**Domain Driven Design approach**

Domaine Model Uses:

Identify essential concepts/items

Map out interactions

Describe system needs

Domain model take into account the most important items. Non-essential items can be set aside as this will not impact the projects.

Domain model focus on the larger picture and capture what we are trying to build and why.

It is a tool to exchange with the client to ensure that the main elements are defined and have the same meaning for both the users and the developer.

Domain represents one area of the business and can connected to others.

Application DDD approach:

1/ Collaboration: it is important that developer and users exchange on the requirements.

2/ Ubiquitous lagugage: ensure the vocabulary used is perfectly define. Users and stakeholders have the same understanding of a specific terms.

Event storming to define application goals:

1/ As the name event storming suggest, the goal is to talk, consider all the event possible. At this stage we want to avoid talking about solution

2/ Gather the different events/ideas that are similar

3/ Find the event trigger. This can be found by continuously asking the question what precedes this event.

Identify client needs with actors, uses cases and entities

Actors: outside the system and interact with it. They can be human or a different system.

They trigger events.

Use cases: interaction between user and system

Entities: system concepts necessary to complete the use cases. Entities can be internal to the system(database). They are long-lasting element and have a unique identity. Changing their attributes will not change the object.

1/ Defining goals. What do we (the actors) want the system to do? How and Why?

2/ Capturing interactions with use case description. What are the interactions between the users (actors) and the system? This is strictly from the user’s point of view and the interface. The background work inside the system is not part of this step.

3/ Defining the entities.

**Unified Model Language**

Use case diagram

Use case diagrams have 4 components:

**Systems:** this could be any interface, application, software, environment

**Actors:** Anyone or anything that interact the system, actors are outside the system, in the representation actors should be described with their role in the system, they should be specifically named. Actors can be Primary and Secondary. Primary are are the one who initiate the use of the system. Secondary are the one reacting to the Primary actor requests.

**Use Cases:** all the actions that can be done through the system.

**Relationships:** link between the actors and the use cases.

Relationship can be of several types:

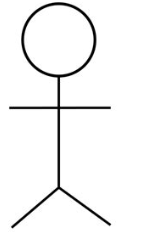
**Association:** between actors and use cases

**Include:** within the system we can have include use cases, they are linked to base use case. The included used cases will be actioned when a use cases is requested. (Base case “log in a website”, will trigger an include use case in the system “verify login details”)

**Extend:** extend use case will happen only if certain criteria are met. They are not always triggered (error message if the log in didn’t pass the verify step).

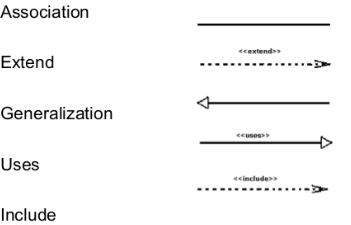
**Specialised:** act as a child use case to a general use case (parent use case), this type of relationship could be applied to actors.

System is represented in the middle with a large rectangle.

Actors are represented with :  their role written underneath. They are outside the system. Primary actors will be on the left of the diagram while secondary actors will be on the right.

Use cases (goal) are written in an oval, they are inside the system.

Relationship between actors and use cases will be marked with a line. Depending on the relationship they type of line to use will be different.



Package diagram

The package diagram is similar to a use case diagram.

The main difference is that instead of use case we would have packages in the system.

Packages act as a group of use cases. It can be useful to draw a package diagram before a use case diagram, which would be more detailed.

Class Diagram:

This is how a class is represented.

|  |
| --- |
| **Class =** the name of the object |
| **Attributes =** characteristics of the object |
| **Methods =** actions that can be performed by the object or attributes we want to be able to amend through methods. |

Each attributes and method should be preceded by the sign below in order to indicate the visibility

- for private, info can’t be access by other classes

+ for public, info can be accessed by other classes

# for protected, can be accessed by subclass or same class

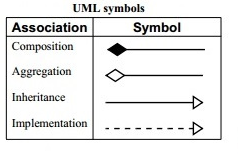
~ for package/default, can be access if the package is the same.

**Association:** When a method is associated between 2 classes, a single line will be drawn with the method on it.

**Inheritance:** We can create subclass that would inherit attribute and methods from the “superclass”.

**Aggregation:** When a class B is composed of aggregation of a class A. An instance of Class A is not necessarily part of class B and can exist independently. Class B is also independent, if Class A ceased to exist, Class B would still exist.

**Composition:** Class B composed class A. In the absence of Class A, class B would no longer exist



Multiplicity

Link between classes have a multiplicity that will be mentioned in on the link symbol.

Multiplicity can be:

n: specific number(e.g. 1 computer for 1 person)

n..\*: from n to any numbers (can start from 0, e.g. company can have from 1(n) to an undefined number of employee.

m..n: between a range of number.

Sequence diagrams

This diagram is based on 2 components actors and objects.

Actors: Actors are the elements that interact with the system.

Objects: all the object/entity of the system, they will be placed horizontally in sequential order in the diagrams.

Event sequence will display all the interactions between actors and objects.

Diagrams is being drawn downward following the order of the sequences.

Messages: they are actions or query they are represented with a line.

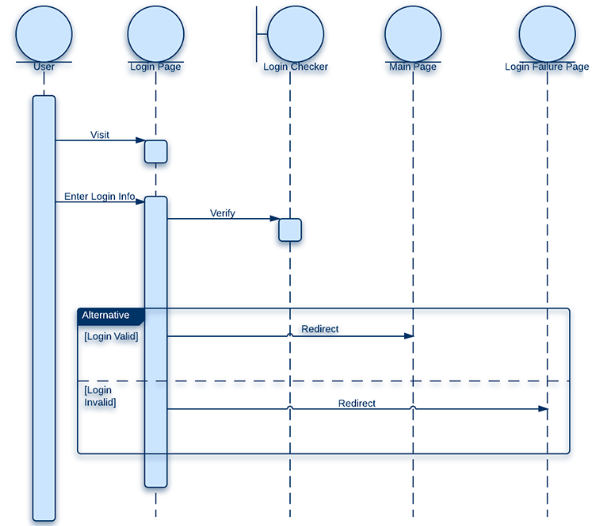
Return/reply message: They are represented in dashed lines. They are the reply form a query.

Alternative frame:

Frame drawn in the sequence diagram when several options are possible (bank card payment, pin can be correct or incorrect, card, can be valid, invalid etc…)

Activation boxes:

Boxes drawn between objects that would display they lifetime during the full process/Sequence.



Activity Diagram

Similar to flow charts but with a different representation.

Entity relationship Diagram (ERD)

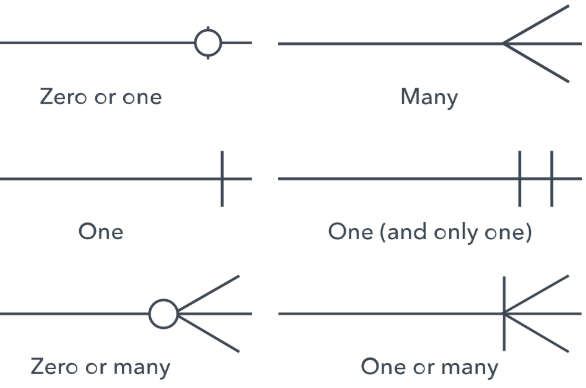
Component

Entities: similar to class

Attributes: link to entities

Relationship between entities.

Cardinality = similar to multiplicity for class diagram.



Attributes can be:

Primary key (PK) = unique for each entity, cannot be change or be null/blank. It is an unique identifier.

Foreign key (FK) = name given to an attribute if it is used as a primary key in a different entity

Composite primary key (PK,FK) = use a combination of foreign key in order to create a primary key.

Bridge table: often used between 2 entities that would have a many to many relationship.

It would break the many to many relationship, which in concept is better for database management.

DDD (ZenD) Bradley HolT talk. (youtube)

Identify core domain

Distil core domain

Focus on core domain

To identify Core domain question below can be asked:

What make the system worth writing?

Why not buy off the shelf?

Why not outsource?

Bounded context:

Delineates the applicability of a particular model

Bounded context allow model to be explored isolately

Define clearly : who is responsible? Which part of the application is the bounded context applicable? What manifestation the bounded context will take code database

Ubiquitous language

Context map: identify what actually exist and the relationship between the context

Entity:

Defined ye a thread of continuituy and identity

Only responsibilities should be around identity and life cycle

Can be composed of other entities or value objects

Value object:

Define by its attributes

Treat as immutable (don’t have to be but good practice)

Aggregate:

Group of entitied and value objects

Useful when definig transactions distribution and concurrency

Bounded context will have multiple context