



Building Blocks of OOP. Part 2

[Home](#) > [Course](#) > [Building Blocks of OOP. Part 2](#) > [Quiz](#) > [Questions](#)


COURSE

DISCUSSION

Questions

16/16 points

What is the composition?


- ☐ Programming paradigm
- ☐ Design pattern
- ☒ Fundamental concept in OOP 

Answer

Correct:

That's correct, composition is one of the fundamental concepts in object-oriented programming. It describes a class that references one or more objects of other classes in instance variables. This allows you to model a has-a association between objects.

What is the main difference between composition and inheritance?

- ☐ Composition is not a part of OOP paradigm.
- ☒ Composition uses different relation type. 

☐ Composition is a type of inheritance.

Answer

Correct:

That's correct, inheritance models 'is-a' relation but composition models 'has-a' relation when composing object 'has-a' composed object.

What is the main benefits of inheritance?

- ☐ Hierarchies are easy to extend.
- ☐ Hard to make changes at the top of incorrect hierarchy.
- ☐ Follows open-closed principle.
- ☐ Does not work when trying to solve wrong kind of problem.



Answer

Correct:

That's true, inheritance allows you to easily extend the class behaviour without modifying existing code, this is pretty safe way to get the needed result.

Correct, when the encapsulation is not violated and classes are inherited in a correct way OCP will not be violated.

What is the main benefits of composition?

- ☐ Lot of small objects make program understanding less obvious.
- ☐ Enforces usage of interfaces which makes easier to add new parts.
- ☐ Enforces creation of small objects with clear interfaces and responsibilities.
- ☐ Composed object must explicitly know which messages to delegate and to whom.





Enforces single responsibility principle with transparent and easy to understand code.



Answer

Correct:

Interfaces need to be used when some part depends on (composes) other part, so it also helps to develop the new functionality.

That's true, small objects are easy to understand and maintain, so it is definitely a benefit.


As we have small objects they are always responsible only for a single concern, so it is pretty easy to meet the SRP with such approach.

What is duck type?



It is a specific interface which is shared among different classes.



It is a role which can be applied to some class in some moment of time. 



It is an abstract class which meets specific requirements related to its interface.

Answer

Correct:

Correct, duck type is not related to the interfaces or abstract classes, it is only about the specific class which plays specific role in specific moment of time.

Which of the following can indicate the hidden duck type?



Overloaded class functionality



Checking that some methods exists



'instanceof' operator usage



Case statement that switch on class



Different classes with similar interface



Answer

Correct:

It is also a good sign that there may be a hidden duck type when do not need to check the exact class but when you only need some specific method to exist.

This is similar to the switch..case situation, but here you check if you are working with instance of some specific class.

That's true, when you are trying to find out which exact class you are working with to apply specific steps to it, this may be a sign of hidden duck type.

What is mixins?

- ☒ It is an OOD tool to share some behavior between non-related objects. ✓
- ☐ It is an approach of specific inheritance usage to share some behavior.
- ☐ It is a code smell which indicates single responsibility principle violation.

Answer

Correct:

Correct, it is exactly what mixins are intended to do, while inheritance and composition working with some classes' hierarchy mixins can share needed behavior between different classes which are not directly related but need to play some specific role.

Which relation type mixins model?

- ☐ part-of
- ☒ behaves-as ✓
- ☐ has-a
- ☐ is-a



Answer

Correct:

That's correct, mixins tells us that some object under some circumstances 'behaves-as' needed under these circumstances.

Which hierarchy is the easiest to understand?

☐ Shallow, wide

☐ Deep, narrow

☒ Shallow, narrow ✓

☐ Deep, wide

Answer

Correct:

That's true, hierarchies of this type are not too big and can be easily read and extended.

The purpose of the law of Demeter is to:

☐ segregate interfaces

☐ lower the complexity

☐ delegate messages

☒ lower the cohesion ✓

Answer

Correct:

That's true, The Law of Demeter tells that you need talk only to your immediate neighbors and that's exactly the thing which will lower the cohesion.

^

What is design by contract?

☒ It is a software correctness methodology. ✓

☐ It is a design approach where interfaces usage is mandatory.

☐ It is a software development methodology where all the interfaces are thoroughly documented.

Answer

Correct:

Correct, it prescribes that software designers should define formal, precise and verifiable interface specifications for software components, which extend the ordinary definition of abstract data types with preconditions, postconditions and invariants.

Correctness can be applied to:

☐ function code

☒ function specification ✓

☐ preconditions

☐ postconditions

Answer

Correct:

That's true, we can only say if function is correct or incorrect when talking about expected results.

Function specification is fully correct when:

☒ both precondition and postcondition are met ✓

☐ when the function executes without errors thrown

☐

☐ when the postcondition is the strongest one

☐ when at least postcondition is met

Answer

Correct:

That's correct, it is the only case when we can say that the function is correct when we are talking about its results and both pre- and postconditions are met.

Condition P1 is stronger than P2 and P2 is weaker than P1 when:

☐ P1 is used as postcondition and P2 is used as a precondition

☒ the fulfillment of P1 implies the fulfillment of P2, and they are not equivalent ✓

☐ when P1 includes more options than P2

Answer

Correct: This is the most correct way to say which condition is stronger.

Which of the SOLID principles is the most closely related to preconditions and postconditions?

☐ Interface segregation principle

☐ Dependency inversion principle

☒ Liskov substitution principle ✓

☐ Open-closed principle

☐ Single responsibility principle

^

Answer

Correct:

That's correct, this principle says that you should be able to replace the base class with its subclass and the program should work correctly with it, or in other words it tells that pre- and postconditions should be met for the subclass.

When replacing the base class with its subclass which of the following conditions need to be met?

☐ The output values of subclass need to be contravariant.

☐ The input values of subclass need to be covariant.

☐ The input values of subclass need to be contravariant.

☐ The input and output values need to be equivalent.

☐ The output values of subclass need to be covariant.



Answer

Correct:

That's correct, when the input values are contravariant you can be sure that subclass will always have the same or weaker preconditions and can be used instead of base class.

Correct, when the output values are covariant or in other words when postcondition is the same or stronger you can be sure that the calling code will always get expected result.

Submit

 Show Answer

✓ Correct (16/16 points)

^ Navigation

Complete «Building Blocks of OOP. Part 2»

^

