**The design of sports game framework**

**Introduction**

In my framework, I use three design patterns to address problem of game of sports similar with car racing. In this kind of game, we have an object which takes part in the game. It can be a car, a ship, or a man riding a bicycle. The game can be held in different place, in different weather. And there will be some data generated in the game, such as the speed of cars.

We can create a framework to this kind of game. We need a part of the program to build the object which is in the game, a generator to create a game scene and a tool to show the data which generated in the game.

**Body**

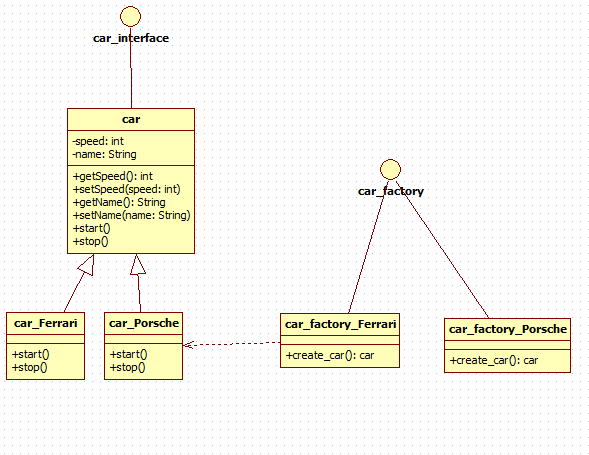
I will describe this framework by using the example blow. Consider this racing game, we choose a car which we want to drive. Then choose a scene which we want to hold the game. In the game, we can see the speed and other data of the car.

First of all, we focus on the object which takes part in the game. We can choose different objects. When we choose one object, the framework should create the object we want. The objects which we can choose are different in many cases, so the create method are different. All these objects have same signature, but they behave differently. We need different “factories” to create the objects. In this case, we can use the factory method pattern to deal with this part of problem.

The factory method pattern defines an interface for creating an object, but let subclasses decide which class to instantiate, and let a class defer instantiation to subclasses.

In our example, we can choose different brands of cars. Then these brands are the “factories”. Each factory can create its own cars, and the cars are different.

The UML diagram of “car factory”:



*figure 1. car factory*

We first create an interface of cars:

**package** car;

**public** **interface** car\_interface {

**public** **interface** car {

**public** **void** start();

**public** **void** stop();

}

}

Create a class of the information of cars:

**package** car;

**public** **class** car **implements** car\_interface{

**private** **int** speed;

**private** String name;

**public** **int** getSpeed() {

**return** speed;

}

**public** **void** setSpeed(**int** speed) {

**this**.speed = speed;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **void** start() {

// **TODO** Auto-generated method stub

}

**public** **void** stop() {

// **TODO** Auto-generated method stub

}

}

Based on the car class, we can create two subclasses which are the concrete “productions” of factory.

**package** car;

**public** **class** car\_Ferrari **extends** car{

**public** **void** start() {

System.*out*.println(**this**.getName() + " is on the road.　Speed："

+ **this**.getSpeed());

}

**public** **void** stop() {

System.*out*.println(**this**.getName() + " finish the game.");

}

}

**package** car;

**public** **class** car\_Porsche **extends** car {

**public** **void** start () {

System.*out*.println(**this**.getName() + " is on the road.　Speed："

+ **this**.getSpeed());

}

**public** **void** stop () {

System.*out*.println(**this**.getName() + " finish the game.");

}

}

We then create an interface of car factory:

**package** car;

**public** **interface** car\_factory {

**public** car create\_car();

}

And create two concrete factories to create the cars:

**package** car;

**public** **class** car\_factory\_Ferrari **implements** car\_factory{

**public** car create\_car() {

car\_Ferrari Ferrari = **new** car\_Ferrari();

Ferrari.setName("Ferrari F60");

Ferrari.setSpeed(300);

**return** Ferrari;

}

}

**package** car;

**public** **class** car\_factory\_Porsche **implements** car\_factory{

**public** car create\_car() {

car\_Porsche Porsche = **new** car\_Porsche();

Porsche.setName("Porsche 911");

Porsche.setSpeed(320);

**return** Porsche;

}

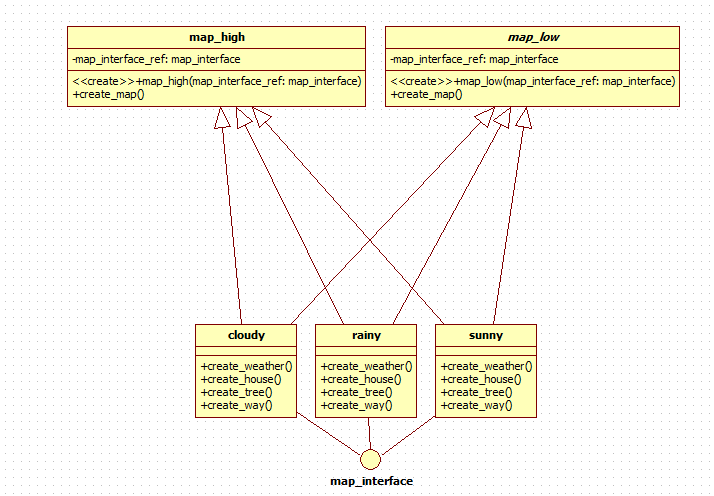
}

By using the factory method pattern, we can delay the creation of the object to subclass to instantiate. Each factory creates different brands of cars. Each car’s function has its own method to achieve. In this way, we can separate the creation of different brands of cars. It helps clarity of the object creation.

Second, we focus on the creation of game scenes. The scenes have some elements, such as the weather, buildings, trees and grass, and road. The elements are different, but the processes of building the scenes are same. Hence, builder pattern is a good choice.

In this example, we have two builders for high quality images and low quality images to create scenes which can adapt different computer.

The UML diagram of “map builder”:



*figure 2. map builder*

We create an interface of map:

**package** map;

**public** **interface** map\_interface {

**public** **void** create\_weather();

**public** **void** create\_house();

**public** **void** create\_tree();

**public** **void** create\_way();

}

Then create three classes for different maps:

**package** map;

**public** **class** sunny **implements** map\_interface {

**public** **void** create\_weather() {

System.*out*.println("Weather: Sunny");

}

**public** **void** create\_house() {

System.*out*.println("Buildings: Glass is bright.");

}

**public** **void** create\_tree() {

System.*out*.println("Trees: Leaves are bright green.");

}

**public** **void** create\_way() {

System.*out*.println("Road: Dry and plat.");

}

}

**package** map;

**public** **class** rainy **implements** map\_interface{

**public** **void** create\_weather() {

System.*out*.println("Weather: Rainy");

}

**public** **void** create\_house() {

System.*out*.println("Buildings: Glass is wet.");

}

**public** **void** create\_tree() {

System.*out*.println("Trees: Leaves are dark green and wet.");

}

**public** **void** create\_way() {

System.*out*.println("Road: Wet and slippery.");

}

}

**package** map;

**public** **class** cloudy **implements** map\_interface{

**public** **void** create\_weather() {

System.*out*.println("Weather: Cloudy");

}

**public** **void** create\_house() {

System.*out*.println("Buildings: Glass is dark.");

}

**public** **void** create\_tree() {

System.*out*.println("Trees: Leaves are dark green.");

}

**public** **void** create\_way() {

System.*out*.println("Road: A little wet and plat.");

}

}

And two builders for different qualities:

**package** map;

**public** **class** map\_high {

**private** map\_interface map\_interface\_ref;

**public** map\_high(map\_interface map\_interface\_ref) {

**super**();

**this**.map\_interface\_ref = map\_interface\_ref;

}

**public** **void** create\_map() {

System.*out*.println("Create a high performance map");

map\_interface\_ref.create\_weather();

map\_interface\_ref.create\_house();

map\_interface\_ref.create\_tree();

map\_interface\_ref.create\_way();

}

}

**package** map;

**public** **abstract** **class** map\_low {

**private** map\_interface map\_interface\_ref;

**public** map\_low(map\_interface map\_interface\_ref) {

**super**();

**this**.map\_interface\_ref = map\_interface\_ref;

}

**public** **void** create\_map() {

System.*out*.println("Create a low performance map");

map\_interface\_ref.create\_weather();

map\_interface\_ref.create\_house();

//map\_interface\_ref.create\_tree();

map\_interface\_ref.create\_way();

}

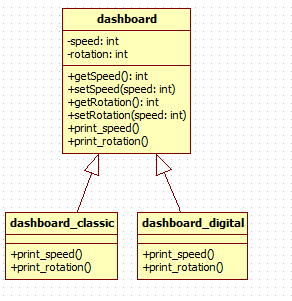
}

By using builder pattern, the process of creation has been encapsulated. The creation process does not need to do in the main method. In the way, the internal creation process and presentation layer code are separated, such that it is easy to modify the creation process.

Third, we finish the method to show the data. In the sports games, we can get the data of a car in the game through the dashboard, or see the big screen. The data is same, although the ways to show the data are different. The template method pattern defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure. Hence, it is good to using template method pattern to deal with the presentation of the data in the game.

In this example, we designed two ways to show the data, the classical and the digital. We can get the speed and rotation speed of the car, and show them in different styles.

The UML diagram of “dashboard”:



*figure 3. dashboard*

We need to create a dashboard class to define the skeleton:

**package** racing;

**public** **class** dashboard {

**private** **int** speed;

**private** **int** rotation;

**public** **int** getSpeed() {

**return** speed;

}

**public** **void** setSpeed(**int** speed) {

**this**.speed = speed;

}

**public** **int** getRotation() {

**return** rotation;

}

**public** **void** setRotation(**int** speed) {

**this**.rotation = speed;

}

**public** **void** print\_speed() {

System.*out*.println("get speed: " + **this**.getSpeed());

}

**public** **void** print\_rotation() {

System.*out*.println("get rotating speed: " + **this**.getRotation());

}

}

Then create two different styles of dashboard:

**package** racing;

**public** **class** dashboard\_classic **extends** dashboard{

**public** **void** print\_speed() {

**super**.print\_speed();

System.*out*.println("The speed " + **this**.getSpeed() + " showed in classical mode.");

}

**public** **void** print\_rotation() {

**super**.print\_rotation();

System.*out*.println("The rotating speed " + **this**.getRotation() + " showed in classical mode.");

}

}

**package** racing;

**public** **class** dashboard\_digital **extends** dashboard{

**public** **void** print\_speed() {

**super**.print\_speed();

System.*out*.println("The speed" + **this**.getSpeed() + "showed in digital mode.");

}

**public** **void** print\_rotation() {

**super**.print\_rotation();

System.*out*.println("The rotating speed" + **this**.getRotation() + "showed in digital mode.");

}

}

This is the template method pattern. It makes a father class to the same function, and then using the subclasses to achieve other different functions.

At last, we need a client class to test all these functions.

**package** game;

**import** car.car\_factory\_Ferrari;

**import** car.car\_factory;

**import** car.car;

**import** map.rainy;

**import** map.map\_high;

**import** racing.dashboard\_classic;

**import** racing.dashboard;

**public** **class** race {

/\*\*

\* **@param** args

\*/

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

// **TODO** Auto-generated method stub

rainy map\_rainy = **new** rainy();

map\_high map = **new** map\_high(map\_rainy);

map.create\_map();

System.*out*.println();

car\_factory car\_factory = **new** car\_factory\_Ferrari();

car car = car\_factory.create\_car();

car.start();

System.*out*.println();

dashboard dashboard\_classic = **new** dashboard\_classic();

dashboard\_classic.setSpeed(300);

dashboard\_classic.print\_speed();

dashboard\_classic.setRotation(6000);

dashboard\_classic.print\_rotation();

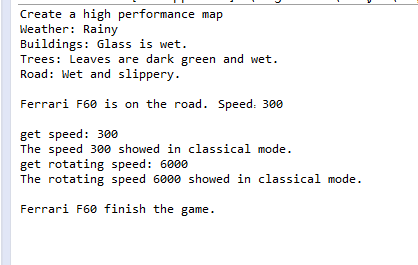
System.*out*.println();

car.stop();

}

}

The test result:



**Conclusion**

This framework use three design pattern: factory method pattern, builder pattern and template method pattern to deal with a kind of problem similar with sports game like racing. We just designed a function of “start” to achieve the car start the game. This function can be made more complex. For example, we can design the details of a car start to run. All this details can use façade pattern to do. And we can make our cars more beautiful by using decorator pattern. So, there are lots of works we can do to make this framework more useful. This framework also has some limitations. It only settles a part of problem of the sports game. For example, users cannot do anything in the process of the game. And there is no other competitor in the game. Hence, we can do further programming to make it describe a complete game.