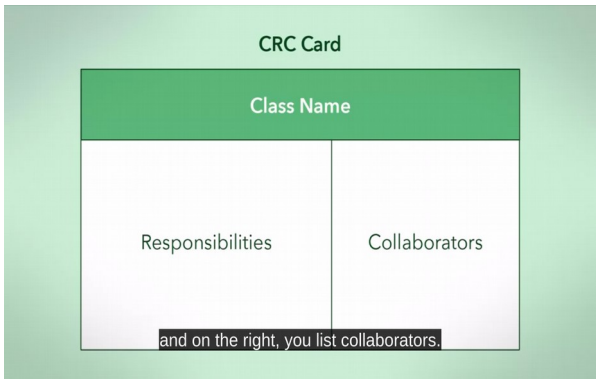
Object Oriented Design

Functional Requirements – Non Functional Requirements

Conceptional Design

- Components (main classes)
- Responsibility (purpose of each class)
- Connections (for each class , which classes you need to communicate with)
- Mockup Design UI (drawing initial design on paper or drawio website)

CRC Class Responsibility Collaborator



Technical Diagrams (Structure and Behavior of the system)

Objects Types

https://www.coursera.org/learn/object-oriented-design/supplement/gkjJv/categories-of-objects-in-design

- Entity : like student , book , regFile , building , room , which describes objects in world.
- Boundary : which communicates with other system (internet : Web APIs , user : GUI , software:database , sockets)
- Control : which connect entity objects (controller , simulator , Design Patterns Objects)

Testing – Design Reviews – Asking Developers Opinion

- to improve the conceptual integrity
- to fix bugs of the code

User Story

- as I __some user__ I want to ___ some requirement ___ so that __ some action __.

US ID #	USER STORY
	Basics
1	As an owner, I want to add a thing to my things. Each thing is created with title, maker, description, status, and dimensions (Length x Width x Height).
2	As an owner, I want to view one of my things, its title, maker, description, dimensions and status and if it is being borrowed, the username of the borrower.
3	As an owner, I want to edit a thing in my things.
4	As an owner, I want to delete a thing in my things.
	Lists
5	As an owner, I want to view a list of **all** my things, each with their title, description, and status.
6	As an owner, I want to view a list of all my things that are **available**, each with their title and description
7	As an owner, I want to view a list of my things being **borrowed**, each with their title and description.
	Photographs
8	As an owner, I want to optionally attach a photograph to a thing of mine.
9	As an owner, I want to view any attached photograph for a thing.
10	As an owner, I want to delete any attached photograph for a thing of mine.
	Borrowing
11	As an owner, I want a thing to have a status of "Available" if the thing can be borrowed or was returned, or a status of "Borrowed" if the thing is currently being borrowed.

Four OO Design Principles

- Abstraction
  - simulates the class with the data we care about for determine purpose.
  - make things abstracted and simple without caring how it is implemented
  - hiding the information at the design level)
  - ex : talk about regFile and his responsibility without caring about how will it implemented in the implementation level
- Encapsulation
  - put the attributes and methods in one class
  - hiding the implementation details from other classes by using access modifiers
  - to encapsulate the details to use it as a black box that can be changed easily without effect on the user, it is a re-usability concept
- Decomposition
  - break down the problem class into smaller pieces
  - what classes can be split from your class?
- Generalization
  - eliminates redundancy
  - inheritance – abstract class … cat , dog – animal , animal is a general class
  - functions its self
  - it may be bad if the subclass dosen't add features to the super class, if inheritance can be replaced by a variable , that would be better.
-

difference between abstraction and encapsulation

https://javarevisited.blogspot.com/2017/04/difference-between-abstraction-and-encapsulation-in-java-oop.html

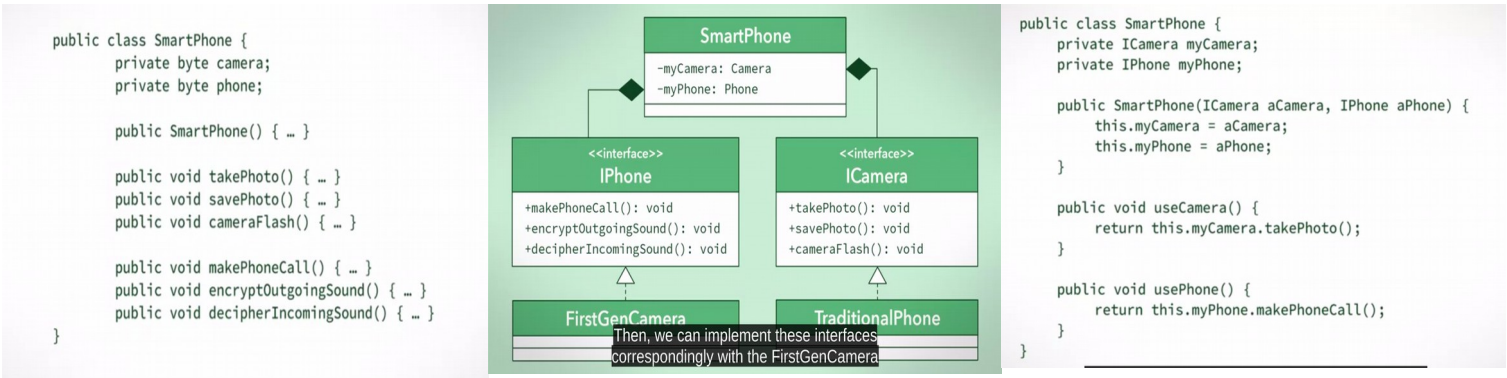
UML

- Association
  - the 2 objects communicates via methods but dosen't contain each other
  - if one of the objects die , the other will remain, they are not dependent
  - in code this is translated to methods communication (one as a parameter to other's method) ex:public class student { public play (Sport my_sport){} }, Sport is Association with student

- Aggregation
  - object contains other object (has-a relationship) but can exist without the other (both can)
  - week relationship , week dependency
    Ex : Plane and Crew , Plan has a Crew but each of them can exist without the other.
  - Ex : BookShelf and Books , BookShelf has a Books but each can exist without the other.
  - in code this is translated into undefined number of the other object (usually as pointers so it can be non-exist) ex : class BookShelf {vector<Book> myBooks; void addBook(Book){..}}
- Composition
  - strong has-a relationship where the object1 must contains object2 to exist
  - whole object cannot exist with out any of its parts, and parts get destroyed if he destroyed
  - in code (attributes of class are composition to it as it is declared with it -strong dependency-) ex : public class ray2 { private String 7ray2; } 7ray2 will be created along with ray2

Design Principles

- Coupling
  - complexity and dependency between classes
  - lower is better
  - degree – ease – flexibility
  - degree is number of connections between classes(Association-Aggregation,Composition)
  - ease how this connection is obvious
  - flexibility is how much the connecting can be replaced to other classes (like ligo games)
  - if the dependency or connection of your module is not clear and require reading the module code to do it , then it is a high (bad) coupling
- Cohesion
  - **clear** and specific purpose of a class
  - if the class has many functionalities so use decomposition to split it into different classes.
- Separation of Concerns
  - like decomposition split big classes to separate classes each one has its high cohesion.
    - Each class has a specific clear purpose (high cohesion)
    - each class can be used again in other systems
    - hide or encapsulate each purpose in one class
  - ex : mobile with camera and phone
    - it is bad design if the smartphone see the implementation of camera and phone.
    - use separation of concern to make camera and phone modules with high cohesion.
    - Camera and phone modules can be used in other systems.
    - Higher cohesion (clear purpose) but also higher coupling(more dependency)
    - note: it dosen't make sense that you make a smartphone inherits from phone class with added camera,this is bad inheritance, we didn't solve the cohesion or coupling problems.



- Conceptual Integrity
  - consistency تناسق .. looks like that one has wrote the code even if many people wrote it.
  - uniform design
  - uniform naming convention
  - https://www.facebook.com/sameh.serag.deabes/posts/2532298693697957
- Information Hiding
  - access modifiers (private public protected)
  - showing only the methods or attributes that would be used by others and hide the else.
  - Interface design
  - protecting data
- Inheritance Problems

- subclass must add **features** or modify **methods** of the superclass or it can be replaced by an additional attribute in superclass and superclass is enough.

- Sequence Diagram
  - Nested activation is used when class call function of its self.
  - Nested activation is used when it is UI action happens in the same class
- Model Checking
  - testing
  - modeling phase : creating the model testing criteria

Software Engineering بالهجايص

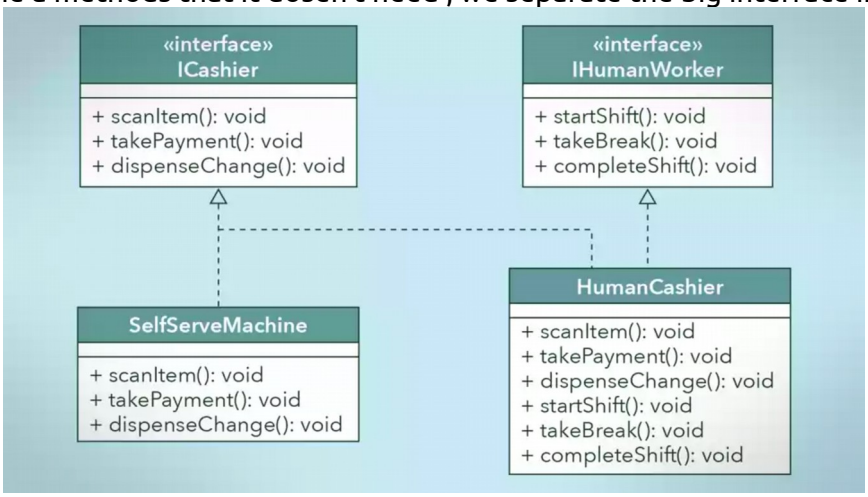
- Breaking Dependency
  https://www.facebook.com/sameh.serag.deabes/posts/2015976898663475
  - favor composition over inheritance
  - ما تعملش وراثة واعمل تجميع/تركيب compostion.

    مثال: لو عندك class اسمها أسد الغابة وناوي تخليها ترث class من inherits تانية اسمها الأسد، فكر تاني، يمكن يكون attribute/field/property و خليها fulltime هو هو نفس الأسد بس شغال في الغابة بحوام كامل للأسد، فلما يجيبلك أسد class جديدة ترث من الأولى. طبعا ال class السيرك هاتلاقيها هاتسد معاك من غير ما تحتاج تعمل class جديدة بتاعت الوظيفة الجديدة دي ممكن تزود فيها بيانات مفصلة عن طبيعة العمل، وأوقات الحوام، و مكان الغابة...إلخ.
- Dependency Degrees https://www.facebook.com/sameh.serag.deabes/posts/2052502805010884
- Liskov https://www.facebook.com/sameh.serag.deabes/posts/2017502638510901
-

SOLID Design Principles

- **Single Responsibility (S)**
  - **A class should have one and only one reason to change, meaning that a class should have only one job.**
  - the attributes and methods of the class should serve that single responsibility of that class.
  - the class itself is a single unit that must either be entirely used or not at all.
  - It is for classes and even functions.
  - https://www.facebook.com/sameh.serag.deabes/posts/2008818352712663

- **Interface segregation (I)**
  - A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use.
  - https://www.facebook.com/sameh.serag.deabes/posts/2015976898663475
  - Separate the big interface to smaller interfaces
  - ex : instead of having casheir interface for both human and machine and having the machine a methods that it dosen't need , we separate the big interface into smaller ones.



- **Liskov Substitution (L)**
  - every subclass/derived class should be substitutable for their base/parent class.
  - If a class, S, is a sub-type of a class, B, then S can be used to replace all instances of B without changing the behaviors of a program.
  - When you decide to inheritance or not. Ask your self is the subclass (is-a) superclass?? or it changes the constrains of the methods of the superclass.
  - If we have a class real duck and subclass toy duck with battery, so it is a bad inheritance.
    - As toy duck needs battery to move and change the constrains of the movement.
    - Toy duck is not (is-a) real duck.
  - https://www.facebook.com/sameh.serag.deabes/posts/2017502638510901

- **Open – Closed Principle**
  - Classes should be **closed**
    - for **tested** – final version – **classes** must be **closed** to **edit**
  - classes should be **open** for **extension** for edit or adding **new features**, extension is done by **inheritance** of these classes or **polymorphism** of **extended behaviors**
  - **separate what varies from what stays the same.**

- **Dependency Inversion Principle**
  - Classes should depend on high level classes (interfaces or abstract classes) not on a low level concrete classes , to allow extension.
  - Very similar to [Program to interface not to an implementation]
  - Strategy Pattern
- **Composing Objects Principle**
  - Favor Composition over inheritance
    - Objects is less coupled by composition
    - allow objects to dynamically add behaviors at run-time.
    - Flexibility
  - Decorator Pattern , Composite Pattern
  - disadvantage, similar code , breaks [Don't repeat yourself Principle]

  - Composition Vs Inheritance
    - Do you have a set of related classes or unrelated classes?
    - What is a common behavior between them?
    - Do you need specialized classes to handle specific cases or do you simply need a different implementation of the same behavior?

- **Principle of Least Knowledge (Law of Demeter)**
  - It is not allowed to used deep nested call methods within classes (use only one dot)
  - Decreasing Dependency between classes
  - driver class calls methods of car class , but it shouldn't call methods of the engine of the class , it breaks abstraction.
  - a method, M, of an object should only call other methods if they are:
    - encapsulated within the same object, encapsulated within an object that is in the parameters of M,
    - encapsulated within an object that is instantiated inside of M, or encapsulated within an object that is reference in an instance variable of the class for M
  - https://www.facebook.com/sameh.serag.deabes/posts/2038661573061674