Software Engineering 2 (2022/2023)

Questions im planning to do -

Question 1 -Software Testing, Design Patterns

Question 3 - Observer design pattterns about restaurant software

Question 4 - DbC and OCL statements

Question 5 - Six Key Practices of Agile Methodology XP and Anticipatory Design VS Refactoring

Question 1

1. (a) Briefly describe and distinguish between the following types of software testing:

- **unit**
- **integration**
- **functional**

In your answer, make reference to white-box and black-box testing and mention who is responsible for writing the different levels of tests in a software development project

→ The types of software testing are as follows →

- Unit Testing - This allows to test individual units of software such as the functions. modules or classes. It focuses on internal logic and functionality of the individual unit. It is performed by developers instead of testers as it involves a more in depth knowledge of the code written. It also involves white-

box testing where the tester also has access to the source code and can examine its logic.

- Integration Testing - Allows to test different software units interact with each other to form a cohesive system. The main focus is on interfaces and data flow between units. It can involve both white-box testing(used to examine how units interact internally) or black-box testing(used to treat units as standalone components and main focus is on external interactions)
- Functional Testing - Tests overall functionaility of the software from the user's perspective. It focuses on requirements and specifications to ensure the software delivers on the intended features. It usually involves black-box testing where the tester interacts with the software's user interface without knowing internal code - this testing can be manual or automated.

(**b) Describe the Adapter design pattern and distinguish between the class adapter and object adapter approaches.**

→ Adapter design pattern is a structural pattern that allows using an existing class(or adaptee) with an incompatible interface to work with a desired interface(target interface). It acts as a bridge between these interfaces - enabling collaboration betweeen otherwise incompatible objects. The approaches for implementing adapter design patterns are →

- Class Adapter - Inherits from the target interface and composes the adaptee class. It adapts the functionality of the adaptee's methods to comform to the targets interface
- Object Adapter - Implements the target interface and holds an instance of the adaptee class. It delegates the method calls from the appropriate methods of the adaptee and potentially with the adaptations. It is much more flexible as it can adapt any class to the target interface- not requiring inheritence

from gemini hehe →

Feature	Class Adapter	Object Adapter
Inheritance	Inherits from target interface	Implements target interface
Composition	Composes the adaptee class	Holds an instance of the adaptee
Use Case	Close relationship between target and adaptee, or multiple classes adapting to the same target	Adapting any class to the target interface
Flexibility	Less flexible	More flexible

Question 3

3. You are required to do some object-oriented design for a standalone restaurant software system that mainly manages bookings. The restaurant software should be able to handle advance reservations, walk-in bookings, assigning tables to reservations and so on.

(a) As part of the design for the restaurant software it was decided to control access to the underlying functionality by creating a single instance of the control class BookingSystem and passing all requests through it. What design pattern is implicit in this? Provide some code fragments for the class BookingSystem showing how this design idea can be realised.

→ In order to create a single instance of the control class BookingSystem and pass requests through it - we have to use a design pattern called Singleton Design Pattern. Singleton Design states as follows →

- There should be only one instance allowed for a class
- We should allow a global point of access to that instance
- Multiple instance can be allowed in the future without affecting singleton class'clients.

This design idea is advantageous and simple in theory yet its slightly complicated to implement in Java. Some code segments if this were to be implemented in Java for the restaurant software system, would look as follows (from gemini lol)→

```
public class BookingSystem {
    private static BookingSystem instance; // Private static
variable for single instance
    private List<Table> tables; // List to store available ta
bles
    private Map<Integer, Reservation> reservations; // Map to
store reservations with ID as key

    private BookingSystem() {
        // Initialize tables and reservations (replace with y
our data loading logic)
        tables = new ArrayList<>();
        tables.add(new Table(1, 4));
        tables.add(new Table(2, 2));
        reservations = new HashMap<>();
    }

    public static BookingSystem getInstance() {
        if (instance == null) {
            synchronized (BookingSystem.class) { // Thread-sa
fe initialization
                if (instance == null) {
                    instance = new BookingSystem();
                }
            }
        }
        return instance;
}
```

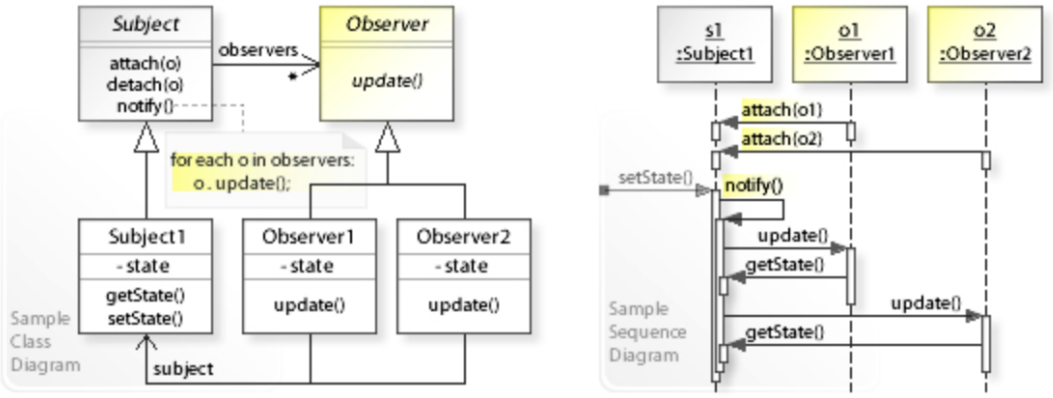
(b) Describe the structure and purpose of the Observer design pattern and in which generic situations it might be applicable.

→ Observer design pattern establishes a one-to-many dependency between objects in a program. The structure of the patttern is mainly an object(called Subject) that keeps track of all the dependent objects(or called Observers). It also allows an interface to attach or detach observers. It notifies the dependent objects

about state changes as well. The observers also each contain an update() operation in order to be notified of the state changes. The subject and the observers are loosely couped. The observers can be added or removed in run time without causing any issues. The purpose of Observer design pattern is to add flexibility to code as well as make it easier to maintain. One scenario in which this pattern can be incredibly useful is when UI and its components need to react to changes in the data or system state. Since the subjects and observers are loosely coupled - this allows for a very quick reflections of a state change. It can also be helpful for creating event driven systems where when certain events take place other components react accordingly.

(c) Show and explain in writing, with the aid of a package/class diagram and a sequence diagram, how the Observer pattern could be incorporated into the design of the restaurant system to couple application code with user interface code.

→ not sure but I found the class and sequence diagram for the observer deisgn pattern in Richards notes(aka a wikipedia page :)))



The Subject class does not depend on any of the Observers. In order to update the state of the Observers - it refers to the update() function rather than updating the state of the Observers directly. This can be incorporated in the restuarant system in the following scenarios →

- A booking has been made and the customer needs a verification for the booking

- A customer has ordered and the system in the kitchen needs to updated for the respective order

All these changes are required in real time and using the Observer design pattern the Subject's state for any component can be synchronised and updated for the Observer's states as well.

Question 4

(a) Explain what is meant by Design by Contract (DbC). Elaborate on how a contract is affected by subclassing/polymorphism.

→ Design by Contract or Programming by Control is an approach to designing software. It states that deisgners should define precise and verifiable specifications for software components - using preconditions, postcoditions and invariants. These specifications are also called 'contracts'. DbC also uses assertations(which is a Boolean statement that should never be false). Contracts are affected by subclassing and polymorphism. They are affected in different ways. Invariants and post-conditions are example must be true for all the subclass which makes them more restrictive. In contrast - pre-conditions are not allowed to be strengthed because it can weaken a post-condition. Since contracts are very important for code - DbC can ensure code quality and consistency is maintained by knowing how contracts are affected by inheritence and polymorphism.

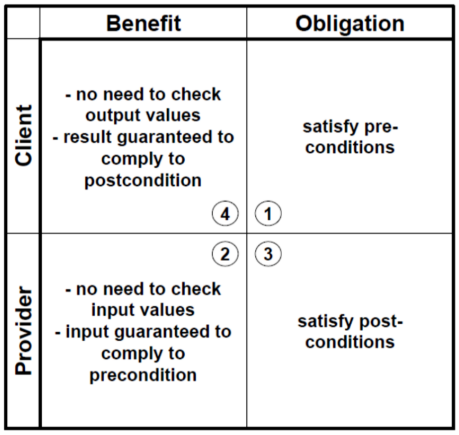
(b) Within the context of DbC, comment on benefits and obligations for both client code and provider code. Mention when exceptions might be appropriate.

→ Client code is when code is calling a method. Provider code is the code implementing the method. The benefits and obligations of client code are as follows →

- No need to check output values - the likelihood of unexpected errors is a lot lower.
- Result complies to postcondition - By checking pre-conditions it can ensure that the data is in the expected state and form for the post condition.
- The obligation for client code is for it to satisfy the pre-conditions - which allows the client code to be easier to maintain and is more reliable.

The benefits and obligations of provider code are as follows →

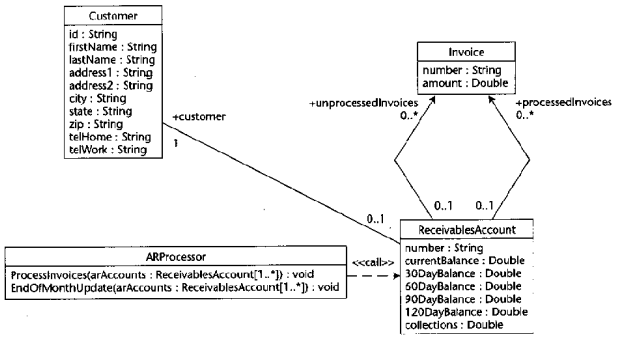
- No need to check input values - since preconditions are enforced, the errors are caught very early on
- Input complies with the preconditions - the input has to comply with the pre-conditions of the code and if it doesn't it can be isolated easily after method calls.
- The obligation for the provider code is for it to satisfy the post conditions - which allows the method to delivery the expected results with the correct input.



The exceptions arise when the pre-conditions are not met. Since the preconditions establishes what the provider expects- any conditions that are not met can be easily found early and will prevent the method from executing with wrong data. Client code is informed about the issue. This is what the exception does.

(c) Given the class diagram below, write an Object Constraint Language (OCL) contract invariant for ReceivablesAccount which states that no invoice can be in both *processedInvoices* and *unprocessedInvoices* collections at the same time.

Write an OCL contract that you deem appropriate to express the business logic *ProcessInvoices()* operation of the class *ARProcessor*



→ The OCl contract for the invariant for ReceivablesAccount which states no invoice can be in processedInvoices and unprocessedInvoices collections at the same time is as such →

```
context ReceivablesAccount inv:
    unprocessedInvoices→intersection(processedInvoices)→isEmpty()
```

The OCL contract to express the business logic ProcessInvoices() operation of the class ARProcesser is as such →

```
context ARProcessor
    def ProcessInvoices(arAccounts: ReceivablesAccount[1..*]): void
    context ARProcessor::ProcessInvoices (arAccounts : Set(ReceivablesAccount)) pre:
        arAccounts→forAll(unprocessedInvoices→notEmpty())
    context ARProcessor::ProcessInvoices (arAccounts : Set(ReceivablesAccount)) post:
        arAccounts→forAll
            (
                unProcessedInvoices→isEmpty() and
                processedInvoices→includes(unprocessedInvoices@pre)
            )
```

Question 5

(a) Describe six of the key practices of the agile methodology XP.

→ The six key practices of the agile methodology XP are as follows →

- Pair Programming - Allows 2 or more engineers to work side-by-side to develop code togrther. This approach is deisgned to optimize quality due to the built in validation that is expected between the engineers.

- Planning Game - It uses the approach of planning work through small batches of work frequenty and with a consistent schedule. Also called Iteration Planning. This also uses a technique called “user stories” which is an important aspect of XP
- Continuous Process - The code is built and released iteratively which enables the team to assess the state of the product and resolve issues as early and as quickly as possible which results in the program having much better quality
- Coding Standards - Rules and coding standards are essential to maintain consistency and decrease the likelihood of defects and issues in the program being developed. These set practices and rules can also needed to reduce confusion between the engineers
- Sustainable Pace - In order to respect work-life balance for teams and ensure high morale and product quality - XP encourages sustainable developmenet. This allows the team to work more efficiently even if the time being spent to develop is not very high.
- Test Driven Development(TDD) - XP insures that unit tests are written prior to writing cde. This makes sure the code passes the unit tests before its integrated with other elements. It increases code quality.

(b) Discuss the diagram below from the point of view: Anticipatory Design versus Refactoring.

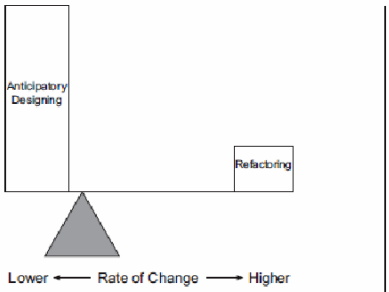


Figure 3 — Balancing design and refactoring, pre-internet.

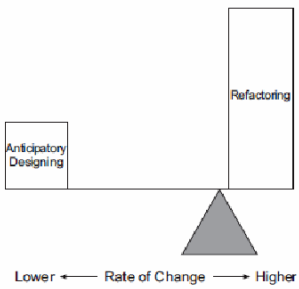


Figure 4 — Balancing design and refactoring today.

→ Anticipatory design is what produces extra facilities for future requirements which allows a product to evolve. Refactoring is the ongoing redesgin of software to improve its responsiveness to change. This approach can also be thought of as continuous design. Reactoring according to Martin Fowler is changing a software

system in such a way that it does not alter the external behaviour of the code yet improves its internal structure.

The figures are different in the following ways →

- Figure 3 is a diagram based on time before the invention of the internet. Before the internet the rate of change and refactoring was much lower. However there was much more anticipation for the future(anticipatory design). Future modifications and requirements were essential to the product. The developers perhaps used much more complex architecture in hopes to integrate these future plans. On the other hand refactoring was less emphasised as the main focus was to make the solid foundation of the product. Certain refactoring tools also might have not been as prevelent.
- Figure 4 is a diagram based on time after the invention of the internet. The ability to update the software much more frequently based on user feedback is now crucial. This would reduce the need for anticipatory design. The approach is more so to design for the current needs and implement them. In contrast, refactoring became more valuable as it allowed developers to adapt existing code. The tools for refactor software are very advanced now. Refactoring allows companies to keep their software “up to date” with all the evergrowing changes.