Software Development

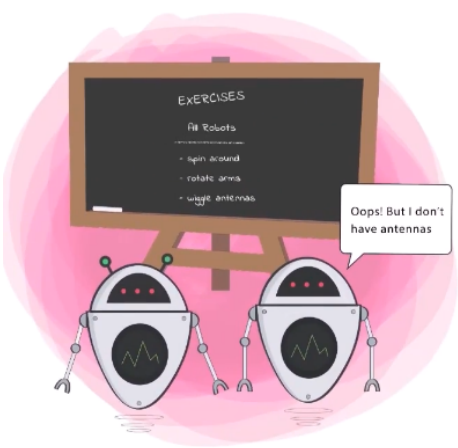
Dr. Hamada I. AbdulWakel[1]

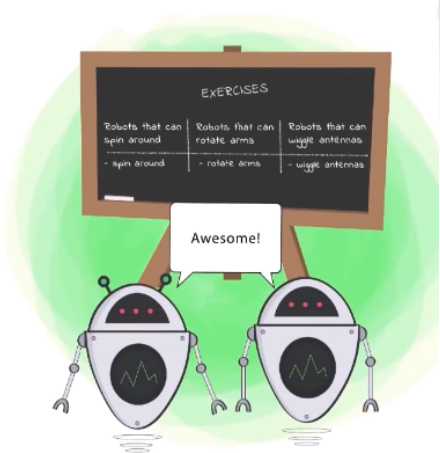
[1]Computer Science Department

2022 - 2023

Ch4: Solid Principles Lec#6

Interface Segregation Principle (ISP)





Interface Segregation

Interface Segregation Principle (ISP) ...

Problem

```
public interface ParkingLot{
   void parkCar(); //decrease empty spot by 1
   void unparkCar(); // increase empty spot by 1
   void getCapacity(); // returns car capacity
   double calculateFee(Car car); // returns price * hours
   void doPayment(Car car);
}
```

Interface Segregation Principle (ISP) ...

Problem ...

```
public class FreeParking implements ParkingLot{
   @Override
   void parkCar(){ .... }
   @Override
   void unparkCar(){ .... }
   @Override
   void getCapacity(){ .... }
   @Override
   double calculateFee(Car car){    return 0;    }
   @Override
   void doPayment(Car car){
      throw new Exception("Parking is free");
   }
}
```

Interface Segregation Principle (ISP) ...

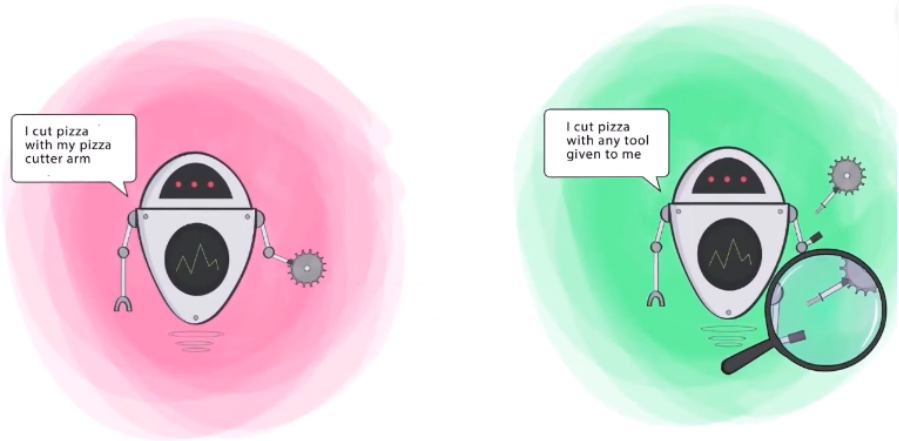
Solution

```
public interface ParkingLot{
   void parkCar(); //decrease empty spot by 1
   void unparkCar(); // increase empty spot by 1
   void getCapacity(); // returns car capacity
}
public interface PaidParkingLot implements ParkingLot{
   double calculateFee(Car car); // returns price * hours
   void doPayment(Car car);
}
public class FreeParking implements ParkingLot{ ..... }
```

Interface Segregation Principle (ISP) ...

- ISP splits interfaces **“fat interfaces”** that are very large into smaller and more specific ones.
- Clients will only have to know about the methods that are of interest to them.

Dependency Inversion Principle (DIP)



Dependency Inversion Principle (DIP) ...

Problem

```
public class DeliveryDriver{
  public void deliverProduct(Product product){ .... }
}
public class DeliveryCompany{
  public void sendProduct(Product product){
      DeliveryDriver delivery = new DeliveryDriver(); // dependency!!!
      delivery.deliverProduct(product);
  }
}
```

Dependency Inversion Principle (DIP) ...

Solution

```
public interface DeliveryService{
   public void deliverProduct(Product product);
}
//low level model
public class DeliveryDriver implements DeliveryService{
   @Override
   public void deliverProduct(Product product){ .... }
}
```

Dependency Inversion Principle (DIP) ...

Solution ...

```
//high level model
public class DeliveryCompany{
  private DeliveryService deliveryservice;
  public DeliveryCompany(DeliveryService deliveryservice){
    this.deliveryservice = deliveryservice;
  }
  public void sendProduct(Product product){
    this.deliverservice.deliverProduct(product);
  }
}
```

Ch5: Design Patterns
Self-study

Ch6: Creational Design Patterns

Singleton Design Pattern

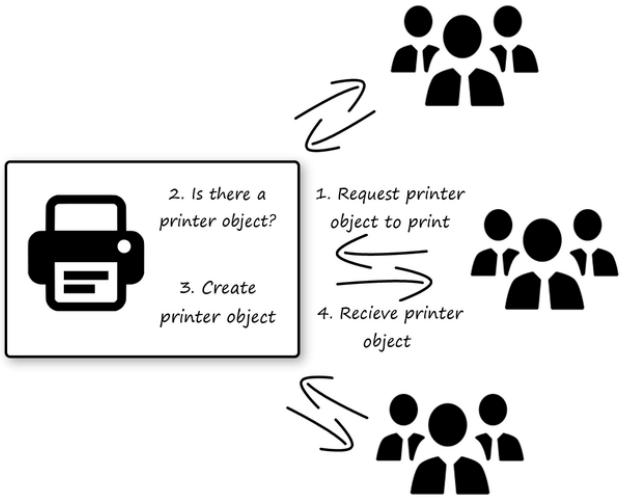
Problem

When dealing with objects that contain **'live data'**, it becomes problematic. Each object has its own copy of the data.

Solution

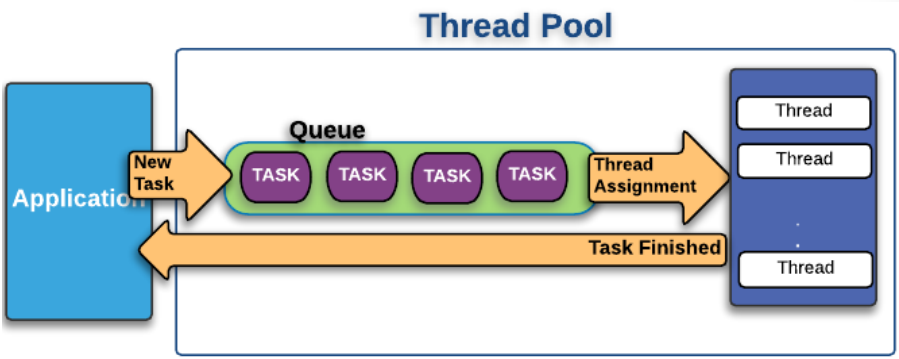
A one-way access to a class, which ensures that whenever this class is retrieved, that it will always be the same instance.

Singleton Design Pattern Examples

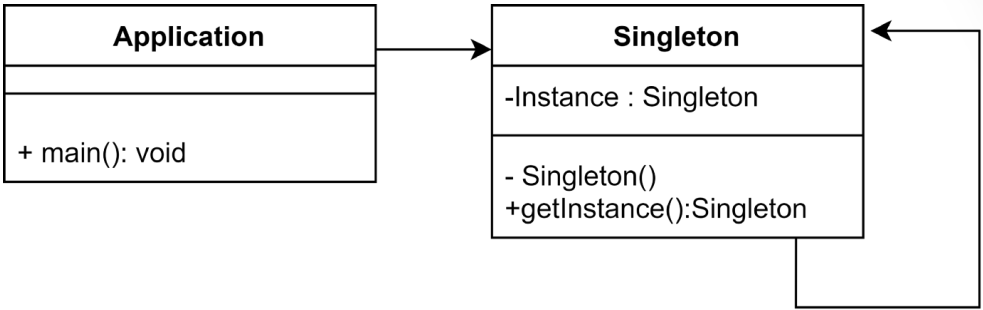


1. Only one instance of the object available to the whole system.
2. No additional arguments used to distribute the object around in the system.

Singleton Design Pattern Examples ...



Singleton Design Pattern UML



Approaches of Singleton Implementation

1- Eager initialization

```
public class EagerInitializedSingleton {
   // created at the time of class loading
   private static EagerInitializedSingleton instance =
                                        new EagerInitializedSingleton();
   private EagerInitializedSingleton(){
      // Do a very expensive task (memory-wise or computation-wise)
   }
   public static EagerInitializedSingleton getInstance(){
     return instance;
   }
}
```

Approaches of Singleton Implementation ...

2- Static block initialization

```
public class StaticBlockSingleton{
   private static StaticBlockSingleton instance;
   private StaticBlockSingleton(){}
   static{
     try{    instance = new StaticBlockSingleton();    }
     catch(Exception e){
          throw new RuntimeException("Creating another instance"); }
   }
   public static StaticBlockSingleton getInstance(){    return instance;    }
}
```

Approaches of Singleton Implementation ...

3- Lazy initialization: To solve the problem of object loading without using.

```
public class LazyInitializedSingleton {
   private static LazyInitializedSingleton instance;
   private LazyInitializedSingleton(){}
   public static LazyInitializedSingleton getInstance(){
     if(instance == null){    instance = new LazyInitializedSingleton();    }
     return instance;
   }
}
```

Approaches of Singleton Implementation ...

4- Thread safe: To solve multithreaded systems problem. This solution decreasing the performance 100 times.

```
public class ThreadSafeSingleton {
  private static ThreadSafeSingleton instance;
  private ThreadSafeSingleton(){}
  // the function converted to critical section
  public static synchronized ThreadSafeSingleton getInstance(){
    if(instance == null){    instance = new ThreadSafeSingleton();    }
    return instance;
  }
}
```

Approaches of Singleton Implementation ...

5- Thread safe 2: Provides good performance using double checked locking.

```
public class ThreadSafeSingleton {
  private static ThreadSafeSingleton instance;
  private ThreadSafeSingleton(){}
  public static ThreadSafeSingleton getInstanceUsingDoubleLocking(){
    if(instance == null){
      //class level synchronization
      synchronized (ThreadSafeSingleton.class) {
          instance = new ThreadSafeSingleton();
      }
    }
  return instance;
  } }
```

Approaches of Singleton Implementation ...

6- Core cache safe: To solve the problem of main memory variable update.

```
public class ThreadSafeSingleton {
  private static volatile ThreadSafeSingleton instance;
  private ThreadSafeSingleton(){}
  public static ThreadSafeSingleton getInstanceUsingDoubleLocking(){
   if(instance == null){
      synchronized (ThreadSafeSingleton.class) {
           if(instance == null){    instance = new ThreadSafeSingleton();    }
      }
   }
  return instance;
  }
}
```