Final-term Mockup Test (20% of your final mark)

Final-test information. The test contains **20 questions** including multiple-choice questions and code analysis questions. You have 40 minutes to complete the test. It covers OOP topics **from Week 1 to Week 11** (inclusive).

The test will be in-class and **at the first hour of Week 12**. **It is close-book test**. **Please bring pens with you**. Once completed, please return the test sheet to your lab instructor. **If you have EAP, you will have 60 minutes in total**. Please speak to your lab tutor prior to the test for more time adjustment or extension if needed.

Your unit. Your Say. Feedback Survey. In Semester 1, 2025, the COS20007 unit has benefited from significant effort by the teaching team in developing a wide range of resources. These include weekly reflective quizzes, detailed lab specifications with pseudocode, partial C# coding solutions, and additional sample source code made available one week in advance. Both lectures and labs have offered enhanced explanations and more interactive guidance for the tasks. The practical verification tasks are also designed to boost your confidence in answering technical interview questions.

We genuinely value your constructive feedback through the Student Survey, which will help us continue improving the unit for future students.

Instruction. Log in to Canvas > Click on **Account** tab > Select Notifications tab> Select Course Evaluation.

Q1. Which of the following is a good practice for debugging unknown errors in C#?

a. Ignore the error and rerun the program
b. Comment out random lines of code
c. Carefully read the error message and stack trace to identify the root cause
d. Restart the IDE

Q2. Why is reading API documentation crucial in C# development?

a. To memorize method names
b. To give the documents to software testers
c. To know available methods, expected parameters, and use details
d. To avoid debugging code

Q3. Why is low coupling desirable in object-oriented design?

a. It reduces the number of classes, making the system simpler
b. It makes the system more modular, flexible, and easier to maintain
c. It ensures that all classes are independent and do not interact
d. It forces all responsibilities into a single class that has “public static void main” method

Q4. What is the purpose of making class “antisocial” in good OO design?

a. To prevent excessive communication between object and reduce dependencies
b. To minimize object creation in a system
c. To ensure objects can access and modify each other’s private data
d. To increase the number of dependencies in the program

Q5. You are working on a logging system where you need to ensure that only one instance of a Logger class exists throughout the application. What design pattern would be most appropriate?

a. Factory
b. Singleton
c. Observer
d. Prototype

Q6. Which of the following design pattern define an interface for object creation and let subclass decide which class to instantiate?

a. Adapter design pattern.
b. Command design pattern.
c. Factory design pattern.
d. Singleton design pattern.

Q7. When passing an object to a method in C#, what is actually passed?

a. A copy of the entire object
b. A pointer to the object's memory on the stack
c. A reference to the object's memory on the heap
d. The object is passed by value and stored in a new memory location

Q8. Which of the following statements is NOT a correct type of Design Patterns?

a. Data Access patterns
b. Structural patterns
c. Creational patterns
d. Behavior patterns

Q9. Refactor the following code so that adding new payment methods does not require modifying existing code?

```
class PaymentProcessor
{
    public void ProcessPayment(string paymentType, double amount)
    {
        if (paymentType == "CreditCard")
        {
            Console.WriteLine($"Processing credit card payment of ${amount}");
        }
        else if (paymentType == "PayPal")
        {
            Console.WriteLine($"Processing PayPal payment of ${amount}");
        }
        else
        {
            Console.WriteLine("Unsupported payment method.");
        }
    }
}

// Test
PaymentProcessor processor = new PaymentProcessor();
processor.ProcessPayment("CreditCard", 100.0);
processor.ProcessPayment("PayPal", 50.0);
```

What is the best way to refactor the PaymentProcessor to follow Protected Variations?

a. Keep the if-else structure but add more cases for future payment methods.
b. Use an interface for different payment methods and apply polymorphism.
c. Move the ProcessPayment method to a static class for easy access.
d. Create a switch-case statement for better readability

Q10. What is the output of the following code?

```
try {
   string text = null;
   Console.WriteLine(text.Length);
} catch (NullReferenceException) {
   Console.WriteLine("Null reference detected.");
} finally {
   Console.WriteLine("End of program.");
}
```

a. Null reference detected
b. End of program
c. Null reference detected. End of program
d. Runtime Error

Q11. Which of the following method overloads is invalid in C#?

a. public void Print (int number) {}
   public void Print (double number) {}
b. public void Display (string text) {}
   public void Display (string text, int count){}
c. public void Show(string message) {}
   public void Show(string msg) {}
d. public void Calculate(int a, int b) {}
   public void Calculate(double a,double b) {}

Q12. Given the following code, what design pattern does the code implement?

```
public interface ICarEngine {
  void StartEngine();
}
public class DieselEngine: ICarEngine {
  public void StartEngine() {
    Console.WriteLine("Ignite Diesel Engine");
  }
}
public class PetrolEngine : ICarEngine {
  public void StartEngine() {
    Console.WriteLine("Ignite PetrolEnginee");
  }
}
```

```
public class EngineFactory {
  public static ICarEngine IgniteEngine(string carEngine) {
    switch (carEngine.ToLower())
    {
      case "petrol":
        return new PetrolEngine();
      case "diesel":
        return new DieselEngine();
      default:
        throw new ArgumentException("Unknown engine type");
    }
  }
}
```

What design pattern does the code above implement?

a. Adapter Pattern - adapt to a legacy design

b. Factory Pattern – centralizes object creation based on input or configuration.

c. Builder Pattern – separates object construction from its representation.

d. Singleton Pattern – guarantee only one instance of a class.