GERMAN INTERNATIONAL UNIVERSITY
الجامعة الألمانية الدولية

Software Design Patterns.

2022

A research about different approaches in design patterns

Research by:
Omar ElBarbary 2001205(Iterator)
Mohamed Bassem 2003731(Singleton)
Momen Yasser 2001627(Builder)
Youssef Khaled 2000925(State)

Sbace_Bots

[30/12/2022]

Abstract

Design Patterns can be treated as blueprints to use when creating a certain software in mind,but they are not taken as is but tailored around the software you want to implement. The following research covers 4 design patterns; 2 behavioral and 2 Creational. Depending on your date you can use one of them or even all of them at once. Patterns covered are: Builder: a Creational pattern that helps you construct complex objects. Singleton: a Creational pattern that aims to create a global instance used by all clients. Iterator: behavioral pattern used to unify iterating through different datasets and Finally State: a behavioral pattern that allows objects to change behaviors given new internal changes.

1.Introduction

Design patterns are reusable Solutions that can be used to help build a software Swiftly and easily. There are different ways to classify a design pattern given its use and the intention, these patterns can be classified into:

- Creational patterns: to create objects increasing flexibility and reuse of code.
- Structural patterns: to Assemble objects and classes into larger structs while maintaining the flexibility and efficiency of structs
- Behavioral patterns: to communicate and assign responsibilities between objects.

Before choosing a design pattern for your software you need to understand it well as you cannot just choose a pattern and put it to your program you need as you can do with ready-to-use functions or libraries You can follow the pattern detail and implement a solution that suits ideas.

And thus the aim of this research is to do an in depth analysis in some of the patterns namely iterator, builder, state and singleton then compare the results to find which patterns to get a better understanding of each them and compare them at the end to find key differences that help on deciding which pattern to choose for each software.

2.Builder Pattern

Explanation

Builder pattern is a creational design pattern that gives you the ability to construct complicated objects step by step. The pattern allows you to produce different types and representations of an object using the same code.

The problem with complicated objects is that they require step by step initialization of many fields and objects. Such initialization is usually done or implemented using huge constructors with many parameters.

Real life examples

For example, let's say we want to create a house. For building a house, you need to build a roof and four walls and some doors and windows. But what if you want a bigger, fancier house, with a swimming pool and a backyard and other good stuff like a heating system or security system? The simplest way to achieve that is to extend the base class which is ‘House’ in this example and create many subclasses to cover all possible combinations. But at the end of the day you will have a big number of subclasses. Any new feature or parameter like a garden for example will make the hierarchy bigger. There is a different approach that does not depend on subclasses. You can make a big constructor in the base class with every single parameter that shapes the house object. There is no need for subclasses with this approach however it creates a different problem. In many cases the vast majority of the parameters will not be used, which makes the calls of the constructor very ugly. For example only a few houses have swimming pools, so nine out of ten times the parameters related to swimming pools will be futile.

It is suggested by the builder pattern that the construction code of an object should be extracted out of its class and be moved to different objects called builders. Object construction is organized by the builder pattern into steps like buildDoor and buildWalls for example. For an object to be created a series of steps is executed on a builder object. The important thing is that there is no need to do all of the steps. You can do the necessary steps only for making a specific variation of an object. When you need to build different representations of the product, you might need to implement some of the construction phases differently. For instance, a cabin's walls can be made of wood, while a castle's walls must be made of stone. In this situation, you can create a variety of builder classes that implement the same set of building processes in various ways. Then, to create various items, you can use these builders in the construction process (i.e., a structured series of calls to the building stages). For instance, Imagine a builder who uses only wood and glass, another who only uses stone and iron, and a third who only uses gold and diamonds. You can acquire a typical house from the first builder, a tiny castle from the second, and a palace from the third by calling the same series of instructions. This would only function, though, if the client code that initiates the building process can communicate with builders over a standard interface.

You may go even farther and extract the builder steps calls that you use to make a product into a different and separate class called ‘Director’. The Order in which the building steps are to be executed is defined by the director class, while the implementation for the steps is provided by the builder. It is not necessary to have a director class in your program. The building steps can always be called directly from the client code in a particular order. However, you may put different construction routines in the director class so that you can reuse them throughout your program. Additionally, the director class totally hides the product construction details from the client code. The client merely needs to pair up a builder with a director, have the director start construction, and have the builder deliver the finished product.

UML Diagram

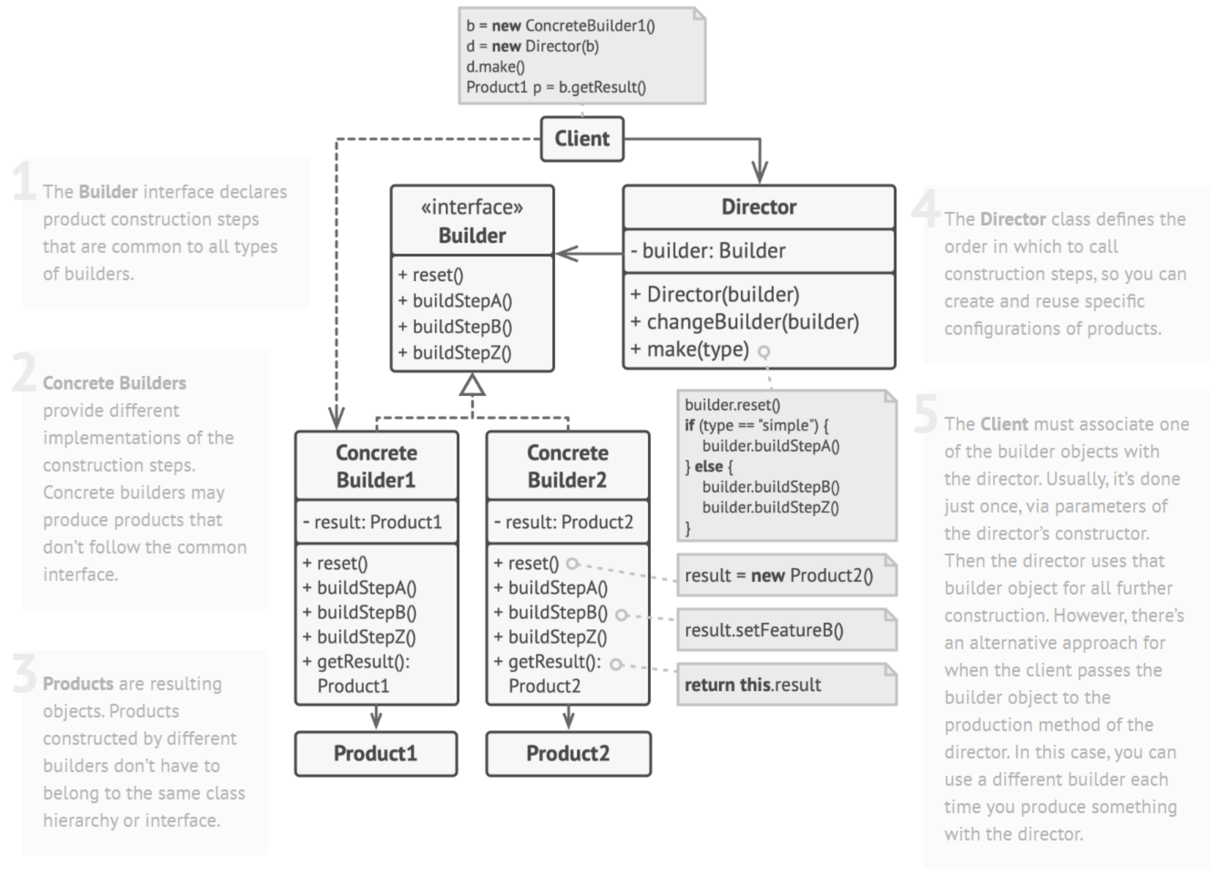
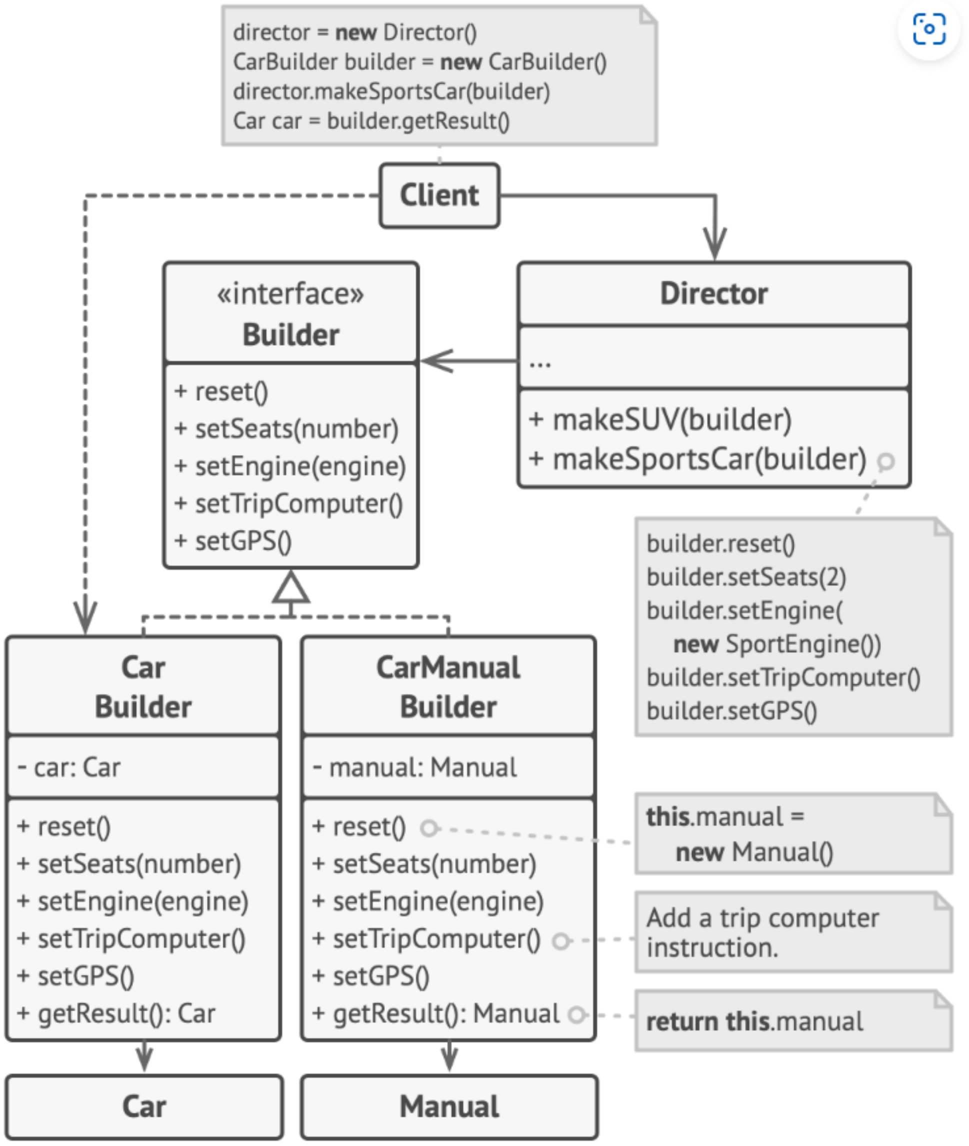

b = new ConcreteBuilder1()
d = new Director(b)
d.make()
Product1 p = b.getResult()
Client
1 The Builder interface declares product construction steps that are common to all types of builders.
«interface» Builder
+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()
Director
- builder: Builder
+ Director(builder)
+ changeBuilder(builder)
+ make(type)
4 The Director class defines the order in which to call construction steps, so you can create and reuse specific configurations of products.
2 Concrete Builders provide different implementations of the construction steps. Concrete builders may produce products that don't follow the common interface.
Concrete Builder1
- result: Product1
+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()
+ getResult(): Product1
Concrete Builder2
- result: Product2
+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()
+ getResult(): Product2
builder.reset()
if (type == "simple") {
builder.buildStepA()
} else {
builder.buildStepB()
builder.buildStepZ()
}
result = new Product2()
result.setFeatureB()
return this.result
5 The Client must associate one of the builder objects with the director. Usually, it's done just once, via parameters of the director's constructor. Then the director uses that builder object for all further construction. However, there's an alternative approach for when the client passes the builder object to the production method of the director. In this case, you can use a different builder each time you produce something with the director.
3 Products are resulting objects. Products constructed by different builders don't have to belong to the same class hierarchy or interface.
Product1
Product2

Figure 1

Figure 1 above is an UML class diagram example explaining the builder pattern.



```
director = new Director()
CarBuilder builder = new CarBuilder()
director.makeSportsCar(builder)
Car car = builder.getResult()
```

```
builder.reset()
builder.setSeats(2)
builder.setEngine(
   new SportEngine())
builder.setTripComputer()
builder.setGPS()
```

```
this.manual =
   new Manual()
```

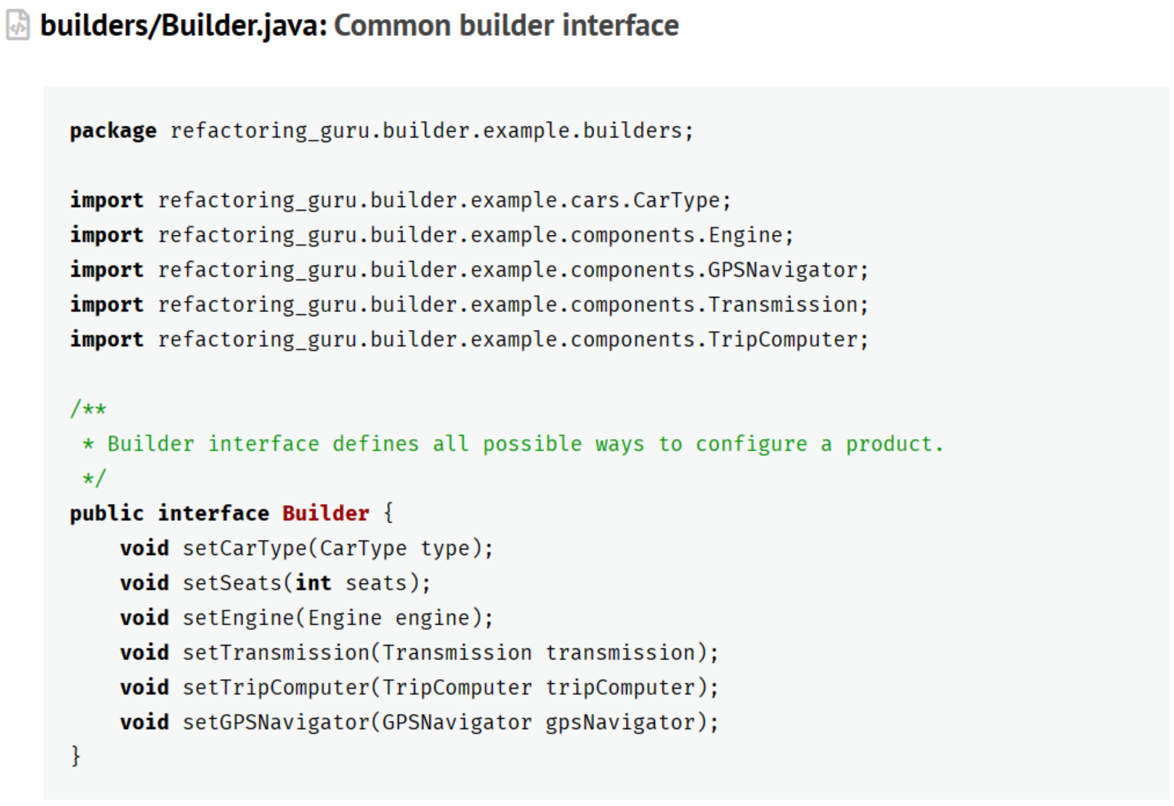
```
return this.manual
```

The example of step-by-step construction of cars and the user guides that fit those car models.

Figure 2

Figure 2 above is a real life example of an UML class diagram for car construction using the builder pattern.

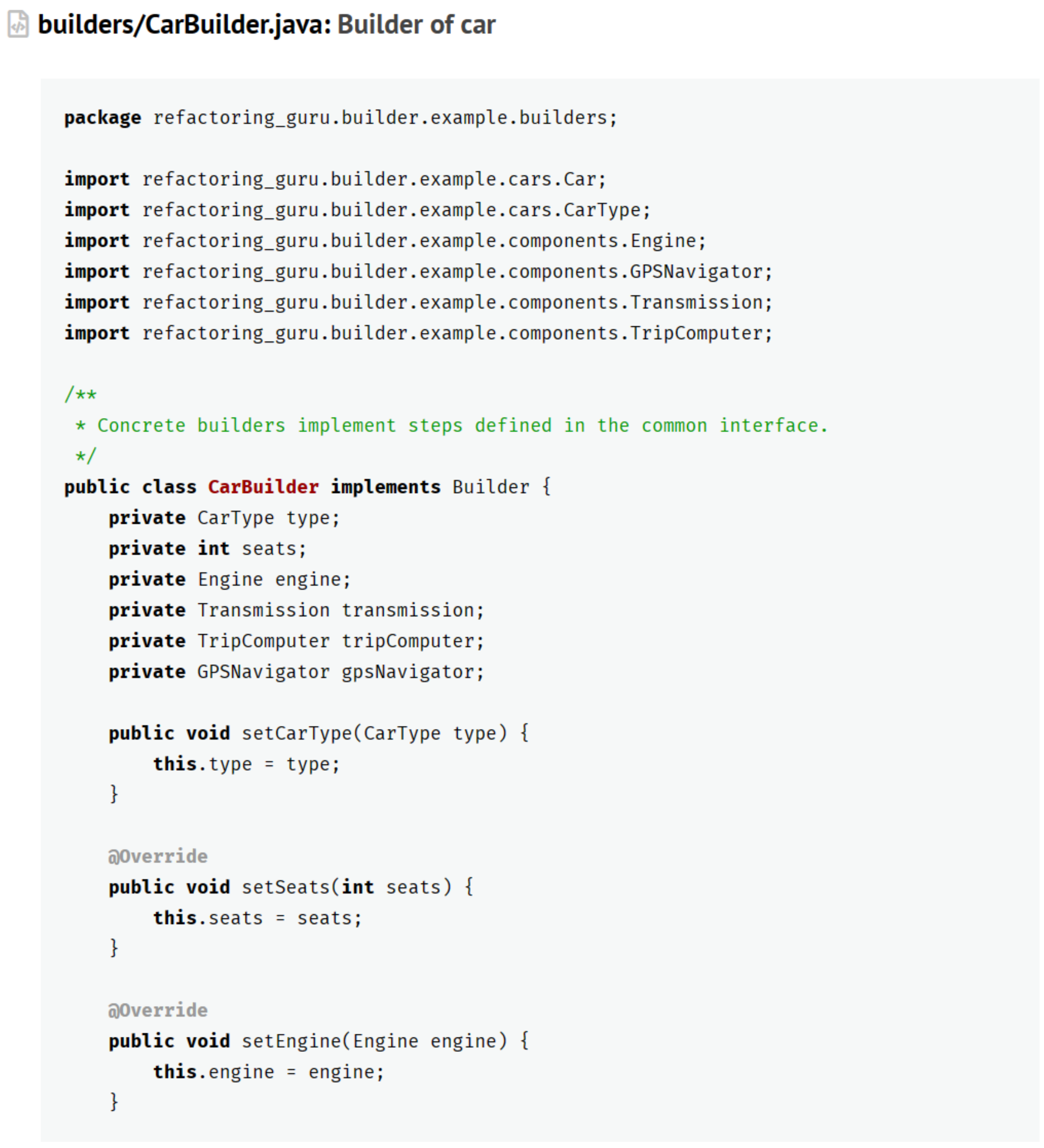
Code example



builders/Builder.java: Common builder interface

```java
package refactoring_guru.builder.example.builders;

import refactoring_guru.builder.example.cars.CarType;
import refactoring_guru.builder.example.components.Engine;
import refactoring_guru.builder.example.components.GPSNavigator;
import refactoring_guru.builder.example.components.Transmission;
import refactoring_guru.builder.example.components.TripComputer;

/**
 * Builder interface defines all possible ways to configure a product.
 */
public interface Builder {
    void setCarType(CarType type);
    void setSeats(int seats);
    void setEngine(Engine engine);
    void setTransmission(Transmission transmission);
    void setTripComputer(TripComputer tripComputer);
    void setGPSNavigator(GPSNavigator gpsNavigator);
}
```

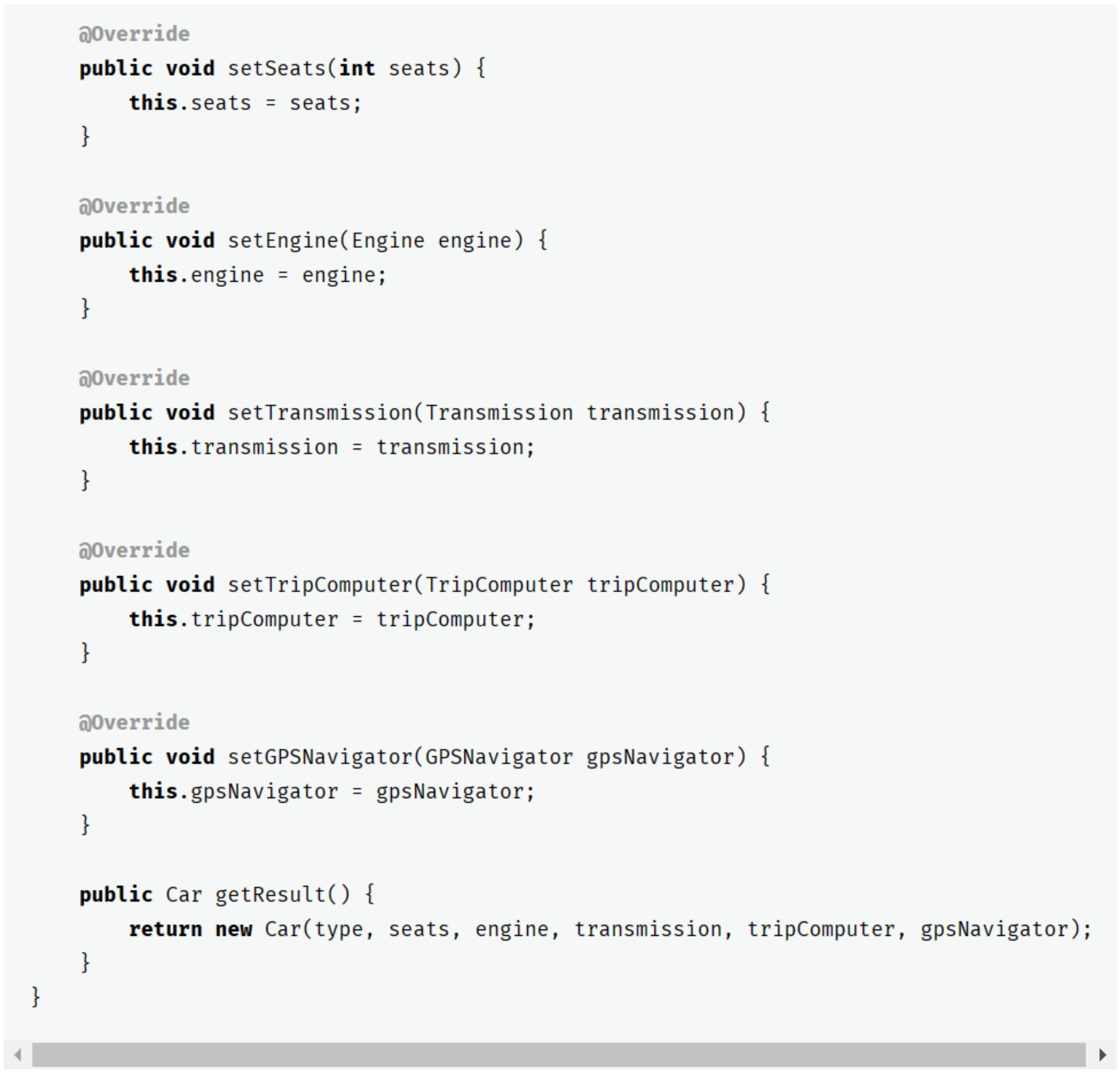
Figure 3



builders/CarBuilder.java: Builder of car

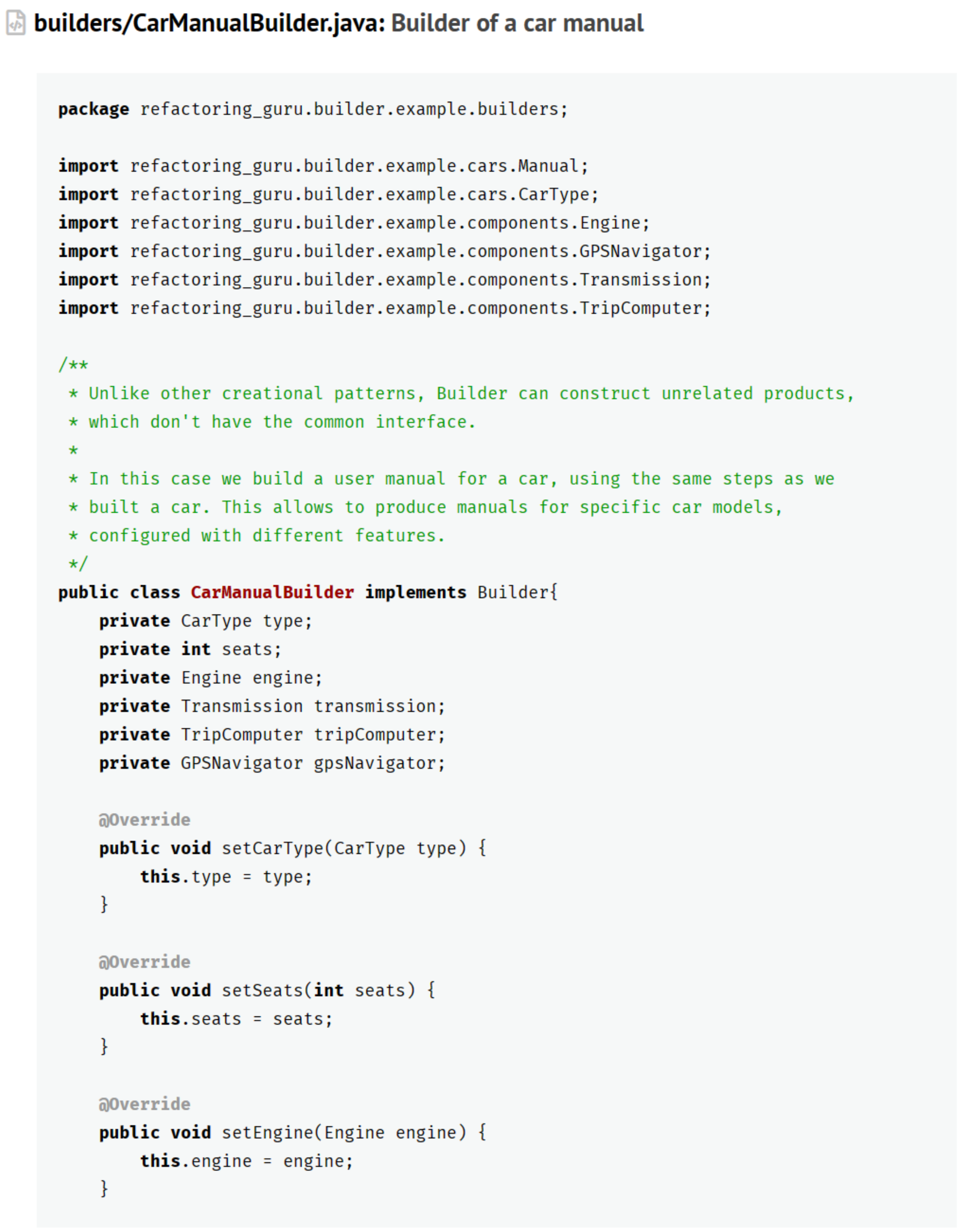
```java
package refactoring_guru.builder.example.builders;

import refactoring_guru.builder.example.cars.Car;
import refactoring_guru.builder.example.cars.CarType;
import refactoring_guru.builder.example.components.Engine;
import refactoring_guru.builder.example.components.GPSNavigator;
import refactoring_guru.builder.example.components.Transmission;
import refactoring_guru.builder.example.components.TripComputer;

/**
 * Concrete builders implement steps defined in the common interface.
 */
public class CarBuilder implements Builder {
    private CarType type;
    private int seats;
    private Engine engine;
    private Transmission transmission;
    private TripComputer tripComputer;
    private GPSNavigator gpsNavigator;

    public void setCarType(CarType type) {
        this.type = type;
    }

    @Override
    public void setSeats(int seats) {
        this.seats = seats;
    }

    @Override
    public void setEngine(Engine engine) {
        this.engine = engine;
    }
```

Figure 4



```java
    @Override
    public void setSeats(int seats) {
        this.seats = seats;
    }

    @Override
    public void setEngine(Engine engine) {
        this.engine = engine;
    }

    @Override
    public void setTransmission(Transmission transmission) {
        this.transmission = transmission;
    }

    @Override
    public void setTripComputer(TripComputer tripComputer) {
        this.tripComputer = tripComputer;
    }

    @Override
    public void setGPSNavigator(GPSNavigator gpsNavigator) {
        this.gpsNavigator = gpsNavigator;
    }

    public Car getResult() {
        return new Car(type, seats, engine, transmission, tripComputer, gpsNavigator);
    }
}
```

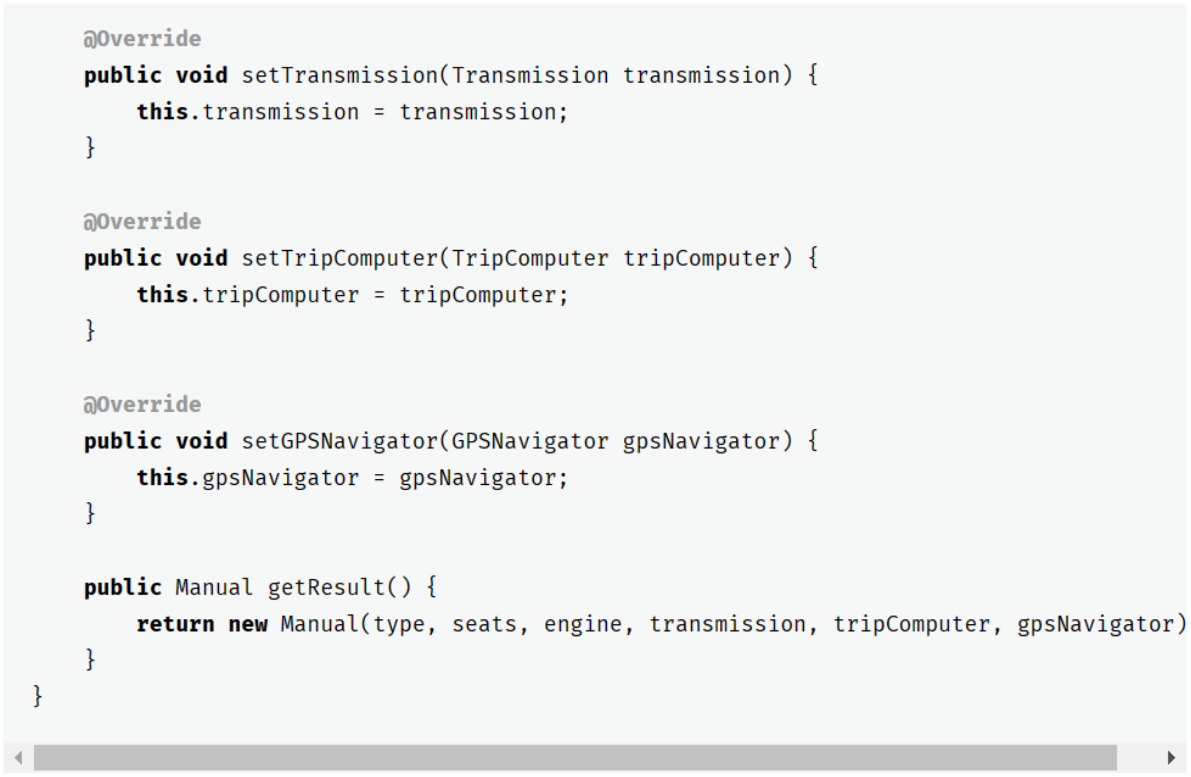
Figure 5



builders/CarManualBuilder.java: Builder of a car manual

```java
package refactoring_guru.builder.example.builders;

import refactoring_guru.builder.example.cars.Manual;
import refactoring_guru.builder.example.cars.CarType;
import refactoring_guru.builder.example.components.Engine;
import refactoring_guru.builder.example.components.GPSNavigator;
import refactoring_guru.builder.example.components.Transmission;
import refactoring_guru.builder.example.components.TripComputer;

/**
 * Unlike other creational patterns, Builder can construct unrelated products,
 * which don't have the common interface.
 *
 * In this case we build a user manual for a car, using the same steps as we
 * built a car. This allows to produce manuals for specific car models,
 * configured with different features.
 */
public class CarManualBuilder implements Builder{
    private CarType type;
    private int seats;
    private Engine engine;
    private Transmission transmission;
    private TripComputer tripComputer;
    private GPSNavigator gpsNavigator;

    @Override
    public void setCarType(CarType type) {
        this.type = type;
    }

    @Override
    public void setSeats(int seats) {
        this.seats = seats;
    }

    @Override
    public void setEngine(Engine engine) {
        this.engine = engine;
    }
```

Figure 6



```
    @Override
    public void setTransmission(Transmission transmission) {
        this.transmission = transmission;
    }

    @Override
    public void setTripComputer(TripComputer tripComputer) {
        this.tripComputer = tripComputer;
    }

    @Override
    public void setGPSNavigator(GPSNavigator gpsNavigator) {
        this.gpsNavigator = gpsNavigator;
    }

    public Manual getResult() {
        return new Manual(type, seats, engine, transmission, tripComputer, gpsNavigator)
    }
}
```

Figure 7

cars/Car.java: Car product

```java
package refactoring_guru.builder.example.cars;

import refactoring_guru.builder.example.components.Engine;
import refactoring_guru.builder.example.components.GPSNavigator;
import refactoring_guru.builder.example.components.Transmission;
import refactoring_guru.builder.example.components.TripComputer;

/**
 * Car is a product class.
 */
public class Car {
    private final CarType carType;
    private final int seats;
    private final Engine engine;
    private final Transmission transmission;
    private final TripComputer tripComputer;
    private final GPSNavigator gpsNavigator;
    private double fuel = 0;

    public Car(CarType carType, int seats, Engine engine, Transmission transmission,
               TripComputer tripComputer, GPSNavigator gpsNavigator) {
        this.carType = carType;
        this.seats = seats;
        this.engine = engine;
        this.transmission = transmission;
        this.tripComputer = tripComputer;
        if (this.tripComputer != null) {
            this.tripComputer.setCar(this);
        }
        this.gpsNavigator = gpsNavigator;
    }

    public CarType getCarType() {
        return carType;
    }

    public double getFuel() {
        return fuel;
    }
```

Figure 8

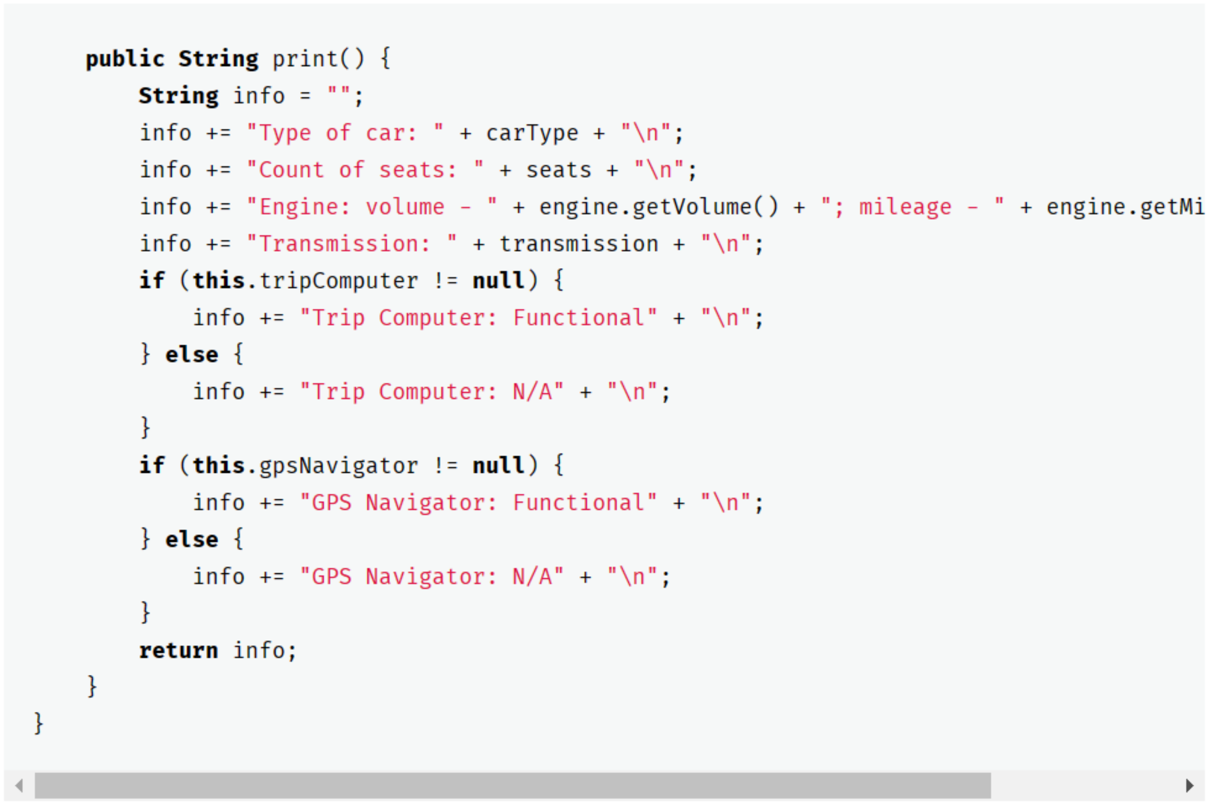
```java
    public void setFuel(double fuel) {
        this.fuel = fuel;
    }

    public int getSeats() {
        return seats;
    }

    public Engine getEngine() {
        return engine;
    }

    public Transmission getTransmission() {
        return transmission;
    }

    public TripComputer getTripComputer() {
        return tripComputer;
    }

    public GPSNavigator getGpsNavigator() {
        return gpsNavigator;
    }
}
```

Figure 9

cars/Manual.java: Manual product

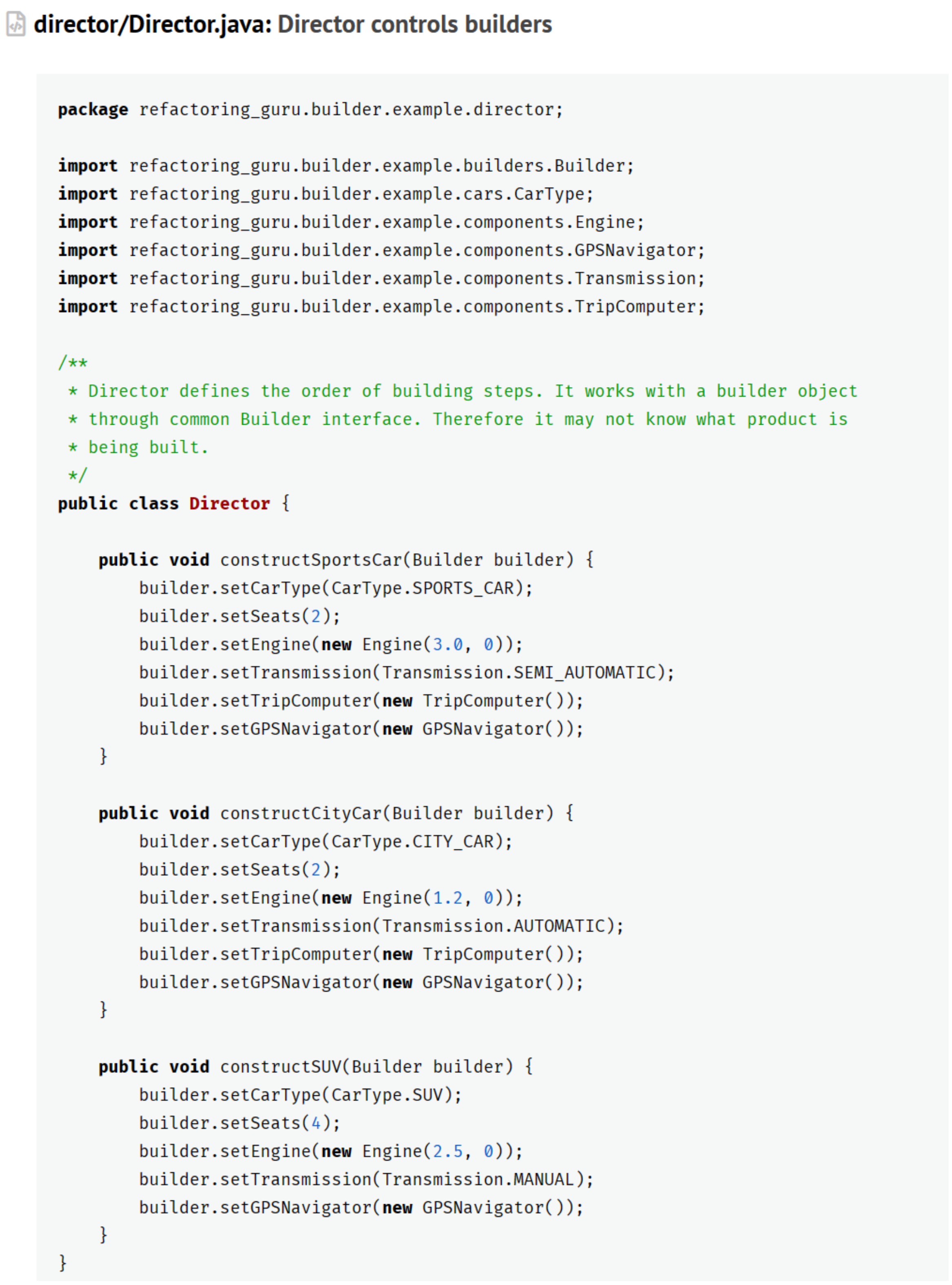
```
package refactoring_guru.builder.example.cars;

import refactoring_guru.builder.example.components.Engine;
import refactoring_guru.builder.example.components.GPSNavigator;
import refactoring_guru.builder.example.components.Transmission;
import refactoring_guru.builder.example.components.TripComputer;

/**
 * Car manual is another product. Note that it does not have the same ancestor
 * as a Car. They are not related.
 */
public class Manual {
    private final CarType carType;
    private final int seats;
    private final Engine engine;
    private final Transmission transmission;
    private final TripComputer tripComputer;
    private final GPSNavigator gpsNavigator;

    public Manual(CarType carType, int seats, Engine engine, Transmission transmission,
                  TripComputer tripComputer, GPSNavigator gpsNavigator) {
        this.carType = carType;
        this.seats = seats;
        this.engine = engine;
        this.transmission = transmission;
        this.tripComputer = tripComputer;
        this.gpsNavigator = gpsNavigator;
    }
```

Figure 10



```
    public String print() {
        String info = "";
        info += "Type of car: " + carType + "\n";
        info += "Count of seats: " + seats + "\n";
        info += "Engine: volume - " + engine.getVolume() + "; mileage - " + engine.getMi
        info += "Transmission: " + transmission + "\n";
        if (this.tripComputer != null) {
            info += "Trip Computer: Functional" + "\n";
        } else {
            info += "Trip Computer: N/A" + "\n";
        }
        if (this.gpsNavigator != null) {
            info += "GPS Navigator: Functional" + "\n";
        } else {
            info += "GPS Navigator: N/A" + "\n";
        }
        return info;
    }
}
```

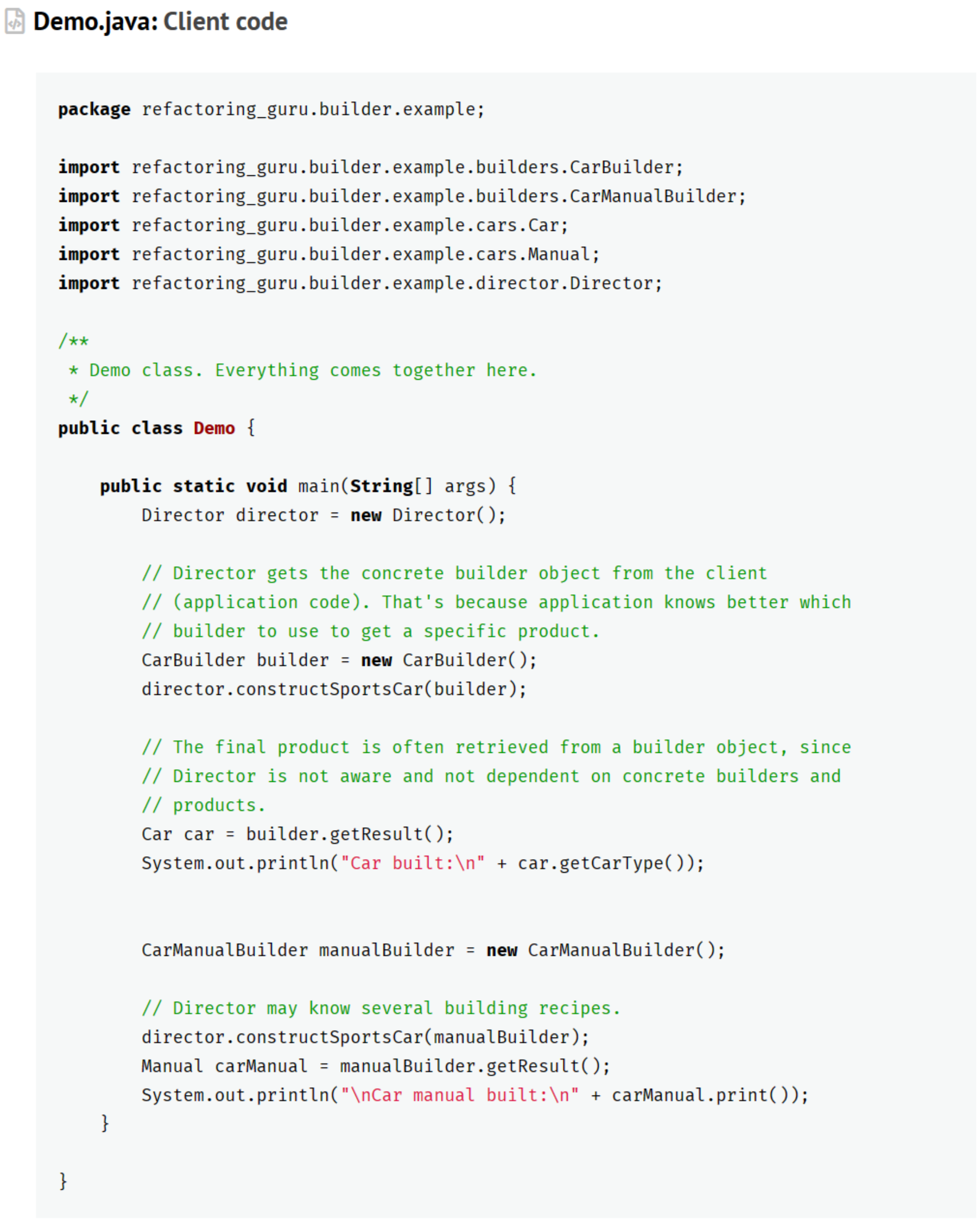
Figure 11



director/Director.java: Director controls builders

```java
package refactoring_guru.builder.example.director;

import refactoring_guru.builder.example.builders.Builder;
import refactoring_guru.builder.example.cars.CarType;
import refactoring_guru.builder.example.components.Engine;
import refactoring_guru.builder.example.components.GPSNavigator;
import refactoring_guru.builder.example.components.Transmission;
import refactoring_guru.builder.example.components.TripComputer;

/**
 * Director defines the order of building steps. It works with a builder object
 * through common Builder interface. Therefore it may not know what product is
 * being built.
 */
public class Director {

    public void constructSportsCar(Builder builder) {
        builder.setCarType(CarType.SPORTS_CAR);
        builder.setSeats(2);
        builder.setEngine(new Engine(3.0, 0));
        builder.setTransmission(Transmission.SEMI_AUTOMATIC);
        builder.setTripComputer(new TripComputer());
        builder.setGPSNavigator(new GPSNavigator());
    }

    public void constructCityCar(Builder builder) {
        builder.setCarType(CarType.CITY_CAR);
        builder.setSeats(2);
        builder.setEngine(new Engine(1.2, 0));
        builder.setTransmission(Transmission.AUTOMATIC);
        builder.setTripComputer(new TripComputer());
        builder.setGPSNavigator(new GPSNavigator());
    }

    public void constructSUV(Builder builder) {
        builder.setCarType(CarType.SUV);
        builder.setSeats(4);
        builder.setEngine(new Engine(2.5, 0));
        builder.setTransmission(Transmission.MANUAL);
        builder.setGPSNavigator(new GPSNavigator());
    }
}
```

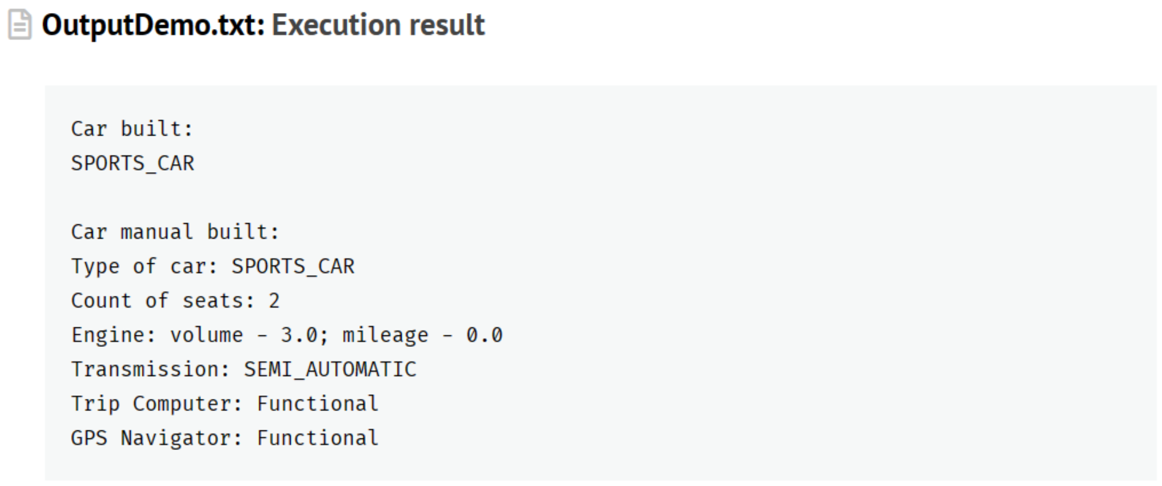
Figure 12



Demo.java: Client code

```java
package refactoring_guru.builder.example;

import refactoring_guru.builder.example.builders.CarBuilder;
import refactoring_guru.builder.example.builders.CarManualBuilder;
import refactoring_guru.builder.example.cars.Car;
import refactoring_guru.builder.example.cars.Manual;
import refactoring_guru.builder.example.director.Director;

/**
 * Demo class. Everything comes together here.
 */
public class Demo {

    public static void main(String[] args) {
        Director director = new Director();

        // Director gets the concrete builder object from the client
        // (application code). That's because application knows better which
        // builder to use to get a specific product.
        CarBuilder builder = new CarBuilder();
        director.constructSportsCar(builder);

        // The final product is often retrieved from a builder object, since
        // Director is not aware and not dependent on concrete builders and
        // products.
        Car car = builder.getResult();
        System.out.println("Car built:\n" + car.getCarType());


        CarManualBuilder manualBuilder = new CarManualBuilder();

        // Director may know several building recipes.
        director.constructSportsCar(manualBuilder);
        Manual carManual = manualBuilder.getResult();
        System.out.println("\nCar manual built:\n" + carManual.print());
    }

}
```

Figure 13



OutputDemo.txt: Execution result

```
Car built:
SPORTS_CAR

Car manual built:
Type of car: SPORTS_CAR
Count of seats: 2
Engine: volume - 3.0; mileage - 0.0
Transmission: SEMI_AUTOMATIC
Trip Computer: Functional
GPS Navigator: Functional
```

Figure 14

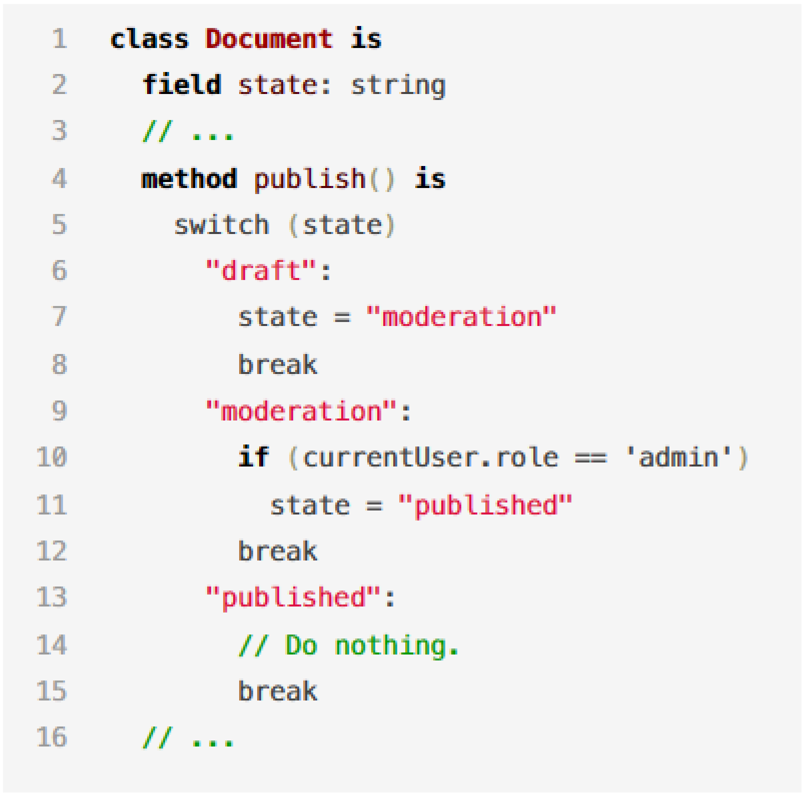
The figures from 3 to 14 above is a code example for car construction using the builder design pattern.

The rest of the code is on thethirdsh/car-builder-design-pattern (github.com).

3.State Design Pattern

Explanation

The State Pattern concept is nearly related to Finite-State-Machines. The main concept of Finite-State-Machines is a program that behaves differently depending on the state which the program is at. In each different state, the program will have to behave differently. State Machines are typically implemented with several conditional operators. This becomes a problem because if the program has several states, then the code will not be maintainable due to the several If-conditions and switch statements. Let's take the example of the Document class in figure 14. A document consists of 3 states: Draft, Moderation and Published. The document has a Publish method that will perform differently depending on the current state. We will have to write different switch statements for each state. Obviously this is not an efficient way to code.



```
class Document is
  field state: string
  // ...
  method publish() is
    switch (state)
      "draft":
        state = "moderation"
        break
      "moderation":
        if (currentUser.role == 'admin')
          state = "published"
        break
      "published":
        // Do nothing.
        break
  // ...
```

Figure 14

The State Design Pattern was introduced to solve this issue. The idea of State Design Pattern is that every state of the program should be implemented in separate classes. This pattern is divided into 3 main types of classes which are Context, State, and ConcreteState. The Context class provides the user with an interface to interact, and it contains references to concrete state objects that are used to set its current state. It interacts with the state object through the state interface. The State class is an interface for each concrete state and contains methods that are relevant to each concrete state. As for the concrete state class, it contains implementation of each method defined in the State interface. Every concrete state may contain reference to the context object to be able to access any information required. The Context class and Concrete classes have the ability to change the current state. There can be multiple concrete state classes depending on the number of states the program has. This is visually explained in figure 15.

UML Diagram

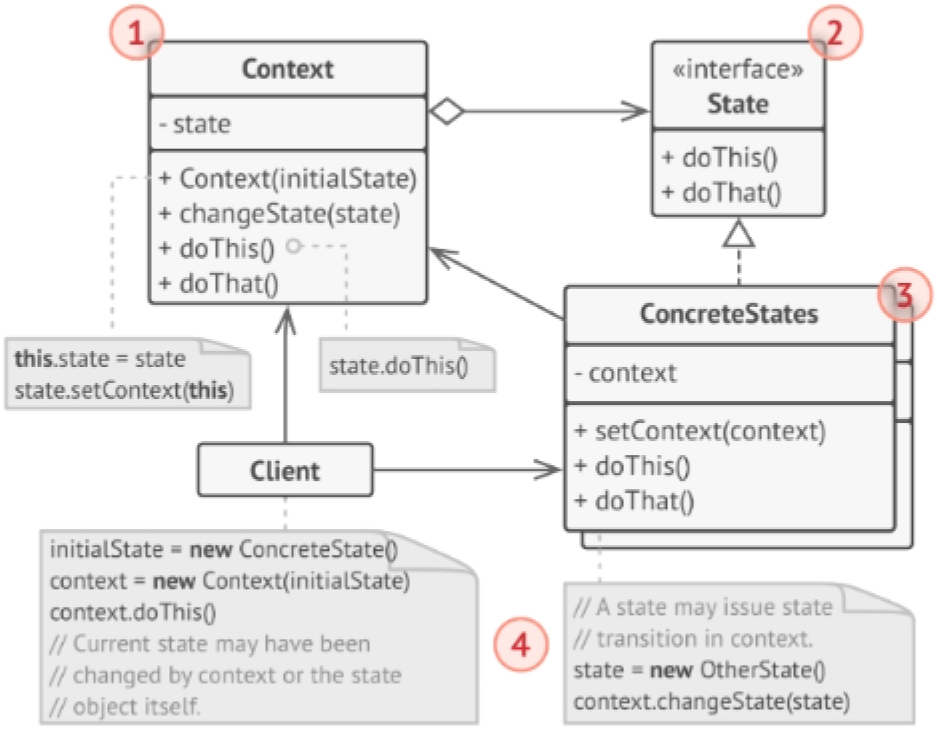


Figure 15

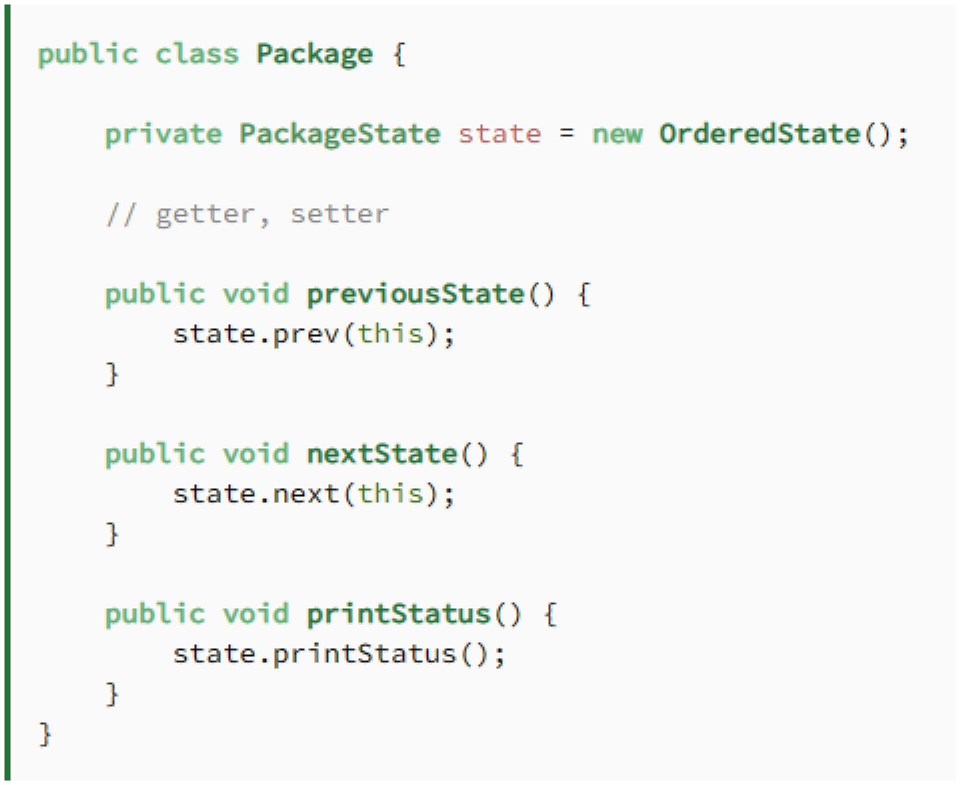
Real life example

Let's take for example a TV remote controller. The TV has two states: ON and OFF. We can change the TV's state by pressing the power button. The rest of the buttons will behave differently depending on the TV's state. When the TV's state is ON, you could change the volume, mute, change channels and switch it OFF. However if the state is OFF, you could only turn it back ON.

Another example is your mobile phones. The phone has two states: locked and unlocked. If the phone is unlocked, you can change the brightness, volume, make calls, etc. However when the phone is locked, pressing any buttons will change its state to unlocked.

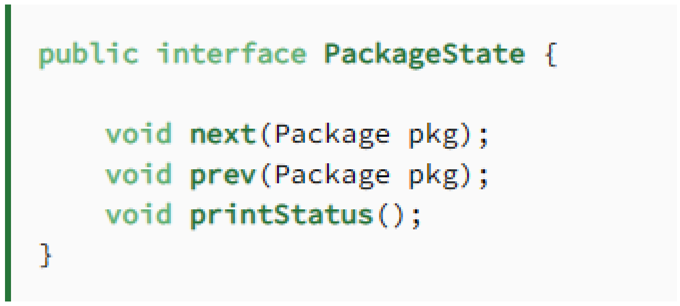
Code example

Let's see the example of a delivery package that has 3 states: Ordered, Delivered and Received.



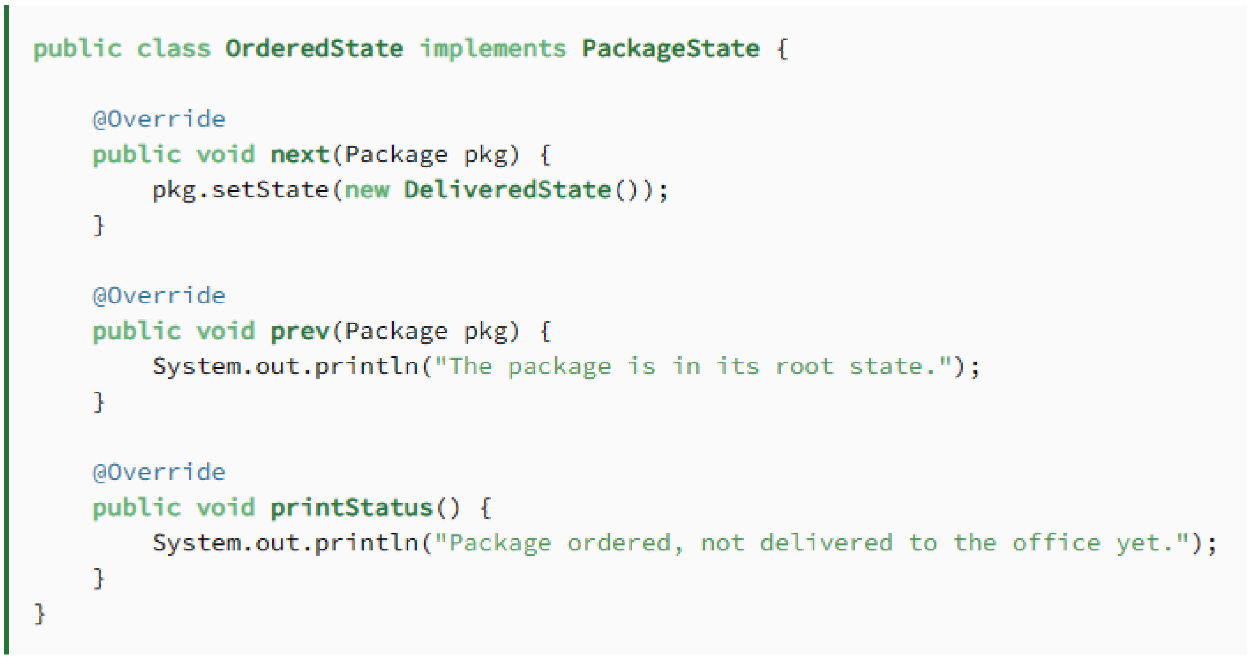
```java
public class Package {

    private PackageState state = new OrderedState();

    // getter, setter

    public void previousState() {
        state.prev(this);
    }

    public void nextState() {
        state.next(this);
    }

    public void printStatus() {
        state.printStatus();
    }
}
```

Figure a: Context Class



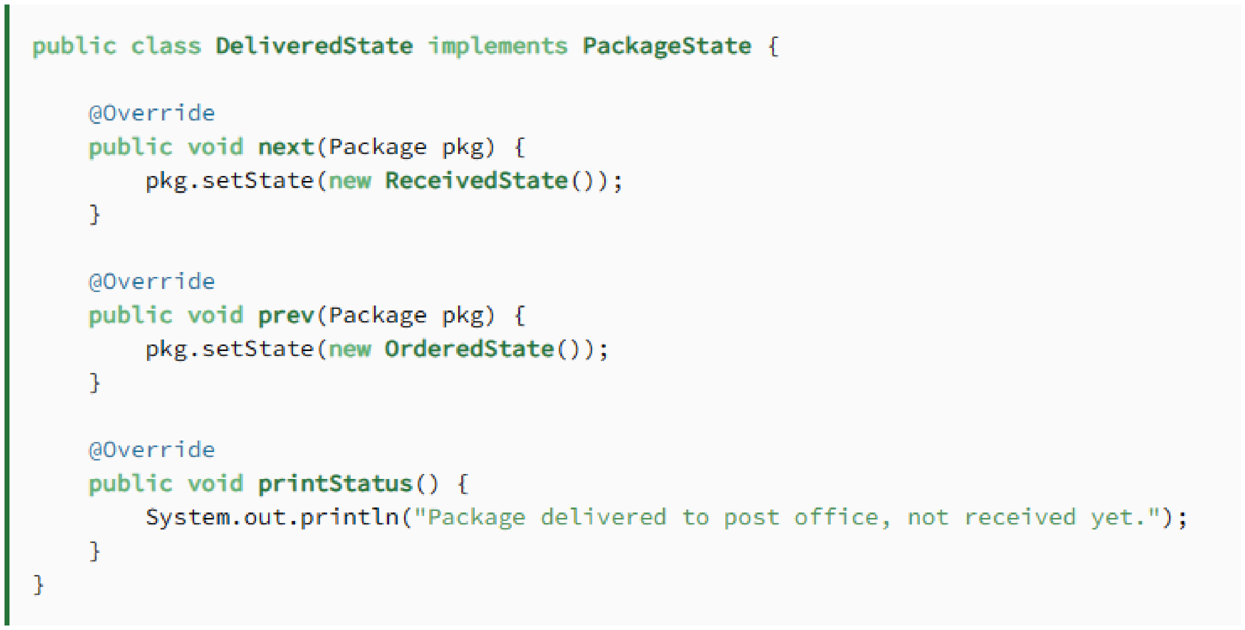
```java
public interface PackageState {

    void next(Package pkg);
    void prev(Package pkg);
    void printStatus();
}
```

Figure b: PackageState Class (interface)



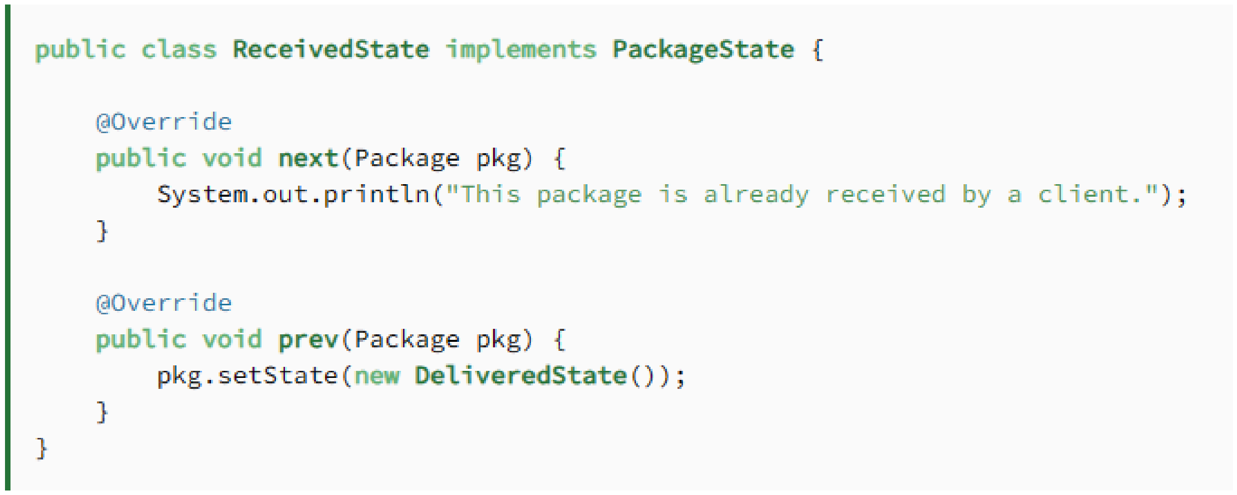
```java
public class OrderedState implements PackageState {

    @Override
    public void next(Package pkg) {
        pkg.setState(new DeliveredState());
    }

    @Override
    public void prev(Package pkg) {
        System.out.println("The package is in its root state.");
    }

    @Override
    public void printStatus() {
        System.out.println("Package ordered, not delivered to the office yet.");
    }
}
```

Figure c: OrderedState Class (Concrete state)



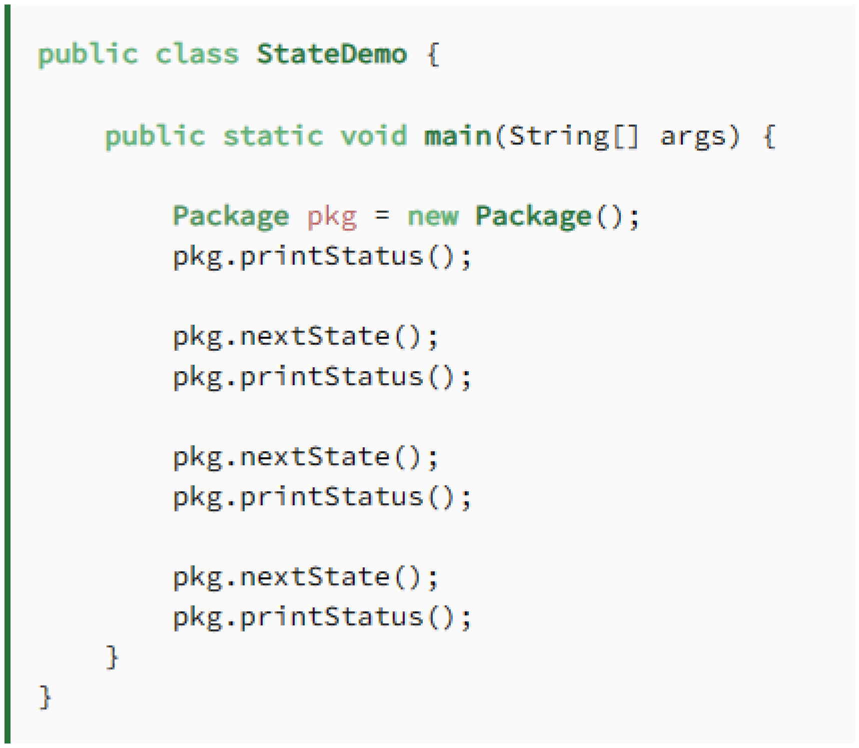
```java
public class DeliveredState implements PackageState {

    @Override
    public void next(Package pkg) {
        pkg.setState(new ReceivedState());
    }

    @Override
    public void prev(Package pkg) {
        pkg.setState(new OrderedState());
    }

    @Override
    public void printStatus() {
        System.out.println("Package delivered to post office, not received yet.");
    }
}
```

Figure d: DeliveredState Class (Concrete state)



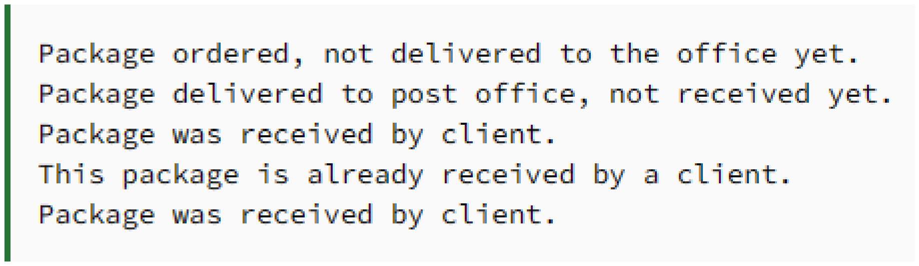
```java
public class ReceivedState implements PackageState {

    @Override
    public void next(Package pkg) {
        System.out.println("This package is already received by a client.");
    }

    @Override
    public void prev(Package pkg) {
        pkg.setState(new DeliveredState());
    }
}
```

Figure e: Received State Class (Concrete state)



```java
public class StateDemo {

    public static void main(String[] args) {

        Package pkg = new Package();
        pkg.printStatus();

        pkg.nextState();
        pkg.printStatus();

        pkg.nextState();
        pkg.printStatus();

        pkg.nextState();
        pkg.printStatus();
    }
}
```

Figure f: Main Class



```
Package ordered, not delivered to the office yet.
Package delivered to post office, not received yet.
Package was received by client.
This package is already received by a client.
Package was received by client.
```

Figure g: Output of Main class

As we can see from the above example, every concrete state class has reference to the Package context so they are able to switch states. The code is more organized and much more maintainable than without using State Design Patterns.

4.Iterator Pattern

Explanation

Data-structures and algorithms are an essential part in software design. Depending on the data, using the appropriate data structure for a specific situation could be more efficient and. Distinct data structures frequently have different functions. Your solution will be improved by selecting the appropriate data structure and combining it with the appropriate algorithm.

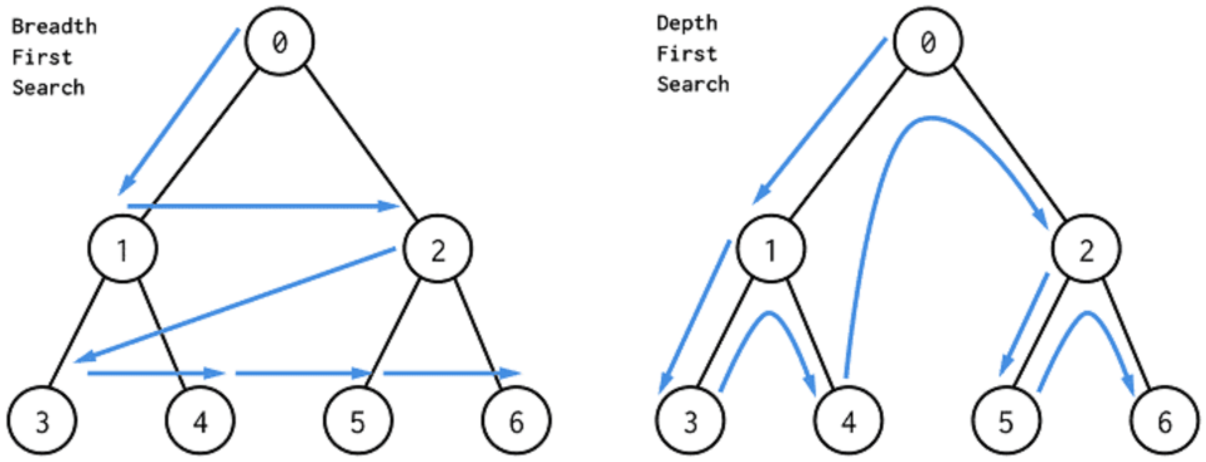


Figure 16: depth vs breadth iteration

Iterator pattern is a behavioral pattern that is meant to provide an object called iterator which iterates through a collection without exposing its underlying implication as the current position and how many elements are left till the end.

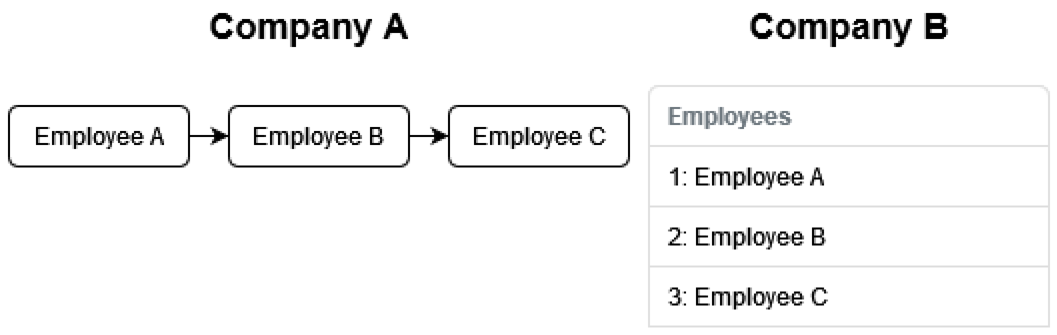
To further understand, imagine going on a trip around Cairo to visit monuments, travel office A has a specific package that goes as follows(Pyramids Grand Museum then Cairo tower) all while using a tour bus to show the street of Cairo meanwhile Travel office B (Grand Museum Pyramids then Cairo tower) and traveling using Car. In these cases the trip is considered a collection and Travel offices are considered iterators while the Cairo is considered the collection. Each office has their own price and conveniences. Similarly, breadth first and depth first can be made into an iterator instead of having tedious loops.

<u>Real life examples</u>

<u>Different data types</u>:
Consider the following scenario:

- 2 companies, Company A and Company B.
- Company A saves its employee records(employee's name, address, salary information. etc)in a linked list data structure.
- Company B stores its employee data in a hashmap.
- Then a fortune 500 company comes in and buys both companies.

Instead of having to change the structure of the data or creating a new one, the Iterator approach comes in handy in this scenario because you don't have to start from scratch with the code. In a scenario like this, you can create a shared interface to access the data for both businesses, allowing you to use those functions without having to rewrite the code.



Same collection different purposes:

Now suppose inside the new found mega corp the HR department wants to go through the employees of company A by name and the finance department wants to go through them by their Salary. The iterator approach is also beneficial here as a collection can have multiple iterators (see breadth first vs depth first in 4.1)

UML Diagram

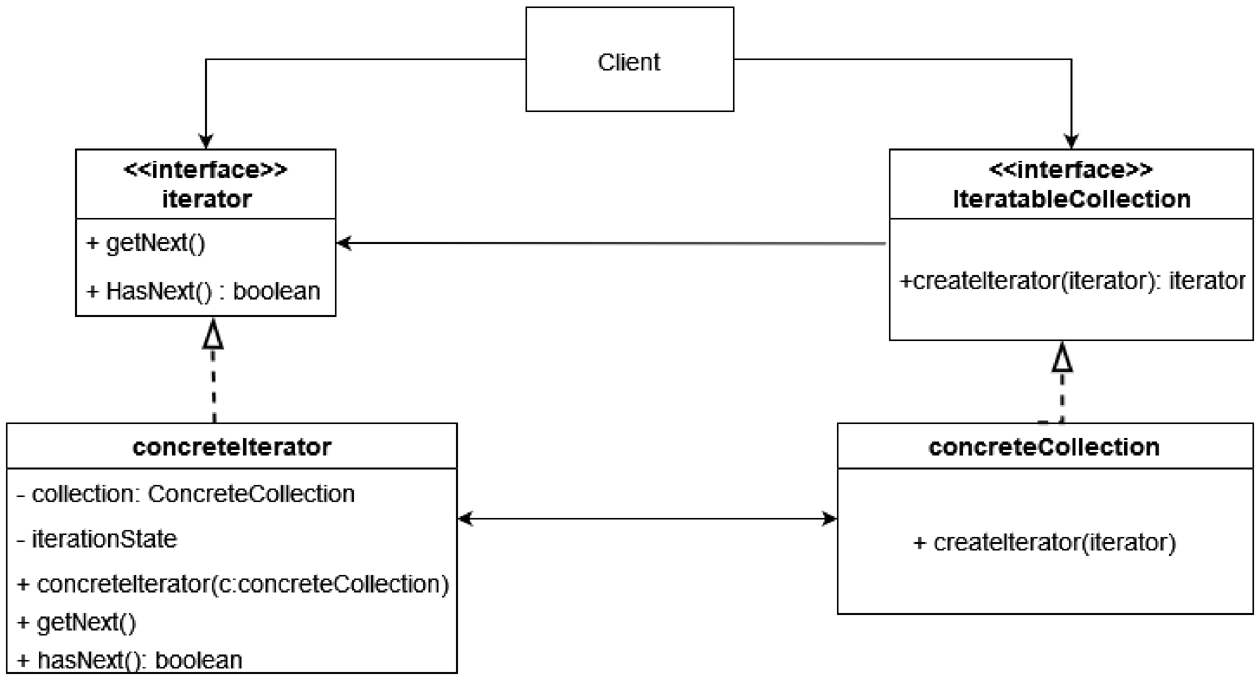
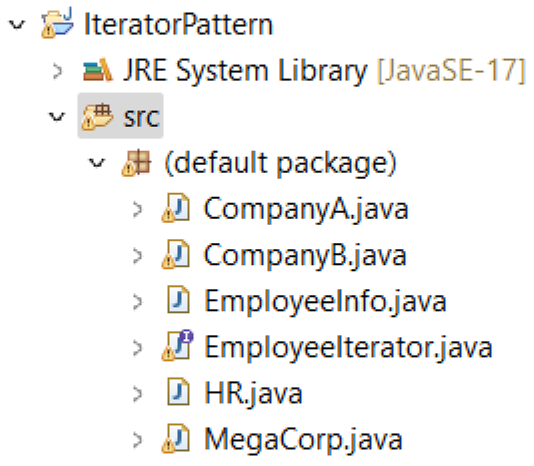


Figure 15 :UML for iteration pattern

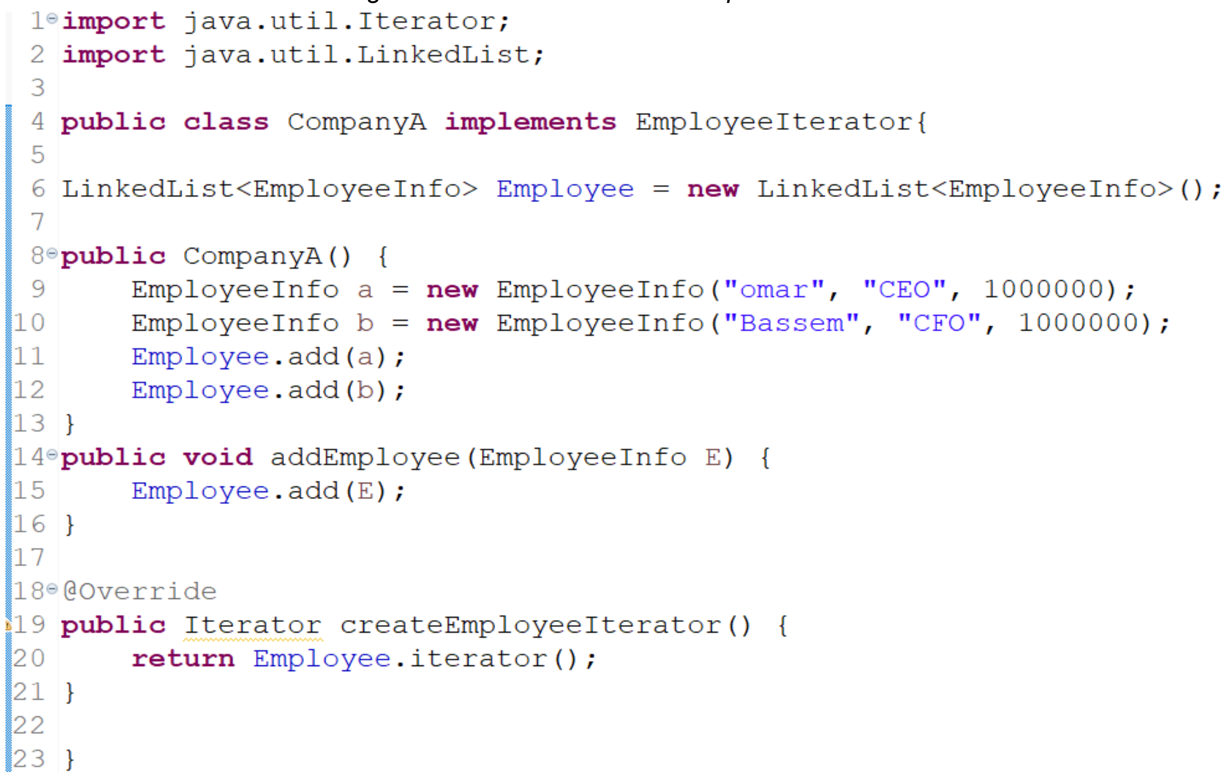
- Iterator interface: creates the operations required for traversing through the collection (getting the next element getting the value of current element ..etc)
- Concrete Iterators: implements iterator interface,provides specific algorithms for traversing a collection. The iterator object should track the traversal progress on its own. This enables multiple iterators to traverse the same collection independently of each other (see figure 14).
- The Collection interface declares one or multiple iterators to traverse the collection. for the concrete collections to have various kinds of iterators,the method returns an iterator interface
- Concrete Collections : implements collection interface return instances of a particular concrete iterator class to fit the client requests.
- Client: Do not create iterators instead retrieve them from collections this ensure that the iterators are compatible with collections in addition it encapsulates the underlying mechanisms

Code example



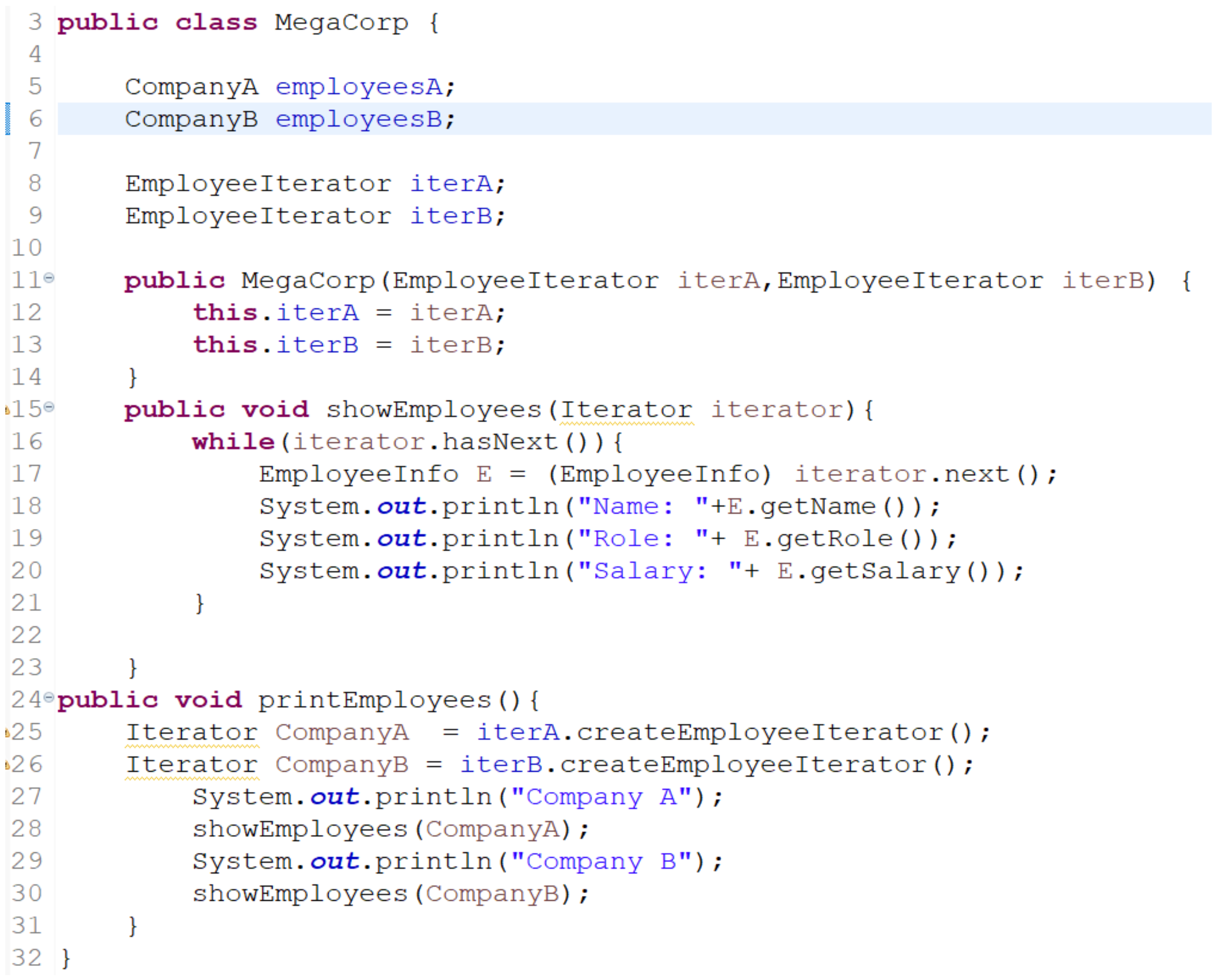
```
IteratorPattern
  JRE System Library [JavaSE-17]
  src
    (default package)
      CompanyA.java
      CompanyB.java
      EmployeeInfo.java
      EmployeeIterator.java
      HR.java
      MegaCorp.java
```

Figure x: Architecture of Iterator pattern



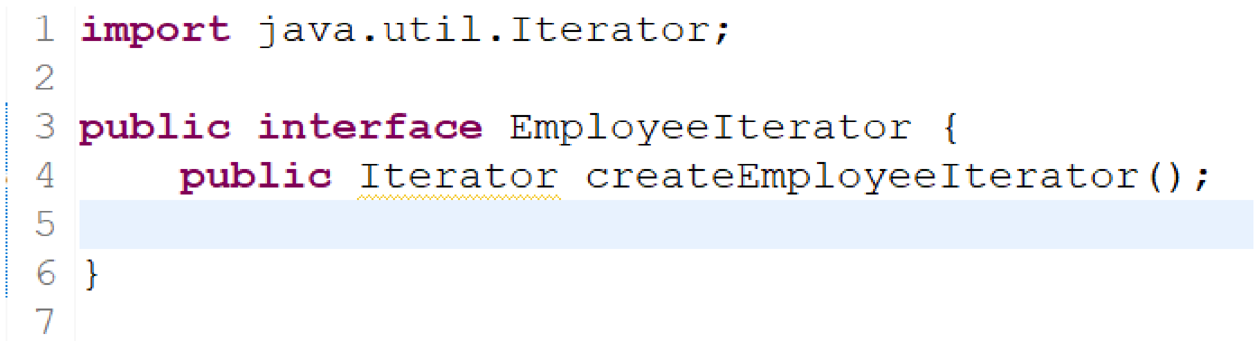
```java
import java.util.Iterator;
import java.util.LinkedList;

public class CompanyA implements EmployeeIterator{

LinkedList<EmployeeInfo> Employee = new LinkedList<EmployeeInfo>();

public CompanyA() {
     EmployeeInfo a = new EmployeeInfo("omar", "CEO", 1000000);
     EmployeeInfo b = new EmployeeInfo("Bassem", "CFO", 1000000);
     Employee.add(a);
     Employee.add(b);
}
public void addEmployee(EmployeeInfo E) {
     Employee.add(E);
}

@Override
public Iterator createEmployeeIterator() {
     return Employee.iterator();
}

}
```

Figure x: Iterator concrete class



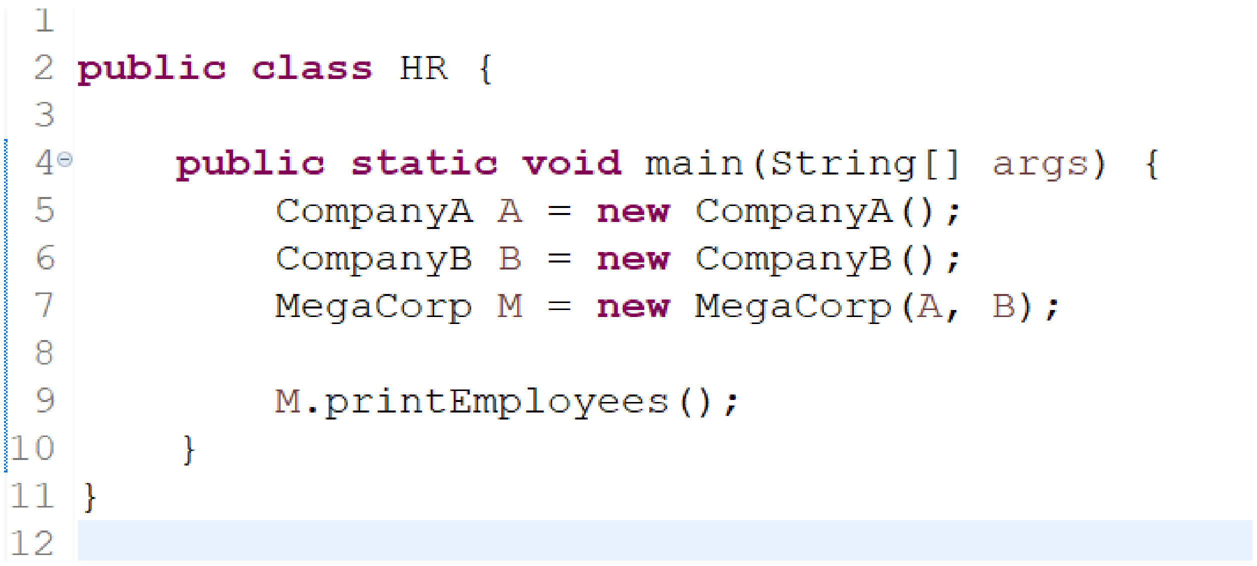
```java
public class MegaCorp {

    CompanyA employeesA;
    CompanyB employeesB;

    EmployeeIterator iterA;
    EmployeeIterator iterB;

    public MegaCorp(EmployeeIterator iterA,EmployeeIterator iterB) {
        this.iterA = iterA;
        this.iterB = iterB;
    }
    public void showEmployees(Iterator iterator){
        while(iterator.hasNext()){
            EmployeeInfo E = (EmployeeInfo) iterator.next();
            System.out.println("Name: "+E.getName());
            System.out.println("Role: "+ E.getRole());
            System.out.println("Salary: "+ E.getSalary());
        }

    }
public void printEmployees(){
    Iterator CompanyA  = iterA.createEmployeeIterator();
    Iterator CompanyB = iterB.createEmployeeIterator();
        System.out.println("Company A");
        showEmployees(CompanyA);
        System.out.println("Company B");
        showEmployees(CompanyB);
    }
}
```

Figure x: collection class



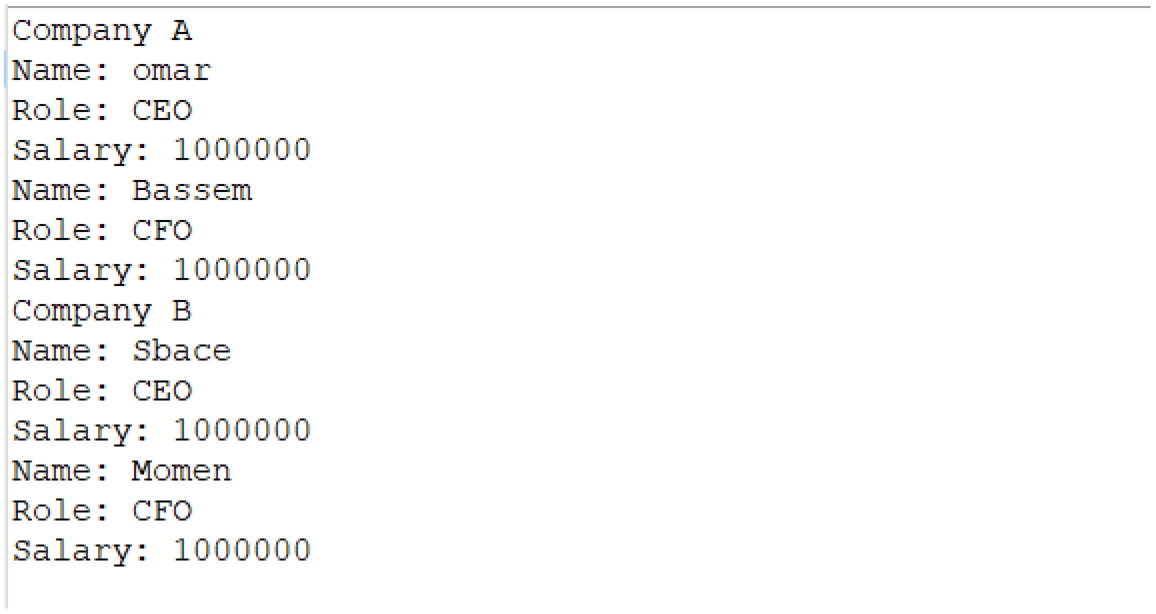
```java
import java.util.Iterator;

public interface EmployeeIterator {
    public Iterator createEmployeeIterator();

}

```

Figure x: Iterator Interface



```java

public class HR {

    public static void main(String[] args) {
        CompanyA A = new CompanyA();
        CompanyB B = new CompanyB();
        MegaCorp M = new MegaCorp(A, B);

        M.printEmployees();
    }
}

```

Figure x: Client



```
Company A
Name: omar
Role: CEO
Salary: 1000000
Name: Bassem
Role: CFO
Salary: 1000000
Company B
Name: Sbace
Role: CEO
Salary: 1000000
Name: Momen
Role: CFO
Salary: 1000000
```

Figure x: Output

Going back to our companies example, each company(concrete collection) has its list of employees(concrete iterators) where each company implements EmployeeIterator interface(iterator interface) forcing it to to make and iterator for their datasets HR or Finance(client) to get what they need by calling MegaCorp (Collection class). You check the code here.

5. Singleton Pattern

In software engineering, the Singleton design pattern is used to ensure that a class only has one instance This entails limiting class instantiation in a system to a single object (or perhaps just a small number of objects). When the system runs more quickly or uses less memory with fewer things than with many similar objects, this pattern may be employed. Singleton design patterns are helpful for facilitating the production of high-quality designs, but they are also helpful for reverse engineering, or the analysis of already-existing systems, such as rebuilding documentation from source code. Design patterns must be identified and correctly detected in order to be able to sustain and improve the legacy systems design patterns must to be identified and correctly detected. At this level, it is challenging to identify design pattern usage in source code because the patterns are not officially specified and there are numerous ways to present the code.

When using a resource that the entire application depends on but that could have problems when used by multiple instances at once, a singleton design is extremely helpful (it helps solve concurrency issues). This may occur in a variety of common development circumstances, including recording data to a file and gaining access to hardware interfaces.

Real life examples

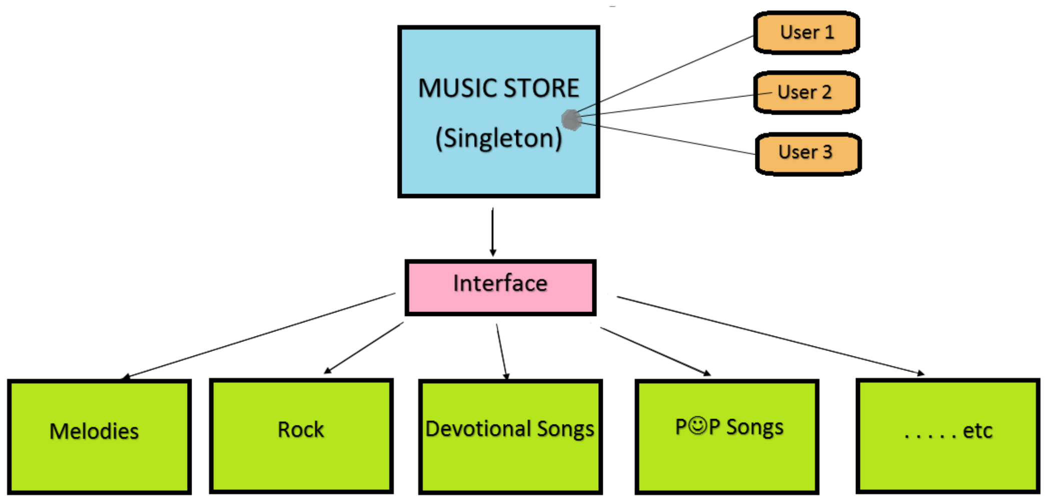
Let's imagine that I am asked to teach Java to numerous individuals. It would be difficult to spend the required time and energy on their individual training. I therefore requested that they view my Java course on YouTube. The Singleton pattern has been applied here. For the same context, individuals have requested me to construct many objects (with distinct Java coaching for each member), but I have responded that Java coaching is the same for everyone. So, kindly visit YouTube and watch my Java course (Singleton object).Singleton allows just one-time object creation so only one time I am going to post the videos. That video will be watched for N number of times when the bits of help.

Another Cocoa touch example is a physical device running an iOS application. There is only one iPhone or iPad with a single battery and screen for an app to execute. UIDevice is a Singleton class in this case because it only provides one channel for interacting with the underlying features. If the unique resource has a writable configuration, this type of discrepancy can cause issues like race conditions and deadlock. Singletons act as a control because they are unique, ensuring orderly access to the shared resource.

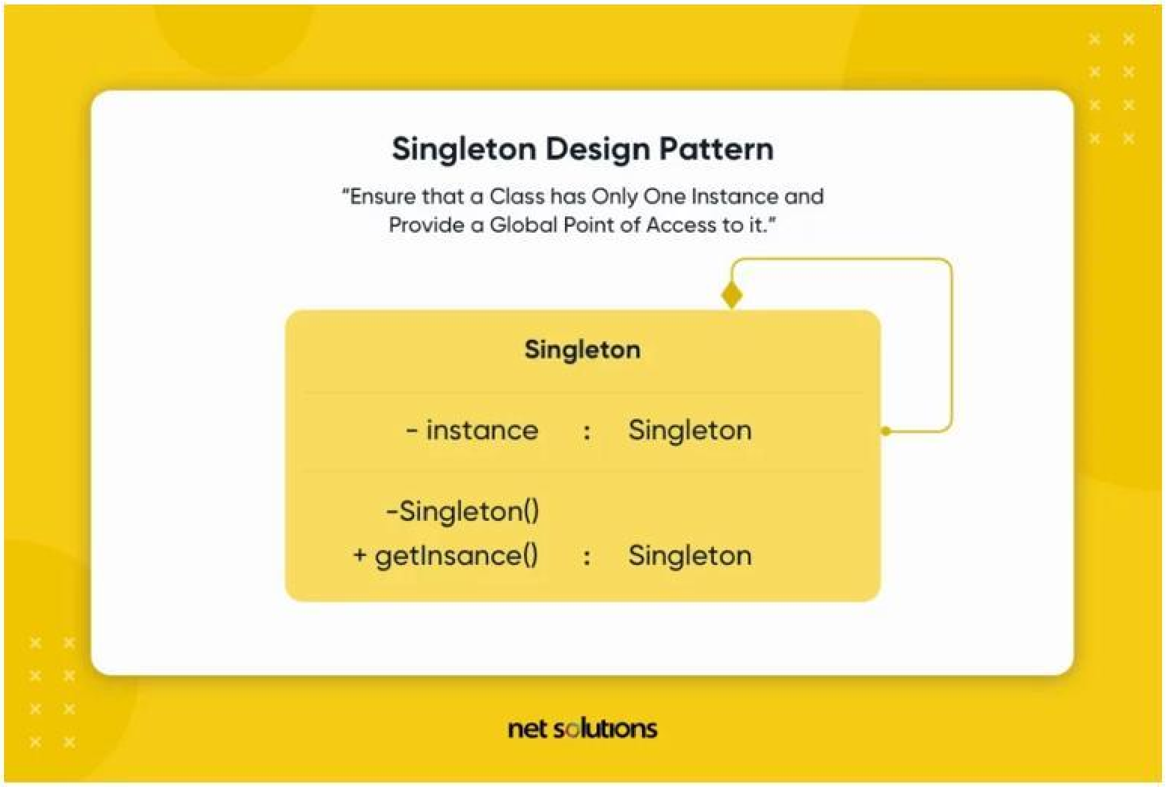
NodeJS and singleton pattern

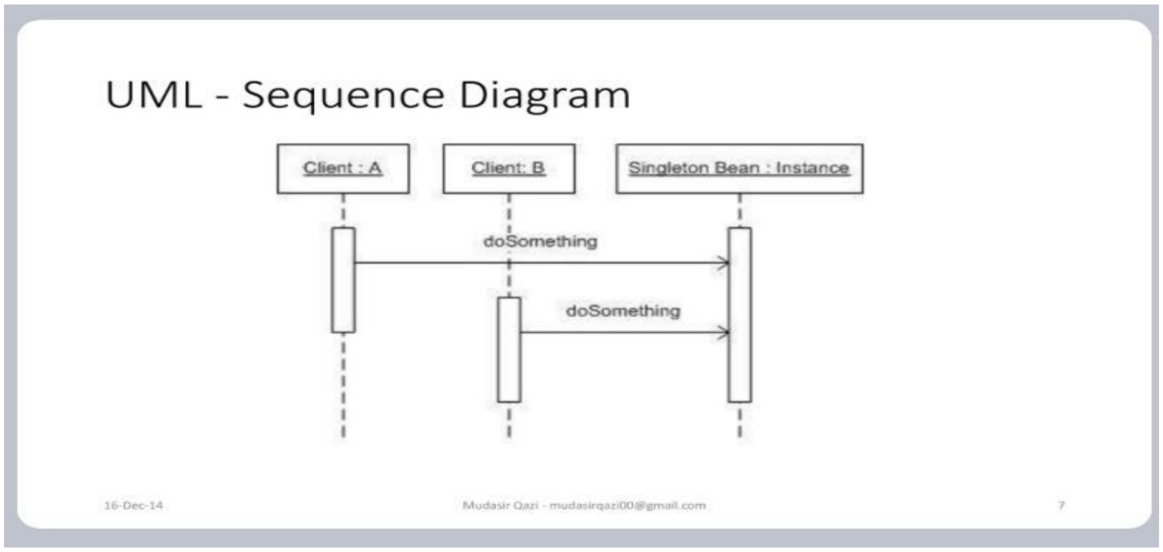
One well-known application of the Singleton pattern can be found in the Java class library's hidden parts.The audio device example I gave earlier is not a figment of my imagination. The sun.audio.AudioDevice class does exist and is used in the manner I described.The Gang of Four mentions several other known applications of Singleton patterns, including 1-Changeset in SmallTalk-80 and the 2-Session and WidgetKit classes in the InterViews user interface toolkit.

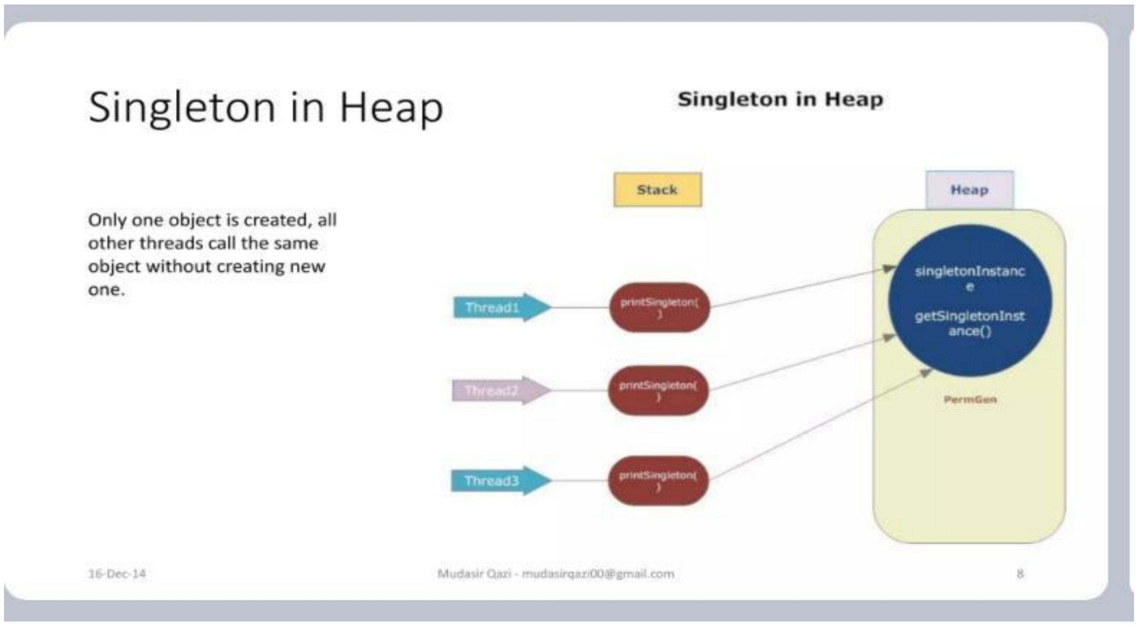
Once you've learned to recognise a design pattern, you'll notice it everywhere. The Java class library, in particular, is rife with classic design patterns. It is fairly obvious that at least some of the programmers writing code for the Java packages are pattern enthusiasts. As a result, the object in this case lacks the specific properties. It has a universal property, which allows it to be used an unlimited number of times. This is a real-world application of Singleton.



singleton pattern UML Diagrams


Singleton Design Pattern
"Ensure that a Class has Only One Instance and Provide a Global Point of Access to it."
Singleton
- instance : Singleton
-Singleton()
+ getInsance() : Singleton
net solutions




Singleton in Heap

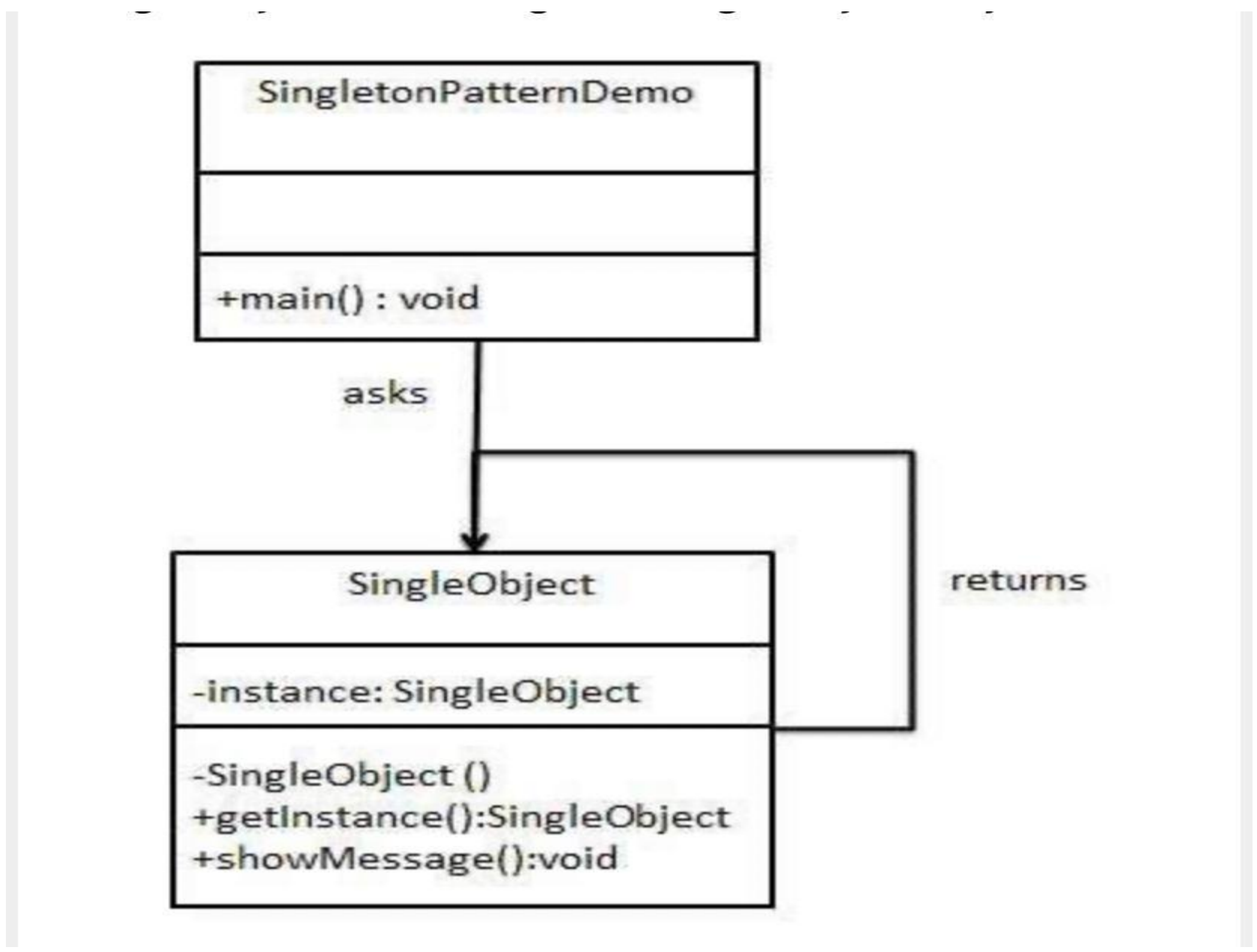
Only one object is created, all other threads call the same object without creating new one.

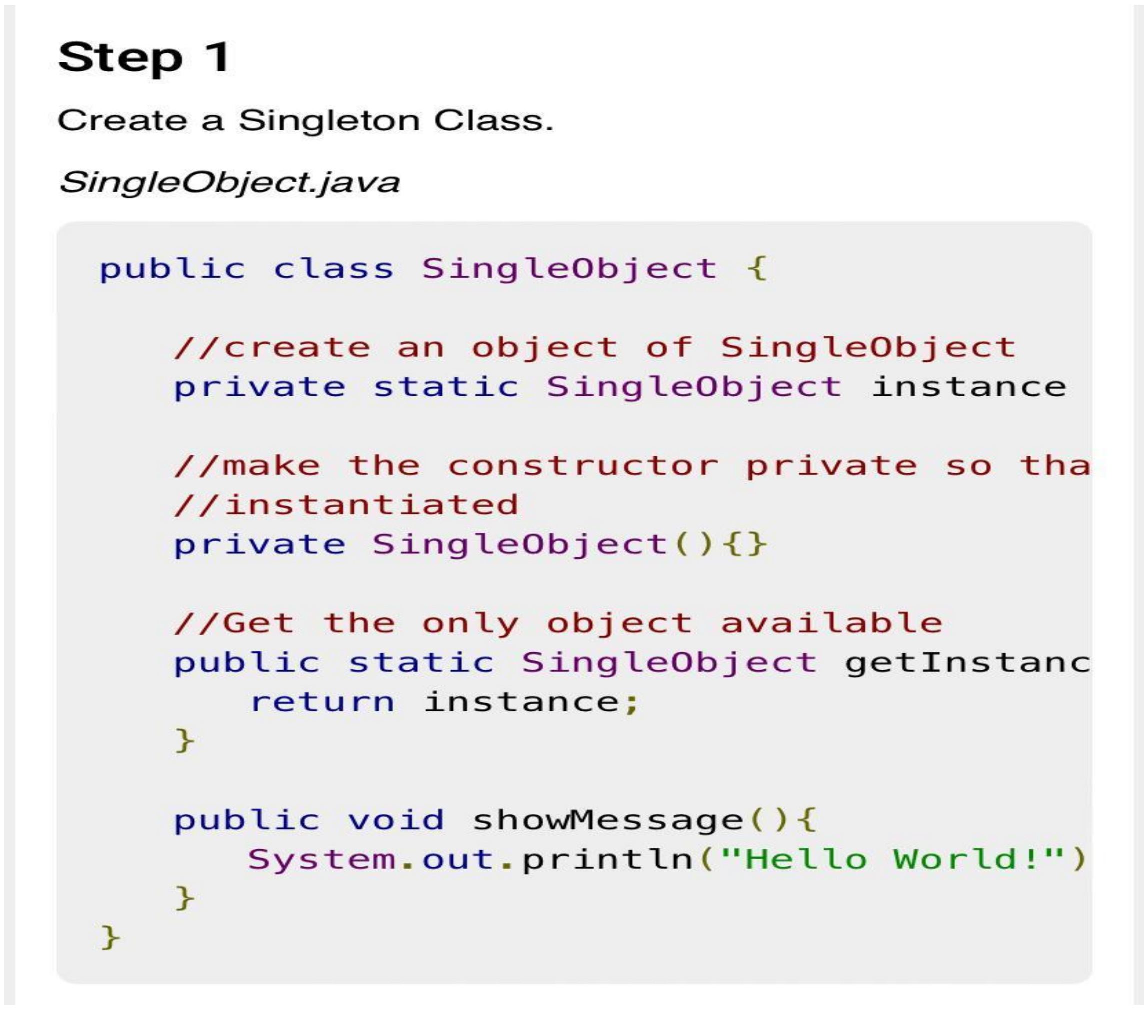
16-Dec-14 Mudasir Qazi - mudasirqazi00@gmail.com 8

Singleton implementation

We're going to create a *SingleObject* class. *SingleObject* class has its constructor as private and has a static instance of itself.

***SingleObject* class provides a static method to get its static instance to outside world. *SingletonPatternDemo*, our demo class will use *SingleObject* class to get a *SingleObject* object.**


SingletonPatternDemo
+main() : void
asks
returns
SingleObject
-instance: SingleObject
-SingleObject ()
+getInstance():SingleObject
+showMessage():void



Step 1

Create a Singleton Class.

SingleObject.java

```
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance

   //make the constructor private so tha
   //instantiated
   private SingleObject(){}

   //Get the only object available
   public static SingleObject getInstanc
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!")
   }
}
```

Step 2

Get the only object from the singleton class.

SingletonPatternDemo.java

```java
public class SingletonPatternDemo {
   public static void main(String[] args

      //illegal construct
      //Compile Time Error: The construc
      //SingleObject object = new Single

      //Get the only object available
      SingleObject object = SingleObject

      //show the message
      object.showMessage();
   }
}
```

Step 3

Verify the output.

```
Hello World!
```

Disadvantages

1. The global nature leads to dependency hiding.
2. It can be difficult to unit test the code.
3. It can lead to tightly coupled code.
4. If the single Instance of the object becomes corrupted, the entire system is compromised.

Advantages

1-Instance control: Singletons control access to a single instance by preventing other objects from creating their own instances of the Singleton object.

2-Flexibility: Because the class controls how objects are instantiated, it can modify that process as needed.

3-Saves Memory: It should go without saying that generating an object of any class takes up memory space, therefore creating several objects increases the amount of memory used. Singleton classes conserve memory because they can only contain one object.

6.Conclusion

In conclusion, Design Patterns exist to help us, developers. Of course that also exists an initial learning curve to know how to work with them, but once that you learned it, they can make your life easier. Design pattern is separated into 3 types.

In this article we talked about 2 types: the behavior design pattern and the creational design pattern . A creational design pattern deals with object creation and initialization, providing guidance about which objects are created for a given situation. These design patterns are used to increase flexibility and to reuse existing code.

Example on creational design pattern is the builder A step-by-step pattern for creating complex objects, separating construction and representation. Which is good for reuse when building more than an object with shared attributes.they are also more maintainable if the number of fields required to create an object is more than 4 or 5.However, they can be verbose as Builder needs to copy all fields from Original or Item class.

We also discussed the singleton which Restricts object creation for a class to only one instance. It is useful when you want to reuse something expensive to create resources. Since the creation is performed only once for the entire lifetime of the application, you are paying the price only once. It also solves two problems at the same time, violating the Single Responsibility Principle:

- Ensure that a class has just a single instance
- Provide a global access point to that instance

However the singleton pattern is hard to test as the test units depend on inheritance.and tightly coupled code.

As for the behavioral pattern, we covered the iterator pattern which is used on different data structures to access all the elements while encapsulating the data behind the interface. It is good for keeping the code clean by removing bulky loops for iteration and iterating over the same collection with different algorithms but it can be too complicated when used on simple data as arrays.

Finally the state pattern was discussed which is used to write code for programs that can have multiple different states and behave differently in each state. The main advantage of State Design Pattern is that it reduces the conditional complexity. Instead of writing multiple if statements for each state, we divide each state into their own separate classes with their own implementation. The disadvantage of State Patterns is that there needs to be a lot of code written. If the program has several states, each state must be written in separate classes and should override the methods defined in the State interface. For example if the program has 10 states, then 10 different classes will have to be implemented with their own behavior.

References

Alexander. S. (2018) Dive Into Design Patterns

GIBBONS, J., & OLIVEIRA, B. (2009). The essence of the Iterator pattern. *Journal of Functional Programming, 19*(3-4), 377-402. doi:10.1017/S0956796809007291

Sarcar, V. (2022). Iterator Pattern. In: Java Design Patterns. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-7971-7_18

Builder (refactoring.guru)
Builder in Java / Design Patterns (refactoring.guru)
thethirdsh/car-builder-design-pattern (github.com)
ElBarBary01/IteratorPattern(github.com)
https://www.baeldung.com/java-state-design-pattern
https://refactoring.guru/design-patterns/state