Глава 23

СТРУКТУРНЫЕ ШАБЛОНЫ

Структурные шаблоны GoF отвечают за композицию объектов и классов, и не только за объединение частей приложения, но и за их разделение.

К структурным шаблонам относятся:

Adapter (**Адаптер**) — применяется при необходимости использовать классы вместе с несвязанными интерфейсами. Поведение адаптируемого класса при этом изменяется на необходимое (interface to an object);

Bridge (**Мост**) — разделяет представление класса и его реализацию, позволяя независимо изменять то и другое (implementation of an object);

Composite (**Компоновщик**) — группирует объекты в иерархические древовидные структуры и позволяет работать с единичным объектом так же, как с группой объектов (structure and composition of an object);

Decorator (**Декоратор**) — представляет способ изменения поведения объекта без создания подклассов. Позволяет использовать поведение одного объекта в другом (responsibilities of an object without subclassing);

Facade (**Фасад**) — создает класс с общим интерфейсом высокого уровня к некоторому числу интерфейсов в подсистеме (interface to a subsystem);

Flyweight (Легковес) — разделяет свойства класса для оптимальной поддержки большого числа мелких объектов (storage costs of objects);

Proxy (**Заместитель**) — подменяет сложный объект аналогичным и осуществляет *контроль* доступа к нему (how an object is accessed... its location).

Шаблон Bridge

Необходимо отделить абстракцию (Abstraction) от ее реализации (Implementor) так, чтобы и то, и другое можно было изменять независимо. Шаблон **Bridge** используется в тех случаях, когда может быть выделена иерархия абстракций и независимая иерархия реализаций. Точное соответствие между абстракциями и реализациями в общем случае невозможно. Обычно абстракция определяет операции более высокого уровня, чем реализация.

Шаблон позволяет снизить общее число классов за счет того, что несколько абстракций могут использовать одну реализацию, в простейшем случае определяя ее как ссылку на интерфейс в иерархии абстракций.

Реализация во время выполнения при необходимости может быть изменена.

Каркас базовой реализации шаблона представлен в виде:

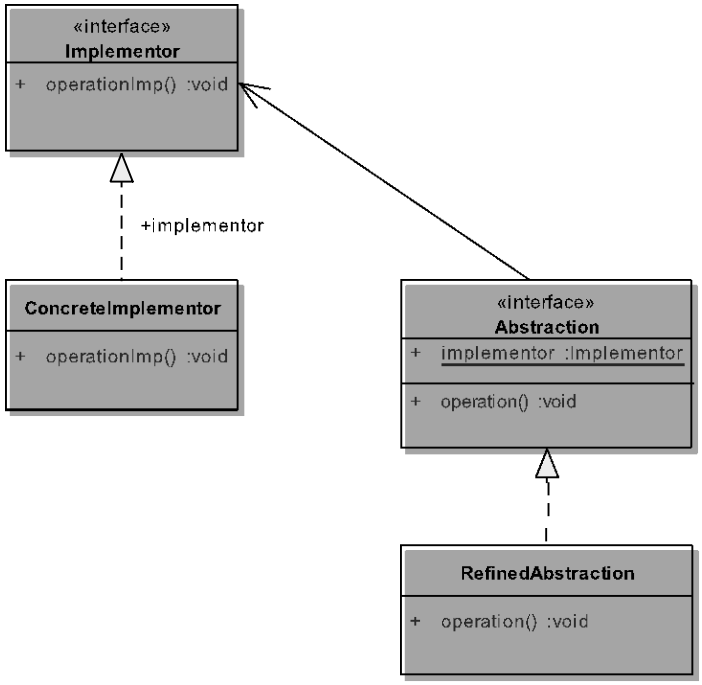

Рис. 23.1. *Базовая реализация шаблона Bridge*

```
/* # 1 # Implementors # Implementor.java # ConcreteImplementor.java */
package by.bsu.bridge.base;
public interface Implementor {
    void operationImp();
}
package by.bsu.bridge.base;
public class ConcreteImplementor implements Implementor {
    public void operationImp() { // more code
    }
}
/* # 2 # Abstractions # Abstraction.java # RefinedAbstraction.java */

package by.bsu.bridge.base;
public interface Abstraction {
    public static Implementor implementor;
    public void operation();
}
package by.bsu.bridge.base;
public class RefinedAbstraction implements Abstraction {
    public void operation() {
       // more code
    }
}
```

Элементы шаблона:

1. **Abstraction** — собственно абстракция. Определяет базовый интерфейс абстракции и агрегирует объект типа **Implementor**;
2. **RefinedAbstraction** — уточненный элемент абстракции. Подкласс основной абстракции, реализующий интерфейс ею определенный;
3. **Implementor** — исполнитель. Определяет интерфейс для классов реализации. Обычно интерфейс **Implementor** содержит только элементарные операции, а тип **Abstraction** определяет операции более высокого уровня или композиции элементарных операций;
4. **ConcreteImplementor** — конкретный исполнитель, содержащий конкретную реализацию интерфейса, определяемого типом **Implementor**.

Пусть существует некое банковское учреждение, совершающее действия по кредитным, депозитным и прочим счетам. Экземпляры счетов-**Abstraction** выполняют действия. Действия могут быть обычными, срочными и другими. Срочное обслуживание стоит клиенту дороже, так как повышаются ежемесячные платежи, теряются проценты и проч. Видов счетов в реальной банковской сфере существует достаточно много, как и действий, производимых над ними.

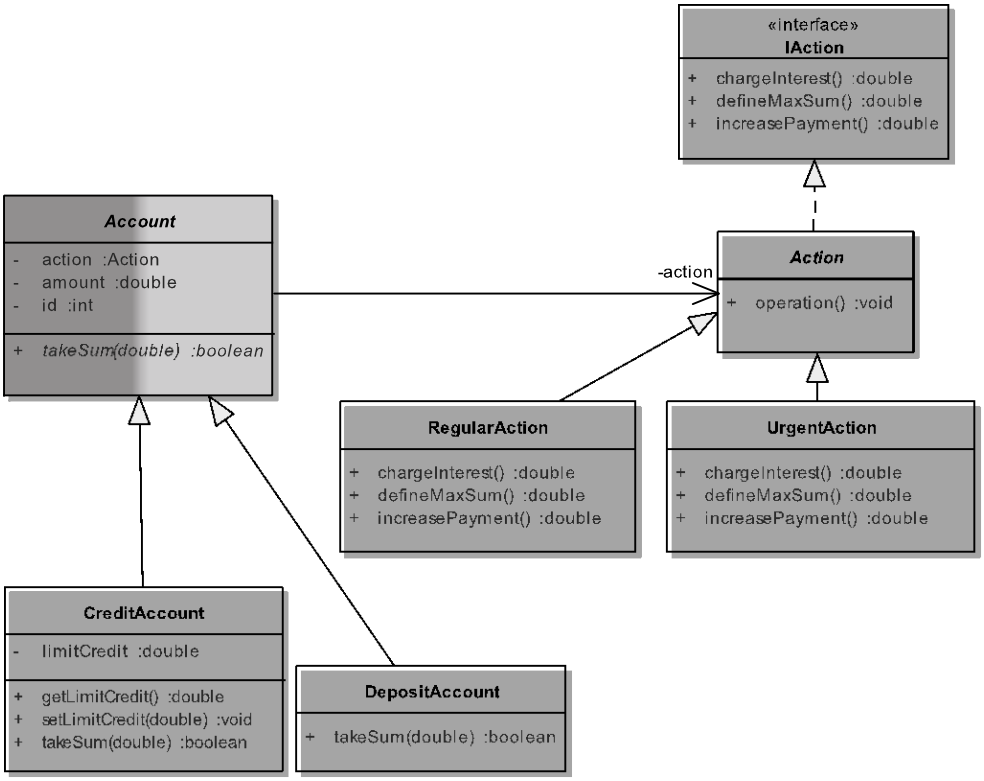

Рис. 23.2. *Реализация шаблона Bridge*

Если каждому типу счета ставить в соответствие свою реализацию действия, то общее число классов, требуемых к созданию, будет равно произведению чи-сла типов счетов на число возможных действий. Добавление же одного нового типа счета, потребует создания классов в количестве, равном числу действий.

Классы **Action-Implementor** отделены от **Account-Abstraction**, причем так, чтобы реализации ничего не знали в идеале об абстракциях.

```
/* # 3 # Implementors # IAction.java # Action.java # RegularAction.java
# UrgentAction.java */

package by.bsu.bridge.bank;
public interface IAction {
    double chargeInterest();
    double defineMaxSum();
    double increasePayment();
}
package by.bsu.bridge.bank;
public abstract class Action implements IAction {
      // поля и методы, общие для всех реализаций
      public void operation() {
      // more code
    }
}
package by.bsu.bridge.bank;
public class RegularAction extends Action {
    private final static int MAX_SUM = 100; // read from base
    private final static int NORMAL_INTEREST = 3; // read from base
    @Override
    public double chargeInterest() {
        // charge NORMAL interest on account"
        return NORMAL_INTEREST;
    }
    @Override
    public double defineMaxSum() {
        // max sum is unbounded"
        return MAX_SUM;
    }
    @Override
    public double increasePayment() {
        return 0; // stub
    }
}
package by.bsu.bridge.bank;
public class UrgentAction extends Action {
    final static int MONTHLY_PAYMENT = 10; // read from base
```

```
    private final static int MAX_SUM = 50; // read from base
    @Override
    public double chargeInterest() {
        // charge LOW interest on accounts
        return 0; // stub
    }
        @Override
        public double defineMaxSum () { // check credit
            // max sum is bounded"
            return MAX_SUM;
        }
        @Override
        public double increasePayment () {
            // MAX increase in monthly payments
            return MONTHLY_PAYMENT;
        }
    }
```

Класс **Action** — абстрактный, реализующий **Implementor** (**IAction**). Может содержать общие методы для всех действий, например: проверку прав пользователя, блокировку счета и т. д. Классы **RegularAction** и **UrgentAction** уточняют подклассы класса **Action**.

/ # 4 # абстракция и ее уточнения # Account.java # DepositAccount.java # CreditAccount.java */*

```
package by.bsu.bridge.bank;
public abstract class Account {
    private int id;
    private double amount;
    private Action action;
    protected Account(Action action) {
        this.action = action;
    }
    public Action getAction() {
        return action;
    }
    protected void setAction(Action action) {
        this.action = action;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public double getAmount() {
        return amount;
```

```
    }
    public void setAmount(double amount) {
        this.amount = amount;
    }
    public abstract boolean takeSum(double sum);
}
package by.bsu.bridge.bank;
public class DepositAccount extends Account {
    public DepositAccount(Action action) {
        super(action);
    }
    @Override
    public boolean takeSum(double sum) {
        System.out.println("Performing by deposit account:");
        double interest = getAction().chargeInterest();
        double maxSum = getAction().defineMaxSum();
        // check amount
        System.out.print("accountID: " + getId()
                                    + " : interest is " interest);
        System.out.print(" : recording of changes in the state   "
                                                + "accounts");
        System.out.println(": withdrawal : " + sum);
        return true;
    }
}
package by.bsu.bridge.bank;
public class CreditAccount extends Account {
    private double limitCredit;
    public double getLimitCredit() {
        return limitCredit;
    }
    public void setLimitCredit(double limitCredit) {
        this.limitCredit = limitCredit;
    }
    public CreditAccount(Action action) {
        super(action);
    }
    public boolean takeSum(double sum) {
        System.out.println("Performing by credit account:");
        double maxSum = getAction().defineMaxSum();
        double payment = getAction().increasePayment();
        System.out.print("accountID: " + getId()
                    + " increase monthly payments: " + payment);
        System.out.print(": recording of changes in the state "
                                                + "accounts");
        System.out.println(" : withdrawal :" + sum);
        return true;
    }
}
```

Класс **Account** — абстракция, классы **DepositAccount** и **CreditAccount** — уточненные абстракции.

/ # 5 # использование шаблона Bridge # BridgeClient.java */*

```
package by.bsu.bridge.bank;
public class BridgeClient {
    public static void main(String[] args) {
        Action action = new RegularAction();
        DepositAccount depositAccount = new DepositAccount(action);
        depositAccount.setId(777);
        depositAccount.setAmount(1500);
        depositAccount.takeSum(200);
        action = new UrgentAction();
        depositAccount.setAction(action); // replacement action
        depositAccount.takeSum(100);
        new CreditAccount(action).takeSum(50);
    }
}
```

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 могут иметь любое число подклассов.

Ниже приведен более простой пример использования шаблона **Bridge** при создании собственного логгера. Реализация может быть применена и для разработки с собственным псевдо-логгером.

/ # 6 # Implementors # LoggerImplementor.java # MultiThreadedLogger.java # SingleThreadedLogger.java */*

```
package by.bsu.bridge.logger;
public interface LoggerImplementor {
    void logToConsole();
    void logToFile();
    void logToSocket();
}
package by.bsu.bridge.logger;
public class MultiThreadedLogger implements LoggerImplementor {
    @Override
    public void logToConsole() {
        System.out.println("Multithreaded console log");
    }
    @Override
    public void logToFile() {
```

```
        System.out.println("Multithreaded file log");
    }
@Override
public void logToSocket() {
        System.out.println("Multithreaded socket log");
    }
}
package by.bsu.bridge.logger;
public class SingleThreadedLogger implements LoggerImplementor {
    @Override
    public void logToConsole() {
        System.out.println("Singlethreaded console log");
    }
    @Override
    public void logToFile() {
        System.out.println("Singlethreaded file log");
    }
    @Override
    public void logToSocket() {
        System.out.println("Singlethreaded socket log");
    }
  }
```

/ # 7 # абстракция и ее уточнения # Logger.java # ConsoleLogger.java # FileLogger.java*

```
package by.bsu.bridge.logger;
public abstract class Logger {
    protected LoggerImplementor logger;
    public Logger() {
    }
    public Logger(LoggerImplementor logger) {
        this.logger = logger;
    }
    public void setLogger(LoggerImplementor logger) {
        this.logger = logger;
    }
    public abstract void log();
}
package by.bsu.bridge.logger;
public class ConsoleLogger extends Logger {
    public ConsoleLogger() {
}
    public ConsoleLogger(LoggerImplementor logger) {
        super(logger);
    }
    public void log() {
```

```
        logger.logToConsole();
    }

}
package by.bsu.bridge.logger;
public class FileLogger extends Logger {
    public FileLogger() {
    }
    public FileLogger(LoggerImplementor logger) {
        super(logger);
    }
    public void log() {
        logger.logToFile();
    }
}
```

/ # 8 # использование шаблона Bridge # BridgeClientMain.java */*

```
package by.bsu.bridge.logger;
public class BridgeClientMain {
    public static void main(String[] args) {
        LoggerImplementor loggerImpl = new SingleThreadedLogger();
        Logger logger = new ConsoleLogger(loggerImpl);
        logger.log();
        loggerImpl = new MultiThreadedLogger();
        logger.setLogger(loggerImpl);
        logger.log();
        new FileLogger(loggerImpl).log();
    }
}
```

Шаблон Decorator

Расширяет функциональные возможности объекта, изменяя его поведение. Расширяющий класс реализует тот же самый интерфейс, что и исходный класс, делегируя исходному классу выполнение базовых операций. Может добавлять собственные операции. Представляет собой альтернативу множественному наследованию, то есть может добавлять функциональность классу, от которого нельзя наследоваться. Добавляемая функциональность может быть легко исключена при переработке кода. Шаблон **Decorator** позволяет динамически изменять поведение экземпляров в процессе выполнения приложения.

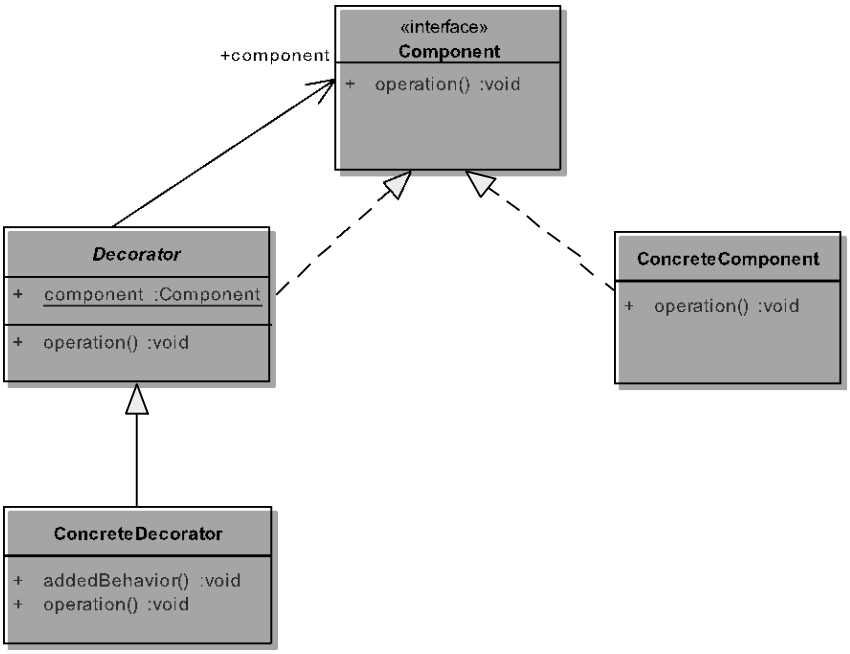

Рис. 23.3. *Базовая реализация шаблона Decorator*

Каркас базовой реализации шаблона представлен в виде:

```
/* # 9 # декорируемые типы # Component.java # ConcreteComponent.java */
package by.bsu.decorator.base;
public interface Component {
    void operation();
}
package by.bsu.decorator.base;
public class ConcreteComponent implements Component {
    public void operation() { // more code
    }
}
```

```
/* # 10 # абстракция декоратора и конкретный декорируемый тип # Decorator.java
# ConcreteDecorator.java */
package by.bsu.decorator.base;
public abstract class Decorator implements Component {
    public static Component component;
    public void operation() {
    // more code
    }
package by.bsu.decorator.base;
public class ConcreteDecorator extends Decorator {
```

```
    public void addedBehavior() {
    // more code
        }
    public void operation() {
    // использует реализацию типа Component и метод addBehavior()
    }
}
```

Элементы шаблона:

1) **Component** — определяет базовый интерфейс декорируемого типа;
2) **ConcreteComponent** — декорируемый тип с реализацией базовых операций. Таких классов может быть несколько;
3) **Decorator** — агрегирует декорируемый тип **Component** и наследует его реализацию. Определяет интерфейс для подклассов декораторов;
4) **ConcreteDecorator** — конкретный декоратор, содержащий конкретную реализацию интерфейса, определяемого типом **Decorator**, может объявлять дополнительное поведение. Может использовать как агрегированный тип, так и переопределять унаследованный интерфейс.

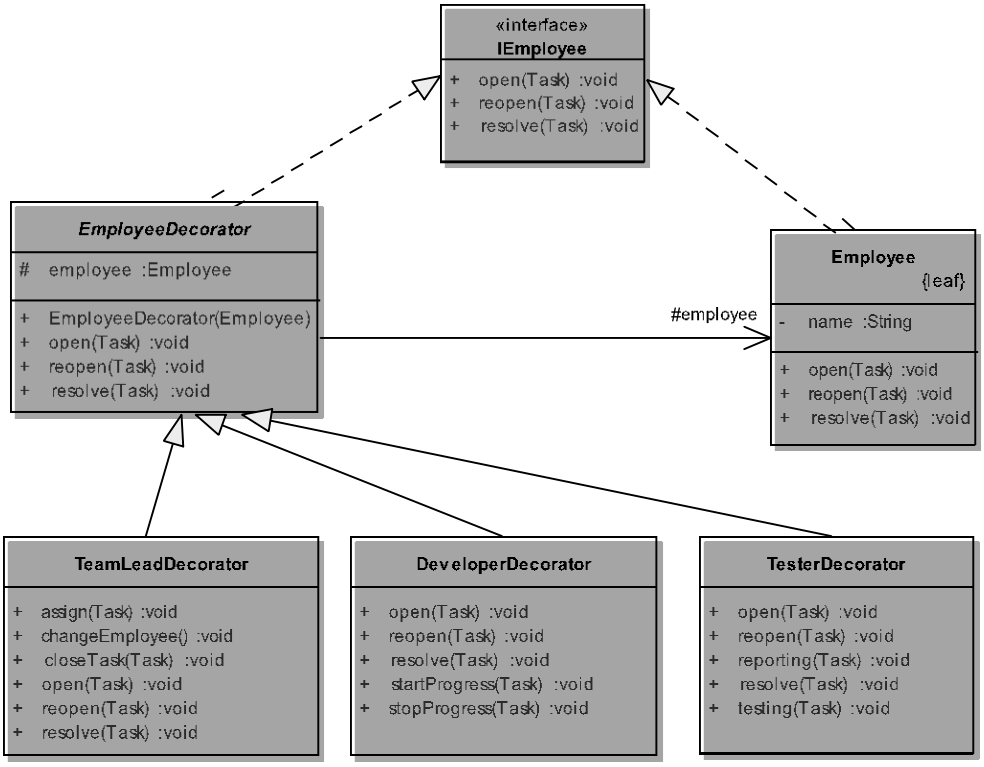

Рис. 23.4. *Реализация шаблона Decorator*

Позволяет уменьшить число подклассов по сравнению с аналогичным решением без использования декоратора. Однако при большом количестве классов-декораторов решение резко утяжеляется, становится малопонятным и утрачивает все преимущества от коротких применений шаблона.

В качестве примера можно рассмотреть систему управления заданиями в IT-проекте. Система предназначена для визуализации процесса выполнения отдельных частей проекта участниками с различными профессиональными навыками. Проект выполняется сотрудниками, которые могут выполнять общие для всех действия по открытию задания для выполнения, выставлению пометки о выполнении, «переоткрытию» задания в случае, например, нахождения ошибки тестировщиком. Каждое из этих действий может иметь дополнительные особенности, зависящие от роли сотрудника в проекте. Использование шаблона **Decorator** позволяет учесть эти особенности без построения дополнительной иерархии сотрудников.

/ # 11 # определение интерфейса для общих действий # IEmployee.java */*

```
package by.bsu.decorator;
public interface IEmployee {
    // может быть представлен абстрактным классом
    void openTask();
    void reopenTask();
    void resolveTask();
}
```

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

/ # 12 # класс-декоратор для интерфейса IEmployee # EmployeeDecorator.java */*

```
package by.bsu.decorator;
public abstract class EmployeeDecorator implements IEmployee {
    protected Employee employee;
    public EmployeeDecorator() {
        super();
    }
    public EmployeeDecorator(Employee employee) {
        this.employee = employee;
    }
    @Override
    public void resolveTask() {
        employee.resolveTask();
    }
    @Override
    public void openTask() {
```

```
        employee.openTask();
    }
    @Override
    public void reopenTask() {
        employee.reopenTask();
    }
}
```

Класс **Employee** определяет класс, функциональность которого будет расширена за счет применения декоратора. Сам класс в общем случае может даже запрещать наследование, то есть быть объявленным как **final**.

/ # 13 # декорируемый класс, наследования которого нежелательны # Employee.java */*

```
package by.bsu.decorator;
public class Employee implements IEmployee {
    private String name;
    public Employee() {
    }
    public Employee(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    @Override
    public void openTask() {
        System.out.println(this.getName() + " open task");
    }
    @Override
    public void reopenTask() {
        System.out.println(this.getName() + " reopen task");
    }
    @Override
    public void resolveTask() {
        System.out.println(this.getName() + " resolve task");
    }
}
```

Класс **DeveloperDecorator** объявляет дополнительные функциональности **startProgress()** и **endProgress()**, необходимые для разработчика, дополняя (декорируя) функциональности **openTask()**, **reopenTask()**, **resolveTask()** класса **Employee**.

/ # 14 # класс-декоратор, уточняющий базовую функциональность сотрудника # DeveloperDecorator.java */*

```
package by.bsu.decorator;
public class DeveloperDecorator extends EmployeeDecorator {
    // поля, методы
```

```
    public DeveloperDecorator(Employee employee) {
        super(employee);
    }
    @Override
    public void openTask() {
        super.openTask();
        startProgress();
    }
    @Override
    public void reopenTask() {
        super.reopenTask();
        startProgress();
    }
    @Override
    public void resolveTask() {
        super.resolveTask();
        stopProgress();
    }
    public void startProgress() {
        System.out.println(employee.getName() + " starting task");
    }
    public void stopProgress() {
        System.out.println(employee.getName() + " stopping task");
    }
}
```

Классы **TesterDecorator** и **TeamLeadDecorator** каждый в свою очередь добавляют функциональность, свойственную его деятельности, но никак не меняющую функциональность основного класса **Employee**.

/ # 15 # классы-декоратор сотрудника тестировщика и team-лидера # TesterDecorator.java # TeamLeadDecorator.java */*

```
package by.bsu.decorator;
public class TesterDecorator extends EmployeeDecorator {
    // поля, методы
    public TesterDecorator(Employee employee) {
        super(employee);
    }
    @Override
    public void openTask() {
        super.openTask();
        testing();
    }
    @Override
    public void reopenTask() {
        super.reopenTask();
```

```
        testing();
    }
    @Override
    public void resolveTask() {
        reporting();
        super.resolveTask();
    }
    public void testing() {
        System.out.println(employee.getName() + " testing task");
    }
    public void reporting() {
        System.out.println(employee.getName() + " create report");
    }
}
package by.bsu.decorator;
public class TeamLeadDecorator extends EmployeeDecorator {
    // поля, методы
    public TeamLeadDecorator(Employee employee) {
        super(employee);
    }
    @Override
    public void openTask() {
        super.openTask();
        assignTask();
    }
    @Override
    public void reopenTask() {
        super.reopenTask();
        changeEmployee();
    }
    @Override
    public void resolveTask() {
        super.resolveTask();
        closeTask();
    }
    public void assignTask() {
           System.out.println(employee.getName()
                                          + " is assigning task");
    }
    public void changeEmployee() {
        System.out.println(employee.getName()
                                        + " is changing employee");
    }
    public void closeTask() {
        System.out.println(employee.getName() + " is closing task");
    }
}
```

Создав экземпляр класса **Employee,** можно делегировать ему выполнение задач, связанных с разработчиком, тестировщиком или team-лидером, без создания специализированных подклассов.

/ # 16 # использование шаблона Decorator # RunnerDecorator.java */*

```
package by.bsu.decorator;
public class RunnerDecorator {
    public static void main(String[] args) {
        IEmployee employee = new TesterDecorator(
                                        new Employee("Ivanov"));
        employee.reopenTask();
        employee = new TeamLeadDecorator(new Employee("Petrov"));
        employee.openTask();
    }
}
```

Шаблон Façade

Шаблон позволяет упростить доступ к сложной системе, объединив несколько действий различных классов под одним интерфейсом и делегировав ему обязанности отправлять сообщения этим классам. Клиент системы знает только об интерфейсе фасада и ничего не знает о структуре классов и последовательности вызовов, приводящих его к желаемому результату. В роли интерфейса в общем случае будет выступать совокупность классов, а не один класс со множеством обязанностей. Примером **Façade** может служить библиотека **mysql-connector-[***версия***].jar** для организации соединения и выполнения запросов к базе данных MySQL.
При разработке приложения часто приходится использовать классы/пакеты/библиотеки, созданные другими разработчиками. Причем не суть важно, являются ли эти программисты членами вашей команды или это библиотека отфирмы-производителя. При использовании таких библиотек для получения результата обычно необходимо выполнить вызов нескольких методов по определенному правилу. Причем обращение к этому функционалу может производиться довольно часто из разных частей кода программиста. В такой ситуации имеет смысл выделить этот интерфейс в отдельный код, организованный в классы и пакеты отдельного класса и пользоваться им при необходимости, не усложняя собственный код.

Как частный случай решения в качестве **Façade** может выступать единственный класс с единственным методом. Такое примитивное решение уже конкурирует с аналогичными шаблонами. Существует опасность получить сильно связанный класс с громоздким «волшебным» методом.

```
/* # 17 # базовая реализация шаблона Facade # IFacade.java # Facade.java */
package by.bsu.facade.base;
public interface IFacade {
    void generate();
    void find();
}
package by.bsu.facade.base;
public class Facade implements IFacade {
    private SecuritySystem securitySystem;
    private SubSystem subSystem;
    public Facade() {
        // варианты инициализации могут быть разными
        this.subSystem = new SubSystem();
        this.securitySystem = new SecuritySystem();
    }
    @Override
    public void generate() {
        // проверка пользователя и его прав
        securitySystem.checkUser();
        securitySystem.checkRights();
        // действие create
        subSystem.createNode();
    }
    @Override
    public void find() {
        // проверка пользователя
        securitySystem.checkUser();
        // действие parse
        subSystem.parse();
    }
}
```

Скрывать классы, используемые фасадом, не обязательно. Клиент может даже предоставлять данные для их инициализации.

```
/* # 18 # реализация шаблона Facade # SecuritySystem.java # SubSystem.java
# ClientRunner.java */

package by.bsu.facade.base;
public class SecuritySystem {
    boolean checkUser() {
        return true;
    }
    boolean checkRights() {
        return true;
    }
}
```

```
package by.bsu.facade.base;
public class SubSystem {
    public void parse() {
        System.out.println("Parsing");
    }
    public void createNode() {
        System.out.println("Creating program node");
    }
}
package by.bsu.facade.base;
public class ClientRunner {
    public static void main(String[] args) {
        Facade facade = new Facade();
        facade.generate();
    }
}
```

Снижается число связей клиентского класса с системой. Каждый метод фасада решает свою конкретную задачу, упрощая клиенту обращение к системе. При реализации фасада не следует допускать прямое обращение класса-клиента к классу системы, минуя фасад.

Например, в главе «XML & Java» в примере даны примеры кода для валидации XML-документа. Одну из вариаций можно записать в виде:

```
/* # 19 # проверка корректности документа XML без определения Façade #
ValidatorSAXwithXSD.java */
```

```
package by.bsu.valid;
import java.io.*;
import javax.xml.XMLConstants;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.xml.validation.Schema;
import javax.xml.validation.SchemaFactory;
import javax.xml.validation.Validator;
import org.xml.sax.SAXException;
public class ValidatorSAXwithXSD {
    public static void main(String[] args) {
        String language = XMLConstants.W3C_XML_SCHEMA_NS_URI;
        String filename = "data/students.xml";
        String schemaname = "data/students.xsd";
        SchemaFactory factory = SchemaFactory.newInstance(language);
        File schemaLocation = new File(schemaname);
        try {
            Schema schema = factory.newSchema(schemaLocation);
            Validator validator = schema.newValidator();
            Source source = new StreamSource(filename);
```

```
            StudentErrorHandler sh = new StudentErrorHandler("log.txt");
            validator.setErrorHandler(sh);
            validator.validate(source);
            System.out.println(filename + " validating is ended "
                                                    + "correctly");
        } catch (SAXException | IOException e) {
            //запись log
        }
    }
}
```

Данный код выполняет некоторую последовательность обязательных действий, определенных разработчиком библиотеки. Чтобы решением можно было воспользоваться повторно, следует сделать фасад для него и фактически скрыть детали реализации за фасадом метода, а именно:

/ # 20 # вынесение реализации за Facade # FacadeValidator.java */*

```
package by.bsu.valid;
import java.io.File;
import java.io.IOException;
import javax.xml.XMLConstants;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.xml.validation.Schema;
import javax.xml.validation.SchemaFactory;
import javax.xml.validation.Validator;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;
public class FacadeValidator {
    public boolean validateSAXwithXSD(String xmlFileName,
                    String schemaFileName, DefaultHandler handler) {
        boolean result = false;
        String language = XMLConstants.W3C_XML_SCHEMA_NS_URI;
        SchemaFactory factory = SchemaFactory.newInstance(language);
        File schemaLocation = new File(schemaFileName);
        try {
            Schema schema = factory.newSchema(schemaLocation);
            Validator validator = schema.newValidator();
            Source source = new StreamSource(xmlFileName);
            validator.setErrorHandler(handler);
            validator.validate(source);
            result = true;
        }
```

```
        catch (SAXException | IOException e) {
             //запись   log
        }
        return result;
    }
}
```

Имена файлов и экземпляр класса-обработчика ошибок передаются в метод в качестве параметров. Метод же содержит стандартную последовательность действий, вмешиваться в которую нет разумных оснований.

пример использования паттерна "Фасад" в системе обработки банковских транзакций:

Представим, что у вас есть сложная система для обработки банковских транзакций, которая включает в себя множество классов и подсистем, таких как классы для управления счетами, классы для обработки платежей, классы для взаимодействия с внешними платежными шлюзами и т.д. Каждый из этих классов имеет свои методы и логику.

Однако, чтобы упростить взаимодействие с системой и предоставить простой интерфейс для выполнения сложных операций, вы можете использовать паттерн "Фасад".

Вариант реализации паттерна "Фасад" в системе обработки банковских транзакций может выглядеть следующим образом:

```
public class BankingFacade {

    private AccountManager accountManager;

    private PaymentProcessor paymentProcessor;

    private ExternalGatewayManager externalGatewayManager;

    public BankingFacade() {

        // Инициализация необходимых подсистем

        accountManager = new AccountManager();

        paymentProcessor = new PaymentProcessor();

        externalGatewayManager = new ExternalGatewayManager();
```

```
    }

    public void processPayment(String accountId, double amount) {
        // Выполнение операции обработки платежа с использованием различных подсистем
        Account account = accountManager.getAccount(accountId);
        Payment payment = new Payment(account, amount);
        boolean isValidated = paymentProcessor.validatePayment(payment);
        if (isValidated) {
            externalGatewayManager.processPayment(payment);
            accountManager.updateAccountBalance(accountId, -amount);
        }
    }

    // Другие методы фасада для выполнения банковских операций
}
```

В данном примере класс BankingFacade представляет собой фасад, который скрывает сложность внутренней системы обработки банковских транзакций. Он инкапсулирует различные подсистемы, такие как AccountManager, PaymentProcessor и ExternalGatewayManager, и предоставляет простой интерфейс для выполнения операций, например, processPayment(String accountId, double amount).

При вызове метода processPayment фасад выполняет все необходимые шаги для обработки платежа, включая получение аккаунта с помощью AccountManager, валидацию платежа с помощью PaymentProcessor, обработку платежа с использованием ExternalGatewayManager и обновление баланса аккаунта.

Таким образом, паттерн "Фасад" помогает упростить взаимодействие с системой обработки банковских транзакций, предоставляя простой интерфейс через фасадный класс BankingFacade, который скрывает сложность внутренней системы и делает ее более легкой в использовании.

Шаблон Composite

Предоставляет возможность строить сложные объекты с использованием рекурсии. Позволяет рассматривать объект как комбинацию более простых в целом составляющих древовидную структуру. Составной элемент представляет собой набор из частей с аналогичной природой. Часть целого в таком случае представляется как набор более мелких частей, и так до тех пор, пока не будет выделена некая элементарная часть. Элементарная часть уже неделима.

Примером может служить соотношение между текстом и его частями, текст может состоять из абзацев и листингов, последние, в свою очередь, — из предложений, предложения — из слов и знаков препинания, неделимый элемент — символ. В другом случае **Composite** может описывать карту местности, состоящей из областей, районов, городов и т. д.

Все классы из древовидной структуры реализуют общий интерфейс, тогда при создании любого объекта из любой части «ветки» действия будут выполняться идентичные и способ обращения ко всем элементам будет абсолютно одинаковым.

Интерфейс **Component** задает интерфейс для всех составных объектов. У интерфейса **Component** обычно существует одна или несколько реализаций типа **Leaf**, не имеющие потомков и описывающие неделимые элементы структуры. Тип **Composite** хранит составные и неделимые компоненты и определяет их поведение. Способ организации и поведения **Composite** зависит от задач, решаемых этим самым составным объектом.

/ # 21 # общий интерфейс древовидной структуры # Component.java */*

```
package by.bsu.composite.base;
public interface Component {
    void operation();
    void add(Component c);
    void remove(Component c);
    Component getChild(int index);
  }
```

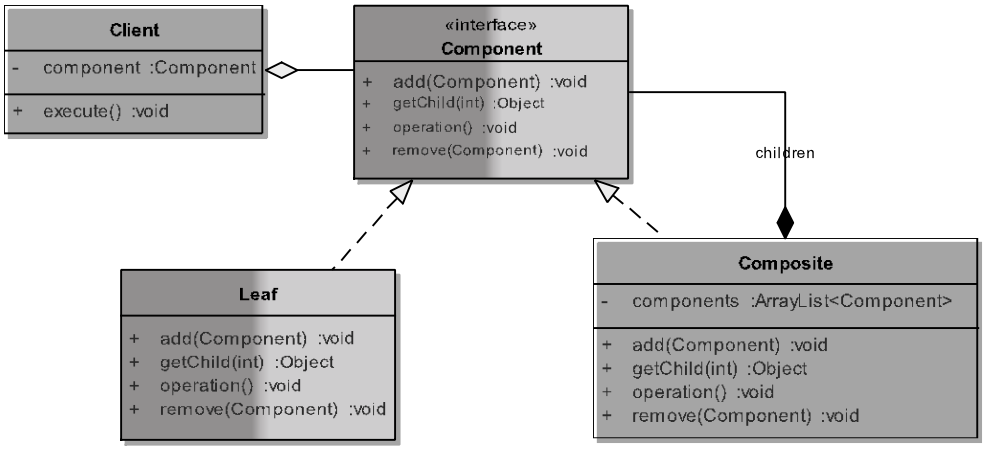

Рис. 23.5. *Базовая реализация шаблона Composite*

```
/* # 22 # неделимый элемент древовидной структуры # Leaf.java */
```

```
package by.bsu.composite.base;
public class Leaf implements Component {
    // поля
    public void operation() {
        System.out.println("Leaf -> Performing operation");
    }
    public void add(Component c) {
        System.out.println("Leaf -> add. Doing nothing");
        // генерация исключения и return false (если метод не void)
    }
    public void remove(Component c) {
         System.out.println("Leaf -> remove. Doing nothing");
         // генерация исключения и return false (если метод не void)
    }
    public Object getChild(int index) {
         throw new UnsupportedOperationException();
    }
}
```

```
/* # 22 # составной объект # Composite.java */
```

```
package by.bsu.composite.base;
import java.util.ArrayList;
```

```
public class Composite implements Component {
    private List<Component> components = new ArrayList<>();
    public void operation() {
        System.out.println("Composite -> Call children operations");
```

```
        int size = components.size();
        for (int i = 0; i < size; i++) {
             components.get(i).operation();
        }
    }
    public void add(Component component) {
        System.out.println("Composite -> Adding component");
        components.add(component);
    }

    public void remove(Component component) {
        System.out.println("Composite -> Removing component");
        components.remove(component);
    }
    public Object getChild(int index) {
        System.out.println("Composite -> Getting component");
        return components.get(index);
    }
}
```

/ # 23 # клиентский класс (необязателен), которому необходим составной в качестве поля # Client.java */*

```
package by.bsu.composite.base;
public class Client {
    private Component component;
    public Client(Component component) {
        this.component = component;
    }
    public void execute() {
        component.operation();
    }
}
```

/ # 24 # запуск процесса организации Composite # CompositeRunner.java */*

```
package by.bsu.composite.base;
public class CompositeRunner {
    public static void main(String[] args) {
        Component composite = new Composite();
        Component leaf = new Leaf();
        leaf.add(composite); // nothing happens
        composite.add(leaf);
        Client client = new Client(composite);
        client.execute();
    }
}
```


Методы, управляющие добавлением/удалением/изменением любой состав-

ной части, не могут поддерживаться неделимыми частями в принципе.

Пусть необходимо разработать элементы организации набора каналов для загрузки больших информационных файлов, где пулы каналов могут состоять как из отдельных соединений разных видов, так и из пулов, которые, в свою очередь, могут состоять из отдельных каналов и других пулов. В этом случае явно необходима реализация рекурсивного включения одного пула в другой.

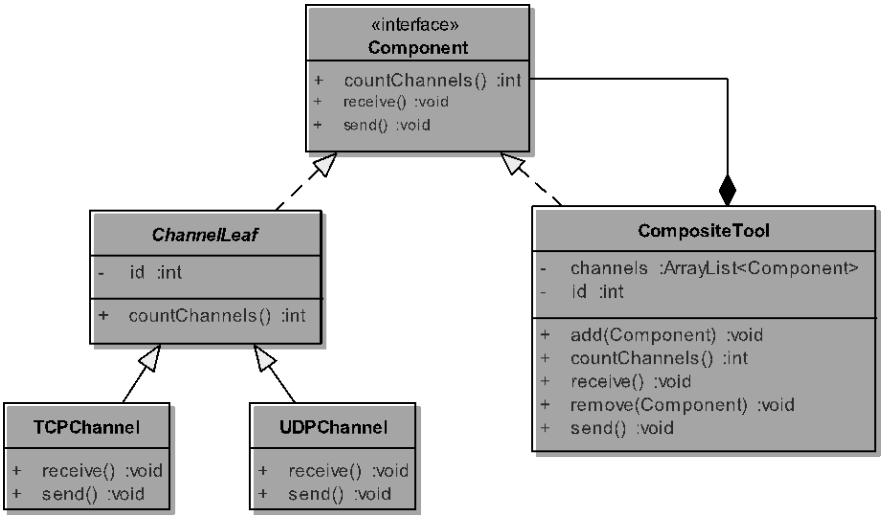

Рис. 23.6. *Реализация шаблона Composite*

В интерфейсе **Component** отсутствуют объявления методв **add()** и **remove()**, что исключает необходимость для классов типа **Leaf** предоствлять их реализацию. Каналы могут собираться в **CompositTool**. Сам **CompositTool** может быть частью другого экземпляра **CompositTool**. Каналы и экземпляры **CompositTool** могут удаляться и добавляться в набор в любой момент по требованию пользователя.

/ # 25 # общий интерфейс для каналов и их наборов # Component.java */*

```
package by.bsu.composite;

public interface ProjectComponent {
    void add(ProjectComponent component);
    void remove(ProjectComponent component);
    long defineTimeRequired();
}
package by.bsu.composite;

public class Task implements ProjectComponent {//Code → Task
    private long codeId;
    private ReferenceCVS reference;
    private long timeRequired;
    private CodeType type;
```

```
    private boolean done;

    public Task(long codeId, long timeRequired, CodeType type) {
        this.codeId = codeId;
        this.timeRequired = timeRequired;
        this.type = type;
    }
    @Override
    public void add(ProjectComponent component) {
        // empty or generating an exception
    }
    @Override
    public void remove(ProjectComponent component) {
        // empty or generating an exception
    }
    public boolean isDone() {
        return done;
    }
    public void setDone(boolean done) {
         this.done = done;
    }
    @Override
    public long defineTimeRequired() {
        System.out.println("     codeId=" + codeId +
                ", timeRequired=" + timeRequired);
        return timeRequired;
    }
    public long getTimeRequired() {
        return timeRequired;
    }
    public void setTimeRequired(long timeRequired) {
        this.timeRequired = timeRequired;
    }
    public CodeType getType() {
        return type;
    }
    @Override
    public String toString() {
        return "Code{codeId=" + codeId + ", reference=" + reference +
                ", timeRequired=" + timeRequired + ", type=" + type + '}';
    }
}
package by.bsu.composite;

public enum CodeType {
    CODE, TEST
}
package by.bsu.composite;

public class ReferenceCVS {
    // more code
}
```

/ # 27 # реализация для составного элемента (Composite) # CompositeTool.java */*

```
package by.bsu.composite;
import java.util.LinkedList;

public class CompositeProject implements ProjectComponent {// TaskComposite
    private long taskId;//????
    private LinkedList<ProjectComponent> listComponent = new LinkedList<>();
```

```
    public CompositeProject(long taskId) {
        this.taskId = taskId;
    }
    @Override
    public void add(ProjectComponent component) {
        listComponent.add(component);
    }
    @Override
    public void remove(ProjectComponent component) {
        listComponent.remove(component);
    }
    @Override
    public long defineTimeRequired() {
        long fullTime = 0;
        for (ProjectComponent projectComponent : listComponent) {
           if (!done) {
            fullTime += projectComponent.defineTimeRequired();
        }
        System.out.println(taskId + ", timeRequired = " + fullTime);
        return fullTime;
    }
    public long getTaskId() {
        return taskId;
    }
    public void setTaskId(long taskId) {
        this.taskId = taskId;
    }
}
```

/ # 28 # организация набора каналов и демонстрация процесса # Composite.java */*

```
package by.bsu.composite;

public class CompositeDemo {
    public static void main(String[] args) {
        Task code0 = new Code(700, 100, CodeType.CODE);
        Task code1 = new Code(701, 10, CodeType.CODE);
        Task code2 = new Code(707, 12, CodeType.CODE);
        Task code3 = new Code(777, 9, CodeType.CODE);
        Task code4 = new Code(717, 10, CodeType.CODE);
        Task code5 = new Code(771, 11, CodeType.CODE);
        Task codetest0 = new Code(552, 8, CodeType.TEST);
        Task codetest1 = new Code(522, 7, CodeType.TEST);
        Task codetest2 = new Code(555, 6, CodeType.TEST);
        Task codetest3 = new Code(525, 8, CodeType.TEST);
        Task codetest4 = new Code(558, 9, CodeType.TEST);
        CompositeProject project = new CompositeProject(1);
        CompositeProject task1 = new CompositeProject(55);
        CompositeProject task2 = new CompositeProject(77);

        project.add(task1);
        project.add(task2);
        project.add(code0);
        task1.add(code1);
        task1.add(code2);
        task1.add(code3);
        task1.add(codetest0);
        task1.add(codetest1);
        task1.add(codetest2);
        task1.add(codetest3);
        task2.add(code4);
        task2.add(code5);
        task2.add(codetest4);
```

```
        System.out.println("Total time=" + project.defineTimeRequired());
        task1.remove(codetest0);
        task1.remove(codetest1);
        codetest2.setDone(true);
        code0.setDone(true);
        System.out.println("Total time=" + project.defineTimeRequired());
    }
}
```

В результате будет выведено:

```
    codeId=701, timeRequired=10
    codeId=707, timeRequired=12
    codeId=777, timeRequired=9
    codeId=552, timeRequired=8
    codeId=522, timeRequired=7
    codeId=555, timeRequired=6
    codeId=525, timeRequired=8
55, timeRequired = 60
    codeId=717, timeRequired=10
    codeId=771, timeRequired=11
    codeId=558, timeRequired=9
77, timeRequired = 30
    codeId=700, timeRequired=100
1, timeRequired = 190
Total time=190
    codeId=701, timeRequired=10
    codeId=707, timeRequired=12
    codeId=777, timeRequired=9
    codeId=555, timeRequired=0
    codeId=525, timeRequired=8
55, timeRequired = 39
    codeId=717, timeRequired=10
    codeId=771, timeRequired=11
    codeId=558, timeRequired=9
77, timeRequired = 30
    codeId=700, timeRequired=0
1, timeRequired = 69
Total time=69
```

Шаблон Adapter

Пусть существуют две системы, выполняющие сходные действия, но несовместимые по интерфейсу, которые, тем не менее, необходимо заставить работать в одном приложении.

Позволяет определить интерфейс-адаптер, доступный его пользователям-клиентам, в то же время способ реализации отделяет от клиентов и не известен им. Однако функциональность, предоставляемая клиентам, соответствует за-

данному контракту. Обеспечивается взаимодействие несовместимых интерфейсов, предоставляемых классами типа **Adaptee**.

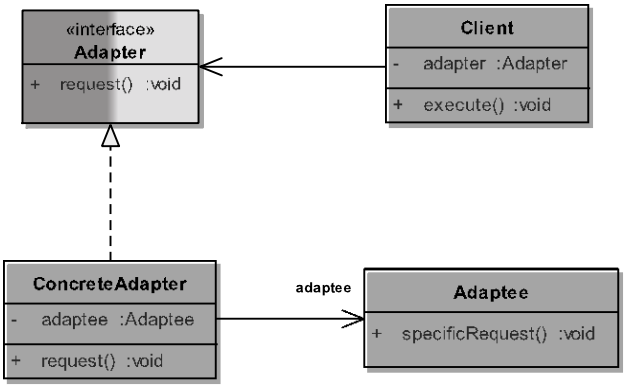

Рис. 23.7. *Базовая реализация шаблона Adapter*

```
/* # 29 # организация адаптера # Adapter.java # ConcreteAdapter.java */
package by.bsu.adapter.base;
public interface Adapter {
    void request();
}
package by.bsu.adapter.base;
public class ConcreteAdapter implements Adapter {
    private Adaptee adaptee;
    public ConcreteAdapter(Adaptee adaptee) {
        this.adaptee = adaptee;
    }
    public void request() {
        System.out.println("Return type - void.");
        adaptee.specificRequest();
    }
}
```

```
/* # 30 # адаптируемый класс # Adaptee.java */
package by.bsu.adapter.base;
public class Adaptee {
    public boolean specificRequest() {
        System.out.println("Return type - boolean");
        return true; // stub
    }
}
```

Класс **Client** может взаимодействовать только с экземплярами, реализующими интерфейс **Adapter**. Экземпляр класса **Client**, вызывая метод **execute()**, не будет знать, метод какого класса он вызывает — основного или адаптируемого. Предназначение шаблона **Adapter** — без серьезной переработки системы включить необходимый функционал в общем случае несовместимый с существующим.

/ # 31 # адаптируемый класс # Adaptee.java */*

```
package by.bsu.adapter.base;
public class Client {
    private Adapter target;
    public Client(Adapter target) {
        this.target = target;
    }
    public void execute(Adapter target) {
        target.request();
    }
}
```

/ # 32 # запуск процесса адаптации # RunnerAdapter.java */*

```
package by.bsu.adapter.base;
public class RunnerAdapter {
    public static void main(String[] args) {
        Adapter target = new ConcreteAdapter(new Adaptee());
        Client client = new Client(target)
        client.execute();
    }
}
```

Реализация, представленная выше, использует один класс типа **Adaptee**. В общем случае эти классы могут быть организованы в иерархию и при инициализации объекта типа **Adapter** будет передаваться объект из иерархии, и скрытое действие будет выполняться в соответствии с конкретным типа объекта. Но это уже будет не совсем шаблон **Adapter**.
Класс **Adapter** должен обладать механизмом согласования параметров и способом трансляции вызова методов основного приложения иадаптируемого класса.

Пусть в систему JSON-парсинга необходимо подключить разбор XML-документа. Для этого выделяется интерфейс **BaseParser** с методом **parse()**. Существующий класс **JsonParser** теперь должен имплементировать новый интерфейс. Для адаптации интерфейса класса **XmlParser** создается класс **XmlParserAdapter**, объявляющий адаптируемый класс в качестве поля, а также в качестве полей объявляет необходимые для согласования интерфейса параметры. Метод **parse()** адаптера теперь может корректно вызвать метод парсинга адаптируемого класса.

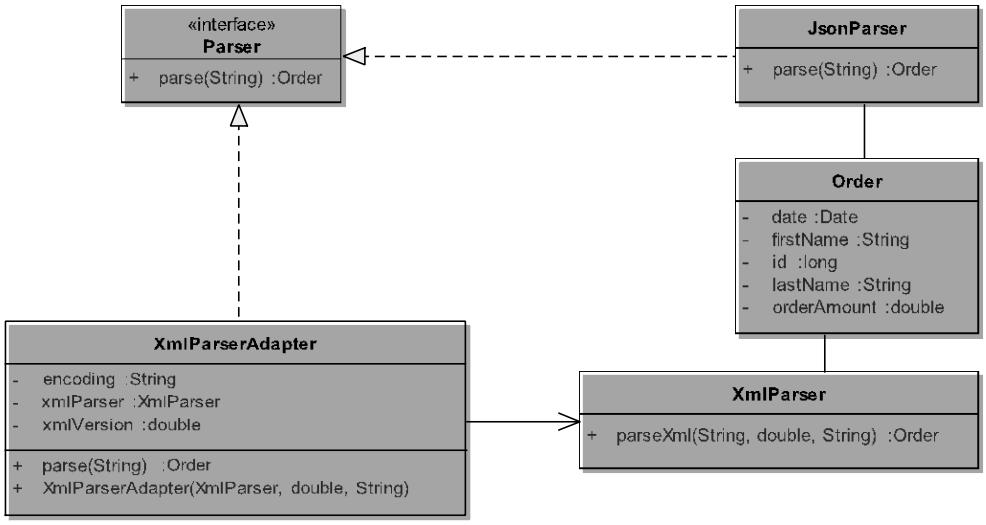

Рис. 23.8. *Парсинг документов с шаблоном Adapter*

/ # 33 # адаптируемый функционал # XmlParser.java */*

```
package by.bsu.adapter;
public class XmlParser {
    public Order parseXml(String str, String encoding) {
        Order order = new Order();
        System.out.println("Parse xml file for Order");
        return order;
    }
}
```

/ # 34 # организация адаптера # Parser.java # XmlParserAdapter.java */*

```
package by.bsu.adapter;
public interface BaseParser {
    Order parse(String str);
}
package by.bsu.adapter;
public class XmlParserAdapter implements BaseParser {
    private XmlParser xmlParser;
    private String encoding; /* согласование параметров для
                                                адаптации */
```

```
    public XmlParserAdapter(XmlParser xmlParser,
              String encoding) {
        this.xmlParser = xmlParser;
        this.encoding = encoding;
    }
    @Override
    public Order parse(String strOrder) {
        return xmlParser.parseXml(strOrder, encoding);
    }
}
```

/ # 35 # существующий функционал # JsonParser.java */*

```
package by.bsu.adapter;
public class JsonParser implements BaseParser {
    public Order parse(String jsonOrder) {
        Order order = new Order();
        System.out.println("Parse JSON for Order");
        return order;
    }
}
```

/ # 36 # запуск процесса адаптации # AdapterMain.java */*

```
package by.bsu.adapter;
public class AdapterMain {
    public static void main(String args[]) {
        String jsonOrder =
                  "\"id\": \"1456\", \"firstName\": \"John\", "
                  + "\"lastName\": \"Smith\" ...";
        BaseParser parser = new JsonParser();
        Order order = parser.parse(jsonOrder);
        System.out.println(order.getOrderAmount());
        String xmlOrder =
                 "<order id=\"1456\"><person firstName=\"John\""
                  + " lastName=\"Smith\">EPAM</person></order>";
        parser = new XmlParserAdapter(new XmlParser(), "UTF-8");
        order = parser.parse(xmlOrder);
        System.out.println(order.getOrderAmount());
    }
}
```


/ # 37 # создаваемая сущность # Order.java */*

```
package by.bsu.adapter;
public class Order {
}
```

Шаблон Flyweight

Экземпляр класса может использоваться как набор независимых экземпляров. С помощью разделяемого содержимого экземпляра класса возможна экономия ресурсов при инициализации большого числа схожих объектов. **Flyweight**, точнее, разделяемое содержание, которое в тоже время представляет собой единое целое, можно использовать в различных представлениях, не утрачивая исходного смысла класса-приспособленца. Приспособленцам недоступно представление, в котором они существуют.

В основе лежит различие между внутренней частью класса и его внешним состоянием. Внутренняя часть отделена от окружения, в котором работает объект и недоступна для изменения с его стороны. Отделение внешней части делает все экземпляры идентичными. Внешняя часть зависит от окружения и может быть им изменена, вследствие чего не может быть отделена. Объекты-клиенты имеют возможность передавать внешнее состояние приспособленцу. Применение **Flyweight** позволяет снизить число объектов в системе, а следовательно, экономить ресурсы, увеличивать скорость работы.

/ # 38 # интерфейс приспособленца и его реализация # Flyweight.java # ConcreteFlyweight.java */*

```
package by.bsu.flyweight.base;
public interface Flyweight {
    void operation();
}
package by.bsu.flyweight.base;
public class ConcreteFlyweight implements Flyweight {
    public void operation() {
       // реализация
    }
}
```

/ # 39 # неразделяемая сущность, существующая вне рамок шаблона # UnsharedConcreteFlyweight.java */*

```
package by.bsu.flyweight.base;
public class UnsharedConcreteFlyweight implements Flyweight {
    public void operation() {
       // more code
    }
}
```

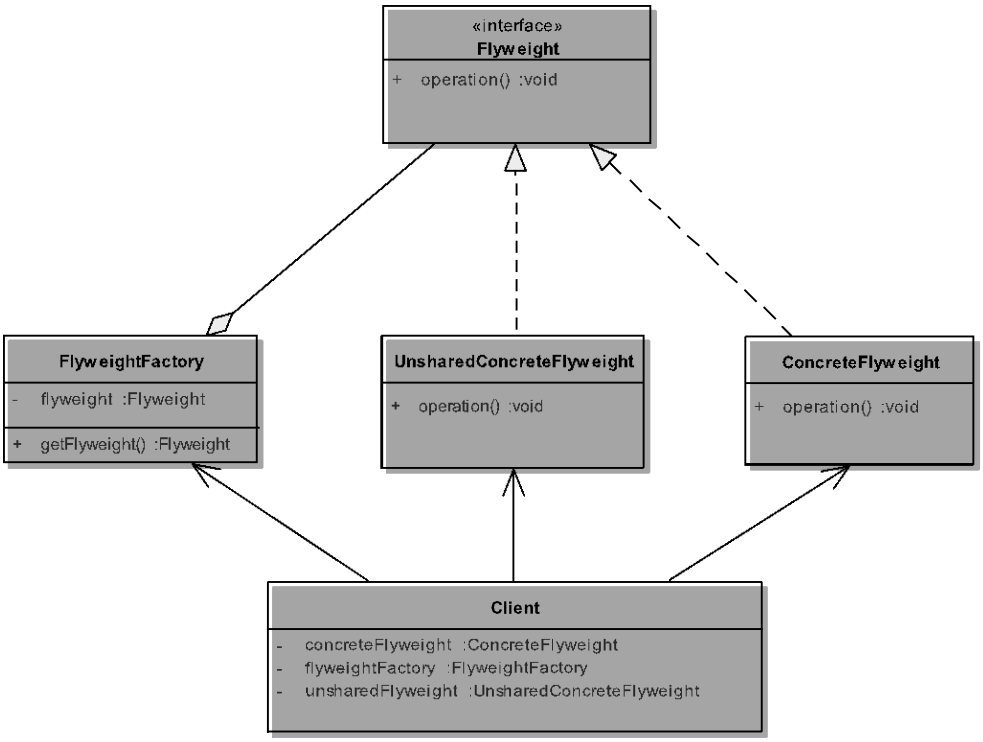

Рис. 23.9. *Базовая реализация шаблона Flyweight*

/ # 40 # фабрика легковесов и их использование # FlyweightFactory.java # Client.java*

```
package by.bsu.flyweight.base;
public class FlyweightFactory {
    private Flyweight flyweight;
    public Flyweight getFlyweight() {
        return new ConcreteFlyweight();
    }
}
package by.bsu.flyweight.base;
public class Client {
    private FlyweightFactory flyweightFactory;
    private ConcreteFlyweight concreteFlyweight;
    private UnsharedConcreteFlyweight unsharedFlyweight;
    // some code here
}
```

При моделировании динамической имитационной модели термитника и его обитателей может понадобиться создание нескольких миллионов однотипных объектов.

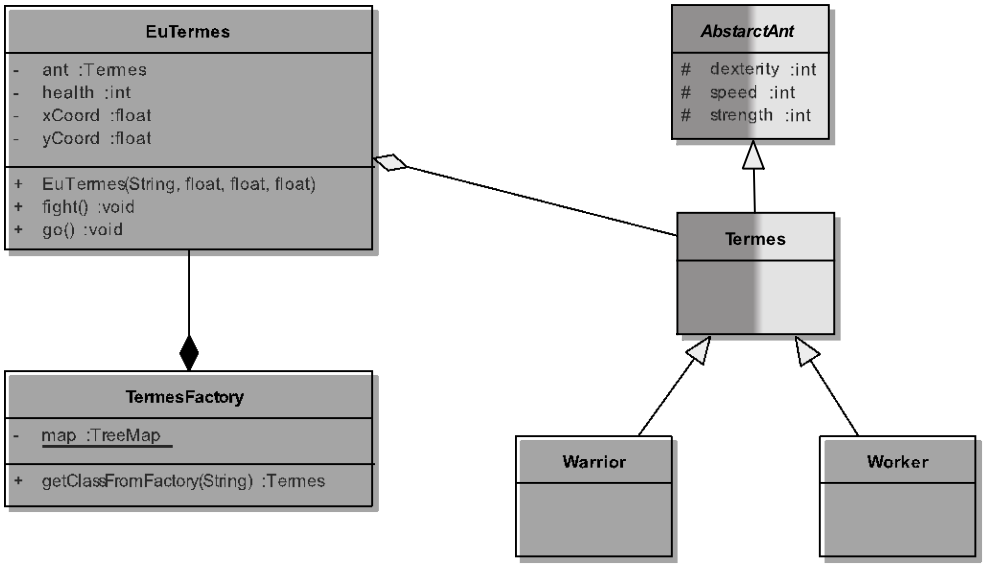

Рис. 23.10. *Термитник на основе шаблона Flyweight*

Пусть определяются два вида термитов: **Worker** и **Warrior**.

```
package by.bsu.flyweight;
public abstract class AbstarctAnt {
    protected int strength;
    protected int dexterity;
    protected int speed;
    public int getStrength() {
        return strength;
    }
    public void setStrength(int strength) {
        this.strength = strength;
    }
    public int getDexterity() {
        return dexterity;
    }
    public void setDexterity(int dexterity) {
        this.dexterity = dexterity;
    }
    public int getSpeed() {
        return speed;
    }
```

```
public void setSpeed(int speed) {
        this.speed = speed;
    }
}

package by.bsu.flyweight;
import java.io.Serializable;
public class Termes extends AbstarctAnt implements Serializable {
    // more code here
}
package by.bsu.flyweight;
import java.io.Serializable;
public class Warrior extends Termes implements Serializable {
    public Warrior() {
        strength = 10;
        dexterity = 4;
        speed = 6;
    }
    // more code here
}
package by.bsu.flyweight;import java.io.Serializable;
public class Worker extends Termes implements Serializable {
    public Worker() {
        strength = 6;
        dexterity = 9;
        speed = 10;
    }
    // more code here
}
package by.bsu.flyweight;
public enum TermesType {
    WORKER, WARRIOR
}
```

Основной задачей является грамотное разделение состояний на внутренне и внешнее. Сами термиты представляются практически идентичными, но каж-дый из них располагается в конкретной точке пространства.

/ # 42 # разделение состояний на основе агрегирования # EuTermes.java */*

```
package by.bsu.flyweight;
import java.io.Serializable; // flyweight (приспособленец)
public class EuTermes implements Serializable {
    private Termes ant;
    // внутренняя часть
    // внешнее состояние: начало описания
    private int health;
    private float xCoord;
```

```
    private float yCoord;
    // внешнее состояние: окончание описания
    public EuTermes(String type, float xCoord, float yCoord,
                                                  float zCoord) {
        this.ant = TermesFactory.getClassFromFactory(type);
        health = 180;
        this.xCoord = xCoord;
        this.yCoord = yCoord;
    }
    public int getHealth() {
        return health;
    }
    public void setHealth(int health) {
        this.health = health;
    }
    public float getXCoord() {
        return xCoord;
    }
    public void setXCoord(float xCoord) {
        this.xCoord = xCoord;
    }
    public float getYcoord() {
        return yCoord;
    }
    public void setYcoord(float ycoord) {
        this.yCoord = ycoord;
    }
    public void go() {
      // more code here
    }
    public void fight () {
      // more code here
    }
}
```

Чем больше атрибутов можно сделать внешними, тем меньше потребуется экземпляров-приспособленцев, но чем больше атрибутов сделать внутренними, тем меньше времени будет затрачено экземплярами для доступа к своим собственным полям.

/ # 43 # фабрика # TermesFactory.java */*

```
package by.bsu.flyweight;
import java.util.Map;
import java.util.TreeMap;
public class TermesFactory {
    private static TreeMap<String, Termes> map = new TreeMap<>();
    public static Termes getClassFromFactory(String name) {
```

```
        if (map.containsKey(name)) {
            return map.get(name);
        }
        else {
           TermesType type = termesType.valueOf(name.toUpperCase());
            switch (type) {
                case WORKER: {
                    Worker worker = new Worker();
                    map.put(name, worker);
                    return worker;
                }
                case WARRIOR: {
                    Warrior warrior = new Warrior();
                    map.put(name, warrior);
                    return warrior;
                }
                default:
                   throw
new EnumConstantNotPresentException(TermesType.class, type.name());
            }
        }
    }
}
```

/ # 44 # клиентское приложение # FlyweightRunner.java */*

```
package by.bsu.flyweight;
import java.util.ArrayList;
public class FlyweightRunner {
    private final static int SIZE = 4_000_000;
    public static void main(String[] args) {
        ArrayList<EuTermes> legion = new ArrayList<>();
        try {
            for (int i = 0; i < SIZE; i++) {
                legion.add(new EuTermes("Worker", 12.3f,
                                        10.1f, 5.0f));
            }
        }
        catch (OutOfMemoryError e) {
            System.err.println("OutOfMemoryError for "
                                    + "Termes Worker");
        }
        finally {
            System.out.println("Instance: " + i);
        }
    }
}
```

В результате будет получено:

OutOfMemoryError for Termes
Worker Instance: 2261945

т.е. память закончится только после создания **2261945** объектов.

Если же создавать объекты в «лоб», как показано в следующем коде, то память закончится значительно раньше.

```
/* # 45 # реализация без разделения состояний # TermesBad.java # BadRunner.java */
```

```
package by.bsu.flyweight;
public class TermesBad {
    private int strength;
    private int dexterity;
    private int speed;
    private int health;
    private float xCoord;
    private float yCoord;
    {
        strength = 10;
        dexterity = 4;
        speed = 6;
        health = 180;
        xCoord = 10.2f;
        yCoord = 12.8f;
    }
    // constructors, setters, getters & more
}
package by.bsu.flyweight;
import java.util.ArrayList;
public class BadRunner {
    private static int size = 4_000_000;
    public static void main(String[] args) {
        ArrayList<TermesBad> legion = new ArrayList<>();
        try {
            for (int i = 0; i < size; i++) {
                legion.add(new TermesBad());
                }
        } catch (OutOfMemoryError e) {
            System.err.println("OutOfMemoryError for TermesBad");
        } finally {
            System.out.println("Instance: " + i);
        }
    }
}
```

Получено будет:
OutOfMemoryError for TermesBad
Instance: 1507963

Шаблон Proxy

Прокси-объект представляет другой объект. С точки зрения клиента интерфейс и функциональность класса остаются практически неизменными. Чтобы это представление было естественным, прокси-объект обязан реализовывать тот же интерфейс, что и реальный класс. Кроме того, прокси-объект должен содержать реальный класс в качестве поля, чтобы при необходимости обращаться к функционалу реального класса. В общем случае реализует технологию обертывания (wrapping) класса с целью повышения безопасности или оптимизации взаимодействия.

/ # 46 # интерфейс и класс # BaseInterface.java # BaseSubject.java */*

```
package by.bsu.proxy.base;
public interface BaseInterface {
    void perform();
}
package by.bsu.proxy.base;
public class BaseSubject implements BaseInterface {
    public void perform() {
        System.out.println("...performing...");
    }
}
```

/ # 47 # класс proxy # Proxy.java */*

```
package by.bsu.proxy.base;
public class Proxy implements BaseInterface {
    private BaseSubject base = new BaseSubject();
    Proxy() {}// new constructors
    @Override
    public void perform() {
        base.perform();
    }
// new methods
}
```

/ # 48 # запуск # DemoProxy.java */*

```
package by.bsu.proxy.base;
public class DemoProxy {
    public static void main(String args[]) {
        BaseInterface base = new Proxy();
        base.perform();
```

```
    }
}
```


Существует некоторый пул соединений с БД. Соединения в виде объекта типа **Connection** по мере необходимости извлекаются из пула и после использования возвращаются. При такой реализации пул оказывается незащищенным от попадания в него «диких» соединений, созданных вне пула, в то время как соединение, взятое из пула, не сможет возвратиться в него, так как пул может оказаться заполнен «дикими» соединениями. В итоге приложения из-за создания несанкционированных соединений и нарушений работы пула будет работать медленнее и с ошибками, возможно, критическими. Также возможно преждевременное (несанкционированное) закрытие соединения.

Для решения проблемы следует создать класс прокси-соединения и переписать реализацию класса-пула.

```
public class ConnectionPool {
    private BlockingQueue<ProxyConnection> queue;
    public void put (Connection connection) {
            if(connection instanceof ProxyConnection) {
                queue.put((ProxyConnection)connection);
            } else {
                // throw exception
            }
    }
    public Connection take() throws InterruptedException {
            return queue.take();
    }
}
```

то соединение, извлеченное из пула, позволяет выполнять все необходимые для объекта такого рода действия. При возвращении соединения в пул методом **put(Connection connection)** можно передать только прокси-объект. Попытка возвращения обычного экземпляра **Connection** приведет к ошибке компиляции. То есть «дикие» соединения попасть в пул не могут, и безопасность пула обеспечена.

/ # 50 # прокси-класс для Connection # ProxyConnection.java */*

```
package by.bsu.project.proxy;
import java.sql.Connection;
// other imports
class ProxyConnection implements Connection {
    private Connection connection;
    ProxyConnection(Connection connection) { // package private
        this.connection = connection;
    }
    @Override
    public Statement createStatement() throws SQLException {
        return connection.createStatement();
    }
    @Override
    public void close() throws SQLException {
        ConnectionPool.getInstance().put(this);
```

```
    }
    void closeConnection() throws SQLException { // package private
        connection.close();
    }
    @Override
    public void commit() throws SQLException {
        connection.commit();
    }
    @Override
    public boolean isClosed() throws SQLException {
        return connection.isClosed();
    }
    @Override
    public PreparedStatement prepareStatement(String sql) throws SQLException {
        return connection.prepareStatement(sql);
    }
    @Override
    public void rollback() throws SQLException {
        connection.rollback();
    }
    @Override
    public void setAutoCommit(boolean flag) throws SQLException {
        connection.setAutoCommit(flag);
    }
    // other override/delegate methods
}
```