# Example 50 - Recall that Interfaces have only abstract methods, act as a supertype, and make guarantees on classes that implement them

If code could talk:

**Animal:** “I promise you that any concrete class that implements me must provide code for the method **makeSound()**. By default the method is **public**.”

|  |
| --- |
| public interface Animal {    String makeSound(); } |

**Cat:** “Since I implement **Anima**l, I must keep the promise and implement **makeSound()** exactly according to the signature given**.** If I do not, I will become an abstract class”

|  |
| --- |
| public class Cat implements Animal{  @Override  public String makeSound(){  return "meow";  } } |

**Animal a:** “I promise you that variable **a** can hold a reference to any object that implements **Animal** interface, and thus I promise you that **a** is able to execute the **makeSound()** method. ”

|  |
| --- |
| public static void main(String[] args) {   Animal a = new Cat();   a.makeSound();  } |

**Example 50a – When writing software you will have to deal with changing requirements**

Imagine that you are writing a software to print bus tickets based on the distance travelled on the bus route. You realize that the following tasks could change

* The way bus fares are calculated
* The information printed on the bus ticket

Based on what you have learnt so far, you would use inheritance and overriding to change the implementation of each of the methods when the need arises. You will soon get many classes with various combinations of the implementation of the two tasks, violating the following principle:

|  |
| --- |
| **OOP Design Principle**  **Single Responsibility Principle** – each class should have only one reason to change |

|  |
| --- |
| public class BusTicket {   private String busService;  private final int SHORTFARE = 100;  private final int LONGFARE = 120;   BusTicket(String busService){  this.busService = busService;  }   public int getBusFare(int distance){  return distance < 32? SHORTFARE : LONGFARE;   }  public String getTicketPrintout(int distance){  return "Service:"+busService + "Fare:" + getBusFare(distance);  } } |

**Example 51 – Depending directly on another concrete class is tight coupling and makes your code less flexible.**

To follow the **single responsibility principle**, the solution is to give the calculation of bus fares to another object. This is called **delegation**. Consider the following implementation. However, notice that the **BusTicket** class depends directly on **BusFareConcrete** class.

What if you are required to update bus fares? You can:

1. Modify **getBusFare()** in **BusFareConcrete** directly
2. Sub-class **BusFareConcrete** and override **getBusFare()**

|  |
| --- |
| public class BusTicket {  private String busService;  private BusFareConcrete busFare;   BusTicket(String busService, BusFareConcrete busFare){  this.busService = busService;  this.busFare = busFare;  }  public String getTicketPrintout(int distance){  return "Service:"+busService + "Fare:" + busFare.getBusFare(distance);  } } |

|  |
| --- |
| public class BusFareConcrete {  private final int SHORTFARE = 100;  private final int LONGFARE = 120;   public int getBusFare(int distance){  if( distance < 32){  return SHORTFARE;  }else{  return LONGFARE;  }  } } |

**Example 52 – Use interfaces/abstract class instead and this makes your code loosely coupled.**

**Modifying existing code directly is not encouraged** in a production situation as it would have been well-tested and integrated in the software, thus you would not want to risk causing the software to fail and annoy your stakeholders.

Sub-classing and writing a new method would be better, but it would be hard to justify the inheritance relationship. Also, inheritance does not FORCE you to override.

**To loosely couple to another class, make your instance variables depend on an Interface instead of a concrete class.** This gives you the flexibility to assign ANY concrete class to it, as long as it implements the Interface.

Here are the steps to make the code more flexible.

1. Write an interface **BusFare** with the necessary abstract method(s) for the calculation of bus fares.
2. Write a concrete class **BusFare2020** that implements the **BusFare** interface.
3. Declare an attribute in **BusTicket** of type **BusFare**, and adjust the constructor accordingly. This means that **BusTicket** can work with ANY object that implements **BusFare** Interface.
4. When bus fares change, write a new concrete class **BusFare2024** that implements the **BusFare** interface. Then simply change the object passed to the **BusTicket** constructor when bus fares change, and the information printed on the bus ticket is correct.

**Favour Composition over Inheritance**

Example 52 illustrates the use of **composition**, a **has-a relationship** between the classes, to make software more flexible è If you want to change the bus fare calculation, simply write a new class that implements the **BusFare** interface. **Loose coupling**



In this design, **BusTicket** only needs the guarantee of methods from the **BusFare** interface to function correctly, and does not need to depend on any specific concrete class. This is the “*Program to an interface, not an implementation*” principle.

If the calculation of bus fares change, all the software designer needs to do is to write a new class that implements the **BusFare** interface. Existing concrete classes need not be modified.

This design illustrates the following principle.

|  |
| --- |
| **OOP Design Principle**  **Open/Closed Principle** – classes should be open for extension and closed to modification |

* “Open for extension” - change the bus fare calculation? simply write a new class that implements the **BusFare** interface.
* “Closed to modification” – don’t modify existing code.

# Example 53 – Inheritance can lead to duplicated code

This example illustrates why inheritance is not the best solution to deal with changing requirements. Notice that as some birds can/cannot fly and can/cannot swim, each method is overridden accordingly, and this leads to many classes with duplicated code.

Furthermore, the coder has to deal with changing requirements – in this case, what if a bird can fly but can’t do it well (like a chicken) or can fly really long distances (like a migratory bird)?

|  |
| --- |
| public class Bird {   public void fly(){  System.*out*.println("I CAN fly");  }  public void swim(){  System.*out*.println("I can't swim");  } }  class DomesticDuck extends Bird{  @Override  public void fly(){  System.*out*.println("I can't fly");  }  @Override  public void swim(){  System.*out*.println("I CAN swim");  } }  class CanadaGoose extends Bird{  @Override  public void swim(){  System.*out*.println("I CAN swim");  } }  class Pigeon extends Bird{  @Override  public void fly(){  System.*out*.println("I CAN fly long distances");  }  }  class Ostrich extends Bird{  @Override  public void fly(){  System.*out*.println("I can't fly");  }  @Override  public void swim(){  System.*out*.println("I can't swim");  } } |

# Example 54 – Strategy Design Pattern – How do you make behaviours of classes flexible?

A **design pattern** is an existing solution to a known problem in OOP. You have seen how composition is used in Example 52. The Strategy Design Pattern uses the same idea.

The idea in the **Strategy Design Pattern** is to “**encapsulate what varies**” i.e. to make the flying behaviour into separate classes so that you can use composition to specify the behaviour that you want.

1. Write an interface that specify what method **FlyBehaviour** objects will have.

|  |
| --- |
| public interface FlyBehaviour {  void doFly(); } |

1. Write as many concrete classes that implement **FlyBehaviour** as required. (*Single Responsibility Principle*)

|  |
| --- |
| public class CanFly implements FlyBehaviour{  @Override  public void doFly() {  System.*out*.println("I CAN fly");  } } |

1. Rewrite the **Bird** class to have (i) instance variable to store **FlyBehaviour** objects (ii) a setter to assign such objects to the instance variable (this can also be specified at the constructor)
2. Rewrite the **fly()** method to execute the method that the **FlyBehaviour** interface guarantees. (*Program to an interface*)

|  |
| --- |
| public class Bird {   private FlyBehaviour flyBehaviour;   public void setFlyBehaviour(FlyBehaviour flyBehaviour){  this.flyBehaviour = flyBehaviour;  }   public void fly(){  flyBehaviour.doFly();  } } |

The benefit of the Strategy Design Pattern:

* Since you encapsulate the various behaviours, the client can change them during runtime if needed
* If there are new requirements or specifications for the behaviour, a new class can simply be written and passed to the setter (*Open/Closed principle*)

# Example 55 – Observer design pattern – how do you send messages to objects that depend on your data?

**Recall loose coupling - make your instance variables depend on an Interface instead of a concrete class.**

Suppose you had a number of classes who need data from an external source for their own calculations. You could write the code to pull that data in the classes, but the code is likely to be duplicated across all the classes.

The solution is to apply the **Single Responsibility Principle** and **separate the concerns** i.e. one class to pull the data, the other classes then focus on the calculations and **loosely couple** them together. This is the observer design pattern. Here’s how.

You have an object (the “**Subject**”) whose responsibility is to

* get some data from a source, e.g. sensors, database.
* update one or more objects (the “**Observer**”) which depend on the data

è Next page

The **Observer** (eg user of data) would have one method to be executed when it receives the data. The programmer is free to specify what the concrete classes will do with this data.

This provides a guarantee that when data comes, this method will be executed.

|  |
| --- |
| public interface Observer{  void update(double airPollutionIndex); } |

The **Subject (data source)** would have to

* hold references to the **Observer** objects, passed through the **register()** method below
* notify these **Observer** objects when a change of data happens, by executing notifyObservers();

|  |
| --- |
| public interface Subject {  void register(Observer o); // come to me if you want  void unregister(Observer o);  void notifyObservers(); // when I get data I will tell you  } |

The Observer pattern is used in GUI frameworks to execute code written when a button click is detected.

* The **Observer** is the class which contains code to be executed when the button is clicked, designed by the programmer.
* The **Subject** is the class for the Button. When a button click is detected, the **Observer** is notified and executed.

# Example 56 – Singleton Design Pattern – How do you ensure that only one instance is available in the software?

Sometimes, you want a class to have only ONE instance in your software, examples are:

* when a class talks to a database, so that you do not want to handle simultaneous connections from more than one object.
* When a class is for logging

Limited situations where this is **useful** and **justified.**

1. Write a constructor but declare the constructor private. This prevents instantiation through the constructor and also prevents subclassing.
2. Declare a static instance variable of your own class.
3. Write a static method which has code that: (i) if the instance variable is null, call the Constructor, else return the instance variable.

|  |
| --- |
| public class Connection {   private static Connection *connection*;   private Connection(){}   public static Connection getInstance(){  if( *connection* == null){  *connection* = new Connection();  }  return *connection*;   }  public String getData(){  return "here's some data";  } } |

# Summary

You should be aware of the basic object-oriented principles.

* Abstraction
* Encapsulation / Information-hiding
* Inheritance
* Polymorphism

However, while fundamental, these are not enough to help you write software that is flexible and can adapt to changing requirements.

There are five principles organized as the SOLID principles that guide OOP design, as listed by Robert C. Martin. We have discussed two of them here.

|  |
| --- |
| **OOP Design Principles (SOLID)**  **Single Responsibility Principle** – each class should have only one reason to change  **Open/Closed Principle** – classes should be open for extension and closed to modification |

We will leave you to discover the other three for yourself. Suggested reading: <https://blog.devgenius.io/contribution-of-solid-principle-to-low-coupling-and-high-cohesion-pattern-973c33574572>

There are also other design principles that are not part of SOLID, but stated in similar ways:

* Separation of concerns
* Encapsulate what varies
* **Favour composition over inheritance**
* Program to an interface, not an implementation
* **Strive for a loosely coupled design**