SENG 275 MID TERM EXAM 3 (10%) Solution-O6

Instructor: Dr. Navneet Kaur Popli, Date: 23 Mar 2023, Time: 11:30-12:20 PM PST
Mode: Pen-Paper, Synchronous, Timed, closed book, to be done in the question paper
Total Marks: 31, Total Pages: 5, Total questions
Note: This is an individual activity. Copying or cheating of any kind is not allowed.

Q1) Consider a Travel website using a UserService. The UserService is responsible for adding, deleting, and modifying user records. It is also responsible for travel planning for the users. The UserService keeps the user data in a UserRepository. The UserRepository is implemented using MongoDB and is implemented in the following manner: (Total 12M)

```
public class UserService
{
        private UserRepository ur;
        public UserService()
                { ur=new UserRepository();}
        ur.addUser();
}
```

A tester looks at the code and points out that this code has low testability.

a) Why does the tester think this code is less testable? Explain. (2M)

   This code is less testable because there is tight coupling between the UserService and the UserRepository. The instance of the UserRepository is created inside the UserService therefore both UserRepositoryand UserService cannot be tested independently or in isolation of each other.

b) The tester also says that the testability can be increased using dependency injection and inversion of control. What are these two concepts and explain how they increase testability in an application.? (2+2=4M)

   Dependency injection: the entities should not have dependencies as the concrete implementations, instead they should be injected from outside.

   The entities will get these dependencies when we inject them through constructor or setter() methods.

Inversion of control: create the dependencies first and then create the instance of the entity that will be used in those dependencies.

This makes our code loosely coupled, and eventually easily testable.

Also, dependencies can be easily replaced by mocks.

c) Refactor the above code to implement the dependency injection concept. (2M)

```
public class UserService
{
    private UserRepository ur;
    public UserService(UserRepository ur)
        { this.ur=ur;}
    ur.addUser(); ur.userPlan();
}
```

d) Explain how this refactoring increased the testability of the code? (2M)

By injecting the dependency in the constructor of the code and not having it inside the class as a concrete implementation, the testability increases because the UserService and the UserRepository can be tested in isolation and the UserRepository dependency can easily be mocked.

e) How does this refactoring improve the controllability of the test for the UserService? (2M)

This refactoring improves the controllability of the test for the UserService because we can mock and thus control the UserRepository dependency and thus increase the controllability of the test.

Q2) Consider the java code for an e-commerce application with various user, product, order, and other functionalities. Identify any two technical debts in the code and explain ways to reduce those debts. (4M)

```
public class ECommerceApplication {
    // User-related functionalities
    public void createUser(String username, String password) {
        // Code to create a new user in the database
        // ... (database operations)
    }

    public User getUserById(int userId) {
        // Code to fetch user from the database
        // ... (database operations)
        return user;
    }
```

```
    // Product-related functionalities
    public void createProduct(String productName, double price) {
        // Code to create a new product in the database
        // ... (database operations)
    }

    public Product getProductById(int productId) {
        // Code to fetch product from the database
        // ... (database operations)
        return product;
    }

    // Order-related functionalities
    public void createOrder(int userId, List<Integer> productIds) {
        // Code to create an order for the user in the database
        // ... (database operations)
    }

    public Order getOrderById(int orderId) {
        // Code to fetch order from the database
        // ... (database operations)
        return order;
    }

    // Other functionalities...
}
```

A2)The class tightly couples these functionalities with database operations and lacks proper separation of concerns. As the application grows, adding new features or fixing bugs becomes challenging due to the code's tangled nature.

Issues and Technical Debt:

1. **Tight Coupling**: The class combines unrelated functionalities, leading to high coupling between different parts of the application. This makes it difficult to change or update one part without affecting others.
2. **Lack of Modularity**: The monolithic design lacks proper module separation, making the codebase harder to maintain, understand, and navigate.
3. **Scalability**: As the application grows, the monolithic design can become a bottleneck, limiting the application's scalability and performance.
4. **Code Duplication**: Without proper modularization, code duplication can occur when similar functionality is implemented multiple times in different parts of the application.

Addressing Technical Debt:

To address the technical debt and improve the maintainability of the application, you can refactor the code by adopting a more modular architecture, such as a microservices-based approach. Here are the steps you might take:

1. **Identify Microservices**: Identify distinct functionalities and extract them into separate microservices. For example, create separate services for users, products, and orders.
2. **Use Dependency Injection**: Refactor the code to use dependency injection to decouple components and enable better unit testing.
3. **Encapsulate Data Access**: Separate database operations into dedicated Data Access Objects (DAOs) or repositories to abstract the database interaction.
4. **Implement API Gateways**: Use API gateways to provide a unified entry point for clients to interact with the microservices.
5. **Containerization and Orchestration**: Consider containerization using tools like Docker and orchestration platforms like Kubernetes to deploy and manage microservices.

By refactoring the monolithic application into a microservices-based architecture and addressing tight coupling, the code becomes more maintainable, scalable, and easier to evolve over time.

Q3) Consider the following java code which reads from a file. Identify what is causing the resource optimism smell in the code. Also suggest one way in which this smell can be removed. (2M)

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class ReadingFromFile {
public static void main(String[] args) {
  String filename = "example.txt";
  BufferedReader reader = new BufferedReader(new FileReader(filename));
  String line;
  while ((line = reader.readLine()) != null){
                    System.out.println(line);}
  reader.close();
    }}
```

A3) In this example, the code reads data from a file named "example.txt" using a BufferedReader. However, the code lacks proper error handling in case the file is missing or if there are any I/O errors during file reading. To address resource optimism and write more robust code, it's essential to handle exceptions properly and implement proper error handling and resilience mechanisms.

Here's an improved version of the code with better error handling: (The students are not expected to write this code. This is just for better understanding)

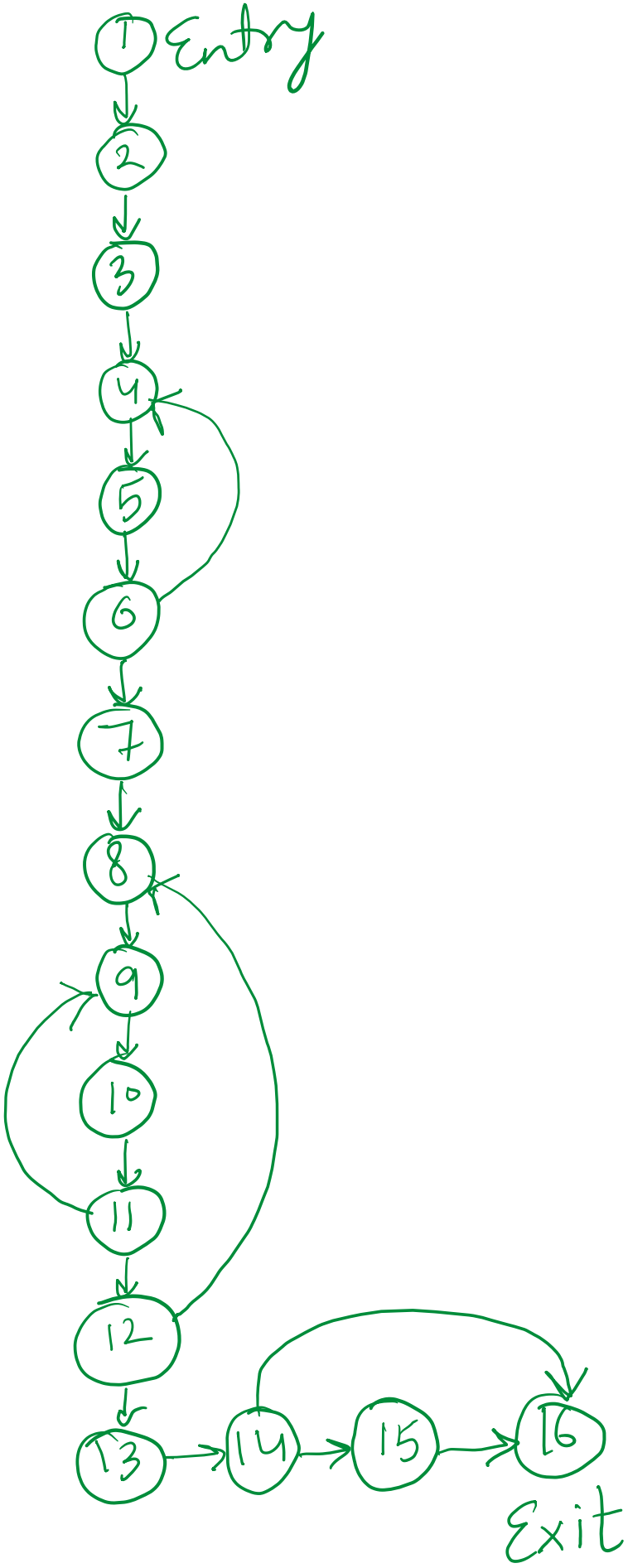
```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class ResourceOptimismExample {
    public static void main(String[] args) {
        String filename = "data.txt";

        try (BufferedReader reader = new BufferedReader(new
FileReader(filename))) {
```

```
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            // Proper error handling: Logging the exception and
potentially notifying the user
            System.err.println("Error while reading the file: " +
e.getMessage());
        }
    }
}
```

Q4) Consider the code to find the prime factors for an integer n.

1. class PrimeFactor{
2. public static void primeFactors(int n)
3. {
4. while(n%2==0){
5. System.out.println(2+ " ");
6. n/=2;
7. }
8. for (int i=3,=<=Math.sqrt(n);i+=2){
9. while(n%1==0){
10. System.out.println(i+" ");
11. n/=i;
12. }
13. }
14. If(n>2)
15. {System.out.println(n);}
16. }

a) Draw the control flow graph for the code. (4M)


1
Entry
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Exit

b) Find cyclomatic complexity for the code. (1M)

V(G)=E-N+2=19-16+2=5

c) What can you say about the testability of the code by the value of cyclomatic complexity? Find out all independent paths from the graph. (6M)

This is a highly testable code because of the low cyclomatic complexity.

Paths:

P1: 1 to 16

P2: 1 to 6, 6 to 4, 4 to16

P3: 1 to 12, 12 to 8, 8 to 16

P4: 1 to 11, 11 to 9, 9 to 16

P5: 1 to 14, 14 to 16

Q5) Consider the following code to find factorial of a positive integer.

```
public class Factorial {
   public int calculateFactorial(int n) {
      if (n == 0) {
         return 1;
      } else {
         return n * calculateFactorial(n - 1);
      }
   }
}
```

Find any 5 possible mutations for the given code. (5M)

A5)

These are some of the possible mutations. Students can come up with other mutations as well.

1. Change `n == 0` to `n != 0`
2. Change `n == 0` to `n == 1`
3. Change `return 1;` to `return 0;`
4. Change `return n * calculateFactorial(n - 1);` to `return n * calculateFactorial(n);`
5. Change `return n * calculateFactorial(n - 1);` to `return n + calculateFactorial(n - 1);`