Waukesha County Technical Institute
152-135 Advanced Java Class

DIP (Dependency Inversion) Checklist v1.00

When writing code use this checklist to make sure you are following the best practices presented. This may require some critical thinking and analysis so be prepared to use your brain and ***think*** about what you are doing and what could be done better. A glossary of terms is provided at the end of this doc.

How to Use the DIP:

Brief Summary

- ❑ Start by deciding the type of work you want your object to perform. Create the class and give it an appropriate name. Example: FoodServiceTipCalculator

- ❑ Next, ask yourself if you may eventually need other versions of this class, say for example, a BaggageServiceTipCalculator. This is only necessary if the work is done differently (if the common methods have different bodies (implementations). If you do this means you have variety in how the work gets performed. If not, you don't need the DIP.

- ❑ Base these classes on an abstraction (interface), such as "TipCalculator". Put common methods there as abstractions.

- ❑ Create a high-level class (a "boss" class) ending in the name "Service". For example "TipCalculatorService". Have the high-level class give orders to the low-level objects via abstraction. But don't forget to pass in the required worker objects(s).

Detailed Summary

Start With the Low-Level Classes

- Start by deciding the type of work you want your object to perform. This can result in one or more methods and properties. Create the class and give it an appropriate name. Example: FoodServiceTipCalculator

- Next, ask yourself if you may eventually need other versions of this class, say for example, a BaggageServiceTipCalculator. This is only necessary if the work is done differently (if the common methods have different bodies (implementations). If you do this means you have variety in how the work gets performed. If not, you don't need the DIP.

- Assuming you realize you need other types of low-level objects to perform the same work in a variety of ways (different method body implementations) then you need to make sure the important methods are named in a common way. For example the tip calculator examples named above might have a common "getTip()" method, but the bodies of those methods would vary because the forumulas for calculating tips are different. Now create an Interface that has those common methods. Give the interface an appropriate, generic name. For the examples above, you coud use "TipCalculator".

- For the common methods used there may be a need for some initialization of data. For example you can't calculate a tip without some data. In the food service example you would need the bill amount and a percentage representing service quality to calculate a tip. For the baggage service you would need the base tip amount plus the bag count plus a percentage representing service quality. These pieces of data should not be part of the common method parameter list. That's because the methods would have different signatures. You must keep the signatures the same so that the objects are polymorphic – so that they are interchangeable. So initialize you objects before they are used by using other, non-common methods, or the constructor.

Now Create A Service Class that acts as a High-Level Class

- Now that you have your low-level, worker classes you need a high-level class (the boss) to give orders to the workers. We call these "service" classes. For the above example we could create a

"TipCalculatorService". Any part of your program that needs the value of a tip should talk only to this class, never to the low-level classes.

- Now declare a component property for the low-level class you need, but do this by using the Liskov Substitution Principe. In other words, use the abstraction as the data type, so that it can represent any of several versions of your low-level classes. In the example above you would declare a property like this:

  private TipCalculator tipCalculator:

- Now you need getter and setter methods for this, and a constructor that lets you pass an instance of one of your low-level classes. Save this instance in your component property.

- Now create a method that commands the component to do something. For example, in the above we would create a method to "getTip()" and return a double. For the method body, delegate the work to the component, like this:

  return tipCalculator,getTip();

How to Use Your DIP Service

Now that you have your DIP architecture complete you need to put your service to work. First, you will need to instantiate one of your low-level classes using the Liskov Substitution Principle:

TipCalculator calc = new FoodServiceTipCalculator(100.00, 0.20);

Notice how the calculator object is initialized with data using the constructor. This data may vary between your low-level classes. That's ok. Only the common worker methods must be the same.

Next create your service object, passing the low-level object as a constructor argument:

TipCalculatorService tipService = new TipCalculatorService(calc);

Now you service is ready to use:

Double tipAmount = tipService.getTip();

Your service object should delegate the work of getting the tip amount to the low-level object you passed in, and return the amount to you.

Now to prove that your system is really flexible, change the TipCalculator to another option. For example:

TipCalculator calc = new BaggageServiceTipCalculator(2.00,5, 0.20);

Now run your app again. Notice that nothing broke and you switched calc objects without problems. That's flexibility.

Final Thoughts

In the TipCalculatorService example there was only one component – a TipCalculator. This will not always be the case. For example in the Copier example you saw earlier we need both a Reader and Writer component. The number of components needed by your service is completely up to you, based on requirements.

Glossary

High-Level Class

Any concrete class that gives orders to a component class (a low-level class). The high-level class does not do the work, rather it delegates the work order to the component class, which actually does the work. The component class should be referenced as a pure abstraction. A high-level class is always a normal, concrete class and should have a name that ends with "Service".

Low-level Class

Any concrete class that actually performs some work. It should be based on a pure abstraction, which means it should implement an

interface that dictates common methods for similar low-level classes. Two low-level classes that implement the same interface should be able to be used interchangeably.

DIP Abstraction

A pure abstraction – i.e., an Interface.

Details

The body (the implementation) of a method.