Лекция 5. Основы объектно-ориентированного моделирования программных систем

Программной системой в широком смысле называется система взаимосвязанных компонентов - программ, файлов конфигурации, которые используются для настройки этих программ, системной документации, которая описывает структуру системы, и пользовательской документации, которая объясняет, как использовать систему.

В более узком смысле программная система часто отождествляется с комплексом взаимосвязанных программ. «Взаимосвязанных» означает наличие обмена данными между программами (связь по информации) и обращения к функциям внешней программы (связь по управлению).

Программная система обладает свойством целостности и в то же время всегда может быть разделена на подсистемы, а в конечном итоге на элементарные компоненты (объекты, классы). Внутренние связи отдельной подсистемы заведомо сильнее внешних.

В контексте дисциплины можно понимать программную систему как совокупность объектов, где каждый объект описывается с помощью класса. Классы группируются в пакеты. Взаимосвязь между классами/объектами осуществляется через вызов методов с передачей параметров и получением возвращаемых значений, импортирования классов и пакетов и другими способами, которые рассматриваются в текущем разделе курса.

Чтобы создать качественную программную систему, недостаточно просто знать язык программирования, уметь писать синтаксически правильные и работоспособные программы. Необходимо до начала написания программного кода тщательно продумать архитектуру будущей системы, чтобы минимизировать возникновение проблем в ходе написания кода, тестирования системы, ее эксплуатации и модификации.

В этой связи вы жизненном цикле программных систем выделяют стадию ***проектирования***. Цель — разработать модели программной системы, выработать технические и алгоритмические решения.

5.1. Описание класса средствами языка визуального моделирования

На стадии проектирования программной системы возникает вопрос об удобной форме описания классов предметной области. Словесное описание на естественном языке хорошо понятно человеку, но не достаточно наглядно и плохо формализовано. Использование языков программирования, в том числе псевдокода, не очень наглядно. Кроме того, именно для написания качественного кода и необходима наглядная, понятная модель программной системы.

Выход — использование графических моделей, которых сочетаются понятность и наглядность. Для этих целей применяется ***UML — унифицированный язык моделирования (Unified Modeling Language)***.

UML, с одной стороны, позволяет описать предметную область, бизнес-процессы, организационную структуру предприятия и т. п. С другой стороны,

UML используется и для описания программных систем, которые создаются для автоматизации бизнес-процессов.

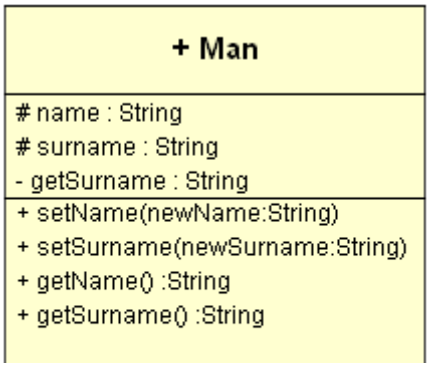
UML — графический язык. В его основе лежит богатая система графических обозначений, благодаря которой можно наглядно представить сложную систему. Графические модели, создаваемые на UML, называют UML-диаграммами. UML-диаграммы можно рисовать в любом редакторе и даже от руки, но в настоящее время существует множество инструментов, специально предназначенных именно для построения UML-диаграмм.

UML позволяет представить систему с разных точек зрения, в связи с чем выделяют различные виды UML-диаграмм. Мы ограничимся рассмотрением только одного вида диаграммы — диаграммы классов.

Класс на UML-диаграмме изображается в виде прямоугольника, разделенного по вертикали на три части. В верхней части записывается имя класса. В средней — перечень ***атрибутов***. В нижней — перечень ***операций***.

В классах конкретного языка программирования атрибутам соответствуют поля, а операциям — методы.

На рисунке ниже приводится изображение класса *Man*, который представляет данные о человеке и возможные операции по обработке этих данных[1].



5.2. Тип данных и область видимости атрибута

Атрибут, как было отмечено выше, представляет собой элемент данных. Каждый атрибут имеет:

- имя, уникальное в пределах класса (в разных классах могут быть одноименные атрибуты, в одном классе — нет);
- тип данных;
- область видимости.

На UML-диаграмме имя и тип атрибута записываются в формате «имя: тип».

Тип данных определяет множество допустимых значений атрибута, его «природу» (число, символ, символьная строка и т. д.) и во многих случаях объем памяти компьютера, требуемый для хранения максимального из возможных значений атрибута.

---

1 Пример взят отсюда: https://habr.com/ru/post/150041/

При определении типов данных на этапе анализа предметной области имеет смысл ориентироваться не на язык программирования, а на суть: что должен представлять собой данный атрибут? Поэтому в примере использованы такие наименования типов, как «строка», «целое», «логический», «дата».

Область видимости атрибута определяет его доступность в программной системе:

- ***частный атрибут (private)*** — атрибут непосредственно доступен для чтения и изменения только внутри данного класса; операции данного класса могут обращаться к атрибуту непосредственно, операции других классов — нет; на UML-диаграмме обозначается знаком «-» перед именем атрибута;
- ***защищенный атрибут (protected)*** — атрибут непосредственно доступен для чтения и изменения внутри класса, а также из подклассов данного класса; наличие подклассов связано с принципом наследования — см. далее в этой лекции; на UML-диаграмме обозначается знаком «#» перед именем атрибута;
- ***общедоступный атрибут (public)*** - непосредственно доступен для чтения и изменения из любого класса; на UML-диаграмме обозначается знаком «+» перед именем атрибута;

В соответствии с принципом инкапсуляции, чаще всего атрибуты классов определяют либо как частные, либо как защищенные. Общедоступными их делать избегают. Причина заключается в том, что изменение атрибутов, как правило, подчиняется определенной логике, которая реализуется в операциях класса. Изменение значения атрибута имеет смысл выполнять не непосредственно, а через операции класса.

Атрибуты подразделяются на статические атрибуты и атрибуты экземпляра.

Статические атрибуты на диаграмме подчеркиваются одной сплошной линией.

5.3. Параметры, возвращаемые значения и область видимости операций

Операция имеет:

- имя;
- область видимости;
- параметры;
- тип возвращаемого значения.

Область видимости для операций имеет тот же смысл, что и для атрибутов. Значительная часть операций любого класса является общедоступной. Операции могут быть защищенными или частными, если они по сути своей являются вспомогательными и должны вызываться только из других операций данного класса, но не извне.

Параметры операции — дополнительные данные, необходимые для корректного выполнения операции. Операция может не иметь параметров, если в этом нет необходимости.

Описание параметров обычно включает имя и тип параметра, разделяемые двоеточием. Если параметров у операции несколько, описания параметров перечисляются через запятую. Перечень параметров заключается в круглые скобки; для операции без параметров указываются пустые круглые скобки, что позволяет визуально отличить такую операцию от атрибута.

Тип возвращаемого значения — тип выходного значения, полученного в результате выполнения данной операции.

Операция может не возвращать значения: в этом случае предполагается, что операция является процедурой обработки данных, которая выполняет некоторые полезные действия, но не позволяет получить информацию об объекте.

На UML-диаграмме тип возвращаемого значения указывается после имени операции и списка параметров через двоеточие.

5.4 Отношения между классами[2]

5.4.1. Обобщение

Обобщение — отношение «предок-потомок». Реализует один из базовых принципов объектно-ориентированного подхода — ***наследование***.

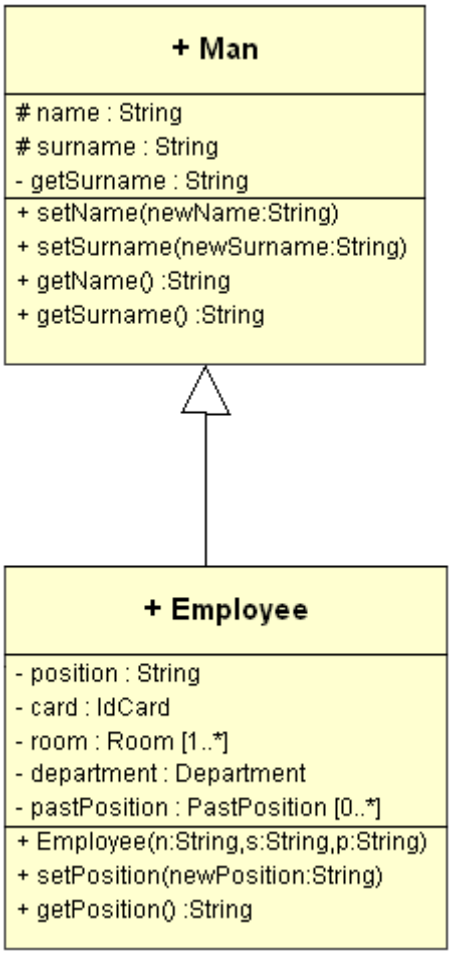
Отношение обобщения показывает, что один из классов является подклассом другого. Экземпляр подкласса является одновременно экземпляром суперкласса.

Подкласс наследует от суперкласса общедоступные и защищенные атрибуты и операции; частные — не наследует.

На UML-диаграмме обобщение показывается с помощью линии с пустым треугольником у суперкласса. Наследуемые атрибуты и операции в подклассах не показываются.

Пример: класс *Man* («Человек») является обобщением класса *Employee* («Сотрудник»). Каждый сотрудник организации является человеком, но не всякий человек сотрудник организации. Класс *Employee* наследует общедоступные и защищенные элементы суперкласса, а также содержит собственные атрибуты и операции.

2 Использованы методические материалы курса «проектирование информационных систем», разработанные доцентом кафедры ИВС Довбуш Г.Ф.



Обобщение в Java реализуется при помощи механизма наследования, или расширения (*extends*) одного класса другим.

```
public class Man{
protected String name;
   protected String surname;
   public void setName(String newName){
      name = newName;
   }
   public String getName(){
      return name;
        }
   public void setSurname(String newSurname){
      name = newSurname;
   }
   public String getSurname(){
      return surname;
   }
}
// наследуем класс Man
public class Employee extends Man{
```

```
    private String position;
    // создаем и конструктор
    public Employee(String n, String s, String p){
        name = n;
        surname = s;
        position = p;
    }
    public void setPosition(String newProfession){
        position = newProfession;
    }
    public String getPosition(){
        return position;
    }
}
```

5.4.2. Ассоциация

Ассоциация – именованное структурное отношение, описывающее набор связей, в котором каждая из них представляет собой соединение между объектами.

Например, сотрудник должен иметь идентификационную карточку в организации. Но предположим, идентификационная карточка — сложный объект, который должен описываться с помощью отдельного класса *IdCard*. Как показать, что сотрудник имеет идентификационную карточку? Именно в этом и заключается смысл ассоциации: показать, что объекты неких разных классов имеют между собой относительно устойчивые связи.

Ассоциация изображается в виде направленной стрелки (дуги) или ненаправленной линии (ребра).

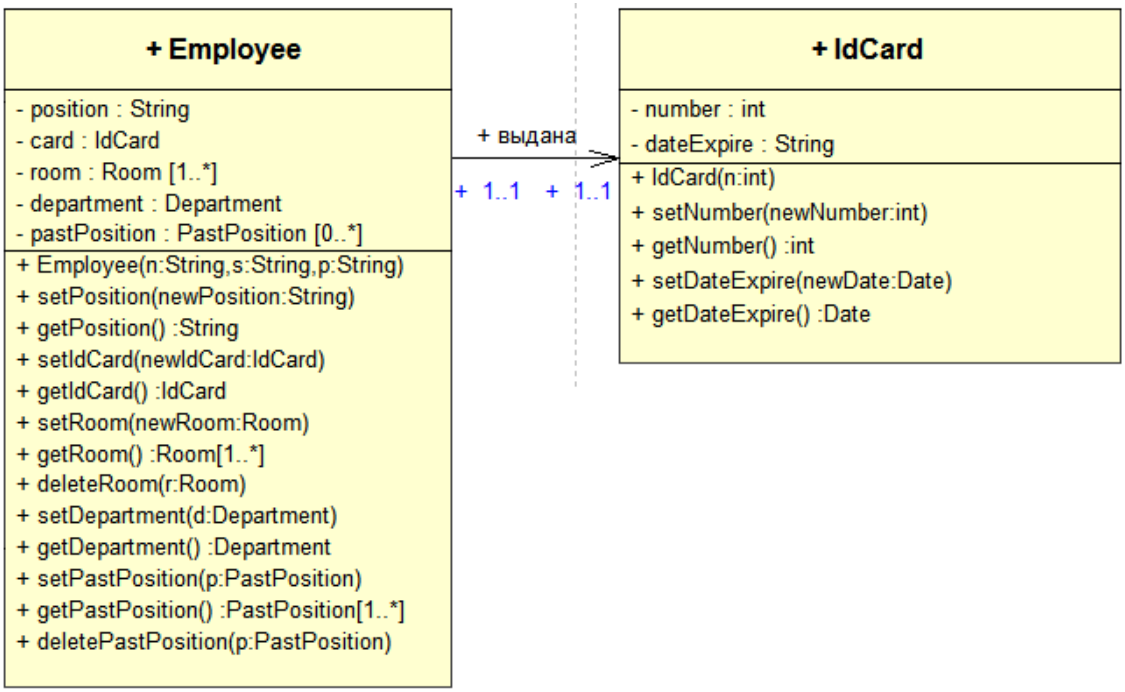
На концах ассоциации указывается ***множественность*** (***мощность***) — свойство ассоциации, которое показывает, какое число экземпляров класса может участвовать в связи с экземпляром противоположного класса.

Основные типы множественности в UML следующие:

- 0..1 — в связи участвует от нуля до одного экземпляров данного класса;
- 1 (или 1..1) — в связи участвует один и только один экземпляр данного класса;
- 1..* - в связи участвует 1 ил более экземпляров данного класса;
- * - в связи участвует 0 и более экземпляров данного класса;

В некоторых программных продуктах, используемых для создания UML-моделей, множественность «ноль и более» обозначается «0..*». В этом случае множественность «*» означает «более одного».

Пример: каждый сотрудник должен иметь идентификационную карточку, и притом только одну. Карточка также должна принадлежать сотруднику, и притом только одному. Тип множественности - «один к одному».



Обратим внимание, что ассоциация на рисунке имеет направление: подчёркивается, что именно сотруднику выдана карта, то есть каждый экземпляр класса *Employee* должен «знать» свой экземпляр класса *IdCard*. Реализовать это в программе очень просто: в классе Employee должно быть дополнительное поле — ссылка на объект типа *IdCard*. А для установки/разрыва связи сотрудника с картой необходимо предусмотреть «геттер» *getIdCard()* и «сеттер» *setIdCard()*.

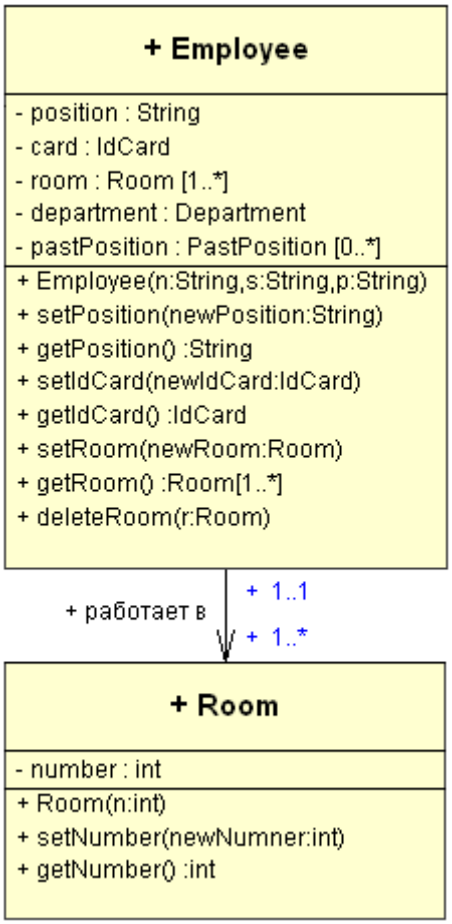
```
public class Employee extends Man{
    private String position;
    private IdCard iCard;
    public Employee(String n, String s, String p){
        name = n;
        surname = s;
        position = p;
    }
    public void setPosition(String newPosition){
        position = newPosition;
    }
    public String getPosition(){
        return position;
    }
    public void setIdCard(IdCard c){
        iCard = c;
    }
    public IdCard getIdCard(){
        return iCard;
    }
```

```
}
public class IdCard{
	private Date dateExpire;
	private int number;
	public IdCard(int n){
		number = n;
	}
	public void setNumber(int newNumber){
		number = newNumber;
	}
	public int getNumber(){
		return number;
	}
	public void setDateExpire(Date newDateExpire){
		dateExpire = newDateExpire;
	}
	public Date getDateExpire(){
		return dateExpire;
	}
}
```

Пример создания и связывания объектов в теле программы:

```
IdCard card = new IdCard(123);
card.setDateExpire(new SimpleDateFormat("yyyy-MM-dd").parse("2015-12-31"));
sysEngineer.setIdCard(card);
System.out.println(sysEngineer.getName() +" работает в должности "+ sysEngineer.getPosition());
System.out.println("Удостовирение действует до " + new SimpleDateFormat("yyyy-MM-dd").format(sysEngineer.getIdCard().getDateExpire()) );
```

Теперь рассмотрим пример ассоциации «один-ко-многим». Представим, что в организации положено закреплять за работниками помещения. Помещение представлено классом *Room*. Каждому сотруднику может соответствовать одно и более рабочих помещений.



Реализация класса *Room* примитивно проста:

```
public  class Room{
    private int number;
    public Room(int n){
       number = n;
    }
    public void setNumber(int newNumber){
       number = newNumber;
    }
    public int getNumber(){
       return number;
    }
}
```

В класс Employee добавляются следующие элементы:

```
private HashSet<Room> rooms = new HashSet(); //  Коллекция  ссылок  на помещения
…

// Установка связи сотруднка с одним (!) помещением
```

```
public void setRoom(Room newRoom){
   room.add(newRoom);
}

// Получение коллкции ссылок на все помещения сотрудника
public HashSet<Room> getRooms(){
   return room;
}

// Разрыв связи сотрудника с указанным помещением
public void deleteRoom(Room r){
   room.remove(r);
}
```

Пример использования:

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

   Employee sysEngineer = new Employee("John", "Connor", "Manager");
   IdCard card = new IdCard(123);
     card.setDateExpire(new  SimpleDateFormat("yyyy-MM-dd").parse("2015-
12-31"));
       sysEngineer.setIdCard(card);
       Room room101 = new Room(101);
   Room room321 = new Room(321);
   sysEngineer.setRoom(room101);
   sysEngineer.setRoom(room321);
     System.out.println(sysEngineer.getName() +" работает в должности "+
sysEngineer.getPosition());
                System.out.println("Удостовирение действует до " +
sysEngineer.getIdCard().getDateExpire());
   System.out.println("Может находиться в помещеньях:");
   Iterator iter = sysEngineer.getRoom().iterator();
   while(iter.hasNext()){
      System.out.println( ((Room) iter.next()).getNumber());
       }
}
```

5.4.3. Агрегирование

Агрегирование — структурное отношение «целое — часть».

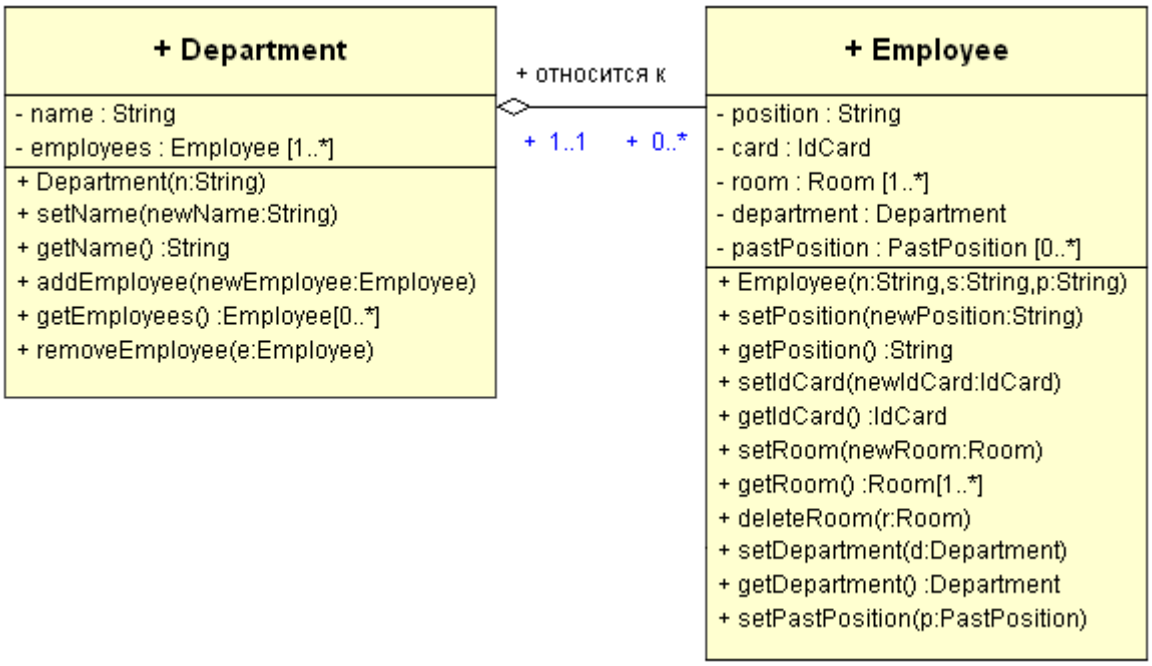
Используется, когда класс *A* является целым по отношению к классу *B* или, проще говоря, когда объект класса *A* включает объект класса *B* как одну из составных частей. При этом говорят, что *A* является агрегатом *B*.

Агрегирование обозначается линией с ромбом на конце, ромб направлен в сторону агрегата.

Различают слабое и сильное агрегирование.

Слабое агрегирование — при котором часть может существовать сама по себе, отдельно от целого. Например, двигатель может существовать отдельно от автомобиля. Уничтожение автомобиля не обязательно приводит к уничтожению двигателя. На UML-диаграмме слабое агрегирование обозначается в виде линии с незакрашенным ромбом на конце.

Пример. В каждом отделе может работать один или более человек. Можно сказать, что отдел включает в себя одного или более сотрудников и таким образом их агрегирует. При этом на предприятии могут быть сотрудники, которые не принадлежат ни одному отделу, такие как директор предприятия.



Особенности программной реализации следующие.

1. Класс *Department*, помимо конструктора и методов доступа к имени отдела, методы занесения в отдел нового сотрудника, для удаления сотрудника из отдела и для получения всех сотрудников данного отдела. Навигация здесь является двунаправленной: от объекта типа *Department* можно узнать о сотруднике и от объекта типа *Employee* можно узнать, к какому отделу он относится.

```
public class Department{
   private String name;
   private HashSet<Employee> employees = new HashSet();

   public Department(String n){
      name = n;
   }

   public void setName(String newName){
      name = newName;
```

```
    }

    public String getName(){
       return name;
    }

    public void addEmployee(Employee newEmployee){
       employees.add(newEmployee);
       // связываем сотрудника с этим отделом
       newEmployee.setDepartment(this);
    }

    public HashSet<Employee> getEmployees(){
       return employees;
    }

    public void removeEmployee(Employee e){
       employees.remove(e);
    }
}
```

2. В классе *Employee* находится поле типа *Department* и методы доступа к нему:

```
...
private Department department;
...
public void setDepartment(Department d){
   department = d;
}
public Department getDepartment(){
   return department;
}
```

Пример использования:

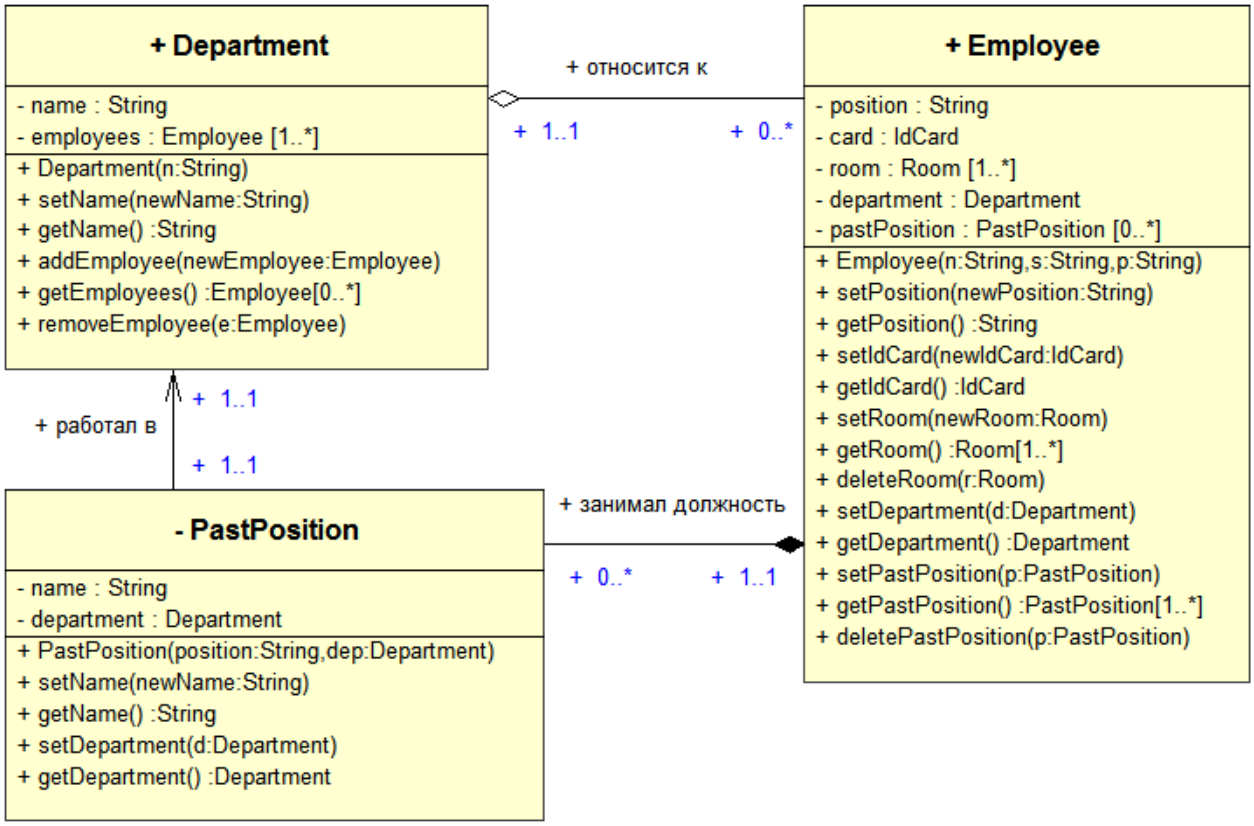
```
Department programmersDepartment = new Department("Программисты");
programmersDepartment.addEmployee(sysEngineer);
System.out.println("Относится к отделу "+sysEngineer.getDepartment().getName());
```

Сильное агрегирование (композиция) — при котором часть уничтожается вместе с целым. Например, дом (целое) и квартира (часть): квартира не существует без дома.

Что касается рассматриваемой системы учета сотрудников организации, можно привести следующий пример композиции. Предположим, что одним из

требований к системе является требование о том, чтобы хранить данные о прежней занимаемой должности на предприятии.

Введем новый класс PastPosition. В него, помимо свойства «имя» (name), введем и свойство *department*, которое свяжет его с классом *Department*. Данные о прошлых занимаемых должностях являются частью данных о сотруднике, т.е. между ними связь «целое-часть». В то же время данные о прошлых должностях не могут существовать без объекта типа *Employee*. Уничтожение объекта *Employee* должно привести к уничтожению принадлежащих ему объектов класса *PastPosition*.



Особенности реализации.

1. Класс PastRoom удобно сделать вложенным в Employee:

```
public class Employee extends Man{
  private class PastPosition{
    …
  }
  …
}
```

2. Также в класс Employee следует добавить свойства и методы для работы с данными о прошлой должности:

```
...
private Set pastPosition = new HashSet();
```

```
...
public void setPastPosition(PastPosition p){
    pastPosition.add(p);
}
public Set getPastPosition(){
    return pastPosition;
}
public void deletePastPosition(PastPosition p){
    pastPosition.remove(p);
}
…
```

3. Чтобы реализовать ассоциацию между *PastPosition* и *Department*, достаточно просто добавить необходимые для этого поля и методы по аналогии с одним из рассмотренных ранее примеров:

```
private class PastPosition{
   private String name;
    private Department department;
    public PastPosition(String position, Department dep){
       name = position;
       department = dep;
    }
    public void setName(String newName){
       name = newName;
    }
    public String getName(){
       return name;
    }
    public void setDepartment(Department d){
       department = d;
    }
    public Department getDepartment(){
       return department;
    }
}
```

5.4.4. Зависимость

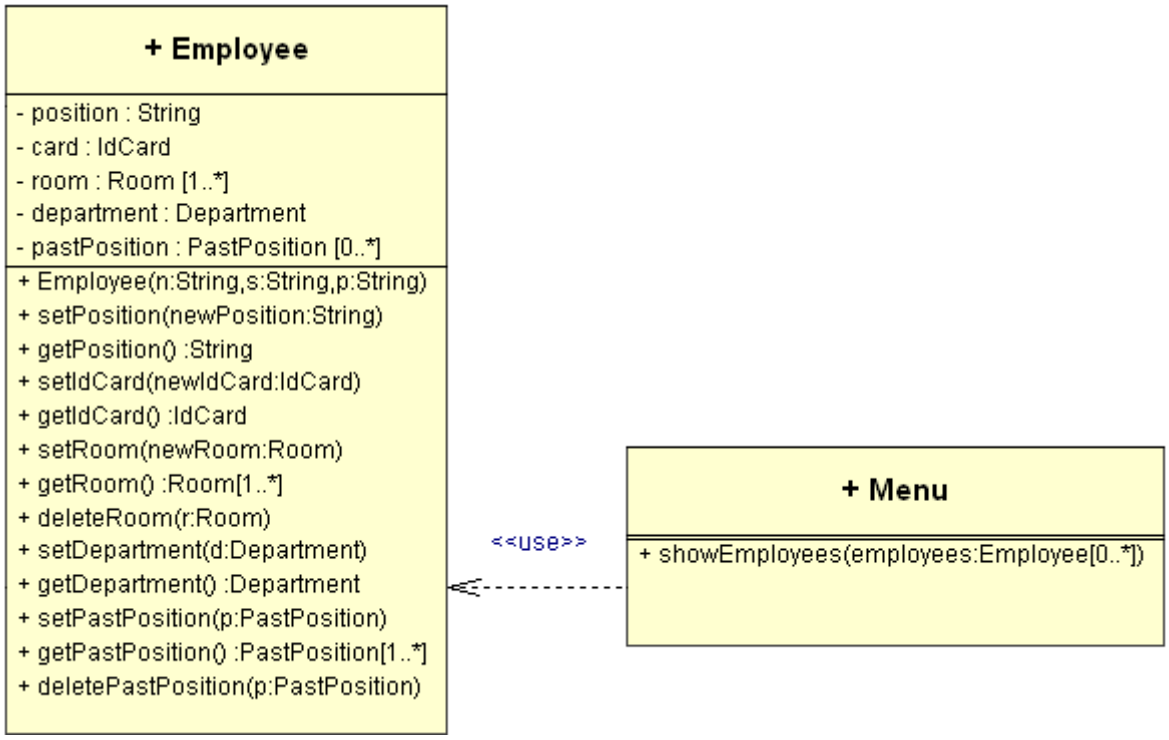
Зависимость - это отношение, которое показывает, что объект одного класса требует объекты другого класса для его спецификации или реализации. Типичный пример: ссылка на объект передается операции другого объекта в качестве параметра.

Введем в систему класс *Menu* для организации диалога с пользователем. Пусть этот класс содержит единственную операцию *showEmployees()*, которая

отображает список сотрудников и их должности. Параметром операции является массив объектов *Employee*.

Заметим, что изменения, внесенные в класс *Employee*, могут потребовать и изменения класса *Menu*. В этом и выражается зависимость класса *Menu* от *Employee*.

На UML-диаграмме отношение зависимости показывается в виде направленной пунктирной дуги. направление дуги — в сторону независимого класса.



Реализация

```
public  class Menu{
   private static int i=0;
   public static void showEmployees(Employee[] employees){
      System.out.println("Список сотрудников:");
      for (i=0; i<employees.length; i++){
         if(employees[i] instanceof Employee){
                              System.out.println(employees[i].getName()  +" - " +
employees[i].getPosition());
                           }
                  }
   }
}
```

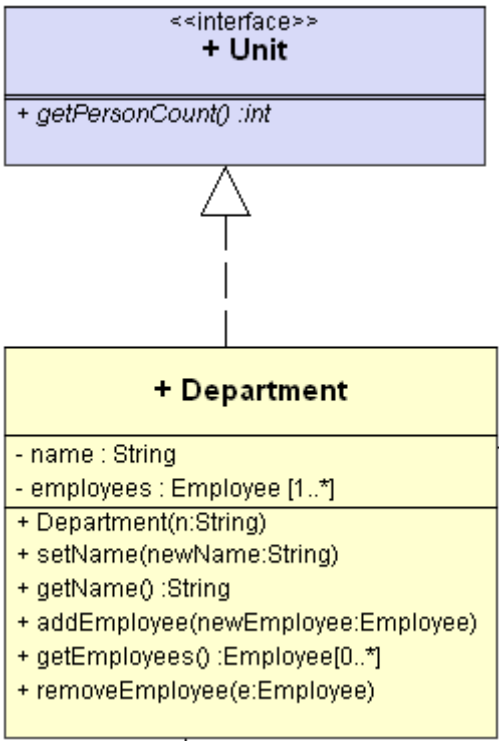
Пример использования:

```
Employee director = new Employee("Федор", "Дубов", "Директор");
Menu menu = new Menu();
Employee employees[] = new Employee[10];
employees[0]= sysEngineer;
```

```
employees[1] = director;
Menu.showEmployees(employees);
```

5.4.5. Реализация интерфейса

Реализация интерфейса, как и наследование имеет явное выражение в языке Java. Для демонстрации отношения «реализация» создадим интерфейс *Unit*. Если представить, что организация может делиться не только на отделы, а например, на цеха, филиалы и т. д., то именно интерфейс Unit и представляет собой некую абстрактную единицу деления. В каждой единице деления работает какое-то количество сотрудников, поэтому операция для получения количества работающих людей будет реализовываться для каждого класса реализующего интерфейс *Unit*.

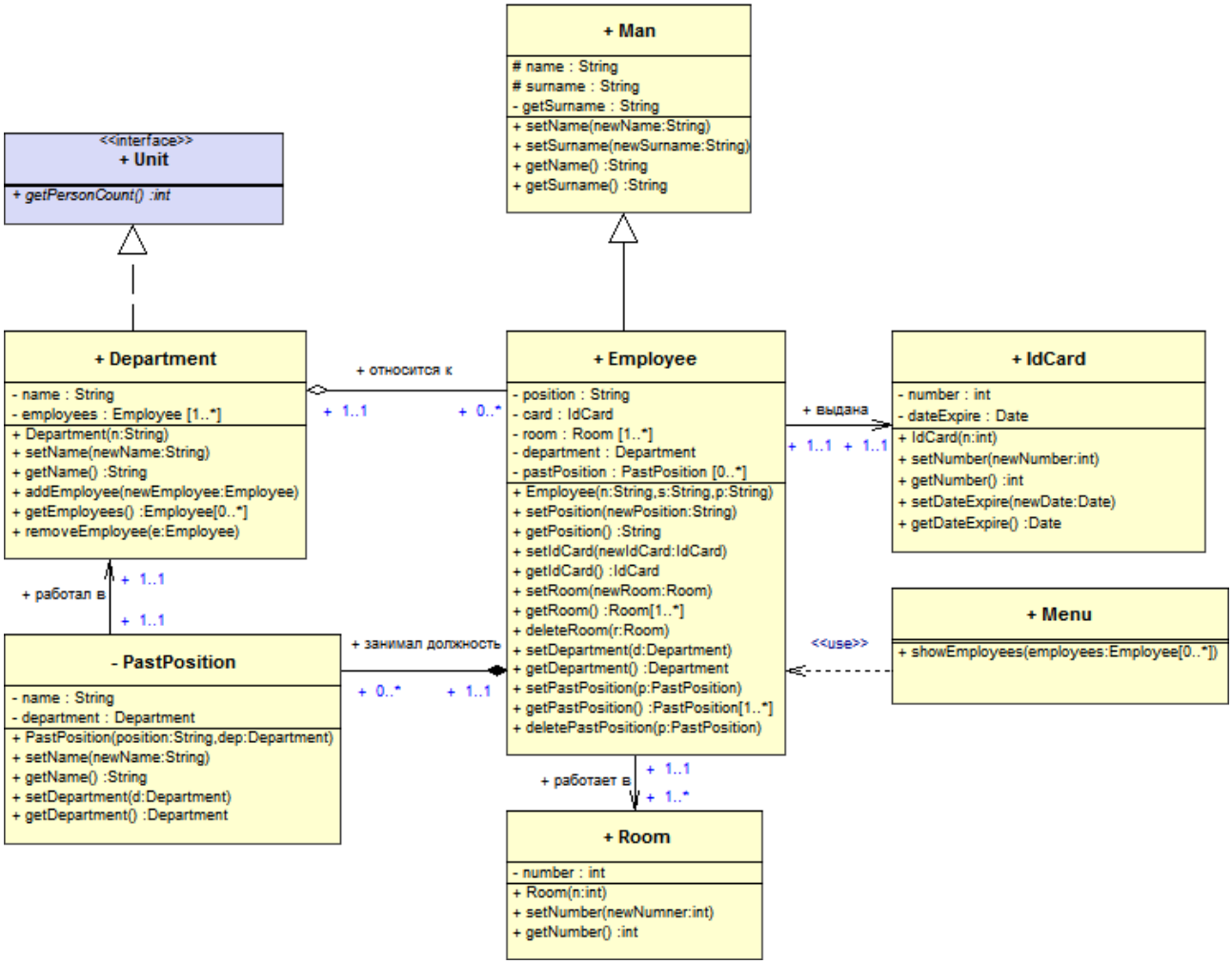


На языке Java приведенному фрагменту UML-диаграммы соответствует код:

```
public interface Unit{
    int getPersonCount();
}

public class Department implements Unit{
   ...
   public int getPersonCount(){
      return getEmployees().size();
   }
```

Резюме

Разработка сложной программной системы начинается именно с построения ее модели, с проектирования классов. Результатом визуального моделирования является UML-диаграмма классов системы. Для рассмотренного примера она выглядит следующим образом:



Лишь когда модель системы построена, имеет смысл приступать к ее реализации на выбранном языке программирования, пользуясь типовыми правилами построения кода для каждого типа отношений между классами.

5.4.6. Множественность «многие ко многим» и класс-ассоциация

В рассмотренном примере не учтен случай, когда ассоциация имеет множественность «многие ко многим», то есть экземпляр некоего класса *A* вступает во взаимодействие с одним и более экземпляров класса *B*, а экземпляр класса *B* — с одним и более экземпляров класса *A*.

Например, поезд на пути следования останавливается на различных станциях. В свою очередь на каждой из этих станций может останавливаться множество поездов.

На UML-диаграмме такой тип множественности ассоциации показывается следующим образом:

Для программной реализации ввести не два, а три Java-класса:

- Train — для представления поезда;
- Station — для представления станции;
- Stops — для представления сведений об остановке поезда на станции.

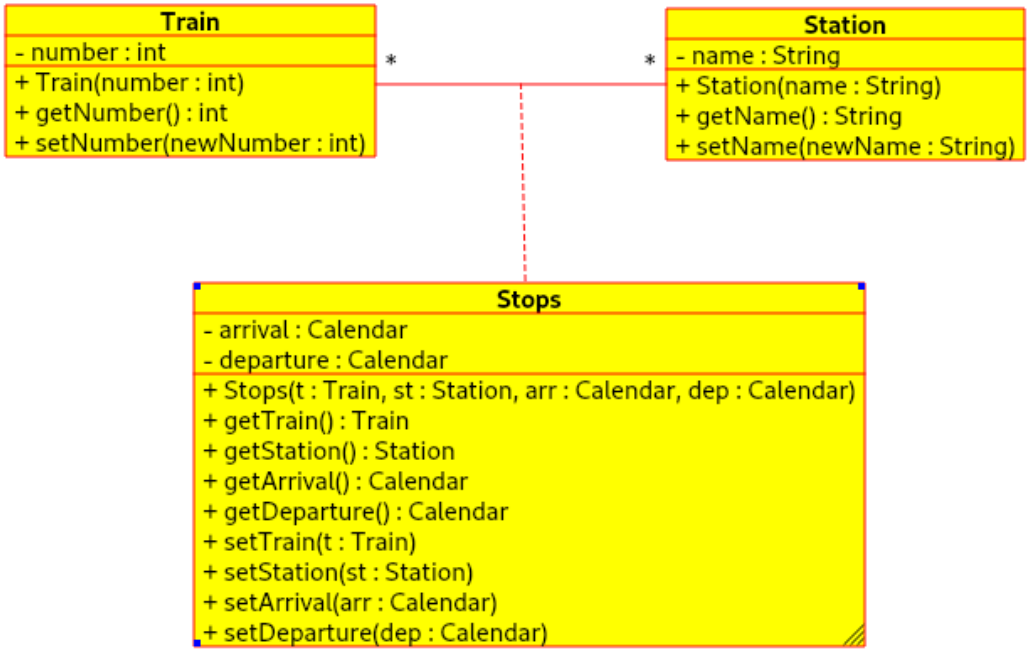
Класс Stops содержит два поля: ссылку на Train и на Station. Конструктор класса связывает конкретный поезд с конкретной станцией.

```
public class Stops{
  private Train train;
  private Station station;

  public Stops(Train t, Station st){
    train = t;
    station = st;
  }

  public Train getTrain(){
    return train;
  }

  public Statin getStation(){
    return station;
  }

  public void setTrain (Train t){
    train = t;
  }

  public void setStation(Station st){
    station = st;
  }
}
```

Заметим, что для хранения всех связей между поездами и станциями нужно иметь коллекцию объектов класса *Stops*, а ещё лучше — добавить именно в класс *Stops* необходимые статические элементы, чтобы сделать его управляющим собственными экземплярами.

На практике распространен случай, когда ассоциация (особенно «многие ко многим») между классами имеет атрибуты. В данном примере напрашивается два таких атрибута: время прибытия поезда на станцию и время убытия поезда со станции. Эти атрибуты невозможно отнести ни к поезду (поезд на разные станции прибывает в разное время; аналогично с убытием), ни к станции (разные поезда прибывают в разное время; аналогично с убытием).

Атрибуты ассоциации на UML-диаграмме показываются в виде класса-ассоциации. Например:



При наличии атрибутов ассоциации конструктор класса Stops содержит два дополнительных параметра для инициализации времени прибытия и убытия. Также присутствуют «геттеры» и «сеттеры» для атрибутов ассоциации.

5.6. Шаблоны проектирования[34]

Шаблоны проектирования (также называемые в популярных публикациях ***паттернами проектирования***) - это типовые решения часто возникающих в программировании задач.

Типы шаблонов проектирования:

- порождающие;
- структурные;
- поведенческие;

Порождающие шаблоны предоставляют механизмы инициализации, позволяя создавать объекты удобным способом.

Структурные шаблоны определяют отношения между классами и объектами, позволяя им работать совместно.

Поведенческие шаблоны используются для того чтобы упростить взаимодействие между компонентами программной системы.

---

3 https://javarush.ru/groups/posts/496-patternih-proektirovanija-v-java

4 https://javarush.ru/groups/posts/584-patternih-proektirovanija

5.6.1. Порождающие шаблоны

К порождающим шаблонам относятся:

- «Одиночка» (Singleton);
- «Фабрика» (Factory);
- «Абстрактная фабрика» (Abstract Factory);
- «Строитель» (Builder);
- «Прототип» (Prototype).

Из перечисленных шаблонов «Прототип» может быть охарактеризован предельно коротко: он предназначен для создания копии объекта. Фактически он реализуется стандартными средствами: содержит интерфейс клонируемого объекта (*Clonable* или его аналог) и классы, реализующие этот интерфейс.

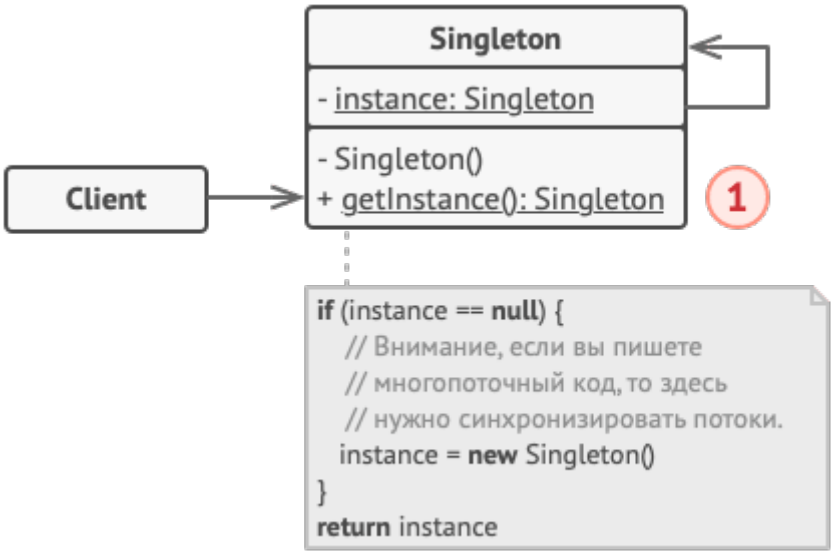
Прочие порождающие шаблоны необходимо рассмотреть более подробно.

«Одиночка»[5]

Шаблон «Одиночка» (Singleton) jграничивает создание одного экземпляра класса, обеспечивает доступ к его единственному объекту. Содержит закрытый конструктор по умолчанию и статический метод *getInstance()*, который создает и возвращает единственный объект. Не содержит других конструкторов.

Ограничение количества экземпляров класса полезно для доступа к общему ресурсу, например, базе данных.

Структура классов шаблона (*instance* – статическое поле, *getInstance()* - статический метод):



Пример:

```
// Класс одиночки определяет статический метод getInstance(),
// который позволяет клиентам повторно использовать одно и то же
// подключение к базе данных по всей программе
public class Database
{
```

5 https://radioprog.ru/post/1469

```
    // Поле для хранения объекта-одиночки
    private static Database instance;

    // Конструктор одиночки — закрытый,
    // чтобы клиенты не могли самостоятельно создавать
    // экземпляры этого класса
    private Database()
    {
       // Здесь может жить код инициализации подключения к
       // серверу баз данных.
       // ...

    // Основной статический метод одиночки служит альтернативой
    // конструктору и является точкой доступа к экземпляру этого
    // класса.
    public static Database getInstance()
    {
       if(instance == null)
         instance = new Database();

       // ...
       return instance;
    }

    // ...
}
```

«Фабрика»[6]

Пусть имеется базовый класс с подклассами. Логика выполнения программы предполагает, что в определённый момент времени должен быть создан объект одного из подклассов, но какого именно, зависит от сложившихся условий. Для решения такой задачи используют ***шаблон проектирования "Фабрика" (Factory)***. Он реализуется в виде класса, называемого классом-фабрикой (имя такого класса заканчивается словом *Factory*), который содержит статический метод *getInstance()* - как правило, с одним или более параметрами. Тип возвращаемого значения совпадает с именем суперкласса. В теле метода *getInstsnce()* выполняется проверка условия и в зависимости от результата создается экземпляр того или иного подкласса. Возвращается ссылка на созданный объект.

Имя метода может отличаться от *getInstance()*, но суть от этого не меняется.

6 https://javarush.ru/groups/posts/2370-pattern-proektirovanija-factory

Пример: класс-фабрика по созданию порции кофе. В программе имеется суперкласс *Coffee* и его подклассы: *Americano, Espresso, Cappuccino, CaffeLatte*.

Также имеется перечисление *CoffeeType* с элементами *AMERICANO, ESPRESSO, CAPPUCCINO, CAFFE_LATTE*.

Фабрика *SimpleCoffeeFactory* содержит метод *createCoffee()*, который принимает на вход тип кофе. В зависимости от того, какой тип указан, метод создает экземпляр конкретного подкласса *Coffee* и возвращает ссылку на него.

```
public class SimpleCoffeeFactory
{
   public Coffee createCoffee (CoffeeType type)
{
      Coffee coffee = null;

      switch (type)
      {
         case AMERICANO:
            coffee = new Americano();
            break;
         case ESPRESSO:
            coffee = new Espresso();
            break;
         case CAPPUCCINO:
            coffee = new Cappuccino();
            break;
         case CAFFE_LATTE:
            coffee = new CaffeLatte();
            break;
      }

return coffee;
   }
}
```

По сравнению с другими возможными решениями "Фабрика" дает следующее преимущество: в случае изменения "ассортимента" не нужно исправлять код во всех частях программы, где производится выбор нужного подкласса. Кроме того, класс-фабрику можно расширять, переопределяя метод получения экземпляра подкласса.

«Абстрактная фабрика»[7]

В более сложных случаях фабрика может создавать не один объект, а ***группу (семейство)*** взаимосвязанных объектов. Классы этих объектов не могут

7 https://radioprog.ru/post/1295

быть однозначно заданы на этапе разработки, но могут быть определены в ходе выполнения программы.

Проиллюстрируем проблему следующим примером.

Предположим, нужно создать симулятор мебельного магазина. Должны быть представлены:

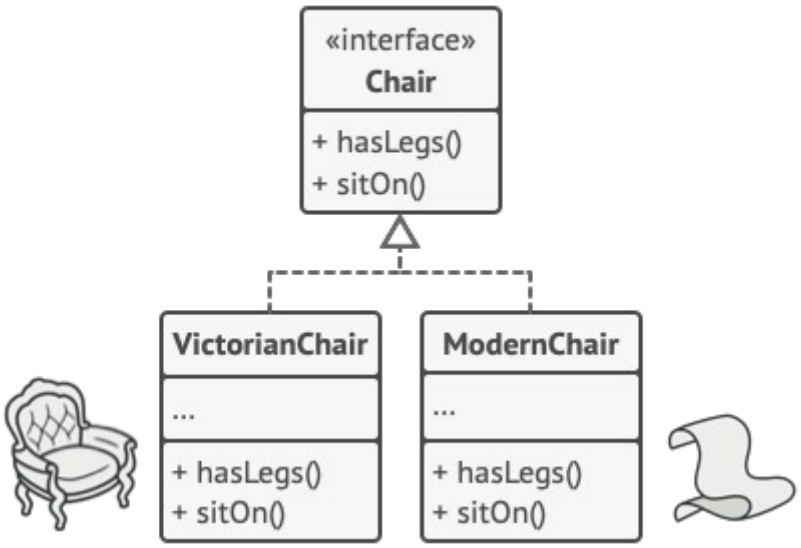
1. Семейство зависимых продуктов. Например, Кресло + Диван + Столик.

2. Несколько вариаций этого семейства. Например, стили: Ар-деко, Викторианском и Модерне.

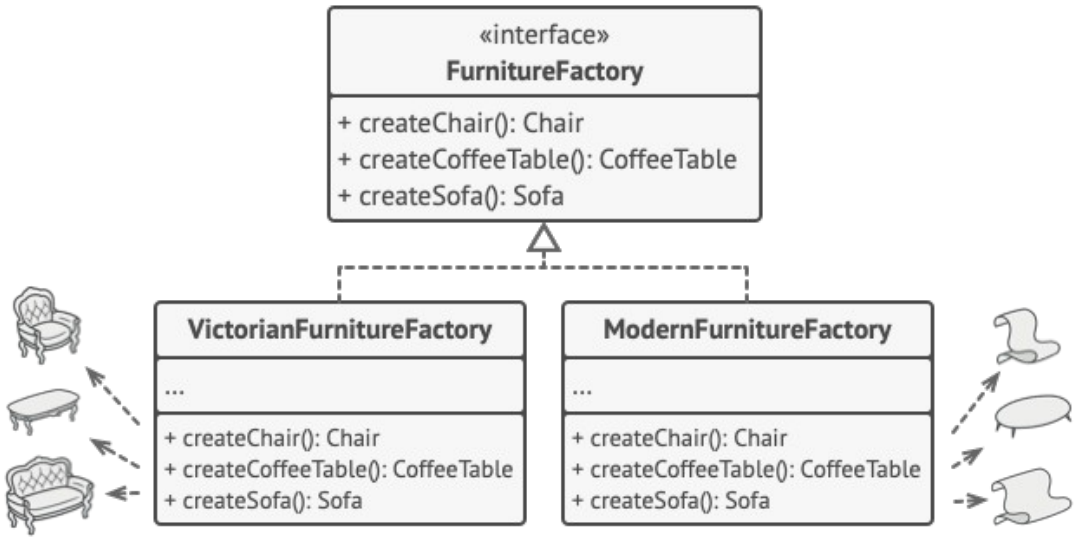
Допускается добавление новых продуктов и семейств, которое не должно приводить к изменениям в уже существующем коде.

Решение:

1. Каждый тип продукта описывается при помощи интерфейса. все вариации кресел получат общий интерфейс Кресло, все диваны реализуют интерфейс Диван и так далее.

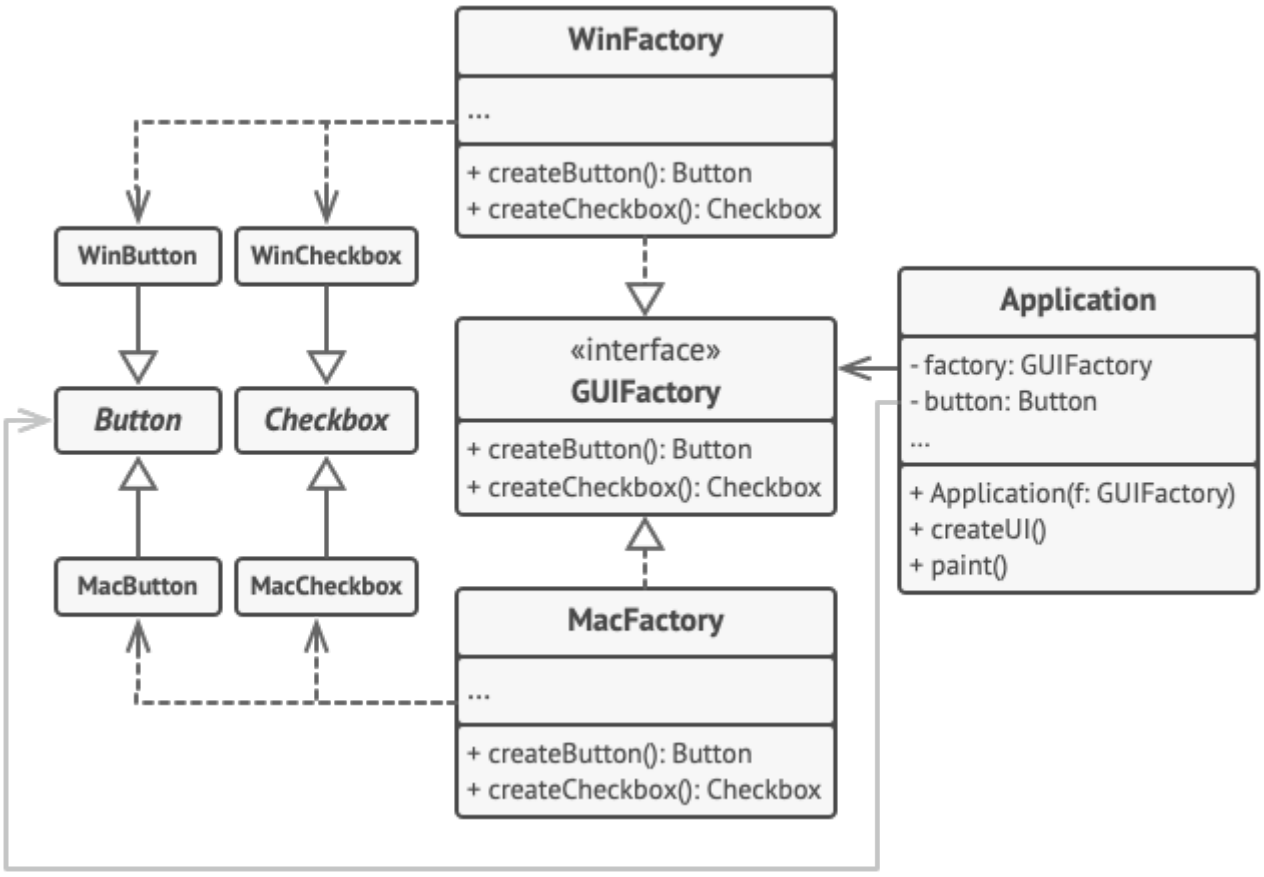


Также создается общий интерфейс *FurnitureFactory*, который содержит методы создания всех продуктов семейства: *createChair()*, *createCoffeeTable()*, *createSofa()*. Именно он и играет роль абстрактной фабрики. Каждая реализация данного интерфейса должна быть фабрикой для создания семейства продуктов конкретного стиля: *VictorianFurnitureFactory*, *ModernFurnitureFactory* и др.



Клиентский код должен работать как с фабриками, так и с продуктами только через их общие интерфейсы. Например, клиентский код просит фабрику сделать стул. Он не знает, какого типа была эта фабрика. Он не знает, получит викторианский или модерновый стул. Для него важно, чтобы на стуле можно было сидеть и чтобы этот стул отлично смотрелся с диваном той же фабрики.

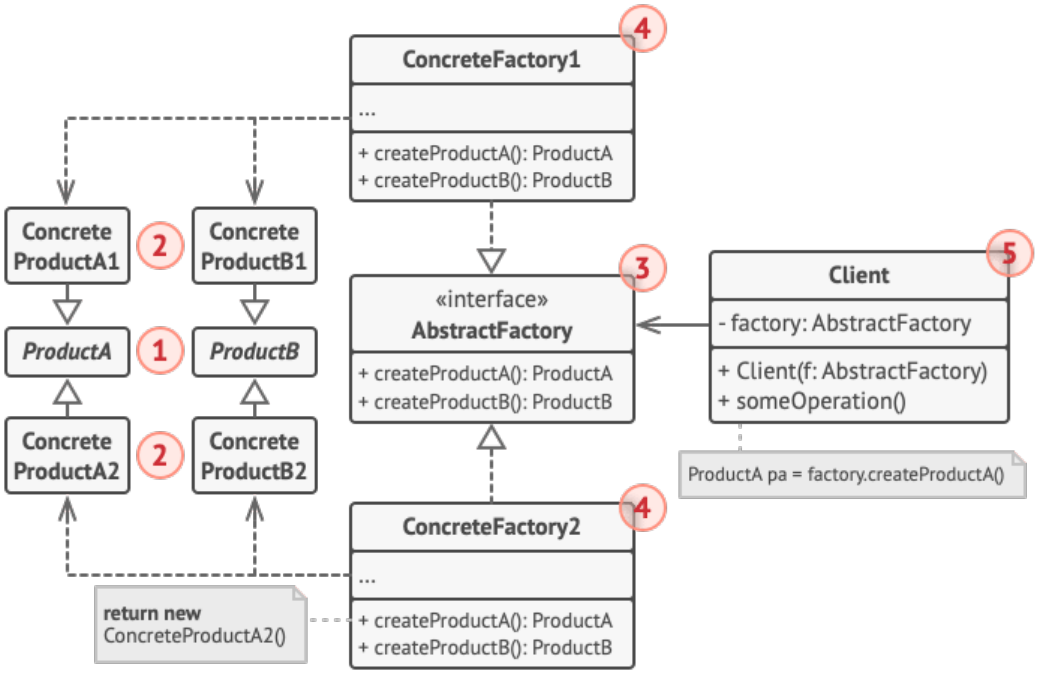
Второй типичный пример использования абстрактной фабрики — создание кроссплатформенных элементов пользовательского интерфейса. Она показывает одни и те же элементы интерфейса, которые выглядят не совсем одинаково в различных операционных системах. В такой программе важно, чтобы все создаваемые элементы всегда соответствовали текущей операционной системе.



Вначале программа определяет, какая из фабрик соответствует текущей операционной системе. Затем создаёт нужную фабрику и возвращает её клиентскому коду. В дальнейшем клиент будет работать только с этой фабрикой, чтобы исключить несовместимость возвращаемых элементов интерфейса. Чтобы добавить в программу новую вариацию элементов (например, для поддержки Linux), достаточно создать новую фабрику, производящую эти элементы.

Таким образом, в общем виде шаблон «Абстрактная фабрика» имеет следующую структуру.

1. ***Абстрактные продукты*** — объявляют интерфейсы продуктов, которые связаны друг с другом по смыслу, но выполняют разные функции.
2. ***Конкретные продукты*** – большой набор классов, которые относятся к различным абстрактным продуктам (кресло/столик), но имеют одни и те же вариации (Викторианский/Модерн).
3. ***Абстрактная фабрика*** объявляет методы создания различных абстрактных продуктов (кресло/столик).
4. ***Конкретные фабрики*** относятся каждая к своему семейству продуктов (Викторианский/Модерн) и реализуют методы абстрактной фабрики, позволяя создавать все продукты определённого семейства.
5. ***Клиент***, который сможет работает с любыми семействами продуктов через абстрактные интерфейсы.



Сильные стороны шаблона «Абстрактная фабрика»:

- поддерживается сочетаемость продуктов, моделируемых в программе, без необходимости жестко привязывать клиентский код к конкретным классам продуктов;
- код создания объектов конкретного семейства сосредоточен в одном классе, что упрощает поддержку кода;
- достаточно просто реализуется добавление новых семейств продуктов в программу.

Слабые стороны «Абстрактной фабрики»:

- код программы усложняется из-за введения множества дополнительных классов;
- в каждом семействе нужна поддержка всех типов продуктов типов продуктов в каждой вариации.

«Строитель»[89]

Стандартный способ создания объекта – вызов конструктора класса, в том числе конструктора с параметрами. Но бывают сложные объекты, начальная инициализация которых - сложный, многоэтапный процесс. Если реализовывать его в конструкторе, понадобится длинный перечень параметров и сложная реализация, противоречащая правилам декомпозиции кода. Появляются основания реализовать каждый этап при помощи отдельного метода, но:

- все эти методы должны быть вызваны при создании объекта;
- и в правильной последовательности.

Для решения подобного рода задач и используется ***шаблон "Строитель"***.

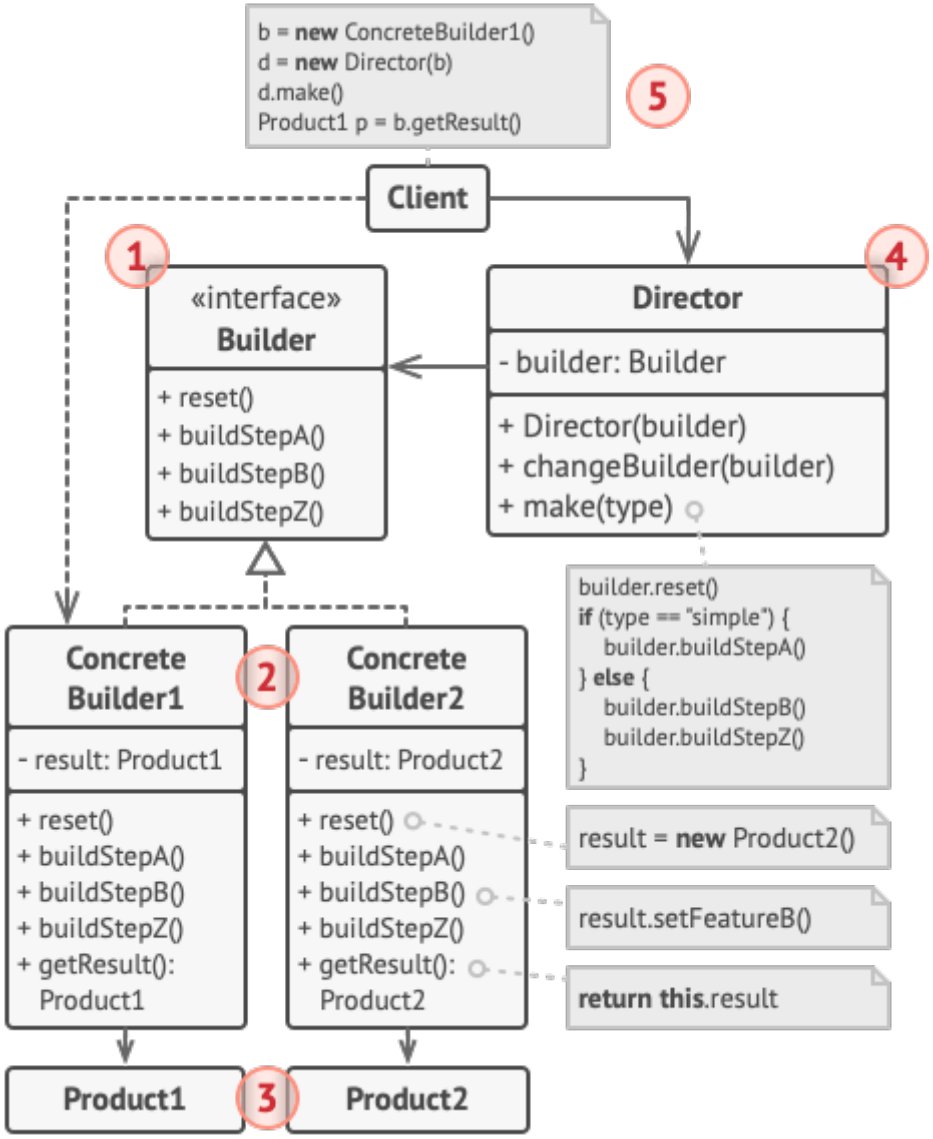
---

8 https://habr.com/ru/company/otus/blog/552412/

9 https://radioprog.ru/post/1465

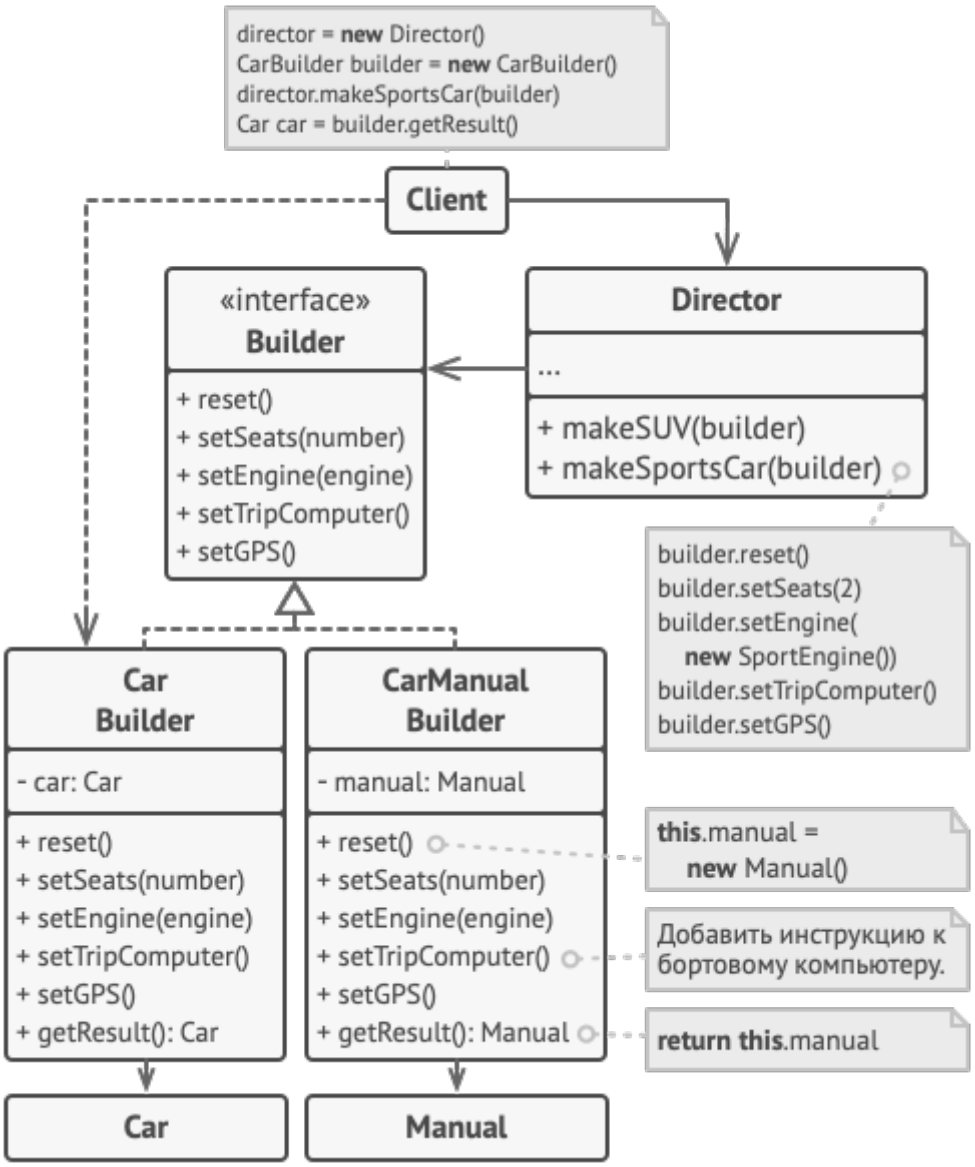
Шаблон включает:

- класс сложного ***объекта-продукта***;
- абстрактный класс/интерфейс ***строителя***, который определяет все этапы, необходимые для производства сложного объекта-продукта; здесь объявляются все этапы создания продукта, а их реализация относится к классам конкретных строителей;
- ***конкретные классы-строители*** - разными способами реализуют абстрактные методы абстрактного объекта-строителя;
- ***распорядитель*** - супервизионный класс, под контролем которого строитель выполняет скоординированные этапы для создания объекта-продукта.



Пример: строитель для пошагового конструирования автомобилей. Автомобиль – это сложный объект, который может быть сконфигурирован сотней разных способов. Вместо того чтобы настраивать автомобиль через конструктор, его сборка выносится в отдельный класс-строитель, в котором

предусматриваются методы для конфигурации всех частей автомобиля. Клиент поручает сборку автомобиля директору, который знает, какие шаги строителя нужно вызвать, чтобы получить несколько самых популярных конфигураций автомобилей.



К каждому автомобилю нужно руководство (*Manual*), ориентированное на его конфигурацию. Для руководства создается ещё один класс строителя, который вместо конструирования автомобиля печатает страницы руководства к той детали, которая встраивается в продукт. Пропустив оба типа строителей через одни и те же шаги, мы получим автомобиль и подходящее к нему руководство пользователя.

Основное различие между строителем и абстрактной фабрикой: строитель предоставляет подробный и детальный контроль над процессом создания объекта. Шаблон "Абстрактная фабрика" отвечает на вопрос «что», "Строитель" - «как».

5.6.2. Структурные шаблоны

К структурным шаблонам относятся:

- «Легковесный элемент» (Flyweight);
- «Компоновщик» (Composite);
- «Адаптер» (Adapter);
- «Заместитель» (Proxy);
- «Фасад» (Facade);
- «Мост» (Bridge);
- «Декоратор» (Decorator).

«Легковес»[10]

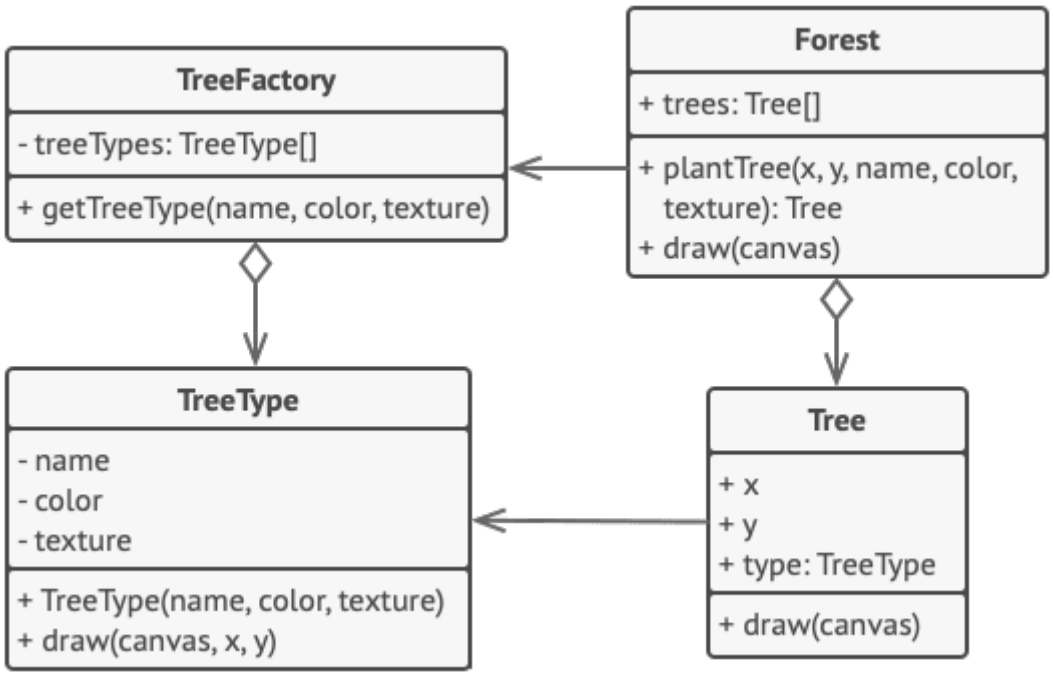
Основное назначение *шаблона «Легковес» (Flywieght)* — компактное хранение множество однотипных объектов за счет правильной декомпозиции классов.

Один из наиболее убедительных примеров использования шаблона — приложения, активно использующие графику, в том числе игры. Такие приложения включают огромное количество объектов, которые должны быть представлены в оперативной памяти.

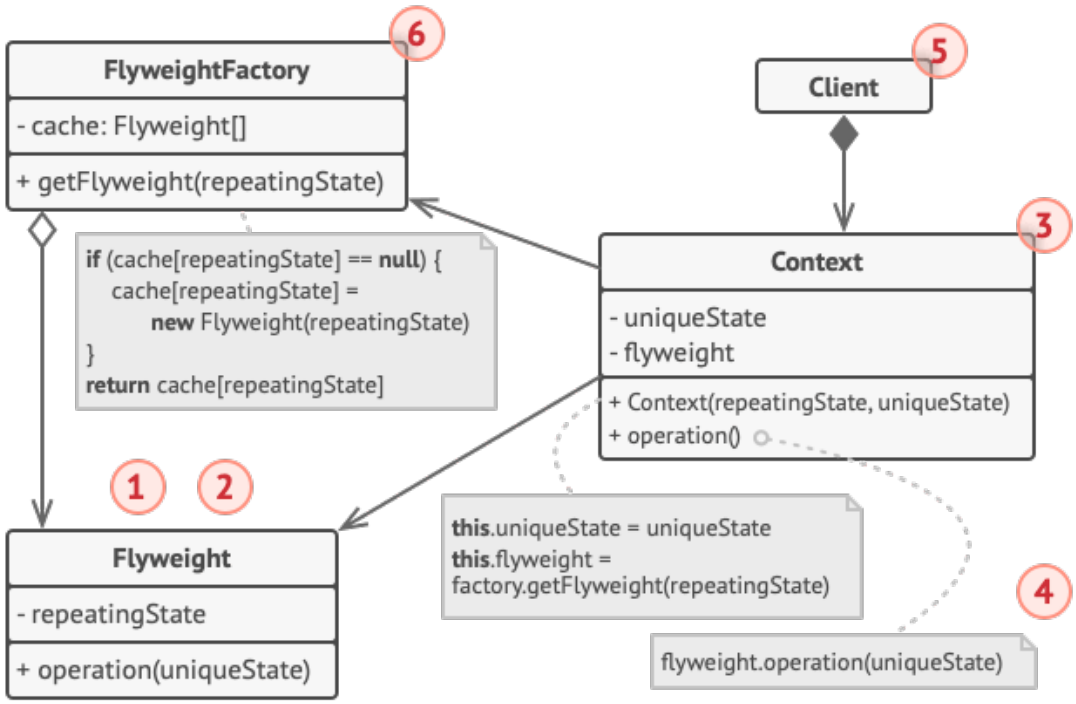
В качестве примера рассмотрим лес, состоящий из множества деревьев. Эти деревья должны быть отрисованы на экране. Каждое дерево содержит набор значений параметров отрисовки и других атрибутов, а именно: название дерева, цвет, текстуру, координаты *x* и *y*. При внимательном рассмотрении мы можем видеть, что строго индивидуальными для каждого дерева являются координаты *x* и *y*. Что касается названия, цвета и текстуры, то значения этих атрибутов повторяются у разных деревьев. Причем повторяются не только значения отдельных атрибутов, но и комбинации значений: можно предположить, что для всех берез сочетание «название — цвет — текстура» будут одинаковыми; для сосен — отличающимися от берез, но одинаковыми для всех сосен.

Хранение всех свойств дерева в экземпляре класса «Дерево» приведет к неэкономному использованию оперативной памяти и, возможно, к нестабильной работе игры вплоть до аварийного завершения. Гораздо экономнее память будет использована, если вынести название дерева, цвет и текстуру в отдельный класс, как показано на рисунке: *Tree* – это дерево, *TreeType* – тип дерева, причем каждое дерево содержит ссылку на конкретный тип.

10 https://radioprog.ru/post/1486



Обобщая сказанное, опишем структуру шаблона в общем виде.



Собственно ***легковес*** (***Flyweight***) — содержит ***внутреннее состояние***, которое повторялось во множестве первоначальных объектов. В примере, рассмотренно выше, легковес — это *TreeType*.

Контекст (***Context***) - содержит «внешнюю» часть состояния, уникальную для каждого объекта. Контекст связан с одним из объектов-легковесов, хранящих оставшееся состояние. Один и тот же легковес можно использовать в связке со множеством контекстов.

Клиент (***Client***) - вычисляет или хранит контекст, то есть внешнее состояние легковесов. Для клиента легковесы выглядят как шаблонные

объекты, которые можно настроить во время использования, передав контекст через параметры.

Фабрика ***легковесов*** (***FlyweightFactory***) управляет созданием и повторным использованием легковесов. Фабрика получает запросы, в которых указано желаемое состояние легковеса. Если легковес с таким состоянием уже создан, фабрика сразу его возвращает, а если нет – создаёт новый объект.

Поведение оригинального объекта чаще всего оставляют в легковесе, передавая значения контекста через параметры методов. Тем не менее, поведение можно поместить и в контекст, используя легковес как объект данных.

«Компоновщик»[11]

Шаблон «Компоновщик» (Composite) знаком нам по теме «Графический интерфейс пользователя» и может быть охарактеризован достаточно коротко: он позволяет задать иерархию компонентов, в которой существуют классы простых элементов и классы контейнеров. И простые элементы, и контейнеры реализуют единый интерфейс. Контейнер поддерживает основные возможности простых компонентов и при этом содержит методы добавления компонентов контейнер (вложенность контейнеров поддерживается автоматически), удаления компонента из контейнера и методы массовой обработки вложенных компонентов - например, метод прорисовки GUI-контейнера может вызывать методы прорисовки всех вложенных в него компонентов. Способы реализации этого шаблона:

- класс контейнера расширяет класс компонента - см. AWT, SWING;
- классы простых компонентов и контейнеров расширяют один и тот же интерфейс, который объявляет методы работы с компонентами, а класс контейнера добавляет к ним методы добавления, удаления и массовой обработки.

«Адаптер»[12]

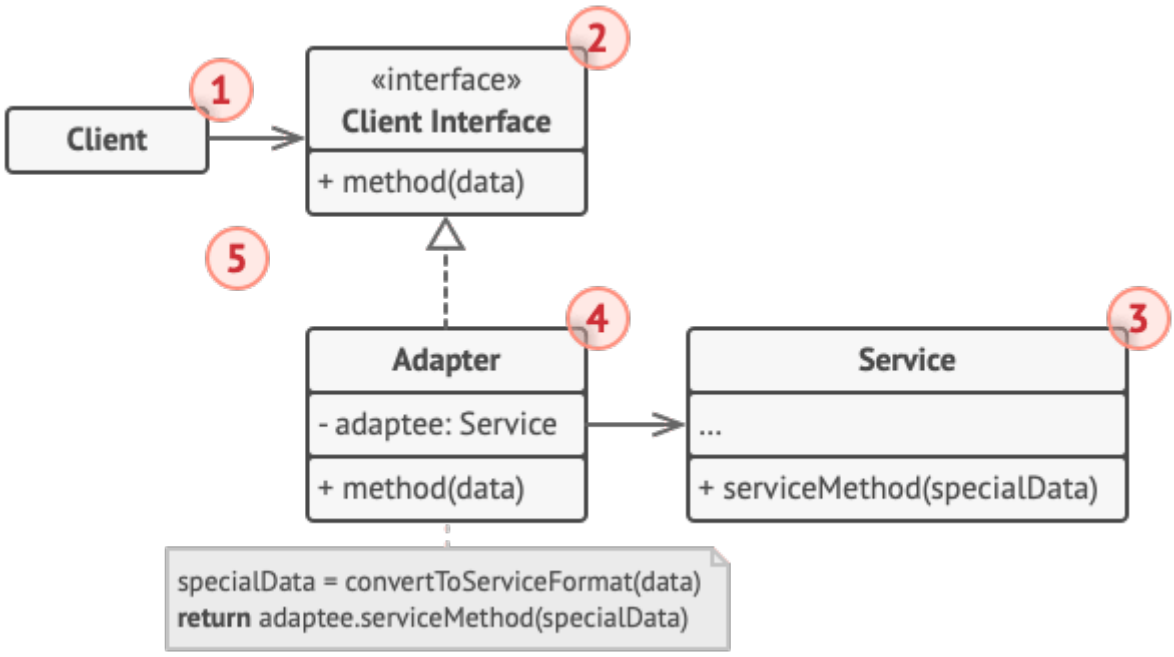
Данный шаблон предназначен для обеспечения совместной работы объектов с несовместимыми интерфейсами.

Предположим, приложение выполняет обработку данных в формате XML. Для его улучшения разработчик желает подключить сторонний модуль анализа данных, однако этот модуль принимает данные в формате JSON. Переделать код аналитического модуля нет возможности.

Шаблон "Адаптер" (Adapter) основан на использовании объекта-переводчика (именно его и называют адаптером), задача которого - обеспечить совместимость интерфейсов и/или форматов данных.

11 https://radioprog.ru/post/1480

12 https://radioprog.ru/post/1471



Структура шаблона при использовании только одиночного наследования включает следующие компоненты.

1. ***Клиент*** - класс, который реализует существующие прикладные функции программы.
2. ***Клиентский интерфейс*** - содержит методы работы клиента с другими классами.
3. ***Сервис*** - некий полезный класс, с которым нужно обеспечить совместимость. Часто сервис принадлежит стороннему разработчику, хотя и ситуация, когда совсем сервис и клиент принадлежат одному разработчику, не исключена.
4. ***Адаптер*** - класс, который реализует клиентский интерфейс. Он получает вызовы от клиента через методы клиентского интерфейса, и переводит эти вызовы в понятную форму для сервиса.

Работа с адаптером через клиентский интерфейс позволяет создавать различные реализации адаптеров, не привязывая код клиента к конкретному классу адаптера.

«Заместитель»[13]

Шаблон "Заместитель" (Proxy) похож на "Адаптер" в том плане, что он также предоставляет программе вместо оригинального объекта некий замещающий объект. Но с программой точки зрения данный шаблон отличается тем, что и оригинальный, и замещающий объекты реализуют один и тот же интерфейс. Кроме того, заместитель находится в отношении слабой агрегации с оригиналом: заместитель - целое, оригинал — часть.

Шаблон "Заместитель" применяется при необходимости:

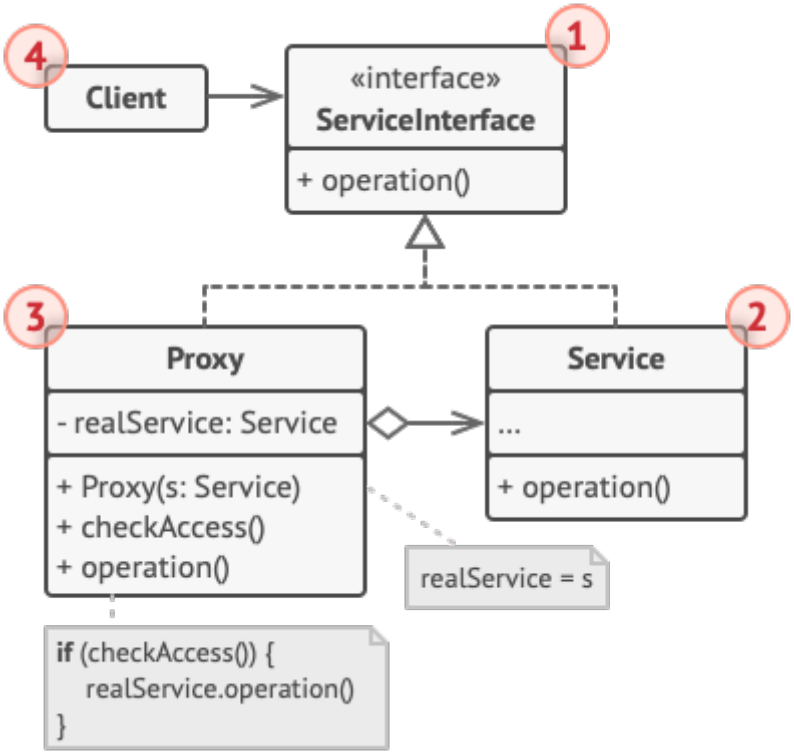
- "ленивой инициализации" некоторого "тяжелого" ресурса: базы данных, файла большого объёма и т.п.; вместо моментальной загрузки

13 https://radioprog.ru/post/1488

"тяжелого" объекта можно выполнить её, когда объект действительно понадобится;

- защиты доступа к оригинальному объекту (защищающий прокси): замещающий объект может проверять правомочность доступа при каждом вызове метода и передавать вызов оригинальному объекту только если доступ разрешён;
- локального запуска сервиса: если оригинальный объект находится на удалённом сервере, он загружается один раз, далее работа идёт с заместителем;

логирования запросов: заместитель сохраняет историю запросов к оригинальному объекту.



В структуру шаблона входят следующие интерфейсы и классы.

1. ***Интерфейс сервиса*** – общий интерфейс для оригинального объекта и заместителя.

2. ***Сервис*** - оригинальный объект, содержит полезные прикладные функции.

3. ***Замещающий объект*** - хранит ссылку на объект сервиса. Передает ему вызовы со стороны клиента после того как сам закончит свою работу: инициализацию, защиту, логирование и т.п. Может сам отвечать за создание и удаление объекта сервиса.

4. ***Клиент***, который работает с объектами через интерфейс сервиса.

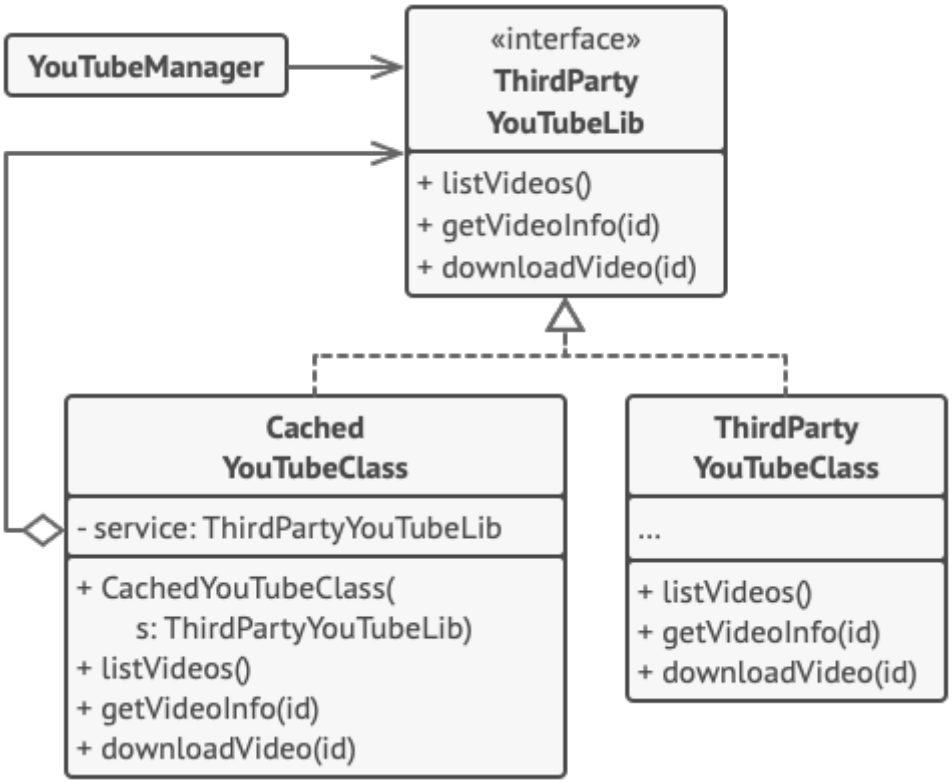
Пример применения шаблона "Заместитель": приложение, которое выполняет "ленивую" инициализацию и кэширование результатов работы библиотеки интеграции с видеосервисом в Интернете.

Класс *YouTubeManager* - клиент. Интерфейс сервиса имеет название *YouTubeLib*. Содержит описания методов

получения списка роликов, получения информации о конкретном ролике и загрузки ролика.

Интерфейс реализуется двумя классами:

- *ThirdPartyYouTubeClass* – класс объекта сервиса, в нем метод загрузки видео скачивает ролик непосредственно из Интернета, даже если ролик загружался ранее.
- *CashedYouTubeClass* - класс замещающего объекта. Замещающий объект загружает видео из сети только при первом обращении, обращаясь для этого к объекту сервиса. В остальных случаях возвращает кэшированный файл.



«Фасад»[14]

В сложных программных системах объекту-клиенту приходится взаимодействовать со сложными системами классов, в том числе библиотеками или фреймворками. Зачастую при этом клиент должен придерживаться правильной последовательности создания и инициализации объектов с учетом всех зависимостей между ними. Это может существенно перегружать код клиента и делать его сильно зависимым от конкретной реализации библиотеки/фреймворка.

Шаблон "Фасад" (Facade) разгружает класс клиента, обеспечивая простой интерфейс для взаимодействия клиента с другими классами.

Аналогия из жизни: заказ товара в магазине по телефону или через сайт. Сотрудник службы поддержки (или Интернет-сайт) является "фасадом", который упрощает процесс взаимодействия покупателя с магазином. Без "фасада" покупателю пришлось бы самому лично работать с системой создания

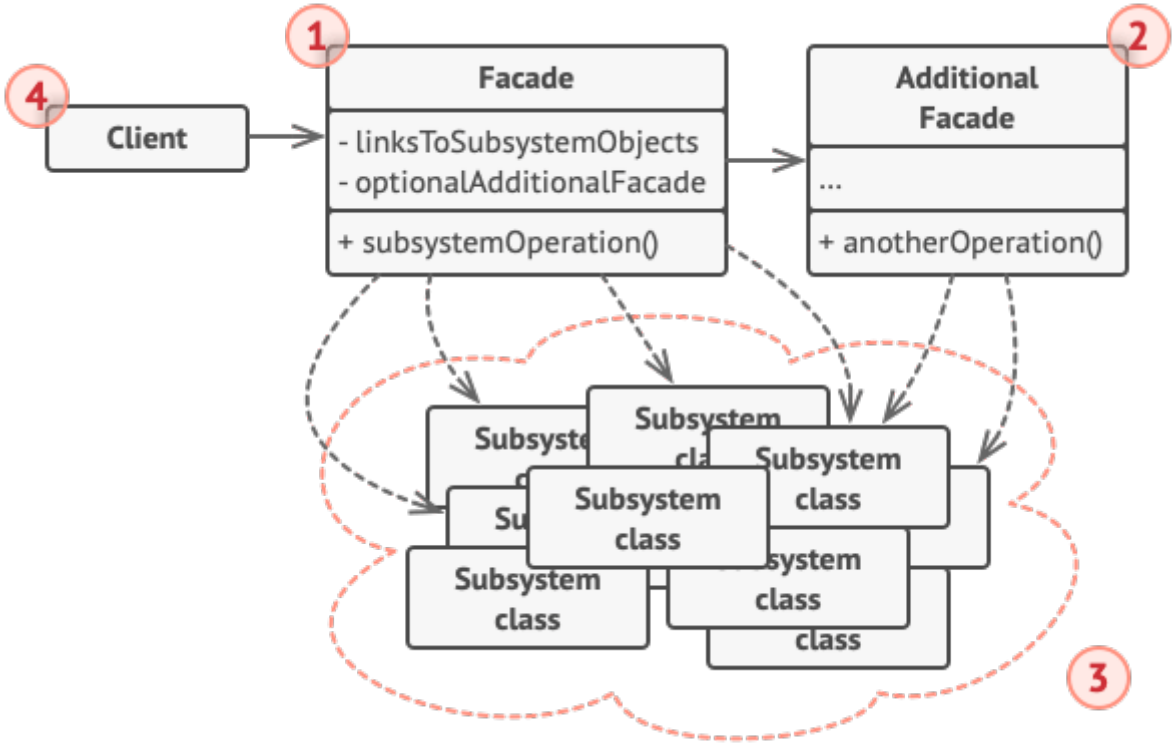
14 https://radioprog.ru/post/1484

заказа, складом, отделом доставки, платёжной системой и т.д., вникая в различные детали.

Структура шаблона в базовом варианте включает три звена:

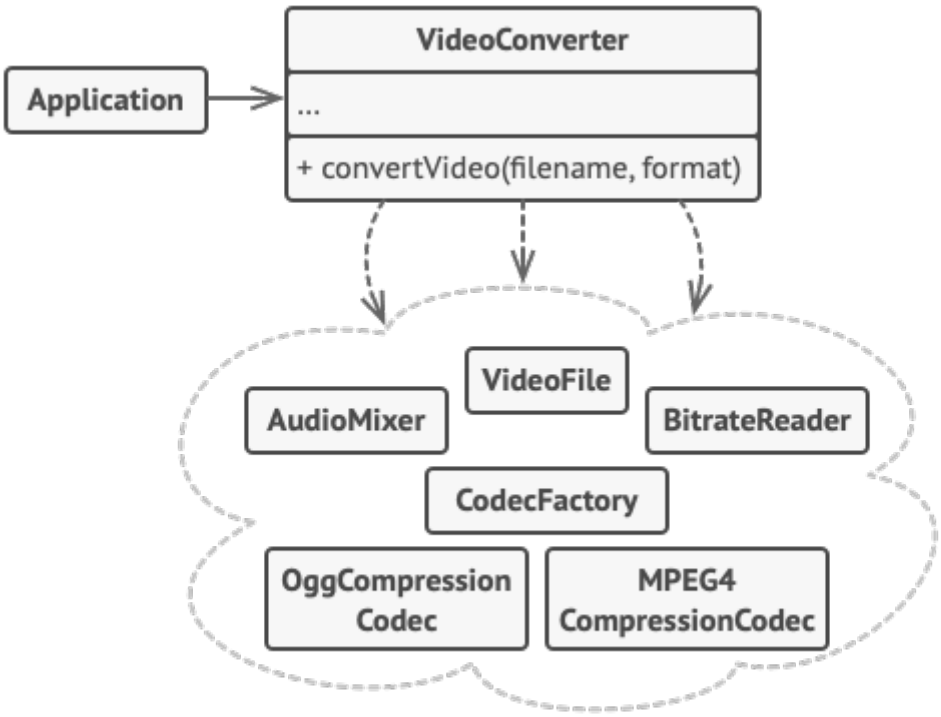
- сложную подсистему классов, реализующих бизнес-логику;
- собственно фасад, который предоставляет доступ к определённой функциональности подсистемы; он "знает", каким классам нужно переадресовать вопрос и какие данные для этого нужны;
- клиент.

Также могут присутствовать дополнительные фасады, цель применения которых – разделить "зоны ответственности". Например, один из фасадов отвечает за продажу товаров, другой за доставку, третий за поступление в магазин и т.п. Дополнительные фасады могут использоваться как непосредственно клиентами, так и другими фасадами.



С шаблоном "Фасад" хорошо совместим шаблон "Одиночка": в программе достаточно иметь один экземпляр основного фасада и по одному экземпляру дополнительных.

В качестве примера можно привести фасад для работы со сложным фреймворком видеоконвертации. Фасад *VideoConverter* предоставляет коду приложения единственный метод для конвертации видео, который сам заботится о том, чтобы правильно сконфигурировать нужные объекты фреймворка и получить требуемый результат.



Принципиальное отличие "Фасада" от "Адаптера" и "Заместителя" в том, что он определяет собственный интерфейс взаимодействия с подсистемой, а не дублирует интерфейсы сервисных объектов.

«Мост»[15]

Описание шаблона ***"Мост" (Bridge)*** может быть сделано на примере GUI: контейнеры и менеджеры компоновок.

Методы размещения компонентов со всеми деталями теоретически можно было бы возможно реализовать непосредственно в классах контейнеров, но это лишало бы их необходимой гибкости. Шаблон «Мост» отделяет иерархию контейнеров от иерархии компоновок. Изменения в одной иерархии не приводят к изменениям в другой.

Говоря о шаблоне "Мост", пользуются терминами "абстракция" и "реализация".

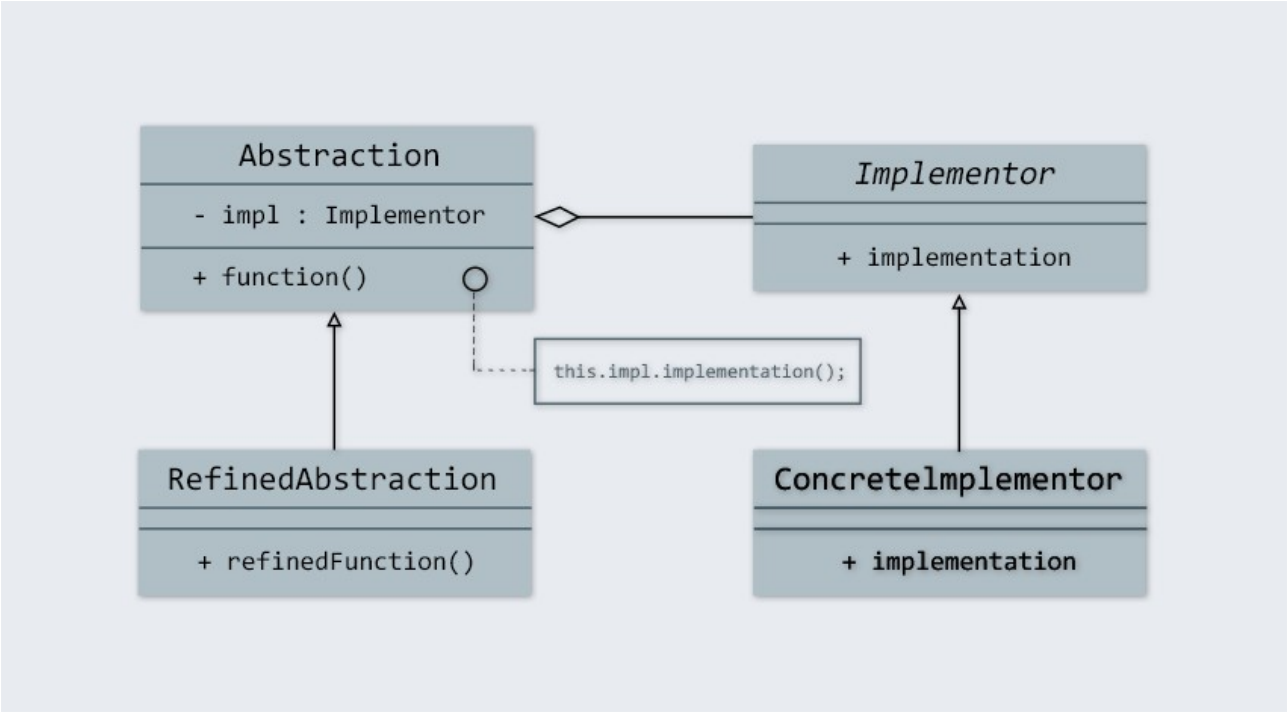
Абстракцией в нашем примере является контейнер. Классы *Panel*, *Window*, *Dialog* называют ***уточненной абстракцией***.

Интерфейс *LayoutManager* - ***реализация абстракции*** в том смысле, что компоновка призвана придать контейнеру конкретную форму, благодаря чему контейнер перестает быть "вообще контейнером", т.е. неким "обтекаемым" вместилищем компонентов.

Диаграмму классов и интерфейсов шаблона в общем виде можно представить следующим образом[16]:

15 https://habr.com/ru/company/vk/blog/325492/

16 https://javarush.com/groups/posts/2570-znakomstvo-s-patternom-proektirovanija-bridge



«Декоратор»[17]

Основной и фундаментальный механизм расширения функциональности классов в ООП – наследование. Однако он имеет недостатки:

- статичность - наследование не позволяет динамически расширить функциональность объектов в ходе выполнения программы;
- невозможность создания объекта, в котором было бы реализовано поведение сразу нескольких подклассов некоторого суперкласса.

В реальной жизни наследование не единственный механизм, который позволяет приобретать объектам дополнительные свойства. Например, когда человек надевает свитер, он становится более защищенным от холода, но не благодаря созданию нового подкласса "Человек в свитере", а в результате взаимодействия со свитером как неким объектом-обёрткой. Свитер в данном примере и является своего рода декоратором.

Шаблон "Декоратор" (***Decorator***; альтернативное название ***"Обёртка"***) позволяет объектам динамически приобретать новые свойства и/или поведение через агрегацию, а не через наследование. Объект-обёртка содержит ссылку на базовый объект. Он вызывает методы базового объекта, добавляя к ним при этом что-то своё.

Структура шаблона включает следующие интерфейсы и классы.

1. ***Компонент*** - общий интерфейс всех обёрток и оборачиваемых объектов.

2. ***Конкретный компонент*** – класс оборачиваемого объекта.

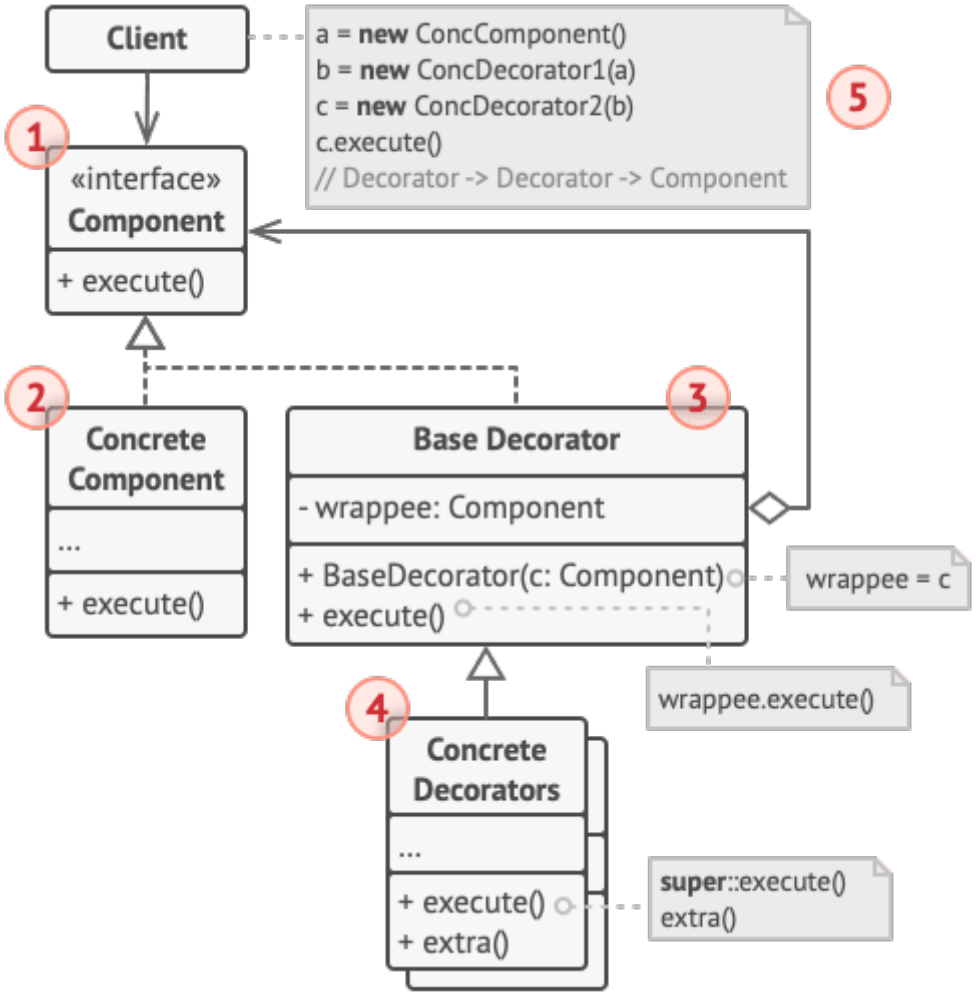
3. ***Базовый декоратор*** - класс (возможно, абстрактный), который определяет свойства и поведение всех типов "обёрток". Содержит ссылку на вложенный объект-компонент. Им может быть как базовый компонент, так и

17 https://radioprog.ru/post/1482

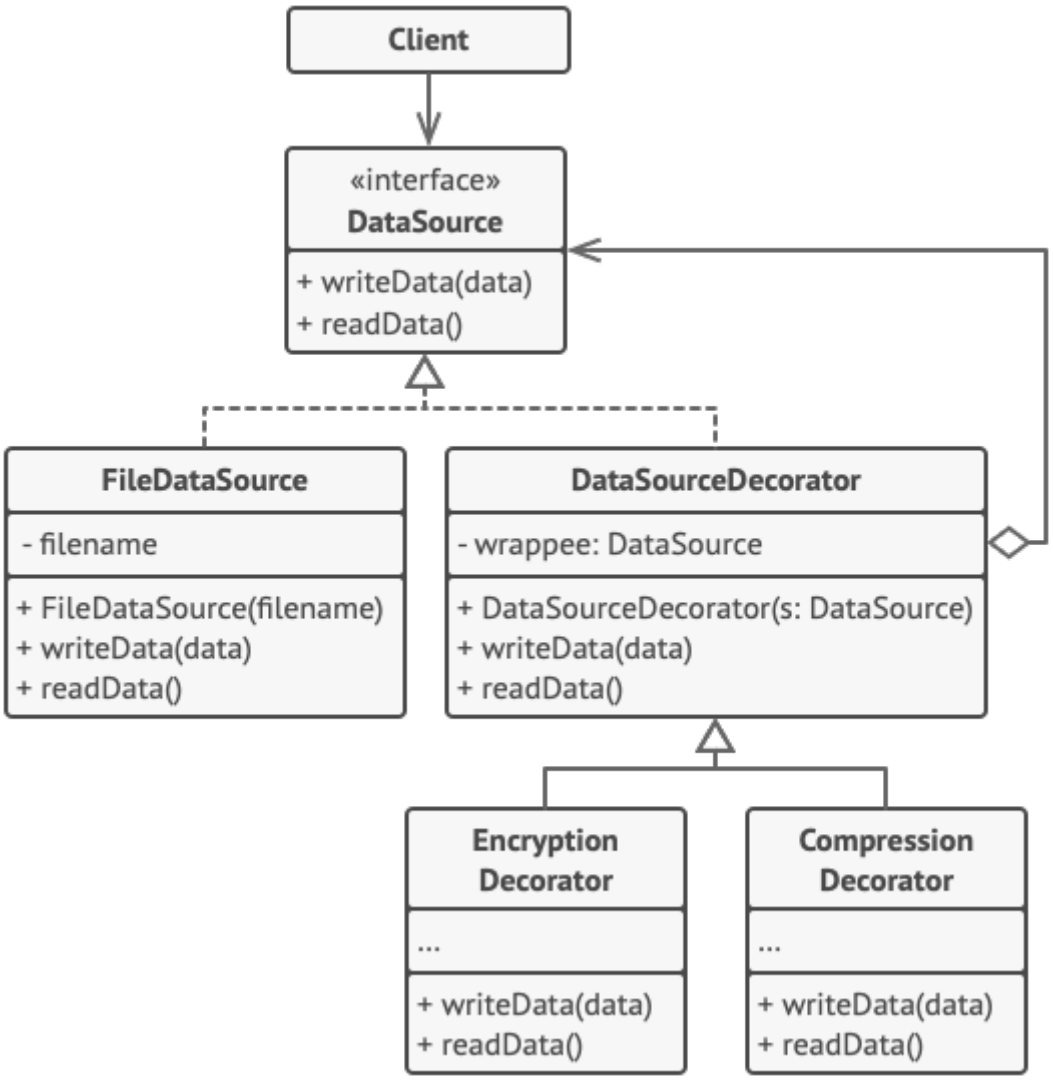
один из декораторов. Базовый декоратор делегирует свои операции вложенному объекту.

4. ***Классы конкретных декораторов*** содержат реализацию дополнительного поведения. Дополнительные действия выполняются до или после вызова аналогичных методов "обернутого" объекта.

5. ***Клиент*** - может "оборачивать" базовые объекты в декораторы или вкладывать одни декораторы в другие, работая со всеми перечисленными объектами через базовый интерфейс компонента.



Пример. В приложении реализовано хранение документов различных пользователей. Одни пользователи согласны хранить документы в незащищенном виде, другие предпочитают шифровать перед сохранением, третьи - сжимать перед сохранением, а четвертым требуется и шифрование, и сжатие. При этом шифрование и сжатие выполняются в прозрачном режиме, то есть дополняют обычный метод сохранения документа.



Отличие "Декоратора" от ранее перечисленных структурных шаблонов:

- "улучшает" поведение объекта без изменения его интерфейса - в отличие от "Адаптера";
- поддерживает рекурсивную вложенность.

5.6.3. Поведенческие шаблоны[18]

К поведенческим шаблонам относятся:

- «Шаблонный метод» (Template Method);
- «Посредник» (Mediator);
- «Цепочка ответственности» (Chain of Responsibility);
- «Наблюдатель» (Observer);
- «Стратегия» (Strategy);
- «Команда» (Command);
- «Состояние» (State);
- «Посетитель» (Visitor);
- «Итератор» (Iterator).

Шаблон «Итератор» изучался в разделе 7 «Структуры данных».

18 https://radioprog.ru/post/1506

Прочие шаблоны имеет смысл рассмотреть более подробно.

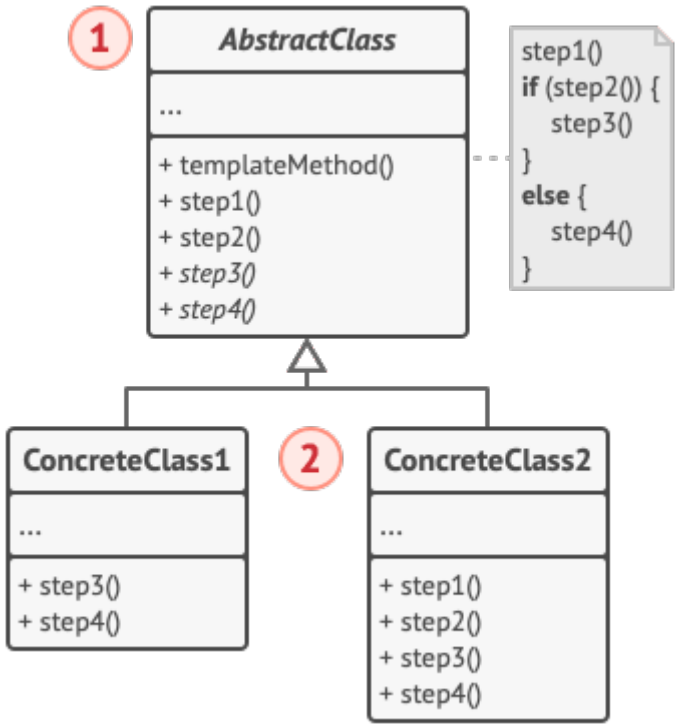
«Шаблонный метод»[19]

При проектировании иерархии наследования классов нередко обнаруживается некий алгоритм, шаги которого удобно реализовывать в виде отдельных методов. Содержание каждого шага зависит от конкретного подкласса, но в целом последовательность шагов во всех подклассах одинакова и может быть определена в суперклассе.

"Шаблонный метод" (Template Method) определяет общую последовательность шагов алгоритма, его "скелет", перекладывая ответственность за некоторые или все отдельные шаги на подклассы.

Структура шаблона включает:

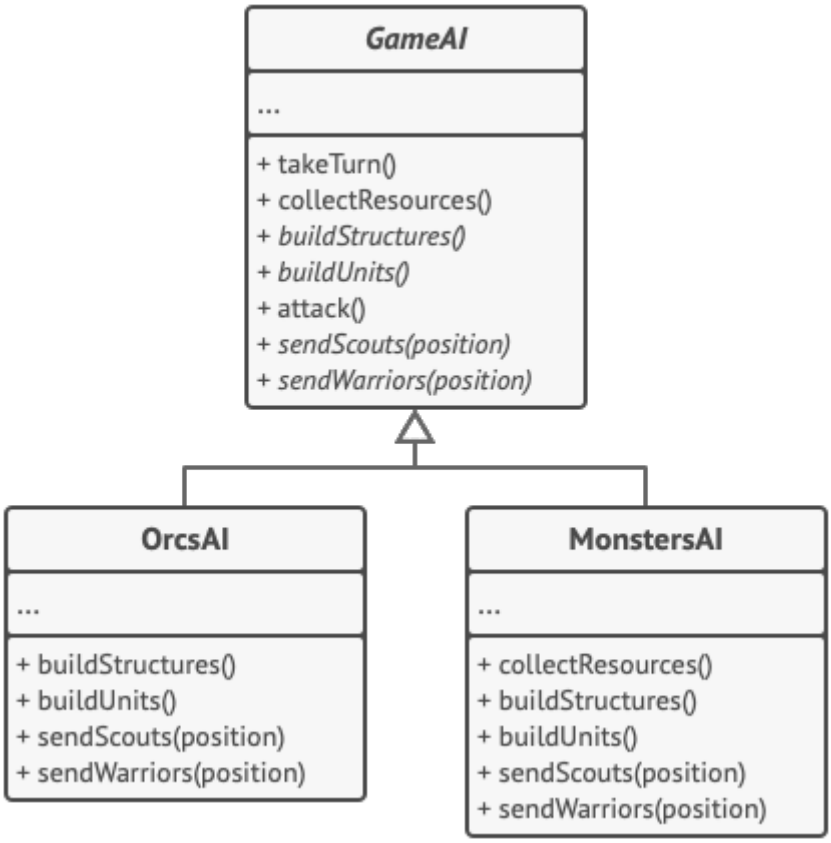
1. ***Абстрактный суперкласс***. Содержит определение ***шаблонного метода*** *templateMethod()*, в котором вызываются ***методы-шаги*** *step1()*, *step2()* и т. д. Некоторые из них могут быть абстрактными. Допускается включение в суперкласс двух и более шаблонных методов.
2. ***Подклассы***, которые переопределяют некоторые или все шаги алгоритма.



Пример: шаблонный метод используется как заготовка для стандартного искусственного интеллекта в простой игре-стратегии. Для введения в игру новой расы достаточно создать подкласс и реализовать в нём наследуемые от суперкласса абстрактные методы. В суперклассе *GameAI* два шаблонных метода: *takeTurn()* и *attack()*. Первый содержит последовательность вызовов *collectResources()*, *buildStructures()*, *buildUnits()* и *attack()*; последний, в свою очередь, вызывает *closestEnimy()*, *sendScouts()* и *sendWarriors()*.

19 https://radioprog.ru/post/1506

Методы *buildStructures()* и *buildUnits()* абстрактные, остальные имеют реализацию в базовом классе.



«Посредник»[20]

Шаблон "Посредник" (Mediator) позволяет уменьшить связанность множества классов между собой за перемещению этих связей в один класс-посредник.

Аналогия из жизни: Пилоты садящихся или улетающих самолётов не общаются напрямую с другими пилотами. Вместо этого они связываются с диспетчером, который координирует действия нескольких самолётов одновременно. Без диспетчера пилотам приходилось бы все время быть начеку и следить за всеми окружающими самолётами самостоятельно, а это приводило бы к частым катастрофам в небе.

Структура шаблона включает следующие классы и интерфейсы.

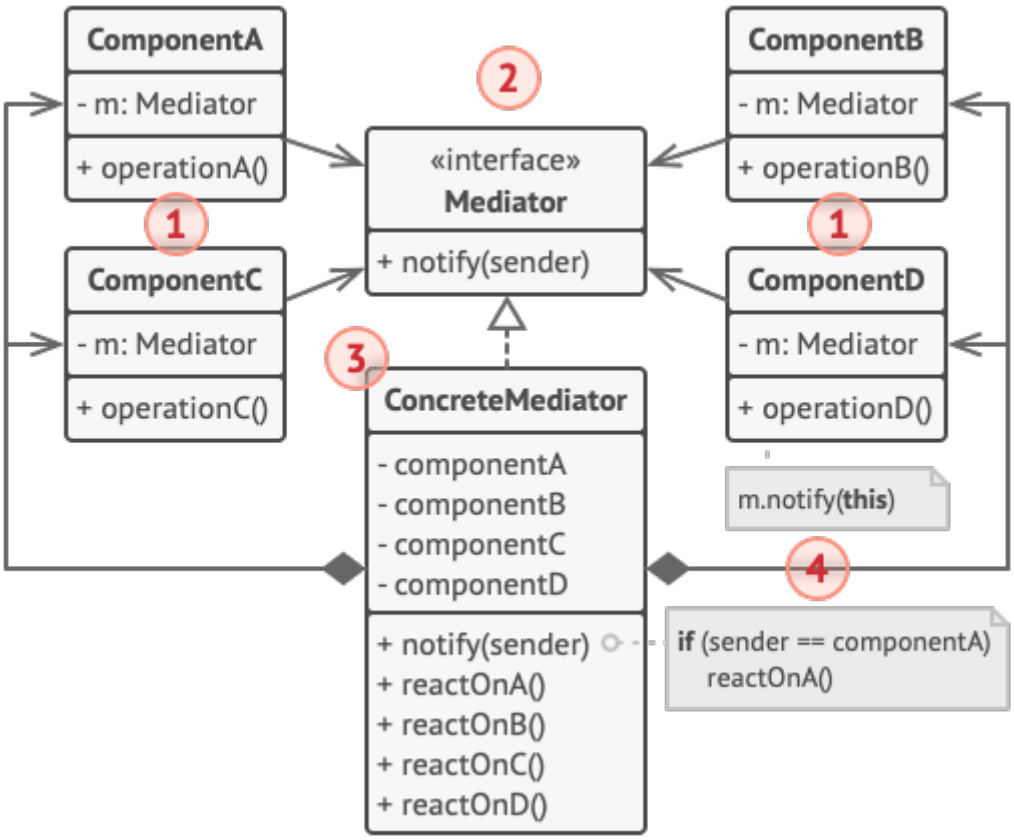
1. ***Классы компонентов***. В них реализуются прикладные функции. Связи между классами-компонентами минимальный или отсутствуют.

2. ***Интерфейс посредника***. Типичный интерфейс посредника содержит метод оповещения о событиях в компонентах, условное наименование *notify()*.

3. ***Класс конкретного посредника*** содержит код взаимодействия нескольких компонентов между собой. Зачастую этот объект не только хранит ссылки на все свои компоненты, но и сам их создаёт, управляя дальнейшим жизненным циклом.

---

20 https://radioprog.ru/post/1496



Компоненты не должны взаимодействовать напрямую. Если в компоненте происходит важное событие, он должен оповестить своего посредника, а тот сам решит, касается ли событие других компонентов, и стоит ли их оповещать. При этом компонент-отправитель не знает, кто обработает его запрос, а компонент-получатель не знает кто его прислал.

«Цепочка ответственности»

Шаблон ***«Цепочка ответственности» (Chain of Responsibility)*** позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передать запрос дальше по цепи.

Хрестоматийный пример: приём онлайн-заказов. В системе есть ограничения, согласно которому делать заказы могут только прошедшие проверку пользователи. Администраторы должны иметь полный доступ к заказам. Уже при такой постановке задачи очевидна последовательность действий: сначала идентификация и аутентификация, потом проверка полномочий. Последняя лишена смысла, если пользователь не прошёл первый этап. Далее в ходе развития программной системы возникает необходимость добавить новые возможности:

- проверка данных в запросе перед внесением в базу: если заказ содержит данные о несуществующих продуктах, нет смысла в его дальнейшей обработке;
- выявление многочисленных попыток входа в систему под одним и тем же именем;

- извлечение формы заказа из кэша, если она уже была однажды показана.

Реализация цепочки проверок в виде одного метода не целесообразна, так как введение новой возможности влечёт изменение кода метода, его существенное усложнение.

Шаблон "Цепочка ответственности" основан на "превращении элементов поведения в объекты". В рассмотренном случае каждая проверка реализуется в отдельном классе с единственным методом выполнения. Данные запроса, над которым происходит проверка, передаются в метод как аргументы. Объекты обработчиков связываются в одну цепь. Каждый из них содержит ссылку на следующий обработчик в цепи. Текущий обработчик после выполнения своей части обработки данных передает эти данные или промежуточный результат обработки следующему объекту в цепочке.

Длина цепочки не ограничивается. Обработка запроса может прерваться на одном из промежуточных звеньев цепочки, причем в основе этого прерывания может лежать одна из двух стратегий:

- прервать обработку при получении отрицательного результата - например, пользователь не пролшёл проверку, продолжать обработку запроса не имеет смысла;
- прервать обработку при получении положительного результата - запрос движется по цепи объектов, пока не найдётся объект, способный его обработать.

Структура шаблона включает следующие интерфейсы и классы.

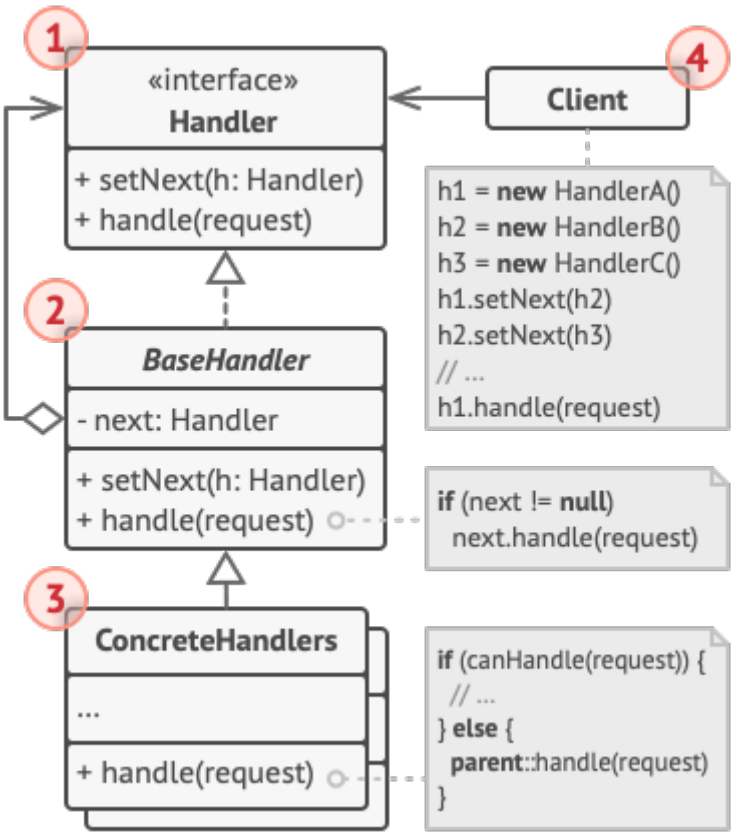
1. ***Обработчик*** - общий интерфейс для всех конкретных обработчиков. Содержит объявление метода обработки запросов.

2. ***Базовый обработчик*** – опциональный класс, который позволяет избавиться от дублирования одного и того же кода во всех конкретных обработчиках.

Обычно этот класс имеет поле для хранения ссылки на следующий обработчик в цепочке.

3. ***Конкретные обработчики*** содержат код обработки запросов. При получении запроса каждый обработчик решает, может ли он обработать запрос, а также стоит ли передать его следующему объекту.

4. ***Клиент*** формирует цепочку обработчиков. Допускается её изменение в ходе выполнения. Клиент может отправлять запросы любому из объектов цепочки, не обязательно первому из них.



«Наблюдатель» (Observer)

Шаблон «Наблюдатель» (Observer) имеет также название ***"Издатель - подписчик"***, которое очень точно передаёт суть и даже не требует дополнительных разъяснений: данный шаблон реализует схему оповещения множества объектов-подписчиков о событиях, происходящих в объекте-издателе.

В структуру шаблона входят:

1. ***Издатель*** – владеет внутренним состоянием, изменение которого интересно отслеживать подписчикам. Содержит коллекцию подписчиков и мтлды оформления/отмены подписки.

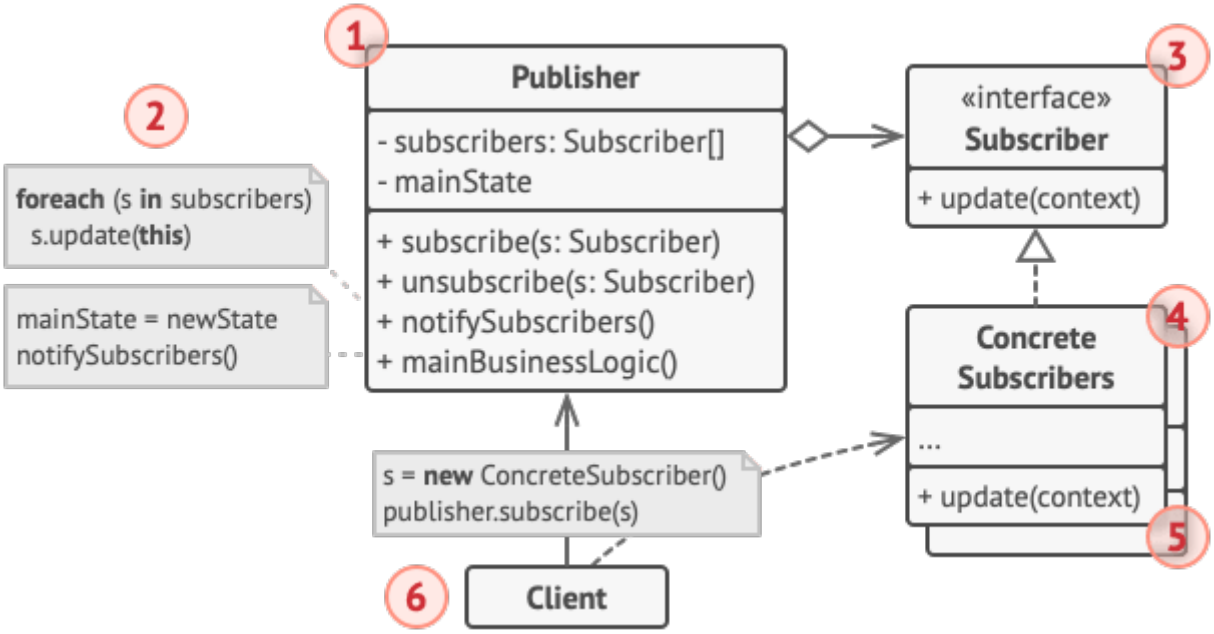
Когда внутреннее состояние издателя меняется, он оповещает своих подписчиков, проходя для этого по списку подписчиков и вызывая их метод оповещения, заданный в общем интерфейсе подписчиков.

2. ***Интерфейс подписчика***, которым пользуется издатель для отправки оповещения. Содержит объявление одного или более методов оповещения.

3. ***Классы конкретных подписчиков***. Описывают реакцию подписчиков на оповещения. Реализуют интерфейс подписчика, благодаря чему издатель работает одинаково с подписчиками всех классов.

Как правило, по приходу оповещения подписчику нужно получить обновлённое состояние издателя. Издатель может передать это состояние через параметры метода оповещения. Более гибкий вариант – передавать через параметры ссылку на объект издателя, чтобы подписчик мог сам получить требуемые данные. Или же подписчик может постоянно хранить ссылку на объект издателя, переданный ему в конструкторе.

4. ***Клиент*** создаёт объекты издателей и подписчиков, а затем регистрирует подписчиков на обновления в издателях.



Данному шаблону близко соответствует модель делегирования событий, где издателем выступает компонент-источник события, а подписчиком - блок прослушивания.

Шаблоны «Стратегия»[21] и "Команда"[22]

Шаблон "Стратегия" (Strategy) определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы. Нетрудно догадаться, что в основу шаблона положены интерфейс (или абстрактный класс) и реализация абстрактных методов в конкретных подклассах.

Пример: пусть имеется приложение-навигатор. Одна из его основных функций - построение маршрута из точки *A* в точку *B*. Можно строить маршрут по отдельности для личного автотранспорта, общественного транспорта, пешего перемещения. Удобно поместить код построения маршрута для различных способов перемещения в отдельные классы. При этом понятно, что во всех случаях речь идёт о реализации одного и того же абстрактного метода.

В структуру шаблона входят следующие интерфейсы и классы.

1. ***Интерфейс стратегии***, который содержит объявление абстрактного метода.

2. ***Классы конкретных стратегий***, которые содержат реализацию метода.

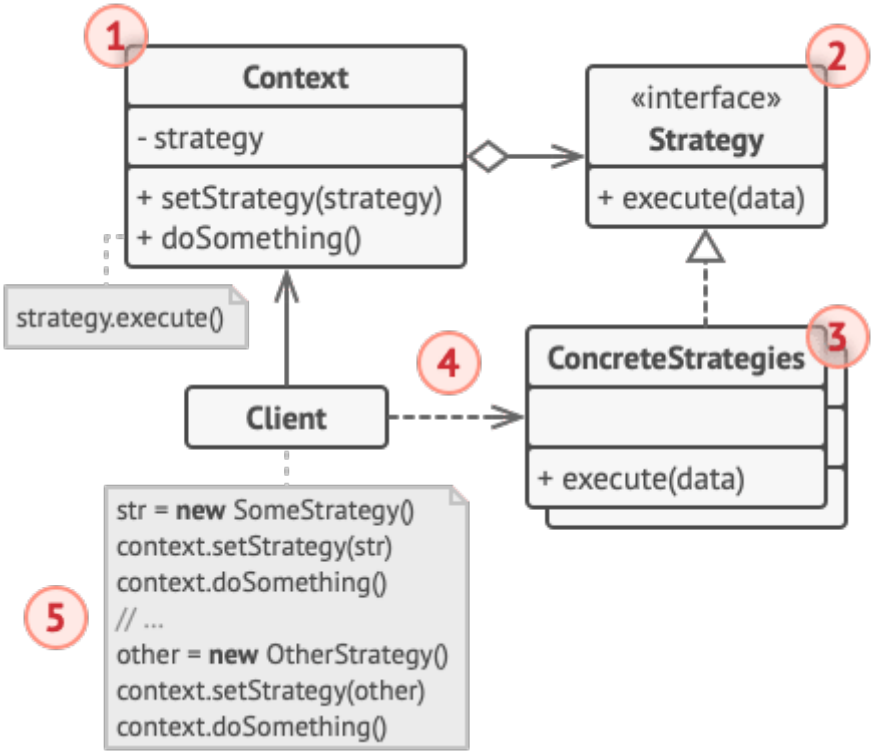
---

21 https://radioprog.ru/post/1504

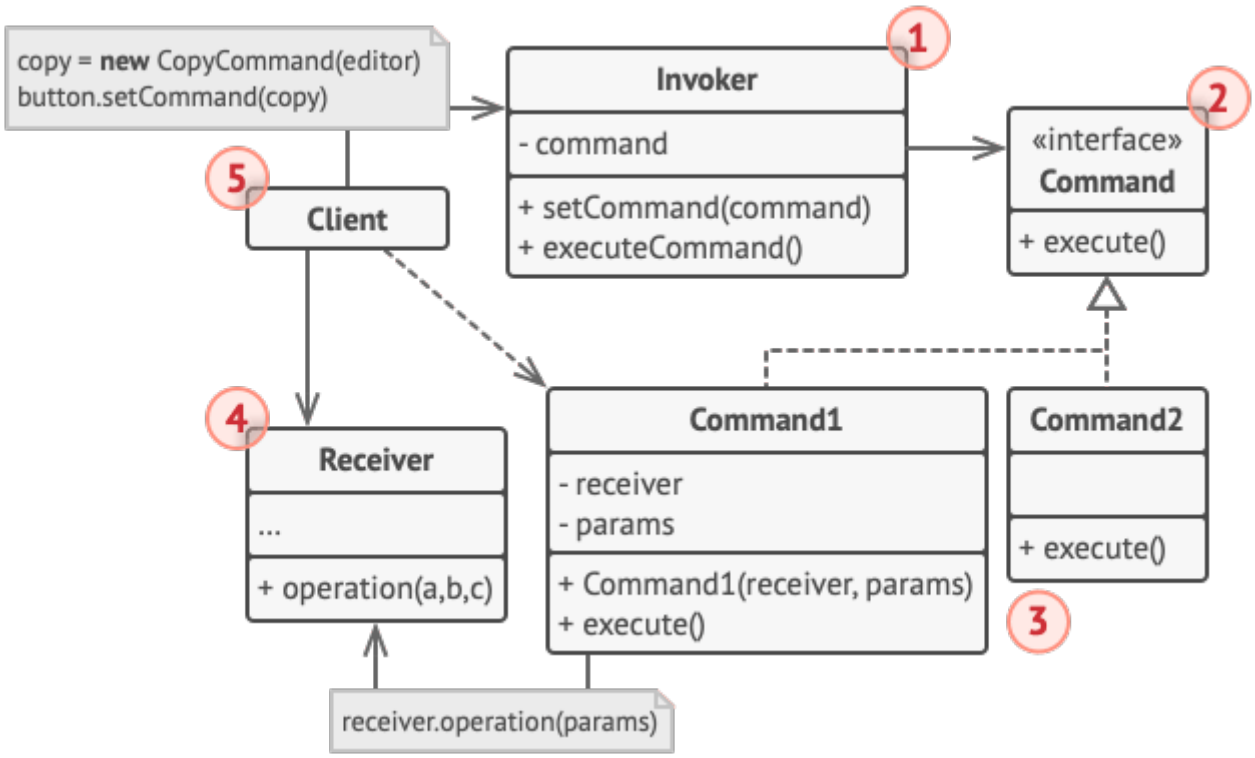
22 https://radioprog.ru/post/1492

3. ***Контекст*** - класс объекта, который хранит ссылку на конкретный объект-стратегию, работая с ним через интерфейс стратегии.

4. ***Класс клиента***. Клиент создаёт объект конкретной стратегии и передать его в конструктор контекста. Может менять объект стратегии через "сеттер" контекста. Во время выполнения программы контекст получает вызовы от клиента и делегирует их объекту конкретной стратегии.



Практически аналогичным образом устроен ***шаблон "Команда" (Command)***, в котором объекты интерпретируются не как стратегии, а как маленькие действия. При описании шаблона часто приводится пример построения текстового редактора. Объекты-действия в нем – копирование, вырезание, вставка фрагмента, отмена последнего действия и др. Действия могут вызываться разными способами: из меню, нажатием комбинации клавиш и т.п. Пользовательский интерфейс может меняться, при этом код, связанный с выполнением действий, остаётся неизменным. Параметры каждого действия и промежуточные результаты хранятся в полях объекта-действия.



Разница между двумя шаблонами заключается в их смысловом содержании. "Стратегия" – это множество разных реализаций одной прикладной функции. "Команда" - множество разных по смыслу объектов, соответствующих одному интерфейсу.

Шаблон «Состояние» (State)[23]

Шаблон «Состояние» (State) основан на представлении программного объекта как автомата с конечным числом состояний (или просто конечного автомата).

В ходе выполнения программы состояния объекта сменяют друг друга. Набор состояний и переходов конечен. Находясь в разных состояниях, объект может по-разному реагировать на одни и те же события, которые происходят с ним.

Например, документ как объект может принимать (предположим) три состояния: "Черновик", "Модерация" или "Опубликован". Есть метод *publish()*, который в каждом из этих состояний работает по-разному:

- из черновика он отправляет документ на модерацию;
- из модерации – в публикацию, но при условии, что именно модератор одобрил документ;
- прямое опубликование из состояния черновика без модерации также возможно, если метод опубликования вызвал администратор, а не обычный пользователь;
- в опубликованном состоянии метод не будет делать ничего.

23 https://radioprog.ru/post/1502

Машину состояний чаще всего реализуют с помощью множества условных операторов, *if* либо *switch*, которые проверяют текущее состояние объекта и выполняют соответствующее поведение. Однако такой способ неудачен, если в ходе развития проекта в документ добавляются всё новые и новые состояния и/или их количество хотя бы больше десяти. Малейшее изменение логики переходов заставит разработчика перепроверять работу всех методов, которые содержат условные операторы машины состояний.

Решение заключается в том, чтобы создать отдельные классы для каждого возможного состояния объекта, а затем вынести туда поведения, соответствующие этим состояниям.

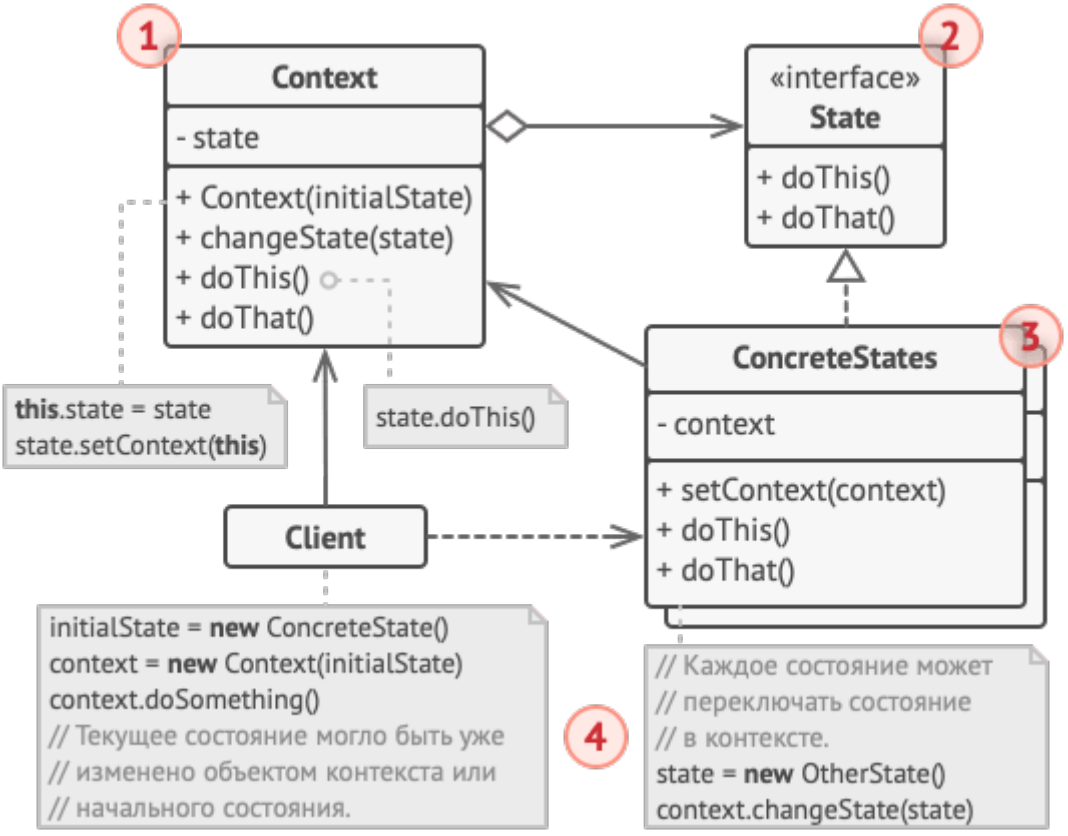
Структура шаблона:

1. ***Интерфейс состояний***.

2. ***Контекст*** хранит ссылку на объект состояния и делегирует ему часть работы, зависящей от состояний.

Контекст работает с этим объектом через общий интерфейс состояний. Контекст должен иметь метод для присваивания ему нового объекта-состояния.

3. ***Конкретные состояния*** реализуют поведения, связанные с определённым состоянием контекста. Иногда приходится создавать целые иерархии классов состояний, чтобы обобщить дублирующий код.



Состояние может иметь обратную ссылку на объект контекста. Через неё не только удобно получать из контекста нужную информацию, но и осуществлять смену его состояния.

И контекст, и объекты конкретных состояний могут решать, когда и какое следующее состояние будет выбрано. Чтобы переключить состояние, нужно

подать другой объект-состояние в контекст. Этим шаблон "Состояние" отличается от "Стратегии" и других шаблонов, имеющих схожую структуру.

Шаблон «Посетитель»

Шаблон "Посетитель" (Visitor) - способ определения дополнительных действий над объектами без внесения изменений в их классы и без расширения этих классов.

Предположим, разрабатывается приложение, работающее с геоданными в виде графа. Узлами графа являются городские локации: памятники, театры, рестораны, важные предприятия и прочее. Каждый узел имеет ссылки на другие, ближайшие к нему узлы. Каждому типу узлов соответствует свой класс, а каждый узел представлен отдельным объектом. Стоит задача сделать экспорт этого графа в XML. Добавлять полиморфный метод экспортирования в классы по каким-либо причинам нежелательно .

Решение: разместить новое поведение в отдельном классе – "посетителе" (*Visitor*). Объекты, с которыми должно было быть связано поведение, не будут выполнять его самостоятельно. Вместо этого ссылки на узлы графа должны передаваться в методы. Обычно это перегружаемый метод *visit()*, каждая отдельная реализация которого работает с объектом конкретного класса.

Бросается в глаза ограниченная применимость данного шаблона: расширение иерархии узлов графа приводит к необходимости добавлять новые методы в интерфейс посетителя и реализовывать их в каждом классе конкретных "посетителей". Следовательно, шаблон больше подходит для работы с завершенными иерархиями классов.

Структура шаблона:

1. ***Интерфейс посетителя***. Объявляет набор методов, отличающихся типом входящего параметра, которые нужны для запуска операции для всех типов конкретных элементов. В языках, поддерживающих перегрузку методов, эти методы могут иметь одинаковые имена, но типы их параметров должны отличаться.

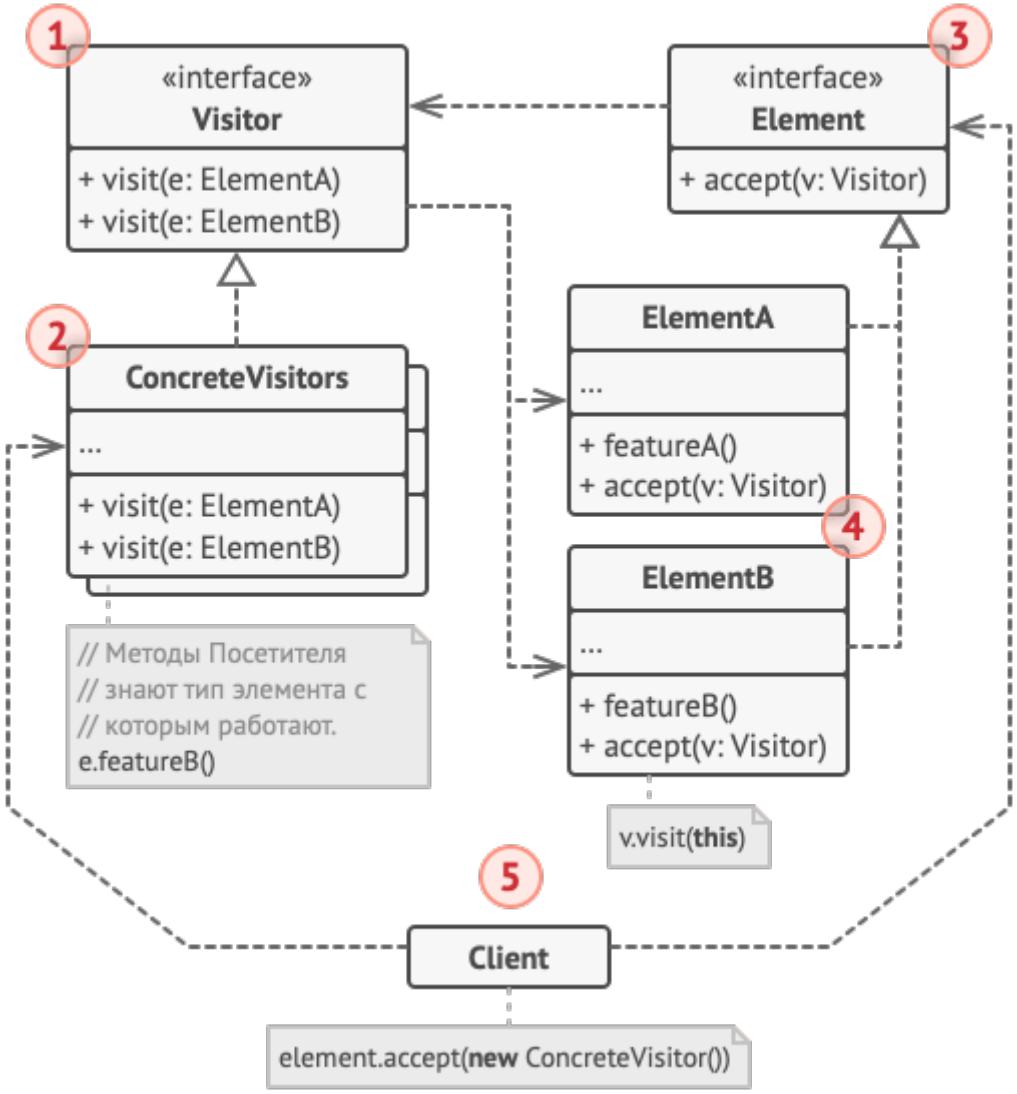
2. ***Классы конкретных посетителей*** реализуют какое-то особенное поведение для всех типов элементов, которые можно подать через методы интерфейса посетителя.

3. ***Интерфейс элемента***. Описывает метод принятия посетителя. Этот метод должен иметь единственный параметр, объявленный с типом интерфейса посетителя.

4. ***Классы конкретных элементов*** реализуют методы принятия посетителя. Цель этого метода – вызвать тот метод посещения, который соответствует типу этого элемента. Так посетитель узнает, с каким именно элементом он работает.

5. ***Клиентом*** зачастую выступает коллекция или сложный составной объект, например, дерево Компоновщика. Зачастую клиент не привязан к

конкретным классам элементов, работая с ними через общий интерфейс элементов.



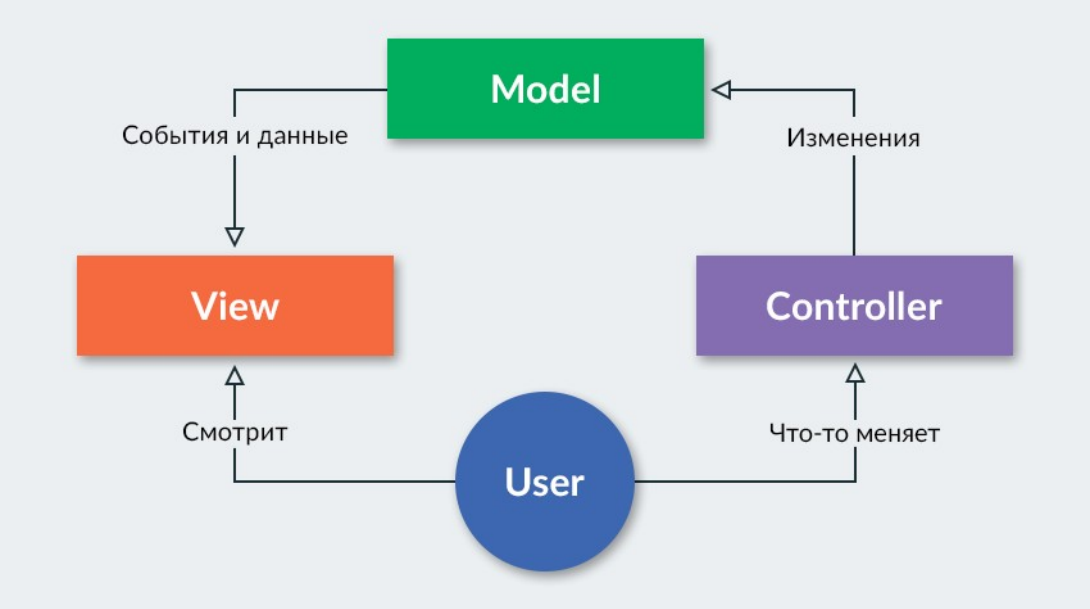
5.7. Архитектура MVC

Название архитектуры MVC упоминалось в контексте темы «Организация графического интерфейса пользователя». В этом разделе MVC рассматривается как архитектура приложений.

При разработке систем с пользовательским интерфейсом, следуя архитектуре MVC, нужно разделять систему на три составные части:

- ***Модель (Model)*** — содержит всю ***бизнес-логику*** приложения, т.е. классы объектов реального мира (называемые также ***бизнес-классами***), имеющих непосредственное отношение к области применения программного продукта; бизнес-классы часто реализуются по правилам JavaBeans. Модель — самая независимая часть системы, которая либо ничего не «знает» о двух других частях (представлении и контроллере), либо взаимодействует с объектами представления с помощью механизма событий.

- ***Представление (View)*** — содержит классы, которые отвечают за ввод и отображение данных в удобном для восприятия пользователя формате. Представление не должно вносить изменения в модель.
- ***Контроллер (Controller)*** — содержит классы, которые отвечают за обработку действий пользователя, служат посредниками между интерфейсом пользователя и бизнес-логикой приложения.



При создании сложной системы имеет смысл сначала выполнить по отдельности проектирование классов модели и представления. Затем создаются классы контроллера с использованием шаблона «Фасад» или «Наблюдатель».

Рассмотрим пример. Приложение «DictumFactum.Жеребьёвка» предназначено для вытягивания жребия электронным способом: выбор производится с помощью генератора случайных чисел. Чтобы вытянуть жребий, пользователь должен создать тему. В текущей версии приложения поддерживается два вида тем: одиночный выбор и множественный выбор. Пользователь должен ввести название темы, краткое описание (по желанию), варианты для выбора. В случае, когда жребий тянут несколько участников, используя одно устройство, и до принятия решения они не должны видеть варианты друг друга, допускается использование настройки «Скрытые варианты».

Приложение разрабатывается для «десктопа» и для мобильных устройств на базе Android. Обе реализации приложения используют одинаковые модель и контроллер, отличаются только представлением: пользовательский интерфейс на Android выглядит и реализуется сильно иначе, чем на «десктопе».

Отличие заключается еще и в том, что под Android работает специальная виртуальная машина, называемая Dalvik, которая принимает специальный формат байт-кодов в виде dex-файлов. При переходе с «декстопа» на Android

требуется либо преобразование class-файлов в dex-файлы, либо перекомпиляция исходных кодов.

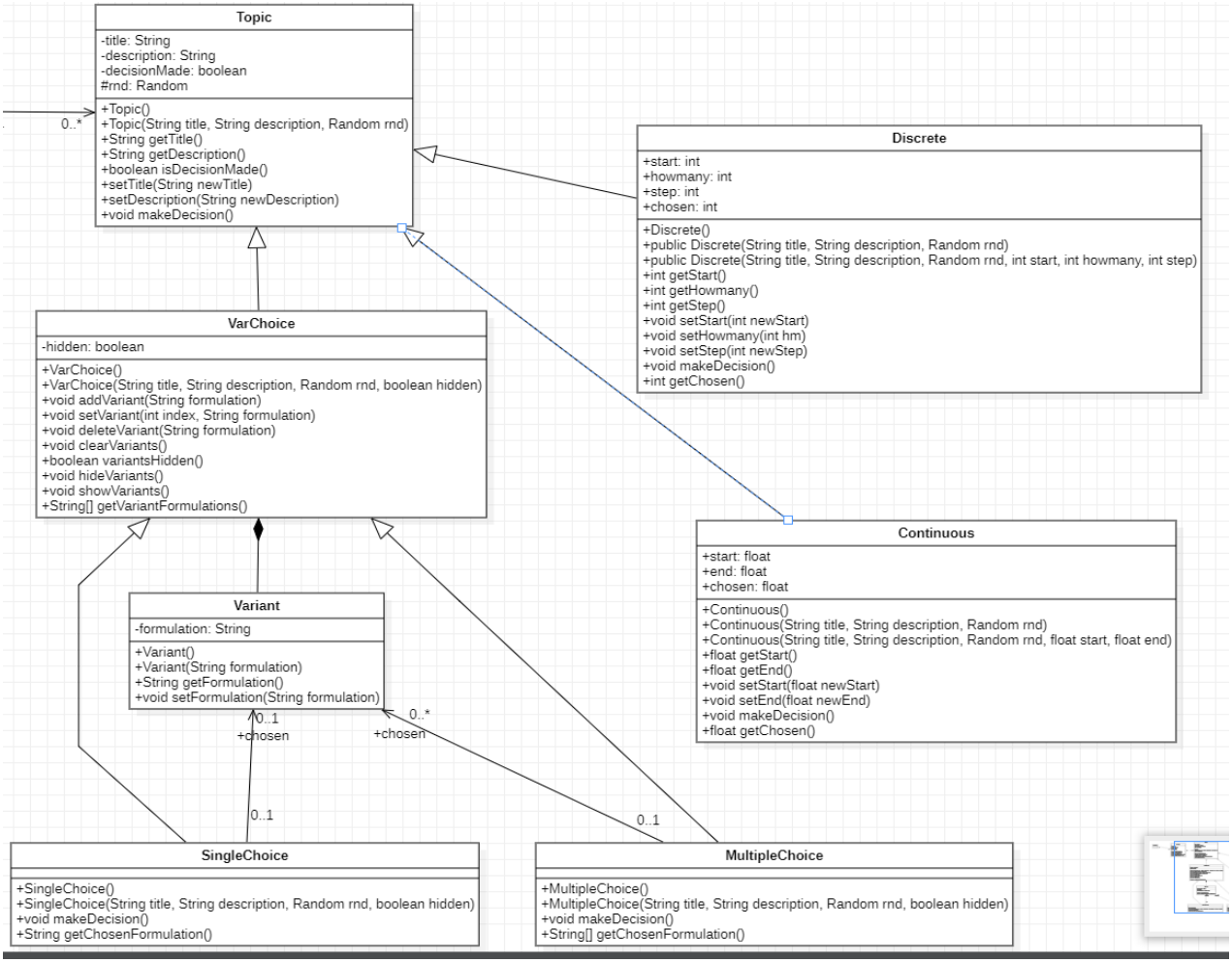


Диаграмма классов. Часть 1

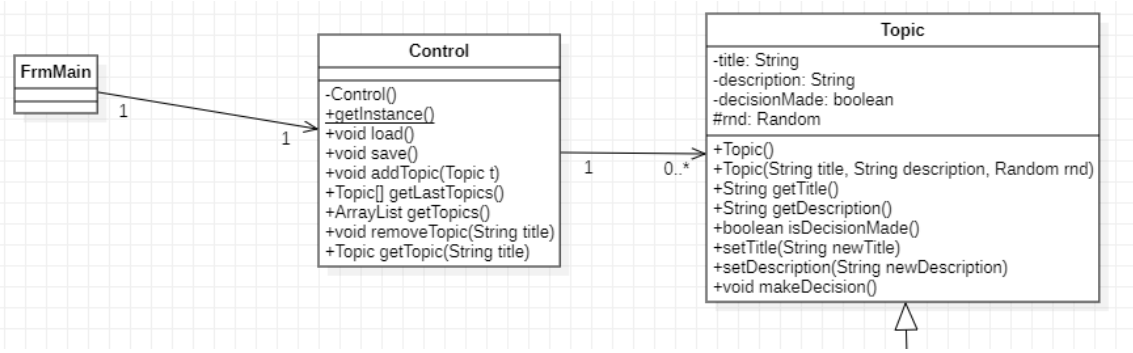


Диаграмма классов. Часть 2

Модель содержит следующие классы:

- *Topic* – абстрактный класс, общий предок всех тем. Содержит название темы, краткое описание, признак того, принято ли по теме решение, ссылку на генератор случайных чисел. Наряду с «геттерами» и «сеттерами» определяет метод *makeDecision()*, который следует переопределять в подклассах.
- *VarChoice* – абстрактный класс, наследник *Topic*. Находится в отношении композиции с классом *Variant* (вариант выбора). С точки зрения программной реализации *Variant* вложен в *VarChoice*. Также *VarChoice* содержит коллекцию вариантов, поле *hidden* – настройку отображения вариантов (*true* – скрытые, *false* – открытые), общедоступные методы работы с вариантами (добавление, удаление, поиск и др.), скрытия и раскрытия вариантов. Метод *makeDecision()* в нем не переопределен.
- *Variant* – класс, который описывает вариант выбора для темы *VarChoice*. В текущей реализации содержит только два закрытых поля: формулировку и ссылку на тему, к которой этот вариант относится. Не исключено, что в новой версии программы класс Variant будет дополнен другими свойствами и станет полноценным.
- *SingleChoice* – наследник *VarChoice*. Описывает тему с выбором единственного варианта. Содержит переопределенный метод *makeDecision()*, который вызывает метод *makeDecision()* родителя, но перед этим устанавливает один из вариантов в качестве выбранного. Метод *getChosenFormulation()* возвращает формулировку выбранного варианта.
- *MultipleChoice* – наследник *VarChoice*. Описывает тему с множественным выбором вариантов. Также содержит собственную реализацию метода *makeDecision()* и метод *getChosenFormulation()*, который возвращает массив формулировок выбранных вариантов.

Для будущих версий программы в модель включены еще два наследника класса *Topic*: *Discrete* и *Continuous*. Первый выбирает целочисленное значение из заданного диапазона (дискретная случайная величина), второй — вещественное (непрерывная случайная величина).

Контроллер в программе представлен классом Control. Он спроектирован по шаблону «Одиночка». Ссылка на единственный экземпляр этого класса хранится в его статическом поле *INSTANCE*. Контроллер содержит коллекцию тем, когда-либо созданных в приложении, и методы их сохранения в файле, загрузки из файла, добавления, удаления и поиска темы, получения перечня имеющихся тем.

Представление в «десктоп»-версии реализуется при помощи класса *FrmMain* – наследника *JFrame*, а в мобильной версии — с помощью множества *Activity*-компонентов.

5.8. Компонентное программирование. JavaBeans[24]

JavaBeans – это стандарт, разработанный компанией Sun Microsystems, для написания повторно используемых компонентов программы.

24 https://ru.wikipedia.org/wiki/JavaBeans

Словосочетание «Java bean» (bean – фасоль, кофейное зерно) применяется по отношению к классу, написанному по определенным правилам. Таким классом можно эффективно управлять, используя графические конструкторы и средства IDE, а его объекты можно легко объединять в один составной объект и передавать.

Компоненты JavaBeans могут принимать различные формы, но наиболее широко они применяются в элементах графического пользовательского интерфейса. Одна из целей создания JavaBeans — взаимодействие с похожими компонентными структурами. Например, Windows-программа, при наличии соответствующего моста или объекта-обёртки, может использовать компонент JavaBeans так, будто бы он является компонентом COM или ActiveX.

Правила описания bean-класса следующие:

1. Класс должен иметь конструктор без параметров, с модификатором доступа *public*. Такой конструктор позволяет инструментам создать объект без дополнительных сложностей с параметрами.

2. Свойства класса должны быть доступны через *getXxx()*, *setXxx()* и другие методы (так называемые методы доступа), которые должны соответствовать стандартному соглашению об именах. Это легко позволяет инструментам автоматически определять и обновлять содержание bean-объектов. Многие инструменты даже имеют специализированные редакторы для различных типов свойств.

3. Класс должен быть сериализуемым (реализовывать интерфейс *Serializable*). Это даёт возможность надежно сохранятьи восстанавливать состояние bean-объекта независимым от платформы и виртуальной машины способом.

4. Класс должен иметь переопределенные методы *equals()*, *hashCode()* и *toString()*.

Пример:

```
// PersonBean.java

public class PersonBean implements java.io.Serializable {
   private String name;
   private boolean deceased;

   public PersonBean() {
   }

   // Методы «геттеры (get) и сеттеры (set)
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
```

```
    public boolean getDeceased() {
        return deceased;
    }
    public void setDeceased(boolean deceased) {
        this.deceased = deceased;
    }

    //Переопределенные методы equals() и hashCode()
    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (o == null || getClass() != o.getClass()) {
            return false;
        }

        PersonBean that = (PersonBean) o;

        if (deceased != that.deceased) {
            return false;
        }
        return !(name != null ? !name.equals(that.name) : that.name != null);

    }

    @Override
    public int hashCode() {
        int result = name != null ? name.hashCode() : 0;
        result = 31 * result + (deceased ? 1 : 0);
        return result;
    }

    //Переопределенный метод toString()
    @Override
    public String toString() {
        return "PersonBean{" +
                "name='" + name + '\'' +
                ", deceased=" + deceased +
                '}';
    }
}

// TestPersonBean.java
```

```
public class TestPersonBean {
   public static void main(String[] args) {

      PersonBean person = new PersonBean();
      person.setName("Bob");
      person.setDeceased(true);

      // Результат: "Bob [deceased]"
      System.out.print(person.getName());
      System.out.println(person.getDeceased() ? " [deceased]" : " [alive]");
   }
}
```