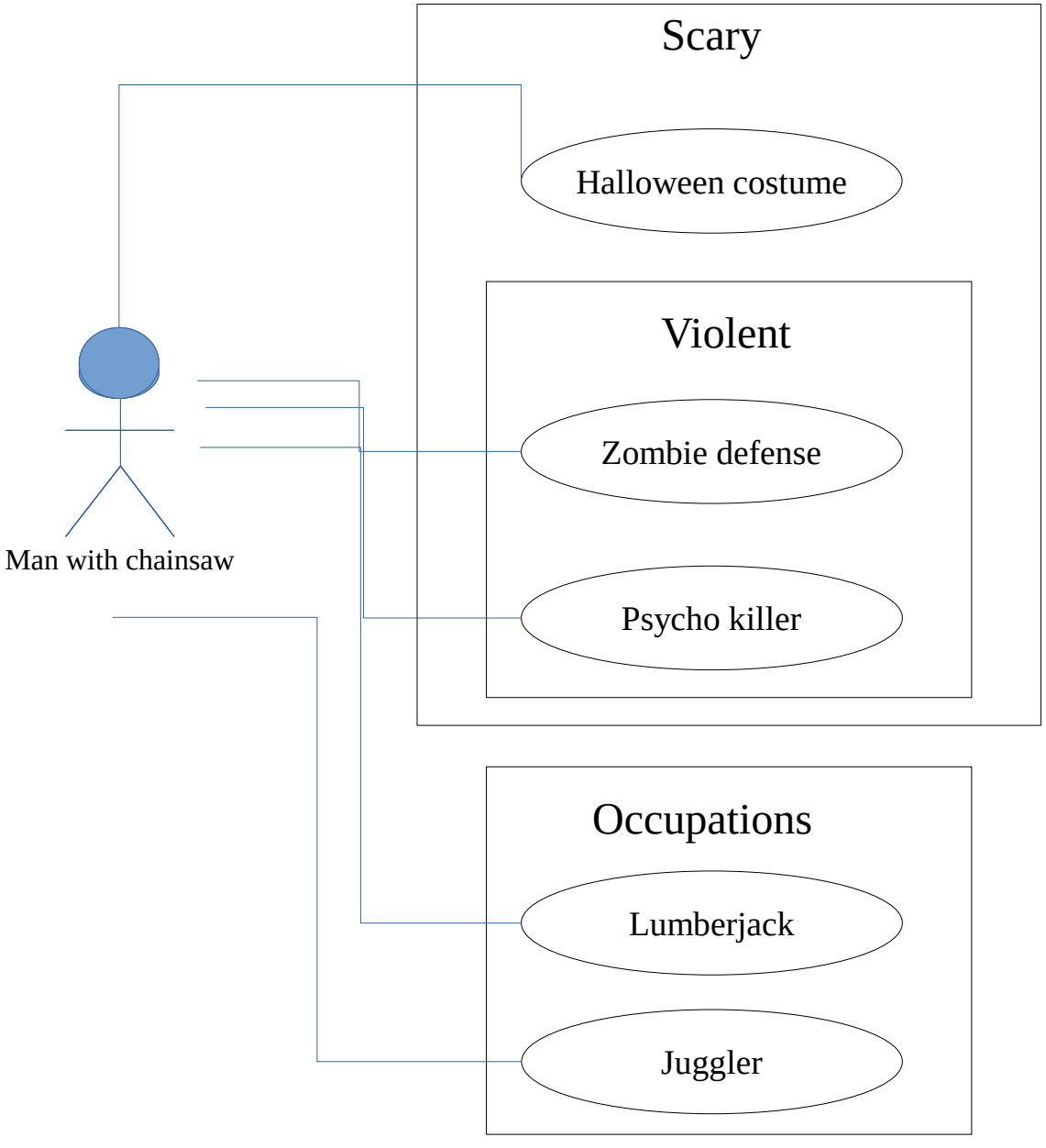
Railway reservation use case diagram example

You can adapt this template for any process where a customer purchases a service.



Chainsaw use case diagram example

Consider this example: A man with a chainsaw interacts with the environment around him. Depending on the situation and the context of th



Class diagrams

What is a class diagram in UML?

The Unified Modeling Language (UML) can help you model systems in various ways. One of the more popular types in UML is the class d

UML was set up as a standardized model to describe an object-oriented programming approach. Since classes are the building block of obje

The class shape itself consists of a rectangle with three rows. The top row contains the name of the class, the middle row contains the attribu

Benefits of class diagrams

Class diagrams offer a number of benefits for any organization. Use UML class diagrams to:

- **Illustrate data models for information systems, no matter how simple or complex.**
- **Better understand the general overview of the schematics of an application.**
- **Visually express any specific needs of a system and disseminate that information throughout the business.**
- **Create detailed charts that highlight any specific code needed to be programmed and implemented to the described structure.**
- **Provide an implementation-independent description of types used in a system that are later passed between its components.**

Basic components of a class diagram

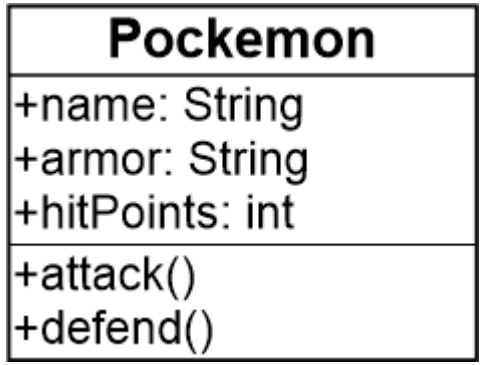
The standard class diagram is composed of three sections:

.Upper section: Contains the name of the class. This section is always required, whether you are talking about the classifier or an obj
.Middle section: Contains the attributes of the class. Use this section to describe the qualities of the class. This is only required when
.Bottom section: Includes class operations (methods). Displayed in list format, each operation takes up its own line. The operations d

Member access modifiers

All classes have different access levels depending on the access modifier (visibility). Here are the access levels with their corresponding sym

.Public (+)
.Private (-)
.Protected (#)
.Package (~)
.Derived (/)
.Static (underlined)



Member scopes

There are two scopes for members: **classifiers** and **instances**.

Classifiers are **static members** while **instances** are the **specific instances of the class**. If you are familiar with basic OO theory, this isn't an

Additional class diagram components

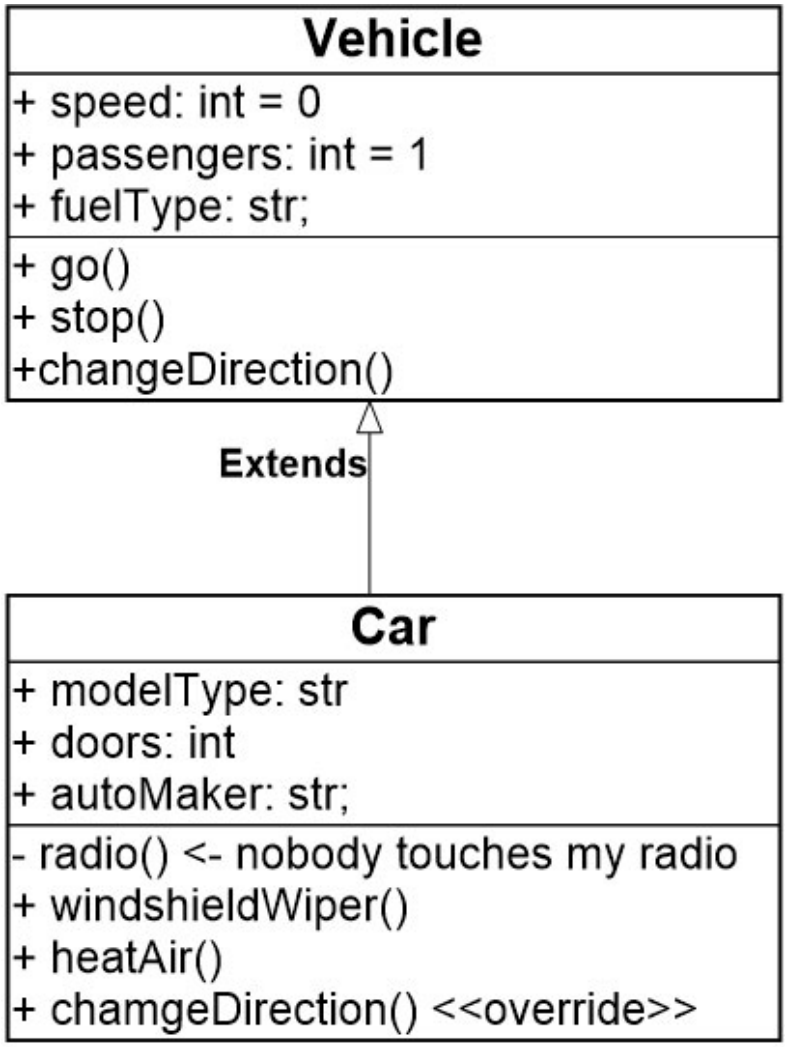
Depending on the context, classes in a class diagram can represent the main objects, interactions in the application, or classes to be programm

- **Classes**: A template for creating objects and implementing behavior in a system. In UML, a class represents an object or a set of objects that
- **Name**: The first row in a class shape.
- **Attributes**: The second row in a class shape. Each attribute of the class is displayed on a separate line.
- **Methods**: The third row in a class shape. Also known as operations, methods are displayed in list format with each operation on its own line
- **Signals**: Symbols that represent one-way, asynchronous communications between active objects.
- **Data types**: Classifiers that define data values. Data types can model both primitive types and enumerations.
- **Packages**: Shapes designed to organize related classifiers in a diagram. They are symbolized with a large tabbed rectangle shape.
- **Interfaces**: A collection of operation signatures and/or attribute definitions that define a cohesive set of behaviors. Interfaces are similar to c
- **Enumerations**: Representations of user-defined data types. An enumeration includes groups of identifiers that represent values of the enume
- **Objects**: Instances of a class or classes. Objects can be added to a class diagram to represent either concrete or prototypical instances.
- **Artifacts**: Model elements that represent the concrete entities in a software system, such as documents, databases, executable files, software

Interactions

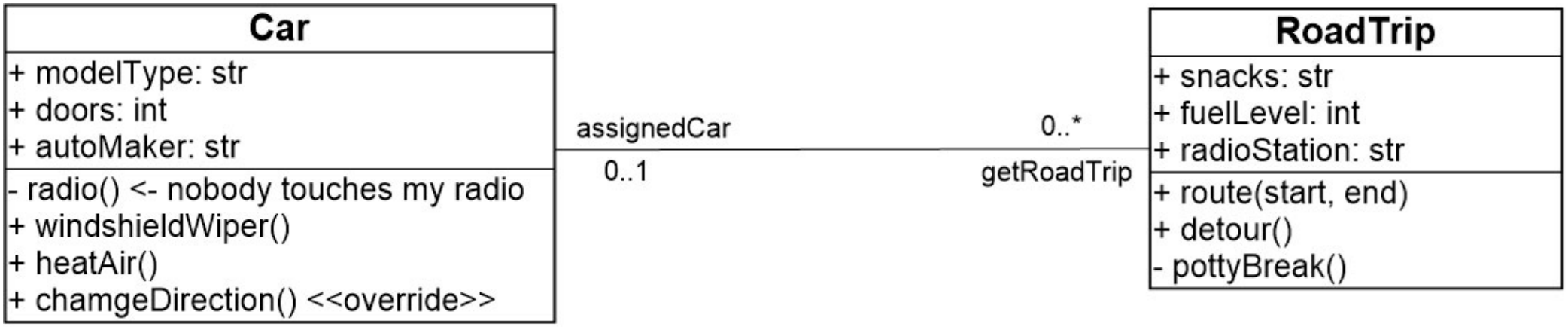
The term "interactions" refers to the various relationships and links that can exist in class and object diagrams. Some of the most common in

Inheritance: The process of a child or sub-class taking on the functionality of a parent or superclass, also known as generalization. It's symb



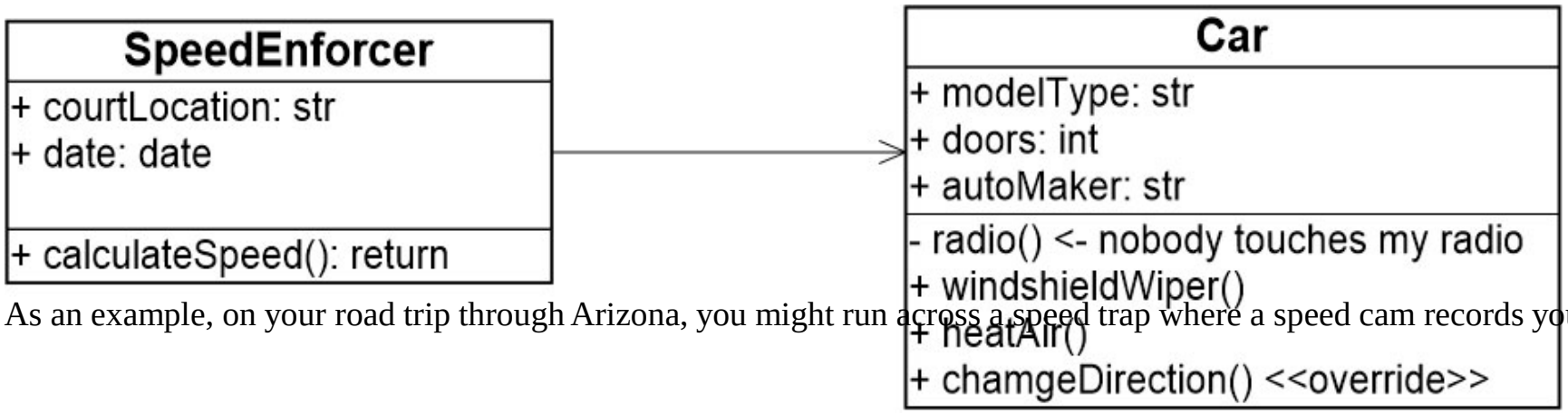
In this example, the object "Car" would inherit all of the attributes (speed, numbers of p

Bidirectional association: The default relationship between two classes. Both classes are aware of each other and their relationship with the



In the example above, the Car class and RoadTrip class are interrelated. At one end of the line, the Car takes on the association of "assigned

Unidirectional association: A slightly less common relationship between two classes. One class is aware of the other and interacts with it. U



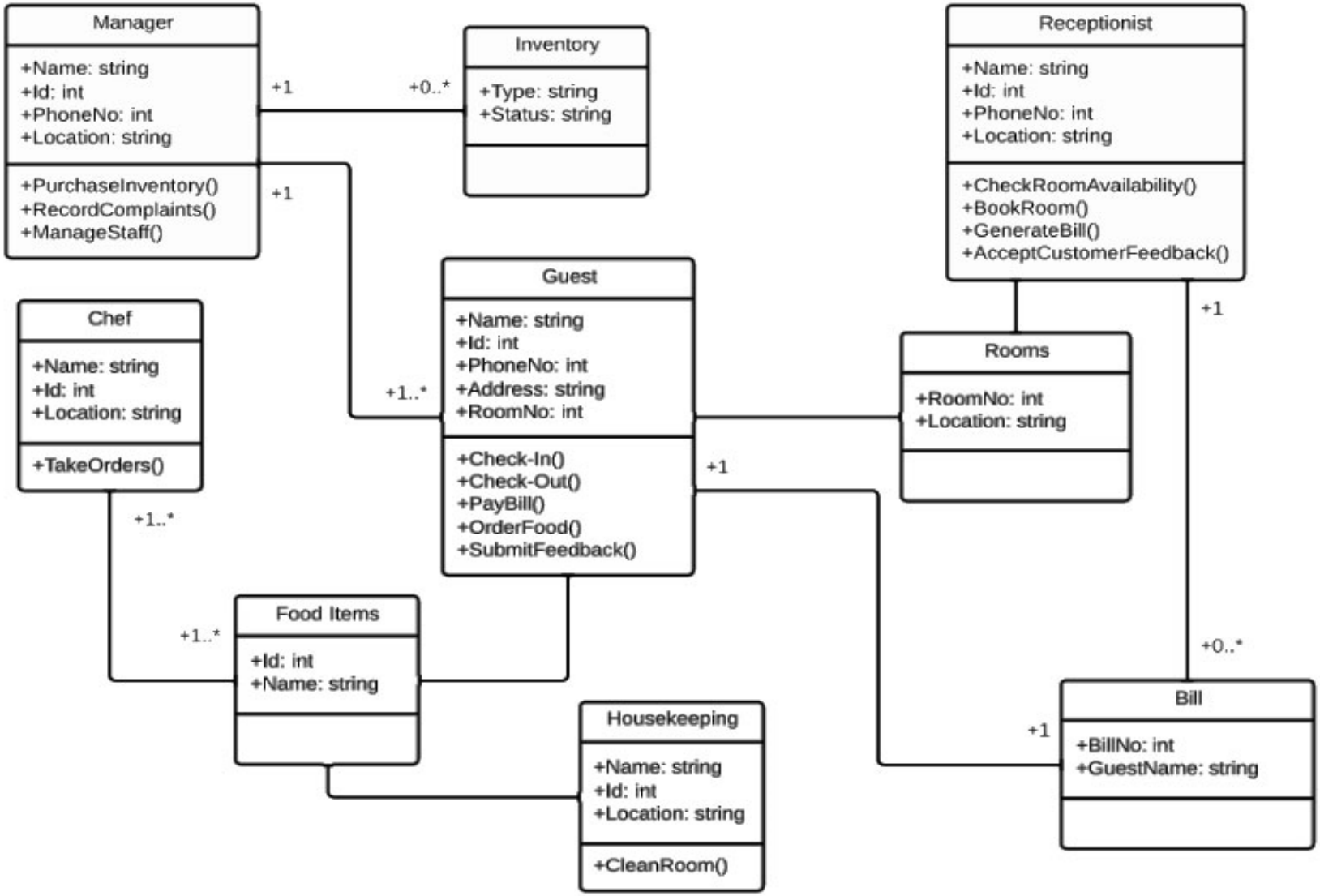
As an example, on your road trip through Arizona, you might run across a speed trap where a speed cam records your driving activity, but yo

Class diagram examples

Creating a class diagram to map out process flows is easy. Consider the two examples below as you build your own class diagrams in UML.

Class diagram for a hotel management system

A class diagram can show the relationships between each object in a hotel management system, including guest information, staff responsibi



Class diagrams details

Visibility

Now, let’s talk about visibility. UML allows us control who can access the attributes and the methods of our classes.
We have the following visibility levels in UML:

We have the following visibility levels in UML:
"+" means public visibility.
A class method or attribute marked as public can be used by code outside of the object.
"-" denotes private access.
Private attributes and methods can only be used within the class that defines them.
Elements marked as private can’t be accessed directly from other classes.
UML uses "#" to mark an element as protected.

+	**public**	**Can be also used by code outside the object**
-	**private**	**Can only be accessed in the defining class**
#	**protected**	**Can be accessed from the defining and child classes**
~	**package**	**Available within the enclosing package**

Protected visibility means that only child classes and the defining class will be able to access that attribute or method. "~" denotes package v
and provide a namespace for this group. Using package visibility, we make our elements available within its enclosing package.

UML provides these visibility tags, but its up to us to adapt it to the language we’re using.

Now, **there’s one rule that’s common for all object-oriented languages**:

You should only expose as much as needed!
And hide everything else.

Class attributes will usually have private or protected access.
We should provide public **setters** and **getters** instead of allowing everybody to access our class's data. This lets us control what the callers do

In our Trip class, we could make the "name" attribute public. Callers could set and retrieve it, which seems to work as expected. But, what if

Another example. The trip's start date needs to be earlier than its end date. Yet, callers can freely set any start or end date. Lets to set the visi
Now, we can access them exclusively from within the class's own methods, and other objects can't set or retrieve these attributes.

They are, well, "private" to the Trip class.

But then, how do we set, retrieve or modify them?

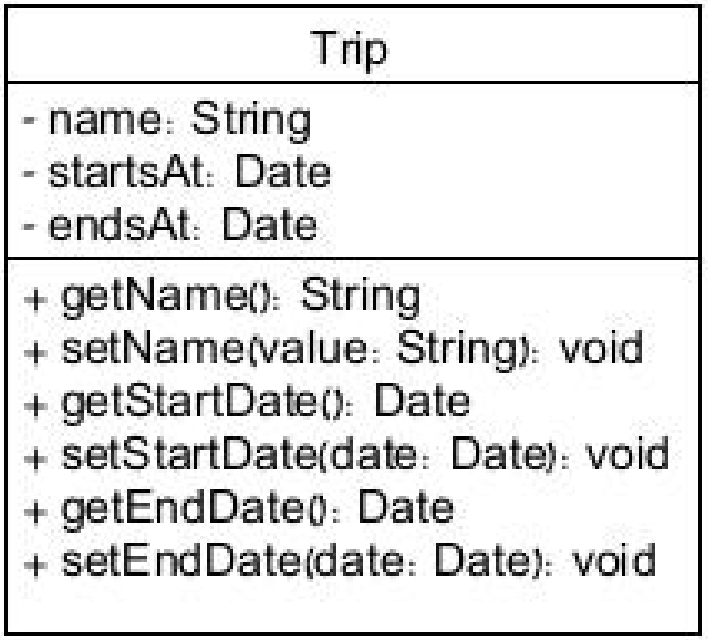
Here's the solution: providing public **getters** and **setters**
for each of these attributes:

getName(), setName(), getStartDate(), setStartDate()
getEndDate(), setEndDate()

Now we can check whether the name is at least three characters long by validating the "name" parameter in the setName() method. If it's sho

Also, we validate the start and the end date in the corresponding setters.
We are now in full control of our class's internal data.
Setters let us check the input argument, and **getters** allow us to modify the value before returning it.
For example, we could return a date in the user's timezone.
So far, we've seen how to represent a single class.
Class diagrams let us also show the relationships between the classes in our system.

```java
public class Trip {

    private String name;
    private Date startsAt;
    private Date endsAt;



    public String getName() {
        return this.name;
    }

    public void setName(String value) throws Exception {
        if(value.length() < 3)
            throw new Exception("Name too short!");

        this.name = value;
    }

    public Date getStartDate() {
        return this.startsAt;
    }

    public void setStartDate(Date valueDate) throws Exception {
        if(valueDate.getValue() > this.endsAt.getValue())
            throw new Exception("The start date cant be before the end date!");

        this.startsAt = valueDate;
    }


}
```



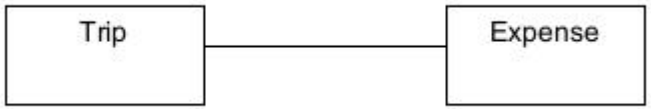
Association relationship

The next logical step after identifying the key classes in our system is figuring out the relationships between them.
Use-cases or user stories will help us during this process. Here's one of the functional requirements of the TravelExpense app. We have a **Tri**

The Trip class needs to know about its expenses. But should the Expense class also know about the Trip class? We've already talked about th

Tight coupling is something that you should definitely try to avoid.

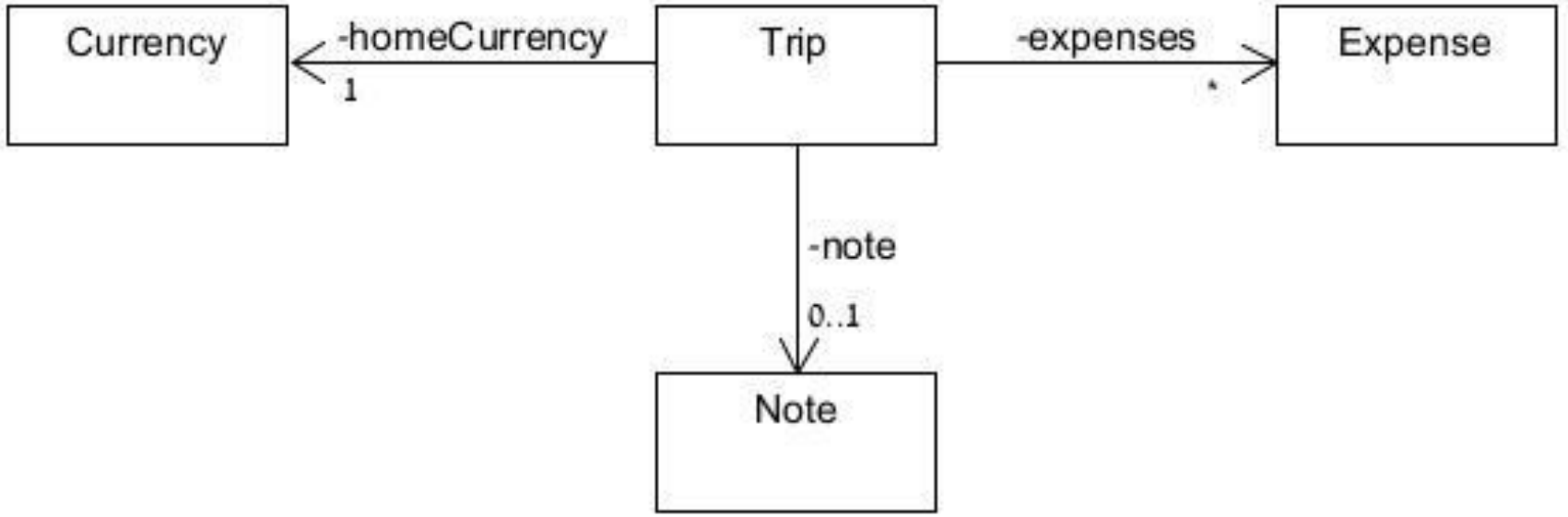
Lets illustrate the issue it causes. The Trip refers to the Expense class. That's fine, since a trip can have expenses associated with it. What hap
if we tried to use the Expense class in other parts of the system, we'd need to also bring the Trip class with it.
This doesn't make sense, as we should be able to use an Expense without a Trip. UML lets us express directed associations.
By drawing a solid line that ends with an open arrowhead, we show that only one of the classes refers to the other one.
The arrow points to the class that's referred to by the other class. In our current example, the association is bi-directional.



Let's change it to a directed association.

Now it shows that the Expense is associated with the Trip, but the Expense class doesn't know anything about the Trip class.
A Trip will usually have multiple expenses.
We can represent the multiplicity of associated objects as follows:

*** - a Trip can have zero or more Expenses**
1 - A Trip must have exactly one homeCurrency
0..1 - A Trip may or may not have a single note

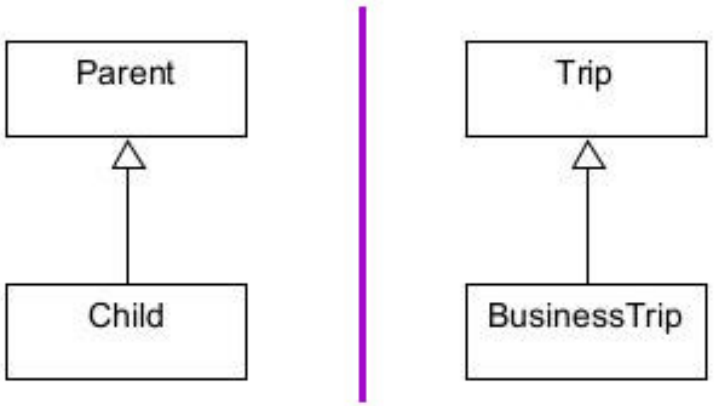


Associations can show multiplicities at both ends of the line.
The default multiplicity is one.
So, if there's no multiplicity shown,
you can safely assume it's one.
We can also display the name of the
class property for the given association.

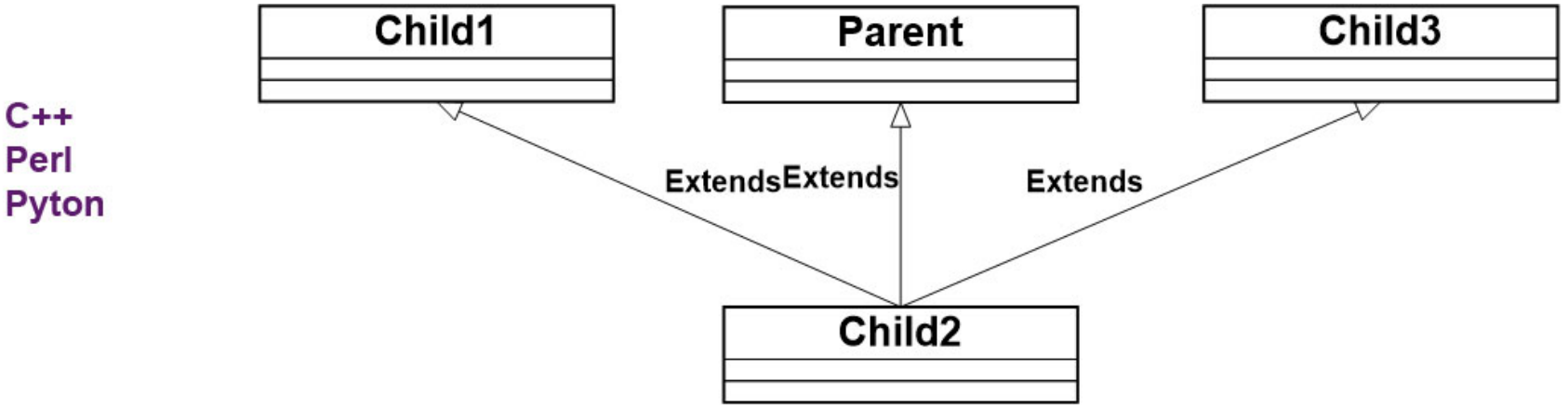
Generalization relationship

In UML, we use generalization to express
that one model element is based on another model element.
Generalization is represented as a solid line with a hollow arrowhead that points to the parent.
Let's say that we need a special Trip class for our business trips.
BusinessTrip would inherit from the Trip class and this is how we represent it in UML.

Because BusinessTrip inherits everything from its parent,
we must only specify the attributes and operations that are specific to the child.

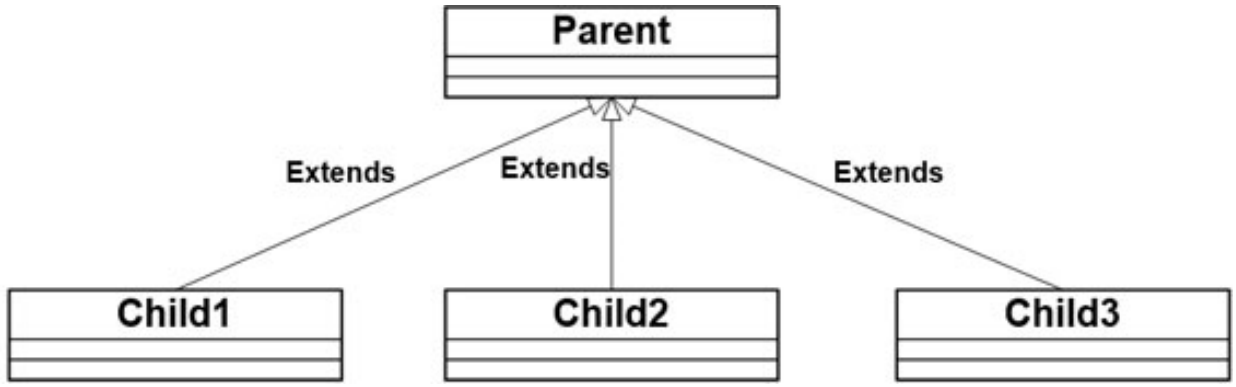


A parent can have multiple children.
And we can also have child classes that inherit from different parents.
Some programming languages support multiple inheritance - C++, Perl, Python just to name a few.



Many modern programming languages only allow single inheritance, that is, inheriting from one parent class.
Single inheritance reduces the complexity and avoids the ambiguity that comes with multiple inheritance.
Some argue that multiple inheritance has more benefits than drawbacks. However, it's certainly easier to make mistakes when using multiple
UML doesn't restrict generalization to classes.
It can also be used in use-case or component diagrams.
This lets us indicate that a child element receives it's parent's attributes,
operations and relationships.

C#
Java



Dependency relationship

In UML, a dependency relationship is a relationship in which one element, the client, uses or depends on another element, the supplier. You
You can also use a dependency relationship to represent precedence, where one model element must precede another.

Typically, dependency relationships do not have names.

As the following figure illustrates, a dependency is displayed in the diagram editor as a dashed line with an open arrow that points from the

Types of dependency relationships

Because a dependency relationship can represent several different types of relationships, keywords or stereotypes show the precise nature of

Type of dependency	Keyword or Stereotype	Description
Abstraction	«abstraction», «derive», «refine», or «trace»	Relates two model elements, or sets of model elements, that represent the same concept at different levels of abstraction, or from different viewpoints
Binding	<<bind>>	Connects template arguments to template parameters to create model elements from templates
Realization	<<realize>>	Indicates that the client model element is an implementation of the supplier model element, and the supplier model element is the specification
Substitution	<<substitute>>	Indicates that the client model element takes the place of the supplier; the client model element must conform to the contract or interface that the supplier model element establishes
Usage	«use», «call», «create», «instantiate», or «send»	Indicates that one model element requires another model element for its full implementation or operation

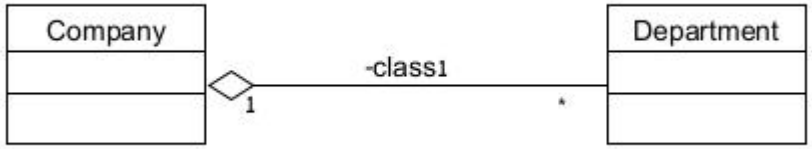
Aggregation relationship

In UML models, an aggregation relationship shows a classifier as a part of or subordinate to another classifier.
An aggregation is a special type of association in which objects are assembled or configured together to create a more complex object. An ag

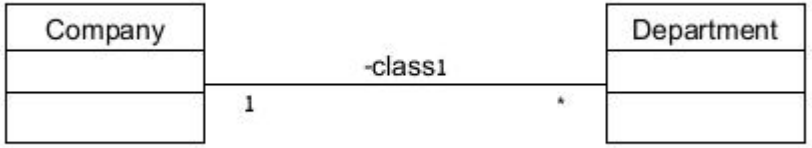
Data flows from the whole classifier, or aggregate, to the part. A part classifier can belong to more than one aggregate classifier and it can ex

You can name an association to describe the nature of the relationship between two classifiers; however, names are unnecessary if you use as

As the following figure illustrates, an aggregation association appears as a solid line with an unfilled diamond at the association end, which i



Aggregation is considered redundant, because it expresses the same thing as association.
So, this two diagrams are equivalent



Composition relationship

Composition is a stronger form of association.
It shows that the parts live and die with the whole. In other words, composition implies ownership:
when the owning object is destroyed, the contained objects will be destroyed, too.
The composition is represented as a filled diamond on the owner's end
connected with a solid line with the contained class. The Expenses of a trip can't exist without the trip. If we delete the Trip, its expenses are

Realization relationship

In UML modeling, the realization is a relationship between two objects, where the client (one model element) implements the responsibility

The realization relationship does not have names. It is mostly found in the interfaces. It is represented by a dashed line with a hollow arrowh

Interface Realization

Interface realization is a kind of specialized relation between the classifier and the interface. In interface realization relationship, realizing cl

A classifier implementing an interface identifies the objects that conform to the interface and any of its ancestors. A classifier can execute on

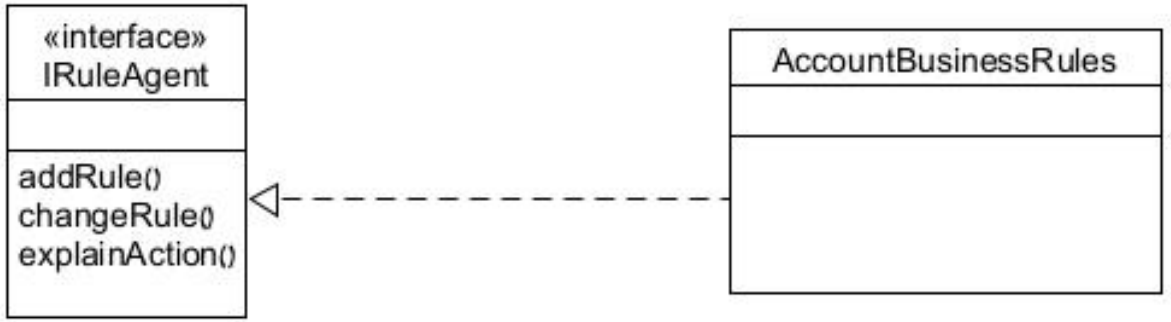
The interface realization relationship does not contain names, and if you name it, then the name will appear beside the connector in the diagr

The interface realization relationship is represented by a dashed line with a hollow arrowhead, which points from the classifier to the given i

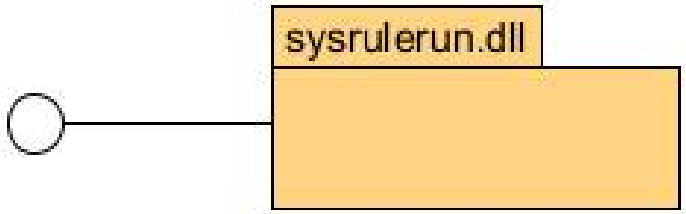
Types of Realization

1. Canonical form: In UML, the canonical form realizes the interfaces across the system. An interface stereotype is used for creating an inte

From the diagram given below, it can be seen that the object Account Business Rules realizes the interface Iruleagent.



2. Elided form: It is that kind of realization relationship in which the interface is represented by a circle, also known as a lollipop notation. V



Using dependency relationships

You can add dependency relationships to models to accomplish the following goals:

- Connect two packages to indicate that at least one element in the client package is dependent on an element in the supplier package. The dep
- Connect two classes to indicate that the connection between them is at a higher level of abstraction than an association relationship. The dep
  - Temporarily uses a supplier class that has global scope
  - Temporarily uses a supplier class as a parameter for one of its operations
  - Temporarily uses a supplier class as a local variable for one of its operations
  - Sends a message to a supplier class
- Connect components to interfaces or other components to indicate that they use one or more of the operations that the interface specifies or

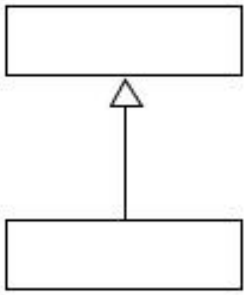
Example

In an e-commerce application, a Cart class depends on a Product class because the Cart class uses the Product class as a parameter for an ac

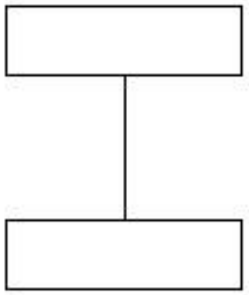
The above relationship indicates that a change to the Product class might require a change to the Cart class.

UML relationships quick summary

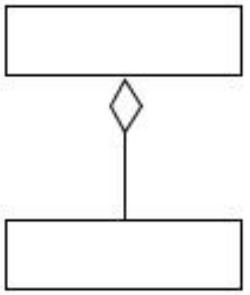
Generalization / "is-a"



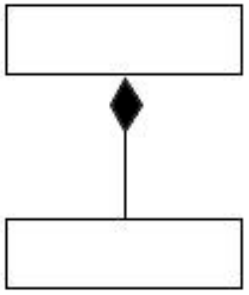
Association / "has-a"



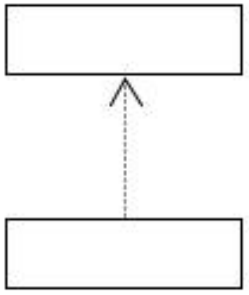
Aggregation / "has-a"



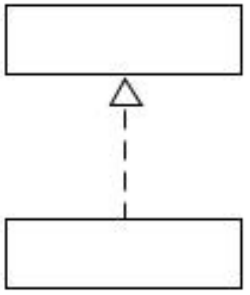
Composition / "part-of"



Dependency / "references"



Realization / "implements behavior"



[UMLet - Free UML Tool for Fast UML Diagrams]

Sequence Diagram

Sequence diagrams are a popular dynamic modeling solution in UML because they specifically focus on the processes and objects that live s

What is a sequence diagram in UML?

A sequence diagram is a type of interaction diagram because it describes how—and in what order—a group of objects works together. These

Note that there are two types of sequence diagrams: UML diagrams and code-based diagrams. The latter is sourced from programming code.

Benefits of sequence diagrams

Sequence diagrams can be useful references for businesses and other organizations. Try drawing a sequence diagram to:

- Represent the details of a UML use case.
- Model the logic of a sophisticated procedure, function, or operation.
- See how objects and components interact with each other to complete a process.
- Plan and understand the detailed functionality of an existing or future scenario.

Use cases for sequence diagrams

The following scenarios are ideal for using a sequence diagram:

- **Usage scenario:** A usage scenario is a diagram of how your system could potentially be used. It's a great way to make sure that you have wo
- **Method logic:** Just as you might use a UML sequence diagram to explore the logic of a use case, you can use it to explore the logic of any f
- **Service logic:** If you consider a service to be a high-level method used by different clients, a sequence diagram is an ideal way to map that o

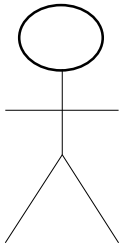
Basic symbols and components

Object symbol

Represents a class or object in UML. The object symbol demonstrates how an o

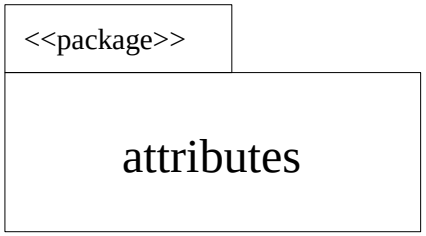
Activation box

Represents the time needed for an object to complete a task. The longer the task



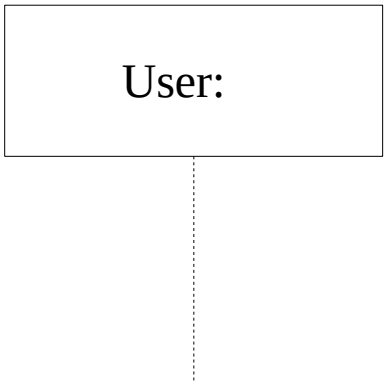
Actor symbol

Shows entities that interact with or are external to the system.



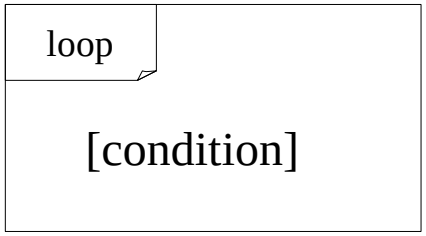
Package symbol

Used in UML 2.0 notation to contain interactive elements of the diagram. Also k



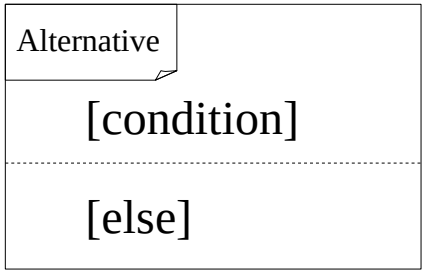
Lifeline symbol

Represents the passage of time as it extends downward. This dashed vertical line



Option loop symbol

Used to model if/then scenarios, i.e., a circumstance that will only occur under c

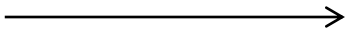


Alternative symbol

Symbolizes a choice (that is usually mutually exclusive) between two or more m

Common message symbols

Use the following arrows and message symbols to show how information is transmitted between objects. These symbols may reflect the sta

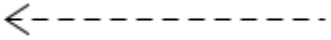


Synchronous message symbol

Represented by a solid line with a solid arrowhead. This symbol is used when a

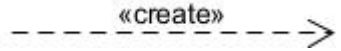
Asynchronous message symbol

Represented by a solid line with a solid arrowhead. This symbol is used when a



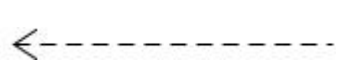
Asynchronous return message symbol

Represented by a dashed line with a lined arrowhead.



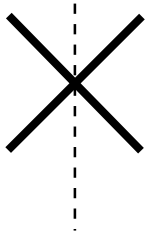
Asynchronous create message symbol

Represented by a dashed line with a lined arrowhead. This message creates a ne



Reply message symbol

Represented by a dashed line with a lined arrowhead, these messages are replies

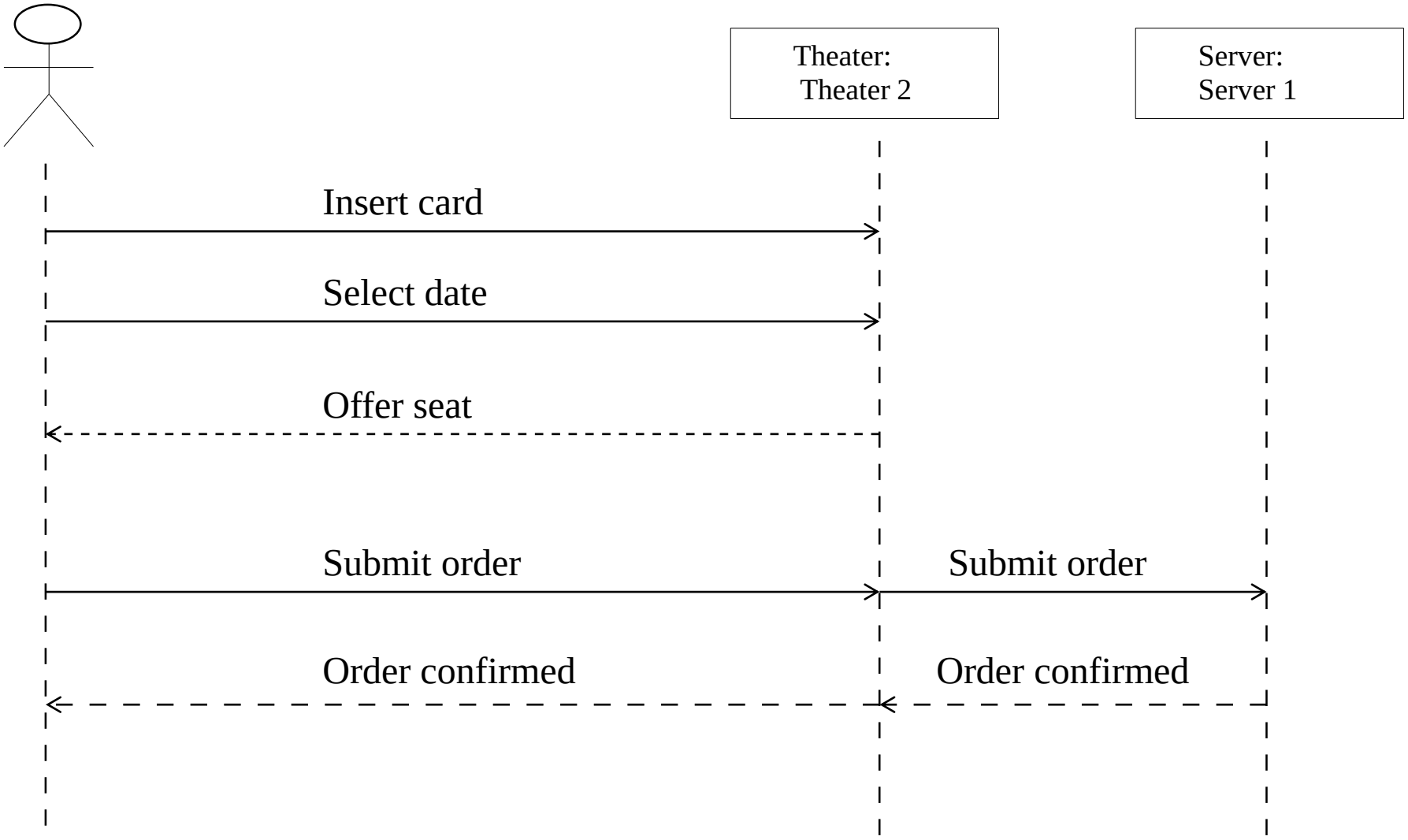


Delete message symbol

Represented by a solid line with a solid arrowhead, followed by an X. This mess

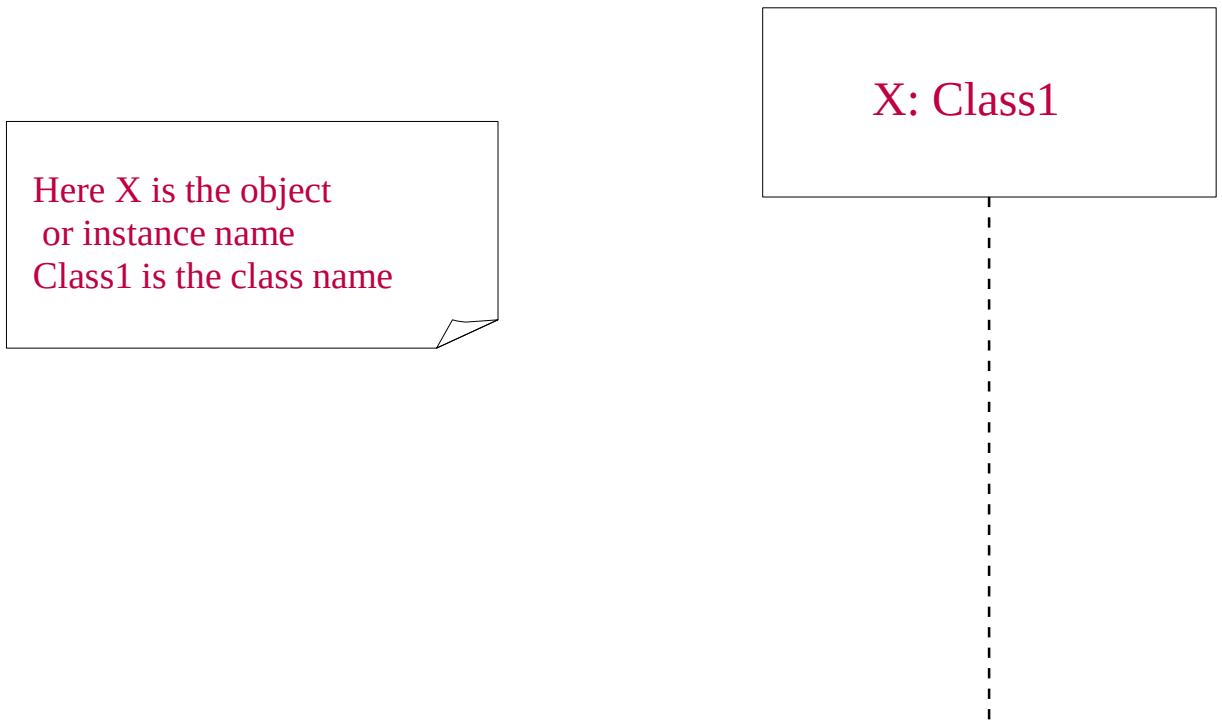
Explainations about the symbols:

Actors – An actor in a UML diagram represents a type of role where it interacts with the system and its objects. It is important to note here t
We use actors to depict various roles including human users and other external subjects. We represent an actor in a UML diagram using a stic
For example – Here the user in seat reservation system is shown as an actor where it exists outside the system and is not a part of the system.

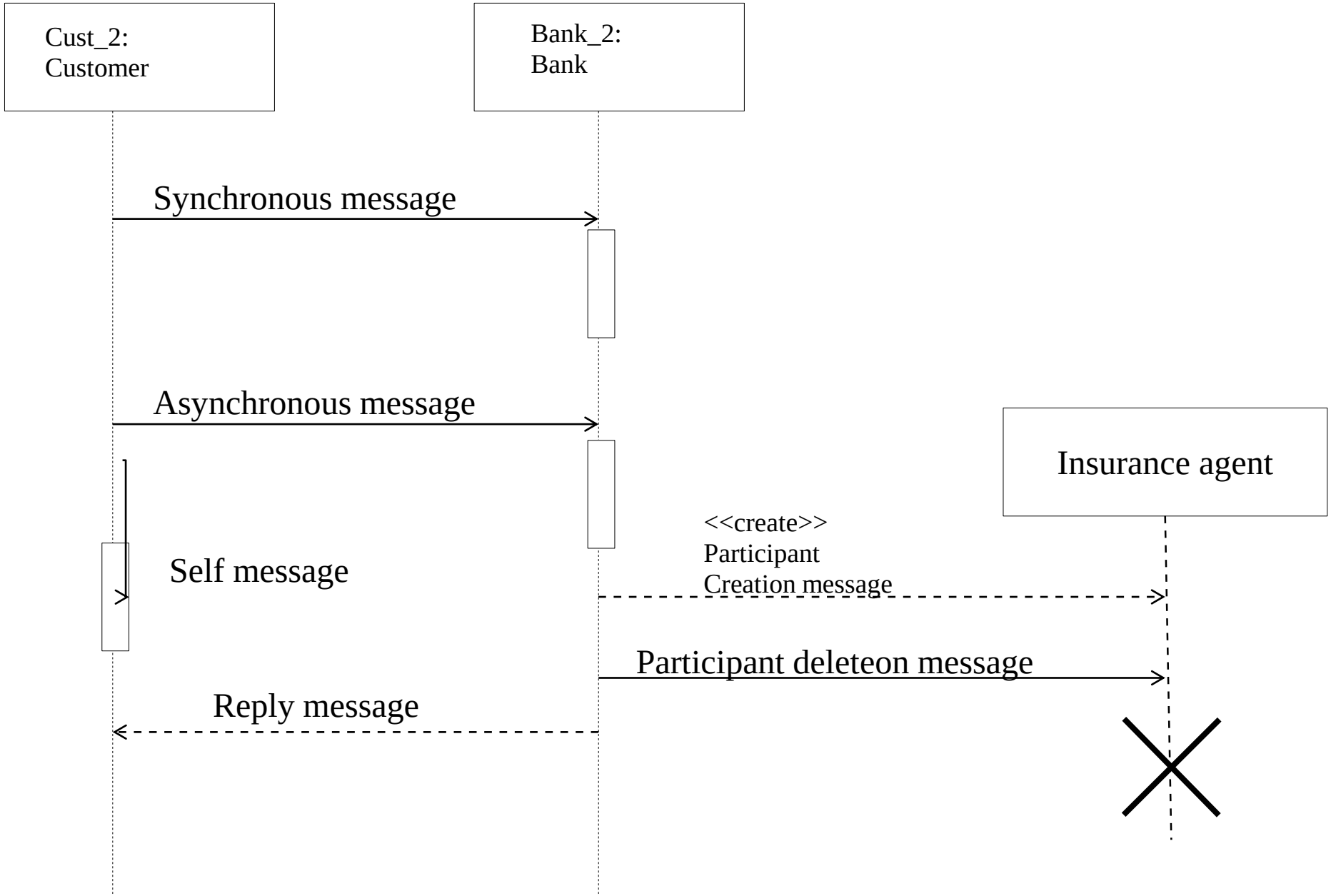


Lifelines – A lifeline is a named element which depicts an individual participant in a sequence diagram. So basically each instance in a seque

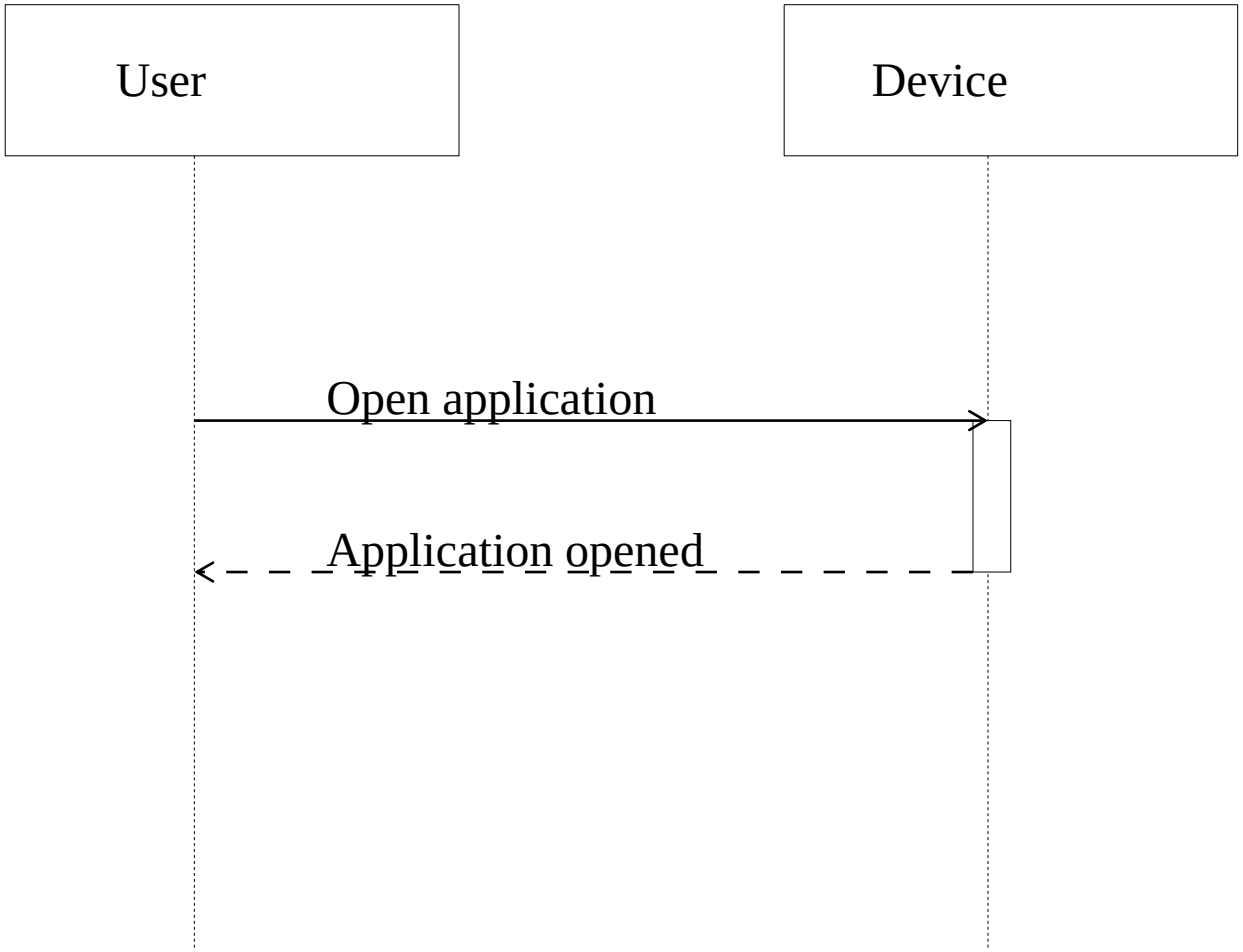
We display a lifeline in a rectangle called head with its name and type. The head is located on top of a vertical dashed line (referred to as the



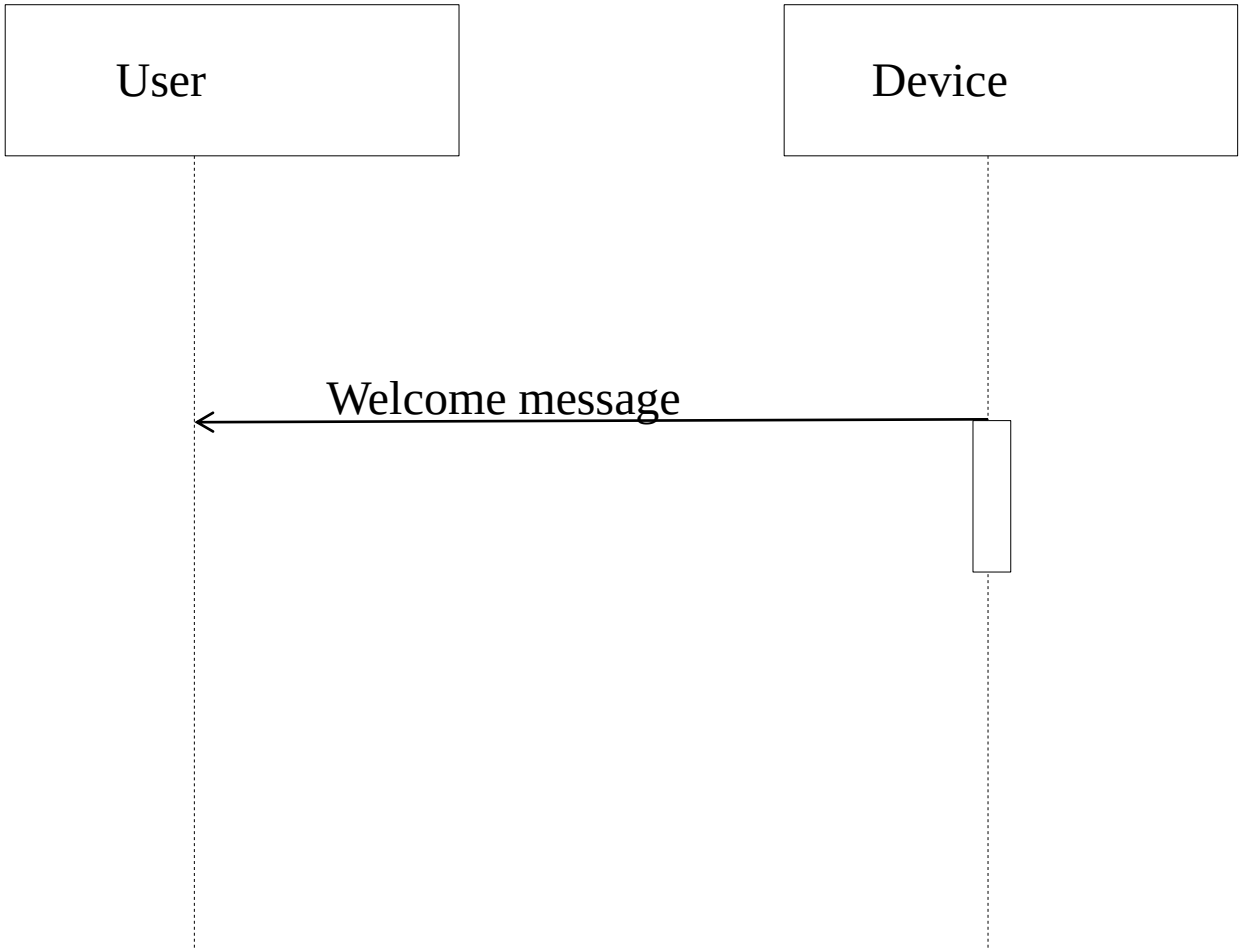
Messages – Communication between objects is depicted using messages. The messages appear in a sequential order on the lifeline. We repre
Messages can be broadly classified into the following categories :



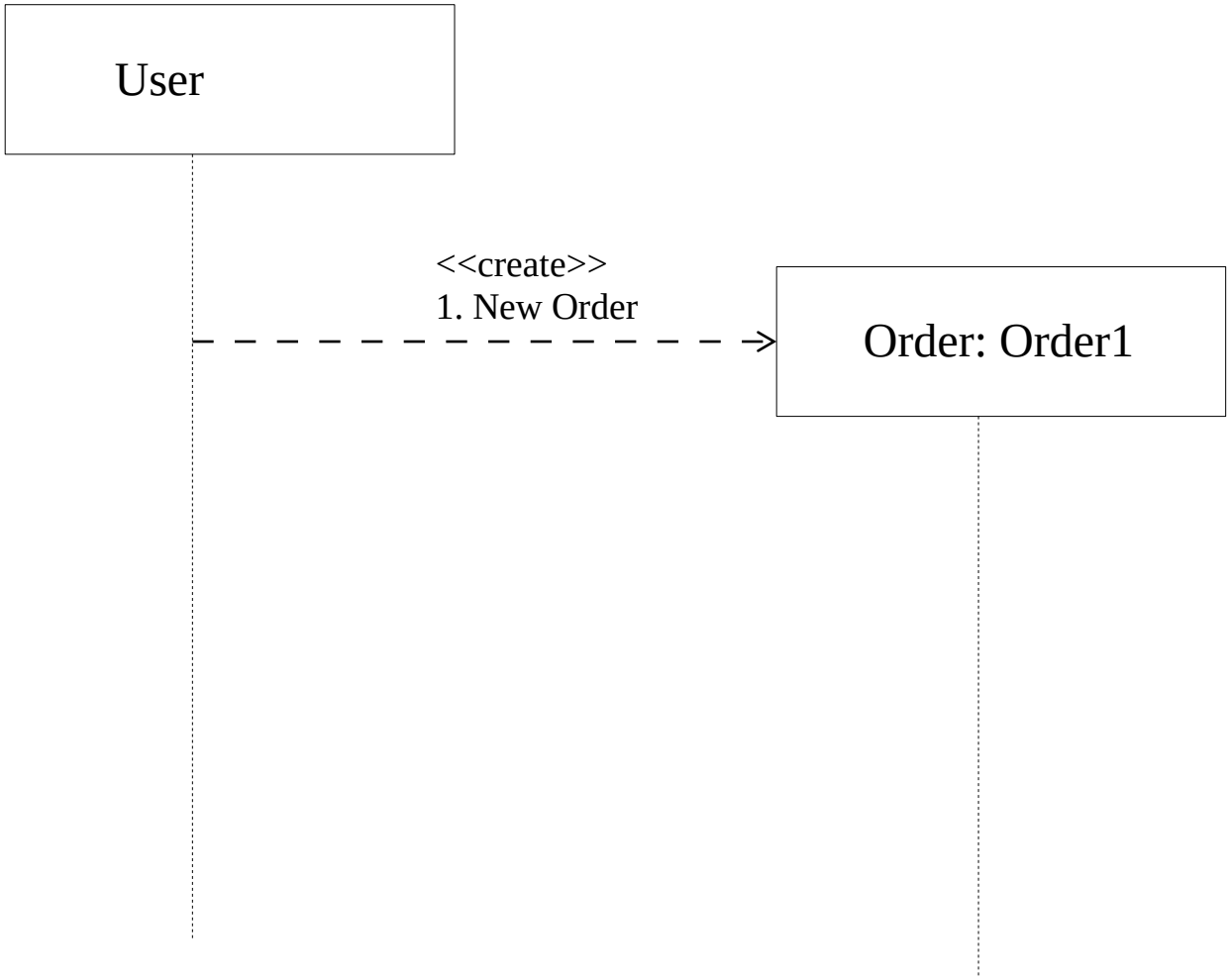
Synchronous messages – A synchronous message waits for a reply before the interaction can move forward. The sender waits until the recei



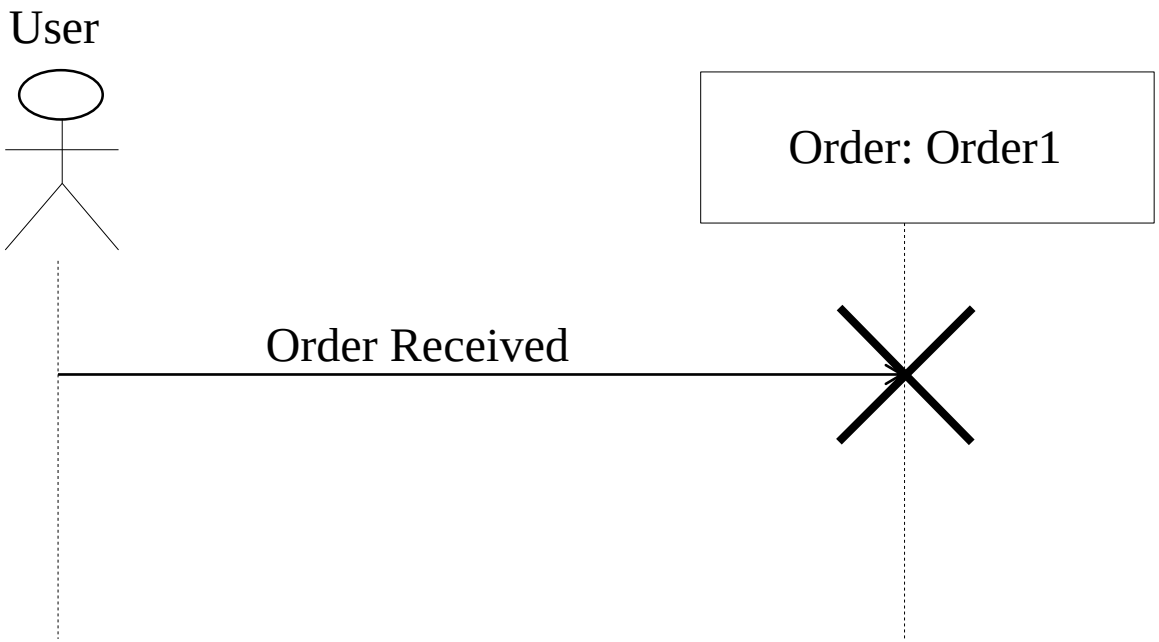
Asynchronous Messages – An asynchronous message does not wait for a reply from the receiver. The interaction moves forward irrespectiv



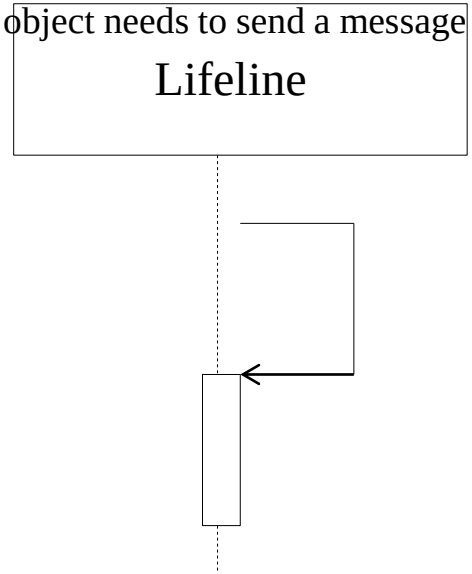
Create message – We use a Create message to instantiate a new object in the sequence diagram. There are situations when a particular messa
For example – The creation of a new order on a e-commerce website would require a new object of Order class to be created.



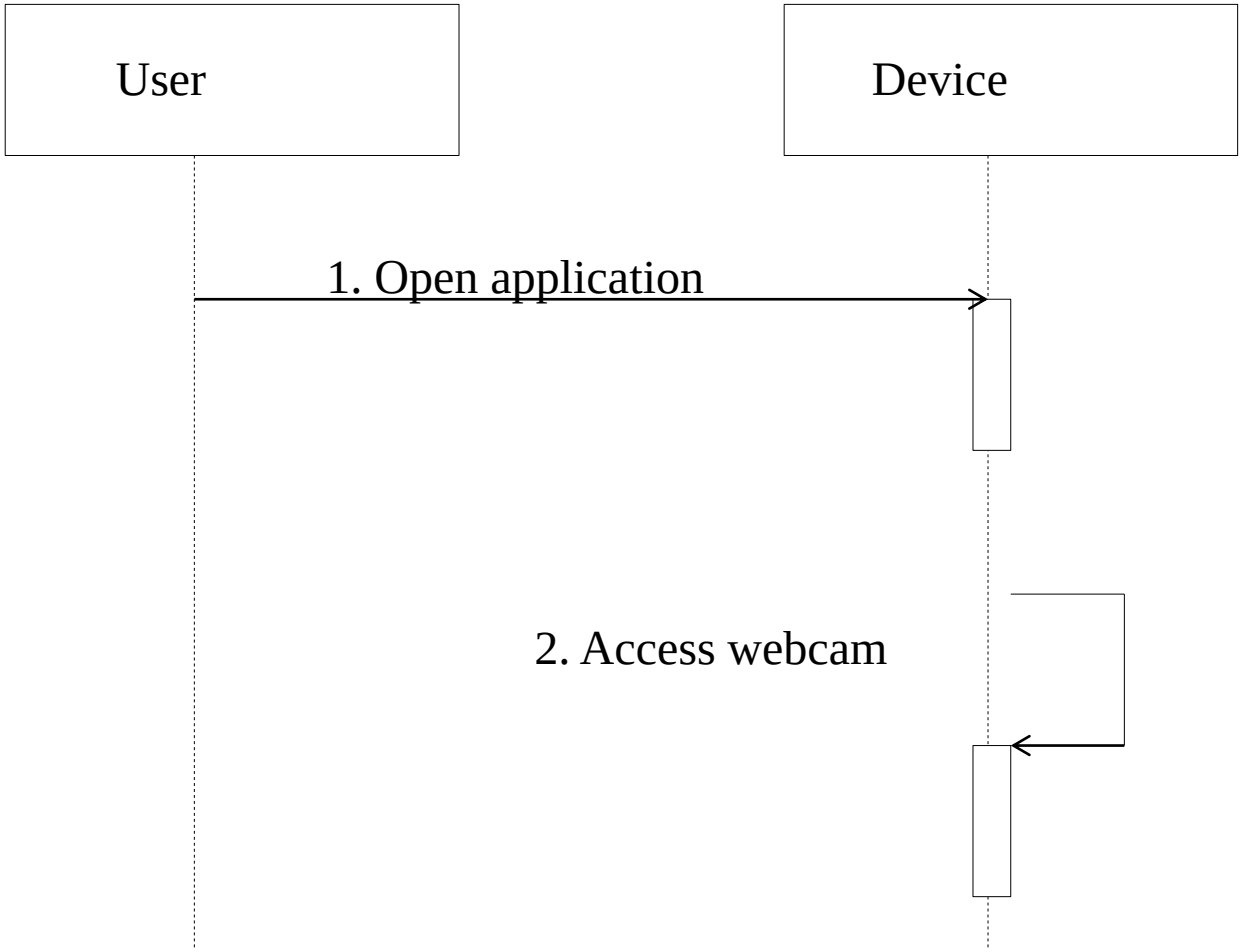
Delete Message – We use a Delete Message to delete an object. When an object is deallocated memory or is destroyed within the system we
For example – In the scenario below when the order is received by the user, the object of order class can be destroyed.



Self Message – Certain scenarios might arise where the object needs to send a message to itself. Such messages are called Self Messages and

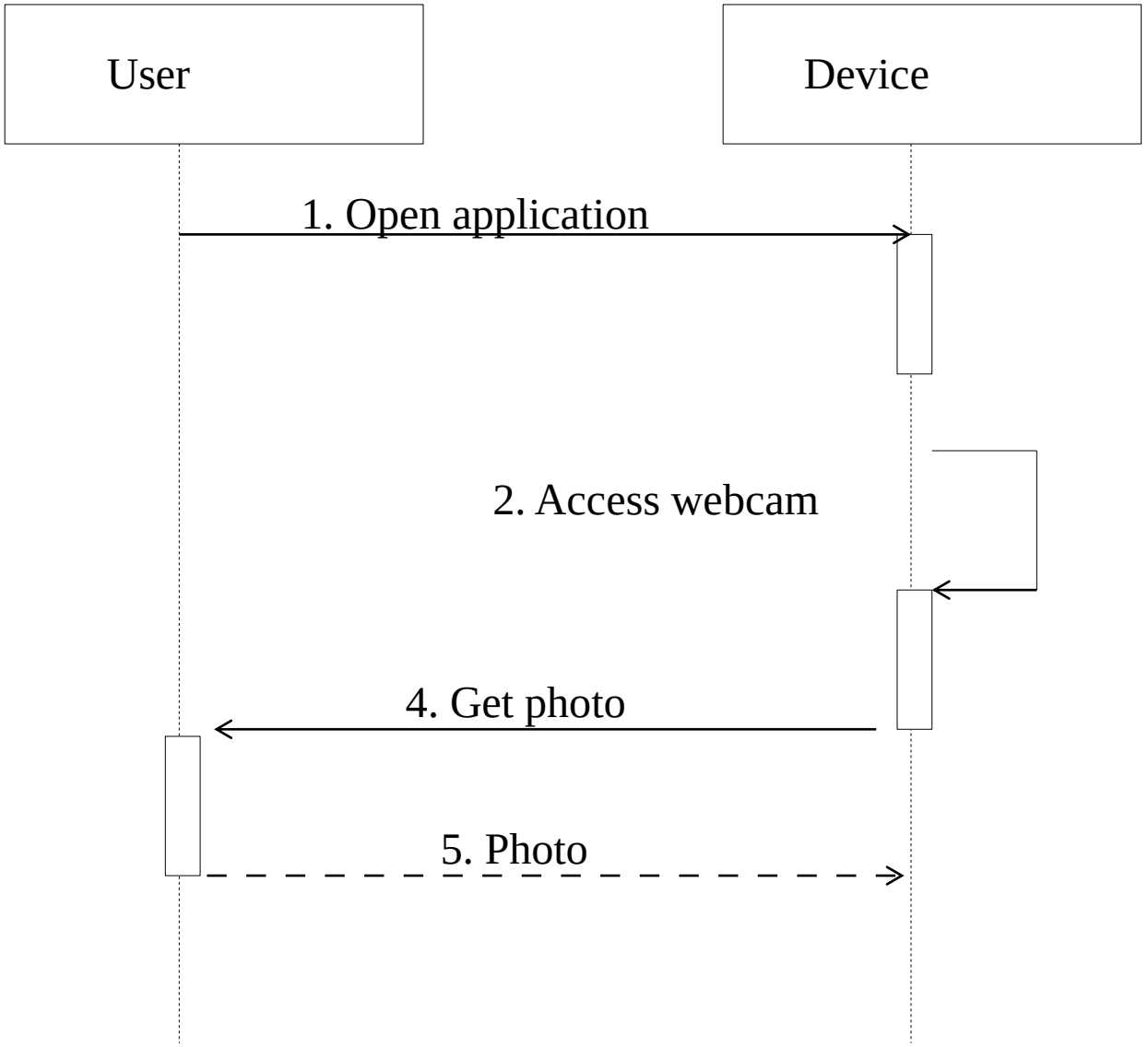


For example – Consider a scenario where the device wants to access its webcam. Such a scenario is represented using a self message.

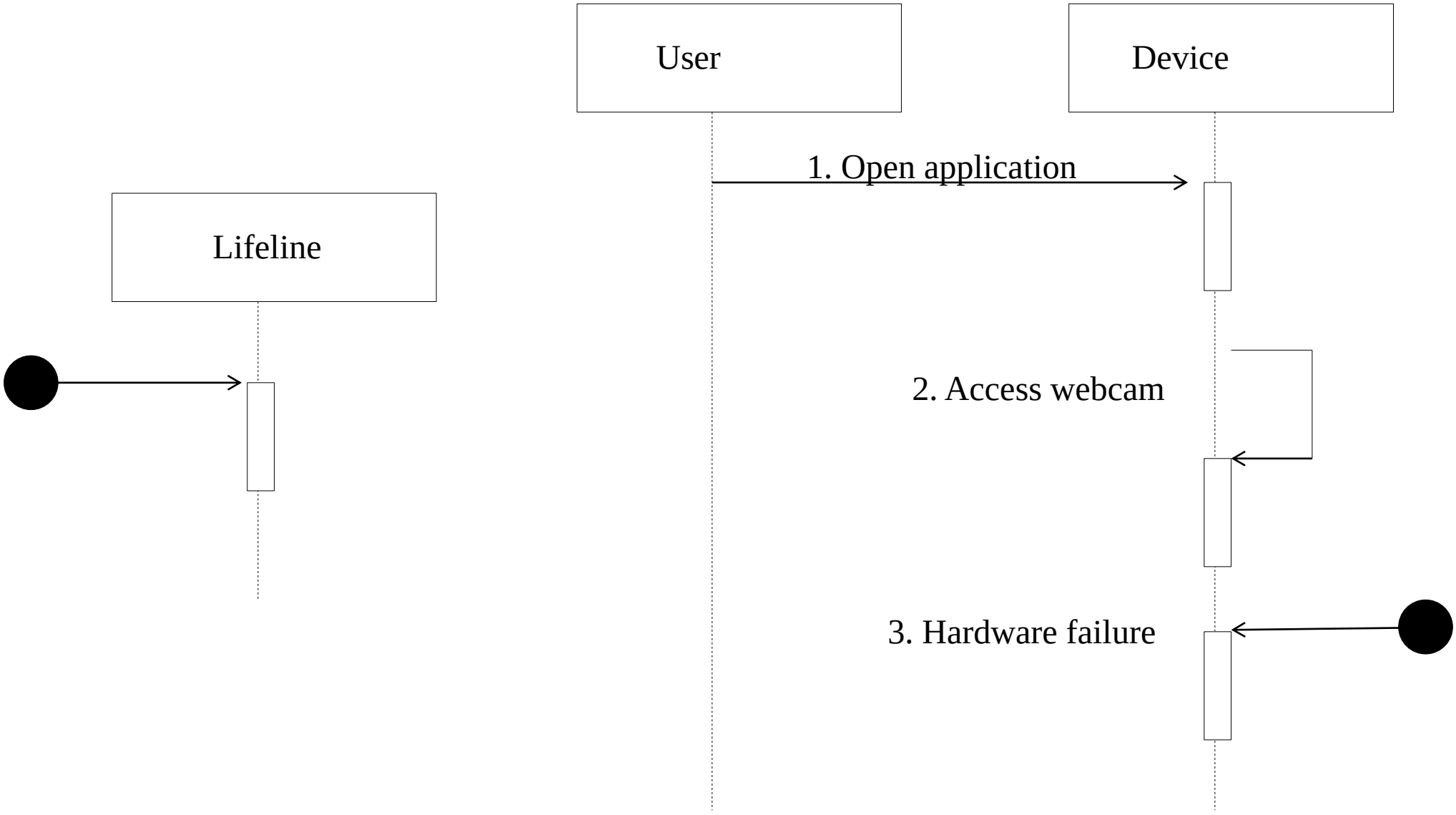


Reply Message – Reply messages are used to show the message being sent from the receiver to the sender. We represent a return/reply mess

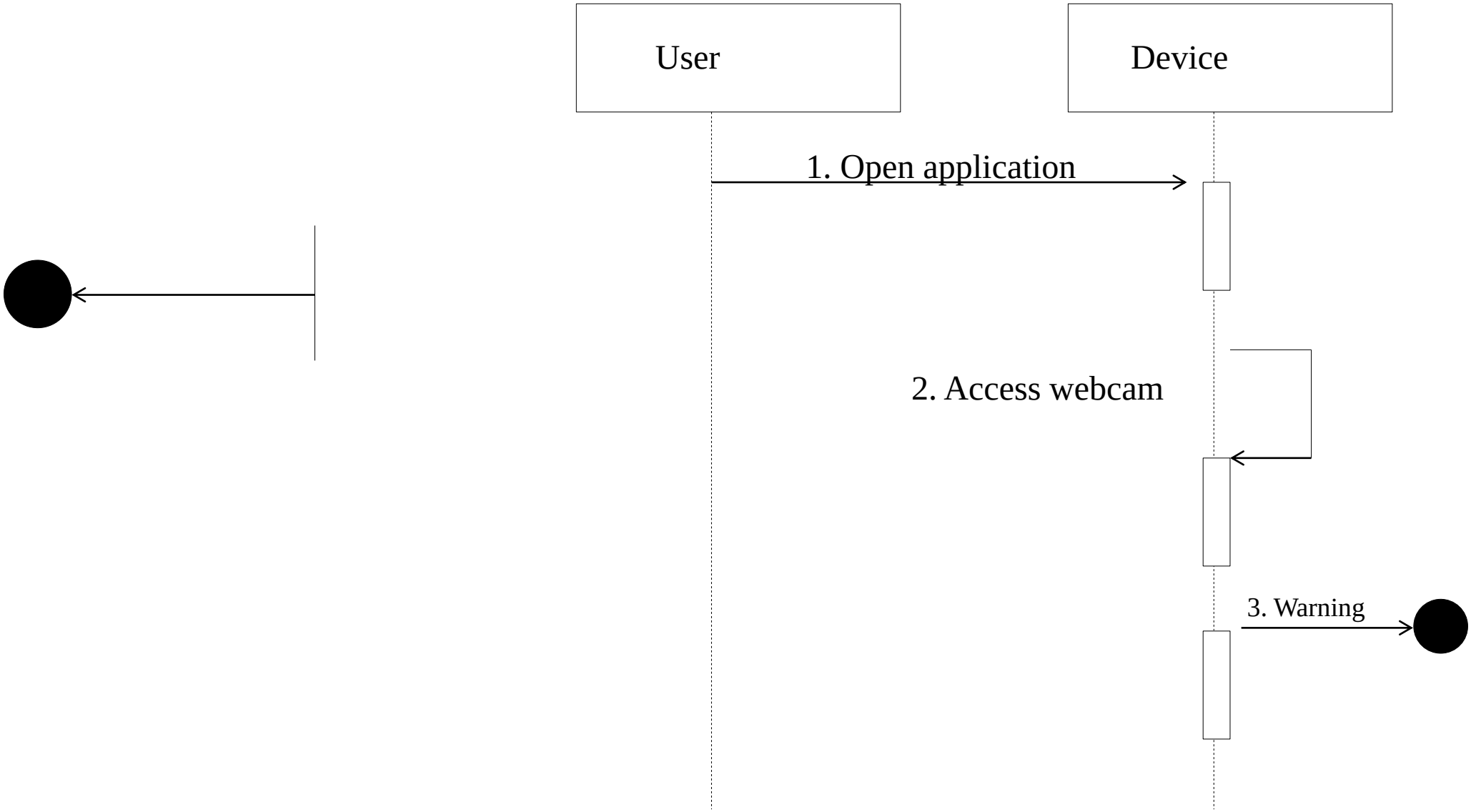
For example – Consider the scenario where the device requests a photo from the user. Here the message which shows the photo being sent is



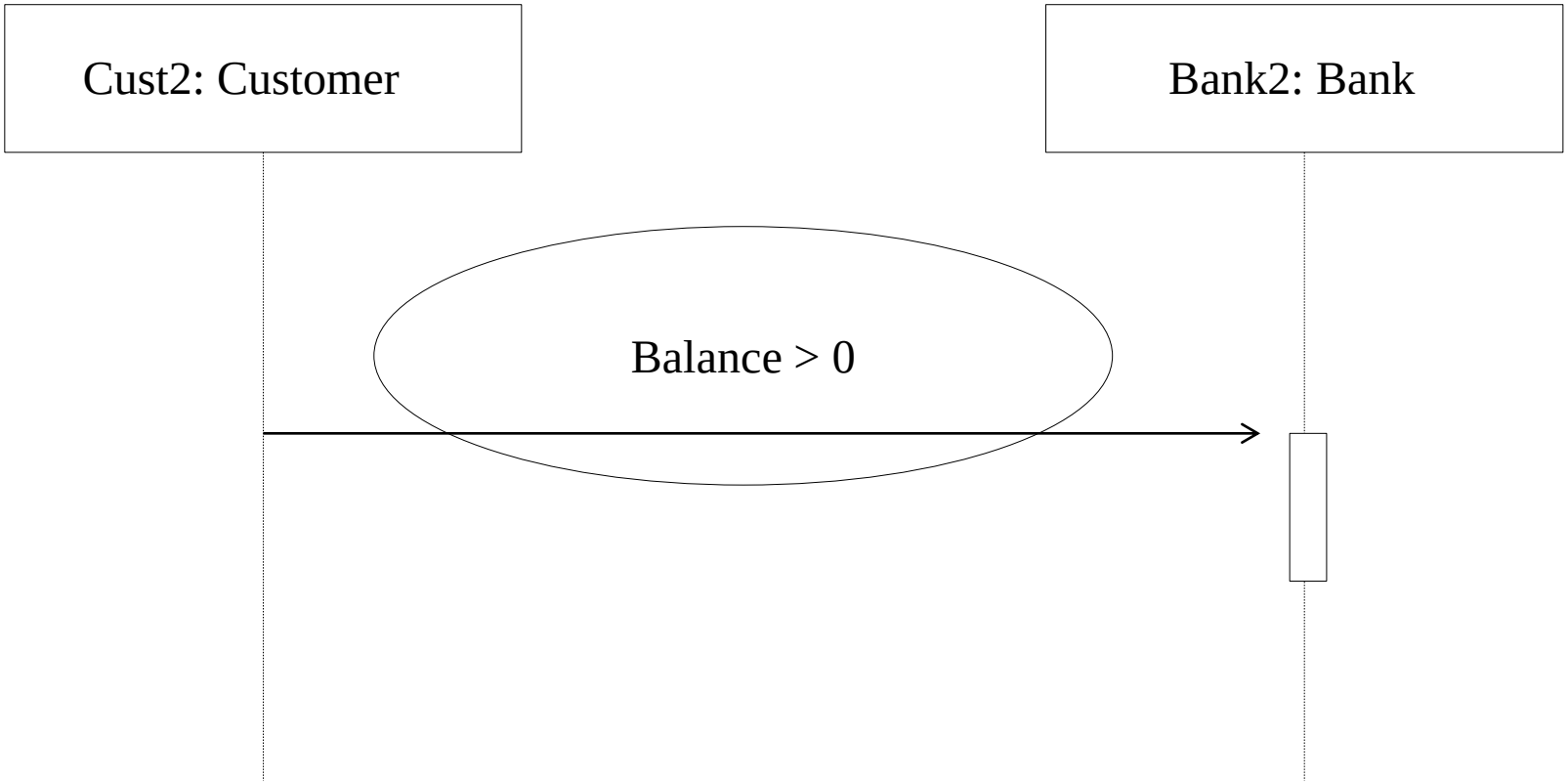
Found Message – A Found message is used to represent a scenario where an unknown sender sends the message. It is represented using a



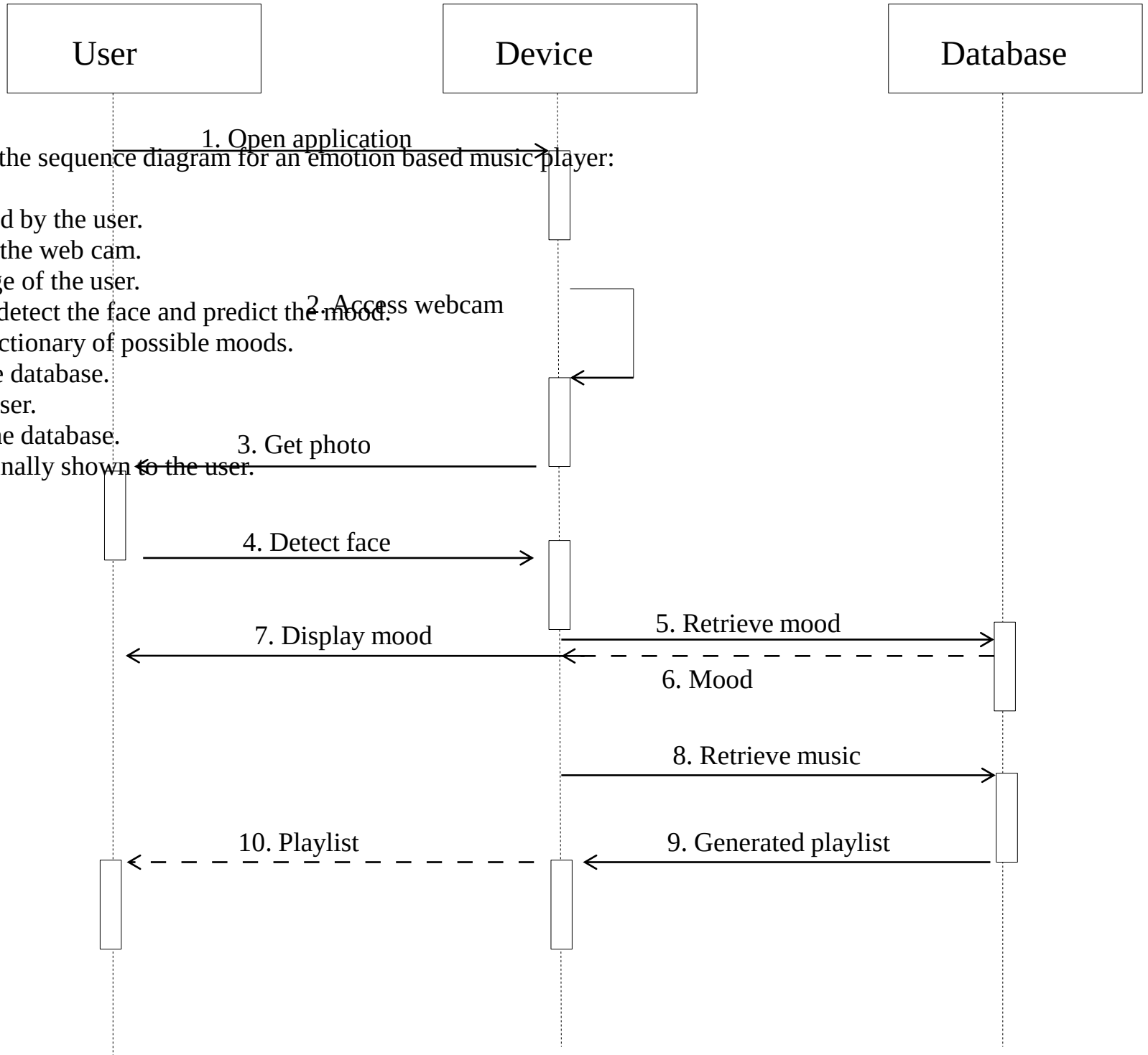
Lost Message – A Lost message is used to represent a scenario where the sender is known but the receiver is not known to the system. It is represented using an arrow



Guards – To model conditions we use guards in UML. They are used when we need to restrict the flow of messages on the pretext of a cond
For example: In order to be able to withdraw cash, having a balance greater than zero is a condition that must be met as shown below.



A sequence diagram for an emotion based music player –



This sequence diagram depicts the sequence diagram for an emotion based music player:

- Firstly the application is opened by the user.
- The device then gets access to the web cam.
- The webcam captures the image of the user.
- The device uses algorithms to detect the face and predict the mood.
- It then requests database for dictionary of possible moods.
- The mood is retrieved from the database.
- The mood is displayed to the user.
- The music is requested from the database.
- The playlist is generated and finally shown to the user.

Uses of sequence diagrams –

- Used to model and visualise the logic behind a sophisticated function, operation or procedure.
- They are also used to show details of UML use case diagrams.
- Used to understand the detailed functionality of current or future systems.
- Visualise how messages and tasks move between objects or components in a system.

https://sequencediagram.org/ - online sequence qiagram tool

Statechart Diagrams

What is a State Diagram?

State Diagram are used to capture the behavior of a software system. UML State machine diagrams can be used to model the behavior of a cl

Statechart diagrams provide us an efficient way to model the interactions or communication that occur within the external entities and a syste

Statechart diagrams are used to describe various states of an entity within the application system.

There are a total of two types of state machine diagram in UML:

Behavioral state machine

- It captures the behavior of an entity present in the system.
- It is used to represent the specific implementation of an element.
- The behavior of a system can be modelled using behavioral state machine diagram in OOAD.

Protocol state machine

- These diagrams are used to capture the behavior of a protocol.
- It represents how the state of protocol changes concerning the event. It also represents corresponding changes in the system.
- They do not represent the specific implementation of an element.

Why State Machine Diagram?

Statechart diagram is used to capture the dynamic aspect of a system. State machine diagrams are used to represent the behavior of an applic

Statechart diagrams are used to design interactive systems that respond to either internal or external event. Statechart diagram in UML visua

It represents the state of an object from the creation of an object until the object is destroyed or terminated.

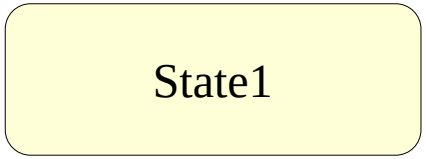
The primary purpose of a statechart diagram is to model interactive systems and define each and every state of an object. Statechart diagrams

Notation and Symbol for State Machine

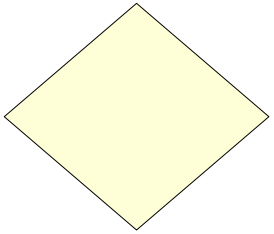
Following are the various notations that are used throughout the state chart diagram. All these notations, when combined, make up a single d

Initial state

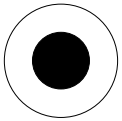
State-box



Decision-box



Final state



Notataion Legend:

Initial state

The initial state symbol is used to indicate the beginning of a state machine diagram.

Final state

This symbol is used to indicate the end of a state machine diagram.

Decision box

It contains a condition. Depending upon the result of an evaluated guard condition, a new path is taken for program execution.

Transition

A transition is a change in one state into another state which is occurred because of some event. A transition causes a change in the state of a

State box

It is a specific moment in the lifespan of an object. It is defined using some condition or a statement within the classifier body. It is used to re

It is denoted using a rectangle with round corners. The name of a state is written inside the rounded rectangle.

The name of a state can also be placed outside the rectangle. This can be done in case of composite or submachine states. One can either plac

A state can be either active or inactive. When a state is in the working mode, it is active, as soon as it stops executing and transits into anothe

Types of State

Unified Modeling Language defines three types of states:

Simple state
- They do not have any substrate.

Composite state
- These types of states can have one or more than one substrate.
- A composite state with two or more substates is called an orthogonal state.

Submachine state
- These states are semantically equal to the composite states.
- Unlike the composite state, we can reuse the submachine states.

How to draw a Statechart diagram?

Statechart diagrams are used to describe the various state that an object passes through. A transition between one state into another state occu

The purpose of these UML diagrams is to represent states of a system. States plays a vital role in state transition diagrams. All the essential o

Following rules must be considered while drawing a state chart diagram:

- The name of a state transition must be unique.
- The name of a state must be easily understandable and describe the behavior of a state.
- If there are multiple objects, then only essential objects should be implemented.
- Proper names for each transition and an event must be given.

When to use State Diagrams?

State diagrams are used to implement real-life working models and object-oriented systems in depth. These diagrams are used to compare th

Statechart diagrams are used to capture the changes in various entities of the system from start to end. They are used to analyze how an even

State char diagrams are used,

- To model objects of a system.
- To model and implement interactive systems.
- To display events that trigger changes within the states.

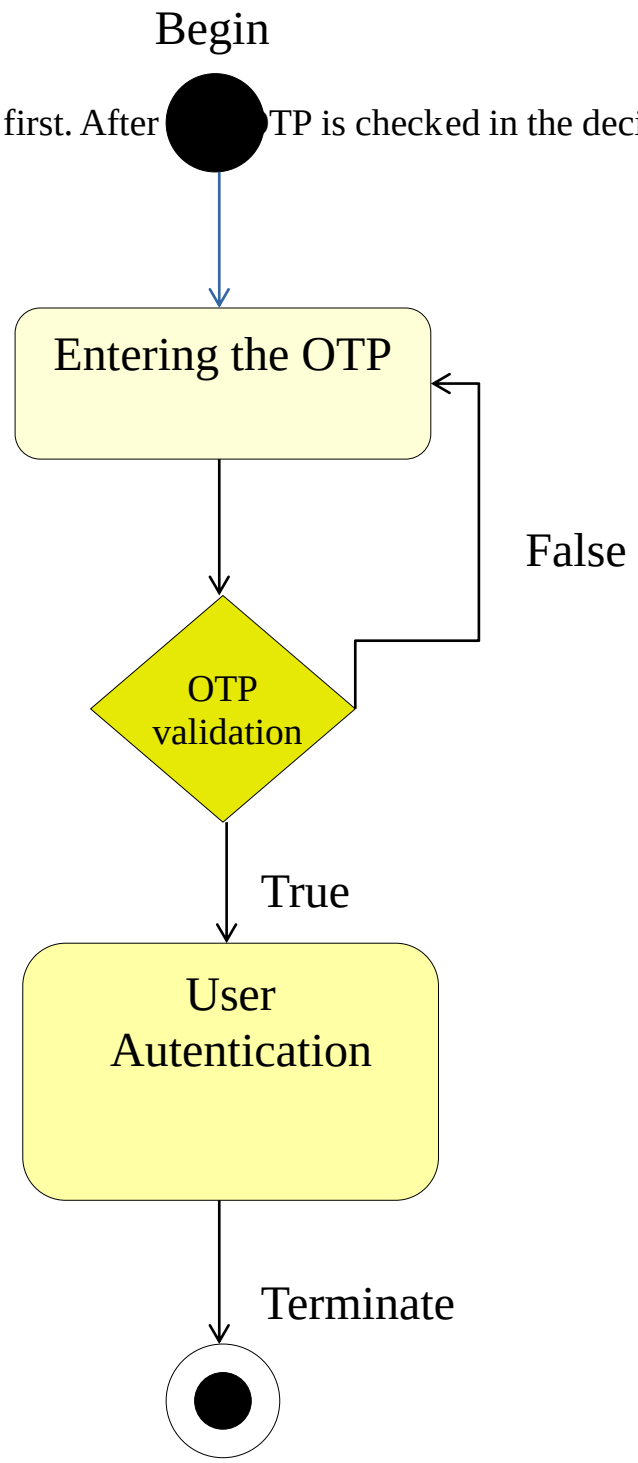
State machine vs. Flowchart

Statemachine	Flow chart
It represents various states of a system.	The Flowchart illustrates the program execution flow.
The state machine has a WAIT concept, i.e., wait for an action or an event.	The Flowchart does not deal with waiting for a concept.
State machines are used for a live running system.	Flowchart visualizes branching sequences of a system.
The state machine is a modeling diagram.	A flowchart is a sequence flow or a DFD diagram.
The state machine can explore various states of a system.	Flowchart deal with paths and control flow.

Example of State Machine

Following state diagram example chart represents the user authentication process.

There are a total of two states, and the first state indicates that the OTP has to be entered first. After TP is checked in the decision box,



Summary

- Statechart diagrams are also called as state machine diagrams.
- These diagrams are used to model the event-based system.
- A state of an entity is controlled with the help of an event.
- There is a total of two types of state machine diagrams: 1) Behavioral 2) State machine 3) Protocol state machine
- Statechart diagram is used to capture the dynamic aspect of a system.
- A state is a specific moment in the lifespan of an object.