```
    public void performWithColor() {
       System.out.println("Performing in Rectangle class");
       color.useBrush();
    }
}
```

Класс **Shape** – абстракция, классы **Circle** и **Rectangle** – уточненные абстракции.

```
/*пример # 20 : использование шаблона Bridge : Main.java */
package chapt05.bridge;
import chapt05.abstraction.*;
import chapt05.implementor.*;
public class Main {
    public static void main(String args[]) {
        YellowColor color = new YellowColor();
        new Rectangle(color).performWithColor();
        new Circle(color).performWithColor();
    }
}
```

Реализация больше не имеет постоянной привязки к интерфейсу. Реализацию абстракции можно динамически изменять и конфигурировать во время выполнения. Иерархии классов Abstraction и Implementor независимы и поэтому могут иметь любое число подклассов.

Шаблон Decorator

Необходимо расширить функциональные возможности объекта, используя прозрачный для клиента способ. Расширяемый класс реализует тот же самый интерфейс, что и исходный класс, делегируя исходному классу выполнение базовых операций. Шаблон Decorator даёт возможность динамического изменения поведения объектов в процессе выполнения приложения.

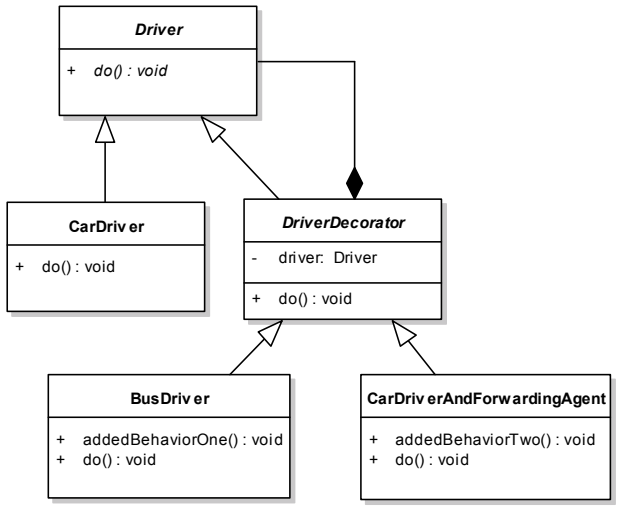

Рис. 5.10. Пример реализации шаблона Decorator

```
/*пример # 21 : определение интерфейса для компонентов : Driver.java */
package chapt05.decorator;
public abstract class Driver {
      public abstract void do();
   }
```

Класс **DriverDecorator** определяет для набора декораторов интерфейс, соответствующий интерфейсу класса **Driver**, и создает необходимые ссылки.

```
/*пример # 22 : интерфейс-декоратор для класса Driver : DriverDecorator.java */
package chapt05.decorator;
public abstract class DriverDecorator extends Driver {
      protected Driver driver;

      public DriverDecorator(Driver driver) {
            this.driver = driver;
      }
      public void do() {
            driver.do();
      }
   }
```

Класс **CarDriver** определяет класс, функциональность которого будет расширена.

```
/*пример # 23 : класс просто водителя : CarDriver.java */
package chapt05.decorator;
public class CarDriver extends Driver {
      public void do() { //базовая операция
            System.out.println("I am a driver");
      }
   }
```

Класс **BusDriver** добавляет дополнительную функциональность **addedBehaviorOne()** необходимую для водителя автобуса, используя функциональность **do()** класса **CarDriver**.

```
/*пример # 24 : класс водителя автобуса: BusDriver.java */
package chapt05.decorator;
public class BusDriver extends DriverDecorator {

      public BusDriver(Driver driver) {
            super(driver);
      }
      private void addedBehaviorOne() {
            System.out.println("I am bus driver");
      }
      public void do() {
            driver.do();
            addedBehaviorOne();
      }
   }
```

Класс **CarDriverAndForwardingAgent** добавляет дополнительную функциональность **addedBehaviorTwo()** необходимую для водителя-экспедитора, используя функциональность **do()** класса **CarDriver**.

```
/*пример # 25 : класс водителя-экспедитора:CarDriverAndForwardingAgent.java*/
package chapt05.decorator;
public class CarDriverAndForwardingAgent
                              extends DriverDecorator {

      public CarDriverAndForwardingAgent(Driver driver){
            super(driver);
      }
      private void addedBehaviorTwo() {
            System.out.println("I am a Forwarding Agent");
      }
      public void do() {
            driver.do();
            addedBehaviorTwo();
      }
   }
```

Создав экземпляр класса **CarDriver** можно делегировать ему выполнение задач, связанных с водителем автобуса или водителя-экспедитора, без написания специализированных полновесных классов.

```
/*пример # 26 : использование шаблона Decorator : Main.java */
package chapt05.decorator;
public class Main {
      public static void main(String args[]){
            Driver carDriver = new CarDriver();
            Main runner = new Main();
            runner.doDrive(carDriver);

            runner.doDrive(new BusDriver(carDriver));
            runner.doDrive(
            new CarDriverAndForwardingAgent(carDriver));
            }
      public void doDrive(Driver driver){
            driver.do();
      }
   }
```

Шаблоны поведения

Шаблоны поведения характеризуют способы взаимодействия классов или объектов между собой.

К шаблонам поведения относятся:

Chain of Responsibility (Цепочка Обязанностей) – организует независимую от объекта-отправителя цепочку не знающих возможностей друг друга объектов-получателей, которые передают запрос друг другу;