

Паттерны: Adapter, Proxy, Bridge

Java Collections
7 уровень, 2 лекция

ОТКРЫТА

— Привет, друг!

— Привет, Билаабо!

— У нас еще осталось немного времени, поэтому я расскажу тебе про еще три паттерна.

— Еще три, а сколько их всего?

— Ну, сейчас есть несколько десятков популярных паттернов, но количество «удачных решений» не ограничено.

— Ясно. И что, мне придется учить несколько десятков паттернов?

— Пока у тебя нет опыта реального программирования, они дадут тебе не очень много.

Ты лучше поднаберись опыта, а потом, через годик, вернись к этой теме и попробуй разобраться в них более основательно. Хотя бы пару десятков самых популярных.

Грех не пользоваться чужим опытом и самому что-то изобретать в очередной 110-й раз.

— Согласен.

— Тогда начнем.

Паттерн Adapter(Wrapper) – Адаптер (Обертка)



Представь, что ты приехал в Китай, а там другой стандарт розеток. Отверстия не круглые, а плоские. Тогда тебе понадобится переходник, или другими словами – адаптер.

В программировании тоже может быть что-то подобное. Классы оперируют похожими, но различными интерфейсами. И надо сделать переходник между ними.

Вот как это может выглядеть:

Пример

НАЧАТЬ ОБУЧЕНИЕ

```
1 interface Time
2 {
3     int getSeconds();
4     int getMinutes();
5     int getHours();
6 }
7
8 interface TotalTime
9 {
10     int getTotalSeconds();
11 }
```

Допустим, у нас есть два интерфейса – Time и TotalTime.

Интерфейс Time позволяет узнать текущее время с помощью методов getSeconds(), getMinutes() и getHours().

Интерфейс TotalTime позволяет получить количество секунд, которое прошло от полночи до текущего момента.

Что делать, если у нас есть объект типа TotalTime, а нужен Time и наоборот?

Для этого мы можем написать классы-адаптеры. Пример:

Пример
<pre>1 class TotalTimeAdapter implements Time 2 { 3 private TotalTime totalTime; 4 public TotalTimeAdapter(TotalTime totalTime) 5 { 6 this.totalTime = totalTime; 7 } 8 9 public int getSeconds() 10 { 11 return totalTime.getTotalSeconds() % 60; //секунды 12 } 13 14 public int getMinutes() 15 { 16 return (totalTime.getTotalSeconds() % (60*60)) / 60; //минуты 17 } 18 19 public int getHours() 20 { 21 return totalTime.getTotalSeconds() / (60*60); //часы 22 } 23 }</pre>

Как пользоваться
<pre>1 TotalTime totalTime = ... ; // программа получает объект, который реализует интерфейс TotalTime 2 Time time = new TotalTimeAdapter(totalTime); 3 System.out.println(time.getHours()+":"+time.getMinutes()+":"+time.getSeconds());</pre>

И адаптер в другую сторону:

Пример

```
1 class TimeAdapter implements TotalTime
2 {
3     private Time time;
4     public TimeAdapter(Time time)
5     {
6         this.time = time;
7     }
8
9     public int getTotalSeconds()
10    {
11        return time.getHours()*60*60+time.getMinutes()*60 + time.getSeconds();
12    }
13 }
```

Как пользоваться

```
1 Time time = ... ; // программа получает объект, который реализует интерфейс Time
2 TotalTime totalTime = new TimeAdapter(time);
3 System.out.println(totalTime.getTotalSeconds());
```

— Ага. Мне нравится. А примеры есть?

— Конечно, например, InputStreamReader – это классический адаптер. Преобразовывает тип InputStream к типу Reader.

Иногда этот паттерн еще называют обертка, потому что новый класс как бы «оборачивает» собой другой объект.

Другие интересные вещи почитать можно [тут](#).

Паттерн Проху — Заместитель

Паттерн прокси чем-то похож на паттерн "обертка". Но его задача – не преобразовывать интерфейсы, а контролировать доступ к оригинальному объекту, сохраненному внутри прокси-класса. При этом и оригинальный класс и прокси обычно имеют один и тот же интерфейс, что облегчает подмену объекта оригинального класса, на объект прокси.

Пример:

Интерфейс реального класса

```
1 interface Bank
2 {
3     public void setUserMoney(User user, double money);
4     public int getUserMoney(User user);
5 }
```

Реализация оригинального класса

```
1 class CitiBank implements Bank
2 {
3     public void setUserMoney(User user, double money)
4     {
5         UserDAO.updateMoney(user, money);
6     }
7
8     public int getUserMoney(User user)
```

```
11     }
12 }
```

Реализация прокси-класса

```
1  class BankSecurityProxy implements Bank
2  {
3      private Bank bank;
4      public BankSecurityProxy(Bank bank)
5      {
6          this.bank = bank;
7      }
8      public void setUserMoney(User user, double money)
9      {
10         if (!SecurityManager.authorize(user, BankAccounts.Manager))
11             throw new SecurityException("User can't change money value");
12
13         bank.setUserMoney(user, money);
14     }
15
16     public int getUserMoney(User user)
17     {
18         if (!SecurityManager.authorize(user, BankAccounts.Manager))
19             throw new SecurityException("User can't get money value");
20
21         return bank.getUserMoney(user);
22     }
23 }
```

В примере выше мы описали интерфейс банка – **Bank**, и одну его реализацию – **CitiBank**.

Этот интерфейс позволяет получить или изменить количество денег на счету пользователя.

А потом мы создали **BankSecurityProxy**, который тоже реализует интерфейс **Bank** и хранит в себе ссылку на другой интерфейс Bank. Методы этого класса проверяют: является ли данный пользователь владельцем счета либо менеджером банка, и если нет – то кидает исключение безопасности – **SecurityException**.

Вот как это работает на деле:

Код без проверки безопасности:

```
1  User user = AuthManager.authorize(login, password);
2  Bank bank = BankFactory.createUserBank(user);
3  bank.setUserMoney(user, 1000000);
```

Код с включённой проверкой безопасности:

```
1  User user = AuthManager.authorize(login, password);
2  Bank bank = BankFactory.createUserBank(user);
3  bank = new BankSecurityProxy(bank);
4  bank.setUserMoney(user, 1000000);
```

В первом примере мы создаем объект банк и вызываем у него метод **setUserMoney**.

Во втором примере мы оборачиваем оригинальный объект банк в объект **BankSecurityProxy**. Интерфейс у них один, так что

— Круто!

— Ага. Таких прокси может быть много. Например, можно добавить еще один прокси, который будет проверять – не слишком ли большая сумма. Может менеджер банка решил положить себе на счет кучу денег и сбежать с ними на Кубу.

Более того. Создание всех этих цепочек объектов можно поместить в класс **BankFactory** и подключать/отключать нужные из них.

По похожему принципу работает **BufferedReader**. Это **Reader**, но который делает еще дополнительную работу.

Такой подход позволяет «собирать» из «кусочков» объект нужной тебе функциональности.

Чуть не забыл. Прокси используются гораздо шире, чем я только что тебе показал. Об остальных типах использования ты можешь почитать [здесь](#).

Паттерн Bridge – Мост



Иногда, в процессе работы программы, надо сильно поменять функциональность объекта. Например, был у тебя в игре персонаж осел, а потом маг превратил его в дракона. У дракона совсем другое поведение и свойства, но(!) это – тот же самый объект.

— А нельзя просто создать новый объект и все?

— Не всегда. Допустим, твой осел был в друзьях у кучи персонажей, или например, на нем были наложены некоторые заклинания, или он участвовал в каких-то квестах. Т.е. этот объект уже может быть задействован в куче мест и привязан к куче других объектов. Так что просто создать новый другой объект в этом случае – не вариант.

— И что же делать?

— Одним из наиболее удачных решений есть паттерн Мост.

Этот паттерн предлагает разделить объект на два объекта. На «объект интерфейса» и «объект реализации».

— А в чем отличие от интерфейса и класса, который его реализует?

— В случае с интерфейсом и классом в результате будет создан один объект, а тут — два. Смотри пример:

Пример	
1	<code>class User</code>
2	<code>{</code>
3	<code> private UserImpl realUser;</code>
4	
5	<code> public User(UserImpl impl)</code>
6	<code> {</code>
7	<code> realUser = impl;</code>
8	<code> }</code>
9	
10	<code> public void run() { //создать</code>

```
12     realUser.run();
13 }
14
15 public void fly() //лететь
16 {
17     realUser.fly();
18 }
19 }
20
21 class UserImpl
22 {
23     public void run()
24     {
25     }
26
27     public void fly()
28     {
29     }
30 }
```

А потом можно объявить несколько классов наследников от UserImpl, например **UserDonkey**(осел) и **UserDragon**(дракон).

— Все равно не очень понял, как это будет работать.

— Ну, примерно так:

Пример	
1	class User
2	{
3	private UserImpl realUser;
4	
5	public User(UserImpl impl)
6	{
7	realUser = impl;
8	}
9	
10	public void transformToDonkey()
11	{
12	realUser = new UserDonkeyImpl();
13	}
14	
15	public void transformToDragon()
16	{
17	realUser = new UserDragonImpl();
18	}
19	}

Как это работает	
1	User user = new User(new UserDonkey()); //внутри мы – осел
2	user.transformToDragon(); //теперь внутри мы – дракон

— Чем-то напоминает прокси.

 +52 

Комментарии (40)

популярные

новые

старые

JavaCoder

Введите текст комментария

Макс Дудин Уровень 40, Калининград, Россия

28 июня, 12:38 

"прозрачное" объяснение паттерна "Заместитель" по ссылке в википедии....

"Создать суррогат реального объекта. «Заместитель» хранит ссылку, которая позволяет заместителю обратиться к реальному субъекту (объект класса «Заместитель» может обращаться к объекту класса «Субъект», если интерфейсы «Реального Субъекта» и «Субъекта» одинаковы). Поскольку интерфейс «Реального Субъекта» идентичен интерфейсу «Субъекта», так, что «Заместителя» можно подставить вместо «Реального Субъекта», контролирует доступ к «Реальному Субъекту», может отвечать за создание или удаление «Реального Субъекта». «Субъект» определяет общий для «Реального Субъекта» и «Заместителя» интерфейс так, что «Заместитель» может быть использован везде, где ожидается «Реальный Субъект». При необходимости запросы могут быть переадресованы «Заместителем» «Реальному Субъекту»."

лично я теряю нить прочитав 0,75 это текста ...
какой-то сон пронесон..
<https://www.youtube.com/watch?v=2sJFWlg7Tj0&t=18s>

Ответить

 0 

Igor Petrashevsky Уровень 47

27 августа, 02:40 

мы проходили Проху раньше.
суть была в том, что если оригинальный объект генерировал исключение и падал, мы catch'ем подставляли прокси-костыль, который был "тоже самое", только с данными-затычками, лишь бы дальше программа работала "как ни в чем не бывало",
Это был один из простых вариантов применения.

Все они - узаконенные костыли под какую-либо задачу, как панели в панельных домах.
Вот окно. Вот окно с балконной дверью, вот козырек, вот - лестница

Ответить

 0 

Макс Дудин Уровень 40, Калининград, Россия

28 августа, 22:37 

да это всё понятно... и с самим ргоху особо вопросов нет, просто объяснение прикольное...

Ответить

 0 

Жора Нет Уровень 39, енакиево, Украина

6 мая, 19:34 

Бляха, как это все запомнить???

Вроде бы понятно, но многие из них(паттернов) настолько похожи, что через пять минут после прочтения уже забыл как реализуется тот или иной шаблон

Ответить

 +1 

Igor Petrashevsky Уровень 47

27 августа, 02:32 

ну да, это сорта костылей, по сути, или велосипедов.
когда выяснится, что сам велосипедов наизобретал, а можно было чуть поправить и взять готовые - жить станет легче.

Ответить

 0 

LuneFox инженер по сопровождению в BIFIT EXPERT

24 февраля, 17:45 

Я ведь не один представлял осла и дракониху из Шрека?

Ответить

 +8 

Anna Avilova architect

26 апреля, 16:09 

Лиза Воренувкина

Уровень 43, Кривой Рог, Ukraine

26 сентября 2021, 13:54

...

Ox my made .

Ответить

⬇️

+1

⬆️

lightning

Уровень 41

20 августа 2021, 22:51

...

Никак не мог понять разницу между проху и decorator. Объяснения от Christopher Okhravi и refactoring.guru не помогли.

В итоге для себя определил следующее правило:

Decorator должен всегда выполнять логику сохраненного в нем объекта и дополнительно может (но не обязательно) выполнить свою логику.

Проху наоборот, должен всегда выполнить свою логику и при выполнении каких-то условий выполнить логику сохраненного в нем объекта.

Ответить

⬇️

+1

⬆️

Aleksandr Alekseenko

Network engineer

17 августа 2021, 04:27

...

А паттерн "Расчехлитель" есть?

Ответить

⬇️

0

⬆️

Михаил

Уровень 37, Россия

1 июня 2021, 16:32

...

Там, где "адаптер в другую сторону" опечатка.

Вместо

1

System.out.println(time.getTotalSeconds());

Нужно

1

System.out.println(totalTime.getTotalSeconds());

Ответить

⬇️

0

⬆️

MermaidMan

Уровень 41, Bikini Bottom

22 февраля 2021, 14:34

...

в классе TotalTimeAdapter я так понимаю ошибка:

1

int getMinutes()

2

{

3

return totalTime.getTotalSeconds() / 60; //минуты - возвращает целое кол-во минут

4

}

а должно быть:

1

return (totalTime.getTotalSeconds() % (60*60)) / 60; //целое кол-во минут с начала

Ответить

⬇️

+2

⬆️

Kex

Уровень 38, Тольятти

EXPERT

27 июля 2020, 21:50

...

Научиться бы это применять правильно еще.

Ответить

⬇️

0

⬆️

Kex

Уровень 38, Тольятти

EXPERT

28 июля 2020, 11:29

...

Бляха - муха запутался с этими adapter - decorator - проху, но вики и гугл дали четкие разделения и представления этих паттернов,. МОСТ - паттерн очень понравился своей гипкостью. Патерны и алгоритмы интересная и. занимательная штука но. наверное когда у тебя есть уже весомый опыт в разработке, а пока ты как маленький ребенок знаешь что оно есть но зачем и для чего слабо себе представляешь в реальном мире, но это дело времени и опыта.

Ответить

⬇️

+4

⬆️

Евгений

Ведущий инженер в ПАО Сбербанк

EXPERT

13 июля 2020, 21:22

...

Паттерн Прокси это интересно, но у нас же вроде стандартный класс для этого есть

Ответить

⬇️

0

⬆️

↺

Показать еще комментарии

НАЧАТЬ ОБУЧЕНИЕ

Курсы программирования

Курс Java

Помощь по задачам

Подписки

Задачи-игры

Пользователи

Статьи

Форум

Чат

Истории успеха

Активности

О нас

Контакты

Отзывы

FAQ

Поддержка



JavaRush — это интерактивный онлайн-курс по изучению Java-программирования с нуля. Он содержит 1200 практических задач с проверкой решения в один клик, необходимый минимум теории по основам Java и мотивирующие фишки, которые помогут пройти курс до конца: игры, опросы, интересные проекты и статьи об эффективном обучении и карьере Java-девелопера.

ПОДПИСЫВАЙТЕСЬ

ЯЗЫК ИНТЕРФЕЙСА

 Русский

▼

СКАЧИВАЙТЕ НАШИ ПРИЛОЖЕНИЯ

 ДОСТУПНО В
Google Play

 Загрузите в
App Store

