CSC207 Notes

Contents

1 Introduction

Good industry software is typically organized into multiple layers. This includes a good user interface and persistence, interface adapters, use cases, core classes, etc.
Each layer has different public interfaces, which are the set of classes and methods that exposes it to the world. These are often called **application programming interfaces** (APIs).
In understanding software design, it is important to understand what parts of a program need to change when moving to different programs, and overall how you can structure your programs on enabling these features. Being able to additionally appreciate the constraints of particular details and physical constraints (e.g., memory, storage, time) while designing a proper structure for a program is an implication of the programmer's good architecture.

2 Use Cases for Programs

Use cases are actions that the user will want to do in a program.
Generally, we want to consider in any program:

1. What data needs to be represented?
2. What data structure might you use while the program is running?
3. What should happen if the user quits and restarts the program?

2.1 Example: User System

For example, suppose you wrote a program that allows users to:

1. Register a new account (with a username and password)
2. Log into a user account
3. Log out of a user account

They also plan on having different kinds of accounts, but for now we can just start with one.
The above actions represent all of the use cases necessary for the program.
Consider some methods and classes with this program.

1. User class that contains the name and password
2. Super class for if they have different kinds of accounts.
3. A database that stores all of the users and their passwords. Alternatively, it could be stored client-side (especially for logging in/out users). Or, a dictionary of the username and password to store them.
4. Security issues (e.g., having a secure password)
5. Method that ensures that usernames are unique when a user registers. Similarly, making sure that when someone logs in, the username is registered.

2.1.1 Registering a New Account

1. The user chooses a username, and a password (entering it twice).
2. If the username already exists, the system alerts the user.
3. If the two passwords don't match, the system alerts the user.

2.1.2 Logging in Users

1. The user enters a username and password.
2. If the username exists in the system and the passwords match, then the system shows that the user is logged in.
3. If the username does not exist in the database, then the system alerts the user.
4. If the password does not match the username, then the system alerts the user.

2.1.3 Logging Out Users

The system logs the user out and notifies the user.

2.1.4 When Logged Out, Choose New Use Case

After a user has logged out, let the user between different use cases to choose what to do with the program next.

2.2 Design Conundrums

Some things to think about when designing a program:

1. **What is the user interface?**
   Is the program a webpage, a Java application, a Python command-line program, a mobile app, etc.?
2. **How to do data persistence?**
   Data persistence refers to the storing of data (particularly when the app has been closed). The persistence mechanism refers to how or where the data is being stored.
   Hence, we want to ask, will we store data in a text file, database, Google Drive/OneDrive, etc.?
3. How can you design your program so it's easy to move to a new UI?
4. How can you design your program so it's easy to store data to different kinds of storage?
5. How can you design your program so that as much code as possible stays the same when you do things?

Hence, we have the following things to consider for a program:

1. How can we design use cases so they do not **directly depend on the UI** and persistence choices?
   (a) In this way, we can test all of the use cases thoroughly.
2. What are the use case APIs?
   (a) What is the interface for each use case?
   (b) What public methods do we want to provide to call the use cases from the UI?
3. What persistence methods will we need in any storage?

2.3 Some Terminology

1. **Entity** - A bit of data stored in a program (e.g., username, password)
2. **Factory** - AN object that can make a class instance (typically by calling a constructor)
3. **Use Case** - Something a user wants to do with a program
4. **Input Boundary** - The public interface for calling a use case
5. **Interactor** - The class that runs the use case

6. **Repository** - The persistence mechanism (a subclass of the Gateway)
7. **Gateway** - The methods the repository needs to implement for the Interactor to do its job
   This is typically designed with the use case.
8. **Controller** - The object that the UI asks to run a use case
9. **Presenter** - The object that tells the UI what to do when a use case is completed
10. **Model** - A temporary object to pass data between layers
    Also can be considered a 'state'.

3 Basics of Java

Programs is a series of code that can be run to perform a specific action. To run a program, it must be translated from its high-level programming language to a low-level machine language whose instructions can be executed.
Roughly, there are two types of translation in a program: interpretation, and compilation.

3.0.1 Interpreted vs. Compiled

1. Interpreted:
   (a) For example: Python
   (b) Translates and executes one statement at a time.
2. Compiled:
   (a) For example: C
   (b) Translates the entire program (once), and then executes (any number of times).
3. Hybrid:
   (a) Example: Java
   (b) Translates to something intermediate (e.g., in Java, we use bytecode).
   (c) Java Virtual Machine (JVM) runs this intermediate code.

To compile Java, you can use the command line by typing this manually.
Alternatively, modern IDEs do this for you with the help of a build system (which takes care of the build process).

3.1 Computer Architecture

Computer Architecture goes as follows:

$$\text{Hardware} \rightarrow \text{Operating Systems} \rightarrow \text{Applications}$$

In previous courses, we focused on the applications of this architecture.
The operating system (OS) manages the various running applications and helps them interact with the hardware.
The OS works directly with the hardware.

3.1.1 Compiled Applications

3.1.2 Virtual Machine Architecture

3.1.3 Java Architecture

3.2 Types

Python used 'Duck Typing', where types are checked at runtime, and any object with the appropraite capabilities (methods) is legal.
However, this method can lead to hard-to-find bugs if we aren't careful in writing our code. Therefore, we encourage the use of 'type annotations'.

Java tries to catch these errors early, so it can check types before the program can be run. Then, a lot of mistakes or typos can easily be found.

3.3 Defining Classes in Java

3.3.1 Instance Variables

An instance variable is a variable contained in a class. Every instance of a class will have that specific variable.

3.3.2 Constructors

Constructors have the same name as the class and does not have a return type.
A class can have multiple constructors, but their signatures must be different. If you do not define a constructor, the compiler supplies one with no parameters and no body. However, if you define any constructor for a class, the compiler will no longer supply the default constructor.

3.3.3 This

This is equivalent to Python's 'self' instance variable. Its value is the address of the object whose method has been called. So, it can be useful to disambiguate between an instance variable and a parameter with the same name.

3.3.4 Defining Methods

Defining methods is very similar to that of Python's:

1. A method must have a return type declared. Use *void* if nothing is returned.
2. The form of a return statement:

   ```
   return {expression};
   ```

   If the expression is omitted or if the end of the method is reached without executing a return statement, nothing is returned.
3. Must specify the accessibility.
   (a) Public - Callable from anywhere.
   (b) Private - Callable only from that particular class.
4. Variables declared in a method are local to that method.

3.3.5 Parameters

1. When passing an argument to a method, you pas what is inside the variable's box:
   (a) For class types, you are passing a reference (like in Python).
   (b) For primitive types, you are passing a value (Python cannot do this).
2. This has important implications!
   (a) You must be aware of whether you are passing a primitive or an object. This is a similar idea to mutable and immutable objects in Python.
   (b) You may hear people talk about 'pass by reference' and 'pass by value' as you begin to learn more about programming languages.

3.3.6 Instance Variables and Static Variables

```
class Sneetch {
    private String name;
    private boolean starBellied;
    private static int HowMany = 0;
    public static final String SAYING =
                                "Best on the
Beeches.";
}
```

1. Instance Variables: name, starBellied
2. Static/Class Variables: howMany
3. Static and final variable: SAYING
   Because these values cannot be changed, it is okay to later make these public.

You can mix instance variable and class variable declarations with method definitions in any order you like, but try to organize things in ways that make sense.

3.3.7 Instance Variables and Access

If an instance variable is private, how can client code use it? We could make everything public, but we probably want to hide the implementation details.
Your class should provide an abstraction or service (also called an encapsulation). This then provides access to information to a well-defined interface (the public methods of the class).
In encapsulating our code, we can change the implementation - to improve speed, reliability, or readability - and no other code has to change.

3.3.8 Conventions

Conventions make all non-final instance variables either:

1. Private - Accessible only within the class.
2. Protected - Accessible only within the package.

When desired, give outside access using 'getter' and 'setter' methods, rather than making instance variables public.

3.3.9 Access Modifiers

Classes can be declared public or package-private,
and members of classes can be declared as public, protected, package-protected, or private,

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default (package private)	Y	Y	N	N
private	Y	N	N	N

3.4 Inheritance

3.4.1 Inheritance Hierarchy

Similar to Python, all classes can form a tree called the **inheritance hierarchy**, with Object at the root. Class Object does not have a parent, and all other Java classes have one parent. If a class has no parent declared, it is a child of class Object. Class Object guarantees that every class inherits methods *toString*, *equals*, and other methods.
Note that a parent class can have multiple child classes.

3.4.2 Inheritance

Inheritance allows one class to inherit the data and methods of another class.
In a subclass, *super* refers to the part of the object defined by the parent class.

1. Use *super*. << *attribute* >> to refer to an attribute (data member/method) in the parent class.
2. Use *super*(<< *arguments* >>) to call a constructor defined in the parent class.

Suppose that class *Child* extends class *Parent*. An instance of *Child* has:

1. A *Child* part, with all the data members/methods of *Child*.
2. A *Parent* part, with all the data members and methods of *Parent*.
3. A *Grandparent* part, ..., etc., all the way up to *Object*.

Note that an instance of *Child* can be used anywhere that a *Parent* is legal, but not the other way around.

3.4.3 Name Lookup Rules

For a method call expression.method(arguments), Java looks for the method in the most specific (bottom-most) part of the object referred to by the expression. If it is not defined, Java looks ‘upward’ until it is found (else it is an error).
For a reference to an instance variable expression.variable, Java determines the type of expression and looks in that box. If it is not defined there, Java looks upward until it is found (else it is an error).

3.4.4 Shadowing and Overriding

Suppose Class A AND ITS SUBCLASS *AChild* each have an instance variable x and an instance method m.
A's m is overriden by *AChild*'s m (which is usually a good idea),
and A's x is shadowed by *AChild*'s x (which can be confusing).

If a method must not be overriden in a descendant, declare it $final$.

3.5 Java Conventions

1. Javadoc: For commenting:

   ```
   /**
   * Comment 1
   * Comment 2
   */
   ```

2. camelCase is used, rather than pothole_case.
3. Class name: A noun phrase starting with a capital.
4. Method name: A verb phrase starting with a lower case.
5. Instance variable: A noun phrase starting with a lower case.
6. Local variable or parameter: Noun starting with a lower case; acronyms and abbreviations are okay.
7. Constant: All uppercase, pothole case.

3.6 Interfaces

Classes can define methods without giving a body; these methods are then called **abstract**, and the class must also be called **abstract**. Child classes can implement some or all of the inherited abstract methods. When a class is completely abstract, we can call it an **interface**:

1. It only has method signatures and return types.
2. Guarantees capabilities.

A class can be declared to implement an interface, defining a body for every method of the interface. Interfaces can also extend other interfaces.

3.7 Generics

Generics allow for fancier usage of type parameters. For example:

```
public interface Comparable<T> {
    /**
     * Compare this object with o for order.
     * Return a negative integer, zero, or a
     * positive integer as this object is less
     * than, or equal to, or greater than o.
     */
    int compareTo(T o); // No body at all.
}

public class Student implements Comparable<Student> {
    ...
    public int compareTo(Student other) {
        ...
    }
}
```

1. ‘class Foo¡T¿’ introduces a class with a type parameter T.
2. ‘¡T extends Bar¿’ introduces a type parameter that is required to be a descendant of the class Bar – which Bar itself a possibility.
   In a type parameter, ‘extends’ is also used to mean ‘implements’.

3. '¡? extends Bar¿' is a type parameter that can be any class that extends Bar.

4. '¡? super Bar¿' is a parameter that can be any ancestor of Bar,

We typically use K, V for naming of maps, X for exceptions, and T otherwise.

3.8 Interfaces and Inheritance

Interfaces are similar to abstract classes, defining types and the names of methods/attributes. However, they do not have any implementation details.
Any class that implements an interface is required to define the interface's methods and attributes.

3.8.1 Services and Interfaces

If your program provides a service for a class, then that class must have the correct methods for that service. To ensure these methods are implemented, the class implements an interface.

Inheritance and is-a Relationships We use inheritance when we know there will be an overlap between the functionality of a parent class and their child. This allows us to reuse code without rewriting it.
Subclasses should have an 'is a' relationship with their parent class.
If Type A 'is a' B, then class A should extend the Account class.

3.8.2 Inheritance versus Interfaces

Consider a program that present the user with an adventure game with the following functionalities:

1. When the user plays the game, they encounter the magical objects, some of which they can take and some of which are used as soon as they are picked up.

2. If you find a +1 sword, your character can keep it and use it later.

3. If you find a magic potion, you have to use it immediately or leave it.

If using **inheritance**, we can define an abstract class, where there are many subclasses, but only some subclasses contain a 'collect' method.

If using **interface**, we implement classes that have a 'collect':

1. The sword class extends MagicItem and it would implement Collectable.

2. The potion class extends MagicItem but not implement collectable.

In this way, we can have other Collectable objects in the game but not necessarily MagicItem objects can be collected.

4 Version Control

Version control is a master repository of files that exist on a server. People can clone the repository to get their own local copy.
As significant progress is made, people can push their changes to the master repository, and pull other people's changes from the master repository. It keeps track of every change, and people can revert to older versions.

Why use version control?

1. Useful for backup and restoring files.

2. Synchronization, so that multiple people can make changes of the same repository at once.

3. Short term and long term undos, to fix different changes.

4. Track changes, which is especially useful when trying to trace bugs.
5. Sandboxing, trying out different things without messing up the main code.
6. Branching and merging, allowing people to work on and try different things at once.

5 Clean Architecture

Architecture of code involves the design of the system, dividing it into logical pieces and specifying how those pieces can communicate with each other.
The goal with architecture is to facilitate the development, deployment, operation, and maintenance of the software system, leaving as many options as possible, for as long as possible.

5.0.1 Policy and Level

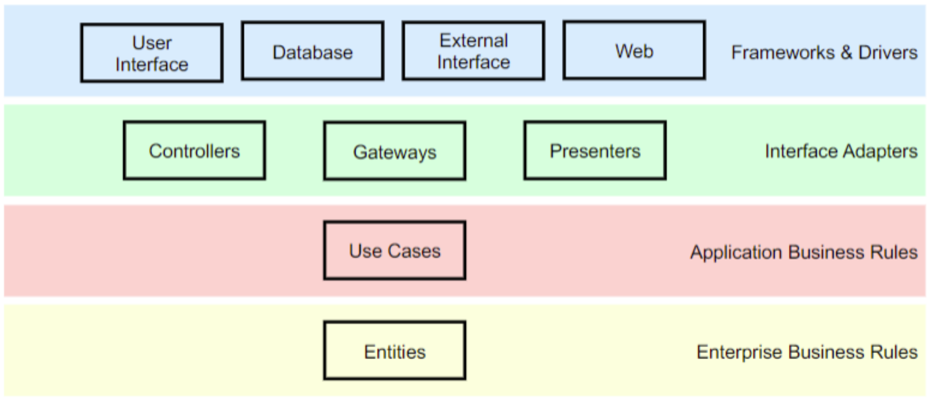
A computer program is a detailed description of the **policy** by which inputs are transformed into outputs. Software design seeks to separate policies, and group them as appropriate.
Each policy has an associated level (i.e., high, low), which signifies the distance between inputs and outputs. Entities are considered the highest-level policies (the core of the program).

5.0.2 Business Rules

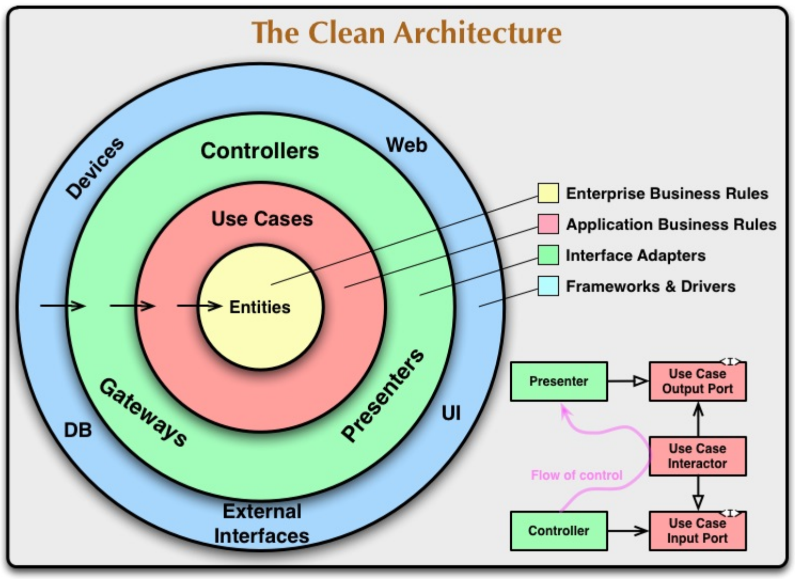
1. **Entities -** An object within the computer system that embodies a small set of critical business rules (**methods**), operating on critical business data (**variables**).
   For example, loans in a bank, players in a game, sets of high scores.

2. **Use Cases -** A description of the way that an automated system is used. It specifies the input for the user, the output is returned to the user, and the processing steps work on creating that output. That is, use cases manipulate entities to return outputs to users.
   A use case describes **application-specific business rules**, as opposed to the critical business rules within entities.
   Note that use cases do not know about user interface or the data storage mechanism.

5.1 Visualizing Clean Architecture



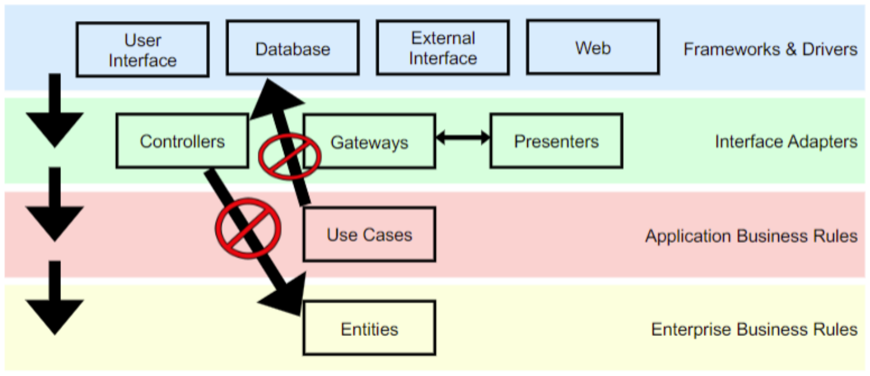
Oftentimes, we use concentric circles to visualize these layers, where entities are the core, and the input/output are in the outer layers.



5.1.1 Dependency Rule

1. All dependencies must point inwards.
2. Dependencies within the same layer is allowed (but try to minimize).
3. You'll definitely want at least 3 layers, but sometimes more than 4.
4. The name of something (functions, classes, variables) declared in an outer layer must not be mentioned by the code in an inner layer.
   To go from the inside to outside:
   (a) Use dependency inversion. An inner layer can depend on an interface, which the outer layer implements.

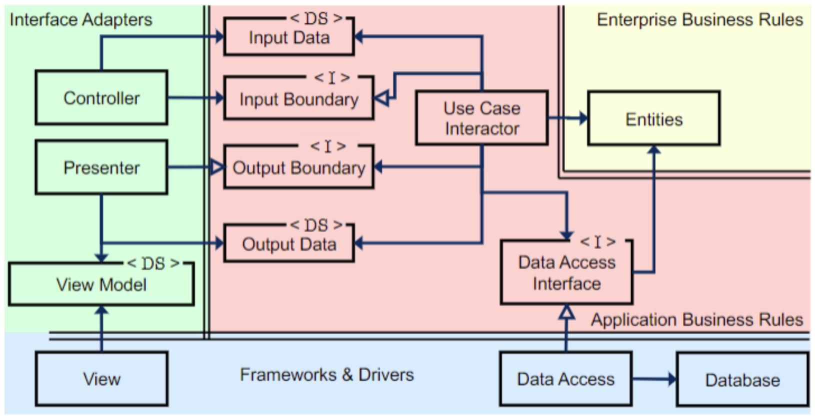
Let the arrows signify dependence:



5.2 Benefits of Clean Architecture

1. All the details (frameworks, UI, databases, etc.) live in the outermost layer.
2. The business rules can be tested easily without worrying about those details in the outer layers.
3. Any changes in the outer layers don't affect the business rules.

5.3 Example



1. ¡DS¿ indicates the info the Interactor needs:
   (a) A class with basic instance variables: the raw data (strings and numbers) from the user interface.
   (b) Transient: IT is created, passed in, unpacked by the use case, and then discarded.
2. ¡I¿ is for how the Controller starts the use case:
   (a) It is an interface with a method.
   (b) One of the parameters is the Input Data.
3. The Use Case Interactor:
   (a) Implements the Input Boundary interface.
   (b) Has the primary method specified by that interface.
   (c) Unpacks the input data and manipulates entities.

6 CRC Cards

CRC stands for Class, Responsibility, and Collaboration.

1. Class: It is an object oriented class name, and includes information about super and sub-classes.
2. Responsibility: It is what information the class stores and what the class does. It involves the behaviour for which an object is accountable.
3. Collaboration: It is its relationship to other classes, that is, the other classes this class uses.

CRC cards are part of the object-oriented development paradigm, creating the initial set of classes and their relationships, allowing to plan the 'what' of your function easily.

They are beneficial since they are quick and easy to make, and force you to be concise and clear with what is required of your class.

6.1 CRC Models

CRC Models are a collection of CRC cards that specify the object-oriented design of a software system.

6.1.1 Creating a CRC Model

1. Read the specifications (description of the requirements) for the software system.
2. Identify core classes (typically they are nouns in a description). Add responsibilities to each of these core classes.
3. Identify any other classes that this class needs to work with in order to fulfill its responsibilities. These are the **collaborators**.
4. Add other classes, and identify abstract classes, inheritance, etc.
5. Refine the set of classes by removing unnecessary classes and adding beneficial classes.

6.1.2 Testing CRC Models

We can use a scenario walkthrough to test if our CRC model works. We simply choose a scenario and a plausible set of inputs.
Then, we manually execute the scenario; take the initial input and find the class that has responsibility for responding to the input. Then, trace through each of the collaborations, and adjust the cards until the scenario has been stabilized.

6.1.3 User Stories and Use Cases

Depending on how one states the scenario walkthrough for the CRC model, they get different aspects of the problem. Framing a particular problem around a specific area allows us to tackle how different perspectives of our program will work:

User Stories are informal, general explanations of a software written from the perspective of the customer. They are typically considered the 'end goal' of software.
The use cases are used in order to test these user stories and ensure that the end product is achieved using the CRC model.

7 SOLID Design Principles

The SOLID principles are the fundamental basic principles of object-oriented design, developed by Robert C. Martin.

1. SRP - Single responsbility principle
2. OCP - Open/closed principle
3. LSP - Liskov substitution principle
4. ISP - Interface segregation principle
5. DIP - Dependency inversion principle

7.1 SRP

Every class should have a single responsibility; and so, they should also have one reason to change.
It is important to isolate modules from the complexities of the organization as a whole, and design systems such that each module is responsible and responds to the needs of one business function.

Making different classes can make keeping track of objects difficult, and so we create façades, delegating multiple classes together into one.

7.2 OCP

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification. In other words, we should add new features by extending and adding new behaviours or adding plugin capabilities, but not by modifying the original class.

7.3 LSP

Programs that use an interface must not be confused by an implementation of that surface.
That is, if S is a subtype of T, then objects of type S can be substituted for objects of type T, without altering any of the desired properties of the program. Then, S would be a child class of T - or, S implements interface T.

7.4 ISP

When we consider the interface (public methods of a class) of a program, it is better to use small specific interfaces, rather than a few larger ones. In this way, it is easier to extend and modify the design.

7.5 DIP

When building complex systems, we use abstraction layers between low-level classes and high-level classes that reflect the flow of control and source code dependencies.
We want to decouple systems so that changes to individual pieces do not affect any other pieces:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend upon details; Details should depend upon abstractions.

To invert the source code dependency, we can introduce an interface such that the flow of control can stay the same.
Suppose we had two modules A, B that have a source code dependency (i.e., A depends on B). We modify our implementation so that A depends on the interface, and B then implements that interface. Then, there will no longer be a source code dependency.