King Saud University
College of Computer and Information Sciences
Information Technology Department

System Analysis and Design Guide

Graduation Project 1

IT 496

Prepared by

Hend Alrasheed

halrasheed@ksu.edu.sa

Aug. 2019

This document is directed to the IT 496 students in the Department of Information Technology, King Saud University. The goal is to clarify the technical details required in their project report. Assuming you have a clear project idea and a solid list of requirements, this document focuses on the System Analysis and Design chapter of the graduation report (Chapter 4 of the Project-1 Guide).

This document does not intend to repeat basic Software Engineering concepts and techniques (as introduced in the Software Engineering course IT320). However, this document sheds some light on some of the major and common questions that students may face during their graduation projects.

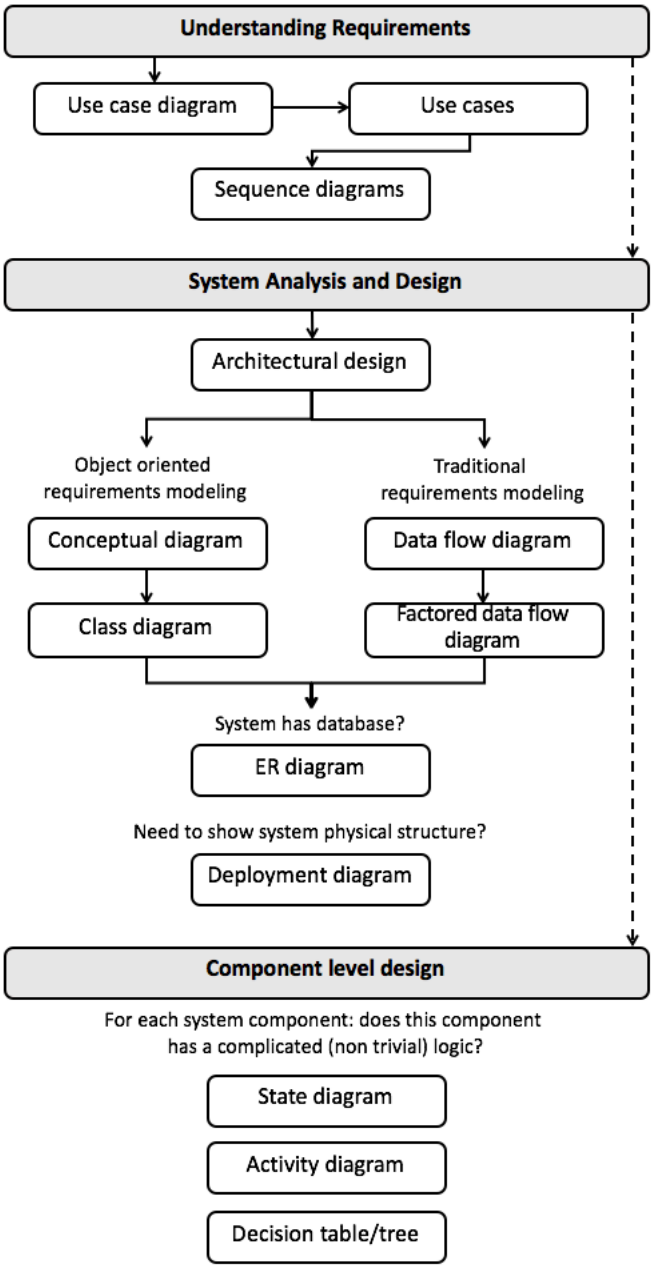
Table of Contents

Overview.

Generally, the first part of your graduation project (GP1) will include three major activities:

- Understanding the software requirements.
- System analysis and design.
- Component level design.

Each of the activities listed above has its own set of deliverables. The figure below shows the most common types of deliverables during each activity. Note that you may not need all diagrams. Also, you may need some diagrams that are not listed in the diagram below. Details of each diagram will be provided in the rest of this document.



Project scope.

The project scope is the part of the project in which the team determines the general goals and deliverables of the project. Here you also mention what will not be included in the project.

When you write your scope statement, make it SMART (Specific, Measurable, Agreed upon, Realistic, and Time bound).

Example:
This project will consist of creating a software system that …… The main objective of this software is to …….. The following functions will be out of the scope of this project …….. The project will be completed on ……

Software requirements.

The software requirements list is a detailed description of the software to be developed. Here you tell the customers *what* can your system do and *who* will use it. Note that the system requirements do not provide any details on *how* those requirements work. Requirements can be functional and non-functional.

Functional requirements: a detailed list of the system features and functions. Those are the functions you will later implement. For each function, specify the input, the output, and the type of user. For each functional requirement, you will write a use-case (explained below) to explain it in more details. It is a good idea to number your requirements so it is easier to refer to them.

Non-functional requirements: a description of the general characteristics (or qualities) of the system such as performance, scalability, security, and maintainability.

Use-case diagram.

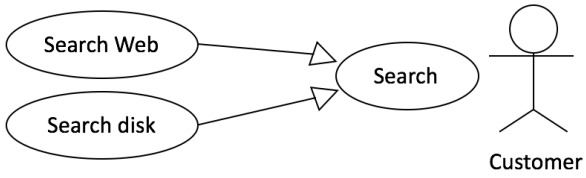
The use-case diagram shows the interaction between the system and the external entities (called actors). The actors may be users or other systems. In a use-case diagram, a (human) user (actor) is usually represented with a stick figure with the name of the actor below the figure. Other systems (that are external and interact directly with the system) are represented with a box with the word “actor”.

In some special cases, Time can be an actor. For example, when the system is automated and triggered by time (this includes events that occur every month, every day, or every minute).

The system is represented as a set of use-cases. Each use-case represents a single meaningful function that is observable to someone or something outside the system. In a use-case diagram, a use-case is denoted by an ellipse.

A generalization/specialization relationship (also called inheritance relationship) may exist between actors when one actor (the descendant actor) inherits the properties of another actor (the ancestor actor). This relationship implies that the descendant actor can use all the use-cases defined for its ancestor actor.

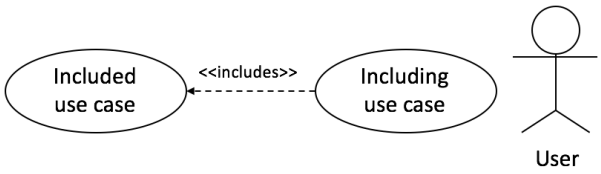
Similarly, a generalization/specialization relationship may exist between use-cases when there is a basic flow of events that is followed by multiple special considerations. We can think of the super use-case as a generalization of the sub use-cases. Look at the example below.



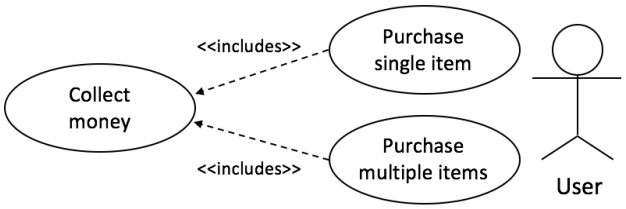
The sub use-case (or cases) in the specialization/generalization relationship carries the underlying business process meaning.

Two types of advanced relationships may exist between use-cases: the “Include” relationship and the “Extend” relationship.

The Include relationship: this relationship is used when a use-case contains the processes of another use-case *as part* of its normal flow of actions.

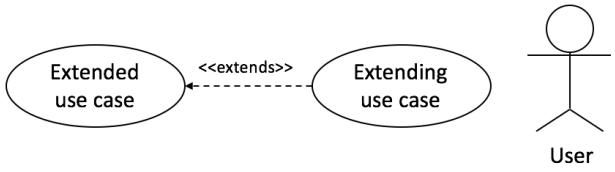


We assume that any included use-case will be always called as part of the including use-case. A use-case may be included by one or more use-cases. See the example below.

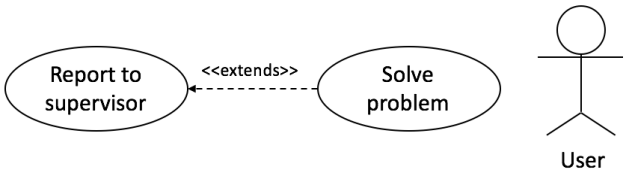


The main goal of using the Include relationship is to avoid repetition.

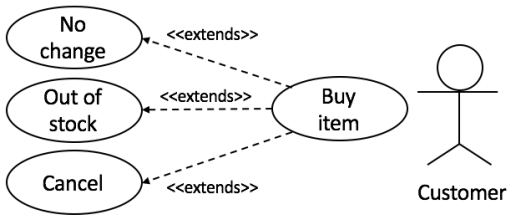
The Extend relationship: this relationship is used when a use-case augments the behavior of another use-case. That is, when the functionality of the original use-case need to be extended as a result of some exceptional circumstance (the original use-case can be executed without the extended use-case).



For example, the use-case Solve Problem below is complete by itself, but can be extended by the use-case Report to Supervisor in a specific scenario in which the user wants to report the problem to the supervisor.



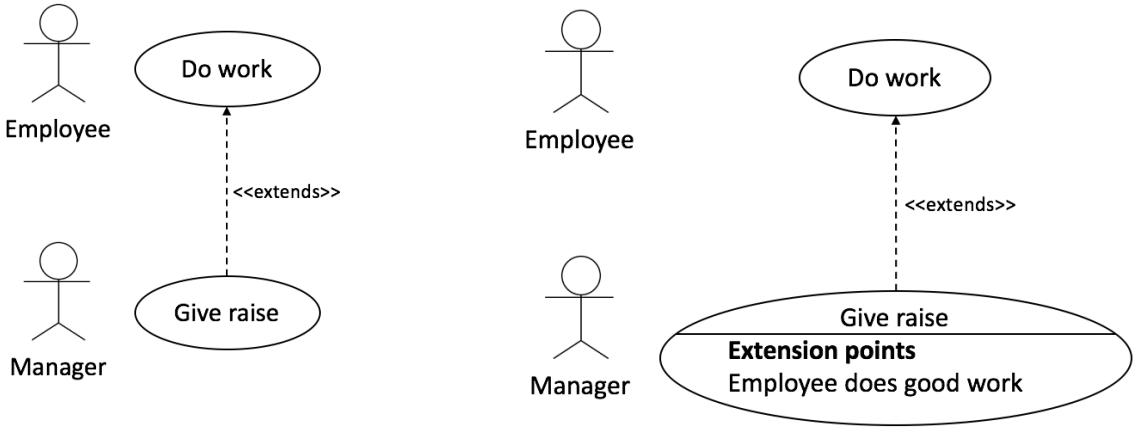
Multiple use-cases can extend a single use-case (look at the example below).



The main goal of using the Extend relationship is to reduce the complexity of a large use-case (by dividing it).

The *Generalization* and the *Extend* relationships are similar and you can think of the Extend relationship as a type of the Generalization relationship. The main difference between them is that in the Generalization relationship, the sub use-cases are expected to redefine the behavior of the super use-case. That is, the behaviors of the super and the sub use-cases can be very different. However, in the Extend relationship, the extending use-case will be executed after the execution of the extended use-case reaches a certain point (a condition). Therefore, the Extend relationship uses what we call an Extension Point.

An extension point show when the extending use-case will be performed. Take a look at the example below. In (a), it is hard to tell when the Give raise use-case will be executed. This problem was resolved using the extension point as in (b).



Use-cases.

A use-case is a template that describes one of the processes of the system from the user's perspective. The use-case describes the process from start to finish as sequence of events and actions that are required to complete the process.
You need to create a use-case for each process in the use-case diagram.

Most use-cases contain the following sections: use-case name, actor, purpose, overview, cross references, constraints, pre-conditions, post-conditions, type, priority, frequency of use, and typical course of actions. Next, a brief description of each section is provided.

- *Use-case name*: the use-case name must be meaningful and unique. For example, Log In, Rate Product, and Buy Items.

- *Actor*: the type of users who can initiate this use-case. A use-case may have a single actor or multiple actors. If a use-case has multiple actors, it is a good idea to decide who is the initiator of the use-case. For example, the Buy Items use-case has two actors: customer (initiator) and cashier.

- *Purpose*: the objective of the use-case. Technical and implementation details should not be mentioned as a part of the use-case purpose.

- *Overview*: what type of actions take place during this use-case? It is a good idea to start the use-case overview with this sentence: “This use-case begins when …..” and ends it with “This use-case ends when …..” or “On completion, …..”.

- *Cross references*: the number of the functional requirement that can be linked to this use-case.

- *Pre-conditions*: the conditions that need to be met before the use-case can begin.

- *Post-conditions*: the changes that occur after the execution of the use-case such as saving information or generating output.

- *Type*: there are two ways to describe the type of a use-case:

  (1) According to *usage* and
  (2) According to *abstraction level*.

  According to usage, use-case type can be *primary*, *secondary*, or *optional*.
  - Primary: when the use-case describes a major and common function.
  - Secondary: when the use-case describes an exceptional function (a function that is rare or unusual).
  - *Optional*: when the use-case describes a function that may or may not be used (may or may not be implemented).

  According to abstraction level, a use-case type can be *essential* or *real*.
  - Essential: when the description of the use-case is written using a high-level language (free of technology and implementation details). Most use-cases will be essential because they are created during the analysis phase.
  - Real: when the description of the use-case involves a detailed design description.

  The following is an example of the language used in an essential and a real use-cases.

Essential use-case	Real use-case
The account holder identifies herself to the ATM.	The account holder inserts the card into the ATM card reader. The account holder is prompted to enter her PIN which she input using a numeric keypad.

- *Typical course of events*:
  The typical course of events is the formal description of the flow of events that occur during the execution of the use-case. It tells us step by step how the system reacts to every user action (keep in mind that the system may not respond to every user action or it may require multiple user actions to respond).
  You can think of it as a textual representation of the corresponding sequence diagram.
  It is called "typical" course of events because it describes how most users interact with the system. It includes "Alternatives" section that handles all the other cases in which an event may happen differently.
  The table below shows how the typical course of events looks like for the Buy Items use-case.

Actor action	System response
1. This use-case begins when a customer arrives at the checkout point.	
2. The cashier scans each item.	3. Determines the item price and adds its information to the current transaction.
4. The cashier indicates that the item entry is complete.	5. Calculates and displays the total.
6. The cashier tells the customer the total.	
7. The customer gives a cash payment. The cashier records the cash received.	8. Shows the balance due back to the customer.
9. The cashier deposits the cash and extracts the balance owing. 11. The cashier gives the balance owing to the customer.	10. Logs the completed sale. Print the receipt.
12. The cashier gives the receipt to the customer. 13. The customer leaves with the items.	
Alternatives: Line 9: insufficient cash in drawer to pay balance. Ask for cash from supervisor.	

In the use-case above, it is expected that the customer will always choose "pay by cash" as her payment method (only one method of payment is provided). If the system has two methods of payment (cash and credit card), then list the typical (most common) one in the table and the other in the Alternatives section. However, if the two methods are equal in their likelihood, then you need to add a new table for each option (look at the example below).

Section: Main

Actor action	System response
1. This use-case begins when a customer arrives at the checkout point.	
2. The cashier scans each item.	3. Determines the item price and adds its information to the current transaction.
4. The cashier indicates that the item entry is complete.	5. Calculates and displays the total.
6. The cashier tells the customer the total.	
7. Customer chooses payment method: If customer chooses cash, see section *pay by cash*. If customer chooses credit card, see section *pay by credit card*.	
	8. Logs the completed sale. Print the receipt.
9. The cashier gives the receipt to the customer. 10. The customer leaves with the items.	
Alternatives:	

Section: Pay by cash

Actor action	System response
1. The customer gives a cash payment. 2. The cashier records the cash received.	
	3. Shows the balance due back to the customer.
4. The cashier deposits the cash and extracts the balance owing. 5. The cashier gives the balance owing to the customer.	
Alternatives: Line 4: insufficient cash in drawer to pay balance. Ask for cash from supervisor.	

Similarly, add a Pay by credit card course of events table.

Traditional or Object-Oriented requirements modeling?

The choice of the software requirements modeling determines your overall view of the systems and the UML diagrams you will need to provide as part of your project.

The choice of the software requirements modeling is determined by the nature of your software project. However, some projects can be developed using any approach. Generally, we use the following rule to decide:

When the basic building blocks of our project is functions, then we use the traditional approach. When the basic building blocks of our project is entities (data), then we use the object-oriented approach.

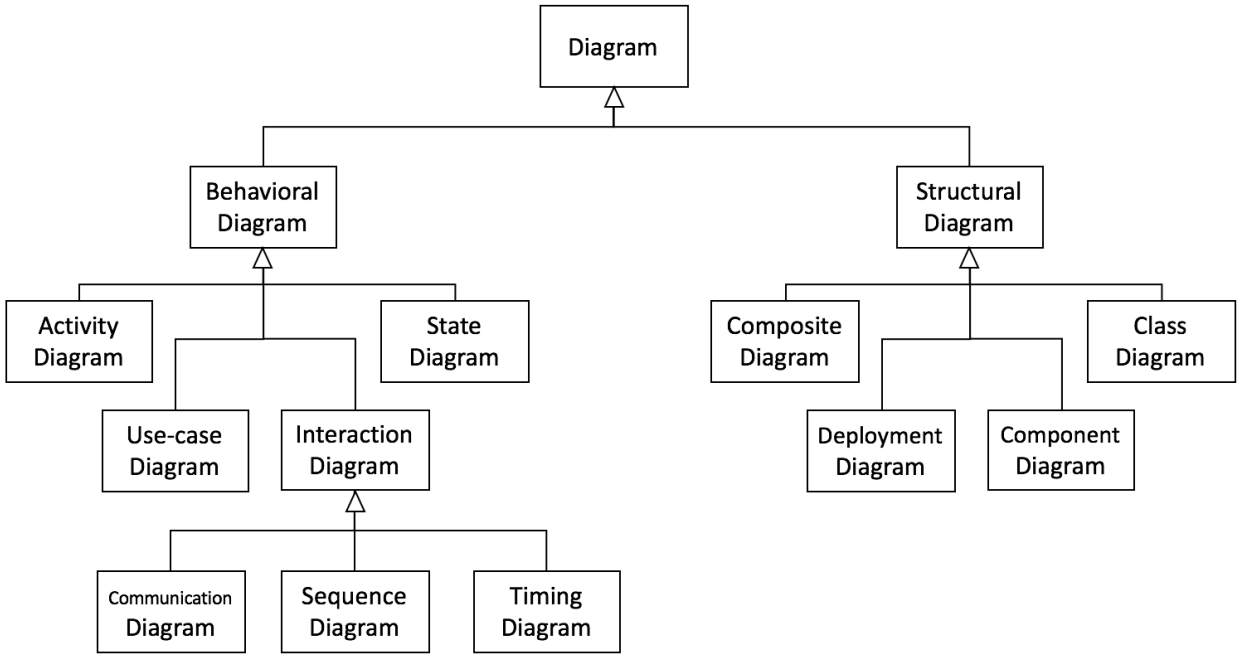
What do we mean by a building block? Think of it as the basic set of elements (or abstractions) of your project. For example, in a banking software, the basic building blocks are the client, the account, the transaction, etc. Those are all entities, and each entity includes sets of attributes and methods. Generally, using an object-oriented approach works well for a banking software. In a document editing software, the basic building blocks are functions such as add document, edit document, add text, change text color, etc. This makes using the traditional approach makes more sense in this case.
Note that it is possible to use any approach; however, there is always one approach that is more efficient (leads to a stronger design).

If you chose the traditional approach, then your components are functions and you need to do a data flow diagram.
If you chose the object-oriented approach, then your components are entities and you need to do a class diagram.

UML design diagrams.

The rest of this document presents a several popular UML diagrams that developers usually create as part of the software development process. Generally, those diagrams can be partitioned into two groups: behavioral and structural (see the figure below).

Behavioral diagrams focus on the data flow within the system (or within a specific part of the system). Structural diagrams show the static (stable) representation of the system.



Architectural Design.

The architectural design shows the structure of the system. That is, the main system components and their relationships. Use a “box and line” diagram representation to show your design. Do not provide any component details at this level (this is not a class diagram).

To come up with your architectural design, you need to figure out the following: the main components of your system and the best organization of those component (how those components are connected).

Remember to use one of the existing architectural patterns if one is similar to your system. Some of the most common software architectural patterns are:

1. The layered pattern.
2. The client-server pattern.
3. The pipe and filter pattern.
4. The repository pattern.
5. The call and return pattern.

Component level design

A component is a class, a module, or a set of interrelated classes or modules that collaborate to achieve a common goal.
Component level design follows the architectural design step. The goal is to elaborate on each of the elements in your architectural diagram by providing details that lead to the actual code.
Component level design can be achieved by using a pseudo-code, a DFD, a state diagram, a collaboration diagram, a decision table, etc.
Remember to follow good component level design principles (such as cohesion and coupling). This will help you create a design that is easier to deal with (test, maintain, etc).

Data Flow Diagrams (DFD).

If you decided to use a traditional software requirements modeling, then you know that the *functions* are the basic building blocks of your system. Therefore, you have to create a detailed list of all the functions (methods) that you are going to implement during the implementation phase. DFDs help you create this list.
A DFD is a top down method that is used as a traditional visual representation of the information flow within a given system. It supports the hierarchical decomposition of the system into its different functions and activities. You will always need multiple DFDs each of which describes the system at a different level of abstraction. The number of DFDs cannot be known in advanced.
Generally, DFDs show the following information:

- Data that enters and leaves the system (system input and output).
- External entities (people or other systems) that interact with your system. This will help you decide the boundary of your system, that is, what is and what is not included in your system.
- The processes that take place within the system.
- The information that is stored in the system.

The following provides a brief description of the first four DFD levels:

- Level 0 DFD (also called the *Context Level* DFD) considers the whole system as a single process that interacts with a set of external entities. The entire system will be encapsulated into one bubble.
- Level 1 DFD presents a more detailed view of the system by showing its main sub processes. The system bubble shown in level 0 DFD will be replaced by multiple bubbles each of which describes (high level) a main system function.
- Level 2 DFD is used to provide more specific details about each of the system functions.
- Level *n* DFD (*n* >= 3) decomposes each bubble (function) that exists in level *n*-1 DFD into basic functional units. The decomposition process finishes when each of the bubbles in the last DFD represents a functional primitive.

The final DFD to be *factored*. Factoring "leads to a program structure in which top-level components perform decision making and low-level components perform most input, computation, and output work. Middle level components perform some control and do moderate amounts of work."

Conceptual Model.

The conceptual model is a system representation that describes the system at a high level of abstraction. It mainly describes

- The main system entities and relationships.
- System states and transitions.
- System interactions and user interfaces.

The main objective of the conceptual model is to enable effective communication between the software team and the stakeholders.

There is no one formal notation for the conceptual diagram. Generally, class diagram notations are used to represent the conceptual diagram. A conceptual model can be used as a starting point for building the class diagram of your system. Therefore, when you start drawing your conceptual diagram, draw a diagram that represents the concepts in the system domain (not your specific system).

Class Diagram.

A class diagram describes the structure of the system. It shows the system classes, attributes, methods, and relationships among the classes. Class diagrams serve two main objectives:

1. System visualization and description.
2. code construction (during implementation).

When drawing your class diagram, keep this question in mind: What type of relationships exist between the classes in real-life? Those relationships must help you draw a class diagram that is easy to understand by humans. It must also help programmers write their code clearly.

ER Diagram.

The ERD (Entity-Relationship diagram or Entity-Relationship model) represents the entities that exist in the system and their relationships the way they will be stored in the system database.

You only need an ERD if your system has a database (most likely your system will need a database).

When drawing your ERD, keep this question in mind: What type of relationships you need to add between your entities to answer your database queries?

ER diagrams may represent system relationships that are more difficult for humans to understand.

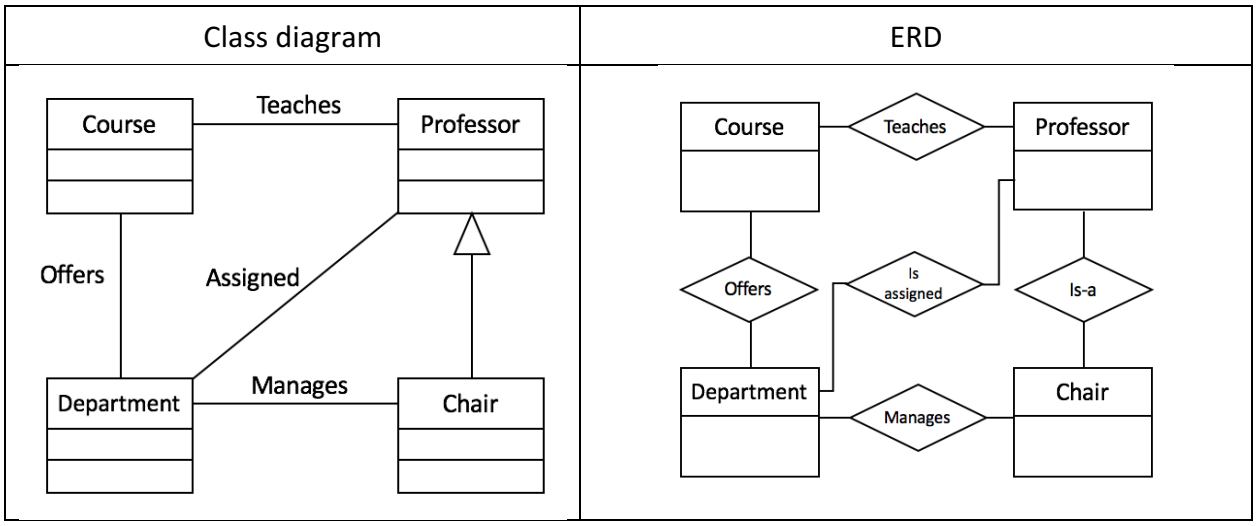
What is the difference between a class diagram and an ERD?

In short, a class diagram represents the system components (the code), while an ERD represents the database.

An ERD is not a replacement to the class diagram. In some projects, you will need both models (for example, if you use object-oriented requirements modeling and you plan to have a system database). In some other cases, you many need only one. Generally, a class diagram and an ERD are similar, but they may not be identical.

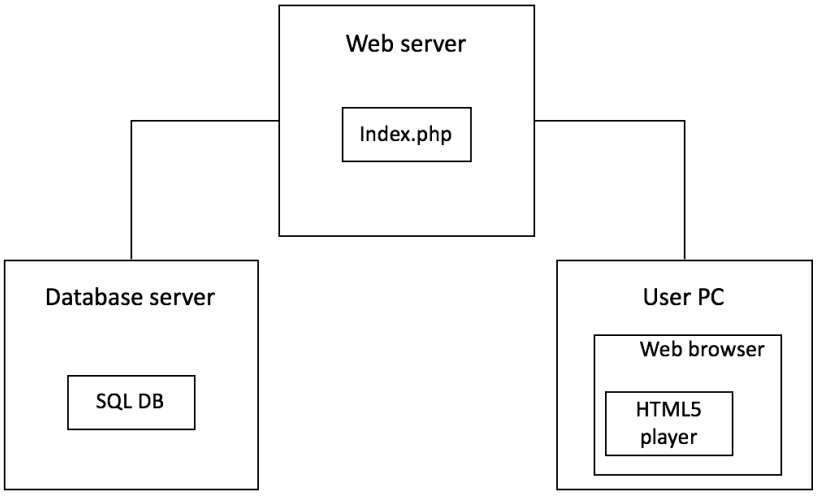
Class diagrams focus on the system static representation and the overall system behavior. On the other hand, ER diagrams focus on system data.

Example: King Saud University has several departments. Each department is managed by a chair. Professors must be assigned to one or more departments. A professor teaches one or more courses. A course may be taught by multiple professors.



Deployment diagram

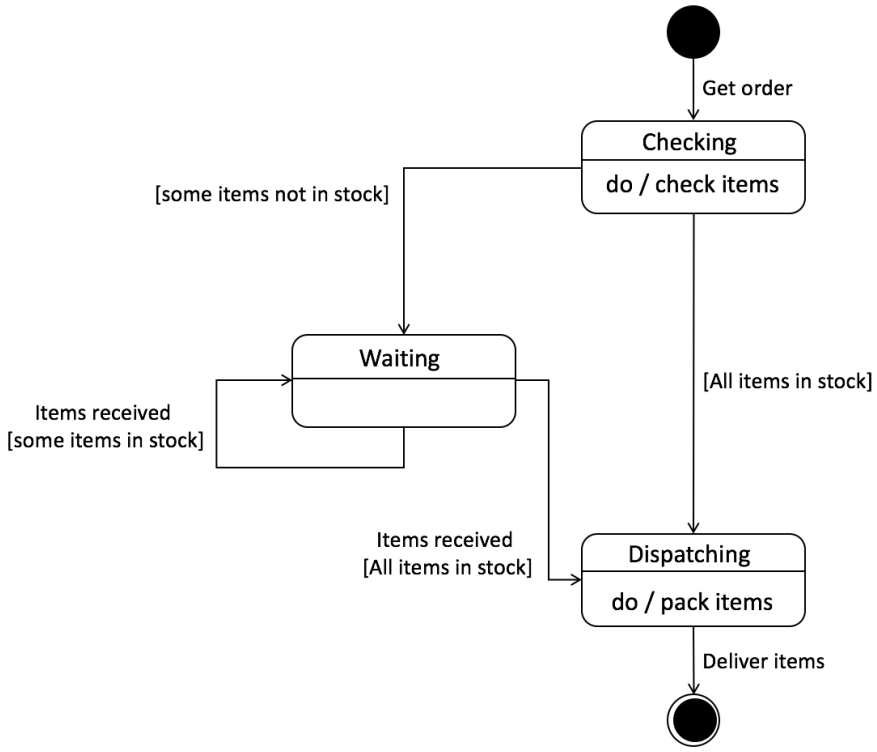
A deployment diagram specifies the physical hardware that will run the system. Each block in the deployment diagram represents a node. A node here is a physical entity (such as a PC) where the system components will be uploaded.



State diagram – Describes the state changes of the system

A state diagram (also called a state machine diagram) models the dynamic behavior of a class, a module, or the entire system as a sequence of states as a reaction to internal or external stimuli. You do not need a state diagram for each class or function.

The diagram below is a state diagram that shows the system states while handling a user order upon checkout.



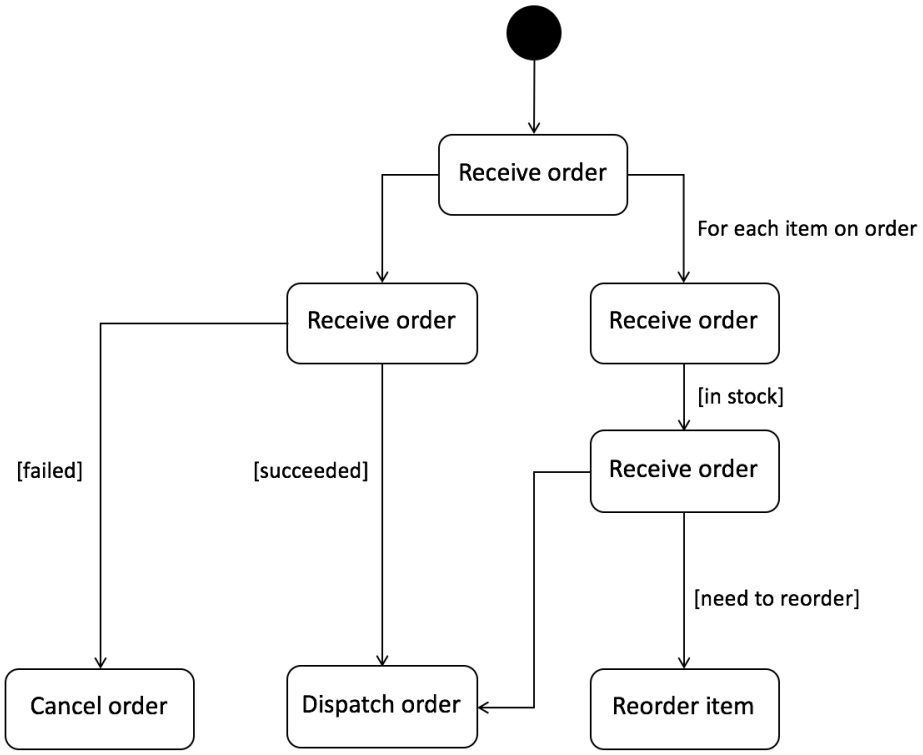
In a state diagram, you need to define each of the following:

- Initial state: the state of the system before the process (the dark circle at the beginning of the diagram above).
- System state: represented as a box with the state name and the action that the system does while in the state. For example, *checking, dispatching*, and *waiting* in the example above.
- Event: the event that causes the system to transition from one state to another. For example, *items received* above.
- Condition: the condition that causes the system to transition from one state to another. For example, *all items in stock* above.
- Action: the action that takes place as a result of a system state. For example, *deliver items* above.

Activity diagram – Represents the activity flow of the system.

Activity diagrams show the flow of actions that happen in parallel or in sequence. Activity diagrams are not identical to state diagrams because they can be used to describe each system function (use-case).

Unlike state diagrams, activity diagrams do not need a trigger event. Below is the activity diagram for the *receive order*.



Decision tables

A decision table is a visual tabular representation that describes which actions to perform under certain conditions. It is especially beneficial when the logic of the process is complex (has multiple conditions and actions).

The table has two main sections: conditions and actions. The conditions evaluate to simple Boolean values (True/False). Each action that occur as a result of satisfying a condition (or a set of condition) will be marked.

Decision trees can be used as alternatives to decision tables.

Example: The table below represents the decision table for the following decision logic, which describes how a printer troubleshooter works.

- If the printer does not print, its red light is flashing, and the printer is unrecognized, then the user needs to check the printer-computer cable, ensures printer software is installed, and check or replace the ink.
- If the printer does not print and its red light is flashing, then the user needs to check or replace the ink and check for paper jam.
- If the printer does not print and the printer is not recognized, then the user needs to check the power cable, check the printer-computer cable, and ensures printer software is installed.
- If the printer does not print, then the user needs to check for paper jam.
- If the printer red light is flashing and the printer is unrecognized, then the user needs to ensure printer software is installed and check or replace the ink.
- If the printer red light is flashing, then the user needs to check or replace the ink.
- If the printer is unrecognized, then the user needs to ensure printer software is installed.

		Rules						
		1	**2**	**3**	**4**	**5**	**6**	**7**
Conditions	Printer does not print	T	T	T	T			
	A red light is flashing	T	T			T	T	
	Printer is unrecognized	T		T		T		T
Actions	Check the power cable			×				
	Check the printer-computer cable	×		×				
	Ensure printer software is installed	×		×		×		×
	Check/replace ink	×	×			×	×	
	Check for paper jam		×		×			

What diagrams do I need?

There are multiple UML diagrams that are included in the Analysis and Design steps of the software development. For example, as mentioned above, a component level design can be accomplished using several ways. Are we required to use all of them? Obviously not.

It depends mainly on the component complexity. Complex components need to be specified very clearly to make the implementation step more efficient. For example, if a component has a complicated logic, it would be a good idea to use decision tables to clarify it. If the system needs to be distributed on several devices, it would be clever to use a deployment diagram to show the specific deployment plan. Finally, if the system has a complicated interface, it is advisable to use an interface design to communicate and discuss the interface with the designers and the end users.

References:

https://sparxsystems.com/resources/tutorials/uml2/use-case-diagram.html
https://www.bridging-the-gap.com/what-is-a-use-case/
https://slideplayer.com/slide/10764095/
https://alagadinc.wordpress.com/2005/02/15/uml-use-case-diagrams/
https://people.ok.ubc.ca/bowenhui/310/8-DFD.pdf
https://www.cs.toronto.edu/~sme/CSC340F/2005/slides/tutorial-classes_ERDs.pdf
https://www.omg.org/spec/UML/2.4.1/Superstructure/PDF
https://www.guru99.com/deployment-diagram-uml-example.html
https://www.visual-paradigm.com/guide/uml-unified-modeling-language/state-machine-diagram-vs-activity-diagram/
https://towardsdatascience.com/10-common-software-architectural-patterns-in-a-nutshell-a0b47a1e9013