THE UNIVERSITY OF SYDNEY

SOFT2201/COMP9201 Week 2 Tutorial

OO Theory in Java

Inheritance

Classes

Classes are a fundamental construct within java, they allow the programmer to aggregate data and functionality under a defined type. This feature isn't just for reusability and simplicity, we are able to safely read and write to memory by ensuring we adhere to the type system rules and satisfy the constraints enforced by the compiler.

Within Java, all classes inherit from the `Object` type, providing a definition for `toString` and `equals` which can be overridden.

Interfaces and Abstract Classes

Interfaces within java allow the programmer to create a type specifying abstract methods. When we specify that a class implements an interface, it must ensure a method body for each method specified in the interface. However, there are some exceptions, the interface may a defined static method or a default method.

```
interface Drivable {
    public boolean start();

    public boolean stop();

    public void move();
}
```

In the `Drivable` example, any type that implements this interface must also implement `start`, `stop` and `move`, allowing us to categorise objects of different types under one. Within our application, types such as `Car`, `Truck` and `Train` can implement the `Drivable` type.

Similar to an `interface`, an `abstract class` can be used to provide a type association with common properties, however within Java, a class is limited to inheriting from just one class where it can implement many interfaces.

- If our application contained a `Car`, `Truck` and `Train`, would we group these classes using an interface or abstract class?

Question 1: Polymorphism

You have been given the following java code, you may assume that the program compiles successfully.

```
interface Animal {
    void talk();
}

public class Zoo {

    public static void makeNoise(List<Animal> animals) {
        for(Animal a : animals) {
            a.talk();
        }
    }

    public static void outputToString(List<Object> objects) {
        for(Object a : objects) {
            System.out.println(a);
        }
    }

    public static void addAnimal(List<Animal> animals, Animal a) {
        animals.add(a);
    }
}
```


- The following classes: `Lion`, `Wolf`, `Bear` and `Cat`, implement the `Animal` interface, what method are they required to implement?
- When we invoke the method `makeNoise`, what object types can be contained within parameter `animals`?
- When we invoke the method `outputToString`, what object types can be contained within parameter `objects`?
- Can we call `talk` for each object in the outputToString method?
- When we invoke the method `addAnimal`, what object types can be contained within parameter `animals` and what types can be passed to `a`?

Question 2: Mapping out the heirarchy

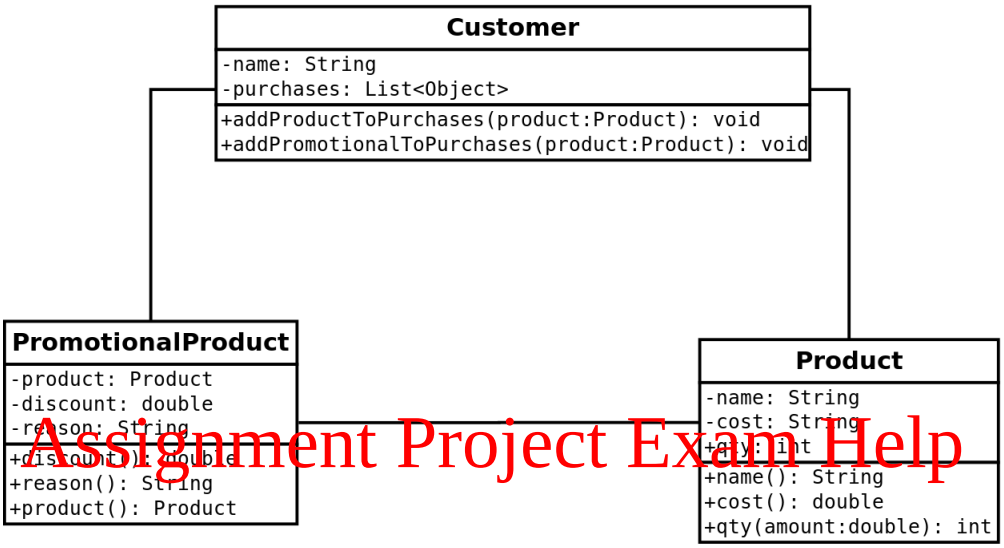
Given the following code, draw an object graph, noting the inheritance heirarchy with a fellow class member.

```
//Cabinet.java
public class Cabinet {
    public boolean broken;
    public boolean assembled;

    public Cabinet() {
        broken = false;
        assembled = false;
    }
}

//Tool.java
public abstract class Tool {
    public abstract void use(Cabinet c);
}

//Hammer.java
public class Hammer extends Tool {
    public void use(Cabinet c) {
        c.broken = true;
    }
}

//Mallet.java
public class Mallet extends Hammer {}

//Screwdriver.java
public class Screwdriver extends Tool {
    public void use(Cabinet c) {
        c.assembled = true;
    }
}
```

Question 3: Refactoring design

You will be asked through the semester to identify software design issues and refactor them. Not only will this improve your ability to debug problems but will help understand and identify issues within a code base.

As a class, identify issues with the following design



SOLID

SOLID is a software design acronym that incorporates five principles to assist with designing flexible, maintainable and robust software. These typically push for application structure to be contain types which can act independently, depend on abstractions and can be easily refactored.

Single Responsibility Principle

A class should be assigned a single responsibility, avoiding a class performing more roles than it should. If you find that you are assigning multiple roles to a class you can break the these roles into separate classes.

Open-closed Principle

Open-Closed principle outlines that an element is open for extension but closed for modification. Extension can be typically inferred as class inheritance but it is also possible to extend a class through composition or traits (interface with default method).

Liskov Substitution Principle

This principle infers that types should be replacable with instances of their subtypes. It asserts that a class should utilise an interface and allow for an instance of that interface to be substituted.

Interface Segregation Principle

Interface segregation asks to break up large interface types. Designing the application to use many client-specific interfaces over large general purpose interface. Each interface should focus on the functionality required instead of a large interface providing methods that some objects may never implement.

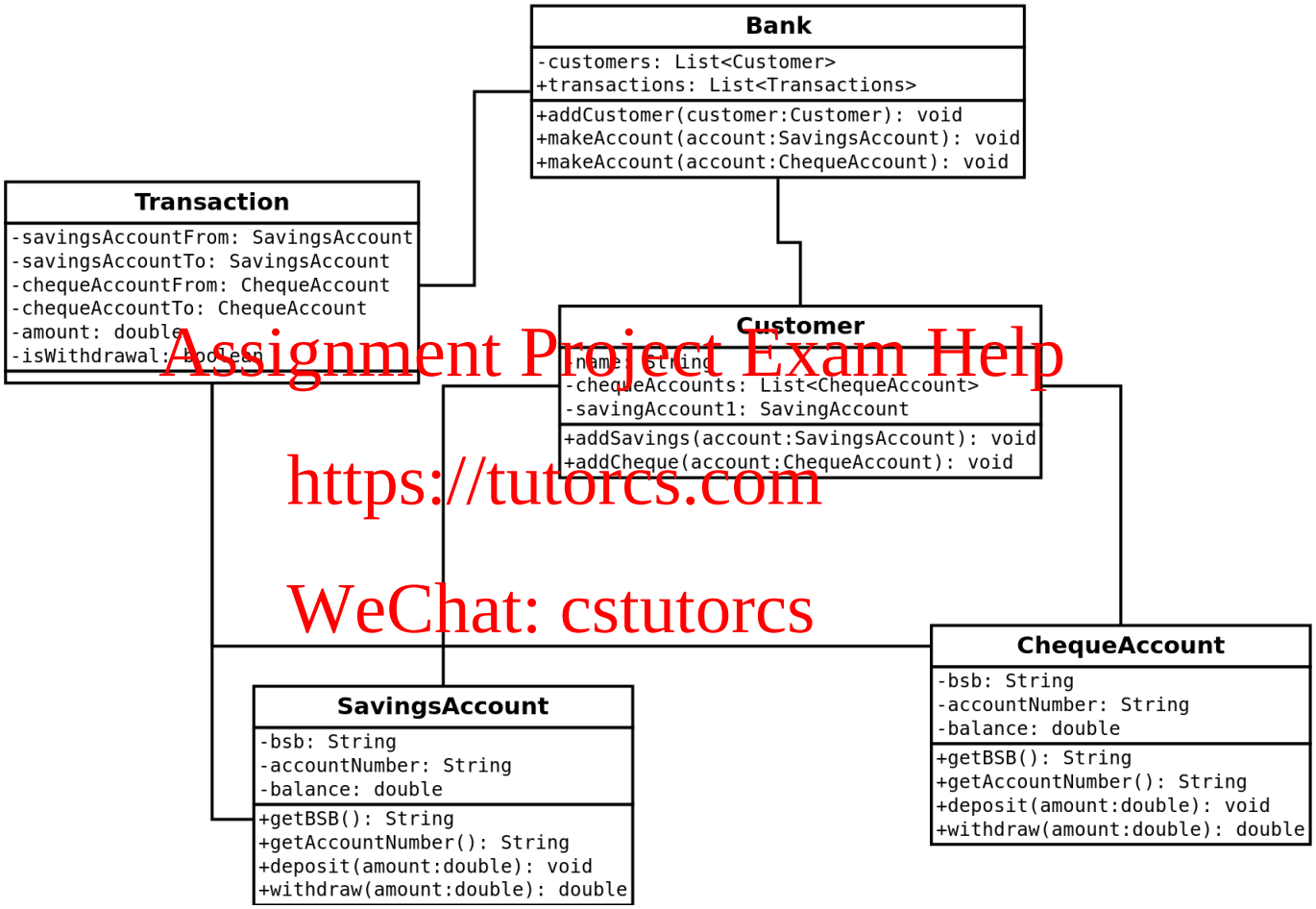
Dependency Inversion Principle

Dependency inversion principle places the constraint that a type should never depend on another concrete type. Allow your types to depend on abstractions, the type binding can be updated and substituted when refactoring.

Question 4: Rule Violations

Before you move to this question, make sure you understand the meaning of each principle in SOLID on page 5. Feel free to discuss any confusion with your tutor or group members.

We want to be able to identify a *design smell*, this is typically when a piece of code is considered unsound or inflexible. Inflexible or restrictive code may require a large refactoring effort if not identified and addressed early. Discuss with the class or with a group where the SOLID principles have been violated with the following application.



Command-query Separation

Command-query separation principle within imperative and object-oriented programming asks the programmer to design their methods to only perform a command or a query. A command mutates and operates on the data provided or the instance itself. A query will not mutate data but provide an answer when invoked.

Question 5: Refactor the code

Identify methods which violate command-query separation and refactor the code, this may require breaking the code up into different functions.

```
public class Product {

    private String name;
    private int qty;
    private int max;
    private double price;

    /**
     * Constructs the product providing name and quantity
     * @param name
     * @param qty
     */
    public Product(String name, int qty, int max, double price) {
        this.name = name;
        this.qty = qty;
        this.max = max;
        this.price = price;
    }

    public Object process(String request, double value) {
        if(request.equals("getqty") {
            return new Integer(qty);
        } else if(request.equals("updateqty")) {
            qty += (int) value;
            return null;
        }  else if(request.equals("getprice")) {
            return new Double(price);
        }  else if(request.equals("updateprice")) {
            price += price;
            return null;
        }
        return null;
    }
```

```
    /**
     * Returns the name of the product
     * @return name
     */
    public String getName() {
        return name;
    }

    /**
     * Passes the order
     * @param o
     */
    public void order(Order o, int qty) {
        o.addProduct(this, qty);
    }

}
```

- What kind of issues does the method `process` create?
- Is it the responsibility of the `Product` class to perform this kind of logic?
- Refactor the code and separate any logic that does not belong in the `Product` class