3.3. Шаблоны проектирования и конструирования

«Пить Анна Петровна не хотела, но, повинуясь моему нескрываемому желанию ее напоить, чтобы она развязала мне руки для шаблонных мужских поступков, она велела заморозить две бутылки шампанского». ***А.С.Бухов. «Шаблонный мужчина»***

«Марксизм – не догма, а руководство к действию». ***В.И.Ленин***

Шаблон проектирования (паттерн) – общепринятое решение часто встречающейся проблемы проектирования, повторимая архитектурная конструкция

Как всегда сущность заключается в деталях:

- уровень решения в шаблоне – решение может быть в виде определенной программной конструкции из классов и объектов - **объектно-ориентированный шаблон**, либо состоять из более крупных аппаратно-программных компонент – **архитектурный шаблон.** Можно интерпретировать это как шаблоны конструирования и проектирования;
- шаблон конструирования может быть проиллюстрирован UML-диаграммой классов (объектов);
- шаблон не следует понимать буквально как некоторую заготовку, воспроизводящую предложенную в образце структуру. В основе шаблона лежит идея, которая может по-разному преломляться для разных задач. И даже сама идея тоже может быть подвергнута критике и ревизии;
- несмотря на разнообразие шаблонов конструирования, в их основе лежит один и тот же инструментарий ООП: классы, объекты, интерфейсы, наследование, полиморфизм. Именно поэтому при поверхностном взгляде шаблоны смотрятся довольно однообразно: сущность шаблона заключена в идее и ее реализации, скрытой в поведении шаблона.

Базовые шаблоны

Делегирование

Некоторая функциональность основного класса выносится в отдельный класс, через ссылку на этот объект осуществляется **делегирование** функционала. Технологическое решение, позволяющее разгрузить основной класс и сделать программу более модульной. В делегируемом коде могут остаться обращения к данным ведущего класса, тогда делегатам придется передавать обратную ссылку на ведущий класс и в нем создавать методы для доступа к нужным данным.

Абстрактный суперкласс

Абстрактный класс играет две роли: интерфейс для группы производных классов как основа для многообразия последующих реализаций и общий функционал и структуры данных. Общий функционал использует полиморфные вызовы интерфейсных методов для *отложенного программирования* функционала в производном классе.

Маркерный интерфейс

Интерфейс, не содержащий методов, используется для обозначения наличия у класса какого-либо свойства, например, Serializable – стандартная сериализация. Наличие интерфейса у класса может быть проверено операцией instanceof.

Заместитель (Proxy)

Заместитель имеет тот же интерфейс, что и замещаемый класс, а также ссылку или возможность передачи сообщений замещаемому объекту (рис.3-17).

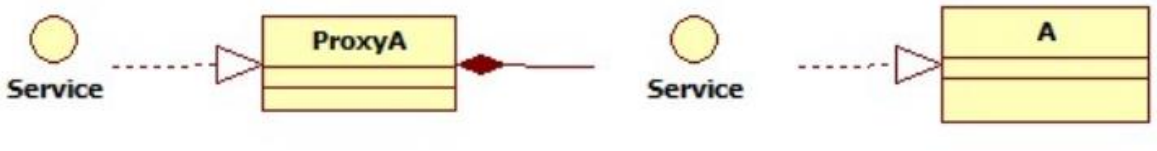


Рис.3-17. Паттерн – заместитель (proxy)

Заместитель изменяет поведение замещаемого объекта в различных аспектах:

- безопасность, ограничение доступа, буферизация и фильтрация данных;
- удаленный доступ по сети. Класс-заместитель передает сообщение серверу, который создает замещаемый объект и выполняет с ним указанные действия;
- метод замещаемого объекта выполняется асинхронно в потоке, объект-заместитель завершает метод, не дожидаясь его исполнения в заместителе.
- шаблон *virtual proxy*. Заместитель не создает замещаемого объекта до тех пор, пока в нем не возникнет реальная необходимость - ленивая инициализация.

Принципиальным для *Proxy* является идентичность интерфейса оригинального класса и его заместителя, в связи с чем заместитель и может выступать в роли оригинала везде, где используется оригинал.

Обратный вызов (CallBack)

Шаблон обратного вызова является альтернативой такой простой и базовой сущности как вызов метода в объекте. У обычного вызова есть несколько особенностей, которые не всегда приемлемы или удобны в использовании:

- вызов является синхронным и выполняется в едином потоке управления;
- результат вне зависимости от варианта завершения вызова передается через один и тот же тип результата, либо фиксируется в формальных параметрах вызова. Например, при отрицательном результате поиска вместо ссылки на объект возвращается *null*;
- сбой или ошибка при выполнении метода сопровождается исключением, которое требует отдельного кода обработки.

Такая жесткая схема иногда неудобна по следующим причинам:

- момент появления результата вызова может быть *асинхронным:* метод запускает поток и завершается, фактические результаты вызова являются результатом работы потока, т.е. имеет место внутренний параллелизм метода;
- метод возвращает *результат в виде множества объектов* и параметров. Можно, конечно, создавать специальный класс, интегрирующий все результаты, либо использовать несколько объектов-параметров, что не совсем удобно;

- исключения, порождаемые методом, можно обрабатывать внутри метода, создавая дополнительный вариант результата – ошибки метода;
- метод возвращает *результат в виде множества однотипных объектов***,** в самом методе происходит их накопление, а затем они возвращаются как одно целое, что приводит к большим промежуточным издержкам памяти;
- метод имеет *множество вариантов* завершения, которые вызывающий код должен обрабатывать.

Во всех случаях *прямой вызов метода* обработку результата в контексте вызывающего класса можно реализовать с использованием одного или нескольких методов **обратного вызова,** реализуемых через интерфейс.

Перечислим ряд технологических решений, для которых может использоваться обратный вызов:

- при обращении к сети непосредственно из GUI могут иметь место существенные задержки, блокирующие клиента, которые внешне воспринимаются как подвисание программы. При использовании обратного вызова метод, работающий с сетью, запускает поток, в котором выполняется необходимое взаимодействие, по окончании которого производится асинхронный обратный вызов.

  Замечание по теме. Необходима, как минимум, синхронизация обратного вызова, выполняемого в отдельном потоке, с основным потоком управления. Если основной поток является потоком GUI, то эта синхронизация делается стандартными статическими методами. Кроме того, необходимо заблокировать возможность повторного прямого вызова этого же метода, пока не завершился текущий, а также разнести данные, с которыми работает основной поток и поток в вызове, либо синхронизировать работу с ними;
- вызываемый метод обращается к БД, от которой в цикле получает выбранные записей, которые должны обрабатываться вызывающим объектом. В этом случае *каждую запись можно возвращать в виде обратного вызова*, а по завершении цикла выполнять еще один обратный вызов специального *метода завершения* в том же интерфейсе;
- при вызове метода возможны *различные варианты ответа*, которые можно реализовать в виде *группы обратных вызовов* в общем интерфейсе. Тогда вызывающему коду не потребуется их повторная селекция;
- при использовании обратного вызова обработка исключений может быть реализована в два этапа – в вызываемом методе при помощи обычного перехвата исключения, а затем в *дополнительном методе обратного вызова*;
- если метод переопределен в группе классов, а в некоторых из них требуется использование дополнительных параметров**,** то эти параметры можно не передавать в общем списке, а запрашивать в нужный момент методами обратного вызова из контекста вызывающего объекта. Обратный интерфейс с методами запроса параметров можно сделать *производным от базового интерфейса*. Классы, принимающие параметры, будут выполнять у себя явное сужение до требуемого интерфейса, а уже затем забирать через него нужные параметры;

Компоненты шаблона изображены на рис. 3-18, их поведение - на рис. 3-19. Класс-клиент создает объект-адаптер с интерфейсом обратного вызова и передает его классу-сервису до начала взаимодействия. Объект-адаптер работает в контексте класса-клиента, т.е. ему доступны его данные и методы, при необходимости он должен выполняться в том же потоке, что и класс-клиент. При исполнении метода в классе-

сервисе и возникновении событий, требующих участия клиента, через интерфейс обратного вызова вызываются методы в объекте-адаптере.

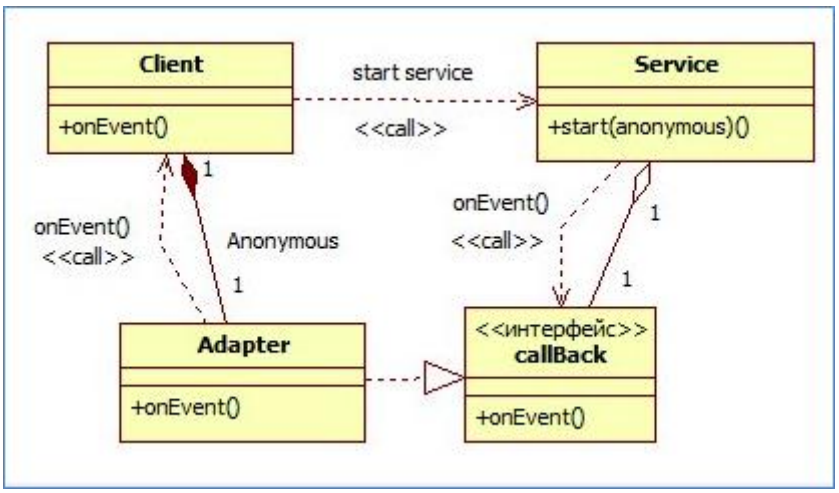


Рис.3-18. Диаграмма классов обратного вызова

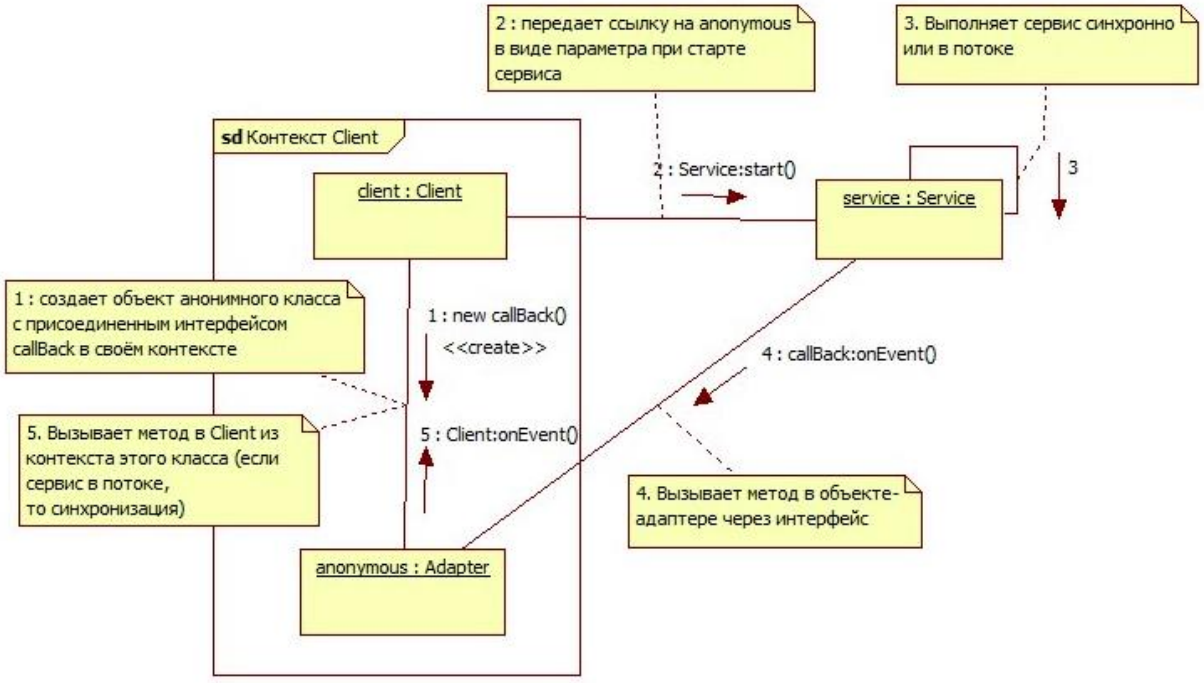


Рис.3-19. Коммуникационная диаграмма обратного вызова

В качестве примера рассмотрим, как выглядит на Java возврат последовательности сгенерированных методом данных через интерфейс обратного вызова (рис.3-20, 3-21):

- создаются интерфейсы, в которых объявляется CallBack-метод с параметром – очередным элементом данных;
- объект класса, в котором реализуется метод генерации, получает в качестве параметра ссылку на объект-слушатель с присоединенным интерфейсом обратного вызова;

```
public interface LambdaCallBack {       // Функциональный интерфейс
    public void onDataElement(Integer oo);   // событие – элемент последовательности
    }
public interface DataSequenceCallBack extends LambdaCallBack{
    public void onBeginSequence();       // событие – начало последовательности
    public void onEndSequence();         // событие – окончание последовательности
    }
public class RandomSequenceProducer {
    public void produce(int width, int count, DataSequenceCallBack back){
        back.onBeginSequence();          // событие - начало
        produceWithLambda(width, count, back);
        back.onEndSequence();            // событие - окончание
        }
    public void produceWithLambda(int width, int count,LambdaCallBack lBack){
        for(int i=0;i<count;i++){
            int val = (int)(Math.random()*2*width - width);
            lBack.onDataElement(new Integer(val));
            }
        }}
```

Рис.3-20. Генерация последовательности с передачей обратным вызовом

- при вызове метода в объекте-делегате создается объект-адаптер обратного вызова на основе анонимного класса и передается объекту-делегату. Объект-делегат вызывает переопределенные методы в объекте-адаптере в контексте вызывающего класса.
- интерфейс обратного вызова с единственным методом может быть реализован с использованием lambda-выражения – анонимной функции.

```
public class DataSequenceSum {
    private int sum=0, cnt=0;
    public void process(int width0, int count0){
        RandomSequenceProducer xx = new RandomSequenceProducer();
        xx.produce(width0, count0, new DataSequenceCallBack(){
            @Override
            public void onBeginSequence() {
                sum=0;cnt=0;
                }
            @Override
            public void onEndSequence() {
                System.out.println("count="+cnt+" middle="+sum/cnt);
                }
            @Override
            public void onDataElement(Integer w) {
                if (w>0){ sum+=w; cnt++; }
                }
            });
        sum=0; cnt=0;
        xx.produceWithLambda(width0, count0,w -> {
                if (w>0){ sum += w; cnt++; }
            });
        System.out.println("count="+cnt+" middle="+sum/cnt);
        }
```

Рис.3-21. Обработка обратного вызова в клиенте

Логику группового обратного вызова иллюстрирует UML-диаграмма взаимодействия (рис.3-22).

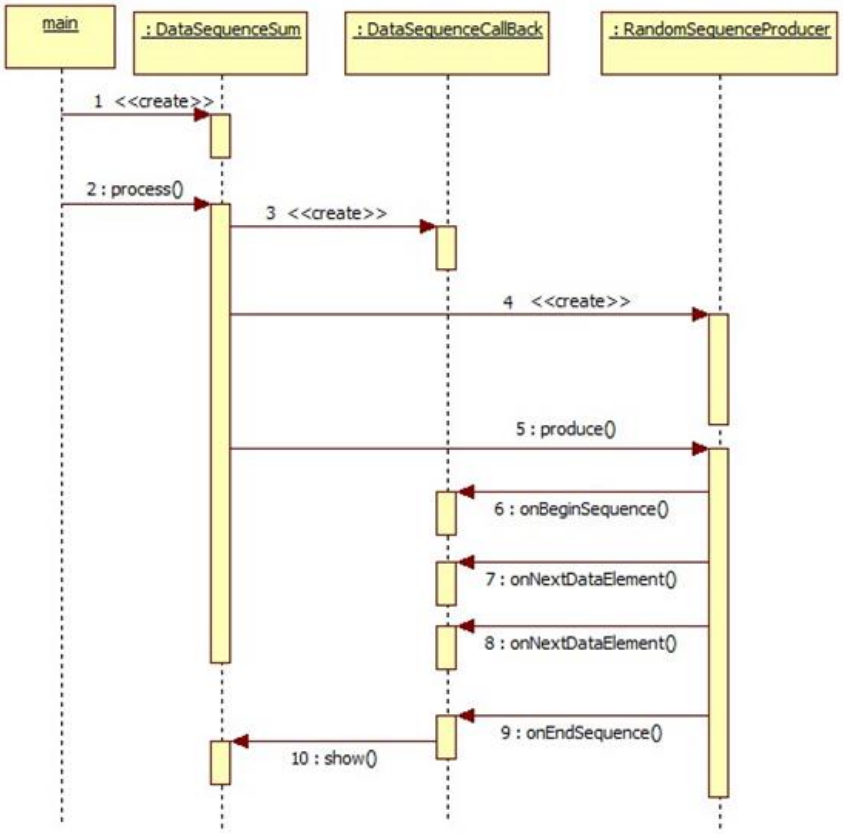


Рис.3-22. Диаграмма взаимодействий для группы обратных вызовов

Порождающие шаблоны

Фабрика (Factory)

Универсальность программы определяется, в том числе, и возможностью создавать объекты, принадлежащие к одной абстракции, но чтобы конкретный класс объекта определялся динамически, т.е. в зависимости от обстоятельств. Для этой цели и существуют классы – фабрики.

Метод – «фабрика» (Factory Method)

Имеется группа родственных классов на основе абстрактного базового класса или интерфейса. Метод-фабрика создает и возвращает объект одного из классов, руководствуясь своими параметрами, например, по типу файла, передаваемого в имени, либо имеет набор статических методов, возвращающих объекты разных классов.

Абстрактная фабрика (Abstract Factory)

Иногда требуется создать не отдельный объект, а группу объектов, каждый из которых является конкретизацией отдельной абстракции. Например, при создании отдельного стиля оформления оконного приложения классы окна и кнопки базируются на абстракциях IWindow и IButton (рис.3-23). Интерфейс фабрики AbstractFactory содержит методы для порождения объектов класса окна и кнопки. Конкретные фабрики

для разных стилей оконного приложения создают каждая свой набор объектов, соответствующих стилю.

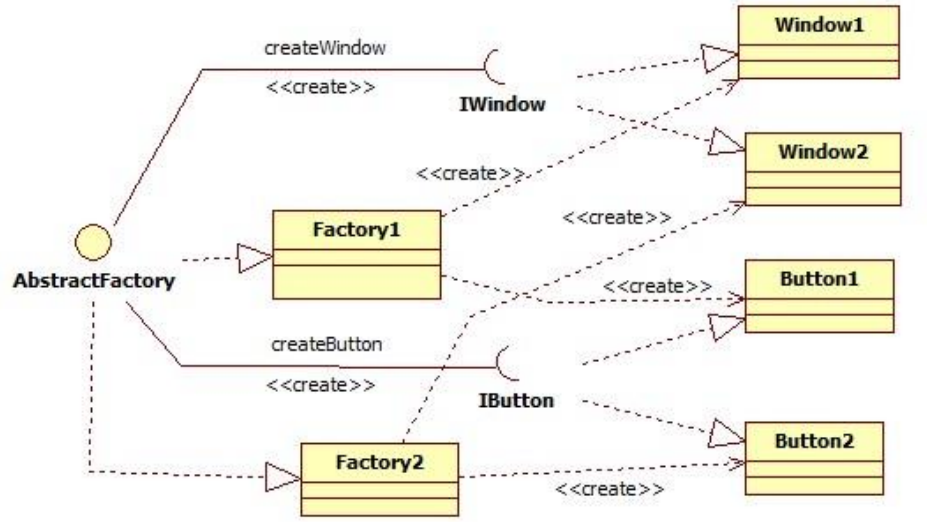


Рис.3-23. Абстрактная фабрика для классов оконного приложения

Фабрика должна обеспечивать возможность создания объектов из некоторого разнообразия, на данный момент *зашитого* в программе. При этом желательно, чтобы перечень доступных типов был также доступен во внешнем представлении. В качестве примера рассмотрим вариант реализации, в котором используется шаблон для создания фабрики для всех классов, принадлежащих указанному пакету и имеющими заданный присоединенный интерфейс. Во внешнем представлении может использоваться аналогичный шаблон для элемента управления – списка, который базируется на объектах соответствующей фабрики.

Определены интерфейсы для *именованных* объектов и типов, а также шаблон-интерфейс для фабрики именованных объектов (рис.3-24). Фабрика содержит вектор именованных объектов и методы получения списка имен объектов и выбора объекта по имени. С помощью библиотеки org.reflections метод generate просматривает список классов указанного пакета и создает для них объекты-прототипы, заполняя ими вектор.

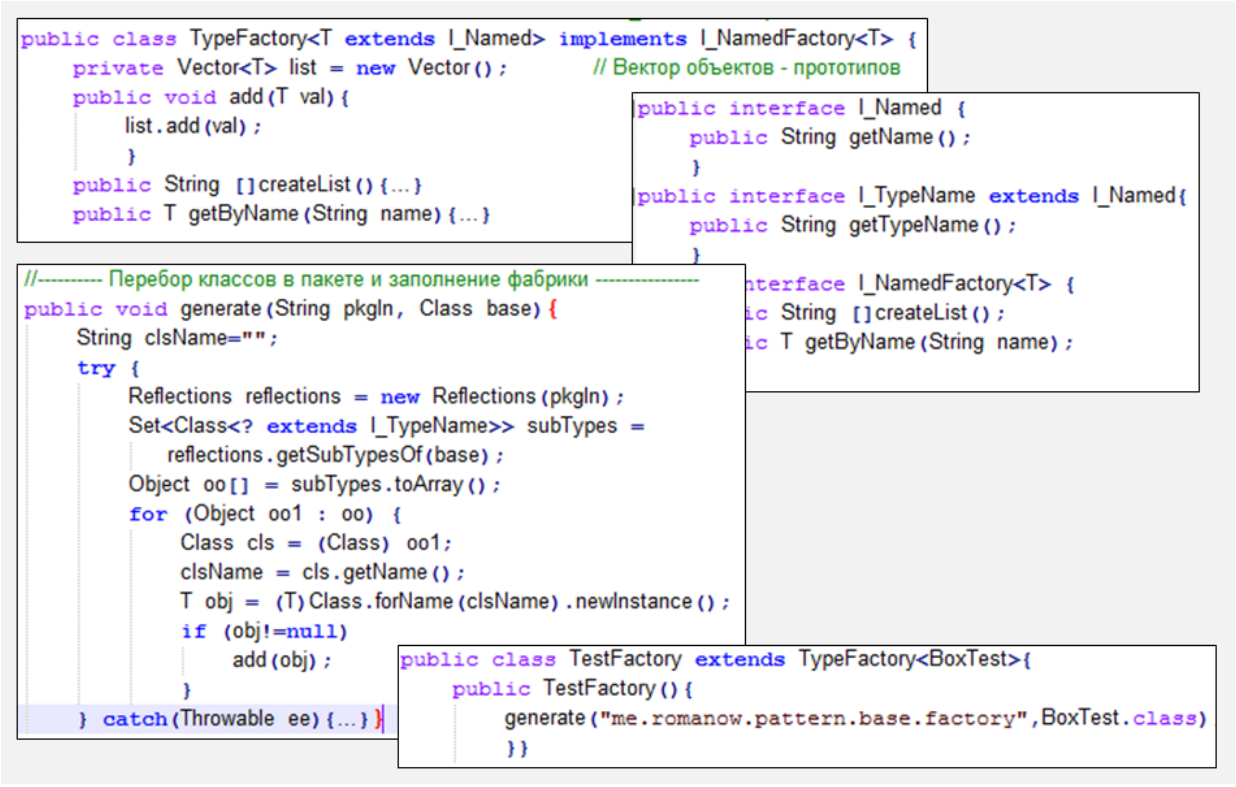


Рис.3-24. Фабрика для производных классов в указанном пакете

Для связывания фабрики с элементом управления – выпадающим списком, используется класс-шаблон BoxFactory с параметром – базовым классом элементов фабрики. Он получает фабрику, по списку именованных типов заполняет выпадающий список. Перехватывает событие выбора элемента и выбирает из фабрики объект по соответствующему имени (рис.3-25).

Для того, чтобы связать выпадающий список в форме (ComboTest) с многообразием на основе класса BoxTest, необходимо создать объект BoxFactory, передав ему фабрику, выпадающий список и адаптер для обратного вызова при выборе элемента списка на основе вложенного класса BoxFactory. Метод getSelected возвращает объект, выбранный из фабрики, соответствующий выбранному элементу списка.

```
public class BoxFactory<T extends I_Named>{
    public interface BoxFactoryCallBack<T>{
        public void getSelected(T selectedItem);
        }
    private I_NamedFactory<T> factory;
    private JComboBox box;
    public T getSelected(){
        return factory.getByName(box.getSelectedItem().toString());
        }
    public BoxFactory(I_NamedFactory<T> factory,JComboBox box,final BoxFactoryCallBack<T> back){
        this.box = box;
        this.factory = factory;
        box.removeAllItems();
        String names[] = factory.createList();
        box.addItem("...");
        for(int i=0;i<names.length;i++)
            box.addItem(names[i]);
        final I_NamedFactory<T> factory1 = factory;
        box.addItemListener(new java.awt.event.ItemListener() {
            public void itemStateChanged(java.awt.event.ItemEvent evt) {
                String ss = (String)evt.getItem();
                T w = factory1.getByName(ss);
                if (w==null)
                    return;
                back.getSelected(w);
                }
            });
        }
```

```
BoxFactory bx = new BoxFactory(new TestFactory(),ComboTest,
new BoxFactory.BoxFactoryCallBack<BoxTest>(){
    @Override
    public void getSelected(BoxTest selectedItem) {
        System.out.println(selectedItem.getTypeName());
        }
    });
```

Рис.3-25. Связывание фабрики с выпадающим списком

Строитель (Builder)

Шаблон аналогичен фабрике, только создает не отдельный класс, а систему из объекта основного класса и связанных с ним объектов других классов. Кроме того, в базовом (абстрактном) классе имеется статический метод, который создает объект производного класса, руководствуясь параметрами вызова.

Прототип (Prototype)

Группа родственных классов имеет в общем интерфейсе метод копирования (*клонирования*), программа создает копию объекта-прототипа по ссылке на объект-оригинал через указанный интерфейс. Класс объекта-прототипа и класс создаваемой копии формально программе неизвестны, т.е. могут быть произвольными. В Java на примитивном уровне можно использовать интерфейс *clone* или средства рефлексии - метод *Class.newInstance*. Описанный выше шаблон *фабрика* может использовать коллекцию объектов-прототипов для создания клонов, используемых программой.

Синглетон (*Singleton*)

Обеспечивает единственность экземпляра объекта некоторого класса. В отличие от статической ссылки на объект, он создается при первом доступе к объекту и закрыт для доступа по ссылке. Фактически использует ленивую инициализацию: создает объект при первом обращении к нему через статический метод (рис.3-26).

```
public class Singleton {
    private static Singleton one=null;
    private Singleton(){ count=0; }
    public synchronized static Singleton getInstance(){
        if (one==null){
            one=new Singleton();
            }
        return one;
        }
```

Рис.3-26. Синглетон

Синглетон может использоваться для хранения общего контекста приложения.

Пул объектов (Object Pool)

Программа может иметь ограничение на создание объектов определенного класса, например, соединений с БД, сетевых соединений, открытых файлов. Также могут быть существенные временные и ресурсные затраты на создание и утилизацию объектов. В этих случаях используется пул объектов: свободные находятся в пуле, выделяются по требованию, по окончании использования не утилизируются, а возвращаются в пул. Варианты реализации:

- предварительная генерация объектов в пуле при его создании, либо создание их требованию при отсутствии в пуле свободных;
- хранение свободных и выделенных, либо только свободных. В последнем случае пул не контролирует корректность работы программы с пулом;
- наличие или отсутствие ограничения на размер пула;
- действие при отсутствии свободного: создание дополнительного объекта или блокировка запрашивающего потока до появления свободного (см. шаблоны параллелизма);
- действие при освобождении: помещение в буферный пул, либо потеря ссылки с последующей утилизацией для сокращения размера пула при наличии ограничений.

Структурные шаблоны

Фильтр (Filter)

Класс имеет тот же самый интерфейс, что и замещаемый. Фильтр получает данные из замещаемого (делегируемого) класса и производит их преобразование, возвращая отфильтрованные данные через собственный интерфейс. Шаблон имеет много общего с заместителем (proxy), но есть существенное различие: он не расширяет функциональность при том же интерфейсе, а удаляет *лишние* данные из потока.

Адаптер (Adapter)

Вариант 1. Требуется реализовать в классе клиента некоторый интерфейс, например, обратный вызов по асинхронному событию. По технологическим соображениям этот интерфейс нельзя навесить на класс – клиент. Например, есть несколько источников событий с таким интерфейсом, либо основной класс уже достаточно нагружен присоединенными интерфейсами. Тогда создается класс-адаптер, который присоединяет к себе указанный интерфейс, и при этом *видит* класса-клиента. Способы *видения* могут быть разными:

- конструктор получает ссылку на объект класса-клиента;
- класс адаптера является вложенным;
- класс адаптера является анонимным;
- для функционального интерфейса с единственным методом адаптером является lambda-выражение - анонимная функция.

Вариант 2. Класс, который обеспечивает сопряжение интерфейсов, незначительно отличающихся друг от друга. Класс-адаптер с целевым интерфейсом получает ссылку на объект с исходным интерфейсом и преобразует обращения к целевому интерфейсу в обращения к исходному.

Композиция (Composite)

Шаблон моделирует древовидная структуру объектов. Общий интерфейс или абстрактный класс *Node* предполагает несколько частных реализаций однокомпонентных классов, а также производный от него класс *NodeGroup*, содержащий вектор ссылок на вложенные компоненты класса, имеющих тип *Node* (рис.3-27). Рекурсивный характер древовидной структуры может быть отражен на диаграмме классов в явном виде, либо условно в виде рефлексивного замыкания класса Node в форме композиции с произвольным количеством потомков.

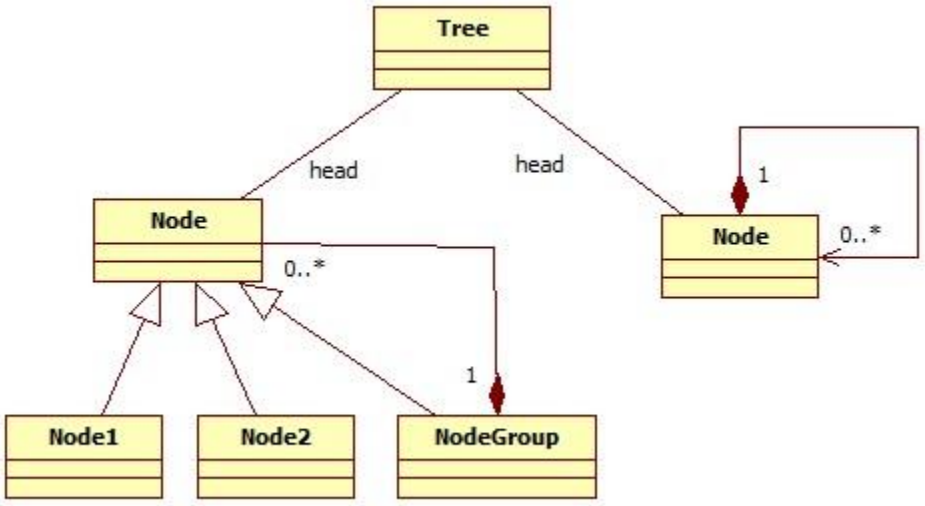


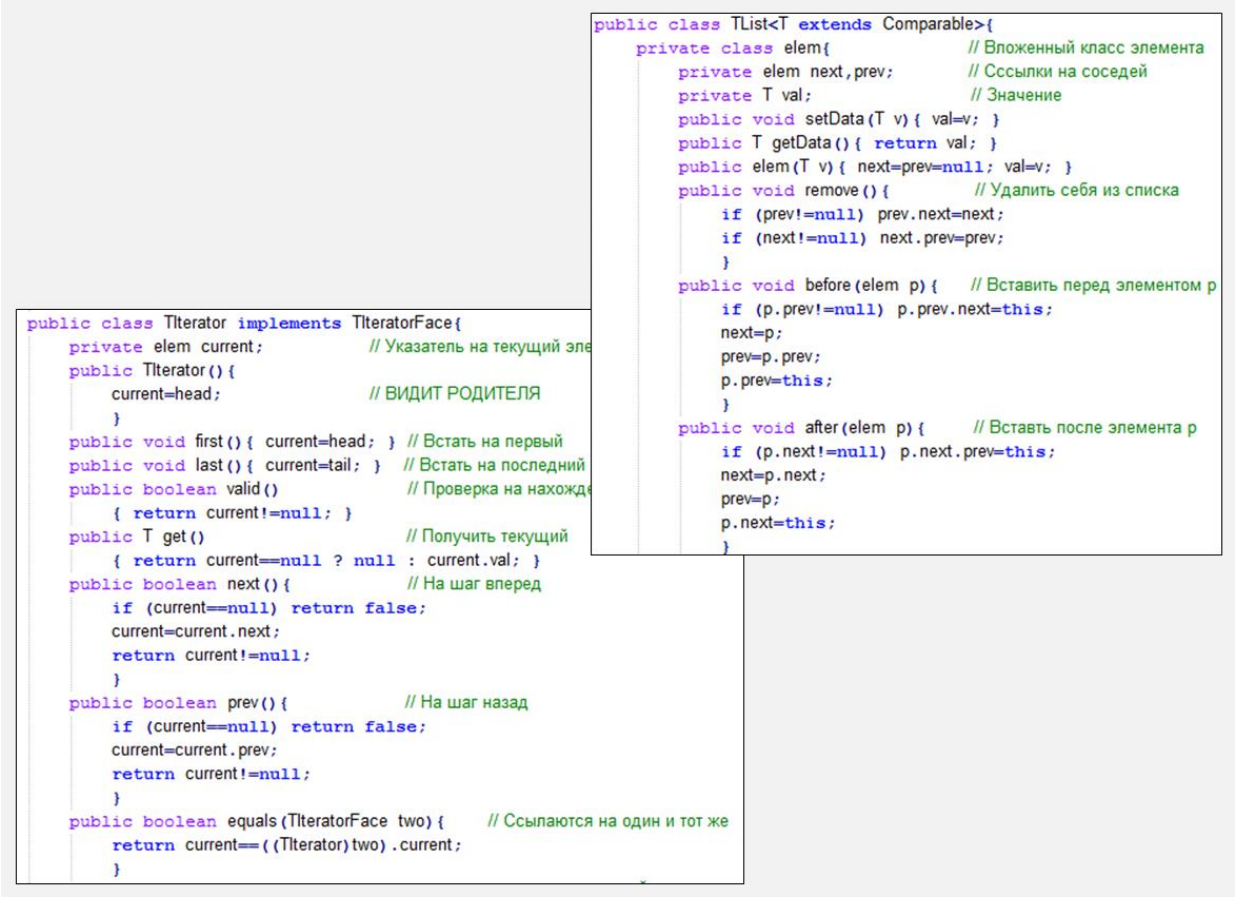
Рис.3-27. Шаблон древовидной структуры - композиция

Итератор (Iterator)

Класс-движок по элементам структуры данных, обеспечивающий интерфейс последовательного перемещения по ее элементам: перемещение в начало, конец, вперед и назад, доступ к текущему элементу. Класс итератора является вложенным в класс структуры данных. Этим обеспечивается *видимость* итератором всей структуры данных, возможность позиционироваться на начало и конец.

Далее приведен пример итератора для структуры данных – двусвязного списка в виде шаблона с параметром – классом, имеющим присоединенный интерфейс сравнения *Comparable*. Класс элемента структуры данных - *элемента списка* и самого итератора вложены в основной класс. Вложенность класса итератора обеспечивает *видимость* ими данных своего родителя. Элемент списка выполняет элементарные манипуляции в над собой в списке – удаление и вставка перед и после указанного. Итератор содержит

ссылку на текущий элемент и манипулирует ей, а также возвращает ссылку на объект данных указуемого элемента (рис.3-28).



```
public class TList<T extends Comparable>{
    private class elem{                  // Вложенный класс элемента
        private elem next,prev;          // Ссылки на соседей
        private T val;                   // Значение
        public void setData(T v){ val=v; }
        public T getData(){ return val; }
        public elem(T v){ next=prev=null; val=v; }
        public void remove(){            // Удалить себя из списка
            if (prev!=null) prev.next=next;
            if (next!=null) next.prev=prev;
            }
        public void before(elem p){      // Вставить перед элементом p
            if (p.prev!=null) p.prev.next=this;
            next=p;
            prev=p.prev;
            p.prev=this;
            }
        public void after(elem p){       // Вставть после элемента p
            if (p.next!=null) p.next.prev=this;
            next=p.next;
            prev=p;
            p.next=this;
            }
```

```
public class TIterator implements TIteratorFace{
    private elem current;            // Указатель на текущий эле
    public TIterator(){
        current=head;                // ВИДИТ РОДИТЕЛЯ
        }
    public void first(){ current=head; } // Встать на первый
    public void last(){ current=tail; }  // Встать на последний
    public boolean valid()               // Проверка на нахожде
        { return current!=null; }
    public T get()                       // Получить текущий
        { return current==null ? null : current.val; }
    public boolean next(){               // На шаг вперед
        if (current==null) return false;
        current=current.next;
        return current!=null;
        }
    public boolean prev(){               // На шаг назад
        if (current==null) return false;
        current=current.prev;
        return current!=null;
        }
    public boolean equals(TIteratorFace two){      // Ссылаются на один и тот же
        return current==((TIterator)two).current;
        }
```

Рис.3-28. Итератор: вложенные классы элемента списка и итератора

Объект-итератор, будучи вложенным классом, может быть создан только в контексте родителя (метод *create* на рис.3-29). Класс списка содержит операции добавления элементов, упорядоченной вставки, а также метод *forEach*, исполняющий для всех элементов списка действие, специфицированное как *toDo* в интерфейсе обратного вызова *TActionFace*. При его вызове может быть использовано лямбда-выражение.

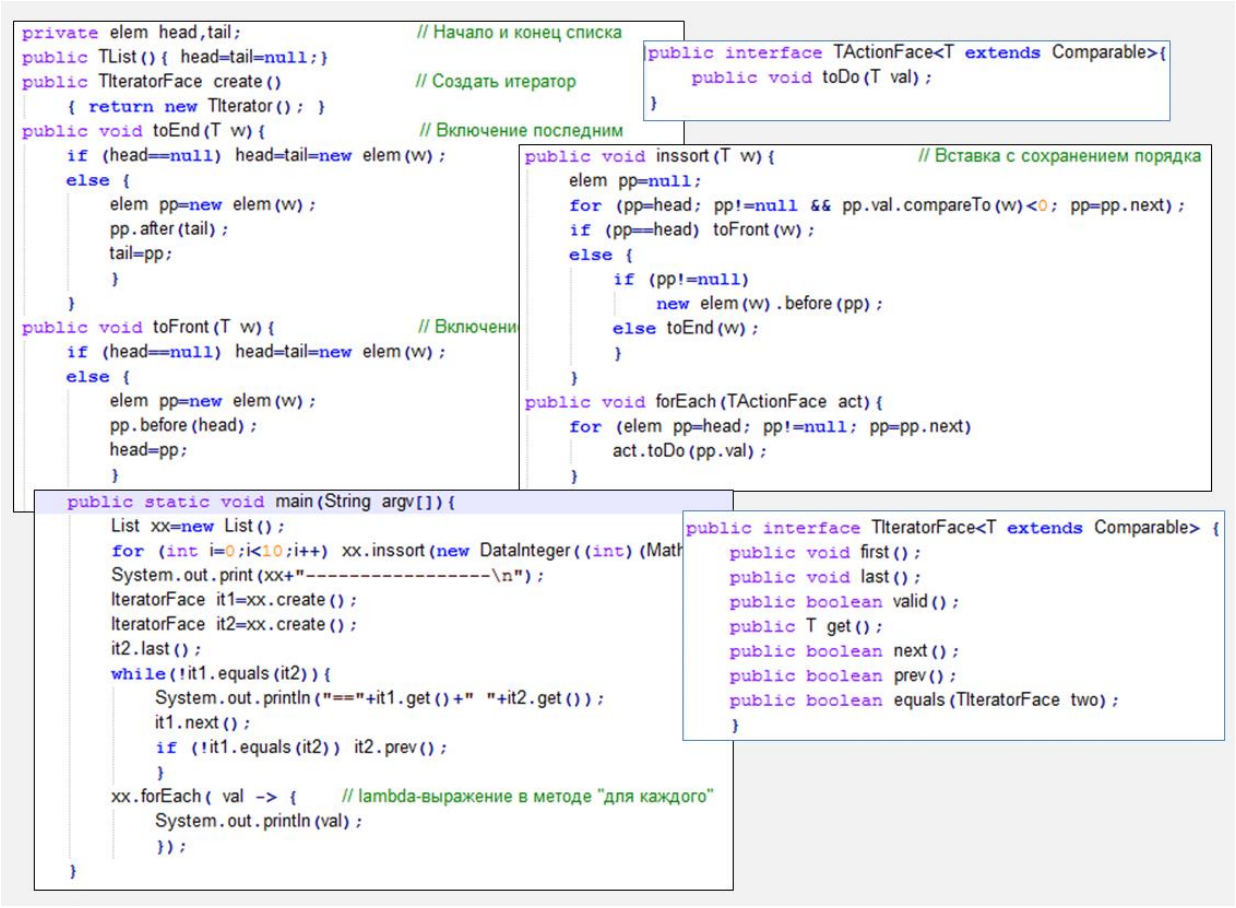


Рис.3-29. Итератор: Класс списка и пример работы

Мост (Bridge)

Вариант 1. Используется при наличии двух многообразий связываемых компонент. Например, несколько таблиц выводятся в файлы нескольких форматов. Группа классов первого многообразия делегируется к объекту группы классов второго через интерфейс (рис.3-30).

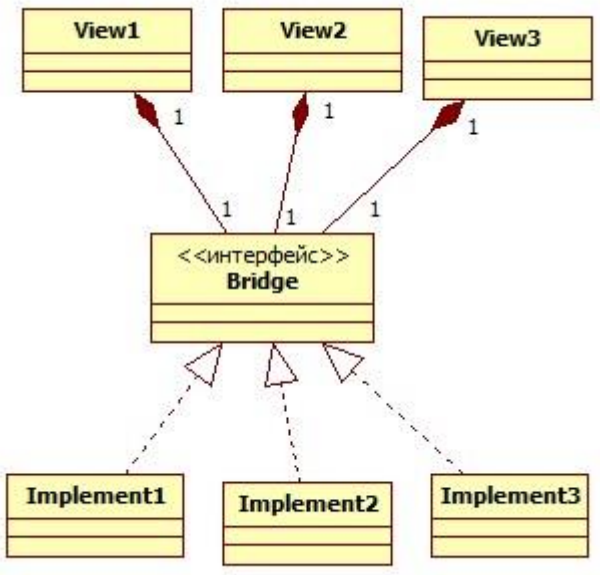


Рис.3-30. Шаблон *мост* для компоновки двух многообразий

Вариант 2. Имеется группа связанных классов логического представления и группа классов физического уровня - реализации. Базовый класс логического представления делегируется к объекту физической реализации через интерфейс или абстрактный класс (рис.3-31).

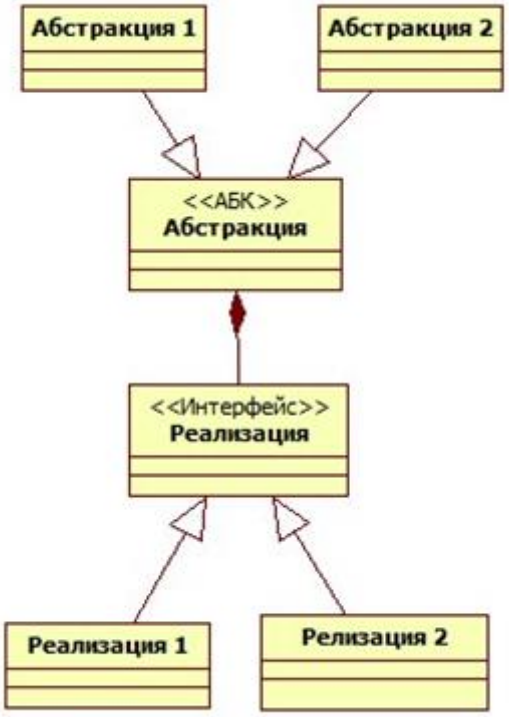


Рис.3-31. Шаблон *мост* для независимого логического и физического уровня

Вариант 3. Группа классов логического уровня и аналогичные группы классов физического уровня – реализаций связаны по одной схеме. Связывание логического и физического уровней производится через интерфейсы (рис.3-32).

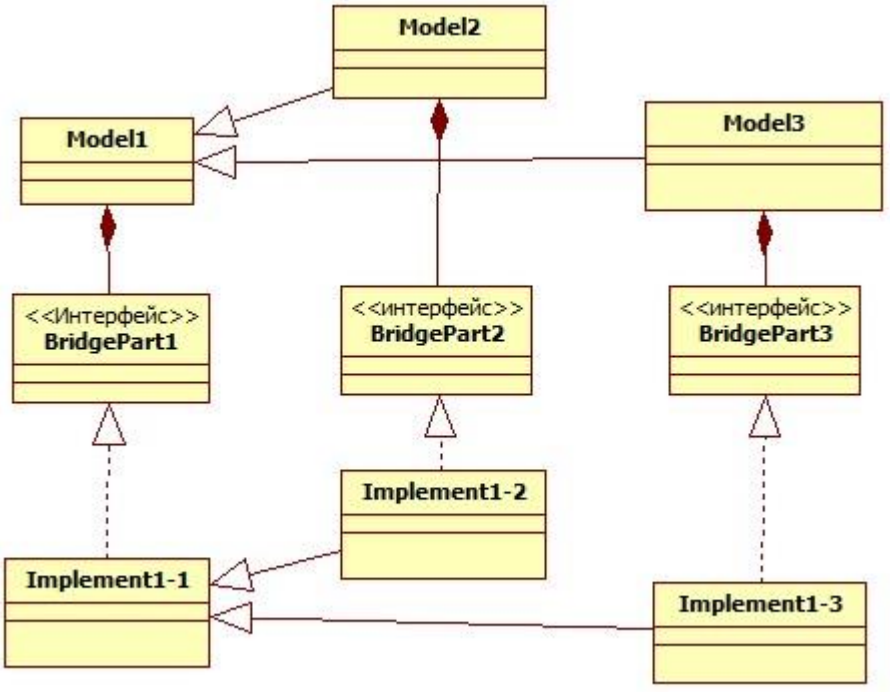


Рис.3-32. Шаблон *мост* для связывания групп классов

Фасад (*Facade*)

Технологическое решение. Вместо того, чтобы хранить набор ссылок на требуемые объекты, они переносятся в промежуточный класс, обычно оставаясь открытыми. Появляется возможность использовать фасад в других компонентах как единое целое, а не передавать все ссылки. Кроме того, в фасаде можно реализовать методы, данные для которых локализованы в фасаде.

Декоратор (Decorator)

Расширение функциональности исходного класса. Класс-декоратор присоединяет интерфейс исходного класса, а также имеет собственную функциональность. В отличие от наследования, он получает или создает объект исходного класса, к которому делегирует старую функциональность, а затем дополняет ее.

Динамическое связывание (загрузка, компоновка) (*Dynamic Linkage*)

В Java благодаря рефлексии имеется возможность динамической загрузки классов при помощи класса *ClassLoader*. Естественно, загружаемый класс должен иметь оговоренный интерфейс использования, поддерживаемый программой.

Приспособленец (Flyweight)

Вариант 1. Имеет место несколько представлений содержимого одного объекта (рис. 3-33). На объект делаются ссылки со стороны нескольких объектов-приспособленцев, каждый класс которых обеспечивает оригинальную интерпретацию содержимого.

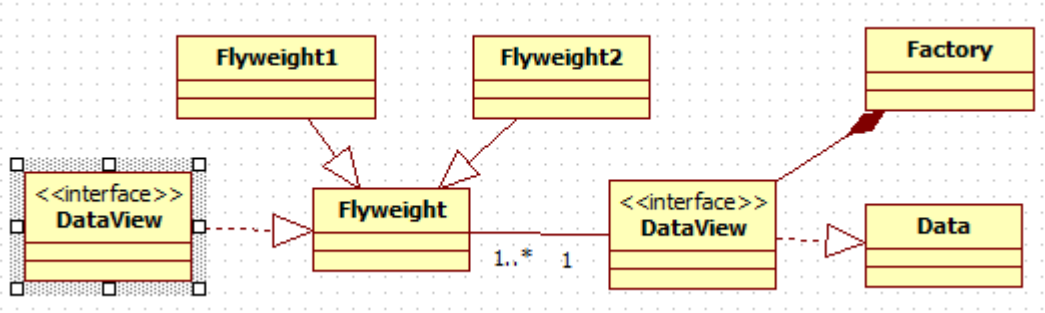


Рис.3-33. Множественность представлений объекта через приспособленцев

Вариант 2. Обеспечивается эффективное разделение объектов данных с одинаковым содержимым путем раздачи ссылок и контроля изменения объектов (рис.3.34).

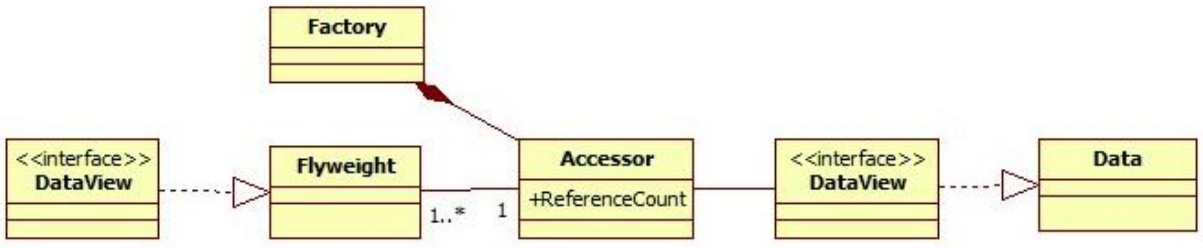


Рис.3-34. Шаблон *приспособленец* для разделения данных объекта

Например, имеется текст или набор версий текста с повторяющимися словами. Ссылки на объекты данных делаются не прямо, а через объекты доступа, которые содержат счетчики ссылок. Класс фабрики контролирует все объекты доступа и через них – объекты данных. В пользовательском коде доступ к объекту данных осуществляется через объект-приспособленец. При попытке изменить значение объекта

данных через приспособленца он может быть изменен только при наличии единственной ссылки. Иначе фабрика создает новый объект данных (или ищет такой же), создает объект доступа к нему и настраивает на него приспособленца. В старом объекте доступа счетчик ссылок уменьшается. При удалении объекта удаляется не сам объект, а приспособленец, а в объекте доступа счетчик ссылок уменьшается. Объект данных удаляется, если счетчик ссылок становится равным 0.

Аналогичное решение на основе раздачи и контроля ссылок используется для контроля целостности данных и автоматической утилизации объектов. Например, в ядре Windows таким образом проверяется возможность безопасного удаления объекта ядра и освобождения памяти.

Управление кэшем (Cashe Management)

Класс-менеджер объектов контролирует процесс получения (загрузки) объектов и сохраняет их копии (или ссылки) в кэше. При повторном обращении к тому же объекту по идентификатору он повторно извлекается (копируется) из кэша. Размер кэша ограничен. Для удаления из кэша *лишних* объектов при его переполнении в процессе загрузки новых известны *стратегии вытеснения*, например FIFO, LRU.

Объект-значение (Value-Object)

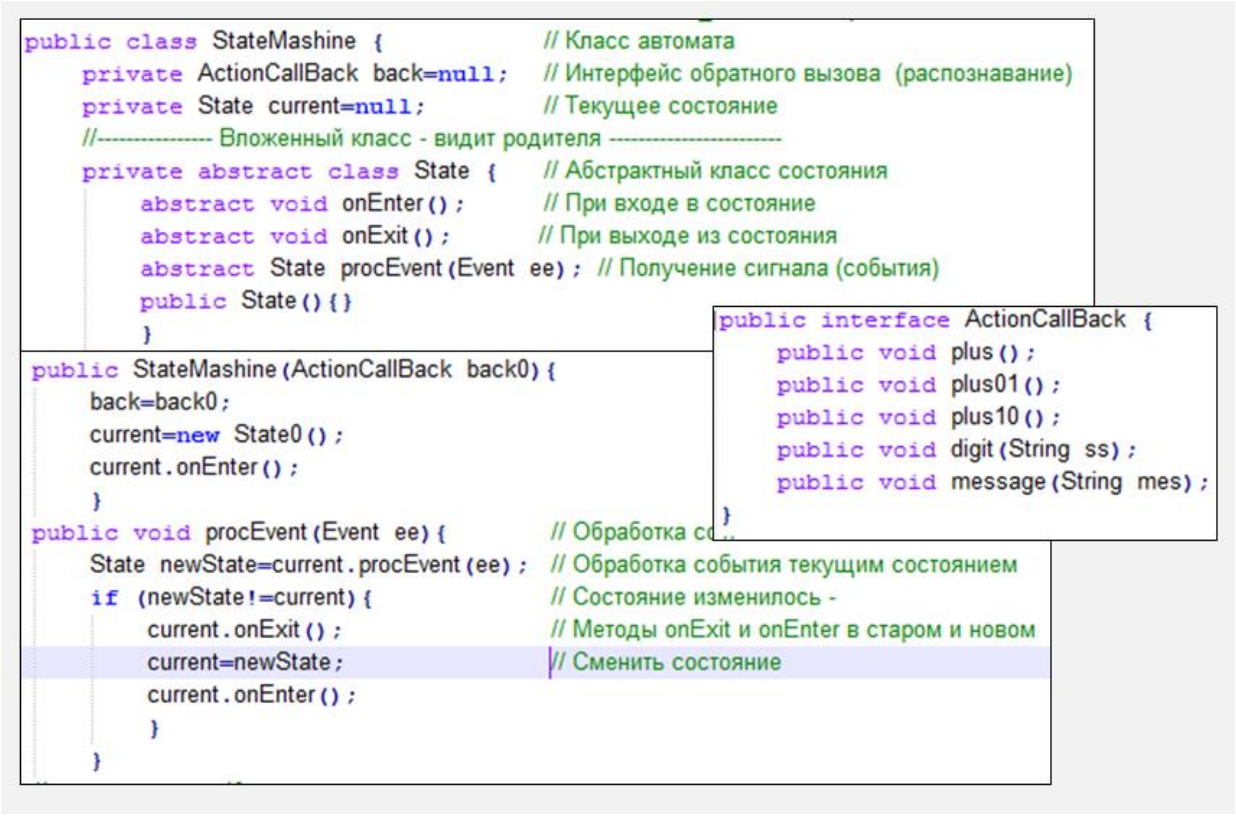
Шаблон обеспечивает передачу результата операции в виде объекта-копии, т.е. не в виде ссылки, а по значению. Значение оригинала при этом не меняется. В Си++ аналогом является передача объекта в метод и возвращение объекта-результата *по значению.* Для реализации необходим конструктор копирования, фактически реализация паттерна объект-значение, который создает копию объекта с клонированием всех его ссылок.

Поведенческие шаблоны

Конечный автомат *(State)*

Конечные автоматы широко используются для интеграции поведения в системах, управляемых событиями. Для его реализации можно воспользоваться определением конечного автомата – *таблицей состояний-переходов,* что плохо не вписывается в ООП-нотацию. Для реализации автомата на принципах ООП создается абстрактный базовый класс состояния автомата, в который закладывается каркас его поведения. Каждому состоянию соответствует производный класс, который описывает поведение автомата в этом состоянии. Для вывода сгенерированных управляющих воздействий используется интерфейс обратного вызова. Классы состояний автомата – вложенные, чтобы видеть данные класса-автомата. Он имеет ссылку на объект – текущее состояние, обработка события в текущем состоянии сопровождается созданием объекта – нового состояния.

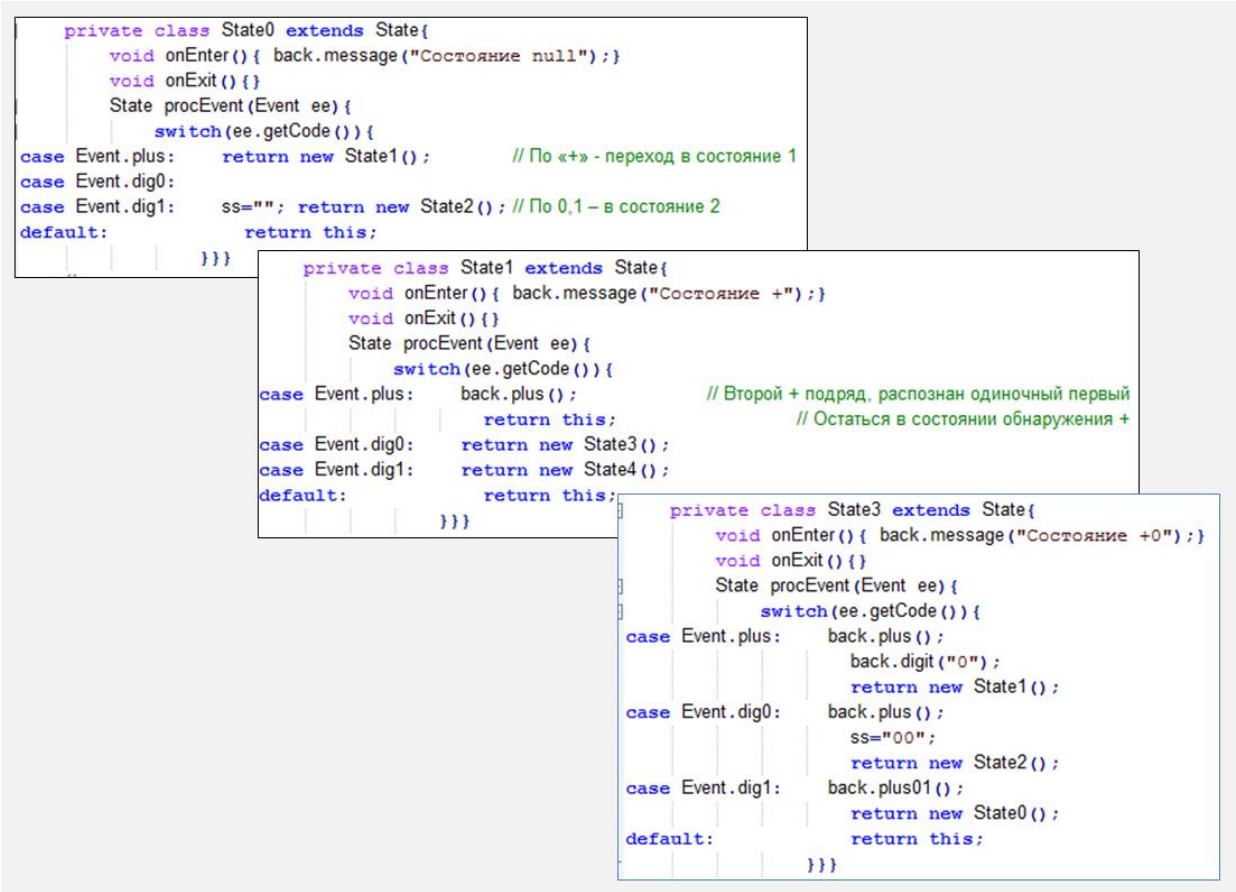
В качестве примера рассмотрим лексический анализатор, выделяющий из последовательности символов {+,0,1} цепочки +01,+10,+,<другая последовательность 0,1> (рис.3.35,3-36).



```
public class StateMashine {                 // Класс автомата
    private ActionCallBack back=null;       // Интерфейс обратного вызова (распознавание)
    private State current=null;             // Текущее состояние
    //---------- Вложенный класс - видит родителя ----------
    private abstract class State {          // Абстрактный класс состояния
        abstract void onEnter();            // При входе в состояние
        abstract void onExit();             // При выходе из состояния
        abstract State procEvent(Event ee); // Получение сигнала (события)
        public State(){}
        }
public StateMashine(ActionCallBack back0){
    back=back0;
    current=new State0();
    current.onEnter();
    }
public void procEvent(Event ee){            // Обработка со
    State newState=current.procEvent(ee);   // Обработка события текущим состоянием
    if (newState!=current){                 // Состояние изменилось -
        current.onExit();                   // Методы onExit и onEnter в старом и новом
        current=newState;                   // Сменить состояние
        current.onEnter();
        }
    }
```

```
public interface ActionCallBack {
    public void plus();
    public void plus01();
    public void plus10();
    public void digit(String ss);
    public void message(String mes);
}
```

Рис.3-35. Абстрактный класс состояния автомата и класс автомата



```
    private class State0 extends State{
        void onEnter(){ back.message("Состояние null");}
        void onExit(){}
        State procEvent(Event ee){
            switch(ee.getCode()){
case Event.plus:    return new State1();          // По «+» - переход в состояние 1
case Event.dig0:
case Event.dig1:    ss=""; return new State2(); // По 0,1 – в состояние 2
default:              return this;
                }}}
```

```
    private class State1 extends State{
        void onEnter(){ back.message("Состояние +");}
        void onExit(){}
        State procEvent(Event ee){
            switch(ee.getCode()){
case Event.plus:    back.plus();                  // Второй + подряд, распознан одиночный первый
                      return this;                // Остаться в состоянии обнаружения +
case Event.dig0:    return new State3();
case Event.dig1:    return new State4();
default:              return this;
                }}}
```

```
    private class State3 extends State{
        void onEnter(){ back.message("Состояние +0");}
        void onExit(){}
        State procEvent(Event ee){
            switch(ee.getCode()){
case Event.plus:    back.plus();
                      back.digit("0");
                      return new State1();
case Event.dig0:    back.plus();
                      ss="00";
                      return new State2();
case Event.dig1:    back.plus01();
                      return new State0();
default:              return this;
                }}}
```

Рис.3-36. Классы конкретных состояний

Моментальный снимок (SnapShot)

Шаблон обеспечивает сохранение текущего содержимого объекта и связанных с ним данных в объекте-снимке с возможностью последующего восстановления содержимого по снимку. В реализации шаблона может использоваться сериализация, либо коллекция именованных объектов для сохранения необходимых значений данных. Например, в ОС Android используется именованная коллекция *Bundle* для сохранения приложением собственных параметров состояния при таких событиях как поворот экрана, уход с переднего плана и т.п..

Последовательность команд (Command)

Шаблон поддерживает в группе команд обработки объекта сервис исполнения, отмены и восстановления отмененных действий *Do/ReDo/UnDo*. Производный класс запоминает параметры, необходимые для выполнения прямой и обратной команды. Класс – менеджер команд поддерживает очередь команд ограниченной длины, текущий обрабатываемый объект, методы *Do/ReDo/UnDo*, выбирая их из очереди и вызывая соответствующие методы в объектах-командах. Возможны варианты запоминания состояния объекта для исполнения команд:

- моментальный снимок структуры данных;
- генерация *обратной* команды. Например, при удалении из строки 5-го символа, которым в данный момент является «f», для команды UnDo генерируется команда вставки этого символа на 5-ую позицию.

Стратегия (Strategy)

Базовый класс реализует структуры данных и базовый функционал их обслуживания. Производные классы реализуют различные варианты решения задачи - алгоритмы, стратегии.

Метод шаблона (Template Method)

Аналог *абстрактного суперкласса***.** Абстрактный метод является дополнением основного алгоритма в базовом классе и работает по принципу *отложенного программирования* при реализации производного класса.

Встраиваемый объект (Pluggable Object)

Аналогичен *методу шаблона***.** При наличии небольших вариаций функциональности основной класс может получать при конструировании ссылку на объект-делегат, реализующий дополнительную функциональность.

Встраиваемый переключатель (Pluggable Selector)

Аналогичен *методу шаблона***,** но при зашитых вариантах функциональности в самом классе - наборе однотипных методов, которые вызываются *по имени с помощью рефлексии.*

Цепочка ответственности (Chain of Responsibility)

Объекты классов с подключенным интерфейсом обработки сообщения (события), образуют линейную или древовидную структуру данных. Основной класс последовательно вызывает метод обработки сообщения во всех объектах в порядке обхода структуры данных. Если один из объектов обработал сообщение, обход прекращается. В моделях систем контроля сообщения также могут передаваться вверх по

дереву иерархии от классов-источников (сенсоров) к классам элементов управления. Элемент управления реагирует на сообщение, либо пересылает его выше.

Наблюдатель (Observer)

Средства *широковещательной или селектируемой рассылки* сообщений между объектами группы классов с общим интерфейсом *Observer*. Объект *подписывается* на прием сообщений к классу-источнику или классу-диспетчеру, передавая интерфейс *Observer*. Класс широковещательной рассылки, получая или генерируя сообщение, рассылает его всем подписавшимся объектам. Наблюдатель также может быть источником сообщений, передавая его для рассылки классу-диспетчеру.

Маленький язык, интерпретатор (Little Language, Interpreter)

Интепретатор произвольного языка с универсальной или настраиваемой лексикой и синтаксисом.

Посредник (Mediator)

Класс, согласующий состояния связанной с ним группы объектов. Классы-клиенты не взаимодействуют между собой. Вместо этого они посылают посреднику сообщения. Посредник реализует общие принципы поведения системы и рассылает клиентам соответствующие команды (рис.3-37).

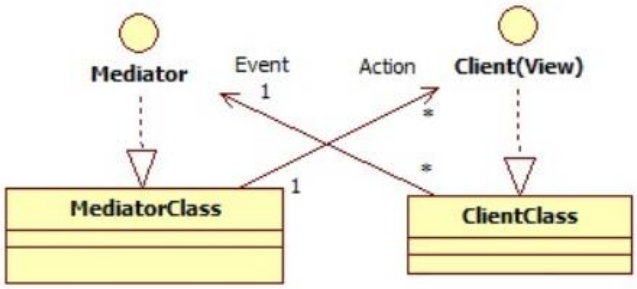


Рис.3-37. Шаблон *посредник*

Посетитель (Visitor)

Класс, выполняющий обход структуры данных и реализующий некоторый общий алгоритм для всех ее элементов.

Null-объект (NullObject)

В группе классов на основе интерфейса создается класс с *нулевой* функциональностью, замещающий *null*-ссылку. В качестве *null*-объекта может использоваться объект базового класса с *нулевой* функциональностью (рис. 3-38).

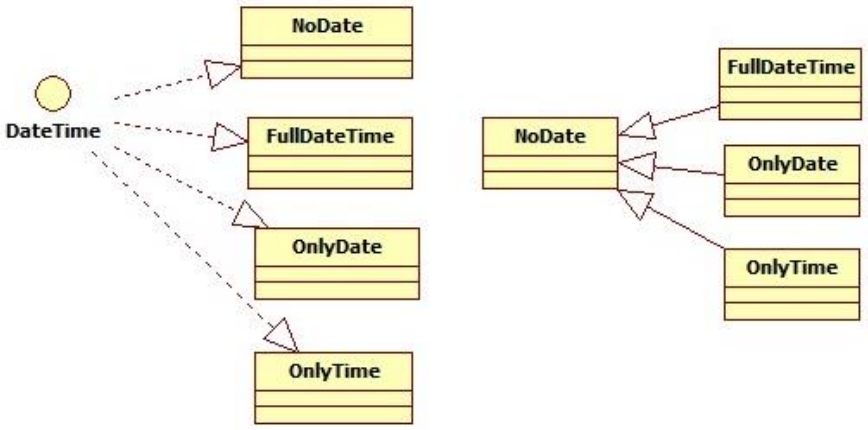


Рис.3-38. Варианты создания *null*-объекта

Возможно другое общее решение проблемы – создание контейнера-шаблона, который для объекта любого типа хранит ссылку на объект и логическую переменную состояния, а также ссылку на объект-исключение, если оно связано с появлением невалидного объекта (рис.3.-39).

```
public class Null<T> {                    // Контейнер для ошибок
    private boolean valid=true;           // Признак валидности
    private UNIException exept=null;      // Сохраненное исключение
    private T ref=null;                   // Ссылка на валидный объект
    public Null(T ref){                 // Конструктор для валидного значения
        valid = true; this.ref = ref;
        }
    public Null(){                        // Невалидное значение по умолчанию
        valid = false;exept = null;
        }
    public Null(UNIException ex){     // Невалидное значение - исключение
        valid = false; exept = ex;
    }
    public T refOrNull(){               // Получение ссылки на объект или null
        return valid ? ref : null;
        }
    public T refOrExept() throws UNIException{
        if (!valid)                       // Получение ссылки или генерация исключения
            throw exception();
        return ref;
        }
    }
```

Рис.3-39. Контейнер-шаблон валидности объекта

Программные компоненты вместо того, чтобы возвращать ссылку, *null* или генерировать исключение, возвращают соответствующий вариант контейнера. Вызывающий код использует контейнер по своему усмотрению – передает дальше в существующем виде, либо извлекает ссылку и проверяет на исключение.

Шаблоны параллелизма

Шаблоны параллелизма являются наиболее сложными с точки зрения логики работы и наличия возможных ошибок синхронизации.

Однопотоковое исполнение (Single Threaded Execution)

Синхронизированное исполнение группы действий друг за другом, как правило, при разделении ими общего ресурса. Возможны различные реализации в зависимости от требования – исполнять планируемые действия в одном потоке, или нет. Возможные варианты реализации:

- блоки *synchronized* в различных классах над общим объектом синхронизации, либо метод *synchronized*, вызываемый в одном объекте несколькими потоками. В этом случае не создается общий поток, поэтому каждое действие выполняется в текущем потоке. Имеющиеся недостатки: если синхронизируемое действие занимает много времени, то это тормозит тот поток, в котором оно выполняется.
- Используется шаблон *планировщик,* организующий исполнение действий последовательно в одном потоке с очередью запросов.

Объект блокировки (Lock Object)

Создается один общий объект блокировки на всю структуру данных вместо множества независимых элементов синхронизации по отдельным данным и операциям над ними. Менее эффективен, но более прост и надежен.

Блокировка (Read/Write Lock)

Обращение за чтением данных к ресурсу выполняется параллельно, без взаимной блокировки, операции записи или обновления монополизируют ресурс, синхронизируя все параллельные операции, в том числе и чтения.

Двухфазное завершение (Two-phase Termination)

Основная идея шаблона: работающий поток нельзя просто уничтожить в любой момент времени. Поток должен получить уведомление о необходимости завершиться, после чего он выполняет финализирующие действия и выходит из метода *run*. Уведомление производится через логическую переменную в классе потока, эта же переменная является условием продолжения его основного цикла. Главный поток устанавливает признак завершения и уведомляет потоки о начале завершения. Потоки завершаются асинхронно, обнаруживая установленный признак. Необходимо за количеством незавершенных потоков. Когда все потоки завершатся, главный поток должен выполнить асинхронный обратный вызов к инициатору завершения.

Асинхронная обработка (Asynchronous Task)

По умолчанию все действия приложения, инициированные событиями графического интерфейса, выполняются в потоке графического интерфейса (GUI). Если выполняется другой поток, то для взаимодействия с элементами GUI, необходимо его синхронизовать с потоком GUI, т.е. передать исполняемый код методу, который запланирует его выполнение в общей последовательности обработки событий в потоке GUI.

Если необходимо выполнить продолжительное действие и при этом не загружать поток GUI, то необходим шаблон *асинхронное задание*.

Запрос на асинхронное исполнение представляет собой расширение абстрактного класса *Request*, в котором должен быть переопределен код, исполняемый в потоке, код завершения, обработки исключения и получения асинхронного сообщения (рис.3-40);

```
public abstract class Request {
    private boolean busy=false;
    synchronized boolean isBusy(){ return busy; }
    synchronized void  setBusy(boolean ff){  busy=ff; }
    public abstract void onExecute() throws Throwable; // Исполняемый в потоке код
    public abstract void onFinish() throws Throwable;    // Завершение, синхронизированное к GUI
    public abstract void onException(Throwable ee);      // Исключение исполняемого кода
    public abstract void onMessage(String ss);           // Сообщение, синхронизированное к GUI
    public void asyncMessage(final String ss){           // Вывод сообщения с синхронизацией
        java.awt.EventQueue.invokeLater(()-> onMessage(ss);
            });
    }}
```

```
Request oneStepCall=new Request(){          // Запрос асинхронного задания
        @Override
        public void onExecute() throws Throwable {
            for(int i=0;i<10;i++){          // Код, исполняемый в потоке
                Thread.sleep(2000);         // Вывод сообщения, синхронизированного к GUI
                asyncMessage("Прошло "+((i+1)*2)+" сек.");
                }}
        @Override                           // Код завершения
        public void onFinish() throws Throwable {
            mes.setText("Финальная часть задания в потоке GUI");
            }
        @Override                           // Код обработки исключения
        public void onException(Throwable ss) {
            mes.setText(ss.getMessage());
            }
        @Override                           // Вывод сообщения, синхронизированного к GUI
        public void onMessage(String ss) {
            mes.setText(ss);
            }
        };
```

Рис.3-40. Интерфейс и пример запроса и пример

Класс асинхронного задания *AsyncTask* получает при конструировании запрос, запускает поток и вызывает к метод *onExecute* в запросе. По окончании он синхронизируется к потоку GUI и выполняет в нем код завершения. Если при исполнении кода в потоке происходят исключения, то они перехватываются, и в потоке GUI выполняется метод *onException* из запроса.

```
public class AsyncTask extends Thread{
        Request code;
        public AsyncTask(Request codex){
            code=codex;              // Запомнить запрос
            code.setBusy(true);
            setPriority(NORM_PRIORITY-2);
            start();                 // Запустить поток
            }
        public void run(){
            try {
                code.onExecute();            // Выполнить код
                java.awt.EventQueue.invokeLater(()->{
                    try {                    // Синхронизация к GUI
                        code.onFinish();     // Код завершения
                        code.setBusy(false);
                        }
                    catch (Throwable ee) {  // Исключение при завершении
                        code.onException(ee);
                        code.setBusy(false);
                        }
                    });
                    }                        // Исключение при исполнении кода
            catch (final Throwable ee2) {
                java.awt.EventQueue.invokeLater(()->{
                    code.onException(ee2);
                    code.setBusy(false);
                    });
                }
        }}
```

Рис.3-41. Класс асинхронного задания

Планировщик (Scheduler)

В шаблоне *планировщик* действия, инициируемые независимыми запросами, выполняются последовательно в отдельном потоке. Рассмотрим подробности реализации:

- класс планировщика имеет вектор или список объектов-запросов, в котором имитируется очередь;
- запрос представляет собой абстрактный класс, аналогичный описанному в шаблоне *асинхронного задания.* В него добавлен метод *onCancel*, вызываемый при отмене запросов по завершению работы планировщика, которые находятся в очереди на обслуживание;
- может быть реализована стратегия планирования – извлечение из очереди очередного запроса на исполнения, в простейшем случае – это FIFO;
- сам планировщик имеет поток, который запускается при создании объекта и выполняет цикл. В начале цикла поток *засыпает* в ожидании события с помощью метода *wait* . При помещении запроса в очередь инициируется его *пробуждение* методом *notify* над объектом синхронизации. Поток извлекает из очереди запрос, исполняет запланированный код и планирует код завершения в потоке GUI;

- поток продолжает просматривать и выполнять запросы из очереди, пока она не опустеет, после чего *засыпает;*
- операции работы с очередью синхронизируются между собой.

Двойная буферизация (Double Buffering)

Шаблон воспроизводит стандартную схему взаимоотношений *поставщик-потребитель* при отсутствии колебаний производительности. Пока потребитель обрабатывает содержимое одного буфера, поставщик готовит другой. По окончании обработки потребитель меняет буферы и пробуждает поставщика.

В приведенной логике имеется небольшой дефект. Поставщик должен работать быстрее потребителя. В противном случае потребитель может поменять буферы в момент их заполнения поставщиком. Чтобы этого не произошло, можно ввести систему исключения двух спящих потоков: если один поток уже спит, то он пробуждается и производится обмен, иначе проверяющий условие поток засыпает сам (рис.3-42).

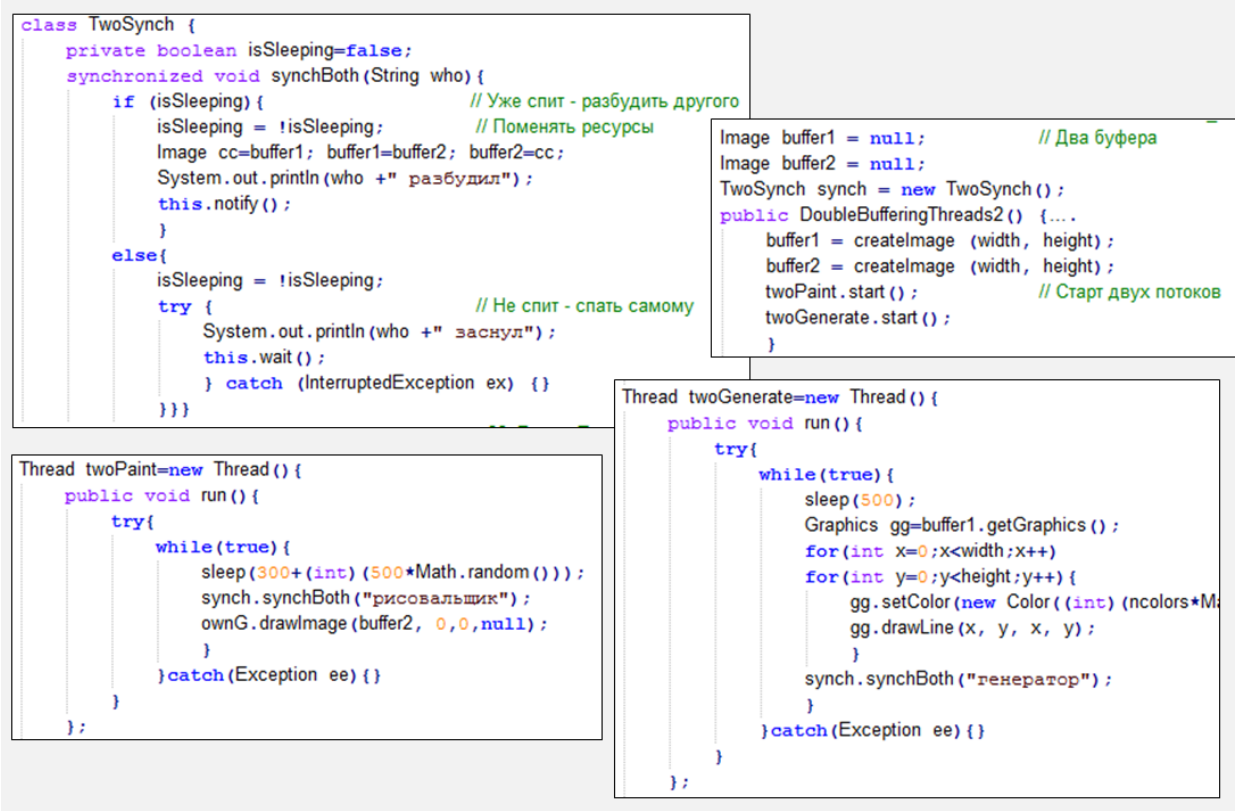


Рис.3-42. Двойная буферизация

Поставщик-потребитель (Producer-Consumer)

Стандартная задача синхронизации потоков генератора и обработчика данных при вариациях производительности. Между поставщиком и потребителем имеет буферный пул блоков, в которые поставщик предварительно подгружает данные. Имеется блокировка поставщика по заполнению буферного пула и блокировка потребителя по отсутствию данных в пуле, а также синхронизация при выполнении операций с данными в буферном пуле. В простейшей реализации достаточно блокировать потоки на одном объекте, т.к. они одновременно не могут быть блокированы. Код методов буфера используется одновременно потоками поставщика и потребителя.

Приведенный выше код моделирует шаблон *поставщик-потребитель,* работающих с циклическим буфером, при случайном интервале времени обращения

(рис.3-43). Все необходимые операции чтения/записи с ожиданием и синхронизацией выполняются в классе буфера.

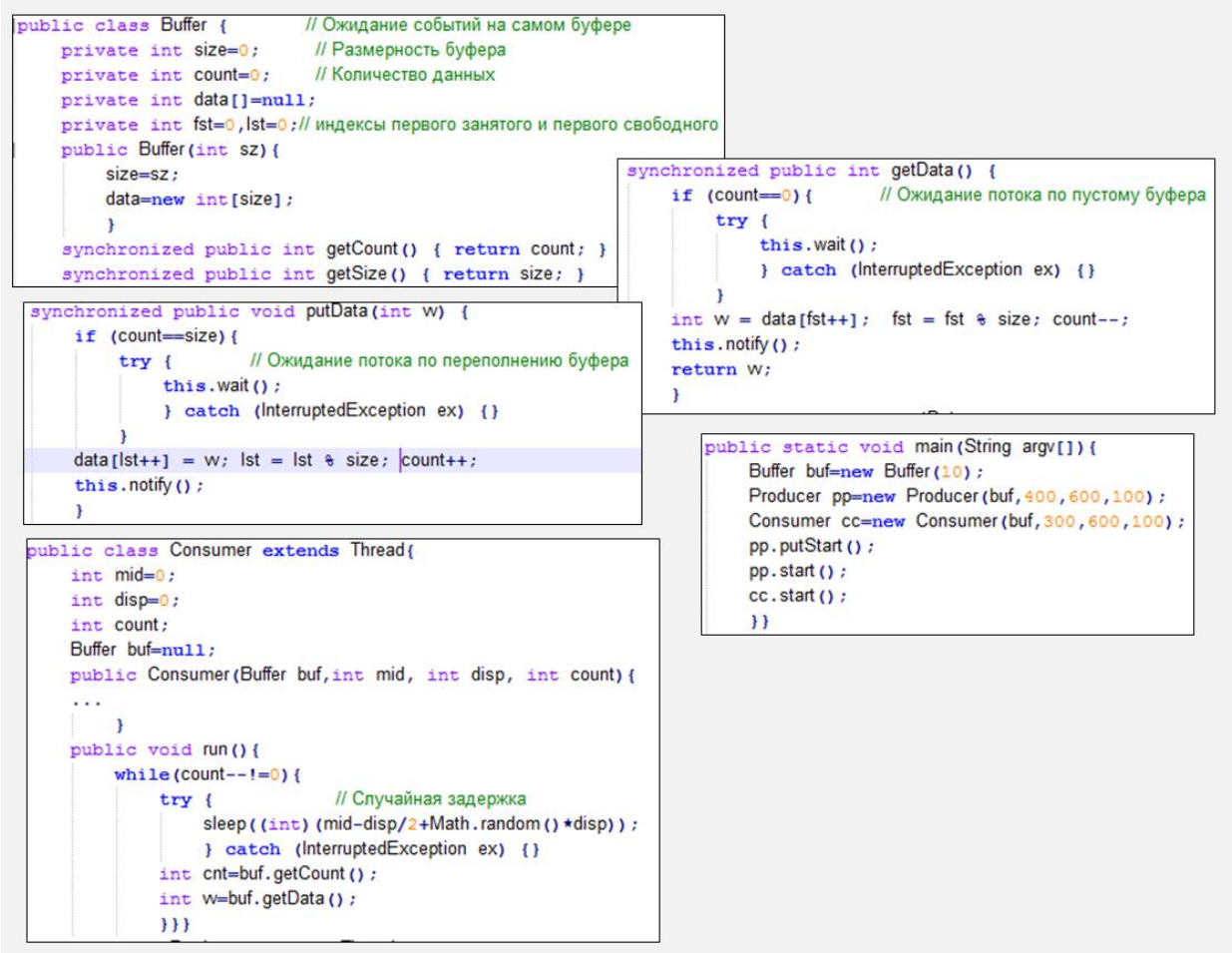


Рис.3-43. Шаблон *поставщик-потребитель*

Пул потоков

В программировании потоков используется шаблон *пул потоков*, аналогичный структурному шаблону *пул объектов* с теми же задачами - исключить затраты на создание и утилизацию объектов (в данном случае, потоков) и повысить реактивность доступа к ним. Однако простая аналогия с пулом объектов здесь неуместна, поскольку в потокам из пула требуется передавать сторонний код, что требует включения в состав шаблона компонент, аналогичных планировщику.

Ограничимся подробной словесной спецификацией шаблона. Диаграмма его классов изображена на рис.3-44:

- код, исполняемый в потоке из пула, должен быть записан в расширении абстрактного класса *Request* с методами *исполнения в потоке, завершения, завершения по исключению, отмены;*
- пулом потоков управляет класс – менеджер *PoolManager*, который включает в себя все структуры данных, связанные с пулом. К нему синхронизированы вызовы от всех внешних компонент;
- сам менеджер также является потоком, в котором реализован алгоритм работы пула. Поток менеджера включает в себя цикл, ограниченный логической переменной *shutdown*, и *засыпает* в начале каждого шага цикла;

- потоки пула являются объектами класса *PoolThread*. Поток пула также включает в себя цикл, ограниченный переменной *shutdown*, и аналогично *засыпает* в начале каждого шага цикла. Поток пула имеет ссылку на исполняемый запрос *Request* и ссылку на менеджер, для чего *PoolThread* может быть сделан внутренним классом *PoolManager* ;

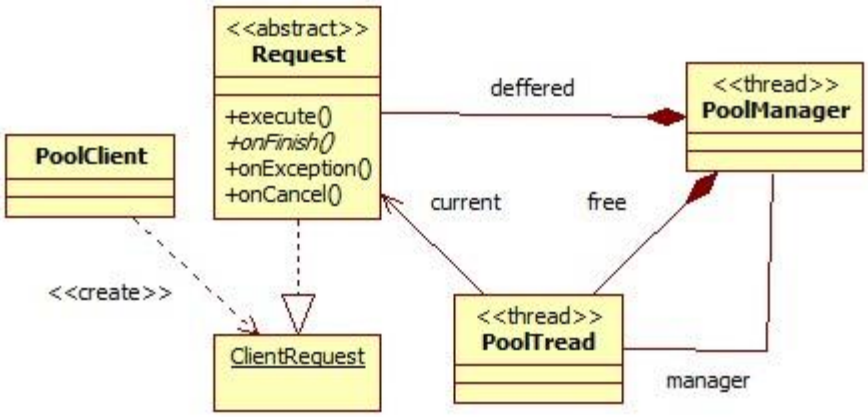


Рис.3-44. Шаблон пул потоков

- менеджер содержит очереди *свободных потоков* и *отложенных запросов;*
- при инициализации класса менеджера создаются объекты класса *PoolThread* и ссылки на них помещаются в очередь свободных, потоки стартуют и тут же засыпают. Поток менеджера также стартует и засыпает;
- класс-клиент создает конкретный запрос - адаптер обратного вызова на основе анонимного класса с переопределением всех методов в контексте вызова, после чего передает его менеджеру. Метод передачи менеджеру просто включает этот запрос в вектор отложенных и *пробуждает* поток менеджера;
- потока менеджера содержит цикл со следующей логикой шага: если векторы свободных потоков и отложенных запросов не пусты, то выбирается пара *свободный поток* - *запрос*, в объект – поток помещается ссылка на запрос и поток *пробуждается*;
- пробудившийся поток выбирает ссылку на запрос и вызывает в собственном потоке управления метод *исполнения в потоке***.** По завершении выполняет *метод завершения*, после чего добавляет себя в вектор свободных потоков и *пробуждает* менеджер. Если при выполнении кода запроса произошло исключение, то оно перехватывается, поток вызывает в запросе метод *завершения по исключению,* после чего добавляет себя в вектор свободных потоков и будит менеджер;
- при выполнении процедуры завершения работы используется паттерн двухфазного завершения. Устанавливается переменная *shutdown* в менеджере и классах потоков. В этом режиме менеджер не принимает новые запросы, для запросов из очереди отложенных вызывается метод *отмены***,** менеджер дожидается завершения всех потоков пула, после чего выполняет асинхронный обратный вызов завершения shutdown. ***Замечание по теме:*** текущие исполняемые запросы в потоках пула не прерываются, а завершаются.

Модель-представление-контроллер (MVC-Model-View-Controller)

Шаблон подробно обсуждался в 4.2.

Сессия (Session)

Позволяет обслуживающему классу-серверу выполнять логически связанную последовательность команд с сохранением в классе промежуточных данных, например, прав, полученных при авторизации, или контекста пользователя. При выполнении команды авторизации в классе-сервере создается структура данных - внутренний объект сессии, с которым ассоциируется некоторый уникальный идентификатор сессии – *handle*, возвращаемый классу-пользователю. С помощью *handle* клиент и сервер идентифицируют сессию и связанные с ней данные. Клиент может использовать класс-представитель сессии, хранящий этот *handle*. Диаграмма классов шаблона изображена на рис.3-45.

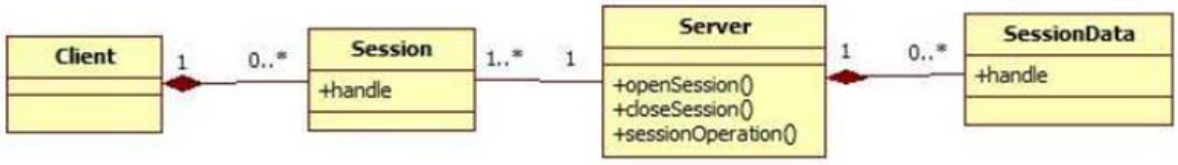


Рис.3-45. Шаблон *сессия*

При реализации шаблона с взаимодействием через сетевое соединение возможно восстановление соединения после аварийного разрыва без авторизации, путем введения специальной команды, передающей серверу выданный *handle* для разорванного соединения. Сервер обязан в таком случае хранить в течение некоторого времени данные сессии с хранящимся в них *handle*.

Транзакция (Transaction)

Выполнение последовательности методов в серверном классе единим блоком. При возникновении ошибки во время выполнения одного из методов производится откат к начальному состоянию транзакции. Кроме того, транзакция реализуется как неделимая операция - критическая секция, синхронизируясь с другими транзакциями. Шаблон транзакции включает в себя:

- примитивы: начать транзакцию, закончить транзакцию, откатить транзакцию - *RollBack*;
- моментальный снимок изменяемых данных в начале транзакции;
- синхронизированное изменение данных - шаблон *блокировка – read/write lock*.

На рис.3-46 приведен пример программной реализации транзакции при работе с БД через интерфейс JDBC.

```
try {    // ТРАНЗАКЦИЯ "ОТ ОБРАТНОГО" - отмена режима авто-изменений в БД
    dbConn.setAutoCommit (false); // Начать транзацкию - отменить авто-изменения
    ss=selectOne("SELECT MAX(id) FROM "+tname+";");
    if (ss!=null) id=Integer.parseInt(ss[0])+1;
    String sql="INSERT INTO "+tname+" (id)  VALUES ("+id+");";
    execSQL(sql);
    dbConn.commit();                // Выполнить изменения
    dbConn.setAutoCommit(true);   // Закончить транзакцию - продолжить авто-изменения
    } catch (Exception ee){
        dbConn.rollback ();            // Откатить изменения
        dbConn.setAutoCommit(true); // Продолжить авто-изменения
        throw new SQLException(ee.getMessage());
        }
    return id;
```

Рис.3-46. Пример кода использования транзакции при работе с БД через JDBC