Test KEN1520 - Software Engineering Exam 2023

Block introduction

Question order: Fixed

Question 1 – 2023_Exam_Airport_Managent_OP – 136224.1.0

A major airport is planning to develop a new baggage handling system to improve efficiency and security, which requires a dynamic and adaptable development process due to changing requirements and emerging technologies. The airport authority has established strict safety guidelines and performance metrics for the system, which must use advanced sensors and AI algorithms to sort and track luggage with minimal human intervention.

As the project manager for the software development process, which of the following process models would you choose for this project, considering the need for adaptability and flexibility in the face of evolving requirements and technologies?

Explain your choice in 3 to 7 sentences.

1. The waterfall method: a sequential, linear approach where each phase of the project must be completed before moving to the next one
2. The iterative incremental model: a cyclic process that involves developing the software in small increments, with regular feedback and adjustments
3. An agile method such as Scrum: an iterative, adaptive approach that focuses on collaboration, self-organization, and continuous improvement

Grading instruction

Correct answer and explanation (Number of points: 10)

Possible answers (all options are correct given the right explanation)

a. The waterfall method

One might choose the waterfall method for its structured approach, ensuring that each phase of the project is completed and thoroughly reviewed before moving on to the next. This could be particularly important in a project with strict safety and security requirements, as it ensures that every aspect of the system is carefully planned, designed, and tested before implementation. However, the waterfall method's lack of flexibility might be a drawback if requirements change or new technologies emerge during development.

b. The iterative incremental model

The iterative incremental model could also be a viable option, as it allows for a more flexible development process compared to the waterfall method while still maintaining a structured approach. In this model, the software is developed in small increments, with regular feedback and adjustments made along the way. This approach can help to ensure that safety and security requirements are met while also allowing the team to adapt to changing requirements and incorporate new technologies as they become available. The iterative incremental model strikes a balance between the rigidity of the waterfall method and the adaptability of agile methods like Scrum.

c. An agile method such as Scrum

The agile method, such as Scrum, would be the most appropriate choice for this project, considering the need for adaptability and flexibility in the face of evolving requirements and technologies. Scrum's iterative, adaptive approach allows the development team to respond quickly to changes in requirements or the introduction of new technologies. This method emphasizes collaboration, self-organization, and continuous improvement, which helps the team to stay aligned with the project's goals and deliver a high-quality software solution. The agile approach also encourages regular communication with stakeholders, ensuring that their needs are understood and incorporated throughout the development process.

Question 2 – 2023_Exam_Refactoring_OP – 136228.1.0

Below is a code snippet representing a simple video rental system. Analyze the code and identify one code smell present in the code. Suggest a refactoring technique to improve the design of the code, and provide a brief explanation of how the refactoring technique improves the code. Although there may be more than one smell in the code, but you only must choose and solve one. You may use source code in your answer to illustrate your solution.

```
class Customer {
    private String name;
    private List<Rental> rentals;

    public Customer(String name) {
        this.name = name;
        this.rentals = new ArrayList<>();
    }

    public void addRental(Rental rental) {
        rentals.add(rental);
    }

    public String generateStatement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        StringBuilder result = new StringBuilder("Rental Record for " + name + "\n");

        for (Rental rental : rentals) {
            double amount = 0;
            switch (rental.getMovie().getPriceCode()) {
                case Movie.REGULAR:
                    amount += 2;
                    if (rental.getDaysRented() > 2) {
                        amount += (rental.getDaysRented() - 2) * 1.5;
                    }
                    break;
                case Movie.NEW_RELEASE:
                    amount += rental.getDaysRented() * 3;
                    break;
                case Movie.CHILDRENS:
                    amount += 1.5;
                    if (rental.getDaysRented() > 3) {
                        amount += (rental.getDaysRented() - 3) * 1.5;
                    }
                    break;
            }

            frequentRenterPoints++;
            if (rental.getMovie().getPriceCode() == Movie.NEW_RELEASE && rental.getDaysRented() > 1) {
                frequentRenterPoints++;
            }


result.append("\t").append(rental.getMovie().getTitle()).append("\t").append(amount).append("\n");
            totalAmount += amount;
        }

        result.append("Amount owed is ").append(totalAmount).append("\n");
        result.append("You earned ").append(frequentRenterPoints).append(" frequent renter points!");
        return result.toString();
    }
}
```

Which of the following code smells can you identify in the code snippet?

- Long Method: Excessively lengthy, complex functions.
- Large Class: Overstuffed class with many responsibilities.
- Data Clumps: Frequently occurring grouped variables.
- Primitive Obsession: Overusing primitive data types.
- Divergent Change: Class modified for multiple reasons.
- Shotgun Surgery: Small changes spread across classes.

Which of the following refactoring techniques would be most appropriate to address the identified code smell?

- Extract Method
- Extract Class
- Introduce Parameter Object

- Replace Primitive with Object
- Extract Subclass
- Move Method
- Replace Conditional with Polymorphism

Grading instruction

Correct Solution (Number of points: 10)

Solution 1:

Refactoring Technique: Extract Method

Explanation: The generateStatement() method in the Customer class is quite long and does multiple things like calculating the amount, updating frequent renter points, and generating the statement text. We can apply the Extract Method refactoring technique to break down the generateStatement() method into smaller, more focused methods:

calculateAmount(Rental rental)
calculateFrequentRenterPoints(Rental rental)
generateStatementText(double totalAmount, int frequentRenterPoints)
By extracting these methods, the code becomes more modular, easier to understand, and easier to maintain.

Solution 2:

Refactoring Technique: Move Method

Explanation: The responsibility of calculating the amount and frequent renter points for a rental should belong to the Rental class, rather than the Customer class. We can use the Move Method refactoring technique to move the calculation logic from the Customer class to the Rental class. This will involve creating new methods in the Rental class:

double calculateAmount()
int calculateFrequentRenterPoints()
By moving these methods, we distribute the responsibilities more appropriately between the classes and make the code more cohesive.

Solution 3:

Refactoring Technique: Replace Conditional with Polymorphism

Explanation: The generateStatement() method in the Customer class contains a switch statement based on the movie's price code to calculate the amount. We can replace this conditional with polymorphism by creating a hierarchy of price classes (e.g., RegularPrice, NewReleasePrice, ChildrensPrice) that each implement a common interface or extend a base class. Then, we can move the calculation logic to the appropriate price class:

RegularPrice.calculateAmount(int daysRented)
NewReleasePrice.calculateAmount(int daysRented)
ChildrensPrice.calculateAmount(int daysRented)
By using polymorphism, we can eliminate the need for a conditional statement, making the code more flexible, easier to extend with new price codes, and easier to maintain.

Question 3 – 2023_Exam_Arch_MC – 136233.2.0

Which of the following software architecture styles emphasises the separation of concerns and is typically used in the development of user-facing web-based and desktop applications?

A Client-server architecture

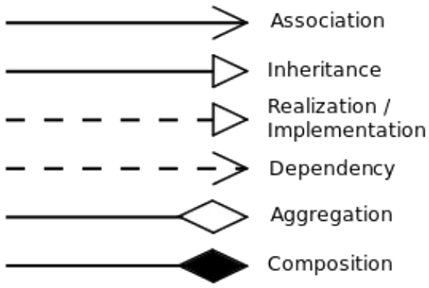
B Service-oriented architecture

C Model-view-controller (i.e. a layered) architecture

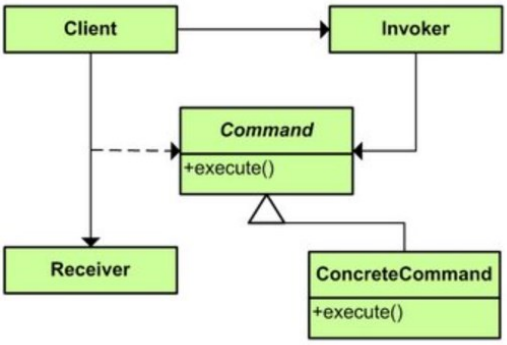
D Event-driven architecture

Design Pattern Questions

Question order: Fixed
Cheat Sheet:



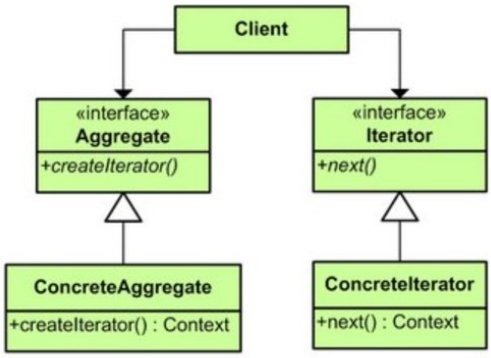
Design Patterns:



Command

Type: Behavioral

What it is:
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.



Iterator

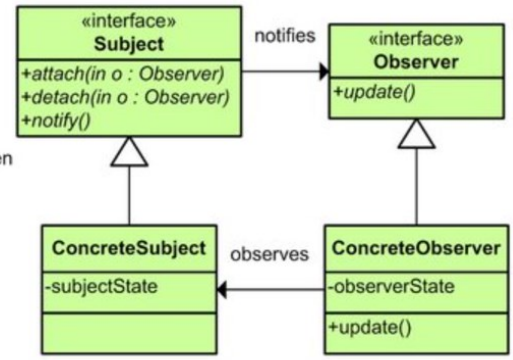
Type: Behavioral

What it is:
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Observer

Type: Behavioral

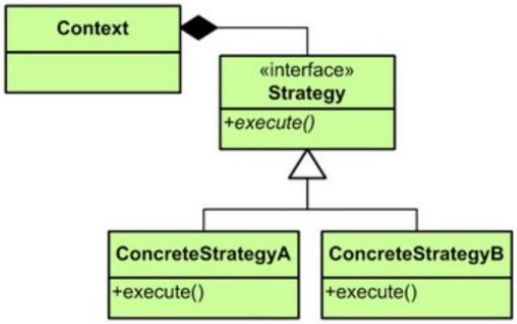
What it is:
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



Strategy

Type: Behavioral

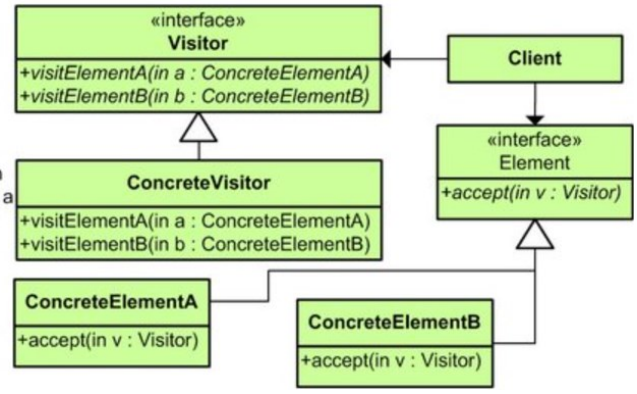
What it is:
Define a family of algorithms, encapsulate each one, and make them interchangeable. Lets the algorithm vary independently from clients that use it.



Visitor

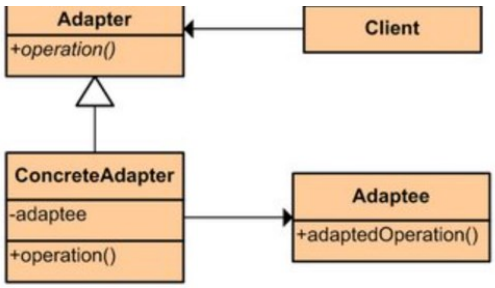
Type: Behavioral

What it is:
Represent an operation to be performed on the elements of an object structure. Lets you define a new operation without changing the classes of the elements on which it operates.



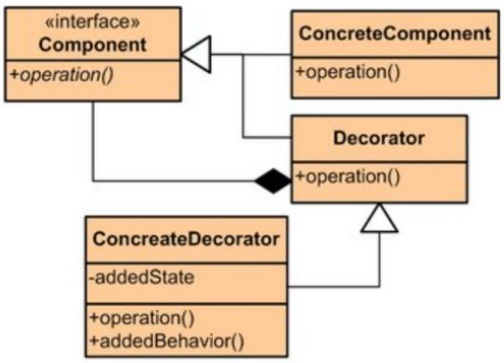
«interface»

Adapter



Type: Structural

What it is:
Convert the interface of a class into another interface clients expect. Lets classes work together that couldn't otherwise because of incompatible interfaces.



Decorator

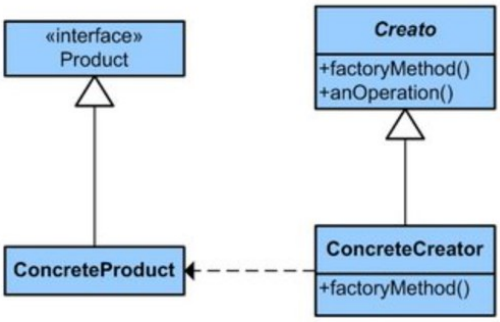
Type: Structural

What it is:
Attach additional responsibilities to an object dynamically. Provide a flexible alternative to sub-classing for extending functionality.

Factory Method

Type: Creational

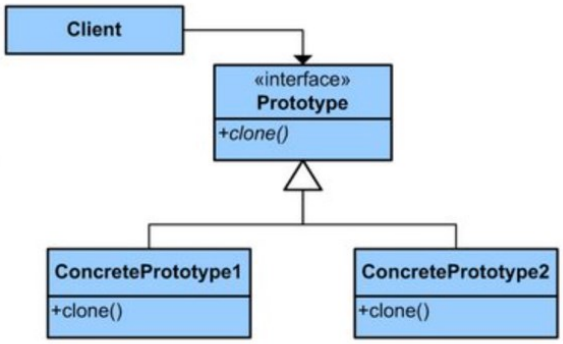
What it is:
Define an interface for creating an object, but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.



Prototype

Type: Creational

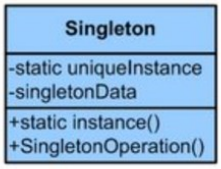
What it is:
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.



Singleton

Type: Creational

What it is:
Ensure a class only has one instance and provide a global point of access to it.



Question 4 − 2023_Exam_Design_Pattern_Match − 136230.1.0

Match each of the following design patterns with the corresponding scenario where it would be most appropriate to apply

Singleton	You are building a logging system that must have only one instance throughout the application, ensuring that all components log their messages to the same place.
Factory Method	You are creating a video game where different types of enemy characters can be spawned. The way these characters are created depends on the level and context of the game.
Observer	You are developing a weather monitoring application that needs to notify various display components (current temperature, humidity, pressure) whenever new weather data is available. These display components can be added or removed at runtime.
Decorator	You are developing a text editor application, and you want to enable users to apply multiple formatting options like bold, italic, or underline to the text. You need to be able to combine different formatting options without creating a separate class for each combination.

Question 5 − 2023_Exam_Undo_Pattern_MC − 136231.3.0

Consider a software application that has a feature to undo and redo user actions. When the user performs an action, it is recorded and stored in a history. The user can then undo the last action, which removes it from the history and reverts the application state to the previous state. The user can also redo the last undone action, which adds it back to the history and restores the application state.

Which of the following design patterns is most suitable for implementing the undo/redo feature described above?

A Singleton

B Strategy

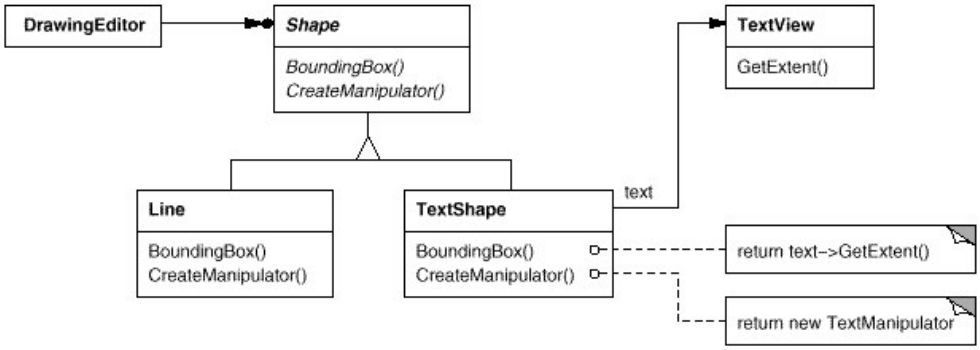
C Command

D Observer

E Decorator

Question 6 – 2023_Exam_Editor_Pattern_MC – 136235.1.0

Consider a drawing editor that lets users draw and arrange graphical elements (lines, polygons, text, etc.) into pictures and diagrams. The drawing editor's key abstraction is the graphical object, which has an editable shape and can draw itself. The interface for graphical objects is defined by an abstract class called Shape. The editor defines a subclass of Shape for each kind of graphical object: a LineShape class for lines, a PolygonShape class for polygons, and so forth.



This class diagram above illustrates the use of a design pattern. It shows how BoundingBox requests, declared in class Shape, are converted to GetExtent requests defined in TextView. Since TextShape adapts TextView to the Shape interface, the drawing editor can reuse the otherwise incompatible TextView class.

Which design pattern is represented in this class diagram?

- **A** Decorator pattern
- **B** Factory pattern
- **C** Adapter pattern
- **D** Singleton pattern
- **E** Command pattern

Question 7 – 2023_Exam_NotificationSystem_MC – 136225.2.0

You are developing a notification system for a web application. The system currently sends email notifications to users, but you anticipate that in the future, it might need to support other types of notifications, such as SMS or push notifications. Based on the SOLID principles, which of the following design approaches would best ensure that the system is easy to extend and maintain?

- **A** Create a single class with multiple methods, one for each notification type (email, SMS, push), and modify the class whenever a new notification type is needed.
- **B** Implement a base class for the notification system with a concrete sendNotification() method and inherit from this class for each new notification type, overriding the method as necessary
- **C** Create an interface or abstract base class for the notification system, implement separate classes for each notification type, and ensure that higher-level modules depend on the interface or abstract base class rather than specific notification sender classes.
- **D** Use a procedural approach, writing separate methods for each notification type, and call the appropriate method based on the notification type needed.

Question 8 − 2023_Exam_Media_Fill − 136227.2.0

A colleague wrote the following interface for use in a media streaming application that supports movies, music, and podcasts:

```
interface Media {
    public void play();
    public void pause();
    public int getDuration();
    public String getGenre();
    public String getTitle();
    public String getDirector();
    public String getAlbum();
    public int getEpisodeNumber();
}
```

This interface violates the [Interface Segregation] principle, which states that classes or interfaces should not be forced to implement methods they do not use. To adhere to this principle, you could create separate interfaces for each media type, such as [PodcastInterface] , [PodcastInterface] , and [PodcastInterface] , with only the relevant methods for each type.

[Alphanumeric]	[Alphanumeric]	[Alphanumeric]	[Alphanumeric]
Interface Segregation	**PodcastInterface**	**PodcastInterface**	**PodcastInterface**
interface segregation	Podcast Interface	Podcast Interface	Podcast Interface
	Podcast	Podcast	Podcast
	Movie	Movie	Movie
	Movie Interface	Movie Interface	Movie Interface
	MovieInterface	MovieInterface	MovieInterface
	Music Interface	Music Interface	Music Interface
	MusicInterface	MusicInterface	MusicInterface
	Music	Music	Music
	Movie	Movie	Movie
	Movies	Movies	Movies
	MovieInterface	MovieInterface	MovieInterface
	Movie Interface	Movie Interface	Movie Interface
	Podcasts	Podcasts	Podcasts
	Radio	Radio	Radio
	Radio Interface	Radio Interface	Radio Interface
	RadioInterface	RadioInterface	RadioInterface
	Streaming	Streaming	Streaming
	StreamingInterface	StreamingInterface	StreamingInterface
	Streaming Interface	Streaming Interface	Streaming Interface

Question 9 – 2023_Exam_Errors1_MC – 136241.1.0

The piece of code below contains may or may not contain an error.

Determine whether it will result in static errors, dynamic errors, or whether the program will run with no errors:

```
int n = 5;
if (n) {
  n = n + 1;
}
```

- **A** Static error
- **B** Dynamic error
- **C** No errors

Question 10 – 2023_Exam_Errors2_MC – 136242.2.0

The piece of code below contains may or may not contain an error.
Determine whether it will result in static errors, dynamic errors, or whether the program will run with no errors:

```
int n = 5;
double probability = 1/n;
```

- **A** Static error
- **B** Dynamic error
- **C** No errors

Question 11 – 2023_Exam_Errors3_MC – 136243.2.0

The piece of code below contains may or may not contain an error.
Determine whether it will result in static errors, dynamic errors, or whether the program will run with no errors:

```
int sum = 0;
int n = 0;
int average = sum/n;
```

- **A** Static error
- **B** Dynamic error
- **C** No errors

Question 12 – 2023_Exam_Errors4_MC – 136244.3.0

The piece of code below contains may or may not contain an error.

Determine whether it will result in static errors, dynamic errors, or whether the program will run with no errors:

```
double sum = 7;
double n = 0;
double average = sum/n;
```

A Static error

B Dynamic error

C No errors

Question 13 – 2023_Exam_Git_MC – 136240.2.0

Bob and Alice share a git repository. They also share a timezone. They perform the following actions on the same day, at the timestamps shown:

Bob:

```
10:00 > git pull
10:05 > git add "Bat.java"
10:10 > git commit -m "Added Bat"
10:15 > git add "Man.java"
10:20 > git commit -m "Added Man"
10:30 > git push
```

Alice:

```
10:00 > git pull
10:05 > git branch add_module
10:05 > git checkout add_module
10:15 > git add "Module.java"
10:15 > git commit -m "Added Module"
11:00 > git pull
11:05 > git checkout master
11:10 > git merge add_module
11:15 > git push
```

What will be the resulting history of the repository?

A
```
-- Bat -- Man -- master
   \-- Module    /
```

B
```
-- Bat -- Man -- Merge commit -- master
   \-- Module    /
```

C
```
-- Bat     -- Merge commit -- Man -- master
   \-- Module /
```

D
```
-- Bat -- Module -- Man -- master
```

Question 14 − 2023_Exam_Legal_Java_MC − 136246.2.0

Consider the following code, taken from the labs, executed sequentially:

```
char vowel0 = 'a';
final char vowel1 = vowel0;
String vowel2 = vowel1 + "eiou";
final String vowel3 = vowel2;
char[] vowel4 = new char[] { vowel0, 'e', 'i', 'o', 'u' };
final char[] vowel5 = vowel4;
```

Which of the following statements are legal Java?

A `vowel0 = vowel5[0];`

B `String vowel6 = vowel3.concat("w");`

C `String vowel7 = vowel3.replace("a", "");`

D `vowel4[0] = vowel3[0];`

E char[] vowel8 = vowel3;

F `String vowel9 = vowel2 + vowel3;`

Question 15 − 2023_Exam_Order_Packages_MC − 136232.1.0

Consider the following Java classes in different packages of a software system:

```
package com.example.order;
public class Order {
    private List<OrderItem> orderItems;
    private PaymentService paymentService;
    // ... other fields and methods
}

package com.example.order;
public class OrderItem {
    private Product product;
    // ... other fields and methods
}

package com.example.product;
public class Product {
    private List<Review> reviews;
    private Category category;
    // ... other fields and methods
}

package com.example.product;
public class Category {
    private List<Product> products;
    // ... other fields and methods
}

package com.example.payment;
public class PaymentService {
    private CreditCardProcessor creditCardProcessor;
    // ... other fields and methods
}

package com.example.payment;
public class CreditCardProcessor {
    // ... other fields and methods
}
```

Which of the following changes to the codebase promote loose coupling and high cohesion? Choose two answers:

A Moving the **Category** and **Product** classes into the com.example.order package

B Moving the **Order** and **OrderItem** classes into the com.example.payment package

C Creating a Billing package and moving the PaymentService and CreditCardProcessor classes into it

D Extracting an interface from the PaymentService class and passing an instance of the interface to the Order class constructor

E Removing the reviews field from the Product class and creating a separate Review class in a com.example.review package

F Changing the **PaymentService** class to be a singleton and using a static factory method to create its instance

Question 16 – 2023_Exam_Specifications_MC – 136245.2.1

Which of the following can be true about a pair of specifications A and B?

- **A** A can be stronger than B and have a weaker precondition.
- **B** A can be stronger than B and have the same precondition.
- **C** A can be stronger than B and have a stronger precondition.
- **D** If A has both a stronger precondition and a weaker postcondition than B, then A and B are incomparable.
- **E** A can be weaker than B and have a weaker postcondition.
- **F** If A is underdetermined and B has a weaker precondition than A, then B must be underdetermined as well.

Question 17 – 2023_Exam_Test_Design_Princ_MC – 136236.2.0

What is the design principle behind Test-Driven Development?

- **A** Thoroughly test your codebase dividing the input space into subdomains and creating tests that check each partition.
- **B** Consider boundaries between subdomains and provide tests for them.
- **C** Write tests first, then write the code that pass those tests.
- **D** Write a specification, write tests that exercise it, and finally write the code.

Question 18 – 2023_Exam_TestUnary_MC – 136238.3.0

Consider the following test case:

```
@Test public void testUnary(List<Integer> container, UnaryOp unaryOp) {
  bool condition = true;
  for (int i = 0; i < container.length && condition; ++i) {
    condition &= unaryOp(container[i]);
  }
  assertEquals(true, condition);
}
```

Which of the following statements are true? (Choose all correct answers)

- **A** It tests a unary predicate on all elements of a list of integers.
- **B** It could be made more generic by allowing it to accept HashMaps instead of List<Integer>.
- **C** It can be reused to test a variety of unary predicates.
- **D** An error in the test is not informative enough about the source of the error.