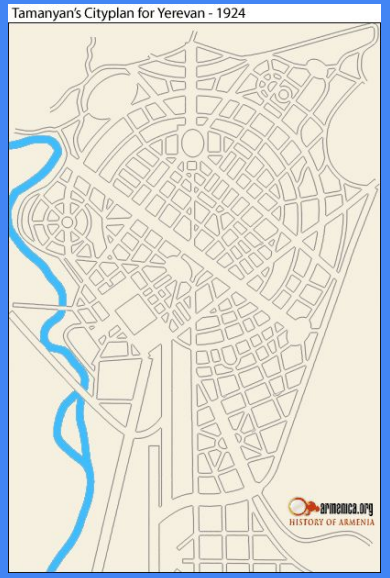
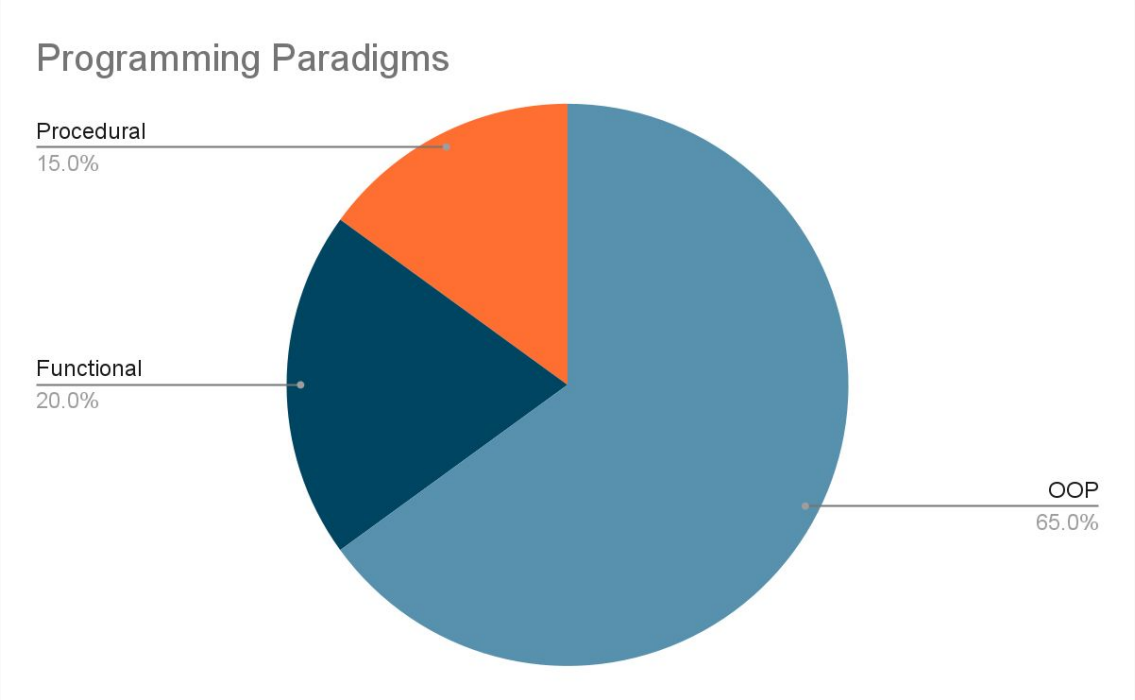
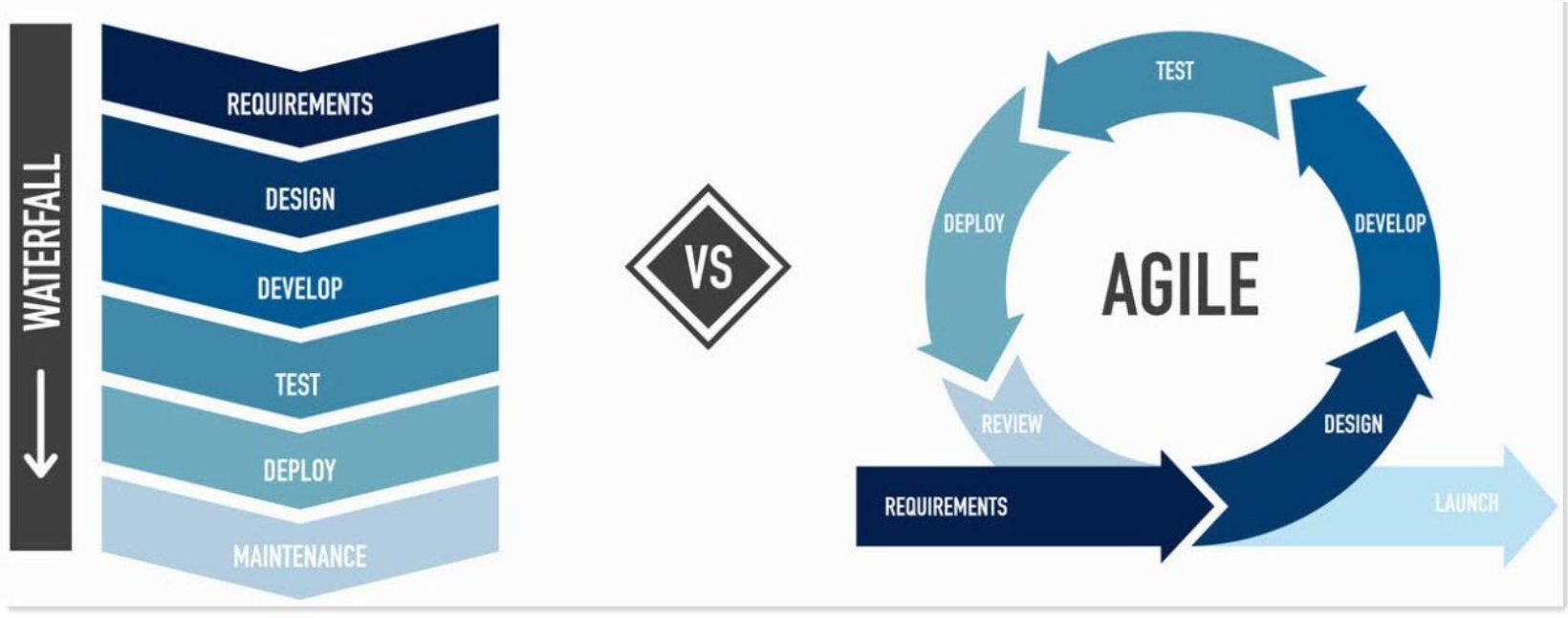
Object Oriented Analysis and Design



Why do we build software as we do ?



Why do we build software as we do ?



Object-Oriented Paradigm

Objects, representing things in code, help the code to stay **organized**, **flexible**, and **reusable**.

- Objects keep code organized by putting related details and specific functions in distinct, easy-to-find places.
- Objects keep code flexible, so details can be easily changed in a modular way within the object, without affecting the rest of the code.
- Objects allow code to be reused, as they reduce the amount of code that needs to be created.

Object Oriented Thinking

- Why it is important
- Thinking in Objects
- Big picture with System Thinking
- Modeling real life models into objects

Object-Oriented Thinking

It is a critical skill for OOA/D to think in systems to be able to break down a whole into components with correctly assigned responsibilities, behavior, interaction.

Object-Oriented Thinking

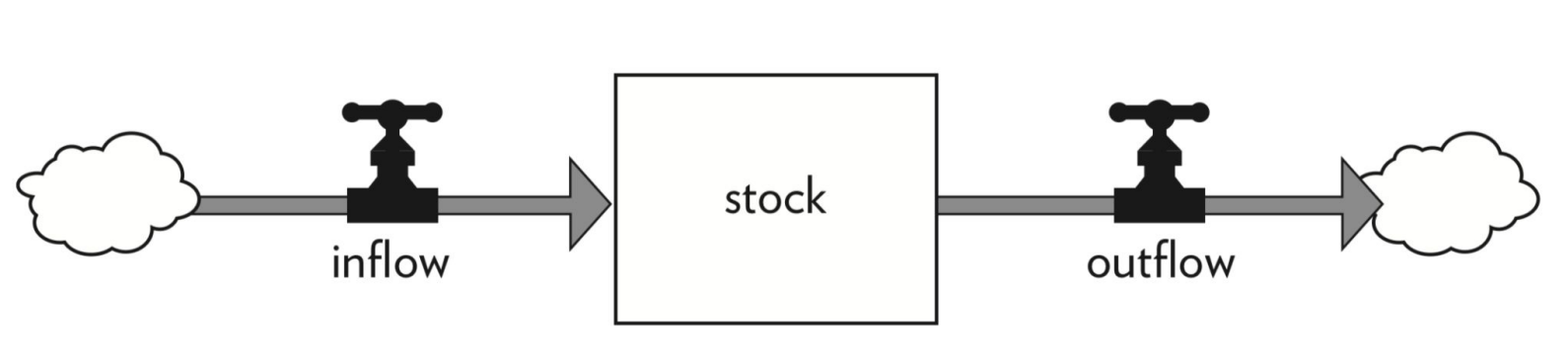
Object-oriented thinking involves examining the problems or concepts at hand, breaking them down into component parts, and thinking of those as objects.

When translated to object-oriented modelling, object-oriented thinking involves representing key concepts through objects in your software.

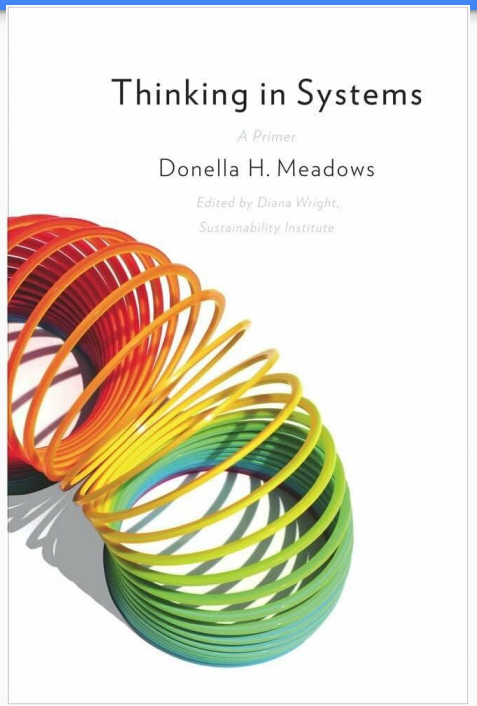
In object-oriented thinking, often everything is considered an object, even if animate or live. And objects are all self-aware, even if inanimate.

Thinking In Systems

System is defined as a set of elements or components that is coherently organized and interconnected in a structure that produces a characteristic set of behaviors, often classified as its “function” or “purpose”.



Thinking In Systems



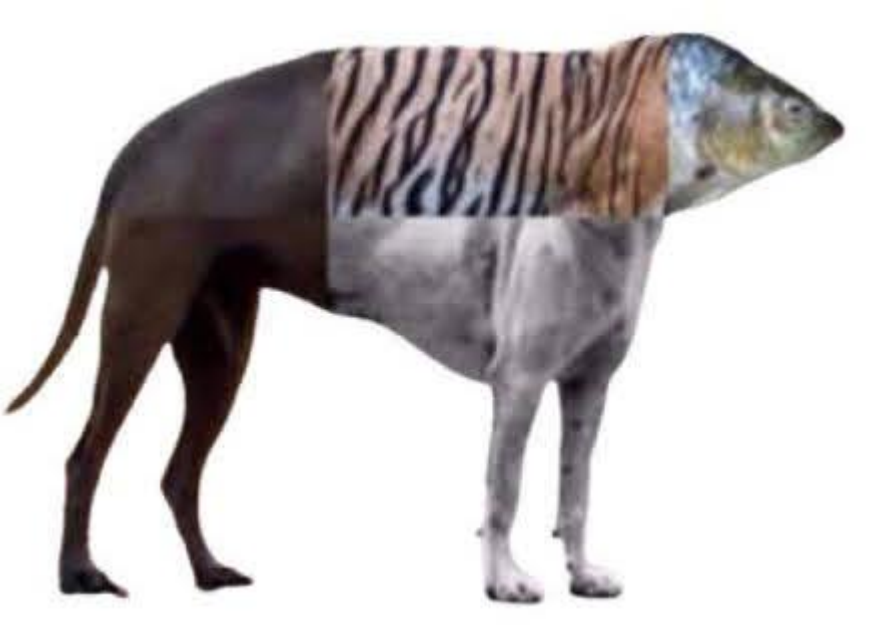
Systems consist of elements, interconnections and purpose or function

- Elements: Systems consist of multiple elements or components that are interconnected and interact with each other.
- Purpose or Function: Every system has a purpose or function it is designed to fulfill. This purpose can be explicit, such as the function of an organization, or implicit, as in the functioning of natural ecosystems.

Object Oriented Analysis and Design

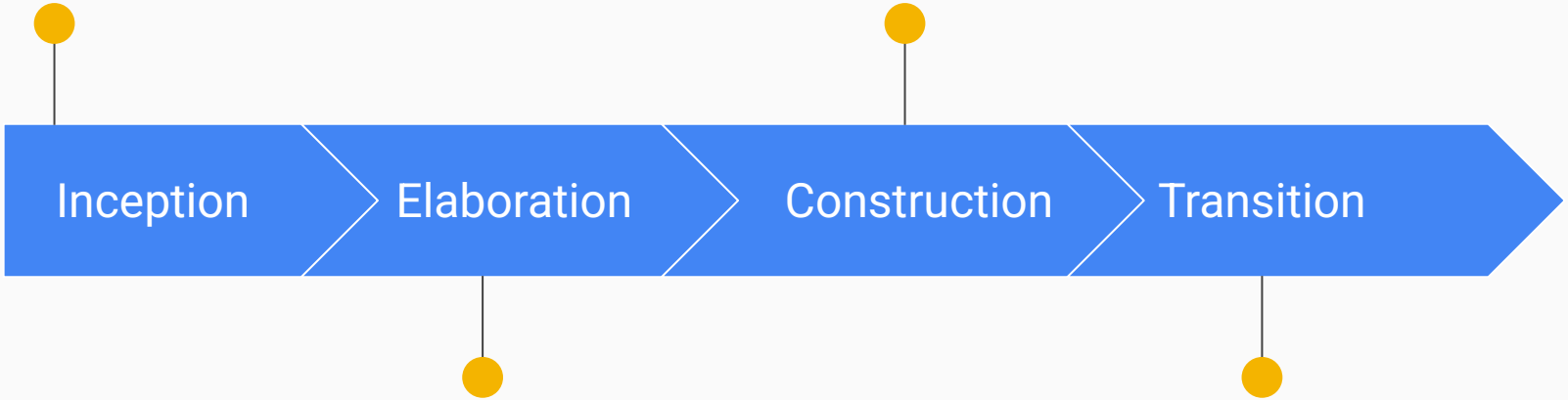
- Phases of Iterative Software Development
- Structure of OOA/D
- Compare and contrast analysis and design.
-

Do the right thing and do the thing right



Inception - approximate vision, business case, scope, vague estimates.

Construction—iterative implementation of the remaining lower risk and easier elements, and preparation for deployment.



Elaboration - refined vision, iterative implementation of the core architecture, resolution of high risks, identification of most requirements and scope, more realistic estimates.

Transition - beta tests, deployment

Analysis and Design

Analysis emphasizes an investigation of the problem and requirements, rather than a solution.

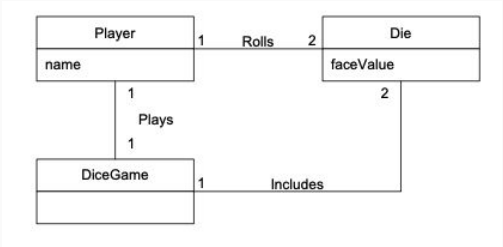
Finding and describe the objects or concepts in the problem domain.

Design emphasizes a conceptual solution that fulfills the requirements, rather than its implementation.

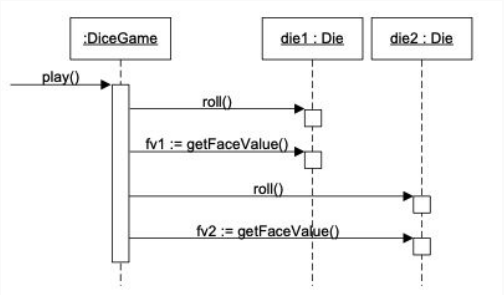
Defining software objects and how they collaborate to fulfill the requirements

Analysis and Design

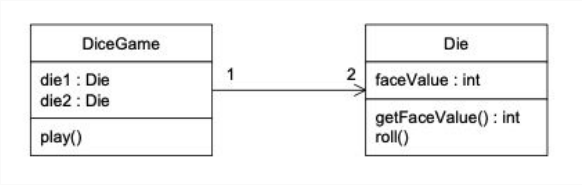
Define Domain Model



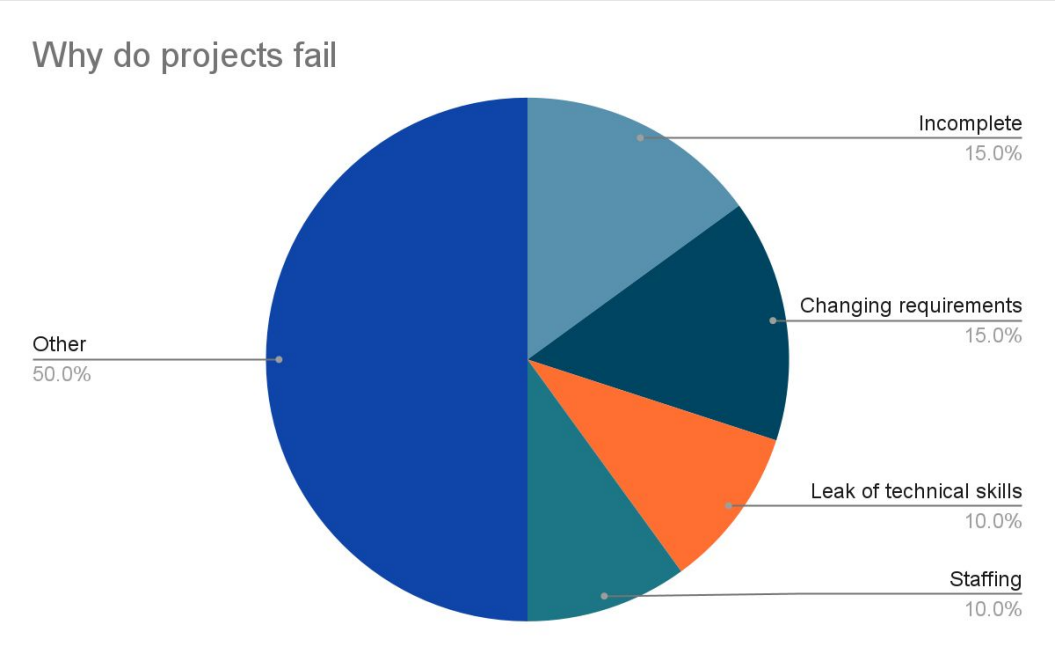
Define Interaction Diagrams



Define Class Diagrams



Requirements



Requirements

Requirements are conditions or capabilities that must be implemented in a product, based on client or user request.

It is more than simply the client's vision. Instead, eliciting requirements involves actively probing the client vision, clarifying what may not have been told, and asking questions about issues the client may not have even considered.
It is also important to establish potential trade-offs the client may need to make in the solution.

Requirements

Functional - what the system or application is expected to do. Features, capabilities, security.

Non-functional - how well the system or application does what it does. Performance, resource usage, and efficiency.

FURPS+
Usability - human factors, help, documentation.
Reliability - frequency of failure, recoverability, predictability.
Performance - response times, throughput, accuracy, availability, resource usage.
Supportability - adaptability, maintainability, internationalization, configurability.
Implementation, Interface, Operations, Legal.

Supplementary Specification

The Supplementary Specification captures other requirements, information, and constraints not easily captured in the use cases, including quality attributes or requirements.

- reports
- hardware and software constraints (operating and networking systems, ...)
- development constraints (for example, process or development tools)
- other design and implementation constraints
- internationalization concerns (units, languages, ...)
- documentation (user, installation, administration) and help
- licensing and other legal concerns
- standards (technical, safety, quality)
- operational concerns (for example, how do errors get handled, or how often to do backups?)
- domain or business rules
- information in domains of interest (for example, what is the entire cycle of credit payment handling?)

Quality Attributes and Trade-Offs

Common trade-offs

- **Performance and maintainability** – High performance code may be less clear and less modular, making it harder to maintain. Alternately, extra code for backward compatibility may affect both performance and maintainability.
- **Performance and security** – Extra overhead for high security may lessen performance.
- **Deadline**

Organize requirements

Organize requirements and iterations by risk, coverage, and criticality.

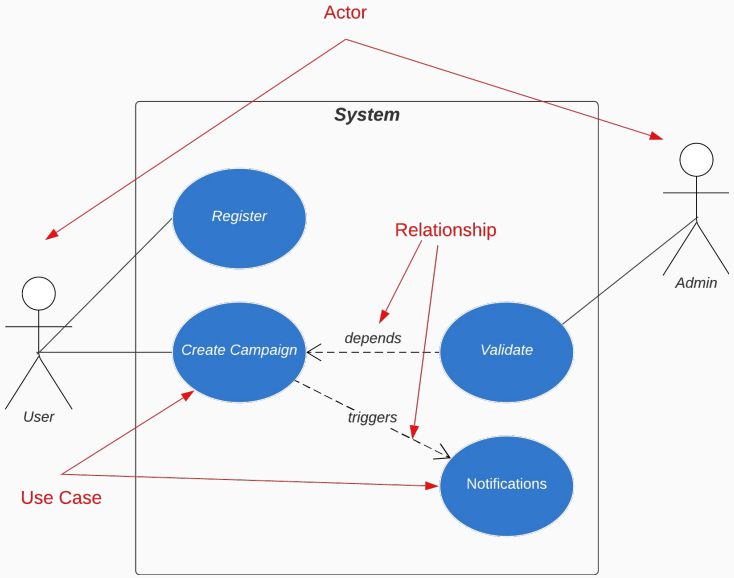
- **Risk** - includes both technical complexity and other factors, such as uncertainty of effort or usability.
- **Coverage** - implies that all major parts of the system are at least touched on in early iterations perhaps a "wide and shallow" implementation across many components.
- **Criticality** - refers to functions of high business value.

Context and Consequences

Context provides important information when deciding on the balance of qualities in design. For example, software that stores personal information, which the public can access, may have different security requirements than software that is only used by corporate employees.

Consequences - sometimes, choices made in software design have unintended consequences. For example, an idea that seems to work fine for a small amount of data may be impractical for large amounts of data.

Use Case Diagrams



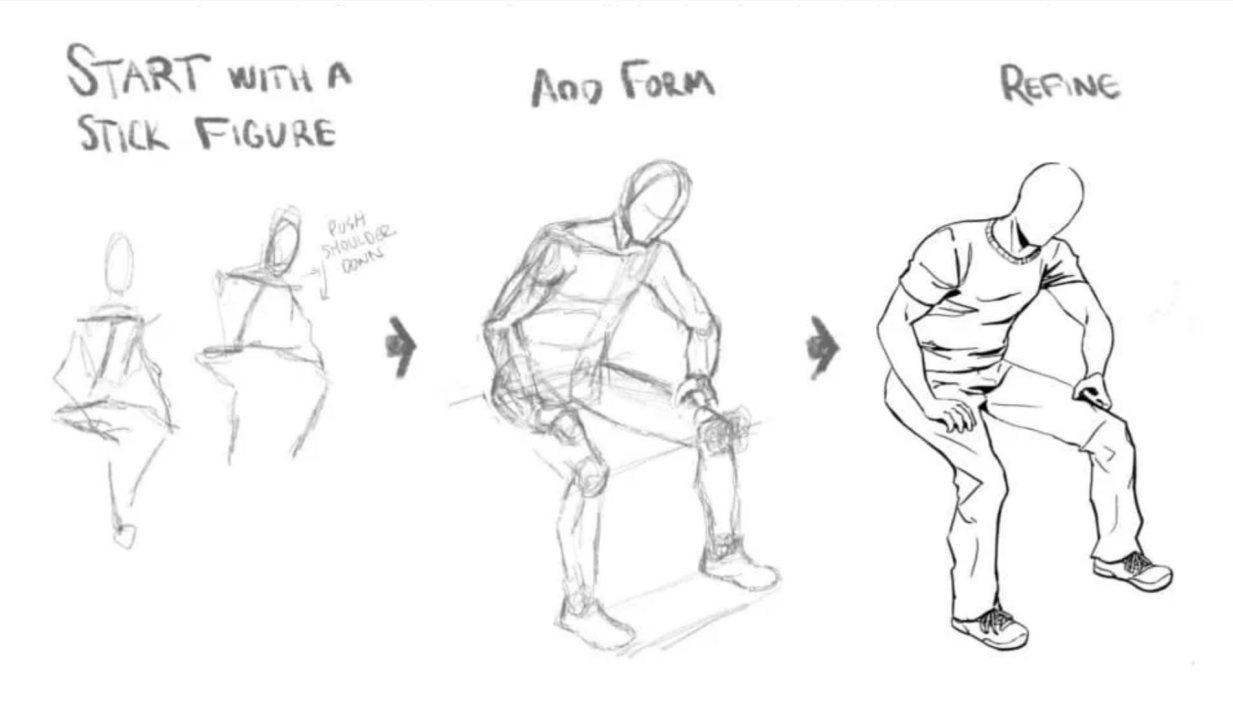
- Scenarios in which your system or application interacts with people, organizations, or external systems
- Goals that your system or application helps those entities (known as actors) achieve
- The scope of your system

Conceptual Design

Conceptual designs are created with an initial set of requirements as a basis. The conceptual design recognizes appropriate components, connections, and responsibilities of the software product.

- Outline the more high-level concepts of final software product.
- Expressed or communicated through **conceptual mock-ups**.
- Every component has a task it needs to perform, as known as its **responsibility**.

What it looks like



Domain Models

A **domain model** is a conceptual representation of a problem domain.

Key components and characteristics

- Entities
- Attributes
- Relationships
- Methods or Behaviors
- Constraints
- Abstraction

Entities or Conceptual Classes

Entities are objects, concepts, or things that exist within the problem domain. They are typically represented as classes in the domain model. Each class represents a specific entity and includes attributes to describe the characteristics of that entity.

Strategies to Identify Entities

Click to see list

Strategies to Identify Entities

Identify noun phrases:

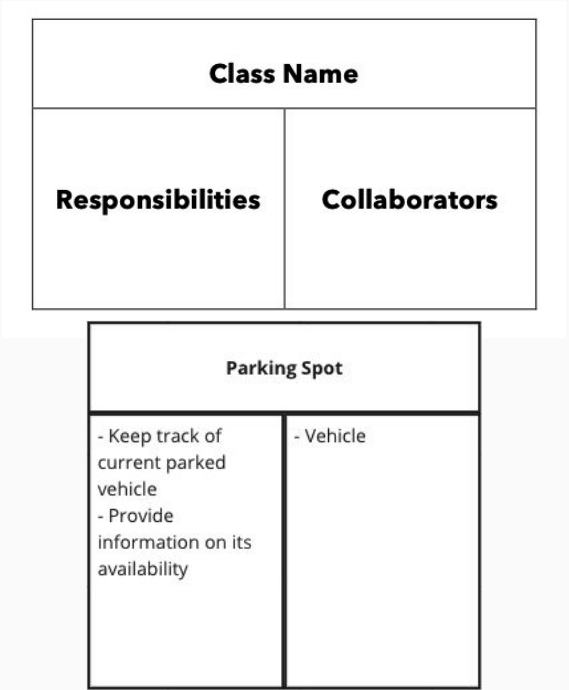
The **parking lot** should have the capacity to park 200 **vehicles**.

Two different types of **parking spots** are compact and large.

Three types of **vehicles** should be allowed to park in the **parking lot**, which are as follows:

Class Responsibility Collaborator (CRC)

CRC cards are used to record, organize, and refine the components of system design. CRC cards are designed with three sections: the **class name** at the top of the card, the **responsibilities** of the class on the left side of the card, and the **collaborators** on the right side of the card.



Steps to create Domain Model

1. List the candidate conceptual classes using the Conceptual Class Category List and noun phrase identification techniques related to the current requirements under consideration.
2. Draw them in a domain model.
3. Add the associations necessary to record relationships for which there is a need to preserve some memory.
4. Add the attributes necessary to fulfill the information requirements.

Good to know

- Constant communication and feedback is key to creating the right solution that meets client needs and works within any restrictions that may exist.
- During the analysis of functional requirements, attention to detail within the relevant context is highly beneficial.