Software Engineering (Spring 2024)

Midsem Exam Answer Key

A. Mayush - Correct options: **b - 2.5**

Option b - Information experts suggest the class with the most knowledge to perform some functionality will have that functionality encapsulated into it. Here, we have a ‘calculate’ function in OrderCal which requires the Order object, so the information expert principle is violated.

Option c - 1.25 partially correct

B. Ayush

Correct options: **b(1mark), c(1mark), d(0.5 marks)**

-0.5 if a option is chosen

Explanation:
Option b: Halstead metrics will potentially indicate an upward trend in program length.

Halstead metrics will potentially show an upward trend as a bigger program corresponds to more operands and operators which directly increases the program length metric.

Halstead Program Length: Nlogn, where
N = N1 + N2
n = n1+n2

n1 -> number of distinct operators (+, -, *, while, for, (), {}, function calls, etc.)
n2 -> number of distinct operands (variables, method names, etc.)
N1 -> total number of occurrence of operators

N2 -> total number of occurrence of operands

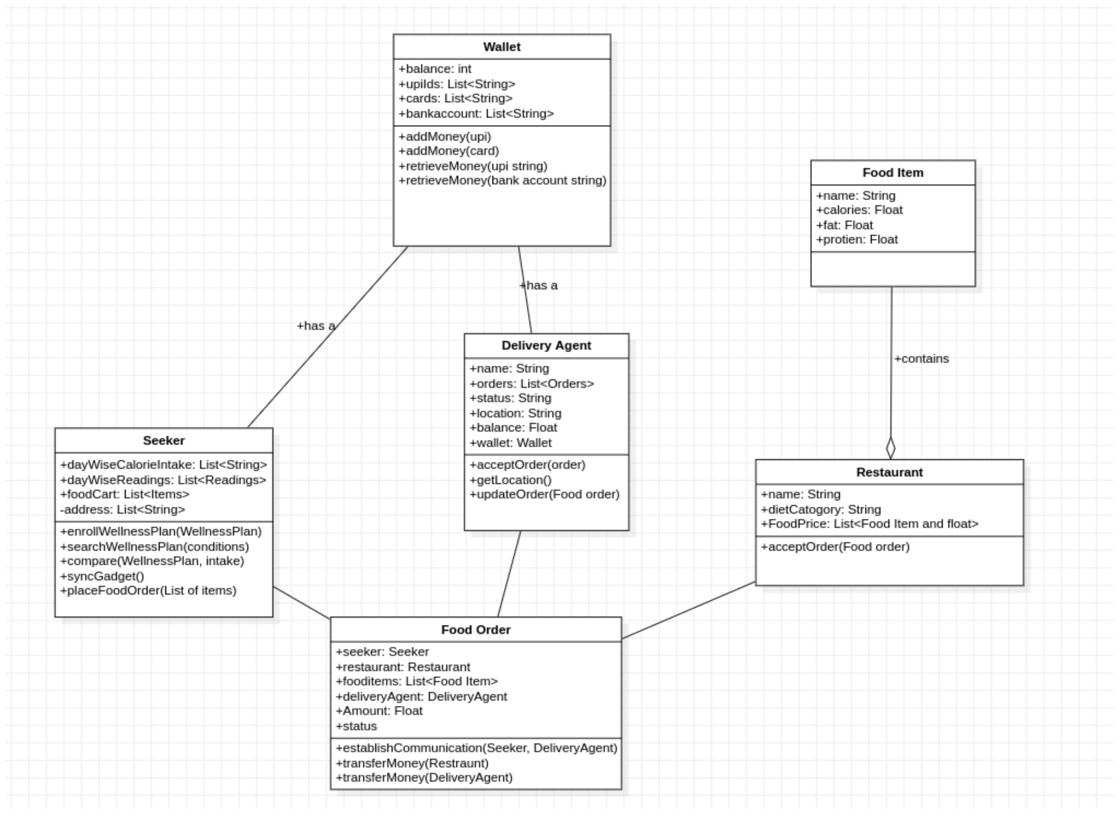
Option c: The number of conditional switches will indicate higher cyclomatic complexity

Cyclomatic complexity measures the number of linearly independent paths that can be taken through the code. Each case in the switch statement adds to the cyclomatic complexity of the program.

Option d: The lines of code also indicate a need for refactoring

The HealthMonitor class is a large class with 400 lines of code. A class as large might be violating the single responsibility principle (SRP), may contain complex logic and duplicated code. Therefore, it indicates a need for refactoring.

C. Venu



- Identifying 4-5 correct classes that capture the actual subsystem correctly (3 marks) and their descriptions (1 mark)
- Correct relationships between these classes (2 marks)
- Sequence diagram based on correctness (2 marks) and it's relevance with class diagram (2 mark)

Note: The above uml diagram is just one possible answer.

D. Ankith

- Code smells include long method, long parameter list among many others
- Design smells: Lack of modularization (modularization smell).
- As far as code smells, refactoring is considered. If they have at least explained how the refactoring needs to be done, provide marks. Full marks on this section to be provided if corresponding refactored pseudocode is provided.
- For design refactoring, one of the key solutions could have been to use the builder pattern by introducing interface and a builder class. But even if some people have refactored in a different way and justified how it brings more modularity compared to the initial design, provide marks.
- Grading the question:
    i. Part A (identifying design and code smells)(3 Marks):
        1. Code Smells (at least 2 of the 4 or any other valid smell) (2 Marks):
            a. **Feature Envy:** The customizeWellnessPlan method seems to be more interested in the DietPlan object's data (meals) than in the WellnessPlan itself. This could indicate that the method should belong to the DietPlan class instead.
            b. **Primitive Obsession:** Passing multiple string parameters (dietPlanName, dietCategory, workoutPlanName, etc.) could be replaced by a more structured approach, like passing objects representing these plans.
            c. **Long Parameter List:** The constructor and method customizeWellnessPlan have long parameter lists, which can make the code harder to understand and maintain.
            d. **Conditional Complexity:** The customizeWellnessPlan method has multiple if-else conditions based on the category parameter, which can make the method hard to understand and maintain.

2. Design Smells (either of the 2 or any other valid smell) (1 Mark):
    a. **Broken modularization**: The customizeWellnessPlan method seems to have responsibility beyond what is expected in a WellnessPlan class. It deals with customizing diet plans based on certain categories, which might be better handled in the DietPlan class itself.
    b. **Insufficient modularization**: The WellnessPlan class might not be completely decomposed, as it contains logic for creating and customizing diet, workout, and mental well-being plans. Each of these could potentially be its own class, leading to better modularization and easier maintenance.

ii. Part B (affected qualities) (2 Marks):

1. Any 2 of the below or any other valid qualities. 50% marks for identifying qualities and explanation each.
    a. **Extensibility:** Adding new types of plans or customization options may require modifying multiple parts of the class, violating the Open/Closed Principle.
    b. **Readability:** The code is less readable due to the long parameter lists and complex conditional logic.
    c. **Maintainability:** Changes to the customization logic or plan types may require modifications in multiple places, increasing the chance of errors and making maintenance harder.

iii. Part C (refactored pseudocode and UML Diagrams) (5 Marks):

1. For Code Smells, pseudocode should follow below suggestions (partial marks for just describing how to refactor, full marks for refactoring)(3 Marks: 2*1.5):
    a. **Feature Envy:** Move the customizeWellnessPlan method to the DietPlan class, as it mainly deals with diet customization.

   b. **Primitive Obsession:** Replace string parameters with objects representing plans, categories, and activities to encapsulate related data.
   c. **Long Parameter List:** Use the Builder Pattern or a parameter object to reduce the number of parameters in the constructor and method.
   d. **Conditional Complexity:** Consider using polymorphism or a strategy pattern to handle different customization options more elegantly.
2. For Design Smells, UML diagram should follow any 1 of the below patterns (partial marks for just describing, full marks for UML)(2 Marks, 1 design pattern is enough):
   a. **Builder Pattern:** It can be applied to construct WellnessPlan objects, especially when there are multiple parameters involved in the construction process.
   b. **Factory Pattern:** It can be used to create instances of DietPlan, WorkoutPlan, and MentalWellBeing based on certain criteria.
   c. **Strategy Pattern:** It can be applied to define different strategies for customizing diet, workout, and mental well-being plans based on user preferences or health goals.

E. Siddharth

- The idea was to use an observer pattern, factory pattern was also there?
- Even if someone has mentioned “observer pattern” some marks can be provided.
- If the full class diagram with some explanation is provided using observer pattern then full marks

- For the quality, if for each of them some relevant explanation is provided, consider giving marks.
- Grading the question:
    i. Part A (Creating UML diagram/pseudocode (5 Marks):
        1. **Observer Pattern (5 Marks):** To notify fitness enthusiasts about new restaurants or menu items. The fitness enthusiasts can subscribe to specific restaurants or categories and be notified when new items are added.
        2. Factory pattern + Observer is also fine. Deduct 2 marks if just the observer pattern is used and the overall design consists of design smells.
        3. Deduct marks for missing out important classes and relationships
        4. No pattern & correct UML => 2.5 marks
        5. No UML & mentioned observer pattern, give 1 mark

    ii. Part B (Assumptions and description of of classes) (3 Marks):
        1. Any valid assumptions are acceptable
        2. Give 3 marks if relevant descriptions for all important classes are available

    iii. Part C (Extensibility, Changeability, Understandability) (2 Marks):
        1. Any 2 of the below points or other relevant points are fine. (2*1)
        2. **Extensibility:**
            a. Allows the addition of new product types without modifying existing code.

    b. Clients can use the factory interface to create objects without being aware of the specific implementation details.
    c. Allows new observers to be added without modifying the subject or other observers.
    d. Makes it easy to add new functionalities that react to changes in the subject.
3. **Changeability:**
    a. Centralizes the object creation process, so changing the way objects are created or initialized only requires modifying the factory implementation.
    b. Client code remains unaffected by changes in the object creation process.
    c. Decouples the subject from its observers, so changes in notifications or the number of observers can be made independently.
    d. The subject or observer implementations can be modified without affecting each other.
4. **Understandability:**
    a. Provides a clear separation of concerns.
    b. Clients only need to know about the factory interface and the products it creates, without needing to understand the complex object creation logic.
    c. Clearly defines the interactions between subjects and observers.
    d. Each observer is responsible for a specific aspect of the subject's state, enhancing the code's understandability and maintainability.