**ANALYSIS OF CODE**

1. **Critical analysis of OOP Principles**

**1.) Favour composition over inheritance**

In our code, we have used the concept of favouring composition over inheritance. The project states that we need to make a separate class for each “Specific” part and based on the data of the specific parts our entire program will run. Now, it is not possible to inherit (or to “extend”) all the classes as that will lead to very less readability and code reuse will also get hampered to some extent. (There are total 22 classes for specific parts!) Moreover Java does not allow multiple inheritance using extends keyword so our program functionality would have halted if we were not using composition. Now, composition is basically having a reference to the different classes (whose property we want to use) using instance variables or creating those class’s objects in the required class.

Code snippet for explaination:

**public** **class** Factory

{

**public** **static** **void** main(String[] args)

{

**final** **long** startTime = System.*nanoTime*();

Scanner sc = **new** Scanner(System.***in***);

System.***out***.println("Welcome to the Car Production Factory!");

System.***out***.println("Enter the number of Assembly Lines");

**int** numAssembly = sc.nextInt();

ArrayList<Car> Cars = **new** ArrayList<Car>(numAssembly);

//Code For Taking Input of Car Specifications and Assembling on different threads

// each Assemblies[i] will be implementing runnable and having a separate thread to itself

**for**(**int** i = 0; i < numAssembly; i++)

{

System.***out***.println("Enter Specifications for Car Number " + (i+1) + " :- ");

**int** chassis, engine, tyre, suspension,ac,paint;

Chassis c=**null**;

Engine e=**null**;

Tyre t=**null**;

Suspension s=**null**;

Air\_conditioning a=**null**;

Paint p=**null**;

System.***out***.println("Enter type of Chassis (1-4) : ");

chassis = sc.nextInt();

**switch**(chassis)

{

**case** 1:

c=**new** Ladder();

**break**;

**case** 2:

c=**new** Backbone();

**break**;

**case** 3:

c=**new** Spacy();

**break**;

**case** 4:

c=**new** Cadillac();

**break**;

**default**:

System.***out***.println("Wrong chassis number entered.");

}

The code snippet above uses a class Factory which drives the entire program. Inside it we are taking input from user regarding the specifications he/she wants in the car. Now, when the user is asked to input the number for chassis (and similarly for other parts as well), the switch case directs the interface object “c” to refer to the Ladder, Backbone, Spacy or the Cadillac class when the user enters 1,2,3,4 respectively. So here we refer to the mentioned classes to use their properties without actually inheriting them. This is how we have favoured composition over inheritance.

**2.) Program to interface not implementation**

Programming to interface eases the effort to change our code whenever there are certain modifications required in our code. Coding to implementation increases the lines of code making it take more reading time and also lots of changes are required.

Code snippet:

**for**(**int** i = 0; i < numAssembly; i++)

{

System.***out***.println("Enter Specifications for Car Number " + (i+1) + " :- ");

**int** chassis, engine, tyre, suspension,ac,paint;

Chassis c=**null**;

Engine e=**null**;

Tyre t=**null**;

Suspension s=**null**;

Air\_conditioning a=**null**;

Paint p=**null**;

In our code we have a total of 6 interfaces representing the “Basic parts”. Each interface has two methods to return the part number and the time taken to build. Now, there are the specific parts which implement the interfaces and provide the required information. Now, the code snippet above is from class “Factory”. Inside for loop we have used coded to an interface by declaring the interface objects.

System.***out***.println("Enter type of Chassis (1-4) : ");

chassis = sc.nextInt();

**switch**(chassis)

{

**case** 1:

c=**new** Ladder();

**break**;

**case** 2:

c=**new** Backbone();

**break**;

**case** 3:

c=**new** Spacy();

**break**;

**case** 4:

c=**new** Cadillac();

**break**;

**default**:

System.***out***.println("Wrong chassis number entered.");

}

This snippet shows that based on the user’s input we can directly refer the required class implementing the “Chassis” interface. This makes our code better.

**3.) Classes should be open for extension and closed for modification**

This principle states that the class driving most of the components should not be modified (unless some logic error is present) and its properties can be used anywhere inside the program.

Now, in our code, this principle has not been implemented. There are two classes of utmost importance namely “Factory” and “Car” which drive our entire program. The class “Car” is basically acting as an intermediate between the Factory and the specific parts. The inputs being sent inside the factory class will then be sent to car class via constructor for fetching the required information.

Code snippet for further explaination:

Car(Chassis c,Engine e,Tyre t,Suspension s,Air\_conditioning a,Paint p){

**this**.c=c;

**this**.e=e;

**this**.t=t;

**this**.s=s;

**this**.a=a;

**this**.p=p;

}

**int** getChassis()

{

**return** **this**.c.getBuildTime();

}

Now, if we want to add another component, let us say Music System to our car. For that we will need to modify the Car class to add the getMusic() method as well as modify the constructor. So here our class is not closed to modification.

**4.) Strive for loose coupling between objects that interact**

Loose coupling is a concept that the entities are mostly independent from each other. It is beneficial as it reduces the number of changes the programmer needs to make when certain modification is done in one portion of the code.

Code snippet for explanation:

**public** **interface** Engine

{

**public** **int** getBuildTime();

**public** **int** getEngineNumber();

}

**public** **class** V8 **implements** Engine{

**int** engineNumber = 1;

**int** buildTime = 50;

**public** **int** getBuildTime()

{

**return** buildTime;

}

**public** **int** getEngineNumber()

{

**return** engineNumber;

}

}

**public** **class** Four\_Cylinder **implements** Engine{

**int** engineNumber = 4;

**int** buildTime = 30;

**public** **int** getBuildTime()

{

**return** buildTime;

}

**public** **int** getEngineNumber()

{

**return** engineNumber;

}

}

**Code from Factory class:**

System.***out***.println("Enter type of Engine (1-5) : ");

engine = sc.nextInt();

**switch**(engine)

{

**case** 1:

e=**new** V8();

**break**;

**case** 2:

e=**new** V10();

**break**;

**case** 3:

e=**new** Two\_Cylinder();

**break**;

**case** 4:

e=**new** Four\_Cylinder();

**break**;

**case** 5:

e=**new** Internal\_Combustion();

**break**;

**default**:

System.***out***.println("Wrong engine number entered.");

}

Here, we have an interface Engine with two classes implementing it. Now, in factory class when the user enters the choice of engine, the respective class object gets created. Here the V8 and Four\_Cylinder classes are loosely coupled. Making changes in interface Engine will only lead to changes in both V8 and Four\_Cylinder classes. (This is applicable to all the classes for specific parts made in the project).

1. **Critical Analysis of Design Patterns**

1.) Strategy Design Pattern

In strategy design pattern, there is a Strategy present (which is an interface) and this strategy is used by the context classes to implement the variations in the algorithm that are required. Now, the “Concrete Strategy” classes implement a proper-defined version. The client supplies a concrete strategy object to context class. The context class then calls the appropriate methods of the strategy object.

In our code, we have implemented the strategy design pattern. The strategy interfaces are the Basic part interfaces which we have made (Engine, Chassis, Suspension, and so on). The concrete strategies are the specific part classes which we have made (V8, V10 for Engine; Red, White for Paint and so on). Now the context class is the “Factory” class which is being used in our program. The Factory class is programming to the interfaces to call the respective concrete strategies classes. We are providing code snippets of the same below with the description so that more clarity is present.

Code snippets:

**Code in Factory class:**

**switch**(tyre)

{

**case** 1:

t=**new** Antiskid();

**break**;

**case** 2:

t=**new** Fast\_Brake();

**break**;

**case** 3:

t=**new** Mountain();

**break**;

**default**:

System.***out***.println("Wrong tyre number entered.");

}

This code shows that the context class (Factory) calls the concrete strategies( Antiskid, Fast\_Brake, Mountain) for fulfilling the requirements.

**public** **interface** Engine

{

**public** **int** getBuildTime();

**public** **int** getEngineNumber();

}

This code shows the Strategy interface (Engine) present.

**public** **class** V8 **implements** Engine{

**int** engineNumber = 1;

**int** buildTime = 50;

**public** **int** getBuildTime()

{

**return** buildTime;

}

**public** **int** getEngineNumber()

{

**return** engineNumber;

}

}

This code shows the Concrete Strategy (V8) of the Engine interface.

Thus, we feel positively that our project is implementing the strategy design pattern.