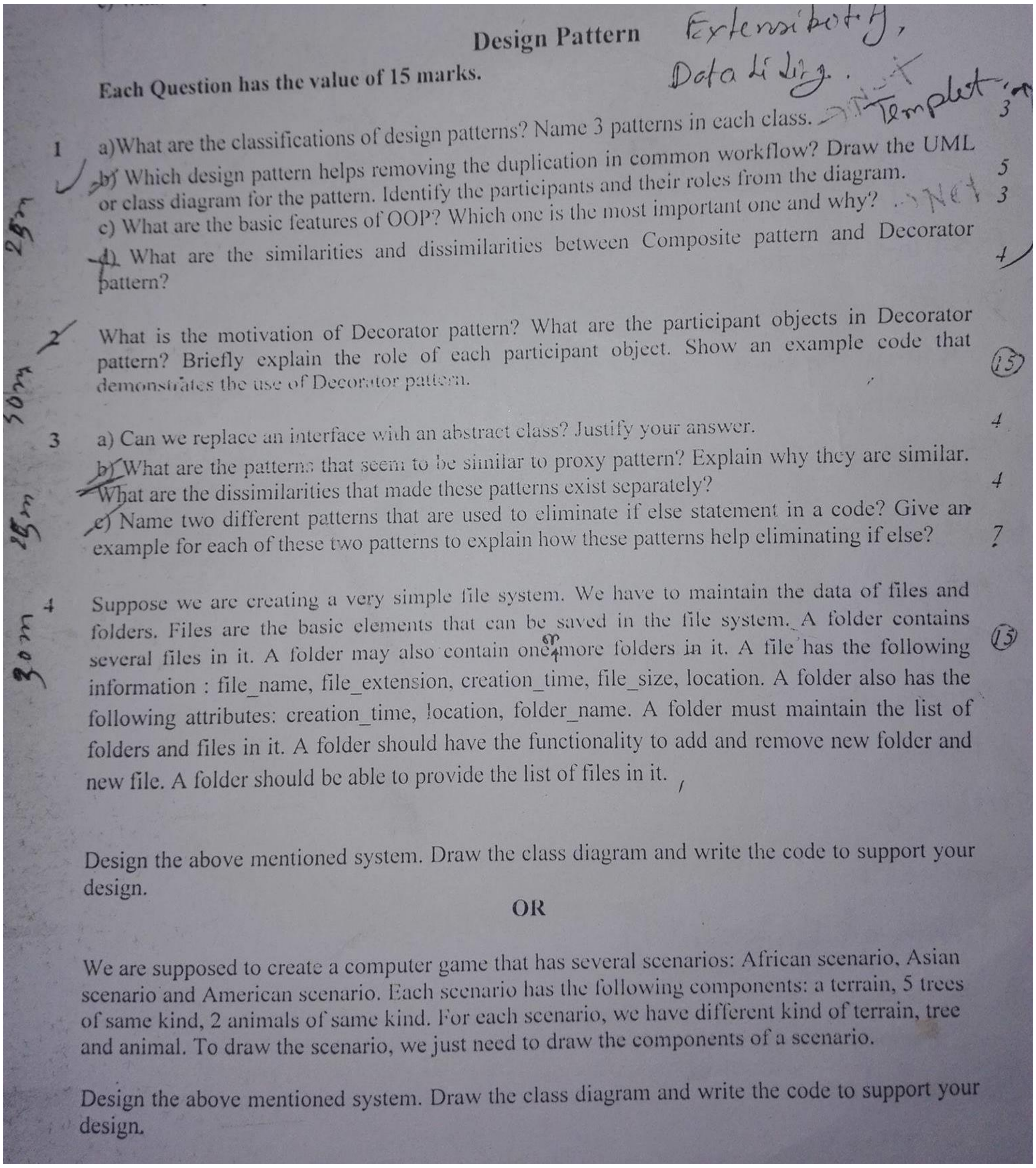
Question



Design Pattern

Extensibility,
Data hiding.

Each Question has the value of 15 marks.

1 a)What are the classifications of design patterns? Name 3 patterns in each class. 3

b) Which design pattern helps removing the duplication in common workflow? Draw the UML or class diagram for the pattern. Identify the participants and their roles from the diagram. Templet 5

c) What are the basic features of OOP? Which one is the most important one and why? .NET 3

d) What are the similarities and dissimilarities between Composite pattern and Decorator pattern? 4

25m

2 What is the motivation of Decorator pattern? What are the participant objects in Decorator pattern? Briefly explain the role of each participant object. Show an example code that demonstrates the use of Decorator pattern. $\boxed{15}$

50m

3 a) Can we replace an interface with an abstract class? Justify your answer. 4

b) What are the patterns that seem to be similar to proxy pattern? Explain why they are similar. What are the dissimilarities that made these patterns exist separately? 4

c) Name two different patterns that are used to eliminate if else statement in a code? Give an example for each of these two patterns to explain how these patterns help eliminating if else? 7

25m

4 Suppose we are creating a very simple file system. We have to maintain the data of files and folders. Files are the basic elements that can be saved in the file system. A folder contains several files in it. A folder may also contain one or more folders in it. A file has the following information : file_name, file_extension, creation_time, file_size, location. A folder also has the following attributes: creation_time, location, folder_name. A folder must maintain the list of folders and files in it. A folder should have the functionality to add and remove new folder and new file. A folder should be able to provide the list of files in it. $\boxed{15}$

30m

Design the above mentioned system. Draw the class diagram and write the code to support your design.

OR

We are supposed to create a computer game that has several scenarios: African scenario, Asian scenario and American scenario. Each scenario has the following components: a terrain, 5 trees of same kind, 2 animals of same kind. For each scenario, we have different kind of terrain, tree and animal. To draw the scenario, we just need to draw the components of a scenario.

Design the above mentioned system. Draw the class diagram and write the code to support your design.

Answer

1.a

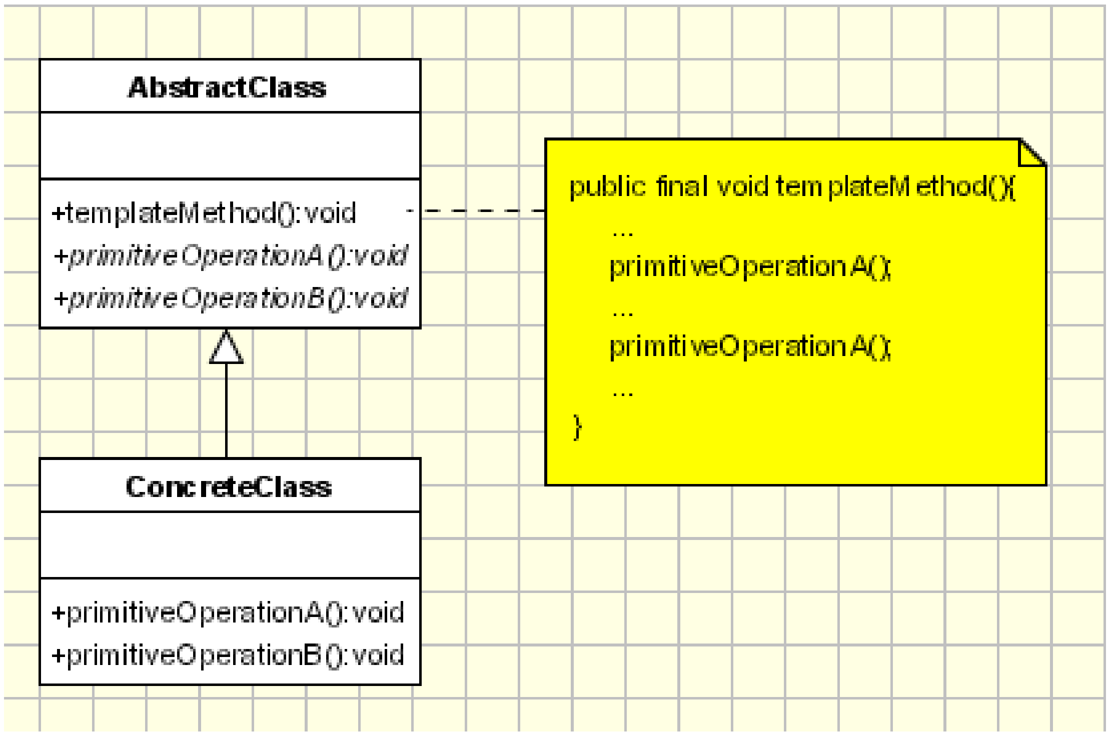
3 types:

1. Creational:
   1.1. Factory
   1.2. Singleton
   1.3. Abstract Factory
2. Structural:
   2.1. Proxy
   2.2. Decorator
   2.3. Composite
3. Behavioral
   3.1. Observer
   3.2. Strategy
   3.3. Template

1.b

Template Pattern.

UML:


AbstractClass
+templateMethod():void
+primitiveOperationA():void
+primitiveOperationB():void
ConcreteClass
+primitiveOperationA():void
+primitiveOperationB():void
public final void templateMethod(){
...
primitiveOperationA();
...
primitiveOperationA();
...
}

Participants and their role:

1. AbstractClass

    - defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm.

    - implements a template method which defines the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.

2. ConcreteClass

    - implements the primitive operations to carry out subclass-specific steps of the algorithm.

    - When a concrete class is called the template method code will be executed from the base class while for each method used inside the template method will be called the implementation from the derived class.

1.c

Basic Features of OOP:

1. **Encapsulation**
2. **Inheritance**
3. **Polymorphism**
4. **Abstraction**

Encapsulation is most important. Because,

Benefits of encapsulation range from protection of intellectual property, ensuring other parts of the program don't change or access data that belongs to another part of the program, or to making a program more manageable and robust by separating parts of code.

Encapsulation helps in isolating implementation details from the behavior exposed to clients of a class (other classes/functions that are using this class) and gives you more control over coupling in your code.

1.d

Similarities:

Decorator is often used with Composite. When decorators and composites are used together, they will usually have a common parent class. So decorators will have to support the Component interface with operations like Add, Remove, and GetChild.

Dissimilarities:

- Decorator is designed to let you add responsibilities to objects without subclassing. Composite's focus is not on embellishment but on representation
- Decorator adds/remove additional responsibilities - it isn't intended for object aggregation.A decorator is likely to add functionality (decorate) an object and a strategy is likely to swap functionality.

2.

Motivation

Sometimes we want to add responsibilities to individual objects, not to an entire class. A graphical user interface toolkit, for example, should let you add properties like borders or behaviors like scrolling to any user interface component.

One way to add responsibilities is with inheritance. Inheriting a border from another class puts a border around every subclass instance. This is inflexible, however, because the choice of border is made statically. A client can't control how and when to decorate the component with a border.

A more flexible approach is to enclose the component in another object that adds the border. The enclosing object is called a decorator. The decorator conforms to the interface of the component it decorates so that its presence is transparent to the component's clients. The decorator forwards requests to the component and may perform additional actions (such as drawing a border) before or after forwarding. Transparency lets you nest decorators recursively, thereby allowing an unlimited number of added responsibilities..

Participants and their role:

1.Component

o defines the interface for objects that can have responsibilities added to them dynamically.

2.ConcreteComponent

defines an object to which additional responsibilities can be attached.

3. Decorator

o maintains a reference to a Component object and defines an interface that conforms to Component's interface.

4. ConcreteDecorator

o adds responsibilities to the component.

Example Code: https://www.tutorialspoint.com/design_pattern/decorator_pattern.htm

3.a

No, Not always:

- a class can extend only one class
- a class can implement more than one interface

Abstract classes are the partial implementation of Abstraction while Interfaces are the fully implementation of Abstraction.Means in Abstract classes we can put methods declaration as well as method body. We can't create an object of Abstract classes(association) and reuse the class by inheritence(not by association). By default in interfaces all declared variables are static final and All methods are public.

Unlike interfaces, abstract classes can contain fields that are not static and final, and they can contain implemented methods. Such abstract classes are similar to interfaces, except that they provide a partial implementation, leaving it to subclasses to complete the implementation. If an abstract class contains only abstract method declarations, it should be declared as an interface instead.

3.b

Similar to proxy pattern:

Adapter: An adapter provides a different interface to the object it adapts. In contrast, a proxy provides the same interface as its subject. However, a proxy used for access protection might refuse to perform an operation that the subject will perform, so its interface may be effectively a subset of the subject's.

Decorator: Although decorators can have similar implementations as proxies, decorators have a different purpose. A decorator adds one or more responsibilities to an object, whereas a proxy controls access to an object.

3.c

Two patterns used to eliminate if else statement:

1. Strategy Pattern

Example:

Too many if else :

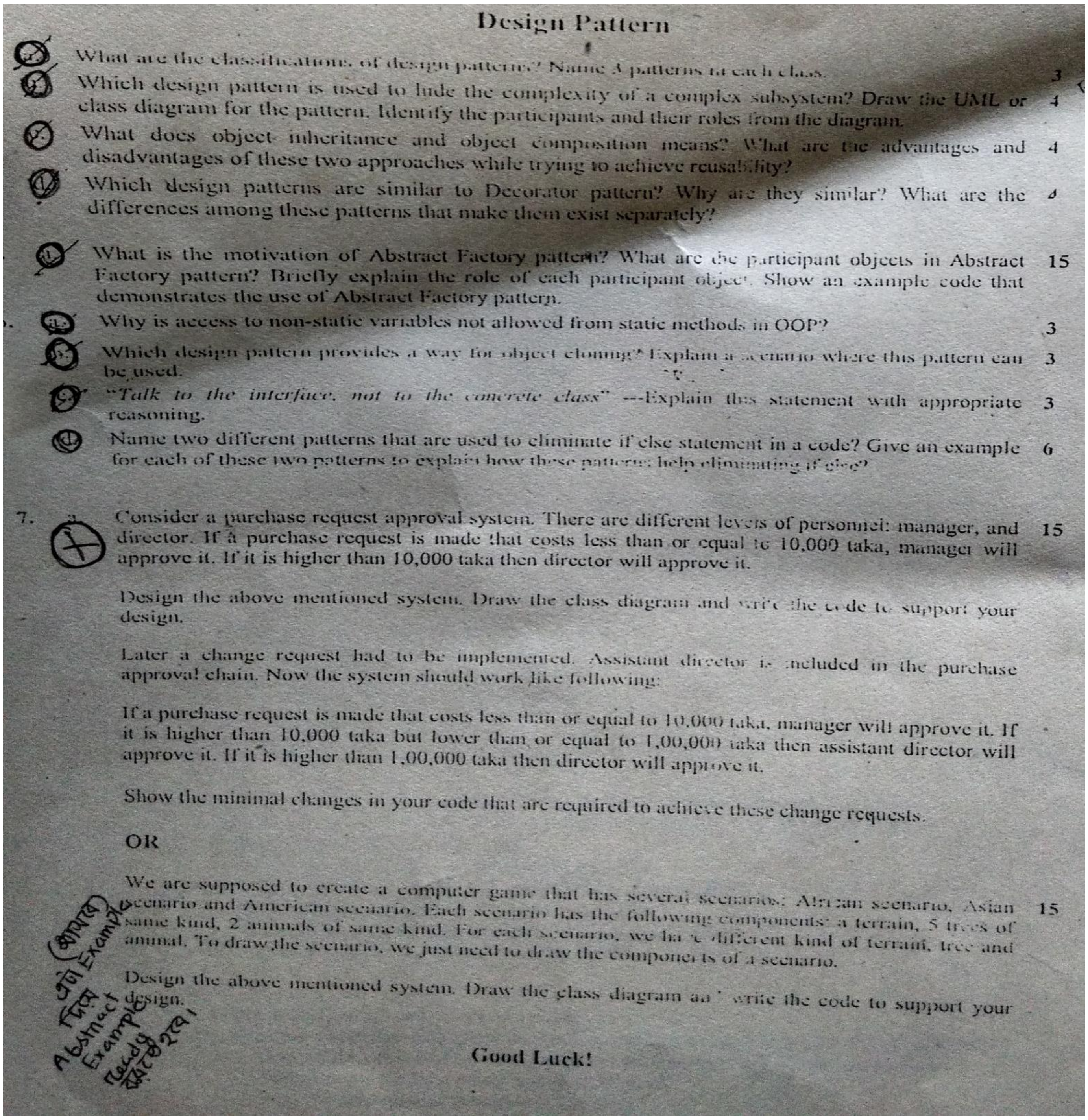
```
public double Calculate(a,b, operation)
{
    if(operation = ‘add‘)
        return a+b;
    else if(operation = ‘subtract‘)
        return a-b;
    else if(operation = ‘multiply‘)
        return a*b;
}
```

Using strategy:

```
public class StrategyPatternDemo {

   public static void main(String[] args) {

      Context context = new Context(new OperationAdd());

      System.out.println("10 + 5 = " + context.executeStrategy(10, 5));


      context = new Context(new OperationSubstract());

      System.out.println("10 - 5 = " + context.executeStrategy(10, 5));


      context = new Context(new OperationMultiply());

      System.out.println("10 * 5 = " + context.executeStrategy(10, 5));

   }

}
```

2. Flyweight (not sure)

Question



Design Pattern

What are the classifications of design patterns? Name 3 patterns in each class. 3

Which design pattern is used to hide the complexity of a complex subsystem? Draw the UML or class diagram for the pattern. Identify the participants and their roles from the diagram. 4

What does object inheritance and object composition means? What are the advantages and disadvantages of these two approaches while trying to achieve reusability? 4

Which design patterns are similar to Decorator pattern? Why are they similar? What are the differences among these patterns that make them exist separately? 4

What is the motivation of Abstract Factory pattern? What are the participant objects in Abstract Factory pattern? Briefly explain the role of each participant object. Show an example code that demonstrates the use of Abstract Factory pattern. 15

Why is access to non-static variables not allowed from static methods in OOP? 3

Which design pattern provides a way for object cloning? Explain a scenario where this pattern can be used. 3

"Talk to the interface, not to the concrete class" ---Explain this statement with appropriate reasoning. 3

Name two different patterns that are used to eliminate if else statement in a code? Give an example for each of these two patterns to explain how these patterns help eliminating if else? 6

7. a. Consider a purchase request approval system. There are different levels of personnel: manager, and director. If a purchase request is made that costs less than or equal to 10,000 taka, manager will approve it. If it is higher than 10,000 taka then director will approve it. 15

Design the above mentioned system. Draw the class diagram and write the code to support your design.

Later a change request had to be implemented. Assistant director is included in the purchase approval chain. Now the system should work like following:

If a purchase request is made that costs less than or equal to 10,000 taka, manager will approve it. If it is higher than 10,000 taka but lower than or equal to 1,00,000 taka then assistant director will approve it. If it is higher than 1,00,000 taka then director will approve it.

Show the minimal changes in your code that are required to achieve these change requests.

OR

We are supposed to create a computer game that has several scenarios: African scenario, Asian scenario and American scenario. Each scenario has the following components: a terrain, 5 trees of same kind, 2 animals of same kind. For each scenario, we have different kind of terrain, tree and animal. To draw the scenario, we just need to draw the components of a scenario. 15

Design the above mentioned system. Draw the class diagram and write the code to support your design.

(আমার) Example [illegible] Abstract Example [illegible]

Good Luck!

Answer

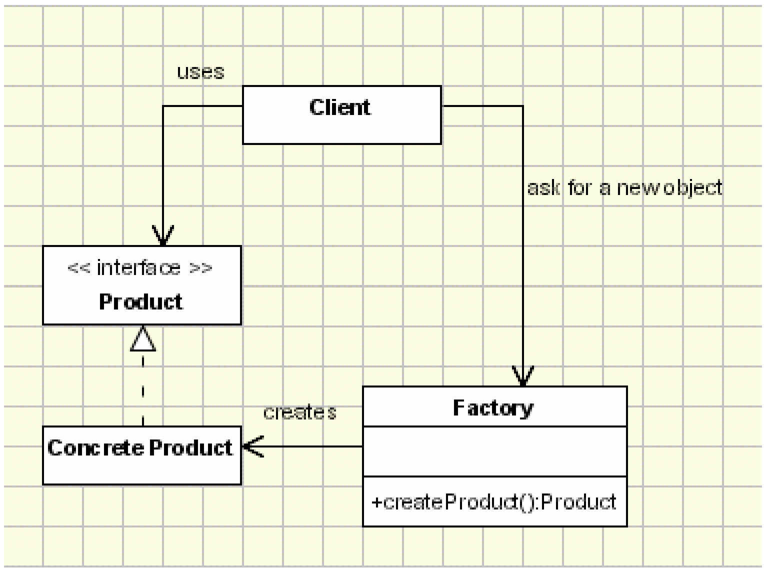
1.a

Common

1.b

Factory.

UML:



Participants and role:

1. **Product** - defines the interface of objects the factory method creates.
2. **ConcreteProduct** - implements the Product interface
3. **Creator** -declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
   -may call the factory method to create a Product object.
4. **ConcreteCreator** -overrides the factory method to return an instance of a ConcreteProduct

1.c

Inheritance is an "is-a" relationship. Composition is a "has-a" relation ship.

Inheritnace means inheriting all the constants, you are committing your object to be of the type specified by the interface; it's still an "is-a" relationship.

Composition is fundamentally different from inheritance. When you use composition, you are (as the other answers note) making a "has-a" relationship between two objects, as opposed to the "is-a" relationship that you make when you use inheritance.

Inheritance and composition advantages and disadvantages according to reusability:

In inheritance,
Code re-usability:- Whatever code defined in super class can be reused by any of its subclass.
In composition,

Code re-usability:- Code re-usability can be achieved same as Inheritance but need to explicitly call the code which needs to be reused.

1.d

Similar to decorator pattern:

1. Adapter: A decorator is different from an adapter in that a decorator only changes an object's responsibilities, not its interface; an adapter will give an object a completely new interface.
2. Composite: A decorator can be viewed as a degenerate composite with only one component. However, a decorator adds additional responsibilities—it isn't intended for object aggregation.
3. Strategy: A decorator lets you change the skin of an object; a strategy lets you change the guts. These are two alternative ways of changing an object.

1.e

Common

2.a

non-static members are part of state of an object

static methods do not have an object associated with the method invocation - so obviously they should not be able to access fields that are part of an object's inner state

2.b

common

2.c

Interfaces are just contracts or signatures and they don't know anything about concrete class.

Coding against interface means, the client code always holds an Interface object which is supplied by a factory. Any instance returned by the factory would be of type Interface which any factory candidate class must have implemented. This way the client program is not worried about concrete class and the interface signature determines what all operations can be done. This can be used to change the behavior of a program at run-time.

2.d

Common