CoGrammar

# Lecture 13: Software Design

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education

## Lecture Housekeeping

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly. **(FBV: Mutual Respect.)**
- No question is daft or silly - **ask them!**
- There are **Q&A sessions** midway and at the end of the session, should you wish to ask any follow-up questions. Moderators are going to be answering questions as the session progresses as well.
- If you have any questions outside of this lecture, or that are not answered during this lecture, please do submit these for upcoming Open Classes. You can submit these questions here: **Open Class Questions**

## **Lecture Housekeeping** cont.

- For all **non-academic questions**, please submit a query: **www.hyperiondev.com/support**

- Report a **safeguarding** incident: **www.hyperiondev.com/safeguardreporting**

- We would love your **feedback** on lectures: **Feedback on Lectures**

## Lecture Objectives

- Learn about the importance of software design in software engineering
- Learn how to design systems
- Learn about the models that can be used in software design

CoGrammar

## Engineering

Imagine a civil engineer is taken to a sight and asked by a client to build a footbridge. The engineer thinks to himself ‘oh how simple’ and gets straight to work.

1. He calls his steel guy and orders some steel
2. He get the construction team assembled and tells the workers where to dig holes
3. He gets the steel workers to set up posts and get the concrete poured in.
4. He gets in a crane and starts moving the rest of the beams into place.
5. The bridge is complete.

**NOTE:** The process goes on exactly like that, after getting the project, the engineer starts building.

## Engineering

From the process that our engineer took, we will have 1 of 2 outcomes.

1. Visibly Faulty
2. Visually Perfect

CoGrammar

## Software

Like the civil engineer, **software people** might run into this situation, where there is a problem statement given, and they jump straight into the solution.

**Behind visibly faulty**

- It’s clear that the project is not well thought out
- It’s clear that there are issues or will be issues looking at the solution as a user
- It’s unlikely that the client will want to use the solution.

**Behind visually perfect**

- This is the most dangerous type of solution
- On the surface and on first glance, the application is well built and able to perform what it need to perform
- Overtime it may degrade
- It might be hard to maintain
- IT will become legacy

## Legacy Code (Technical Debt)

Legacy code can be defined as **1** of 2 things

**Perfect Code**

- As a lazy developer, you will tell your bosses that the code is so good that it shouldn't be touched
- The code was so well written that it transcends human perception

**The actual Reason**

- The person who wrote the code did not plan it and wrote a solution that only they can understand
- The system overtime has become the foundation of a lot of business operations
- Every developer is scared to touch it because they don't want to bring the business to its knees

## Software Developer VS Engineer

Looking back at our civil engineering analogy, we know that a good **engineer** would never start building without a blueprint.

**Software Developer**

- Writes code that makes applications run

**Software engineer**

- Designs and build systems that last.

CoGrammar

## Software Engineer

Like a civil engineer, a software engineer will need to build blueprints for the system that they are building.

**Why**

- Helps with understanding the problem at hand
- Identify patterns and optimize the design
- Have a clear picture of what is being built
- Makes working in large teams easy
- Enforces good programming standards.

## Designing Systems (High Level)

Let's take a high level look at what we can do to design a systems. We want to focus on three main things.

**NOTE:** This is not a full breakdown, each diagram we go over goes into much greater detail that what is required for this bootcamp

- User Requirements
- System interactions
- Structure of the solution

CoGrammar

## User Requirements

Our system will have users, this can be actual humans or other systems that need our resources. We need to understand what each of these users will be doing.

**Use Case Diagram**

- Helps us map a user to the interactions that they will have with the system
- Allows us to show the relationship between features
- Gives us a clear representation of the features that we will need to include into our application.

CoGrammar

## Use Case Diagram

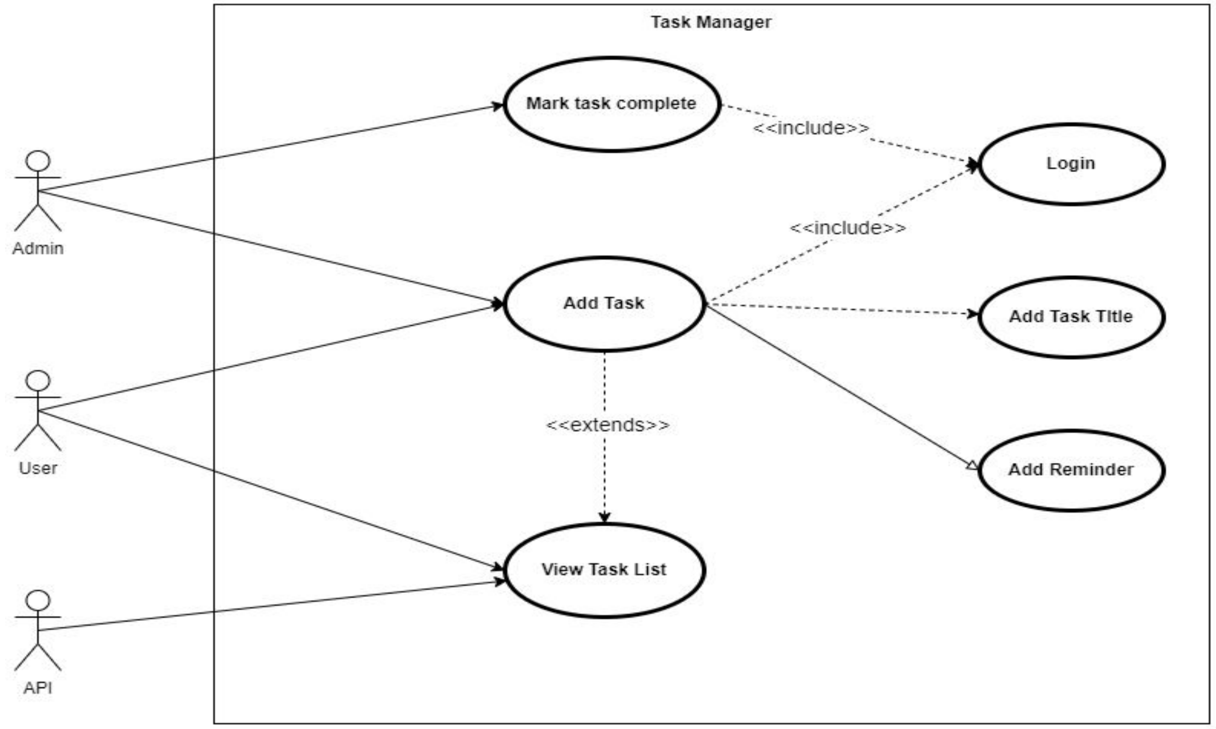
**Components**

- **Actor -** The end user or any external system that has use cases
- **Association -** A line between an actor and a use case
- **System Boundary -** Defines what is in the system and what is not

**Relationships**

- **<<extends>> -** Dashed line that represents extended operations on the base use case
- **<<include>> -** dashed line that shoes that a use case wll use funcitonality from another use case
- **Generalization -** Solid with triangle that shows a parent child relationship

## Use Case Diagram



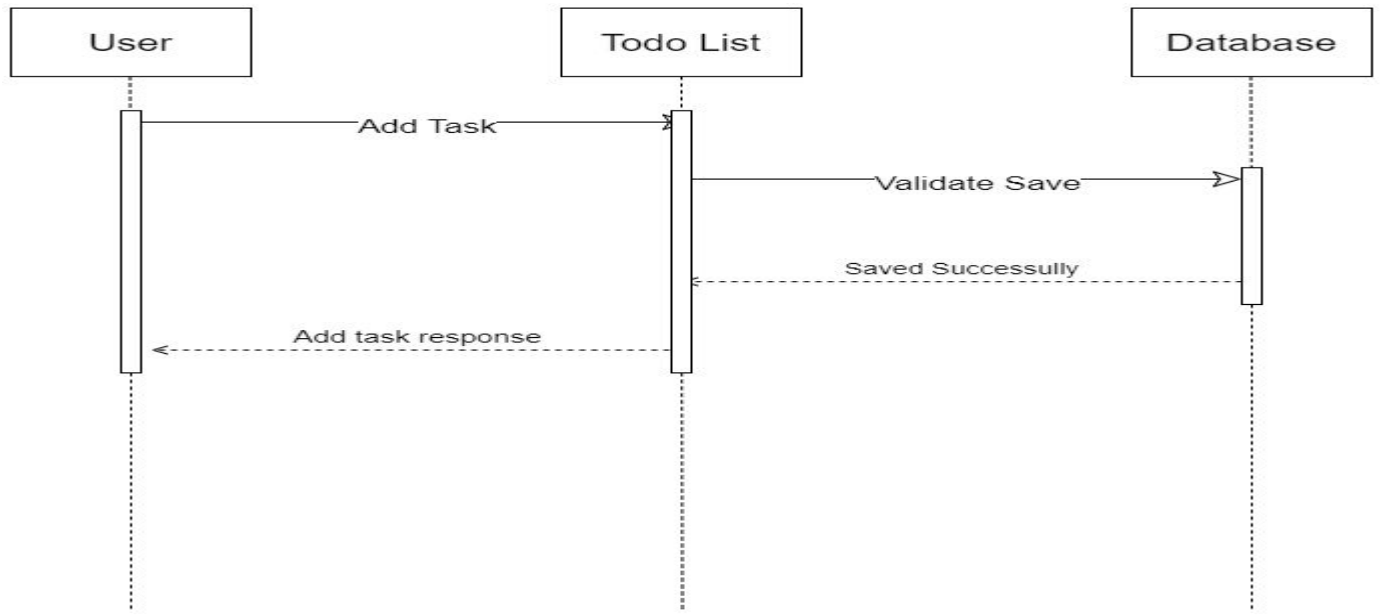
## System Interactions

Once we know who is using our system, we need to know how they will interact with out system and how our system will interact with other systems or other functions within the system.

**Sequence Diagram**

- Shows us how the different parts of the system will communicate
- Shows the flow of communication as a timeline
- Shows how each objects operations are triggered
- Show how each objects responds to operations.

# Sequence Diagram



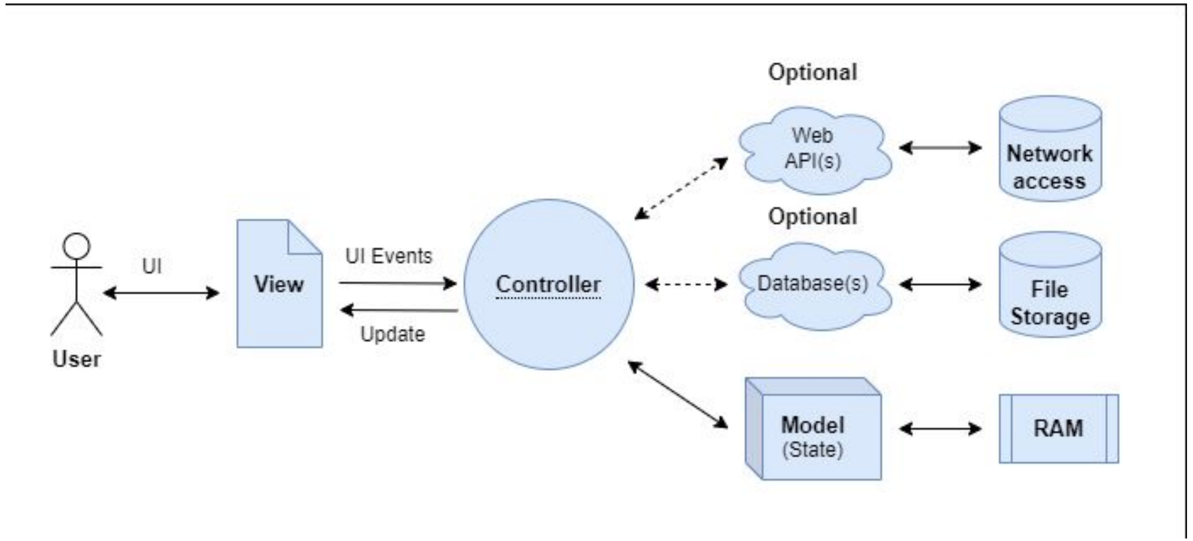
## Design Pattern

A design pattern give us better control of our code by breaking it up into different categories.

- Creates a clear separation between the different parts of the application
- Provides use some key abilities
    - Readability
    - Reusability
    - Scalability
    - Testability

CoGrammar

## MVC



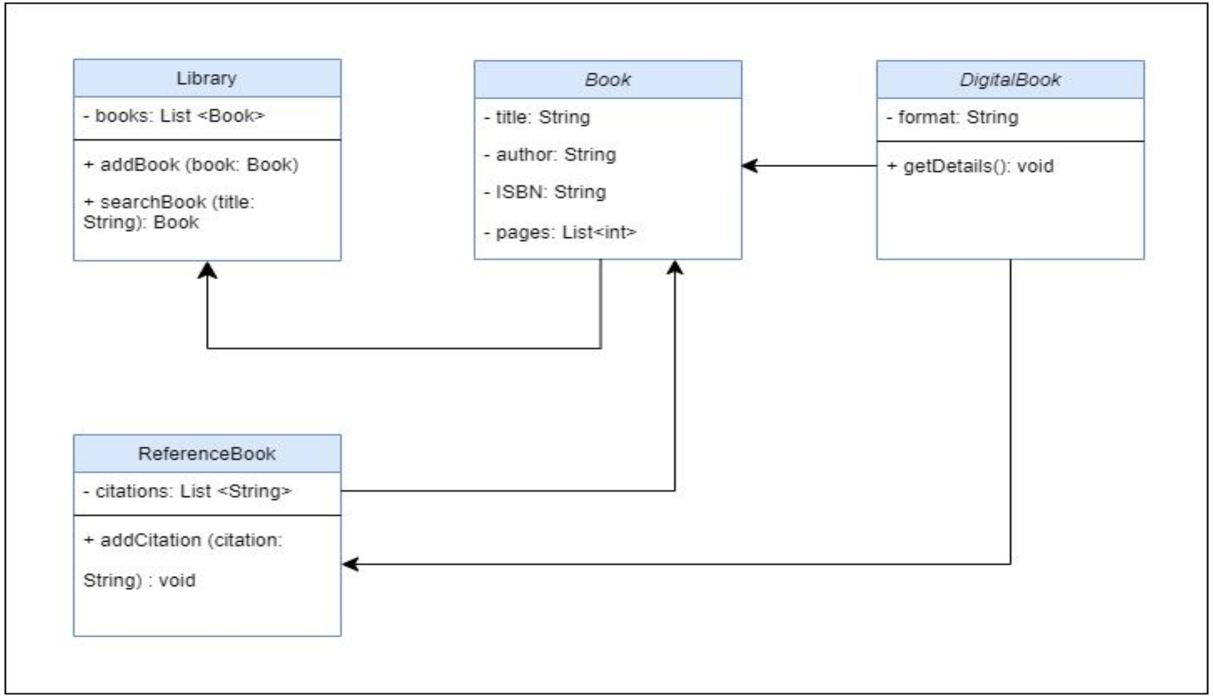
## Code Design

Once we understand the different users, the features they need and how they will interact with the system, we can start working on a design for our code.

**Class Diagrams**

- Take the entities and features and map them to classes
- Each class will handle certain operations that match up the the required features
- Allow us to identify patterns and see opportunities to reuse classes

## Code Design


Library
- books: List <Book>
+ addBook (book: Book)
+ searchBook (title: String): Book
*Book*
- title: String
- author: String
- ISBN: String
- pages: List<int>
*DigitalBook*
- format: String
+ getDetails(): void
ReferenceBook
- citations: List <String>
+ addCitation (citation: String) : void

CoGrammar

## Class Matix

For more insight into what our classes are doing, we can use a class matrix to let us know which CRUD operations are being performed

- Good for documentation
- Makes it easier when implementing the code to know what features are available.
-

## Class Matix

| Entity | Create | Read | Update | Delete |
|---|---|---|---|---|
| Task | X | X | X | X |

## Let’s Breathe

Let’s take a small break before moving on to the next topic.

CoGrammar

## Case Study

We have been approached by a local bank to build a mobile banking application that allows their customers to pay for items in store using their smartphones. The application will connect directly to their banking API to perform key verification operations.

Lets build it!!!

CoGrammar

## Q & A SECTION

Please use this time to ask any questions relating to the topic, should you have any.

CoGrammar

Thank you for joining!