***Глава*** ***2***

**ПОРОЖДАЮЩИЕ** **ШАБЛОНЫ**

*Шаблоны* *—* *это* *странные* *и* *загадочные* *существа* *с* *душевнобольной* *иерархической* *системой.* *Как* *известно,* *тушки* *этих* *мифических* *существ* *часто* *используют* *жре-*

*цы* *ООП* *для* *своих* *бессвязных* *заклинаний.*

<Абсурдопедия>

Шаблоны проектирования GoF — это многократно используемые решения широко распространенных проблем, возникающих при разработке программ-ного обеспечения. Многие разработчики искали пути повышения гибкости и степени повторного использования своих программ. Найденные решения во-площены в краткой и легко применимой на практике форме.

«Любой шаблон описывает задачу, которая снова и снова возникает в нашей работе, а также принцип ее решения, причем таким образом, что это решение можно потом использовать миллион раз, ничего не изобретая заново».

(Кристофер Александер).

В общем случае шаблон состоит из четырех основных элементов:

1) *имя.* Точное имя предоставляет возможность сразу понять проблему и опре-делить решение. Уровень абстракции при проектировании повышается;

2) *задача.* Область применения в рамках решения конкретной проблемы;

3) *решение.* Абстрактное описание элементов дизайна задачи проектирования и способа ее решения с помощью обобщенного набора классов;

4) *результаты.*

Шаблоны классифицируются по разным критериям, наиболее распростра-ненным из которых является назначение шаблона. Вследствие этого выделяют-ся порождающие шаблоны, структурные шаблоны и шаблоны поведения.

Несмотря на все различия, шаблоны взаимосвязаны, и нередко небольшое изменение ответственности или связи того или иного класса может привести к тому, что применение одного шаблона приводит к замене исходного шаблона на другой, часто даже неродственный.

Порождающие шаблоны предназначаются для организации процесса созда-ния объектов и все до единого соответствуют шаблону Creator из GRASP.

К порождающим шаблонам относятся:

**Abstract** **Factory** (**Абстрактная** **Фабрика**) — предоставляет интерфейс для создания связанных между собой объектов семейств классов без указания их конкретных реализаций (families of product objects);

**605**

**ШАБЛОНЫ** **ПРОЕКТИРОВАНИЯ**

**Factory** **Method** (**Фабричный** **метод**) — определяет интерфейс для созда-ния объектов из иерархического семейства классов на основе передаваемых данных (subclass of object that is instantiated);

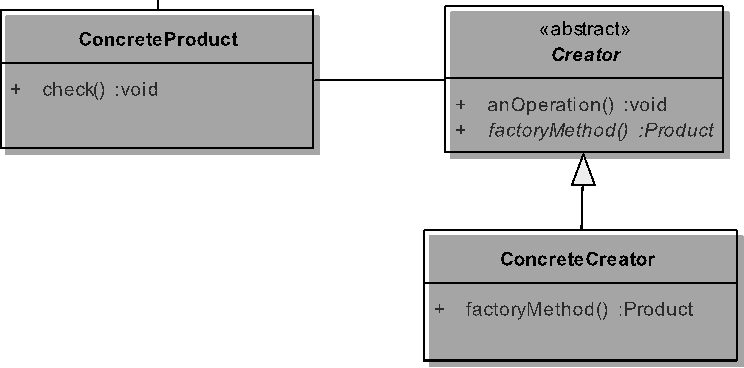
**Builder** (**Строитель**) — создает объект конкретного класса (how a composite object gets created);

**Singleton** (**Одиночка**) — гарантирует существование только одного или ко-нечного числа экземпляров класса (the sole instance of a class);

**Prototype** (**Прототип**) — применяется при создании сложных объектов. На основе прототипа объекты сохраняются и воссоздаются, например, путем копирования (class of object that is instantiated).

**Шаблон** **Factory** **Method**

Необходимо определить механизм создания объектов по заданному призна-ку для классов, находящихся в иерархической структуре. Классу-фабрике и его подклассам разрешается создавать экземпляры с помощью общего для всех интерфейса, причем именно подклассы решают, какой экземпляр следует со-здавать. Этот интерфейс может содержаться в абстрактном классе, интерфейсе или даже в обычном классе. Фабричный метод не должен владеть информаци-ей о конкретном типе создаваемого объекта, но знает о классе-вершине иерар-хии создаваемых объектов. Фабричный метод (**Factory** **Method**) — основа для функционирования абстрактной фабрики (**Abstract** **Factory**).

**Рис.** **2.1.** *Базовая* *реализация* *шаблона* *Factory* *Method*



**606**

**ПОРОЖДАЮЩИЕ** **ШАБЛОНЫ**

Следующим вариантом применения шаблона будет выделение каждому классу создаваемых экземпляров своего персонального метода. Основной класс шаблона представляет собой класс, который имеет различные методы для создания объектов.

*/\** *#* *4* *#* *вариант* *фабрики* *с* *различными* *методами* *для* *создания* *экземпляров,* *объявленных* *в* *одном* *классе* *#* *OrderFactoryVariant.java* *\*/*

**package** by.bsu.factory;

**public** **class** OrderFactoryVariant { *// промежуточная реализация шаблона*

**public** SimpleOrder createSimpleOrder() {

**return** **new** SimpleOrder();

}

**public** ExtendedOrder createExtendedOrder() {

**return** **new** ExtendedOrder();

}

}

Процесс создания экземпляров по-прежнему сосредоточен в одном классе, что делает класс подверженным изменениям при расширении иерархии созда-ваемых объектов.

Если процессу создания экземпляра одного класса выделять собствен-ный класс, находящийся, в свою очередь, в иерархической зависимости от некоего абстрактного создателя **OrderFactory**. В этой ситуации шаблон **Factory** **Method** становится весьма близок простой вариации шаблона **Abstract** **Factory**, особенно если добавить к реализации класс-клиент, ответственный за действия по инициализации фабричного метода и ис-пользованию созданного экземпляра. В базовой реализации фактически такие действия не выделяются в отдельном методе или классе, а произво-дятся по необходимости.

**609**

**ШАБЛОНЫ** **ПРОЕКТИРОВАНИЯ**

Реализация с клиентом и с использованием метода подставки при создании конкретных фабричных методов.

*/\** *#* *5* *#* *вариант* *фабрики* *с* *классом-клиентом* *для* *различных* *классов* *создания* *экземпляров* *#* *OrderFactory.java* *#* *SimpleOrderFactory.java* *#* *ExtendedOrderFactory.java* *#* *Client.java* *\*/*

**package** by.bsu.factory.client;

**public** **abstract** **class** OrderFactory {

**public** **abstract** AbstractOrder createOrder();

}

**package** by.bsu.factory.client;

**public** **class** SimpleOrderFactory **extends** OrderFactory {

@Override

**public** SimpleOrder createOrder() {

**return** **new** SimpleOrder();

}

}

**package** by.bsu.factory.client;

**public** **class** ExtendedOrderFactory **extends** OrderFactory {

@Override

**public** ExtendedOrder createOrder() {

**return** **new** ExtendedOrder();

}

}

**package** by.bsu.factory.client;

**public** **class** Client {

**public** AbstractOrder createOrderFromFactory(OrderFactory factory) {

AbstractOrder order = factory.createOrder();

return order;

}

}

Инициализация фабрики в классе **Client** может производиться не только в методе, но и через соответствующий конструктор или отдельный метод. Таким же образом может создаваться и использоваться экземпляр реализаций класса **AbstractOrder**.

**610**

**ПОРОЖДАЮЩИЕ** **ШАБЛОНЫ**

Недостаток последней реализации состоит в потребности создавать подкласс фабрики для создания экземпляров нового класса в иерархии. Недостаток превращается в достоинство, если код рассматривать с точки зрения структурированности.

*/\** *#* *6* *#* *параметризованная* *фабрика* *#* *InstanceFactory.java* *#* *SimpleOrderFactory.java* *#* *ExtendedOrderFactory.java* *\*/*

**package** by.bsu.factory.generic;

**public** **interface** InstanceFactory<R **extends** AbstractOrder> {

R createInstance(String data);

}

**package** by.bsu.factory.generic;

**public** **class** SimpleOrderFactory **implements** InstanceFactory<SimpleOrder> {

@Override

**public** SimpleOrder createInstance(String data) {

**return** **new** SimpleOrder();

}

}

**package** by.bsu.factory.generic;

**public** **class** ExtendedOrderFactory **implements** InstanceFactory<ExtendedOrder> {

@Override

**public** ExtendedOrder createInstance(String data) {

**return** **new** ExtendedOrder();

}

}

Наличие общих методов, контролирующих корректность процесса создания каждого экземпляра класса определяет преимущество шаблона **Factory** **Method** перед простой инициализацией экземпляра.

Программная реализация может быть представлена в общем виде следующим образом.

*/\** *#* *2* *#* *реализация* *шаблона* *с* *одним* *статическим* *методом* *#* *AbstractOrder.java* *#* *SimpleOrder.java* *#* *ExtendedOrder.java* *#* *TypeOrder.java* *#* *OrederFactory.java* *\*/*

**package** by.bsu.pattern.factory;

**public** **abstract** **class** AbstractOrder {

}Background pattern

Description automatically generated

**package** by.bsu.pattern.factory;

**public** **class** SimpleOrder **extends** AbstractOrder {

//

}

**package** by.bsu.pattern.factory;

**public** **class** ExtendedOrder **extends** AbstractOrder {

//

}

**package** by.bsu.pattern.factory;

**public** **class** OrderFactory { *// primitive factory*

**public static** AbstractOrder createOrder (String type) {

return **switch**(type) {

**case** "SIMPLE" -> **new** SimpleOrder();

**case** "**EXTENDED**" -> **new** ExtendedOrder();

**default** -> **throw** **new** IllegalArgumentException();

}

}

}

В качестве параметра метода передается некоторое значение, в соответствии с которым будет осуществляться инициализация объекта одного из подклассов абстрактного класса.

Приведенная реализация проста для восприятия, но обладает рядом недостатков. При добавлении новых подклассов в иерархию создаваемых объектов потребуется внесение изменений в сам фабричный метод. При разрастании иерархии будет усложняться логика процесса создания объектов, что повлечет разрастание кода фабричного метода с повышением его уровня связанности.

**608**

Background pattern

Description automatically generated**ПОРОЖДАЮЩИЕ** **ШАБЛОНЫ**

*/\** *#* *3* *#* *создание* *объектов* *с* *помощью* *шаблона* *#* *RunFactoryMethodSimplest.java* *\*/*

**package** by.bsu.factory;

**public** **class** RunFactoryMethodSimplest {

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

AbstractOrder ob1 = OrderFactory.createOrder ("SIMPLE");

AbstractOrder ob2 = OrderFactory.createOrder ("EXTENDED");

}

}

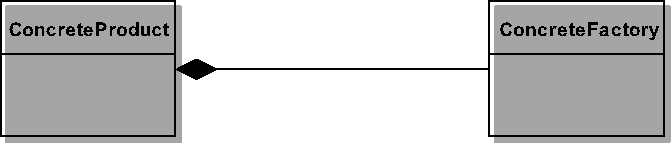
На практике иерархия классов для создаваемых экземпляров может быть достаточно разветвленной, да и сам процесс создания одного экземпляра может состоять из большого числа действий.

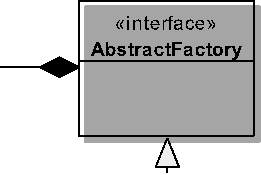
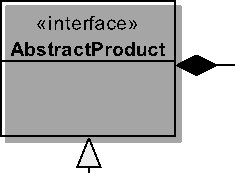
**Шаблон** **Abstract** **Factory**

Необходимо создавать объекты классов, не имеющих иерархической связи, но логически связанных между собой. Абстрактный класс-фабрика определяет общий интерфейс таких фабрик. Его подклассы обладают конкретной реализа-цией методов по созданию разных объектов.

**611**

**ШАБЛОНЫ** **ПРОЕКТИРОВАНИЯ**

**Рис.** **2.2.** *Базовая* *реализация* *шаблона* *AbstractFactory*





Предложенное решение делает конкретные классы обособленными. Так как абстрактная фабрика реализует процесс создания классов-фабрик и саму процедуру инициализации объектов, то она изолирует приложение от деталей реализации классов. Порождаемые классы должны находиться в иерархической зависимости внутри своего семейства.

Шаблон может применяться также и для создания объектов только одного семейства. Базовая реализация шаблона представляет как раз такой случай.

*/\** *#* *7* *#* *базовая* *реализация* *Abstract* *Factory* *#* *AbstractProduct.java* *#* *ConcreteProduct.java* *#* *AbstractFactory.java* *#* *ConcreteFactory.java* *#* *Client.javaunner.java\*/*

**package** by.bsu.abstractfactory;

**public** **interface** AbstractProduct {

**void** about();

}

**package** by.bsu.abstractfactory;

**public** **class** ConcreteProduct **implements** AbstractProduct {

*// fields, consructors, methods*

**public** **void** about() {

System.out.println("Concrete product");

}

}

**package** by.bsu.abstractfactory;

**612**

**ПОРОЖДАЮЩИЕ** **ШАБЛОНЫ**

**public** **interface** AbstractFactory {

**public** AbstractProduct createProduct();

}

**package** by.bsu.abstractfactory;

**public** **class** ConcreteFactory **implements** AbstractFactory {

**public** AbstractProduct createProduct() {

System.***out***.println("Creating concrete product");

AbstractProduct product = **new** ConcreteProduct();

**return** product;

}

}

**package** by.bsu.abstractfactory;

**public** **class** Client {

**private** AbstractFactory abstractFactory;

**private** AbstractProduct abstractProduct;

**public** **void** action() {

abstractProduct = abstractFactory.createProduct();

abstractProduct.about();

}

**public** **void** setAbstractFactory(AbstractFactory factory) {

abstractFactory = factory;

}

}

**package** by.bsu.abstractfactory;

**public** **class** Runner {

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

Client client = **new** Client();

client.setAbstractFactory(**new** ConcreteFactory());

client.action();

}

}

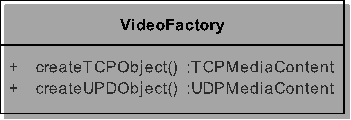
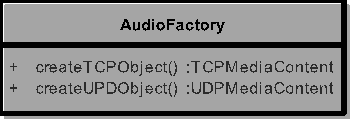
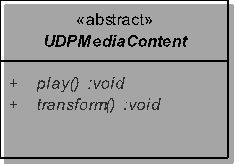
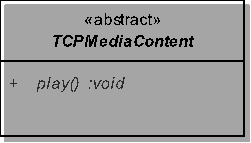
Одна из возможных реализаций шаблона предложена в следующем приме-ре. Классы фабрики создаются по тому же принципу, по которому в предыду-щем шаблоне создавались объекты.

Производители объектов реализуют методы по созданию не связанных ие-рархическими зависимостями объектов. Класс **AbstractMediaFactory** — аб-страктная фабрика, аклассы **AudioFactory** и **VideoFactory** — конкретные произ-водители объектов, наследуемые от нее. Конкретные фабрики могут создавать объекты-продукты для TCP и UDP протоколов.

**613**

 **ШАБЛОНЫ** **ПРОЕКТИРОВАНИЯ**

**Рис.** **21.3.** *Пример* *реализации* *шаблона* *AbstractFactory*



*/\** *#* *8* *#* *классы-фабрики* *по* *созданию* *несвязанных* *объектов* *#* *AbstractMediaFactory.java* *#* *AudioFactory.java* *#* *VideoFactory.java* *\*/*

**package** by.bsu.abstractfactory.media;

**public** **abstract** **class** AbstractMediaFactory {

**public** **abstract** TCPMediaContent createTCPObject();

**public** **abstract** UDPMediaContent createUDPObject();

}

**package** by.bsu.abstractfactory.media;

**public** **class** AudioFactory **extends** AbstractMediaFactory {

**public** TCPMediaContent createTCPObject() {

**return** **new** TCPAudioContent();

}

**614**

**ПОРОЖДАЮЩИЕ** **ШАБЛОНЫ**

**public** UDPMediaContent createUDPObject() {

**return** **new** UDPAudioContent();

}

}

**package** by.bsu.abstractfactory.media;

**public** **class** VideoFactory **extends** AbstractMediaFactory {

**public** TCPMediaContent createTCPObject() {

**return** **new** TCPVideoContent();

}

**public** UDPMediaContent createUDPObject() {

**return** **new** UDPVideoContent();

}

}

Рассматриваются два вида классов-продуктов:  *Имя***AudioContent**, *Имя***VideoContent**. Каждый из них может быть представлен в одном из двух видов: для TCP протокола или для UDP.

*/\** *#* *9* *#* *классы-продукты* *#* *TCPMediaContent.java* *#* *TCPAudioContent.java* *#* *TCPVideoContent.java* *#* *UDPMediaContent.java* *#* *UDPAudioContent.java* *#* *UDPVideoContent.java* *#* *Client.java* *\*/*

**package** by.bsu.abstractfactory.media;

**public** **abstract** **class** TCPMediaContent {

**abstract** **void** play();

}

**package** by.bsu.abstractfactory.media;

**public** **class** TCPAudioContent **extends** TCPMediaContent {

**void** play() {

// more code

}

}

**package** by.bsu.abstractfactory.media;

**public** **class** TCPVideoContent **extends** TCPMediaContent {

**void** play() {

// more code

}

}

**package** by.bsu.abstractfactory.media;

**public** **abstract** **class** UDPMediaContent {

**abstract** **void** play();

**abstract** **void** transform();

}

**package** by.bsu.abstractfactory.media;

**public** **class** UDPAudioContent **extends** UDPMediaContent {

**void** play() {

// more code

}

 **615**

 **ШАБЛОНЫ** **ПРОЕКТИРОВАНИЯ**

**void** transform() {

// more code

}

}

**package** by.bsu.abstractfactory.media;

**public** **class** UDPVideoContent **extends** UDPMediaContent {

**void** play() {

// more code

}

**void** transform() {

// more code

}

}

**package** by.bsu.abstractfactory.media;

**public** **class** Client {

**private** UDPMediaContent contentUDP;

**private** TCPMediaContent contentTCP;

**public** **void** makeMediaFactoryWork(AbstractMediaFactory factory) {

contentUDP = factory.createUDPObject();

contentTCP = factory.createTCPObject();

*//*

}

}

Признаки использования шаблона **Abstract** **Factory** при создании семейств объектов:

— необходимо создавать объекты из одного или нескольких семейств; — семейства имеют иерархическую внутреннюю структуру;

— между классами различных семейств могут прослеживаться логические связи; — способ создания объектов должен быть скрыт.

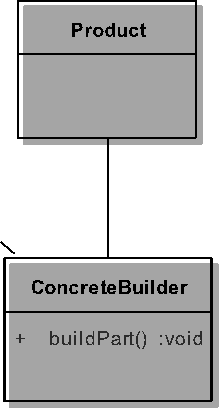
Отличительной чертой шаблона **Abstract** **Factory** является определение типа объекта по внешнему признаку.

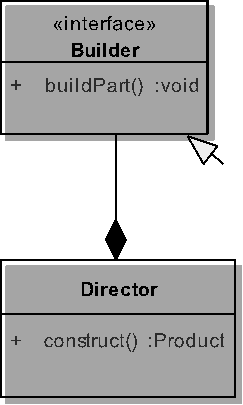
**Шаблон** **Builder**

Шаблон **Builder** заранее знает тип объекта, в то время, как его основной за-дачей является определение способа создания объекта на основе значений пе-редаваемых параметров. Необходимо реализовать процесс инициализации сложного объекта или объекта со сложным процессом конструирования, опре-деляя для него только тип и свойства. Детали построения объекта или его частей

**616**

**ПОРОЖДАЮЩИЕ** **ШАБЛОНЫ**

**Рис.** **21.4.** *Базовая* *реализации* *шаблона* *Builder*



остаются скрытыми и определяются в подклассах. Базовая реализация выглядит следующим образом.

*/\** *#* *10* *#* *базовая* *реализация* *Builder* *#* *Product.java* *#* *Builder.java* *#* *ConcreteBuilder.java* *#* *Director.java* *#* *Runner.java* *\*/*

**package** by.bsu.builder.base;

**public** **class** Product {

**private** **int** part1;

**private** String part2;

**public** **void** setPart1(**int** part1) {

**this**.part1 = part1;

}

**public** **void** setPart2(String part2) {

**this**.part2 = part2;

}

}

**package** by.bsu.builder.base;

**public** **interface** Builder {

Product getProduct();

**void** buildPart1(**int** part1);

**void** buildPart2(String part2);

}

**617**

**ШАБЛОНЫ** **ПРОЕКТИРОВАНИЯ**

**package** by.bsu.builder.base;

**public** **class** ConcreteBuilder **implements** Builder {

**private** Product product = **new** Product();

**public** Product getProduct() {

**return** product;

}

**public** **void** buildPart1(**int** part1) { *// implementation*

product.setPart1(part1);

}

**public** **void** buildPart2(String part2) { *// implementation*

product.setPart2(part2);

}

}

**package** by.bsu.builder.base;

**public** **class** Director {

**private static** Builder builder; *// no need*

**public** Director (String builderMode) {

// init builder

}

**public** Product construct(String sourceName) { // data reading

String part1 = "part1";//read from sourse

String part2 = "part2";//read from sourse

builder.buildPart1(*part1*);

builder.buildPart2(*part2*);

**return** builder.getProduct();

}

}

**package** by.bsu.builder.base;

**public** **class** RunnerMain {

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

Director director = **new** Director("Concrete");

Product prod = director.construct("sourcePath");

}

}

Более эффектной выглядит реализация шаблона с использованием внутреннего класса:

package by.bsu.pattern.builder;  
import java.util.StringJoiner;  
public class Item {  
 private int itemId;  
 private String name;  
 private double price;  
 private Item() {  
 *// private constructor* }  
 public int getItemId() {  
 return itemId;  
 }  
 public String getName() {  
 return name;  
 }  
 public double getPrice() {  
 return price;  
 }  
 public static Builder newBuilder() {  
 return new Item().new Builder();  
 }  
 @Override  
 public String toString() {  
 return new StringJoiner(", ", Item.class.getSimpleName() + "[", "]")  
 .add("itemId=" + itemId)  
 .add("name='" + name + "'")  
 .add("price=" + price)  
 .toString();  
 }  
 public class Builder {  
 private Builder() {  
 }  
 public Builder setItemId(int itemId) {  
 Item.this.itemId = itemId;  
 return this;  
 }  
 public Builder setName(String name) {  
 Item.this.name = name;  
 return this;  
 }  
 public Builder setPrice(double price) {  
 Item.this.price = price;  
 return this;  
 }  
 public Item build() {  
 return Item.this;  
 }  
 }  
}

package by.bsu.pattern.builder;  
  
public class MainBuilder {  
 public static void main(String[] args) {  
 Item.Builder itemBuilder = Item.*newBuilder*()  
 .setItemId(777)  
 .setPrice(14.89)  
 .setName("T-Shirt");  
  
 Item item = itemBuilder.build();  
 System.*out*.println(item);  
  
 Item item2 = Item.*newBuilder*()  
 .setItemId(771)  
 .setPrice(14.80)  
 .setName("Gloves")  
 .build();  
 System.*out*.println(item2);  
 }  
}

Еще одним примером реализации шаблона **Builder** служит процесс создания объекта на основе информации, извлекаемой из XML-документа различными парсерами.

Класс **BaseBuilder** определяет абстрактный интерфейс для создания частей объекта класса **User**. Классы **DomBuilder,** **SaxBuilder** и **StaxBuilder** кон-струируют и собирают вместе части объекта класса **User**, а также представля-яют внешний интерфейс для доступа к нему. В результате объекты-строители могут работать с разными источниками, определяющими содержимое, не тре-буя при этом никаких изменений. При использовании шаблона появляется воз-можность контролировать пошагово весь процесс создания объекта-продукта.

**ПОРОЖДАЮЩИЕ** **ШАБЛОНЫ**

*/\** *#* *11* *#* *создаваемый* *объект* *объект* *#* *User.java* *\*/*

**package** by.bsu.builder;

**public** **class** User {

**private** String login;

**private** String lastname;

**public** **void** setLogin(String login) {

**this**.login = login;

}

**public** **void** setLastname (String lastname) {

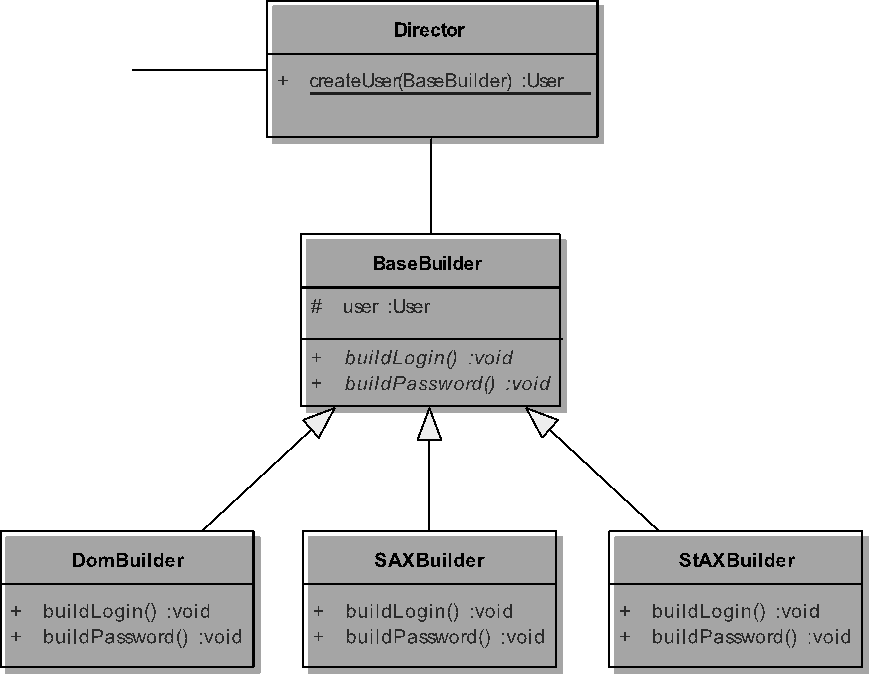
**this**.lastname = lastname;

}

}

Класс **BaseBuilder** — абстрактный класс-строитель, объявляющий в качестве поля ссылку на создаваемый объект и абстрактные методы его построения. Классы **DomBuilder,** **SaxBuilder** и **StaxBuilder** — наследуемые от него классы, реализующие специальные способы создания объекта.

**Рис.** **21.5.** *Пример* *реализации* *шаблона* *Builder*



**619**

**ШАБЛОНЫ** **ПРОЕКТИРОВАНИЯ**

**package** by.bsu.builder;

**public** **abstract** **class** BaseBuilder {

**protected** User user = **new** User();

**public** User getUser() {

**return** user;

}

**public** **abstract** **void** buildLogin();

**public** **abstract** **void** buildPassword();

*/\* public abstract void buildUser();as variant \*/*

}

**package** by.bsu.builder;

**public** **class** DomBuilder **extends** BaseBuilder {

**public** **void** buildLogin() {

}

**public** **void** buildPassword() {

}

}

**package** by.bsu.builder;

**public** **class** SaxBuilder **extends** BaseBuilder {

**public** **void** buildLogin() {

*// login reading*

}

**public** **void** buildPassword() {

*// pass reading*

}

}

**package** by.bsu.builder;

**public** **class** StaxBuilder **extends** BaseBuilder {

*// init StAX parser*

**public** **void** buildLogin() {

// *login reading*

}

**public** **void** buildPassword() {

// *pass reading*

}

}

Процесс создания объектов с использованием одного принципа реализован ниже.

*/\** *#* *13* *#* *тестирование* *процесса* *создания* *объекта* *#* *Director.java* *#* *Runner.java* *\*/*

**package** by.bsu.builder;

**public** **class** Director {

**public** **static** User createUser(BaseBuilder builder) {

builder.buildLogin();

builder.buildPassword();

**return** builder.getUser();

}

}

**package** by.bsu.builder;

**public** **class** RunnerMain {

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

User e1 = Director.createUser(**new** DomBuilder());

User e2 = Director.createUser(**new** SAXBuilder());

}

}

Особенности применения шаблона **Builder**:

— осуществление широкого управления процессами создания объекта;

— процедура создания объекта независима от его внутренней организации.



**Шаблон** **Singleton**

Необходимо создать объект класса таким образом, чтобы гарантировать не-возможность инициализации другого объекта того же класса. Обычно сам класс контролирует наличие единственного экземпляра и он же предоставляет при необходимости к нему доступ. Решение должно подходить для многопо-точных приложений при условии отсутствия опасности возникновения исклю-чительных ситуаций в конструкторе.

В настоящее время существует реализация шаблона, инициализация экзем-пляра которого нетребует защиты при использовании вмногопоточном режиме.

*/\** *#* *14* *#* *реализация* *шаблона* *«Одиночка» c помощью enum* *#* *SingletonType.java* *\*/*

**public** **enum** SingletonType {

*INSTANCE*

*// constructors, fields, methods*

}

**public** **enum** SingletonMultiType {

*INSTANCE\_ZERO, INSTANCE\_ONE, INSTANCE\_TWO*

*// constructors, fields, methods*

}

Любая реализация класса **Enum** гарантирует наличие только одного экзем-пляра каждого из своих элементов. При реализации простого шаблона **Singleton** у перечисления определяется только один элемент.

Возможность возникновения исключения в конструкторе при создании единственного экземпляра налагает определенную ответственность на про-цесс инициализации экземпляра. Повторный процесс инициализации должен быть осуществим в реализации класса-одиночки.

*/\** *#* *15* *#* *реализация* *шаблона* *«Одиночка»* *#* *SingletonType.java* *\*/*

**package** by.bsu.singleton;

**public** **class** SingletonType {

**private static** **final** SingletonType ***instance*** = **new** SingletonType();

*// other fields*

**private** SingletonType() {

}

**public** **static** SingletonType getInstance() {

**return** ***instance***;

}

**public** **static void** operationStatic() {

}

**public** **void** operation() {

}

}

Класс объявляет статический метод **getInstance()**, который инициализирует экземпляр класса при первом вызове, а в последствии позволяет клиентам по-лучать контролируемый доступ к единственному экземпляру. Такая реализа-ция не рекомендует объявлять в классе другие статические методы, так как при вызове любого из них также произойдет инициализация поля **instance**.

Если разрешить классу иметь подклассы, то такой шаблон позволит уточнять методы через подклассы, а также разрешить появление более чем одного эк-земпляра. На практике такой подход используется редко.

Объявление статического внутреннего класса, статическое поле которого пред-ставляет экземпляр-одиночку, решает проблему сторонних статических методов класса-владельца иобеспечивает ленивую инициализацию. Предложено Биллом Пью (Bill Pugh).

*/\** *#* *16* *#* *решение* *Билла* *Пью* *для* *шаблона* *«Одиночка»* *#* *LazySingleton.java* *\*/*

**package** by.bsu.singleton;

**public** **class** LazySingleton {

*// non-static fields*

**private** LazySingleton() {

}

**private** **static** **class** SingletonHolder { *// nested class*

**private** **static final** LazySingleton ***instance*** = **new** LazySingleton();

} *// end of nested class*

**public** **static** LazySingleton getInstance() {

**return** SingletonHolder.***instance***;

}

**public** **static void** operationStatic() {

}

**public** **void** operationNonStatic() {

}

}

Единственный экземпляр инициализируется при первом вызове метода **getInstance()**. Проблема с обработкой исключительных ситуаций в конструкторе по прежнему не решена. Так что, если конструктор класса не вызывает опасений создания исключительных ситуаций, то можно использовать этот подход.

Для обеспечения синхронизации без инициализации экземпляра в статиче-ском поле можно использовать класс **ReentrantLock**:

*/\** *#* *17* *#* *Lock* *для* *шаблона* *«Одиночка»* *#* *LockImpl.java* *\*/*

**package** by.bsu.singleton;

**import** java.util.concurrent.locks.ReentrantLock;

**public** **class** LockSingleton {

**private** **static** LockSingleton *instance*;

**private** **static** ReentrantLock *lock* = **new** ReentrantLock();

**private** **static** AtomicBoolean isCreate = **new** AtomicBoolean(false);// boolean

*// private non-static fields*

**private** LockSingleton () {

}

**public** **static** LockSingleton *getInstance*() {

**if**(!isCreate.get()) {

*lock*.lock(); *// blocking*

**try** {

**if** (*instance* == **null**) {

*instance* = **new** LockSingleton();

isCreate.set(**true**);

}

} **finally** {

*lock*.unlock(); *// unblocking*

}

}

**return** *instance*;

}

*// other non-static methods*

}

Данное решение несколько более производительно, чем при **synchronized**. В случае, если шаблон **Singleton** подразумевает ограничение на количество

ссылок больше одной, то удобно использовать инициализацию через **Semaphore**.

*/\** *#* *18* *#* *Semaphore* *для* *шаблона* *«Одиночка»* *#* *SemaphoreImpl.java* *\*/*

**package** by.bsu.singleton;

**import** java.util.ArrayList;

**import** java.util.concurrent.Semaphore;

**public** **class** SemaphoreImpl { //Single

**private** **static** **final** **int** ***MAX\_AVAILABLE*** = 10; *// лимит экземпляров списка*

**private** **static** Semaphore *semaphore* = **new** Semaphore(***MAX\_AVAILABLE***, **true**);

**private** **static** ArrayList<SemaphoreImpl> *instances* =

**new** ArrayList<SemaphoreImpl>(***MAX\_AVAILABLE***);

**private** SemaphoreImpl() {

}

**public** **static** SemaphoreImpl getInstance(**int** index) **throws** SingletonException {

**if** (index >= 0 && index < *instances*.size()) {

*// доступ к элементу списка*

**return** instances.get(index);

}

**if** (index < *instances*.size() && *semaphore*.tryAcquire()) {

*// уменьшение значение семафора на 1*

SemaphoreImpl tmp = **new** SemaphoreImpl();

*instances*.add(tmp);

**return** tmp;

}

**throw** **new** SingletonException("Превышен лимит на число экземпляров");

}

}

*/\** *#* *19* *#* *класс-исключение* *#* *SingletonException.java* *\*/*

**package** by.bsu.singleton;

**public** **class** SingletonException **extends** Exception {

**public** SingletonException() {

}

**public** SingletonException(String message) {

**super**(message);

}

}

При первом вызове **getInstance()** с любым значением список еще пуст, в первом блоке **if** условие ложное. Второй блок **if** уменьшает значение семафора с 10 до 9 и создает 0, добавляет в список экземпляр **SemaphoreImpl**. В списке теперь присутствует один экземпляр, и он доступен при вызове **getInstance()** с параметром, равным 0. При следующих вызовах, например, с инкрементным увеличением индекса семафор уменьшает свое значение, а список пополняется недостающими экземплярами.

При попытке получить доступ к объекту с индексом вне допустимых значений списка при обнуленном значении семафора будет генерироваться исключение. Приведенное решение — самое медленное, однако его преимущество в настраиваемом количестве ссылок.

При создании единственного экземпляра в режиме многопоточности следует гарантировать невозможность получить недо конца сконструированный объект и при этом не потерять в производительности из-за постоянного контроля ссылки логов синхронизации. В одном из допустимых решений этой проблемы прибегают к **volatile** переменной.

*/\** *#* *20* *#* *volatile* *для* *шаблона* *«Одиночка»* *#* *VolatileImpl.java* *\*/*

**package** by.bsu.singleton;

**public** **class** VolatileImpl {

**private** **static** VolatileImpl *instance*;

**private** **volatile** **static** **boolean** *instanceCreated*;

**private** VolatileImpl() {

}

**public** **static** VolatileImpl getInstance() {

**if** (!instanceCreated) {

**synchronized** (VolatileImpl.**class**) {

**if** (!instanceCreated) {

instance = **new** VolatileImpl();

instanceCreated = **true**;

}

}

}

**return** instance;

}

Модификатор **volatile** объявляет, что по адресу ссылки на объект будет хра-ниться адрес переменной в памяти кэша процессора, исключающий возмож-ность получения по ссылке устаревших данных в многопоточном приложении.

При необходимости запретить клонирование объекта-синглтона метод его выполняющий примет вид:

**@Override**

**public final** Object clone() **throws** CloneNotSupportedException {

**throw new** CloneNotSupportedException();

}

Для защиты от создания объекта-синглтона с помощью reflection в конструктор класса-синглтона необходимо добавить:

**if (instance != null)** {

**throw new** RuntimeException(“attempt ”);

}

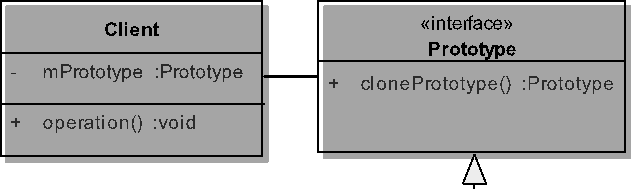
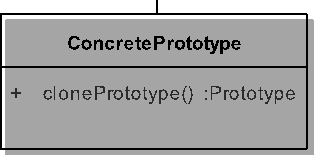
Reflection API при создании объекта неявным образом использует конструктор.

public class ConnectionPool {  
   
 private static ConnectionPool *instance*;  
 private static CountDownLatch *latch* = new CountDownLatch(1);  
 private static AtomicBoolean *isInitialized* = new AtomicBoolean(false);  
  
 private ConnectionPool() {  
 }  
  
 public static ConnectionPool getInstance() {  
 if (*instance* == null) {  
 if (*isInitialized*.compareAndSet(false, true)) {  
 *instance* = new ConnectionPool();  
 *latch*.countDown();  
 }  
 try {  
 *latch*.await();  
 } catch (InterruptedException e) {  
 *// logger* }  
 }  
 return *instance*;  
 }  
}

**Шаблон** **Prototype**

Некоторые шаблоны реализуются и используются в самом языке. Процесс клонирования объектов фактически реализует шаблон **Prototype**. Процесс по-лучения копии объекта всегда проще, чем создание нового. Получение данных для создания новых объектов к тому же может быть затруднено или вообще невозможно.

Пусть имеется эталонный набор объектов, представленный ввиде коллекции. Объекты важны и могут понадобиться владельцу в любой момент времени.

**Рис.** **21.6.** *Базовая* *реализация* *шаблона* *Prototype*

Но эти же объекты могут быть необходимы кому-либо еще. Владелец готов предоставить объекты, причем желательно так, чтобы не нужно было беспокоиться об их целостности. Для обеспечеия безопасности экземпляров в коллекции владельцу объектов достаточно предоставить всем желающим точную копию объекта путем клонирования.

Шаблон **Prototype** позволяет специализировать механизм создания прото-типов для заданной абстракции предметной области. Особенно это может быть полезно при применении глубокого клонирования.

Базовая схема шаблона представлена на следующей диаграмме.

*/\** *#* *21* *#* *определение* *интерфейса* *для* *классов,* *поддерживающих* *прототипирование* *#* *Prototype.java* *\*/*

**package** by.bsu.prototype.base;

**public** **interface** Prototype {

*// methods*

Prototype clonePrototype();

}

*/\** *#* *22* *#* *реализация* *класса* *с* *созданием* *копии* *своего* *экземпляра* *#ConcretePrototype.java* *\*/*

**package** by.bsu.prototype.base;

**public** **class** ConcretePrototype **implements** Prototype {

*// поля и полиморфные методы*

**public** Prototype clonePrototype() {

*// реализация процесса создания*

*// объекта-прототипа (клона)*

**return** object;

}

}

*/\** *#* *23* *#* *класс,* *создающий* *и* *использующий* *копию* *экземпляра* *#* *Client.java* *\*/*

**package** by.bsu.prototype.base;

**public** **class** Client {

**private** Prototype prototype;

**public** **void** operation() {

Prototype instance = prototype.clonePrototype();

*// действия с клоном*

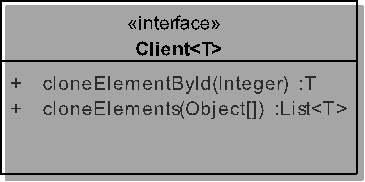
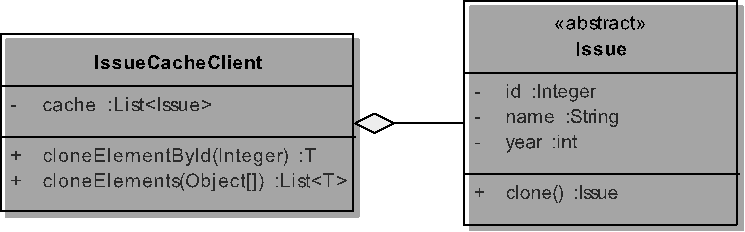
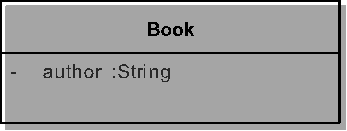
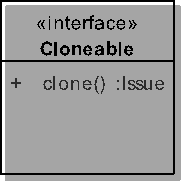
}

}

Для списка книг и журналов необходимо организовать возможность поиска и клонирования экземпляра по идентификационному номеру, а также клониро-вание целого списка. Необходимые методы определены в интерфейсе **Client**.



**Рис.** **2.7.** *Реализация* *Prototype* *для* *клонирования* *элемента* *коллекции*



*/\** *#* *24* *#* *интерфейс* *для* *классов* *для* *работы* *с* *прототипами* *#* *Client.java* *\*/*

**package** by.bsu.protoype.generic;

**import** java.util.List;

**public** **interface** Client<T> {// <- Prototype

T cloneElementById(Integer id);// E

List<T> cloneElements(Object... param); // E

*// другие методы*

}

Класс **IssueCacheClient** представляет конкретные реализации процессов кло-нирования.

 */\** *#* *25* *#* *класс,* *создающий* *и* *использующий* *копию* *экземпляра* *#* *IssueCacheClient.java* *\*/*

**package** by.bsu.protoype.generic;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** IssueCacheClient **implements** Client<Issue> {

*// Prototype*

**private** List<Issue> cache;

**public** IssueCachePrototype() {

cache = **new** ArrayList<Issue>();

}

**public** IssueCachePrototype(List<Issue> issueList) {

**this**.cache = issueList;

}

@Override

**public** Issue cloneElementById(Integer id) {

**for** (Issue issue : cache) {

**if** (issue.getId().equals(id)) {

**return** issue.clone();

}

}

**throw** **new** IllegalArgumentException("illegal ID " + id);

}

@Override

**public** List<Issue> cloneElements(Object... param) {

List<Issue> list = **new** ArrayList<Issue>();

*/\* реализация поиска, клонирования и организации*

*новой коллекции \*/*

**return** list;

}

}

Абстрактный класс **Issue** определяет процесс клонирования для всех своих подклассов, если только они в качестве полей используют неизменяемые типы. В случае использования изменяемых классов в качестве поля подкласса класса **Issue**, метод **clone()** требуется переопределять для этого типа. Во все классы примера # 26 добавить методы **get-**, **set-**, **toString()**.

*/\** *#* *26* *#* *иерархия* *классов,* *поддерживающих* *клонирование* *#* *Issue.java* *#* *Book.java* *#* *Magazine.java* *\*/*

**package** by.bsu.protoype.generic;

**public** **abstract** **class** Issue **implements** Cloneable {

**private** Integer id;

**private** String name;

**private** **int** year;

**public** Issue(Integer id, String name, **int** year) {

**this**.id = id;

**this**.name = name;

**this**.year = year;

}



@Override

**protected** Issue clone() {

Issue cloned = **null**;

**try** {

cloned = (Issue)**super**.clone();

} **catch** (CloneNotSupportedException e) {

e.printStackTrace();

}

**return** cloned;

}

}

**package** by.bsu.protoype.generic;

**public** **class** Book **extends** Issue {

**private** String author;

**public** Book(Integer id, String author, String name, **int** year) {

**super**(id, name, year);

**this**.author = author;

}

}

**package** by.bsu.protoype.generic;

**public** **class** Magazine **extends** Issue {

**private** **int** number;

**public** Magazine(Integer id, **int** number, String name, **int** year) {

**super**(id, name, year);

**this**.number = number;

}

}

*/\** *#* *27* *#* *запуск* *процесса* *создания* *прототипов* *#* *ProRunner.java* *\*/*

**package** by.bsu.protoype.generic;

**import** java.util.ArrayList;

**public** **class** ProRunner {

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

ArrayList<Issue> issueList = **new** ArrayList<Issue>() {

**this**.add(**new** Book(615, "Steve McConnell", "Code Complete", 2012));

**this**.add(**new** Book(453, "Bruce Eckel", "Thinking in Java", 2006));

**this**.add(**new** Book(721, "Joshua Bloch", "Effective Java", 2008));

**this**.add(**new** Magazine(1009, 9, "PC Magazine", 2024));

}

};

IssueCacheClient cache = **new** IssueCacheClient(issueList);

Issue copy = cache.cloneElementById(453);

System.out.println(issueList);

System.out.println(copy);

}

}

Применение шаблона с использованием стандартных возможностей языка фактически представляет декомпозицию стандартного процесса клонирова-ния, рассмотренную при изучении методов класса **Object**.

***Задания*** ***к*** ***главе*** ***2***

В любой задаче обязательно создание иерархии классов сущностей создава-емой системы.

1. Паттерн Builder. Разработать модель системы Автомобиль. Написать код приложения, который будет позволять порождать как серийные автомоби-ли, так и автомобили по специальному заказу. Использовать шаблон.

2. Паттерн Builder. Разработать модель системы Музыкальный коллектив. Написать код приложения, позволяющий создавать певческие, танцеваль-ные и смешанные коллективы.

3. Паттерн Builder. Разработать модель системы Комплексный обед. Написать код приложения, позволяющий создавать как стандартные комплексные обеды, так и обеды, в которые включены дополнительные блюда из меню.

4. Паттерн Builder. Имеется текст статьи вформате TXT. Статья состоит иззаго-ловка, фамилий авторов, самого текста статьи ихеш-кода текста статьи. Написать приложение, позволяющее конвертировать документ вформате TXTвдокумент формата XML, необходимо также проверять корректность хэш-кода статьи.

5. Паттерн Abstract Factory. Написать код приложения, позволяющий универсаль-но записывать данные осовершенном телефонном звонке вбазу данных иxml-файл, атакже считывать эту информацию. Использовать также шаблон DAO.

6. Паттерн Abstract Factory. Написать код приложения, позволяющий сохра-нять регистрационные данные пользователя в базе данных. Состав реги-страционных данных у каждого пользователя может быть различен. Использовать также шаблон DAO.

7. Паттерн Abstract Factory. Разработать систему Кинопрокат. Пользователь мо-жет выбрать определенную киноленту, при заказе киноленты указывается язык звуковой дорожки, который совпадает с языком файла субтитров. Система должна поставлять фильм стребуемыми характеристиками, причем при смене языка звуковой дорожки должен меняться иязык файла субтитров инаоборот.

8. Паттерн Prototype. Существует набор статей в википедии. Реализовать про-цесс раздачи статей по требованию для изменения, сохраняя исходный ва-риант для возможного восстановления статьи в исходном виде.

9. Паттерн Factory Method. Фигуры игры «тетрис». Реализовать процесс слу-чайного выбора фигуры из конечного набора фигур. Предусмотреть появ-ление супер-фигур с большим числом клеток, чем обычные.