

2019

# Introduction to Modeling



Peter Stikker

Inholland

1/26/2019

## Acknowledgement and Preface

Some parts of this reader were copied from the tutorial guides of Visual Paradigm. I'm grateful that they allowed me to do this for educational purposes.

The structure of the reader, and some of the examples are also based on the PowerPoint material from the lecturers. These were supplied by Mr. Thijs Otter, so also a huge thanks to him.

I've tried to use proper APA citations and captions, but will probably have overlooked a few issues.

This is the first edition of this reader, so there are bound to be mistakes or improvements that could be made. Please let me know.

Enjoy reading and learning.

Cheers,

Peter Stikker

## Table of Contents

Acknowledgement and Preface .....	2
1 Introduction.....	5
1.1 What is modeling.....	7
1.2 Modeling Software .....	8
2 Process diagram .....	9
3 Unified Modeling Language (UML).....	11
3.1 The Origin of UML .....	11
3.2 History of UML.....	12
3.3 Why UML.....	13
3.4 UML - An Overview.....	13
4 Use Case Diagrams .....	15
4.1 Use Case Diagram Basic Components .....	16
4.2 Use Case Diagram Relationships .....	19
4.3 Use Case Diagram - Examples .....	22
4.4 Creating a UseCase Diagram .....	23
4.4.1 How to Identify Actor .....	23
4.4.2 How to Identify Use Cases.....	23
4.4.3 Use Case Levels of Details .....	24
4.5 Use Case Description .....	25
4.6 Use Case Diagram Tips .....	26
5 Activity Diagrams.....	27
5.1 Quick history.....	27
5.2 When to Use Activity Diagram .....	28
5.3 Activity Diagram Components.....	28
5.4 Decision and Merge nodes .....	30
5.5 Fork and Join nodes.....	31
6 Class Diagram .....	33
6.1 The Class Node .....	34
6.2 Association and Multiplicity .....	35
6.3 Class Navigation.....	36
6.4 Aggregation and Composition.....	37
6.5 Generalization in Class Diagrams .....	39

7	Sequence Diagram.....	41
7.1	Sequence diagram components.....	42
7.2	Loops in sequence diagrams .....	42
7.3	Sequence diagrams and Class diagrams.....	43
8	State Machine Diagram .....	44
8.1	Why State Machine Diagrams? .....	45
8.2	Basic Components of a State Machine Diagram .....	45
8.3	Guards and Actions .....	47
8.4	State Machine Diagram and Class Diagrams .....	47
9	Final comments .....	48
	References.....	49
	Appendix 1. Dictionaries for discussed diagrams.....	51
	A1.1. UseCase Diagram .....	51
	A1.2. Activity Diagram .....	51
	A1.3. Class Diagram .....	52
	A1.4. Sequence diagram.....	52
	A1.5. State Machine diagram .....	53
	Index.....	54

# 1 Introduction

In Programming 1 you have started with learning how to create a small program. You probably started immediately with opening up Visual Studio and trying out some code. This might be good enough for a small program, but for larger systems it pays off to do some planning. As Eisenhower<sup>1</sup> said in 1957: “plans are worthless, but planning is everything” (The American Presidency Project, n.d.). This shows that although many things don’t go as planned, without the planning it probably would have gone even worse.

To plan and create larger systems many different methods are used: Agile, Rational Unified Process (RUP), Essential Unified Process (EssUP), Rappid Application Development (RAD), etc. The most basic version, and still often used, is probably the Adjusted Waterfall Model (Royce, 1970), as shown in Figure 1.

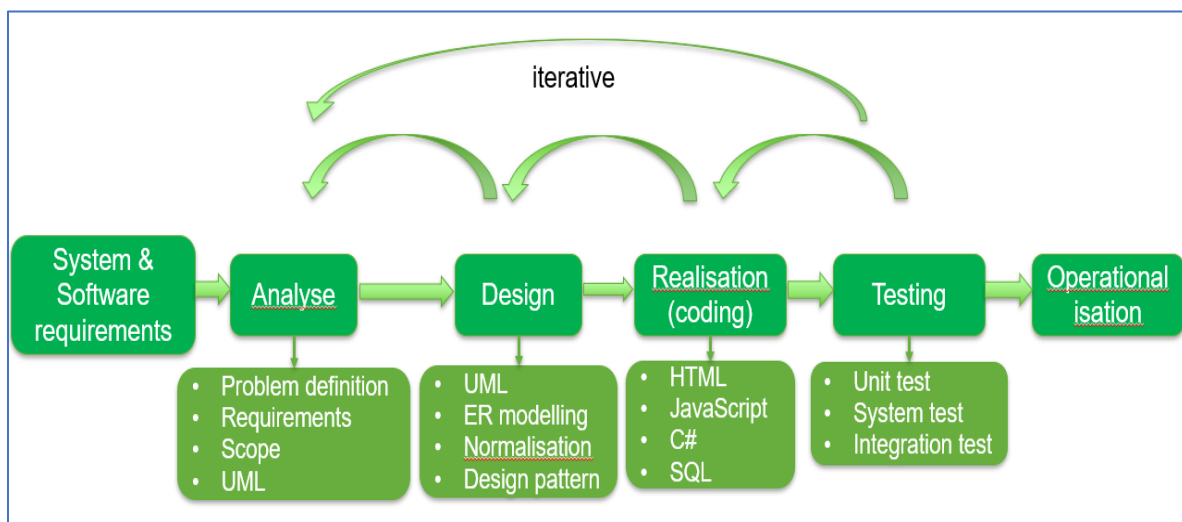


Figure 1. The adjusted waterfall model.

The steps might seem obvious. We first determine the requirements, then analyse the problem (what needs to be solved by the system), then design, then start coding, test the result, and if all works well launch it. This course focuses on the beginning of this process. In a strict Waterfall process each phase has to be completed in full, however most often it should be seen as an iterative process, where sometimes results in one phase might require to go back to a previous phase.


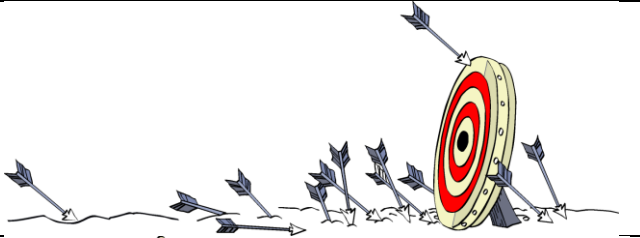

If we don’t know what the problem is, we might end up with the wrong solution. This is often underestimated. A client might at first not even know what his/her real problem is. The next step is to analyse the situation. If we dive right into ‘trial-and-error’ we might end up wasting a lot of time, and perhaps never really find a solution. If we don’t design, we might even have an ‘overkill’ solution.

The importance of the first three phases has been nicely illustrated by McConnell (2004) and shown in Table 1.

<sup>1</sup> Dwight D. Eisenhower was the Supreme Allied Commander in Europe during the 2<sup>nd</sup> World War, and made the planning for operation Overlord (which included the D-day landings). After the war he would also become president of the United States of America.

Table 1

Importance of Requirements, Analyse, and Design

The penalty for failing to define the problem is that you can waste a lot of time solving the wrong problem. This is a double-barrelled penalty because you also don't solve the right problem	
Without good requirements, you can have the right general problem but miss the mark on specific aspects of the problem.	
Without good software architecture, you may have the right problem but the wrong solution. It may be impossible to have successful construction.	

Note. Reprinted from Code Complete (2nd ed.), by S. McConnell, 2004, pp. 38, 39, 44.

McConnell (2004) also looked at some research from 1976 to 2004, and summarized the results of the costs of fixing defects in the different phases. He summarized the results as shown in Table 2.

Table 2

Cost of fixing defects

	Time detected				
Time Introduced	Requirements	Architecture	Construction	System Test	Post-Release
Requirements	1	3	5-10	10	10-100
Architecture		1	10	15	25-100
Construction			1	10	10-25

Note. Reprinted from Code Complete (2nd ed.), by S. McConnell, 2004, p. 29.

The costs are times \$1 000. For example if a defect in the architecture is found during the architecture phase, this would cost on average \$1 000, however if the mistake is not found until the system test, it would cost \$15 000.

In conclusion, it is important to first think about the system and design it, than diving straight into programming.

## 1.1 What is modeling

Systems **modeling** can be defined as: “the interdisciplinary study of the use of models to conceptualise and construct systems in engineering, business and IT development” (Mo, Bil, & Sinha, 2015, p. 54). Modeling therefor deals with **models**, which in turn can be defined as: “an abstraction of a system, aimed at understanding, communicating, explaining, or designing aspects of interest of that system” (Dori, 2002, p. 272). In computer science an **abstraction** can be defined as “the process of removing physical, spatial, or temporal details or attributes in the study of objects or systems in order to focus attention on details of higher importance” (Wikipedia, n.d.)<sup>2</sup>. Abstraction is searching for the simplest representation, without loss of essential information.

Note that a child drawing of a house (for example Figure 2), could be considered a model, but we would see this as an informal model.



Figure 2. Child drawing of a house, an informal model. (common creative licenced)

A formal model of a house might look as shown in Figure 3.

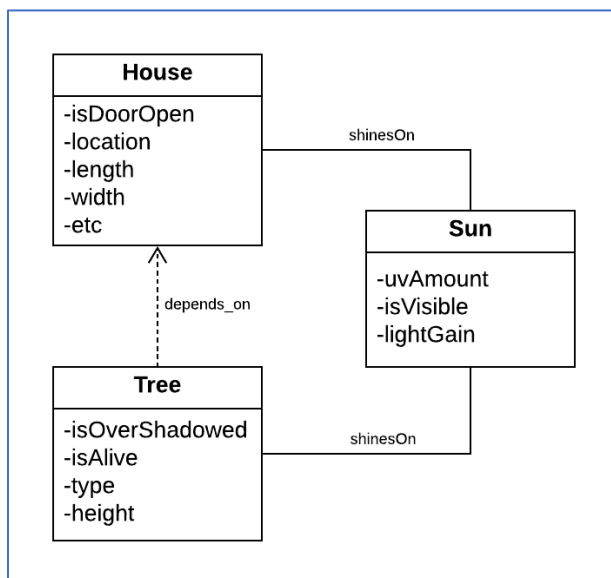


Figure 3. Example of a formal model.

A **formal model** is based on some convention set in the field.

<sup>2</sup> Yes, Wikipedia but this is based on Colburn and Gary (2007), and Kramer (2007). The definition from Wikipedia seems very well fitting for this course.

## 1.2 Modeling Software

To create UML diagrams there are a lot of different software programs available. Although you could simply use MS Paint, PowerPoint or Word to draw the diagrams, it is recommended to use a program that is more designed to draw models. For this course a simple program to draw diagrams would be sufficient. Personally, I use **LucidChart** for this. It is a web-based program, and students can get a free educational version to have an unlimited number of objects. I also often hear about StarUML, but have not tried this myself.

More sophisticated programs also exist. Visual Paradigm has a community edition for free, and Eclipse with Papyrus is also free. Both of these would go beyond the capabilities of LucidChart. In software like this you can manage a project with multiple diagrams linked to each other. This is not required for this course and has a steeper learning curve, but if you want to prepare yourself for what the 'big boys' are using, you might want to give it a go.

Other large modeling software include Enterprise Architect (you can get a trial version of Enterprise Architect, but that will only last for 30 days) or Rational Software Designer (from IBM, based on Enterprise Architect).

MS Visio is also possible, but not free. This is more on the level of LucidChart although it has more diagram options.

Again, there are many software tools out there, and for this course any basic one that can draw basic UML shapes should be fine. A nice comparison of some can be found at:

<http://socialcompare.com/en/comparison/uml-tools>.



## 2 Process diagram

To start our journey into modeling we begin with a model that is very useful at the start of the project: a process diagram. Although all other diagrams we will discuss in this reader are part of something known as UML, this is the only exception.

A **process diagram** is an overview of the core functionalities the system should be able to produce. It consists out of so-called business processes, perhaps split out into elementary business processes. The generic structure is shown in Figure 4.

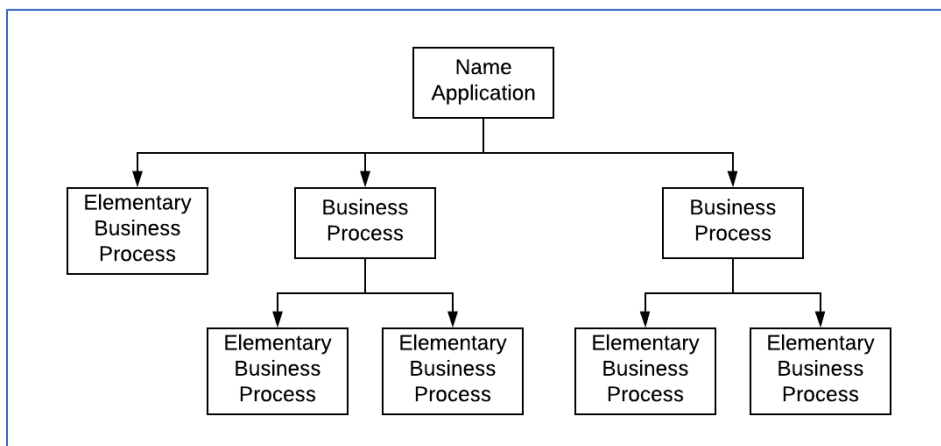


Figure 4. Generic structure of process diagram.

In a process diagram there is no order in the processes, and it only shows which processes there will be in the system. It does not show how or when these will be executed. An **elementary business process** is carried out by one person, in one place, at one time.

This diagram is actually borrowed from business process modeling, which is a field in itself. However, if we consider the 'business processes' as functionalities the system should have, this becomes very useful for our system development. A practical example is shown in Figure 5.

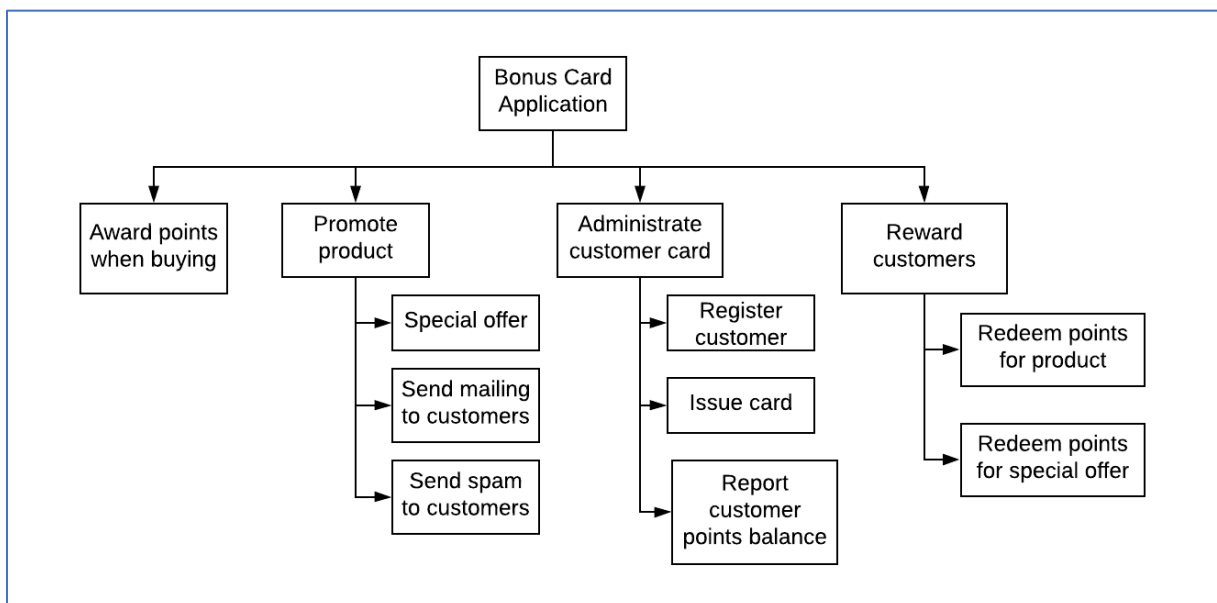


Figure 5. Example of a process diagram

Note that the process names always contain a verb and usually also a noun.

Creating this diagram together with the client gives a good foundation of what the expectations are. If at the end of the project the client decides that s/he had also expected the application to provide input for their logistics database, you can fall back on this diagram. The model should only show processes that will be supported by the application.

When creating a process diagram, you might run into a sequence of actions that will always be the same. For example the 'Send mailing to customers' would include: 'Drafting text mailing', 'Retrieve list of mailing victims', 'Create new mailing', and 'Send mailing'. These processes are simply a sequence of actions that will always take place for this process. The advice is to not show this level of detail. Once you notice your process is just a sequence of events, stop there.

We can work out the details of each elementary business process in a so-called UseCase diagram. This will be the first UML diagram that will be discussed, but before we can discuss that diagram, it is good to know what UML actually is.

Note that the term 'process diagram' can have many different meanings. In this course we used the ideas from Hoogendoorn (2004).

### 3 Unified Modeling Language (UML)<sup>3</sup>

UML, short for **Unified Modeling Language**, is a standardized modeling language consisting of an integrated set of diagrams, developed to help system and software developers for specifying, visualizing, constructing, and documenting the artefacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems. The UML is a very important part of developing object-oriented software and the software development process. The UML uses mostly graphical notations to express the design of software projects. Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of the software. In this article, we will give you detailed ideas about what is UML, the history of UML and a description of each UML diagram type, along with UML examples.

#### 3.1 The Origin of UML

The goal of UML is to provide a standard notation that can be used by all object-oriented methods and to select and integrate the best elements of precursor notations. UML has been designed for a broad range of applications. Hence, it provides constructs for a broad range of systems and activities (e.g., distributed systems, analysis, system design and deployment).

UML is a notation that resulted from the unification of OMT from

1. Object Modeling Technique OMT (Rumbaugh, Blaha, Premerlani, Eddy, & Lorensen, 1991) was best for analysis and data-intensive information systems.
2. Booch (Booch, 1994) was excellent for design and implementation. Grady Booch had worked extensively with the Ada language, and had been a major player in the development of Object Oriented techniques for the language. Although the Booch method was strong, the notation was less well received (lots of cloud shapes dominated his models - not very tidy)
3. OOSE (Object-Oriented Software Engineering) (Jacobson, 1992) featured a model known as Use Cases. Use Cases are a powerful technique for understanding the behaviour of an entire system (an area where OO has traditionally been weak).

In 1994, Jim Rumbaugh, the creator of OMT, stunned the software world when he left General Electric and joined Grady Booch at Rational Corp. The aim of the partnership was to merge their ideas into a single, unified method (the working title for the method was indeed the "Unified Method").

By 1995, the creator of OOSE, Ivar Jacobson, had also joined Rational, and his ideas (particularly the concept of "Use Cases") were fed into the new Unified Method - now called the Unified Modeling Language<sup>1</sup>. The team of Rumbaugh, Booch and Jacobson are affectionately known as the "Three Amigos".

---

<sup>3</sup> Most of this chapter was taken from Visual Paradigm website: <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/>. Their text was copied and some parts adjusted or extended with their permission.

UML has also been influenced by other object-oriented notations:

- Mellor and Shlaer (1988) with Object Oriented Systems Analysis (OOSA).
- Coad and Yourdon (1990) with Object Oriented Analysis (OOA).
- Wirfs-Brock and Wilkerson (Wirfs-Brock & Wilkerson, 1990)
- Martin and Odell (Martin & Odell, 1992)

UML also includes new concepts that were not present in other major methods at the time, such as extension mechanisms and a constraint language.

### 3.2 History of UML

During 1996, the first Request for Proposal (RFP) issued by the Object Management Group (OMG) provided the catalyst for these organizations to join forces around producing a joint RFP response.

Rational established the UML Partners consortium with several organizations willing to dedicate resources to work toward a strong UML 1.0 definition. Those contributing most to the UML 1.0 definition included: Digital Equipment Corp, HP, Logix, IntelliCorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational Software, TI, Unisys.

This collaboration produced UML 1.0, a modeling language that was well-defined, expressive, powerful, and generally applicable. This was submitted to the OMG in January 1997 as an initial RFP response.

In January 1997 IBM, ObjecTime, Platinum Technology, Ptech, Taskon, Reich Technologies and Softeam also submitted separate RFP responses to the OMG. These companies joined the UML partners to contribute their ideas, and together the partners produced the revised UML 1.1 response. The focus of the UML 1.1 release was to improve the clarity of the UML 1.0 semantics and to incorporate contributions from the new partners. It was submitted to the OMG for their consideration and adopted in the fall of 1997 and enhanced 1.1 to 1.5, and subsequently to UML 2.1 from 2001 to 2006 (now the UML current version is 2.5.1 released in December 2017).

In Figure 6 a timeline is shown of the various releases.

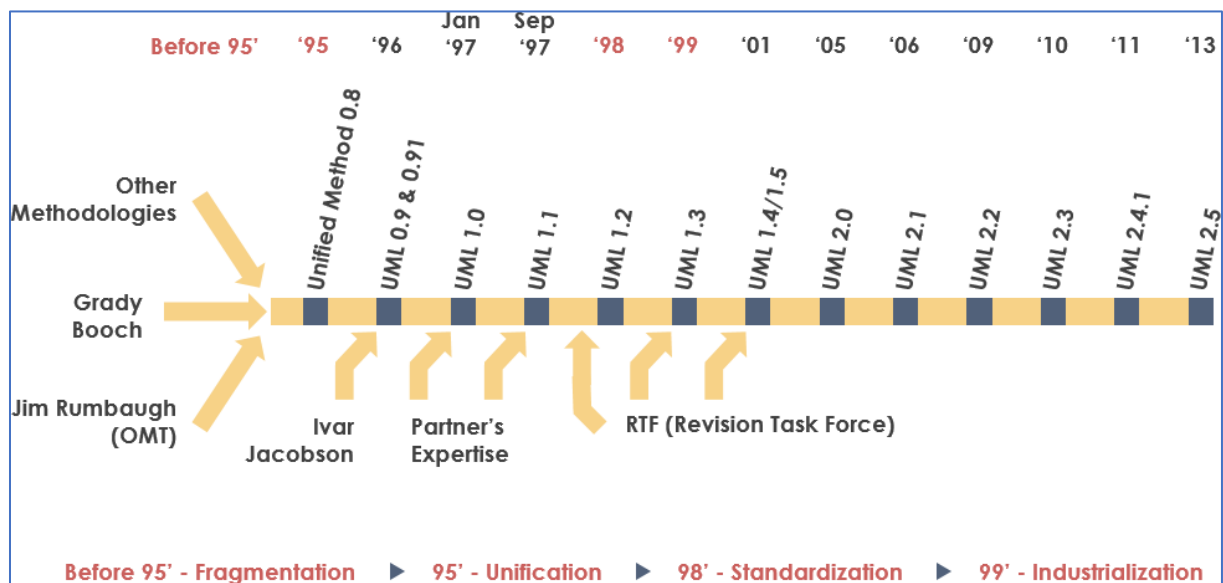


Figure 6. Timeline of UML up to version 2.5. Reprinted from Visual Paradigm with permission.

### 3.3 Why UML

As the strategic value of software increases for many companies, the industry looks for techniques to automate the production of software and to improve quality and reduce cost and time-to-market. These techniques include component technology, visual programming, patterns and frameworks. Businesses also seek techniques to manage the complexity of systems as they increase in scope and scale. In particular, they recognize the need to solve recurring architectural problems, such as physical distribution, concurrency, replication, security, load balancing and fault tolerance. Additionally, the development for the World Wide Web, while making some things simpler, has exacerbated these architectural problems. The Unified Modeling Language (UML) was designed to respond to these needs. The primary goals in the design of the UML summarize by Page-Jones (2000) as follows:

1. Provide users with a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.
2. Provide extensibility and specialization mechanisms to extend the core concepts.
3. Be independent of particular programming languages and development processes.
4. Provide a formal basis for understanding the modeling language.
5. Encourage the growth of the OO tools market.
6. Support higher-level development concepts such as collaborations, frameworks, patterns and components.
7. Integrate best practices.

### 3.4 UML - An Overview

Before we begin to look at the theory of the UML, we are going to take a very brief run through some of the major concepts of the UML.

The first thing to notice about the UML is that there are a lot of different diagrams (models) to get used to. The reason for this is that it is possible to look at a system from many different viewpoints. A software development will have many stakeholders playing a part.

For example: Analysts, Designers, Coders, Testers, QA, The Customer, Technical Authors

All of these people are interested in different aspects of the system, and each of them require a different level of detail. For example, a coder needs to understand the design of the system and be able to convert the design to a low level code. By contrast, a technical writer is interested in the behaviour of the system as a whole, and needs to understand how the product functions. The UML attempts to provide a language so expressive that all stakeholders can benefit from at least one UML diagram.

Figure 7 shows the breakdown of the 14 UML diagrams.

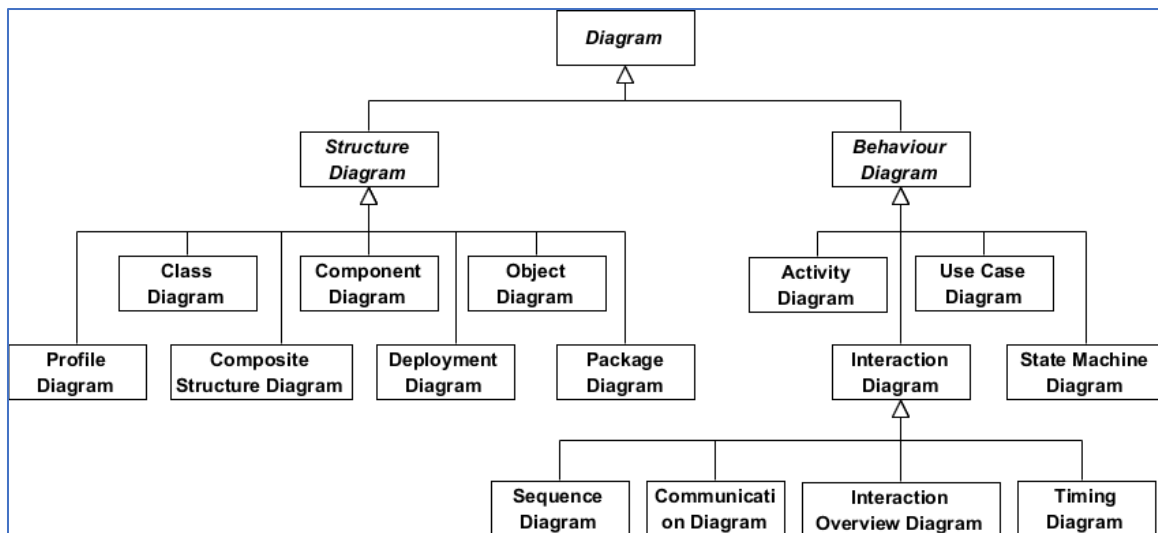


Figure 7. Break-down of UML diagrams. Reprinted from Visual Paradigm with permission.

As can be seen from Figure 7 the diagrams are categorized in two categories: Structure and Behavior.

**Structure diagrams** show the static structure of the system and its parts on different abstraction and implementation levels and how they are related to each other. The elements in a structure diagram represent the meaningful concepts of a system, and may include abstract, real world and implementation concepts, there are seven types of structure diagram as follows:

1. Class Diagram
2. Component Diagram
3. Deployment Diagram
4. Object Diagram
5. Package Diagram
6. Composite Structure Diagram
7. Profile Diagram

**Behavior diagrams** show the dynamic behavior of the objects in a system, which can be described as a series of changes to the system over time, there are seven types of behavior diagrams as follows:

1. Use Case Diagram
2. Activity Diagram
3. State Machine Diagram
4. Sequence Diagram
5. Communication Diagram
6. Interaction Overview Diagram
7. Timing Diagram

In this introduction course we will only discuss five UML diagrams: Use Case, Activity, Class, Sequence and State Machine.

## 4 Use Case Diagrams<sup>4</sup>

Here are some questions that have been asked frequently in the UML world are: **What is a use case diagram?** **Why Use case diagram?** or simply, **Why use cases?**. Some people don't know what use case is, while the rest under-estimated the usefulness of use cases in developing a good software product. Is use case diagram underrated? I hope you will find the answer when finished reading this chapter.

So what is a use case diagram? A UML use case diagram is the primary form of system/software requirements for a new software program underdeveloped. Use cases specify the expected behavior (what), and not the exact method of making it happen (how). Use cases once specified can be denoted both textual and visual representation (i.e. use case diagram). A key concept of use case modeling is that it helps us design a system from the end user's perspective. It is an effective technique for communicating system behavior in the user's terms by specifying all externally visible system behavior.

The UML Specification and the Reference Manual define a **UseCase** as: “UseCases are a means to capture the requirements of systems, i.e., what systems are supposed to do” (Object Management Group, 2017, p. 639; Rumbaugh, Jacobson, & Booch, 1999, p. 63). A slightly longer definition is given by Cockburn (2000): “a use case is a description of the possible sequences of interactions between the system under discussion and its external actors, related to a particular goal” (p. 15).

A use case diagram is usually simple. It does not show the detail of the use cases:

- It only summarizes some of the relationships between use cases, actors, and systems.
- It does not show the order in which steps are performed to achieve the goals of each use case.

As said, a use case diagram should be simple and contains only a few shapes. If yours contain more than 20 use cases, you are probably misusing use case diagram.

Figure 8 shows the UML diagram hierarchy and the positioning of the UML Use Case Diagram.

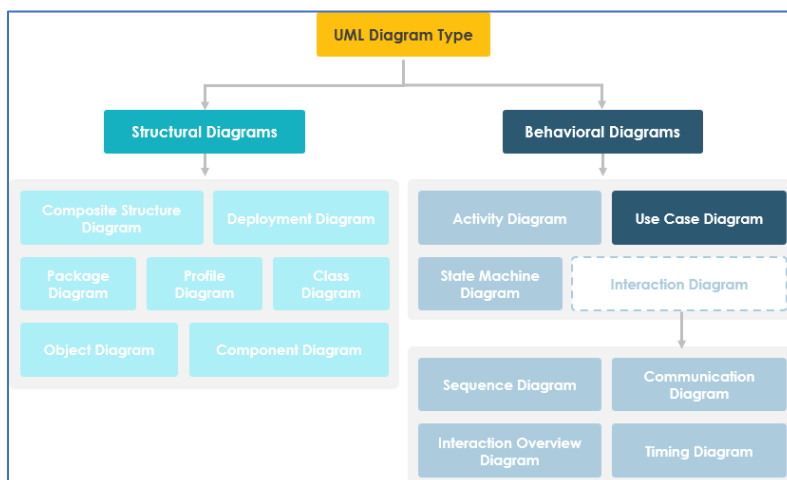


Figure 8. UML diagrams overview. Reprinted from Visual Paradigm with permission.

As you can see, use case diagrams belong to the family of behavioral diagrams.

<sup>4</sup> Most of this chapter was taken from Visual Paradigm website: <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/>. Their text was copied and some parts adjusted or extended with their permission. Chapter 0 is the only section not taken based on their material.

Note that:

- There are many different UML diagrams that serve different purposes (as you can see from the UML diagram tree above). You can describe those details in other UML diagram types and documents, and have them be linked from use cases.
- Use cases represent only the functional requirements of a system. Other requirements such as business rules, quality of service requirements, and implementation constraints must be represented separately, again, with other UML diagrams.
- The UML specification uses UseCase, but in most other textbooks it is written as Use Case.

### Origin of Use Case

These days use case modeling is often associated with UML, although it has been introduced before UML existed. Its brief history is as follow:

- In 1986, Ivar Jacobson first formulated textual and visual modeling techniques for specifying use cases. He then called them Usage Scenarios and Usage Cases.
- In 1992 his co-authored book *Object-Oriented Software Engineering - A Use Case Driven Approach* (Jacobson, 1992) helped to popularize the technique for capturing functional requirements, especially in software development.

### Purpose of Use Case Diagram

Use case diagrams are typically developed in the early stage of development and people often apply use case modeling for the following purposes:

- Specify the context of a system
- Capture the requirements of a system
- Validate a systems architecture
- Drive implementation and generate test cases
- Developed by analysts together with domain experts

#### 4.1 Use Case Diagram Basic Components

A standard form of use case diagram is defined in the Unified Modeling Language as shown in the Use Case Diagram example in Figure 9.

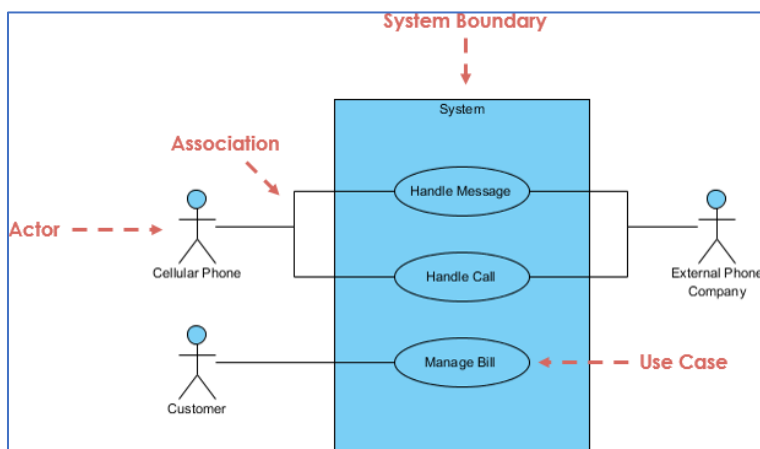


Figure 9. Use case diagram example. Reprinted from Visual Paradigm with permission.

Figure 9 has four basic Use Case elements (indicated with red) that you will often see. Lets go into those components in more detail.



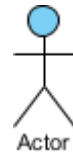
## Actor

### Definition:

Who or what executes the UseCase

### Symbol:

“an Actor is represented by a “stick man” icon with the name of the Actor in the vicinity (usually above or below) the icon” (Object Management Group, 2017, p. 642)



### Comments:

- Someone interacts with use case (system function).
- Named by noun.
- Actor plays a role in the business
- Similar to the concept of user, but a user can play different roles For example: A prof. can be instructor and also researcher plays two roles with two systems
- Actor triggers use case(s).
- Actor has a responsibility toward the system (inputs), and
- Actor has expectations from the system (outputs).
- Although not part of the UML specification often the term primary and secondary actor is used:
  - Primary actor:  
“The external actor with the goal the system is supposed to satisfy as one of its services” (Cockburn, 2000, p. 200) (Cockburn, 2000, p. 200).
  - Secondary actor:  
“An external actor against which the system under design has a goal” (Cockburn, 2000, p. 200)

## Use Case

### Definition:

“a logical description of part of the system” (Rumbaugh, Jacobson, & Booch, 1999, p. 65)



### Symbol:

“an ellipse, with the name of the UseCase in or below it” (Object Management Group, 2017, p. 641)

### Comments

- System function (process - automated or manual)
- Named by verb + Noun (or Noun Phrase) i.e. do something
- Each Actor must be linked to a use case, while some use cases may not be linked to actors.
- Although not part of the UML specification the term primary and secondary use case are often used:
  - Primary use case:  
UseCase where Actor has a direct interest in, to achieve it's goal
  - Secondary use case:  
UseCase where Actor has an indirect interest in, to achieve it's goal

## Boundary of system

### Definition:

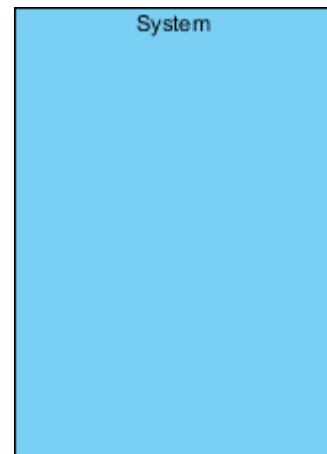
“A subject for a set of UseCases (sometimes called a system boundary)” (Object Management Group, 2017, p. 641)

### Symbol:

“may be shown as a rectangle with its name in the top-left corner, with the UseCase ellipses visually located inside this rectangle” (Object Management Group, 2017, p. 641)

### Comments:

- The system boundary is potentially the entire system as defined in the requirements document.
- For large and complex systems, each module may be the system boundary.
- For example, for an ERP system for an organization, each of the modules such as personnel, payroll, accounting, etc.
- can form a system boundary for use cases specific to each of these business functions.
- The entire system can span all of these modules depicting the overall system boundary



Usually primary actors are placed to the left of the system boundary, and secondary actors to the right

## Association

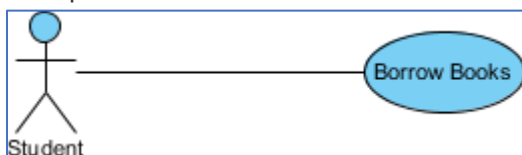
### Definition:

“an Actor can only have Associations to UseCases, Components, and Classes. Furthermore these Associations must be binary” (Object Management Group, 2017, p. 647). Note that this is not a generic definition for ‘association’ but specifically the association from Actor to UseCase.

### Notation:

“normally rendered as a solid line connecting two classifiers” (uml-diagrams.org, n.d.)

### Example:



### Comments:

- The participation of an actor in a use case is shown by connecting an actor to a use case by a solid link.
- Actors may be connected to use cases by associations, indicating that the actor and the use case communicate with one another using messages.

## 4.2 Use Case Diagram Relationships

Use cases share different kinds of relationships. Defining the relationship between two use cases is the decision of the software analysts of the use case diagram. A relationship between two use cases is basically modeling the dependency between the two use cases. The reuse of an existing use case by using different types of relationships reduces the overall effort required in developing a system. Use case relationships are listed as the following:

### Extends

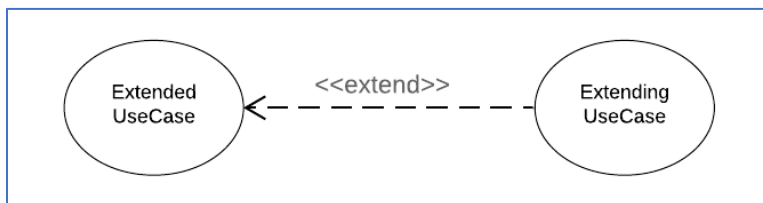
#### Definition:

“a relationship from an extending UseCase (the extension) to an extended UseCase (the extendedCase) that specifies how and when the behavior defined in the extending UseCase can be inserted into the behavior defined in the extended UseCase” (Object Management Group, 2017, p. 640)

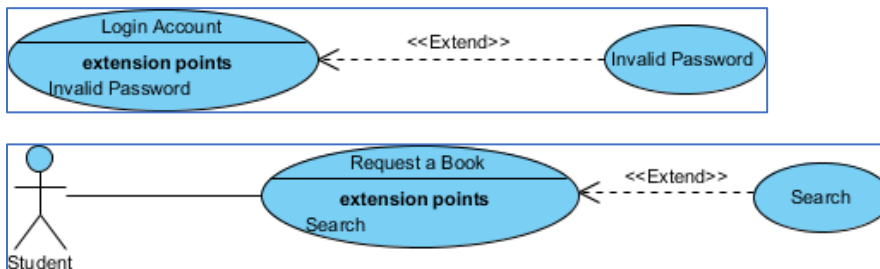
#### Symbol:

“shown by a dashed arrow with an open arrowhead pointing from the extending UseCase towards the extended UseCase. The arrow is labeled with the keyword <<extend>>” (Object Management Group, 2017, p. 642)

#### Generic example:



#### Example:



#### Comment:

- Indicates that an "Invalid Password" use case may include (subject to specified in the extension) the behavior specified by base use case "Login Account".
- Depict with a directed arrow having a dotted line. The tip of arrowhead points to the base use case and the child use case is connected at the base of the arrow.
- The stereotype "<<extends>>" identifies as an extend relationship

## Include

### Definition:

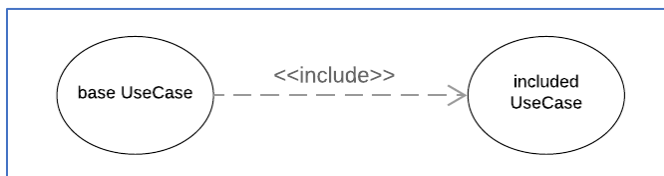
“a DirectedRelationship between two UseCases, indicating that the behavior of the included UseCase (the addition) is inserted into the behavior of the including UseCase (the includingCase)”

(Object Management Group, 2017, p. 641)

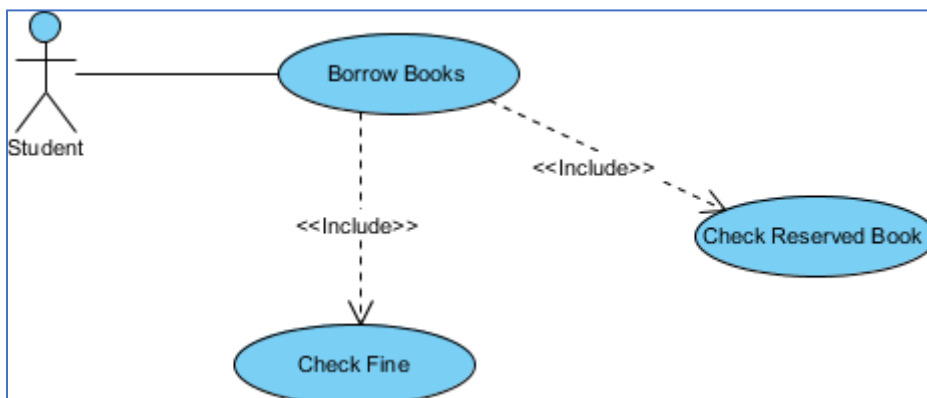
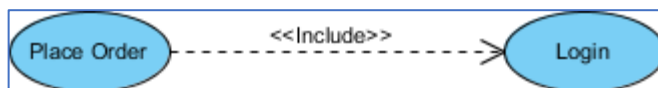
### Symbol:

“shown by a dashed arrow with an open arrowhead pointing from the base UseCase to the included UseCase. The arrow is labeled with the keyword «include»” (Object Management Group, 2017, p. 642)

### Generic example



### Examples:



### Comments:

- When a use case is depicted as using the functionality of another use case, the relationship between the use cases is named as include or uses relationship.
- A use case includes the functionality described in another use case as a part of its business process flow.
- A uses relationship from base use case to child use case indicates that an instance of the base use case will include the behavior as specified in the child use case.
- An include relationship is depicted with a directed arrow having a dotted line. The tip of arrowhead points to the child use case and the parent use case connected at the base of the arrow.
- The stereotype "«include»" identifies the relationship as an include relationship.

## Generalization

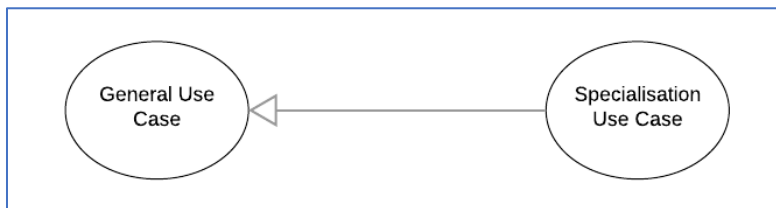
### Definition:

“a generalization relationship between use cases implies that the child use case contains all the attributes, sequences of behavior and extension points defined in the parent use case, and participates in all the relationships of the parent use case” (Cockburn, 2000, p. 95).

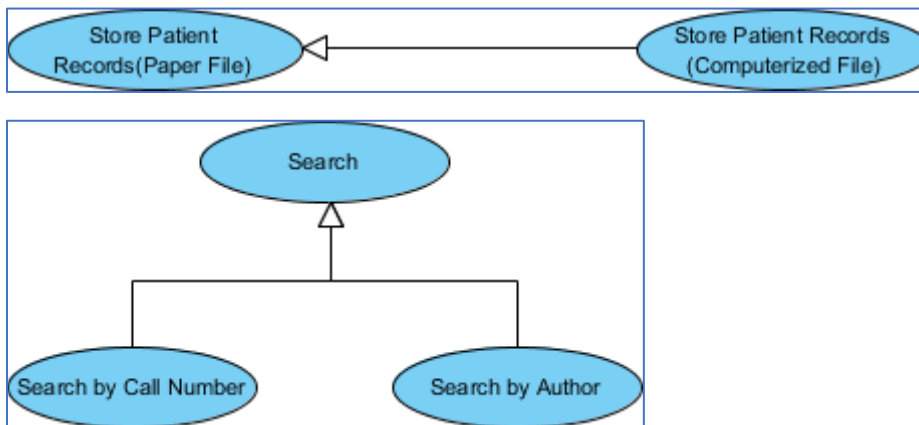
### Notation:

“an arrow goes from the specialized use case back to the general use” (Cockburn, 2000, p. 96), with an open triangle arrow head (Object Management Group, 2017, p. 646).

### Generic example:



### Example:



### Comments:

- A generalization relationship is a parent-child relationship between use cases.
- The child use case is an enhancement of the parent use case.
- Generalization is shown as a directed arrow with a triangle arrowhead.
- The child use case is connected at the base of the arrow. The tip of the arrow is connected to the parent use case.

### 4.3 Use Case Diagram - Examples

The figure below shows a use case diagram example for a vehicle system. As you can see even a system as big as a vehicle sales system contains not more than 10 use cases! That's the beauty of use case modeling.

The use case model also shows the use of extend and include. Besides, there are associations that connect between actors and use cases.

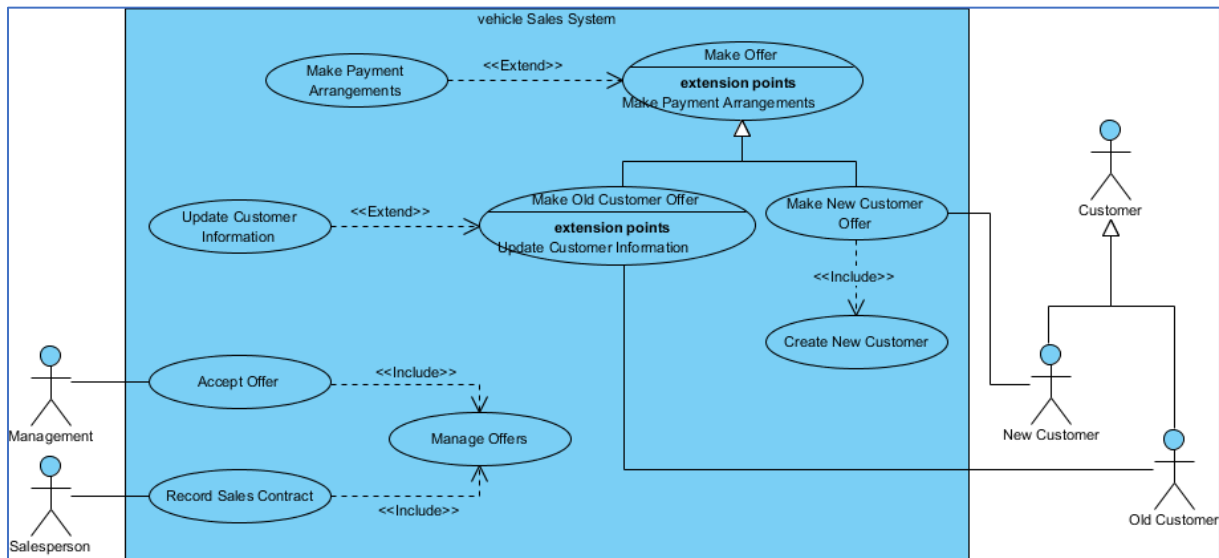


Figure 10. UseCase diagram of a vehicle system. Reprinted from Visual Paradigm with permission.

Another example of a UseCase diagram is shown in Figure 11.

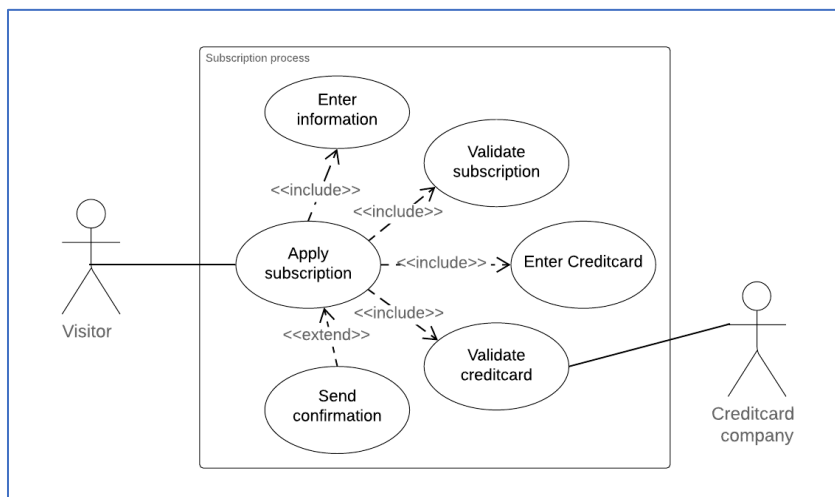


Figure 11. UseCase diagram of subscription.

Note the order of the UseCases. Although a UseCase diagram does not reveal the order the UseCases should/could be executed in, they are usually placed clockwise starting at 12:00. This is sometimes referred to as **'the hand' method** (Hoogendoorn, 2004). Figure 12 shows where this name comes from.

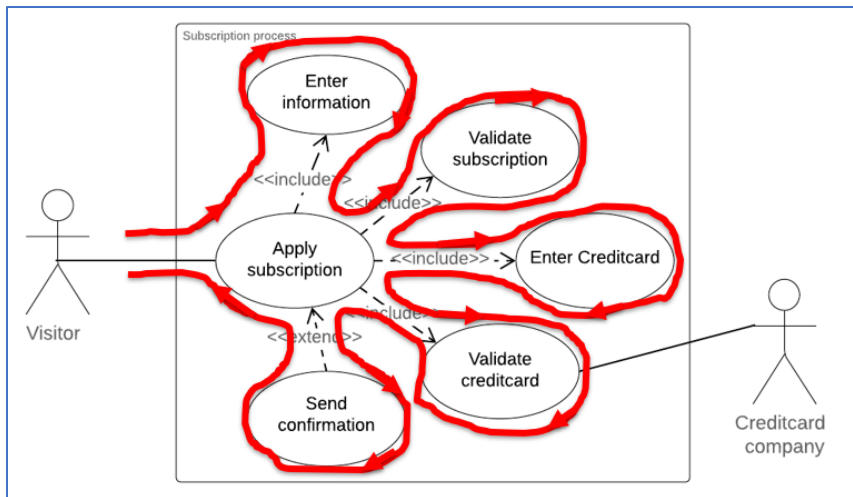


Figure 12. Explanation of term 'hand method'.

The red line in Figure 12 starts at the primary actor and then goes around the use cases, with a bit of imagination it looks like a hand (of Mickey Mouse?).

## 4.4 Creating a UseCase Diagram

### 4.4.1 How to Identify Actor

Often, people find it easiest to start the requirements elicitation process by identifying the actors.

The following questions can help you identify the actors of your system (Schneider & Winters, 1998):

- Who uses the system?
- Who installs the system?
- Who starts up the system?
- Who maintains the system?
- Who shuts down the system?
- What other systems use this system?
- Who gets information from this system?
- Who provides information to the system?
- Does anything happen automatically at a present time?

### 4.4.2 How to Identify Use Cases

Identifying the Use Cases, and then the scenario-based elicitation process carries on by asking what externally visible, observable value that each actor desires. The following questions can be asked to identify use cases, once your actors have been identified (Schneider & Winters, 1998):

- What functions will the actor want from the system?
- Does the system store information? What actors will create, read, update or delete this information?
- Does the system need to notify an actor about changes in the internal state?
- Are there any external events the system must know about? What actor informs the system of those events?

#### 4.4.3 Use Case Levels of Details

Use case **granularity** refers to the way in which information is organized within use case specifications, and to some extent, the level of detail at which they are written. Achieving the right level of use case granularity eases communication between stakeholders and developers and improves project planning.

Alastair Cockburn in *Writing Effective Use Cases* (2000) gives an easy way to visualize different levels of goal level by thinking in terms of the sea as shown in Figure 13.






Level	Example	Summary
 <b>Cloud</b>	Product	<b>High Summary</b>
 <b>Kite</b>	Sell Products	<b>Summary</b>
 <b>Sea</b>	Sell Furniture	<b>User goal</b>
 <b>Fish</b>	Browse Catalog	<b>Sub-function</b>
 <b>Clam</b>	Insert Orderline	<b>Low level</b>

Figure 13. Visualisation of different levels of goal levels. Reprinted from Visual Paradigm with permission.

Note that:

- While a use case itself might drill into a lot of detail about every possibility, a use-case diagram is often used for a higher-level view of the system as blueprints.
- It is beneficial to write use cases at a coarser level of granularity with less detail when it's not required.



#### 4.5 Use Case Description

Although not in the UML specification, most authors of UML will mention to always create a **UseCase description**. This is a text document often based on a template, describing some features of the UseCase (so one for each Use case, NOT just one per UseCase Diagram). In this description you can explain more of the UseCase and avoid making the diagram itself too large.

Unfortunately, though almost each author has his/her own template. In Figure 14 an extensive version is shown from Cockburn (2000).

<b>USE CASE #</b>	< the name is the goal as a short active verb phrase>	
<b>Context of use</b>	<a longer statement of the context of use if needed>	
<b>Scope</b>	<what system is being considered black box under design>	
<b>Level</b>	<one of : Summary, Primary Task, Subfunction>	
<b>Primary actor</b>	<a role name for the primary actor, or description>	
<b>Stakeholder &amp; Interests</b>	<b>Stakeholder</b>	<b>Interest</b>
	<stakeholder name>	<put here the interest of the stakeholder>
	<stakeholder name>	<put here the interest of the stakeholder>
<b>Preconditions</b>	<what we expect is already the state of the world>	
<b>Success End Condition</b>	<the state of the world upon successful completion>	
<b>Failed End Protection</b>	<the proper state of the world if goal abandoned>	
<b>Trigger</b>	<the action upon the system that starts the use case>	
<b>DESCRIPTION</b>	<b>Step</b>	<b>Action</b>
	1	<put here the steps of the scenario from trigger to goal delivery, and any cleanup after>
	2	<...>
	3	
<b>EXTENSIONS</b>	<b>Step</b>	<b>Branching Action</b>
	1a	<condition causing branching> : <action or name of sub-use case>
<b>VARIATIONS</b>		<b>Branching Action</b>
	1	<list of variation s>

Figure 14. UseCase Description template of Cockburn.

In this course we will use another template, that only has some of the basic features that are seen in almost every template. The template is shown in Table 3.

Table 3

Template for UseCase Description

UC Name	
<b>Goal</b>	Description of the goal of the UseCase?
<b>Actor(s)</b>	Which Actor(s) can initialize the UseCase?
<b>Precondition(s)</b>	What must already have been done to initialize the UseCase?
<b>Activities</b>	Which activities should be executed by the UseCase?
<b>Postcondition(s)</b>	What is now true?

An example of a UseCase description of the 'Search Profile' UseCase (see Figure 11) is shown in Table 4.

Table 4  
*Example of UseCase Description*

UC Name	
Goal	Actor searches a profile using search criteria
Actor(s)	Visitor, Subscriber
Precondition(s)	None
Activities	<ol style="list-style-type: none"><li>1. System shows search page</li><li>2. If Actor cancels:<ol style="list-style-type: none"><li>1. Stop</li></ol></li><li>3. Actor enters search criteria</li><li>4. If Actor confirm search:<ol style="list-style-type: none"><li>1. System searches profile</li></ol></li><li>5. If profile not found:<ol style="list-style-type: none"><li>1. Repeat from step 1</li></ol></li></ol>
Postcondition(s)	<ul style="list-style-type: none"><li>• Profile is known OR</li><li>• Actor has cancelled</li></ul>

#### 4.6 Use Case Diagram Tips

Now, check the tips below to see how to apply use case effectively in your software project.

- Always structure and organize the use case diagram from the perspective of actors.
- Use cases should start off simple and at the highest view possible. Only then can they be refined and detailed further.
- Use case diagrams are based upon functionality and thus should focus on the "what" and not the "how".
- Remember the priority of any diagram is always: communication.

Use Cases are probably the most written about diagrams from all UML diagrams. Separate books only about Use Cases can be found. This highlights the importance of this diagram in modeling.

## 5 Activity Diagrams

We started with process diagrams, from each of the Elementary Business Process we made a UseCase diagram and from each UseCase in the UseCase diagram a UseCase Description. In the UseCase description the sequence of events that should take place in there is described. These sequences can be visualised using an Activity Diagram.

Activity diagram is another important behavioral diagram in UML diagram (Figure 15) to describe dynamic aspects of the system. Activity diagram is essentially a version of flow chart that modeling the flow from one activity to another activity.

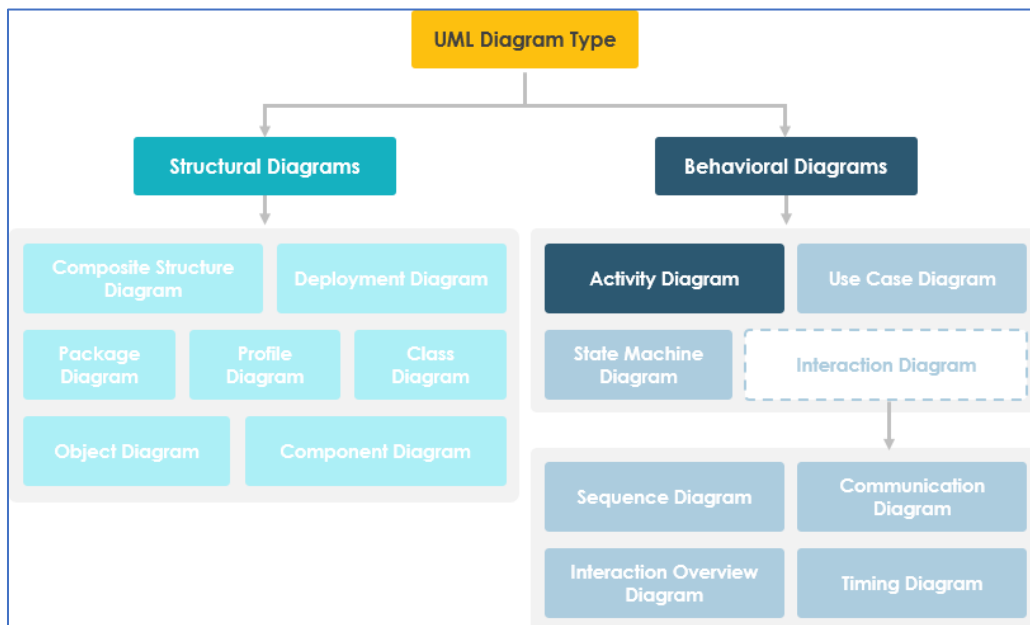


Figure 15. UML diagrams overview. Reprinted from Visual Paradigm with permission.

The UML 2.5.1 Specification and the Reference Manual define an **Activity (diagram)** as: “an Activity is a Behavior specified as sequencing of subordinate units, using a control and data flow model” (Object Management Group, 2017, p. 373; Rumbaugh, Jacobson, & Booch, 1999, p. 81).

### 5.1 Quick history

In versions prior to 2.0 of UML, the Activity Diagrams were seen as a special case of State Machine Diagrams (Object Management Group, 2003, pp. 3-155) (a diagram we will discuss in a later chapter). So, if you look at Activity Diagrams prior to 2003 they will need a different interpretation. In version 2.0 the documentation mentions that the Activity Diagrams are now more in line with so-called Petri-Nets (Object Management Group, 2005, p. 314). Indeed the Activity Diagram can be mapped to Petri-Nets in simple cases, but in advanced diagrams this is no longer possible. (Störrle & Hausmann, 2005).

## 5.2 When to Use Activity Diagram<sup>5</sup>

**Activity Diagrams** describe how activities are coordinated to provide a service which can be at different levels of abstraction. Typically, an event needs to be achieved by some operations, particularly where the operation is intended to achieve a number of different things that require coordination, or how the events in a single use case relate to one another, in particular, use cases where activities may overlap and require coordination. It is also suitable for modeling how a collection of use cases coordinate to represent business workflows

1. Identify candidate use cases, through the examination of business workflows
2. Identify pre- and post-conditions (the context) for use cases
3. Model workflows between/within use cases
4. Model complex workflows in operations on objects
5. Model in detail complex activities in a high-level activity Diagram

## 5.3 Activity Diagram Components

A basic activity diagram - flowchart is shown in Figure 16.

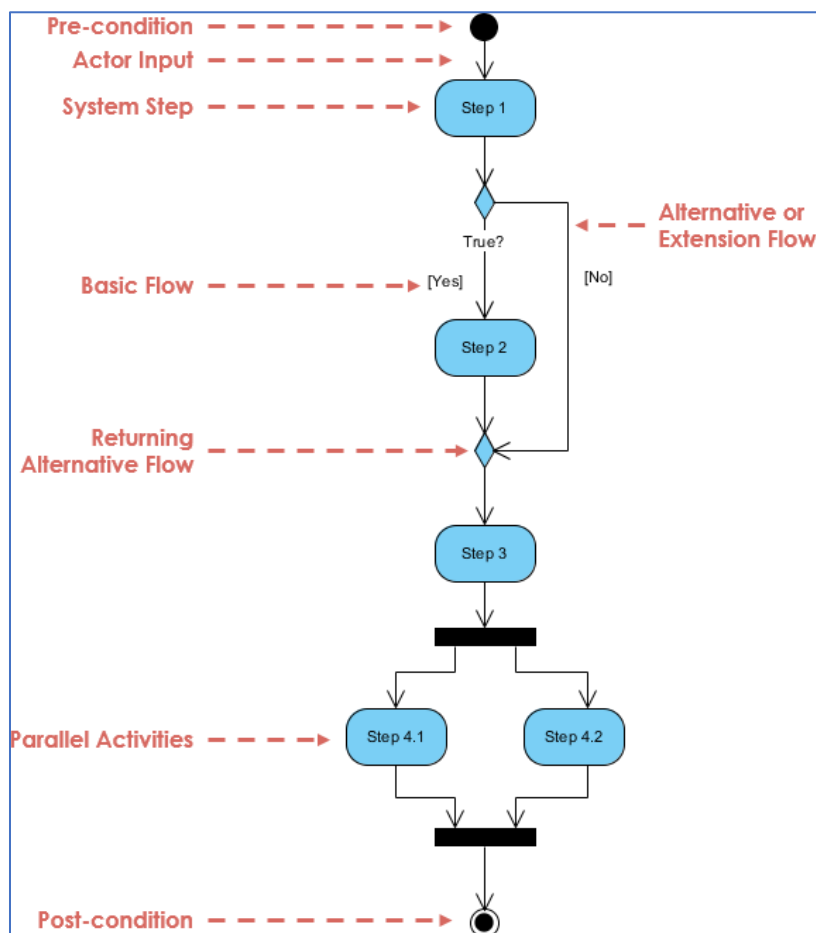


Figure 16. Example structure of Activity Diagram. Reprinted from Visual Paradigm with permission.

Any Activity Diagram will usually have at least the following components: Initial Node, Final Node, Activities, and Activity Edges. Each will briefly be discussed.

<sup>5</sup> This section was copied from Visual Paradigm with their permission.

### Initial node



#### Definition:

“a ControlNode that acts as a starting point for executing an Activity” (Object Management Group, 2017, p. 387).

#### Symbol:

“a solid circle” (Object Management Group, 2017, p. 391).

#### Comments:

- This is the starting point of the Activity Diagram. Usually placed on top (in case the flow is top-bottom) or the left (in case the flow is drawn left-right).
- The start of the activity might require some conditions to be assumed to be true (or false), these preconditions can be added by using a note with a dashed line going to the initial node and the guillemet <<precondition>>.

### Final node



#### Definition:

“a ControlNode at which a flow in an Activity stops. A FinalNode shall not have outgoing ActivityEdges” (Object Management Group, 2017, p. 388).

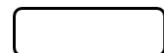
#### Symbol:

“solid circle within a hollow circle... This can be thought of as a goal notated as ‘bull’s eye’ or target” (Object Management Group, 2017, p. 391).

#### Comments:

- This indicates the end of the Activity Diagram. Usually placed at the bottom (in case the flow is top-bottom) or the right (in case the flow is drawn left-right).
- At the end of the activity it might be good to note what can be assumed to be true (or false) at the end. This can be added by using a note with a dashed line going to the final node and the guillemet <<postcondition>>.

### Activity node



#### Definition:

“used to model the individual steps in the behavior specified by an Activity” (Object Management Group, 2017, p. 375).

#### Symbol:

a rectangle with rounded corners (Object Management Group, 2017, p. 380).

#### Comments:

Inside the node the name of the activity is placed. This starts with a verb. If the flow also contains activities that are done by different Actors it starts with the name of the actor or system, followed by the verb and action.

## Activity Edge



### Definition:

“a directed connection between two ActivityNodes along which tokens may flow, from the source ActivityNode to the target ActivityNode” (Object Management Group, 2017, p. 375).

### Symbol:

“an open arrowhead line” (Object Management Group, 2017, p. 380).

### Comments:

These connect the various elements of the activities, they show the flow.

## 5.4 Decision and Merge nodes



As you might have noticed the example shown in Figure 16, also has a diamond shape. This diamond shape is used for both so-called Decision Nodes, as well as Merge Nodes.

### Decision node

#### Definition:

“a ControlNode that chooses between outgoing flows” (Object Management Group, 2017, p. 390).

#### Symbol:

“a diamond-shaped symbol,... must have a single incoming ActivityEdge ... and multiple outgoing ActivityEdges” (Object Management Group, 2017, p. 392).

#### Comments:

- A decision node can be considered like an if-statement in programming, and depending on where the activity goes, even as a loop.
- Usually to indicate which edge represents which decision so-called ‘guards’ are placed in square brackets near the tail of the line (Object Management Group, 2017, p. 380). Try to be as specific as possible in the description of the guard, but ‘else’ is also acceptable.
- Although outside the scope of this course, the UML specification does allow also two incoming edges, but one of those should then be called the primary incoming edge (Object Management Group, 2017, p. 390).

### Merge Node

#### Definition:

“a control node that brings together multiple flows without synchronization” (Object Management Group, 2017, p. 389).

#### Symbol:

“a diamond-shaped symbol,... must have two or more incoming ActivityEdges and a single outgoing ActivityEdge” (Object Management Group, 2017, p. 392).

#### Comments:

- Although you could direct an Activity Edge to an Activity Node that already has an incoming edge, it is preferred to add then a Merge Node.

## Example

The shape of a Merge and Decision node are the same, but based on the number of incoming and/or outgoing flows it should be possible to determine which is which. Have a look at Figure 17 and see if you can identify which is the Merge and which is the Decision node.

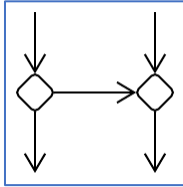


Figure 17. Merge vs. Decision.

The node on the left has only one incoming flow and two outgoing. This makes it a decision node. The node on the right has two incoming flows and only one outgoing. This makes it a merge node. Note that the decision node is lacking any guards, so it is not possible to know when we should move down, and when to the right.

## 5.5 Fork and Join nodes

If you look carefully in Figure 16, you also see two thick horizontal lines (bars). These are a Fork and a Join node, and are used to allow for parallel processes.

### Fork Node

#### Definition:

“a ControlNode that splits a flow into multiple concurrent flows” (Object Management Group, 2017, p. 388).

#### Symbol:

“a line segment” (Object Management Group, 2017, p. 391).

#### Comments:

- Although the specification only mentions a line segment as the symbol, the examples in the specification show that this line should be a thick line.
- A ForkNode should have only one incoming flow and usually has two or more outgoing (Object Management Group, 2017, p. 391).

### Join Node

#### Definition:

“a ControlNode that synchronizes multiple flows” (Object Management Group, 2017, p. 389).

#### Symbol:

“a line segment” (Object Management Group, 2017, p. 391).

#### Comments:

- Although the specification only mentions a line segment as the symbol, the examples in the specification show that this line should be a thick line.
- A JoinNode usually has two or more incoming flows, but must have only one outgoing (Object Management Group, 2017, p. 391).
- Note that ‘Join’ and ‘Merge’ in the English language might be synonyms, in UML they are two different things.

## Example

In Figure 18 an example is shown of an Activity diagram using all discussed elements.

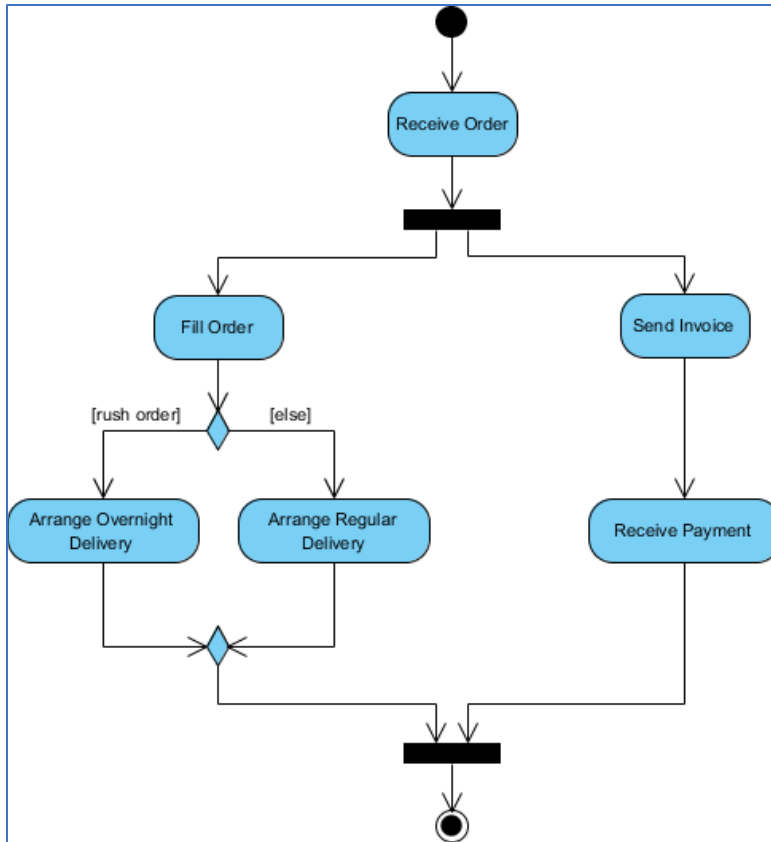


Figure 18. Activity diagram example with Fork and Join nodes. Reprinted from Visual Paradigm with permission.

The diagram starts on top with the Initial Node. Following the activity edge, we reach a Fork Node. Here two processes will start parallel to each other. It indicates we can both start Filling the order, and sending the invoice. At filling the order, we have a decision node, with guards. The guard to the left indicates that we follow the path to the left if there is a rush order, otherwise we go to the right. The result of these are then first merged and then both parallel processes reach the Join node. This indicates that both processes that are done parallel have to have been completed before we can move on. This is then also the end of this activity since we then go straight to the Final node.



## 6 Class Diagram

The Process Diagram, Use Case Diagram, and Activity Diagram, are all diagrams that can be used in discussion with your client to verify you understand each other. They are in comparison not so technical. To design the software, we often have an object-oriented approach. As the name implies this means we look at things as 'objects' and objects are so called instances of a class. For example, this reader could be an instance (object) of the class 'Reader'. To visualise the system in classes we can make use of the Class Diagram.

A Class Diagram describes the system in rest (static) and belongs to the 'Structural Diagrams' category in UML (Figure 19).

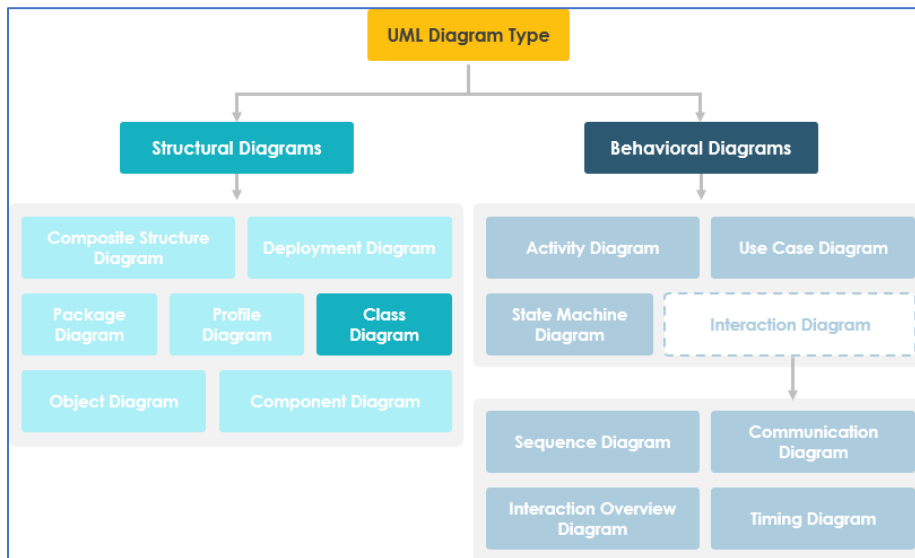


Figure 19. UML diagrams overview. Reprinted from Visual Paradigm with permission.

A **Class Diagram** defined as: “a graphic presentation of the static view that shows a collection of declarative (static) model elements, such as classes, types, and their contents and relationships” (Rumbaugh, Jacobson, & Booch, 1999, p. 190).

According to Visual Paradigm (n.d.) the purpose of a Class Diagram are:

- Shows static structure of classifiers in a system
- Diagram provides a basic notation for other structure diagrams prescribed by UML
- Helpful for developers and other team members too
- Business Analysts can use class diagrams to model systems from a business perspective

Note that with some modeling tools it would even be possible to generate the programming code from the Class Diagram.

## 6.1 The Class Node

As you might anticipate a Class Diagram is made out of Classes. A **Class** is defined as “the descriptor for a set of objects that share the same attributes, operations, methods, relationships, and behavior. A class represents a concept within the system being modeled” (Rumbaugh, Jacobson, & Booch, 1999, p. 185).

A class is often shown in a rectangle with the name of the class on top. The rectangle can also be split into three compartments with the second compartment showing the **attributes** (a.k.a. fields, or properties) of the class, the third showing the **methods** (a.k.a. messages or behaviour or technique or operations). A generic template is shown in Figure 20.

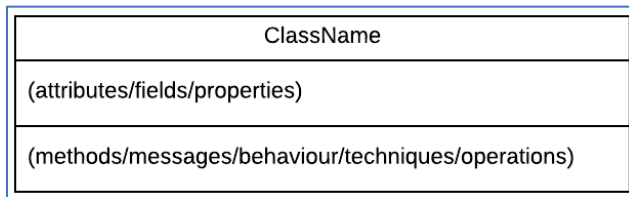


Figure 20. Template of a Class Node

The name of the class and methods are usually in PascalCase, while attributes are usually camelCase.

For the attributes and methods, we can use the **literals** public, private, protected and package (Object Management Group, 2017, p. 60). These are often indicated respectively with +, -, and #. You could also come across a ~ for package, a / for derived, or see the name of the attribute or method being underlined to indicate static. In this course we will only use public and private.

After the literal the name of the attribute is shown, followed by a colon and the type of property or return value in case of a method. An example is shown in Figure 21.

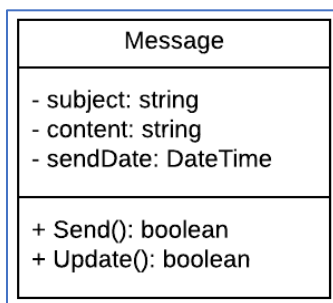


Figure 21. Example of a Class node

Note that sometimes only the class name is shown. This is done to simplify the view of the model and analyse the relations easier.

A class diagram will usually consist out of more than one class. The classes then relate to each other and this is where class diagrams become a bit more difficult.

## 6.2 Association and Multiplicity

Classes can have relations between them. If we play for example a Blackjack game, we probably have a class 'Card' and a class 'Player'. There should be a relation between those two classes. Note however that a Player probably has a Card, but a Card does not have a Player. To indicate this so-called association we use a solid line with an open arrow as shown in (Figure 22).

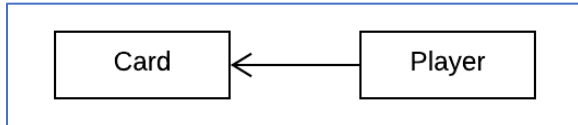


Figure 22. Association with no multiplicity

The figure indicates that at runtime Player knows about Card, but Card does not know about Player.

A Player has probably a few cards. The number of instances that an instance of another class can have is known as the **multiplicity**. This could either be a fixed value, or a range of values.

In Blackjack a player usually starts with 2 cards and could reach a maximum of 22. So the multiplicity would be 2..22. One could argue that at the start of the game the player has no cards, so 0..22 would also work. The cards the player has, are his/her 'hand' so this is also indicated in the diagram (Figure 23).

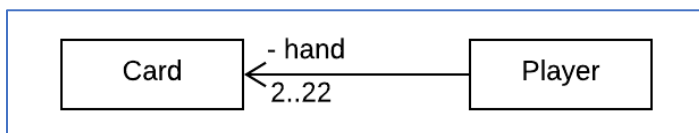


Figure 23. Association with multiplicity.

Note that 'hand' is also now an attribute of Player, even though it won't show in the Class node.

Although a Card is not aware of the Player class, there is a minimum and maximum. A card can at most belong to one player, and could also not belong to any player. It's multiplicity towards Player is therefore 0..1. This can also be added to our diagram as shown in Figure 24.

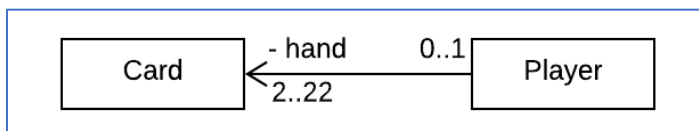


Figure 24. Association with multiplicity on both ends

Since the UML specification is not very clear on what a default multiplicity should be, it is best to simply always add it.

If the upper bound of the multiplicity is unknown you can use \*. If the lower bound and upper bound are equal, the value alone will be enough.

In some cases, no arrow is shown. The assumption is then that this is a bidirectional relation, i.e. it is the same as an association with an arrow at both ends.

### 6.3 Class Navigation

The multiplicity and arrows indicate how we can move from one class to another. Navigation however can be tricky. To investigate this a few examples will be shown in this section.

We start with Figure 25. It shows a simple restaurant system.

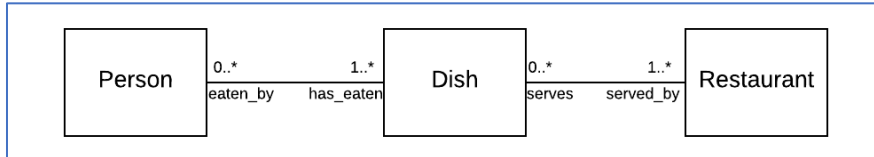


Figure 25. Navigation example 1. Reprinted from 'UML Class Diagrams for Software Engineering'.

According to this model: Is it possible for each Person to see what restaurant this Person went to?

A Person will be connected by a Dish. However, the Dish can be connected to multiple Restaurants. If for example both Peter and Lili eat a Salad, and the Salad can be served by McDonalds and Burger King, then we do know who ate which Dish, but cannot determine where each Person ate it.

Let's add a Reservation class to the diagram, as shown in Figure 26.

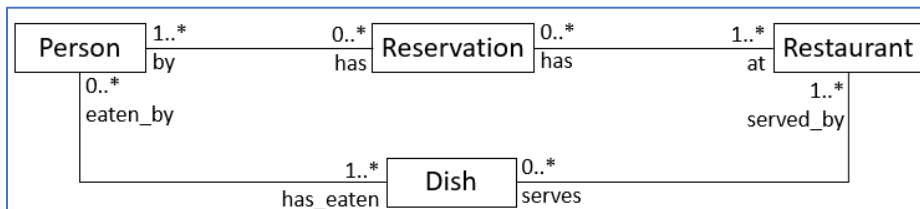


Figure 26. Navigation Example 2. Reprinted from 'UML Class Diagrams for Software Engineering'.

Is the following statement true or false?

In order for a reservation to exist, there always has to be at least one dish.

We want to go from Reservation to Dish. We start by exploring the option to go to the right to Restaurant. A reservation has to be at least at one restaurant. From Restaurant to Dish we get that a Restaurant could serve 0 Dishes. This path results in that a Reservation does not need a Dish. However, we have a second option. If we go to the left to Person. A Reservation is made by at least one person. A Person has eaten at least one Dish. So, the statement is True. Only one path needs to fulfil the requirement. You can think of this like for a Reservation to exist, there must be at least one Person, who also has to eat at least one Dish.

A last example. Two Class diagrams of a school are shown in Figure 26. They look similar but are really different.

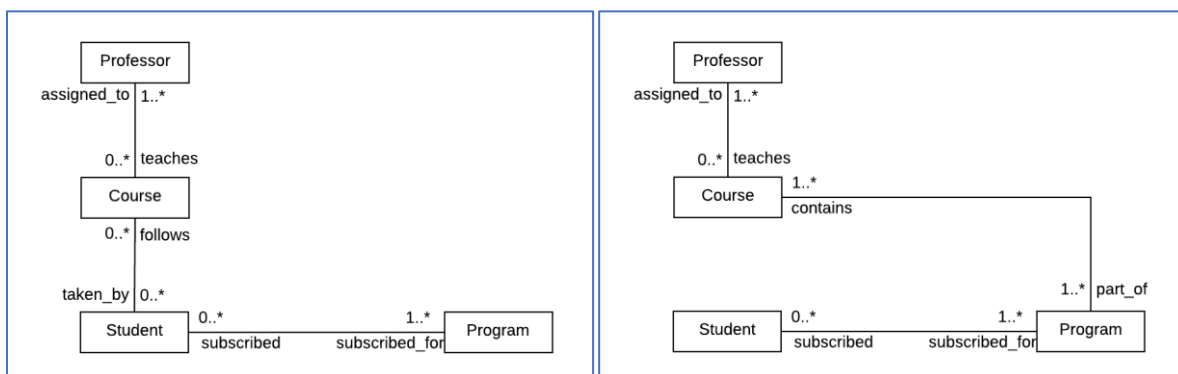


Figure 27. Navigation example 3. Reprinted from 'UML Class Diagrams for Software Engineering'.

Which of these two Class diagrams would allow a Professor to get a list of the students that s/he teaches?

The left diagram will show the requested result. A Professor teaches a course and from those courses can s/he obtain a list of students. In the right diagram the course can be part of multiple programs, but the Program only can get a list of all the students in that program. The Professor will get a list of all the students in all the programs the course is used in. This would include students who do not take the course.

#### 6.4 Aggregation and Composition

In section 6.2 associations were discussed. An association is one of three different types of relations that could exist. An association is a relation between two classes. If the relation is so that one can be considered the whole and the other(s) as part(s), the relation is called an aggregation. If the aggregation is so strong that one cannot exist without the other(s) it is called a composition.

Note that every composition is an aggregation, and every aggregation is an association. However, not every association is an aggregation and not every aggregation is a composition. They can be considered subsets of each other, as illustrated in Figure 28.

To determine if an association is an aggregation ask yourself: Is *ClassA* made up out of *ClassB*? If the answer is yes, it is an aggregation (and perhaps even a composition).

To determine if an aggregation is a composition ask yourself: Can the child class exist without the parent class? If the answer is no, it is a composition.

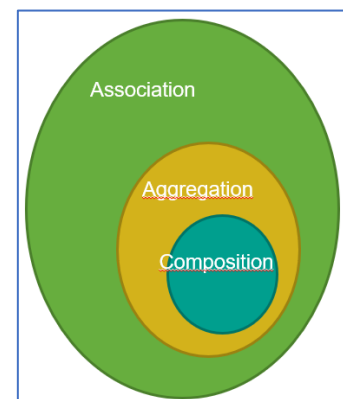


Figure 28. Association, Aggregation and Composition.

##### Example 1.

In an educational system we might have the classes Student and Course. Is a Course made up out of Students? Not really, a course is more than just a collection of students. Also Is a Student made up out of Courses? Well this doesn't make any sense, so no. Conclusion: the relation between Course and Student is an association and not an aggregation (and therefore definitely no composition).

##### Example 2.

In the same educational system we might have a class Student and Class. Is a Class made up out of Students? Well actually yes. A class is a collection of students. The Student-Class association is therefore an aggregation.

##### Example 3.

In the previous example we established that Student-Class was an aggregation, but is it a composition? The Class is the parent and the Student the child (no students are not children, but the class Student is a child class, sigh). So would Students exist if there wouldn't be a Class. Well yes, if we delete the class the student would still exist. This means the Student-Class aggregation is not a composition.

##### Example 4.

If we would be modeling a house, we might have the classes House as parent, and Room as child. You can determine that this will be an aggregation since a House is made up out of rooms. Would the rooms still exist if the house was gone? Well no, no house = no rooms. This means the House-Room relation is a composition.

The UML reference manual defines an **aggregation** as “a form of association that specifies a whole-part relationship between an aggregate (a whole) and a constituent part” (Rumbaugh, Jacobson, & Booch, 1999, p. 146). However, the UML specification calls this a ‘shared aggregation’ and mentions “precise semantics of shared aggregation varies by application area and modeler” (Object Management Group, 2017, p. 112). Snoeck (2019) even argues that this is just ‘syntactic sugar’ and that the symbol could even be removed from the UML specification.

The **composition** is defined in the reference manual as: “a form of aggregation association with strong ownership and coincident lifetime by the whole” (Rumbaugh, Jacobson, & Booch, 1999, p. 226). Also here the UML specification is slightly different. The term used is a ‘composite aggregation’ and mentions “the composite object has responsibility for the existence and storage of the composed objects” (Object Management Group, 2017, p. 112).

The multiplicity for the aggregation can be anything (0..\*), but for a composite it is either 0..1 or simply 1.

The symbol for an aggregate is an open diamond (Figure 29), while for a composite this is a filled diamond (Figure 30).

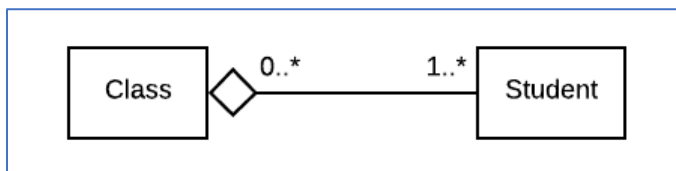


Figure 29. Example aggregation

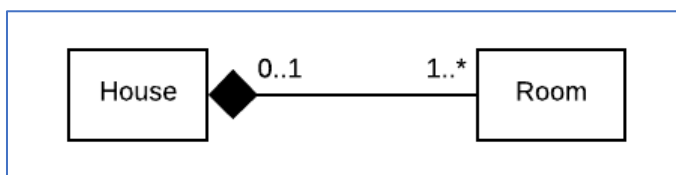


Figure 30. Example composition

Note that a ‘part’ in a composite can only belong to at most one ‘whole’, while in an aggregation the parts can belong to multiple wholes, as seen in Figure 31.

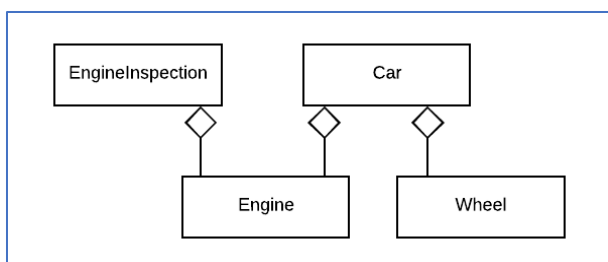


Figure 31. Aggregation with multiple parents

Note that ‘Engine’ is both a part of the Car as well as an EngineInspection. This would not be allowed with a composite relation.

## 6.5 Generalization in Class Diagrams

The last type of relation between two classes discussed in this module is the generalization. We have already discussed generalizations in Use Case diagrams (see 4.2), but they can also be applied in Class Diagrams.

In the UseCase diagram we used a definition specifically for those. A more generic definition can be found in the UML Reference Manual: “a taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element and contains additional information” (Rumbaugh, Jacobson, & Booch, 1999, p. 287).

Note that ‘generalization/specialisation’ relates to the object-oriented idea of **inheritance**. The specialisation element ‘inherits’ the attributes and methods of the generalization. It is also similar to the idea of supersets and subsets.

The notation is the same as in the UseCase diagram: an arrow with an open triangle as arrowhead. In case of multiple specialisations, there are two options to draw the inheritances (see Figure 32).

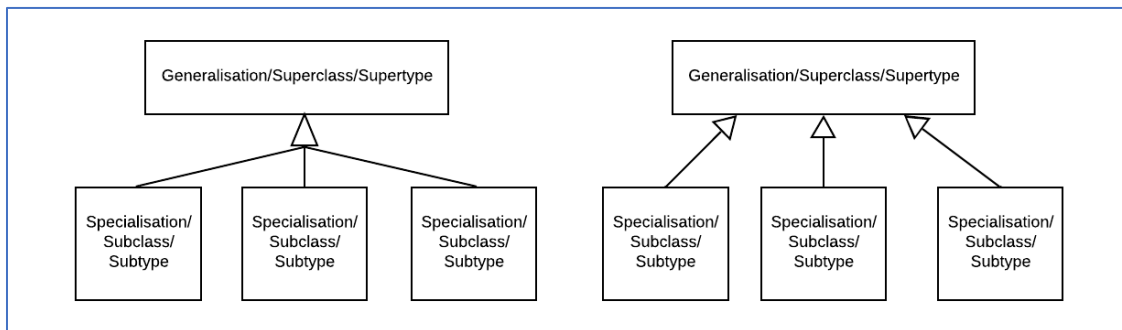


Figure 32. Two notations for generalisation

An example of the use of inheritance is shown in Figure 33.

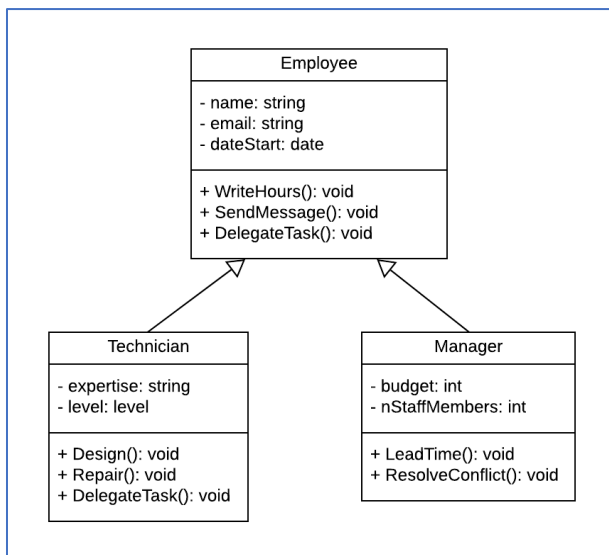


Figure 33. Example of inheritance

The Technician and Manager inherit all the attributes and methods of Employee (name, email, dateStart, WriteHours, SendMessage, and DelegateTask). Both the Technician and the Manager then add a few different attributes and methods. Note that both the generalisation Employee and the specialisation Technician have a method ‘DelegateTask’. This means that this method is overwritten

by a Technician. Technicians apparently need a different implementation of the DelegateTask method.

The benefits of inheritance are:

1. Code is easier to manage  
if you change something in the superclass, it applies immediately to all subclasses
2. Code is easier  
if you only need an Employee use them. Only if you need a specific type of employee do you use one of the subclasses.
3. Code is easily expandable.  
A new type of worker is easily added



## 7 Sequence Diagram<sup>6</sup>

UML Sequence Diagrams are interaction diagrams that detail how operations are carried out. They capture the interaction between objects in the context of a collaboration. As can be seen from the Figure 34, the Sequence diagrams belong to the behavioural diagrams.

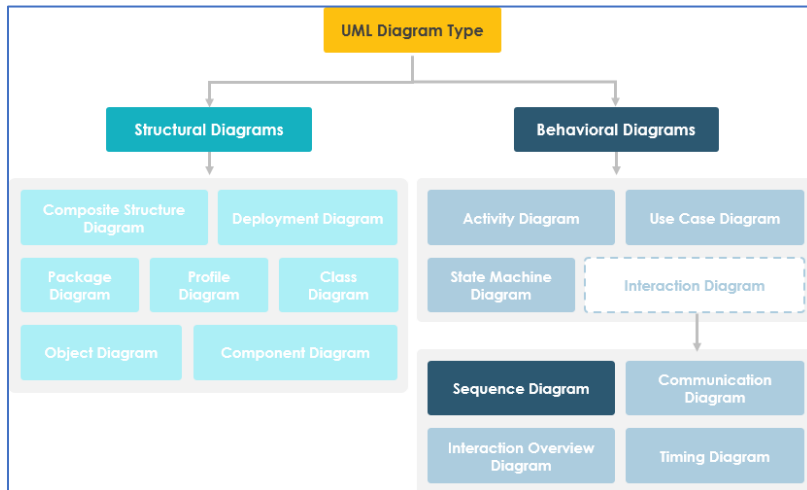


Figure 34. UML diagrams overview. Reprinted from Visual Paradigm with permission.

Sequence Diagrams are time focus and they show the order of the interaction visually by using the vertical axis of the diagram to represent time what messages are sent and when. Note that with 'time' is not meant 'duration' but more the timing of which message is sent when (the order of them).

Sequence Diagrams captures:

- the interaction that takes place in a collaboration that either realizes a use case or an operation (instance diagrams or generic diagrams)
- high-level interactions between user of the system and the system, between the system and other systems, or between subsystems (sometimes known as system sequence diagrams)

Purpose of Sequence Diagram

- Model high-level interaction between active objects in a system
- Model the interaction between object instances within a collaboration that realizes a use case
- Model the interaction between objects within a collaboration that realizes an operation
- Either model generic interactions (showing all possible paths through the interaction) or specific instances of an interaction (showing just one path through the interaction)
- To support use cases by specifying the detailed interaction between them

**Sequence diagrams** are defined in the UML specification as “describes an Interaction by focusing on the sequence of Messages that are exchanged, along with their corresponding OccurrenceSpecifications on the Lifelines” (Object Management Group, 2017, p. 595). In this definition the term ‘message’ is mentioned, which is defined as “a communication in which a sender makes a request for either an Operation call or Signal reception by a receiver” (Object Management Group, 2017, p. 292).

<sup>6</sup> The introduction of this chapter was taken from Visual Paradigm website: <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-sequence-diagram/>. Their text was copied and some parts adjusted or extended with their permission.

## 7.1 Sequence diagram components

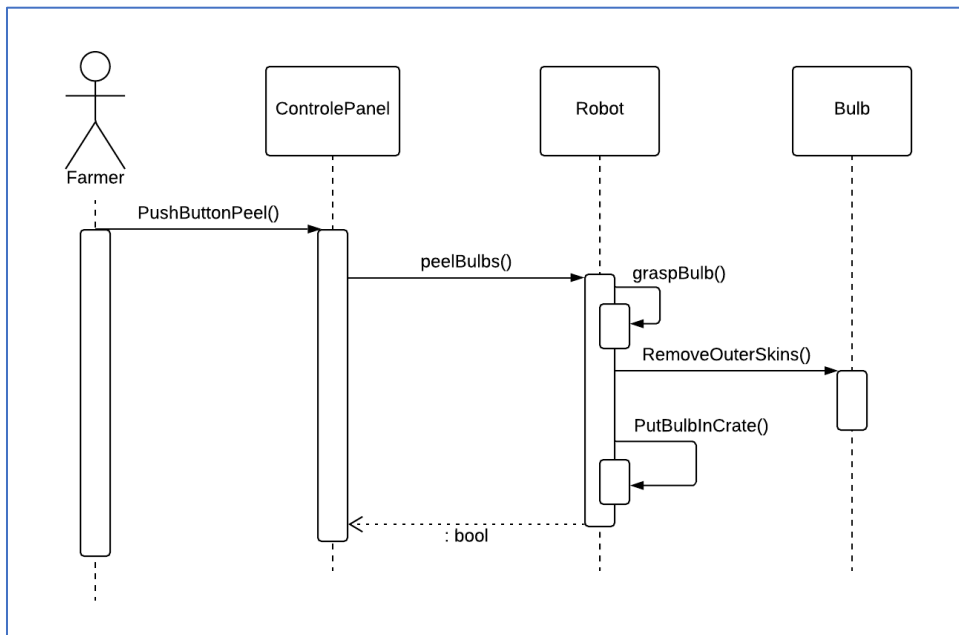


Figure 35. Sequence diagram basic example

On top you see an Actor and three so-called **objects**: ControlePanel, Robot, and Bulb. Out of each of these a dotted line is drawn. These are known as **(object) lifelines** and defined as: “the lifetime of the participant” (Object Management Group, 2017, p. 572). Covering these lifelines are some small thin vertical rectangles, known as the **ExecutionSpecification**, or sometimes Activation Box, or **Method Lifeline**. They represent the duration the participant is active.

To move from one object to another, so-called **messages** can be used. There are a few different types of messages possible: asynchronous, synchronous, reply, object creation, lost, and found. In this course we will only focus on synchronous, and reply messages.

A **synchronous message** is a message that will wait until it is done, before moving on. These are represented in the diagram by an arrow with a filled triangle arrowhead. A **reply message** is as the name implies a return of a previous message, drawn by a dotted line with an open arrowhead.

## 7.2 Loops in sequence diagrams

At the moment the application will result in just one bulb being grabbed by the robot and put in the crate. Probably the intention is to allow for more bulbs to be in one crate. To allow for this we can add a loop in the sequence diagram.

The **loop** is a specific so-called CombinedFragment in UML. The notation of a combined fragment is defined as “solid-outline rectangle. The operator is shown in a pentagon in the upper left corner of the rectangle” (Object Management Group, 2017, p. 588).

The guard of the loop can either be placed inside of this pentagon, but usually is placed nearby.

An example of a loop is added to our example in Figure 36.

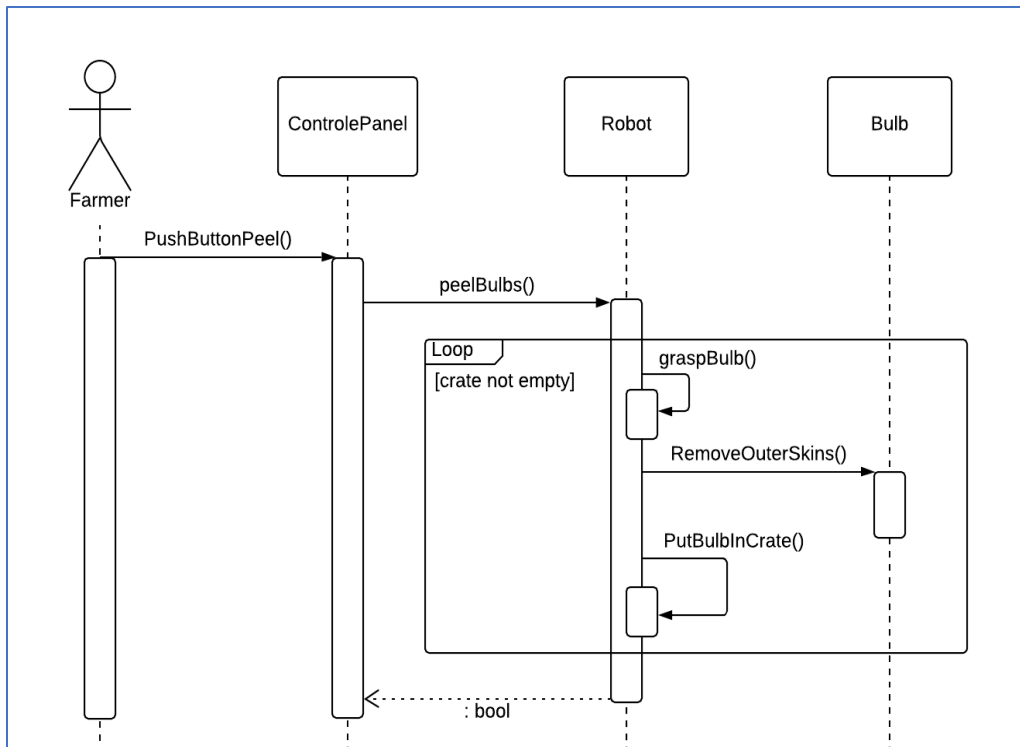


Figure 36. Sequence diagram with loop.

### 7.3 Sequence diagrams and Class diagrams

An additional benefit of a sequence diagram is that it might reveal methods we over looked in our Class Diagram. The messages that come into an object should become methods in the class that corresponds to that object.

In the example of .....this means that ControlePanel will have the method PushButtonPeel().

Robot will have the methods peelBulbs(), graspBulb(), and PutBulbInCrate().

Bulb will have a method RemoveOuterSkins().

A start for the Class diagram would then be something as shown in Figure 37.

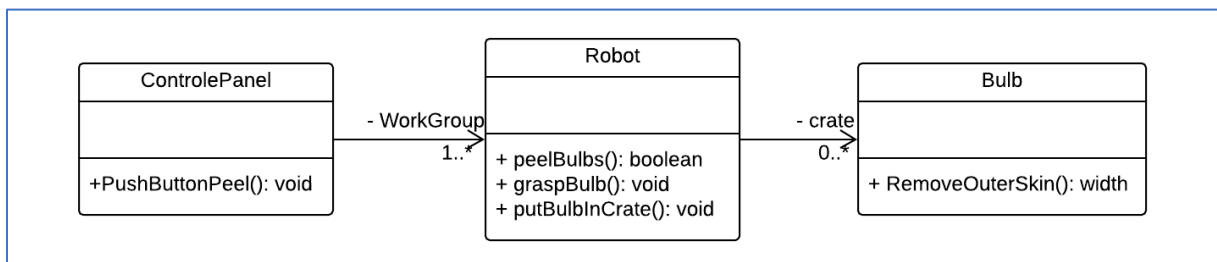


Figure 37. Class diagram methods from Sequence diagram

Note that since the peelBulbs() method eventually sends a return message in the form of a boolean, it's return value is also set to Boolean in the class diagram.

The example could of course be further expanded. At the moment the Farmer is never informed when the robot is done, and to peel a bulb the robot would first have to scan the dimensions of the bulb. These are left as exercise for the reader.

## 8 State Machine Diagram<sup>7</sup>

The behavior of an entity is not only a direct consequence of its inputs, but it also depends on its preceding state. The past history of an entity can best be modeled by a finite state machine diagram or traditionally called automata. UML State Machine Diagrams (or sometimes referred to as state diagram, state machine or state chart) show the different states of an entity. State machine diagrams can also show how an entity responds to various events by changing from one state to another. State machine diagram is a UML diagram used to model the dynamic nature of a system.

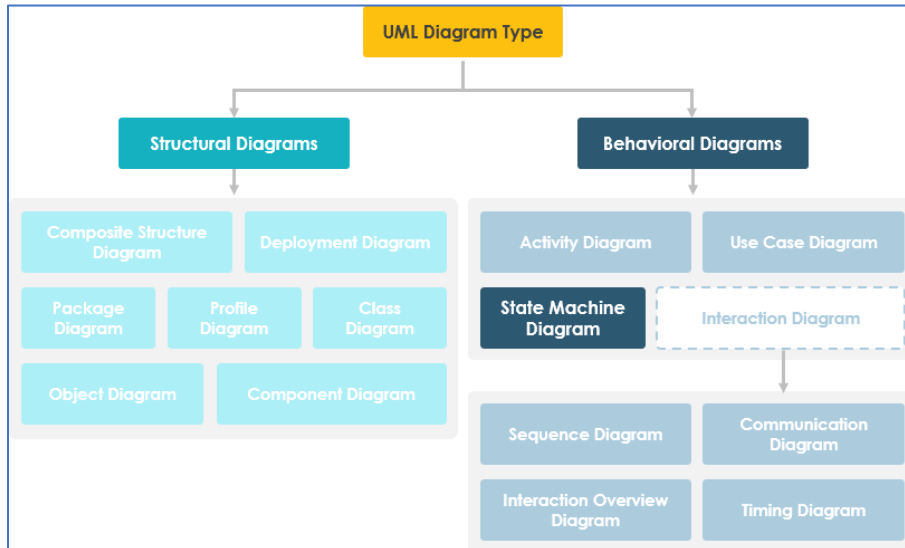


Figure 38. UML diagrams overview. Reprinted from Visual Paradigm with permission.

The State Machine Diagram, is actually an adaptation of a Herald State Chart (Harel, 1987), which is a special case of so-called State Diagrams, which have been around for many years (Shannon & Weaver, 1949).

The UML specification defines a **StateMachine diagram** as: “a graph that represents a StateMachine” (Object Management Group, 2017, p. 319), which is not very helpful. Luckily the UML Reference Manual defines also what a **StateMachine** is: “a specification of the sequences of states that an object or an interaction goes through in response to events during its life, together with its responsive actions” (Rumbaugh, Jacobson, & Booch, 1999, p. 439).

<sup>7</sup> The introduction of this chapter and section 8.1 were taken from Visual Paradigm website <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-state-machine-diagram/>. Their text was copied and some parts adjusted or extended with their permission. In section 8.2 the ‘characteristics’ parts were also taken from them.

## 8.1 Why State Machine Diagrams?

State machine diagrams typically are used to describe state-dependent behavior for an object. An object responds differently to the same event depending on what state it is in. State machine diagrams are usually applied to objects but can be applied to any element that has behavior to other entities such as: actors, use cases, methods, subsystems systems and etc. and they are typically used in conjunction with interaction diagrams (usually sequence diagrams).

For example:

Consider you have \$100,000 in a bank account. The behavior of the withdraw function would be:  $\text{balance} := \text{balance} - \text{withdrawAmount}$ ; provided that the balance after the withdrawal is not less than \$0; this is true regardless of how many times you have withdrawn money from the bank. In such situations, the withdrawals do not affect the abstraction of the attribute values, and hence the gross behavior of the object remains unchanged.

However, if the account balance would become negative after a withdrawal, the behavior of the withdraw function would be quite different. This is because the state of the bank account is changed from positive to negative; in technical jargon, a transition from the positive state to the negative state is fired.

The abstraction of the attribute value is a property of the system, rather than a globally applicable rule. For example, if the bank changes the business rule to allow the bank balance to be overdrawn by 2000 dollars, the state of the bank account will be redefined with condition that the balance after withdrawal must not be less than \$2000 in deficit.

Note That:

- A state machine diagram describes all events (and states and transitions for a single object)
- A sequence diagram describes the events for a single interaction across all objects involved

## 8.2 Basic Components of a State Machine Diagram

We begin with showing a basic example of a State Machine Diagram in Figure 39.

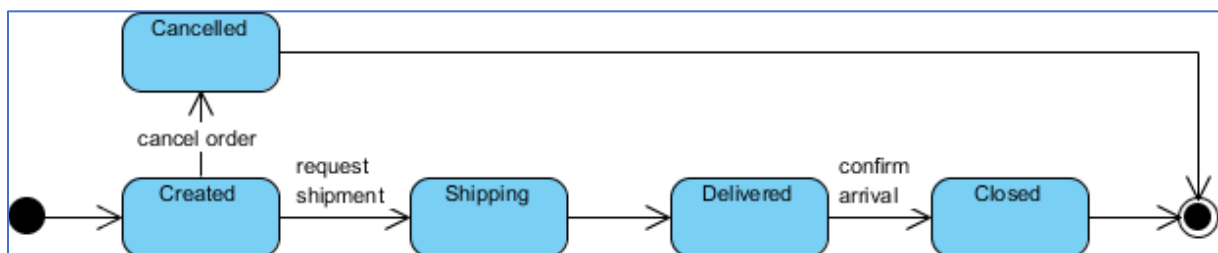


Figure 39. Example of a State Machine Diagram. Reprinted from Visual Paradigm with permission.

The diagram looks a bit similar to an Activity Diagram. However as you might notice the main nodes are not activities, the activity is actually indicated on the lines. In this sense, the State Machine Diagram is more an Activity on Arrow (AoA) diagram than an Activity on Node (AoN, or Precedence Diagram Method (PDM)), as the Activity Diagram is.

The State Machine Diagram consists for the most out of States. A **state** is a constraint or a situation in the life cycle of an object, in which a constraint holds, the object executes an activity or waits for an event. They are shown as “a rectangle with rounded corners, with the State name shown within” (Object Management Group, 2017, p. 319).

#### Characteristics of State

- State represent the conditions of objects at certain points in time.
- Objects (or Systems) can be viewed as moving from state to state
- A point in the lifecycle of a model element that satisfies some condition, where some particular action is being performed or where some event is waited

There are two ‘special’ states. The one at the beginning (which looks like the Initial Node from the Activity Diagram), and the one at the end (which looks like the Final Node from the Activity Diagram). In State Machine Diagrams these are called the initial state, and the final state.

#### Characteristics of Initial and Final States

- The **initial state** of a state machine diagram, known as an **initial pseudo-state**, is indicated with a solid circle. A transition from this state will show the first real state
- The **final state** of a state machine diagram is shown as concentric circles. An open loop state machine represents an object that may terminate before the system terminates, while a closed loop state machine diagram does not have a final state; if it is the case, then the object lives until the entire system terminates.

We can go from one state to another, using so-called transitions. On the line we write the event that might trigger the transition.

#### Characteristics of Transition

- Viewing a system as a set of states and transitions between states is very useful for describing complex behaviors
- Understanding state transitions is part of system analysis and design
- A **Transition** is the movement from one state to another state
- Transitions between states occur as follows:
  1. An element is in a source state
  2. An event occurs
  3. An action is performed
    - The element enters a target state
- Multiple transitions occur either when different events result in a state terminating or when there are guard conditions on the transitions
- A transition without an event and action is known as automatic transitions

### 8.3 Guards and Actions

In some situations, the transition from one state to another, is not because of an event but some condition that is met (or not). The UML specification therefore also allows for **guards** to be placed at transitions (Object Management Group, 2017, p. 315). The guard should be placed between [..].

Another symbol you might come across is an 'action'. **Action** is an executable atomic computation, which includes operation calls, the creation or destruction of another object, or the sending of a signal to an object. An action is associated with transitions and during which an action is not interruptible - e.g., entry, exit. An action is indicated using a /.

In Figure 40 another example of a State Machine Diagram is shown using these two elements.

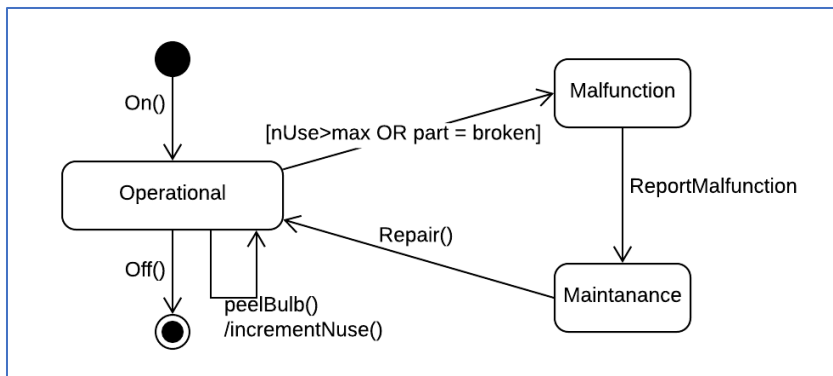


Figure 40. Example of State Machine Diagram with guard and action

The transition from 'Operational' to 'Malfunction' is not an event but done using a guard. The self-transition at Operational also has an action 'incrementNuse()'. This way, each time the robot is peeling a bulb, the nUse will be increased. Once the nUse reaches is above the maximum the robot will malfunction.

### 8.4 State Machine Diagram and Class Diagrams

The Sequence Diagrams were able to reveal some methods to be added to a Class Diagram. A State Machine Diagram also generates some methods and sometimes also some attributes.

In the example from Figure 40 the Robot Class will need to have five new methods: On(), ReportMalfunction(), Repair(), incrementNuse(), and Off(). The method 'peelBulbs()' was already added to the class in section 7.3. Also we will need a new attributes 'nUse' and 'part' because of the guard.

Our Robot Class could now look something as shown in Figure 41.

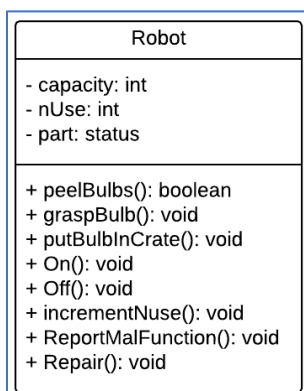


Figure 41. Class diagram of Robot updated.

## 9 Final comments

In this course we discussed a few diagrams that could be used when creating a system. An overview of who is involved and how is shown in Table 5.

Table 5  
*Who is involved and when*

Diagram/Description	Client	Analyst	Designer	Developer	Tester
Requirements	Create	Create and Update	Read	Read	Read
Process diagram	Create	Create and Update	Read		
UseCase diagram	Create	Create	Read and Update	Read	Read
UseCase description		Create	Read and Update	Read	Read
Activity diagram	Create	Create	Read and Update	Read	Read
Class diagram			Create and Update	Read	
Sequence diagram			Create and Update	Read	
State diagram			Create and Update	Read	

As can be seen from the table, in the first stage of the requirements only the Client and Analyst are involved. They create these together. The more 'technical' Designer, Developer and Tester will usually only read these.

Note that the Developer and Tester usually only read diagrams provided to them. The Client and Analyst will create the behavioural diagrams, while the Designer the so-called structural diagrams of UML.

We only discussed 5 out of the 14 UML diagrams, and from each diagram there is a lot more that could be involved. However you hopefully now have a basic foundation to explore these additional features and diagrams yourself.

It is also not necessary to make all diagrams for each project. If your system does not really use states, a state machine diagram might not be needed.



## References

- Booch, G. (1994). *Object-oriented analysis and design: With applications* (2nd ed.). Boston, MA: Addison-Wesley.
- Coad, P., & Yourdon, E. (1990). *Object-oriented analysis* (2nd ed.). Prentice Hall.
- Cockburn, A. (2000). *Writing effective use cases*. Upper Saddle River, NJ: Pearson.
- Colburn, T., & Shute, G. (2007). Abstraction in computer science. *Minds and Machines*, 17(2), 169-184. doi:10.1007/s11023-007-9061-7
- Dori, D. (2002). *Object-process methodology: A holistic systems paradigm*. Berlin: Springer-Verlag.
- Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3), 231-274. doi:10.1016/0167-6423(87)90035-9
- Hoogendoorn, S. (2004). *Pragmatisch modelleren met UML*. Culemborg, NL: Van Duuren Media B.V.
- Jacobson, I. (1992). *Object-oriented software engineering: A use case driven approach*. Addison-Wesley.
- Kramer, J. (2007). Is abstraction the key to computing? *Communications of the ACM*, 50(4), 36-42. doi:10.1145/1232743.1232745
- Martin, J., & Odell, J. J. (1992). *Object-oriented analysis & design*. Prentice Hall.
- McConnell, S. (2004). *Code complete: A practical handbook of software construction* (2nd ed.). Redmond: Microsoft Press.
- Mo, J. P., Bil, C., & Sinha, A. (2015). *Engineering systems acquisition and support*. Cambridge, UK: Woodhead.
- Object Management Group. (2003). *UML 1.5 Specification*.
- Object Management Group. (2005). *UML 2.0 Superstructure*.
- Object Management Group. (2017). *UML 2.5.1 Specification*. Retrieved from <https://www.omg.org/spec/UML>
- Page-Jones, M. (2000). *Fundamentals of object-oriented design in UML*. Boston: Addison-Wesley.
- Royce, W. W. (1970). Managing the development of large software systems. *IEEE WESCON 26* (pp. 328-338). Los Angeles: 1-9.
- Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W. (1991). *Object-oriented modeling and design*. Englewood Cliffs, NJ: Prentice-Hall.
- Rumbaugh, J., Jacobson, I., & Booch, G. (1999). *The Unified Modeling Language reference manual*. Upper-Saddle River, NJ: Pearson.
- Schneider, G., & Winters, J. P. (1998). *Applying use cases: A practical guide*. Boston, MA: Addison-Wesley.
- Shannon, C. E., & Weaver, W. (1949). *The mathematical theory of communication*. Urbana, OH: University of Illinois Press.







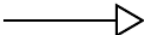
- Shlaer, S., & Mellor, S. J. (1988). *Object-oriented systems analysis: Modeling the world in data*. Upper Saddle River, NJ: Prentice-Hall.
- Snoeck, M. (2019). UML class diagrams for software engineering: Aggregation.
- Störrle, H., & Hausmann, J. H. (2005). Towards a formal semantics of UML 2.0 activities. *Software Engineering*, (pp. 1-13). Essen.
- The American Presidency Project. (n.d.). *Remarks at the National Defense Executive Reserve Conference*. Retrieved from <https://www.presidency.ucsb.edu/node/233951>
- uml-diagrams.org. (n.d.). *UML Association*. Retrieved from <https://www.uml-diagrams.org/association.html>
- Visual Paradigm. (n.d.). *What is Class Diagram?* Retrieved from <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-class-diagram>
- Wikipedia. (n.d.). *Abstraction (computer science)*. Retrieved October 24, 2019, from [https://en.wikipedia.org/wiki/Abstraction\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Abstraction_(computer_science))
- Wirfs-Brock, R., & Wilkerson, B. (1990). *Designing object-oriented software*. Prentice Hall.

## Appendix 1. Dictionaries for discussed diagrams

### A1.1. UseCase Diagram

Table 6









Dictionary for UseCase diagrams

Symbol (word)	(English) Name	Description
	Actor	An actor specifies the role of a user, or other system, that has an interaction with the subject.
	System boundary	A collection of UseCases that all relate to the same subject
	UseCase	A logical description of part of the system
	Include	A directed association between two UseCases. The behavior of one UseCase is implemented in the behavior of the other. It indicates the UseCas HAS to be used.
	Extend	A directed association between two UseCases. The behavior of one UseCase can be implemented in the other. It indicates the UseCase COULD be used.
	Undirected association	From actor to (primary) UseCase
	Generalisation / Specialisation	The generaliation inherits the properties of the specialisations that are connected to it.

### A1.2. Activity Diagram

Table 7

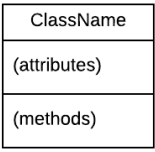
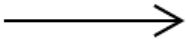
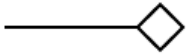
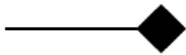
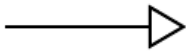
Dictionary for Activity Diagrams

Symbol (word)	(English) Name	Description
	Initial node	a ControlNode that acts as a starting point for executing an Activity
	Final node	a ControlNode at which a flow in an Activity stops. A FinalNode shall not have outgoing ActivityEdges
	Activity node	used to model the individual steps in the behavior specified by an Activity
	Activity Edge	a directed connection between two ActivityNodes along which tokens may flow, from the source ActivityNode to the target ActivityNode
	Decision node	a ControlNode that chooses between outgoing flows
	Merge Node	a control node that brings together multiple flows without synchronization
	Fork Node	a ControlNode that splits a flow into multiple concurrent flows
	Join Node	a ControlNode that synchronizes multiple flows

### A1.3. Class Diagram

Table 8

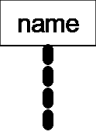



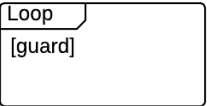
Dictionary for Class Diagram

Symbol (word)	(English) Name	Description
	Class node	the descriptor for a set of objects that share the same attributes, operations, methods, relationships, and behavior. A class represents a concept within the system being modeled
	(directed) association	an association between two classes
	Aggregation	a form of association that specifies a whole-part relationship between an aggregate (a whole) and a constituent part
	Composition	a form of aggregation association with strong ownership and coincident lifetime by the whole
	Generalization/Specialization	a taxonomic relationship between a more general element and a more specific element.

### A1.4. Sequence diagram

Table 9

Dictionary for Sequence Diagram

Symbol (word)	(English) Name	Description
	Lifeline (a.k.a. object lifeline)	represents the lifetime of the participant
	ExecutionSpecification (Activation box / method lifeline)	represents, on a lifeline, the duration the participant is active
	Synchronous message (call message)	a message that will wait until it is done, before moving on
	Reply message	
	Loop	Messages inside the loop frame will continue (loop) until a condition (guard) is reached.

## A1.5. State Machine diagram

Table 10

Dictionary for State Machine Diagram

Symbol (word)	(English) Name	Description
●	Initial State (initial-pseudo-state)	transition from this state will show the first real state
⊙	Final State	the last state an object can be in.
→	Transition	the movement from one state to another state
/.....	Action	an executable atomic computation, which includes operation calls, the creation or destruction of another object, or the sending of a signal to an object
.....()	Event	Represents incidents that cause objects to transition from one state to another
[.....]	Guard	

## Index

### **abstraction, 7**

- action, 47
- Activation Box, 42
- Activity Diagram, 27
- Activity Edge, 30
- Activity node, 29
- Actor, 17
  - Primary actor, 17
  - Secondary actor, 17
- aggregation, 37
- association, 35
- Association, 18
- business processes, 9
- Class Diagram, 33
- Class Navigation, 36
- Class Node, 34
- CombinedFragment, 42
- composition, 37
- Decision node, 30
- elementary business processes, 9
- event, 46
- ExecutionSpecification, 42
- Extend, 19
- Final node, 29
- final state, 46
- Fork Node, 31
- formal model, 7
- generalization
  - in class diagrams, 39
  - in UseCase, 21
- guards, 30

- Herald State Chart, 44
- Include, 20
- Initial node, 29, 51
- initial pseudo-state, 46
- initial state, 46
- Join Node, 31
- lifeline
  - method lifeline, 42
  - object lifeline, 42
- loop
  - in sequence diagram, 42
- Merge Node, 30
- model, 7**
- Modelling Software, 8
- multiplicity, 35
- objects, 42
- Process diagram, 9
- reply message, 42
- Sequence Diagram, 41
- state, 46
- State Machine Diagram, 44
- StateMachine, 44
- synchronous message, 42
- system boundary, 18
- systems modelling, 7**
- the hand method, 22
- transition, 46
- Unified Modeling Language, 11
- Use Case, 17
- Use Case Diagram, 15
- UseCase description, 25

