quiz solution w3

Question 1

What does MVC Stand for? Use spaces between each word, no upper case letters, and no punctuation.

[model view controller]

Correct

Correct! The model view controller pattern is important for user-interface applications, and it previews some software architectures that we will talk about in the next course.

Question 2

Select the **two** elements of the open/closed principle:

0 / 1 point

☐ **Open for modification**

This should not be selected

Incorrect. Try again!

☐ **Open for extension**

Correct

Correct! Well-designed software should strive to be open for extension, implying that the code can be extended without having to change existing parts.

☐ **Open for maintenance**

This should not be selected

Incorrect. Try again!

☐ **Closed for modification**

Correct

Correct! Good software strives to close parts off for modification, which means that they should not need to be opened up again when extending functionality.

- [ ] **Closed for extension.**

This should not be selected

Incorrect. Try again!

- [ ] **Closed for maintenance.**

This should not be selected

Incorrect. Try again!

Question 3

What is the best description of the Dependency Inversion principle?

0 / 1 point

- [ ] **Client objects are dependent on a service interface that directs their requests.**

Incorrect

Incorrect. This is not what the Dependency Inversion principle is!

- [ ] **Service objects subscribe to their prospective client objects as Observers, watching for a request.**

Incorrect

Incorrect. This is not what the Dependency Inversion principle is!

- [ ] **Client objects depend on an Adaptor Pattern to interface with the rest of the system.**

Incorrect

Incorrect. This is not what the Dependency Inversion principle is!

- [ ] **Client objects depend on generalizations instead of concrete objects.**

Correct

Correct! Dependencies at high levels should depend on generalizations (superclasses or interfaces) where possible.

Question 4

Which of these statements is true about the Composing Objects principle?

1. it provides behaviour with aggregation instead of inheritance
2. it leads to tighter coupling

1 / 1 point

☐ **The first statement is true**

Correct

Correct! Behaviour can be built by aggregating objects instead of using inheritance. This is an an inherently more flexible approach.

☐ **The second statement is true**

Incorrect

Incorrect! Generally keeping objects separated and combining their behaviour through aggregation is a more flexible approach.

☐ **Neither statement is true**

Incorrect

Incorrect. At least one of these statements is true.

☐ **Both statements are true**

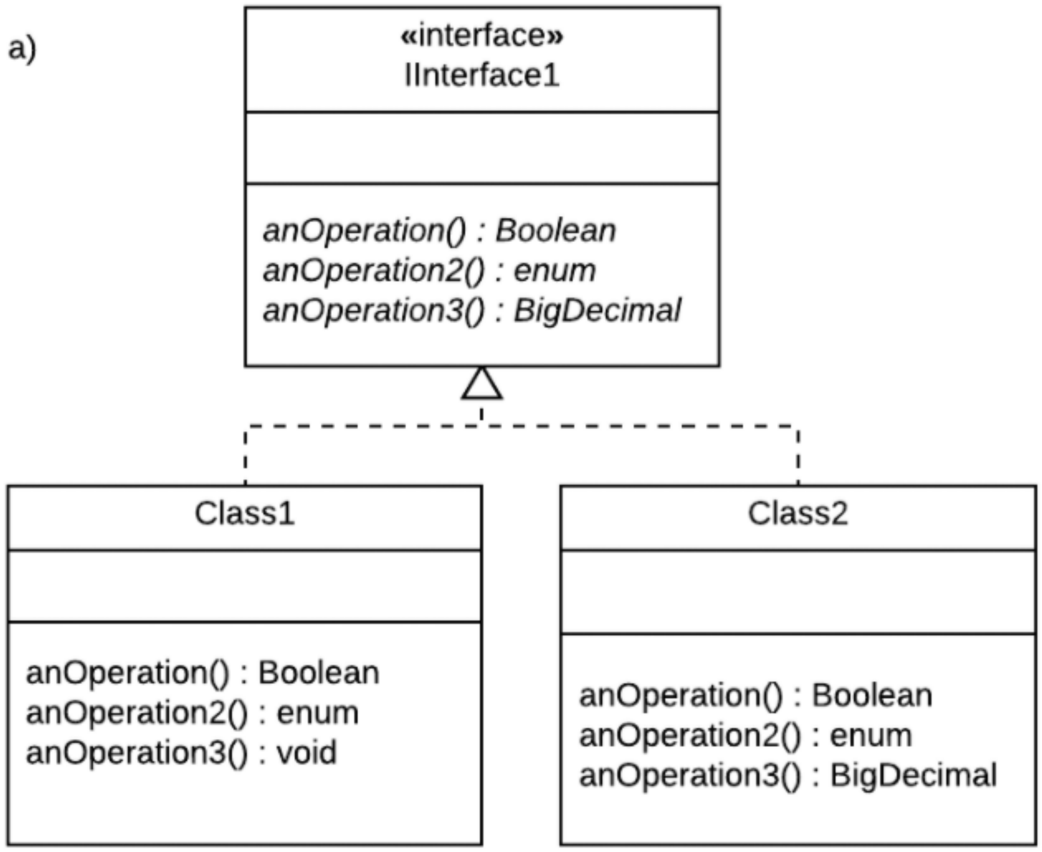
Incorrect

Incorrect. At least one of these statements is false.

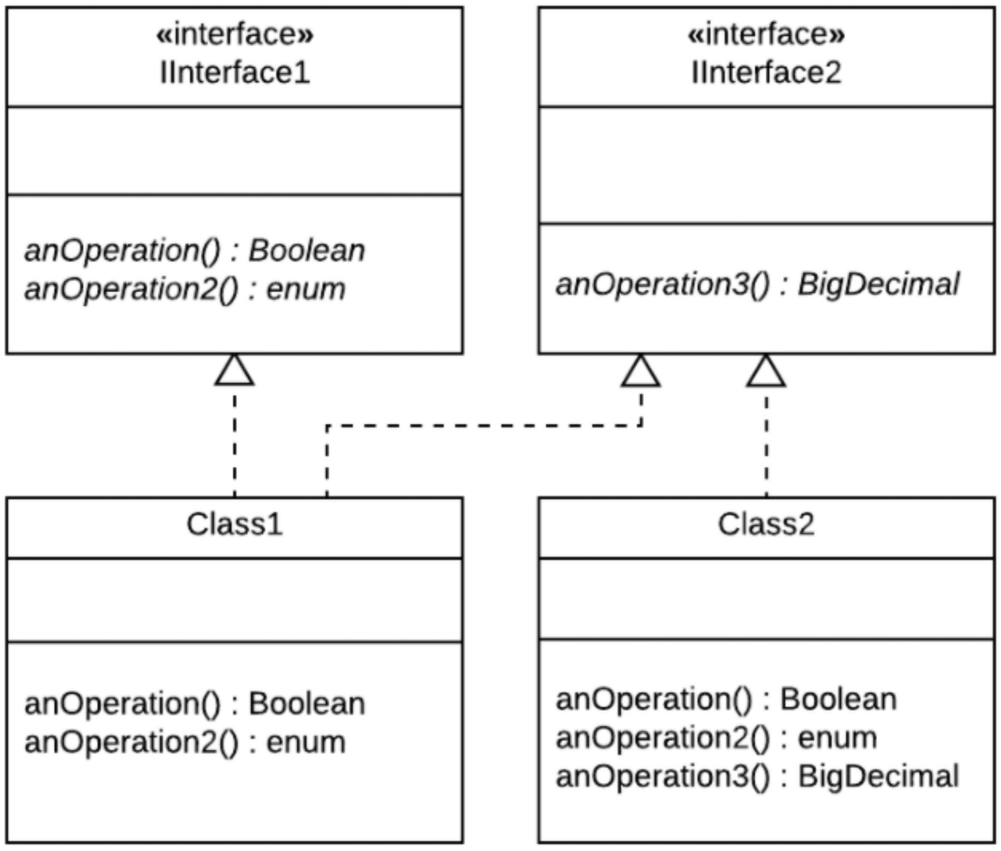
Question 5

Which of these UML diagrams demonstrates the Interface Segregation principle?

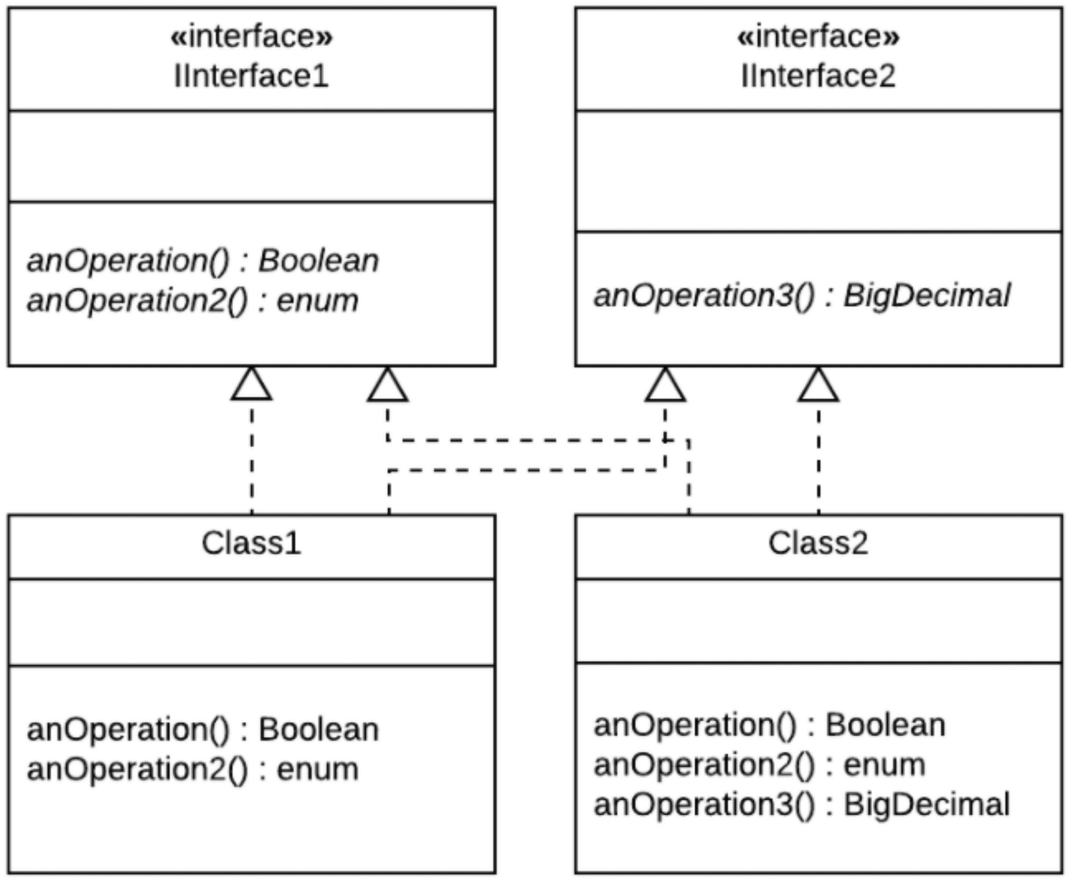
a)


«interface»
IInterface1
anOperation() : Boolean
anOperation2() : enum
anOperation3() : BigDecimal
Class1
anOperation() : Boolean
anOperation2() : enum
anOperation3() : void
Class2
anOperation() : Boolean
anOperation2() : enum
anOperation3() : BigDecimal

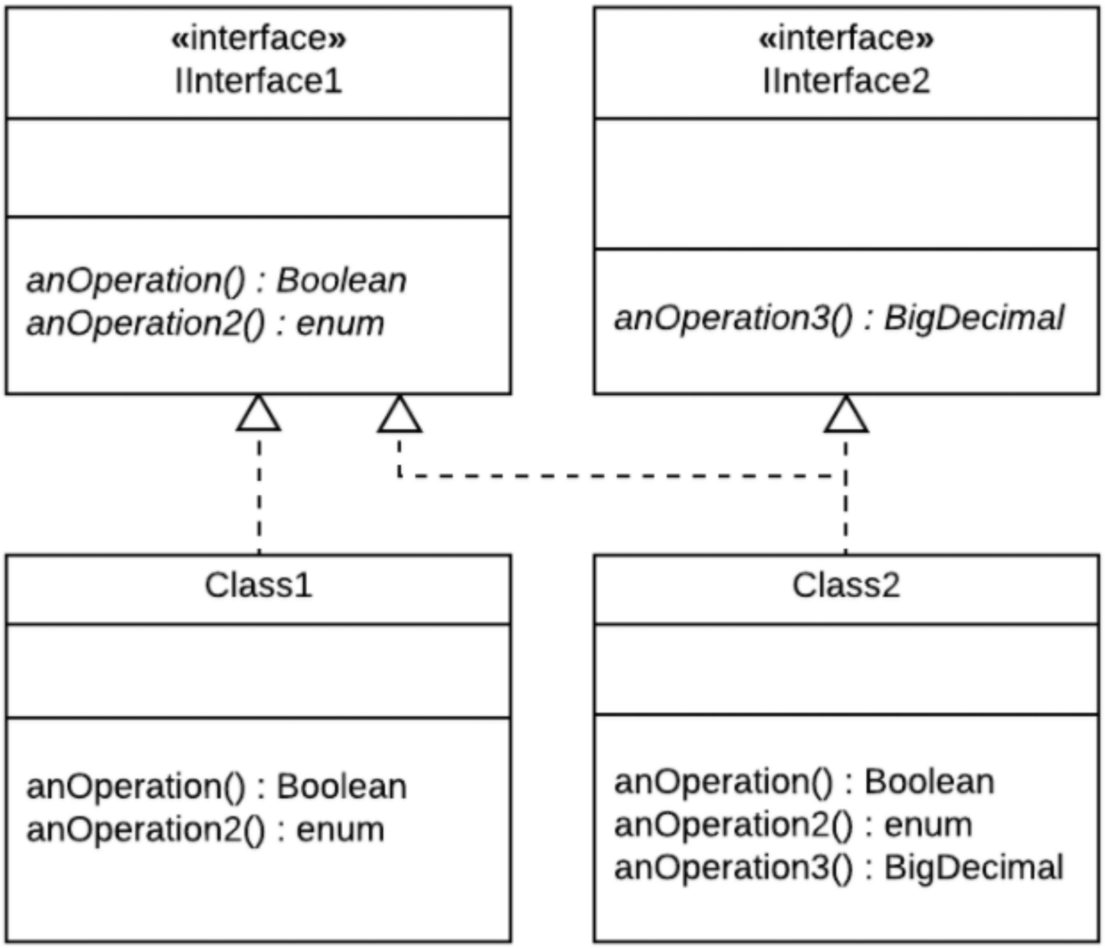
b)


«interface»
IInterface1
anOperation() : Boolean
anOperation2() : enum
«interface»
IInterface2
anOperation3() : BigDecimal
Class1
anOperation() : Boolean
anOperation2() : enum
Class2
anOperation() : Boolean
anOperation2() : enum
anOperation3() : BigDecimal

c)


«interface»
IInterface1
anOperation() : Boolean
anOperation2() : enum
«interface»
IInterface2
anOperation3() : BigDecimal
Class1
anOperation() : Boolean
anOperation2() : enum
Class2
anOperation() : Boolean
anOperation2() : enum
anOperation3() : BigDecimal

d)



☐ a

Incorrect

Incorrect. This diagram shows the situation without interface segregation. The 3rd method is not consistent with the interface in Class1, indicating that Interface Segregation may be useful here.

☐ b

Incorrect

Incorrect. From the provided methods, Class2 is the class that should implement both interfaces, yet Class1 is shown to implement both interfaces.

☐ c

Incorrect

Incorrect. If you look at Class1's methods, it only needs one of these interfaces. Code which followed this diagram would produce a compile-time error.

☐ d

Correct

Correct! Class1 does not need all of the methods, so it makes sense to have two different interfaces.

Question 6

Which of these code examples **violates** the Principle of Least Knowledge, or Law of Demeter?

☐ 1

เรียก method obj อื่น

```
public class O {
    M I = new M();

    public void anOperation2() {
      this.I.N.anOperation();
    }
  }
```

Correct

Correct! In this example, the method call in the class (O) reaches through the object (I) to a method in another object (N). This is not local and therefore the Principle is violated.

☐ 2

```
public class Class1 {
    public void N() {
      System.out.println("Method N invoked");
    }
  }

  public class Class2 {
    public void M(Class1 P) {
      P.N();
      System.out.println("Method M invoked");
    }
  }
```

Incorrect

Incorrect. One rule states that a class (Class2) can call a method (N) in any object that is a parameter object (P) of that class.

☐ 3

```
public class O {
        public void M() {
          this.N();
          System.out.println("Method M invoked");
        }

        public void N() {
          System.out.println("Method N invoked");
        }
    }
```

Incorrect

Incorrect. One rule is that a method can call other methods in the same class.

☐ 4

```
public class P {
      public void N() {
        System.out.println("Method N invoked");
      }
    }

    public class O {
      public void M() {
    P I = new P();
    I.N();
        System.out.println("Method M invoked");
      }
    }
```

Incorrect

Incorrect. One rule says that a method (M) can call a method (N) of an object (I) if it is instantiated within M.

Question 7

How can Comments be considered a code smell?

0 / 1 point

- ☐ **They can't! Comments help clarify code.**

  Incorrect

  Incorrect. While Comments can and should be used to clarify code, excessive commenting can be a sign that the design is not coherent or that the language is being used inappropriately.

- ☐ **Excessive commenting can be a coverup for bad code**

  Correct

  Correct! Sometimes, developers use excessive comments like a "deodorant" for bad code, instead of fixing the code.

- ☐ **When a comment is used to explain the rationale behind a design decision**

  Incorrect

  Incorrect. This sounds like a good use of comments, which should supplement a good design but should not be used as a replacement for making clear code.

- ☐ **Too many comments make the files too large to compile.**

  Incorrect

  Incorrect. The comments are not compiled!

Question 8

What is the primitive obsession code smell about?

0 / 1 point

- ☐ **Code that contains many low-level objects, without using OO principles like aggregation or inheritance.**

Incorrect

Incorrect. Think of what primitive means in development!

- ☐ **Overuse of primitive data types like int, long, float**

Correct

Correct! Excessive use of primitives may mean that you are not identifying appropriate abstractions.

- ☐ **Using many different primitive types instead of settling on a few that together capture that appropriate level of detail for your system.**

Incorrect

Incorrect. Think of how else you may be using primitives inappropriately.

- ☐ **Using key-value pairs instead of abstract data types.**

Incorrect

Incorrect. While this could be a problem as well, it does not refer to primitive obsession!

Question 9

You have a class that you keep adding to. Whenever you add new functionality, it just seems like the most natural place to put it, but it is starting to become a problem! Which code smell is this?

0 / 1 point

- ☐ **Long Method**

Incorrect

Incorrect. We are not just dealing with one method getting more complex. The class itself is getting more methods.

- ☐ **Large Class**

Correct

Correct! This class may also be called a blob class, God class, or black-hole class.

☐ **Divergent Change**

Incorrect

Incorrect. Divergent change is a related problem, since more responsibilities are added to a class, but tends to be more associated with lowering cohesion.

☐ **Speculative generality**

Incorrect

Incorrect. Speculative generality is when you code things that are not needed right now.

Question 10

Why is it important to avoid message chains whenever possible?

0 / 1 point

☐ **If an unexpected object is returned, this could easily lead to runtime errors.**

Incorrect

Incorrect. This is not the primary reason to avoid message chains.

☐ **They lower cohesion in your class.**

Incorrect

Incorrect. They might, but this is not the primary concern with message chains.

☐ **It's a workaround to get to private methods, which are important for encapsulation.**

Incorrect

Incorrect. Private methods can still only be called from within their class.

☐ **The resulting code is usually rigid and complex.**

Correct

Correct! Code with message chains is more difficult to not only maintain, but also to read. They will require Shotgun Surgery when changes need to be made.

Question 11

Look at the code snippet. Which code smell do you detect?

```
public class Class1 {

    ...

      public void M(Class2 C) {
        C.doSomething(x);
        C.foo(y);
        C.foo2(z, i);
      }
    }
```

- ☐ **Long Parameter List**

Incorrect

Incorrect. We can't see an example of a Long Parameter list smell here!

- ☐ **Feature Envy**

Correct

Correct! The method M calls lots of methods in the object C. Perhaps it would be better to have this method in that object.

- ☐ **Inappropriate Intimacy**

Incorrect

Incorrect. Inappropriate intimacy is a two-way, highly-coupled relationship between classes.

- ☐ **Divergent Change**

Incorrect

Incorrect. Divergent change will produce a class with low cohesion. We cannot see enough of this class to determine that!

Question 12

Joseph was developing a class for his smartphone poker game, and decided that one day he would like to be able to change the picture on the backs of the cards, so he created a Deck superclass. Since his app does not have that feature yet, Deck has only one subclass, RegularDeck. What code smell is this?

0 / 1 point

☐ **Refused Bequest**

Incorrect

Incorrect. Refused bequest refers to having methods or variables in the superclass that are not needed in the subclass!

☐ **Divergent Change**

Incorrect

Incorrect. Divergent change would mean that a class is changing in different ways, lowering its cohesiveness.

☐ **Speculative Generality**

Correct

Correct! Coding for anticipated needs instead of the current ones is not good Agile Development.

☐ **Primitive Obsession**

Incorrect

Incorrect. Primitive obsession is the overuse of primitive data types, instead of using abstract data types that would make the purpose of the code more clear