Что такое паттерн проектирования.

Термин "паттерн проектирования" пришел из архитектуры и ввел его в обращение Кристофер Александр, архитектор, при решении задач, возникающих при проектировании зданий и городов [15]. Но его слова «любой паттерн описывает задачу, которая снова и снова возникает в нашей работе, а также принцип ее решения, причем таким образом, что это решение можно потом использовать миллион раз, ничего не изобретая заново» верны и в отношении паттернов проектирования в ПО.

Что же такое паттерны (GoF) – а именно так мы будем обозначать их отделяя от появившихся позже библиотек шаблонов C++, JAVA, PHP и т.п.

В общем случае паттерн состоит из четырех основных элементов [14]:

1. Имя. Сославшись на него, мы можем сразу описать проблему проектирования; ее решения и их последствия. Присваивание паттернам имен позволяет проектировать на более высоком уровне абстракции. С помощью словаря паттернов можно вести обсуждение с коллегами, упоминать паттерны в документации, в тонкостях представлять дизайн системы. *Название паттерна должно четко отражать его назначение.*

2. Задача. Описание того, когда следует применять паттерн. Необходимо сформулировать задачу и ее контекст. Может описываться конкретная проблема проектирования или перечень условий, при выполнении которых имеет смысл применять данный паттерн.

3. Решение. Описание элементов, отношений между ними, функций каждого элемента. Конкретный дизайн или реализация не имеются в виду, поскольку паттерн применим в самых разных ситуациях. Дается абстрактное описание задачи проектирования и того, как она может быть решена с помощью некоего весьма обобщенного сочетания классов и объектов.

4. Результаты - это следствия применения паттерна и разного рода компромиссы. Хотя при описании проектных решений о последствиях часто не упоминают, знать о них необходимо, чтобы можно было выбрать между различными вариантами и оценить преимущества и недостатки данного паттерна.

Таким образом, под паттернами проектирования понимается описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте.

Паттерн проектирования именует, абстрагирует и идентифицирует ключевые аспекты структуры общего решения, которые и позволяют применить его для повторно используемого дизайна. Он вычленяет участвующие классы и экземпляры, их роль и отношения, реализуемые методы. При описании каждого паттерна внимание акцентируется на конкретной задаче проектирования. Анализируется, когда следует применять паттерн, можно ли его использовать с учетом других проектных ограничений,

каковы будут последствия применения метода.

Паттерны должны рассматриваться на определенном уровне абстракции. Под паттернами проектирования понимается *описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте.* Паттерн проектирования именует, абстрагирует и идентифицирует ключевые аспекты структуры общего решения, которые и позволяют применить его для создания повторно используемого дизайна или модифицировать алгоритм решения задачи без полной переделки структуры и кода программы. При описании каждого паттерна внимание акцентируется на конкретной задаче объектно-ориентированного проектирования. Анализируется, когда следует применять паттерн, можно ли его использовать с учетом проектных ограничений, каковы будут последствия применения метода.

Каталог паттернов проектирования (GoF)

Каталог паттернов проектирования (GoF) содержит 23 паттерна. Паттерны проектирования различаются степенью детализации и уровнем абстракции и их можно разделить на две группы (табл. 1). Первая – уровень класса, вторая – объекта.

Каждая группа делится на порождающие паттерны, структурные паттерны и паттерны поведения. Первые связаны с процессом создания объектов. Вторые имеют отношение к композиции объектов и классов. Паттерны поведения характеризуют то, как классы или объекты взаимодействуют между собой.

Таблица 1.

Цель / Уровень	**Порождающие паттерны**	**Структурные паттерны**	**Паттерны поведения**
Класс	(Фабричный метод) Factory Method	Адаптер (Adapter)	Интерпретатор (Interpreter) Шаблонный метод (Template Method)
Объект	Абстрактная фабрика (Abstract Factory) Строитель (Builder) Прототип (Prototype) Одиночка (Singleton)	Адаптер (Adapter) Декоратор (Decorator) Заместитель (Proxy) Компоновщик (Composite) Мост (Bridge) Приспособленец (Flyweight) Фасад (Facade)	Итератор (Iterator) Команда (Command) Наблюдатель (Observer) Хранитель (Memento) Стратегия (Strategy) Состояние (State) Посредник (Mediator) Посетитель (Visitor) Цепочка обязанностей (Chain of Responsibility)

Паттерны уровня **классов** описывают отношения между классами и их подклассами. Такие отношения выражаются с помощью наследования и поэтому они статичны, то есть эти отношения зафиксированы на этапе компиляции.

Паттерны уровня **объектов** описывают отношения между объектами, которые могут изменяться во время выполнения программы, и потому более динамичны. Почти все

паттерны в какой-то мере используют наследование. Поэтому к категории “паттерны классов” отнесены только те, что сфокусированы лишь на отношениях между классами. Обратите внимание: большинство паттернов действуют на уровне объектов.

Порождающие паттерны классов частично делегируют ответственность за создание объектов своим подклассам, тогда как порождающие паттерны объектов передают ответственность другому объекту. **Структурные** паттерны классов используют наследование для составления классов, в то время как структурные паттерны объектов описывают способы сборки объектов из частей. **Поведенческие** паттерны классов используют наследование для описания алгоритмов и потока управления, а поведенческие паттерны объектов описывают, как объекты, принадлежащие некоторой группе, совместно функционируют и выполняют задачу, которая ни одному отдельному объекту не под силу.

Л а б о р а т о р н а я р а б о т а № 1
«Реализация одного из порождающих паттерны проектирования»

Цель работы: Научиться применять порождающие паттерны проектирования.

Продолжительность работы - 4 часа.

Содержание

Порождающие паттерны.
Порождающие паттерны проектирования абстрагируют процесс инстанцирования объектов. Они позволяют сделать код независимым от способа создания, композиции и представления используемых в его работе объектов.

Список порождающих паттернов (GoF):
Фабричный метод (Factory method);
Абстрактная фабрика (Abstract Factory);
Строитель (Builder);
Прототип (Prototype);
Одиночка (Singleton).

Паттерн Абстрактная фабрика (Abstract Factory)

Название и классификация паттерна

Абстрактная фабрика - паттерн, порождающий объекты.

Назначение

Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.

Применимость

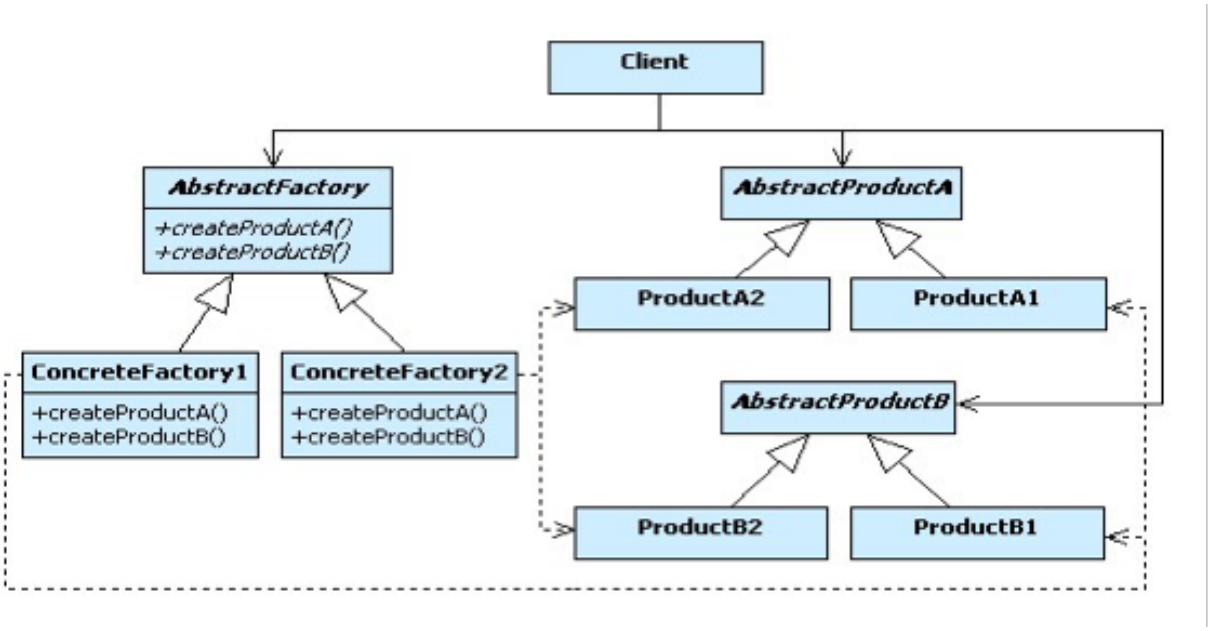
Использование паттерна Abstract Factory (абстрактная фабрика) целесообразно если:

- система не должна зависеть от того, как создаются, компонуются и представляются входящие в нее объекты;

-входящие в семейство взаимосвязанные объекты должны использоваться вместе и вам необходимо обеспечить выполнение этого ограничения;

- система должна конфигурироваться одним из семейств составляющих ее объектов, а вы хотите предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию.

Структура



Участники

- AbstractFactory - абстрактная фабрика: объявляет интерфейс для операций, создающих абстрактные объекты-продукты;

- ConcreteFactory (ConcreteFactory1, ConcreteFactory2) - конкретная фабрика: реализует операции, создающие конкретные объекты-продукты;

- **AbstractProduct** (AbstractProductA, AbstractProductB) - абстрактный продукт: объявляет интерфейс для типа объекта-продукта;

- ConcreteProduct (ProductA, ProductB) - конкретный продукт: определяет объект-продукт, создаваемый соответствующей конкретной - реализует интерфейс Abstract Product;

- **Client** - клиент: пользуется исключительно интерфейсами, которые объявлены в классах AbstractFactory и AbstractProduct.

Отношения

- Обычно во время выполнения создается единственный экземпляр класса ConcreteFactory. Эта конкретная фабрика создает объекты-продукты, имеющие вполне определенную реализацию. Для создания других видов объектов клиент должен воспользоваться другой конкретной фабрикой;

- AbstractFactory передоверяет создание объектов-продуктов своему подклассу ConcreteFactory.

Результаты

Паттерн абстрактная фабрика обладает следующими плюсами и минусами:

- *изолирует конкретные классы.* Помогает контролировать классы объектов, создаваемых приложением. Поскольку фабрика инкапсулирует ответственность за создание классов и сам процесс их создания, то она изолирует клиента от деталей реализации классов. Клиенты манипулируют экземплярами через их абстрактные интерфейсы. Имена изготавливаемых классов известны только конкретной фабрике, в коде клиента они не упоминаются;

- *упрощает замену семейств продуктов.* Класс конкретной фабрики появляется в приложении только один раз: при инстанцировании. Это облегчает замену используемой приложением конкретной фабрики.

- *гарантирует сочетаемость продуктов.* Если продукты некоторого семейства спроектированы для совместного использования, то важно, чтобы приложение в каждый момент времени работало только с продуктами единственного семейства. Класс AbstractFactory позволяет легко соблюсти это ограничение;

- *поддержать новый вид продуктов трудно.* Расширение абстрактной фабрики для изготовления новых видов продуктов - непростая задача. Интерфейс AbstractFactory фиксирует набор продуктов, которые можно создать. Для поддержки новых продуктов необходимо расширить интерфейс фабрики, то есть изменить класс AbstractFactory и все его подклассы.

Пример кода для паттерна Abstract Factory

Приведем реализацию паттерна Abstract Factory для военной стратегии "Пунические войны". При этом предполагается, что число и типы создаваемых в игре боевых единиц идентичны для обеих армий. Каждая армия имеет в своем составе пехотинцев (Infantryman), лучников (Archer) и кавалерию (Horseman).

Структура паттерна для данного случая представлена ниже на рис. 1.

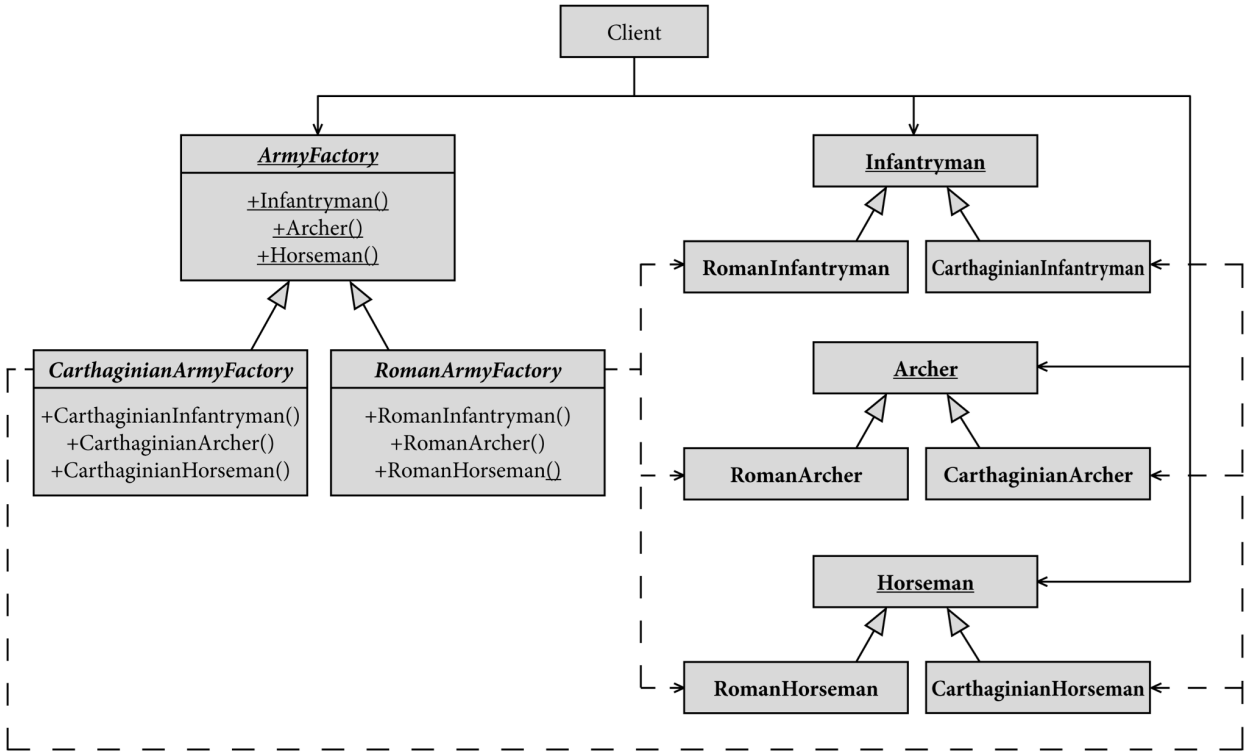


Рисунок 1. UML-диаграмма классов для военной стратегии "Пунические войны"

```
#include <iostream>
#include <vector>

// Абстрактные базовые классы всех возможных видов воинов
class Infantryman
{
  public:
    virtual void info() = 0;
    virtual ~Infantryman() {}
};

class Archer
{
  public:
    virtual void info() = 0;
    virtual ~Archer() {}
```

```
};

class Horseman
{
  public:
    virtual void info() = 0;
    virtual ~Horseman() {}
};


// Классы всех видов воинов Римской армии
class RomanInfantryman: public Infantryman
{
  public:
    void info() {
      cout << "RomanInfantryman" << endl;
    }
};

class RomanArcher: public Archer
{
  public:
    void info() {
      cout << "RomanArcher" << endl;
    }
};

class RomanHorseman: public Horseman
{
  public:
    void info() {
      cout << "RomanHorseman" << endl;
    }
};


// Классы всех видов воинов армии Карфагена
class CarthaginianInfantryman: public Infantryman
{
  public:
    void info() {
      cout << "CarthaginianInfantryman" << endl;
    }
};

class CarthaginianArcher: public Archer
{
  public:
    void info() {
      cout << "CarthaginianArcher" << endl;
```

```
    }
};

class CarthaginianHorseman: public Horseman
{
  public:
    void info() {
      cout << "CarthaginianHorseman" << endl;
    }
};


// Абстрактная фабрика для производства воинов
class ArmyFactory
{
  public:
    virtual Infantryman* createInfantryman() = 0;
    virtual Archer* createArcher() = 0;
    virtual Horseman* createHorseman() = 0;
    virtual ~ArmyFactory() {}
};


// Фабрика для создания воинов Римской армии
class RomanArmyFactory: public ArmyFactory
{
  public:
    Infantryman* createInfantryman() {
      return new RomanInfantryman;
    }
    Archer* createArcher() {
      return new RomanArcher;
    }
    Horseman* createHorseman() {
      return new RomanHorseman;
    }
};


// Фабрика для создания воинов армии Карфагена
class CarthaginianArmyFactory: public ArmyFactory
{
  public:
    Infantryman* createInfantryman() {
      return new CarthaginianInfantryman;
    }
    Archer* createArcher() {
      return new CarthaginianArcher;
    }
    Horseman* createHorseman() {
```

```
    return new CarthaginianHorseman;
  }
};


// Класс, содержащий всех воинов той или иной армии
class Army
{
 public:
  ~Army() {
    int i;
    for(i=0; i<vi.size(); ++i)  delete vi[i];
    for(i=0; i<va.size(); ++i)  delete va[i];
    for(i=0; i<vh.size(); ++i)  delete vh[i];
  }
  void info() {
    int i;
    for(i=0; i<vi.size(); ++i)  vi[i]->info();
    for(i=0; i<va.size(); ++i)  va[i]->info();
    for(i=0; i<vh.size(); ++i)  vh[i]->info();
  }
  vector<Infantryman*> vi;
  vector<Archer*> va;
  vector<Horseman*> vh;
};


// Здесь создается армия той или иной стороны
class Game
{
 public:
  Army* createArmy( ArmyFactory& factory ) {
    Army* p = new Army;
    p->vi.push_back( factory.createInfantryman());
    p->va.push_back( factory.createArcher());
    p->vh.push_back( factory.createHorseman());
    return p;
  }
};


int main()
{
  Game game;
  RomanArmyFactory ra_factory;
  CarthaginianArmyFactory ca_factory;

  Army * ra = game.createArmy( ra_factory);
  Army * ca = game.createArmy( ca_factory);
  cout << "Roman army:" << endl;
```

```
ra->info();
cout << "\nCarthaginian army:" << endl;
ca->info();
// ...
```

Вывод программы будет следующим:

Roman army:

RomanInfantryman

RomanArcher

RomanHorseman

Carthaginian army:

CarthaginianInfantryman

CarthaginianArcher

CarthaginianHorseman

Паттерн Одиночка (Singleton).

Название и классификация паттерна

Одиночка - паттерн, порождающий объекты

Назначение

Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

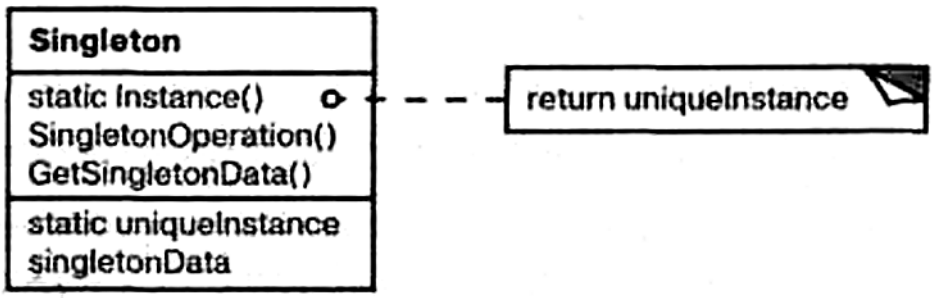
Применимость

Используйте паттерн одиночка, когда:

- должен быть ровно один экземпляр некоторого класса, легко доступный всем клиентам;

- единственный экземпляр должен расширяться путем порождения подклассов, и клиентам нужно иметь возможность работать с расширенным экземпляром без модификации своего кода.

Структура



Участники

Singleton - одиночка:

- определяет операцию Instance, которая позволяет клиентам получать доступ к единственному экземпляру. Instance - это операция класса, и статическая функция-член в C++;

- может нести ответственность за создание собственного уникального экземпляра.

Отношения

Клиенты получают доступ к экземпляру класса Singleton только через его операцию Instance.

Результаты

У паттерна одиночка есть определенные достоинства:

- *контролируемый доступ к единственному экземпляру.* Поскольку класс Singleton инкапсулирует свой единственный экземпляр, он полностью контролирует то, как и когда клиенты получают доступ к нему;

- *уменьшение числа имен.* Паттерн одиночка - шаг вперед по сравнению с глобальными переменными. Он позволяет избежать засорения пространства имен глобальными переменными, в которых хранятся уникальные экземпляры;

- *допускает уточнение операций и представления.* От класса Singleton можно порождать подклассы, а приложение легко сконфигурировать экземпляром расширенного класса. Можно конкретизировать приложение экземпляром того класса, который необходим во время выполнения;

- *допускает переменное число экземпляров.* Паттерн позволяет вам легко изменить свое решение и разрешить появление более одного экземпляра класса Singleton. Вы можете

применять один и тот же подход для управления числом экземпляров, используемых в приложении. Изменить нужно будет лишь операцию, дающую доступ к экземпляру класса Singleton;

Классическая реализация Singleton

Рассмотрим наиболее часто встречающуюся реализацию паттерна Singleton.

```
// Singleton.h
class Singleton
{
  private:
    static Singleton * p_instance;
    // Конструкторы и оператор присваивания недоступны клиентам
    Singleton() {}
    Singleton( const Singleton& );
    Singleton& operator=( Singleton& );
  public:
    static Singleton * getInstance() {
        if(!p_instance)
            p_instance = new Singleton();
        return p_instance;
    }
};

// Singleton.cpp
#include "Singleton.h"

Singleton* Singleton::p_instance = 0;
```

Клиенты запрашивают единственный объект класса через статическую функцию-член *getInstance()*, которая при первом запросе динамически выделяет память под этот объект и затем возвращает указатель на этот участок памяти. Впоследствии клиенты должны сами позаботиться об освобождении памяти при помощи оператора *delete*.

Последняя особенность является серьезным недостатком классической реализации шаблона *Singleton*. Так как класс сам контролирует создание единственного объекта, было бы логичным возложить на него ответственность и за разрушение объекта. Этот недостаток отсутствует в реализации *Singleton*, впервые предложенной Скоттом Мэйерсом.

Singleton Мэйерса

```
// Singleton.h
class Singleton
{
private:
```

```
   Singleton() {}
   Singleton( const Singleton&);
   Singleton& operator=( Singleton& );
public:
   static Singleton& getInstance() {
      static Singleton  instance;
      return instance;
   }
};
```

Внутри getInstance() используется статический экземпляр нужного класса. Стандарт языка программирования C++ гарантирует автоматическое уничтожение статических объектов при завершении программы. Досрочного уничтожения и не требуется, так как объекты Singleton обычно являются долгоживущими объектами. Статическая функция-член getInstance() возвращает не указатель, а ссылку на этот объект, тем самым, затрудняя возможность ошибочного освобождения памяти клиентами.

Приведенная реализация паттерна Singleton использует так называемую отложенную инициализацию (lazy initialization) объекта, когда объект класса инициализируется не при старте программы, а при первом вызове getInstance(). В данном случае это обеспечивается тем, что статическая переменная instance объявлена внутри функции - члена класса getInstance(), а не как статический член данных этого класса. Отложенную инициализацию, в первую очередь, имеет смысл использовать в тех случаях, когда инициализация объекта представляет собой дорогостоящую операцию и не всегда используется.

К сожалению, у реализации Мэйерса есть недостатки: сложности создания объектов производных классов и невозможность безопасного доступа нескольких клиентов к единственному объекту в многопоточной среде.

Достоинства паттерна Singleton

1. Класс сам контролирует процесс создания единственного экземпляра.
2. Паттерн легко адаптировать для создания нужного числа экземпляров.
3. Возможность создания объектов классов, производных от Singleton.

Недостатки паттерна Singleton

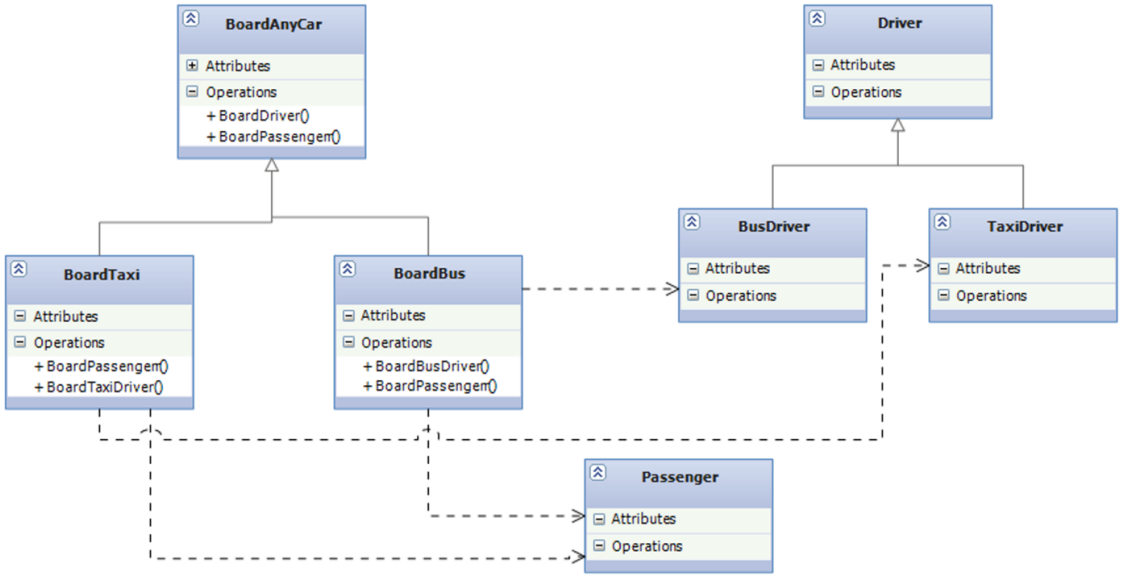
В случае использования нескольких взаимозависимых одиночек их реализация может резко усложниться.

Задание

1. С использованием одного из языков программирования из множества (C++, C#) реализовать паттерн порождающего проектирования Одиночка (singleton).
2. С помощью шаблона Абстрактная фабрика решить следующую задачу.

Обеспечить контроль загрузки и готовности к отправлению автобусов и такси. Водитель такси и автобуса имеют права разной категории. Без водителя машина не поедет. Два водителя в одну машину сесть не могут. Без пассажиров машины не поедут. Есть лимит загрузки машин. Для автобуса 30 чел. Для такси -4 чел.

Рекомендуется для водителя применить паттерн синглтон.



Л а б о р а т о р н а я р а б о т а № 2
«Реализация одного из порождающих паттерны проектирования»

Цель работы: Применение паттерна проектирования Builder (строитель)
Продолжительность работы - 4 часа.

Содержание

Порождающие паттерны.
Порождающие паттерны проектирования абстрагируют процесс инстанцирования объектов. Они позволяют сделать код независимым от способа создания, композиции и представления используемых в его работе объектов.

Паттерн Builder (строитель)
Назначение

Отделяет конструирование сложного объекта от его представления, так что в результате одного и того же процесса конструирования могут получаться разные представления.

Класс каждого конвертора принимает механизм создания и сборки сложного объекта и скрывает его за абстрактным интерфейсом. Конвертор отделен от загрузчика, который отвечает за синтаксический разбор RTF-документа.

В паттерне строитель абстрагированы все эти отношения. В нем любой класс конвертора называется *строителем,* а загрузчик – *распорядителем.*

Применимость

- алгоритм создания сложного объекта не должен зависеть от того, из каких частей состоит объект и как они стыкуются между собой;

- процесс конструирования должен обеспечивать различные представления конструируемого объекта.

Структура

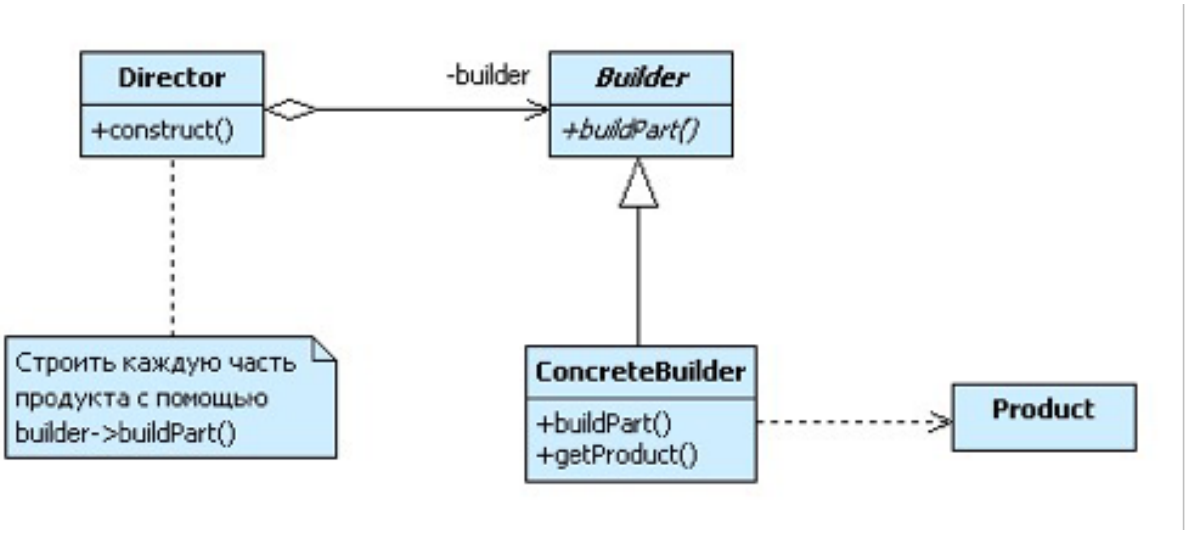


Рисунок 2. UML-диаграмма классов паттерна Builder

Участники

Director - распорядитель: конструирует объект, пользуясь интерфейсом Builder;

Builder - строитель: задает абстрактный интерфейс для создания частей объекта Product;

ConcreteBuilder - конкретный строитель:

- конструирует и собирает вместе части продукта посредством реализации интерфейса Builder;

- определяет создаваемое представление и следит за ним;

- предоставляет интерфейс для доступа к продукту;

Product - продукт:

- представляет сложный конструируемый объект. ConcreteBuilder строит внутреннее представление продукта и определяет процесс его сборки;

- включает классы, которые определяют составные части, в том числе интерфейсы для сборки конечного результата из частей.

Отношения

- клиент создает объект-распорядитель Director и конфигурирует его нужным объектом-строителем Builder;

- распорядитель уведомляет строителя о том, что нужно построить очередную часть продукта;

- строитель обрабатывает запросы распорядителя и добавляет новые части к продукту;

- клиент забирает продукт у строителя.

Следующая диаграмма взаимодействий иллюстрирует взаимоотношения строителя и распорядителя с клиентом.

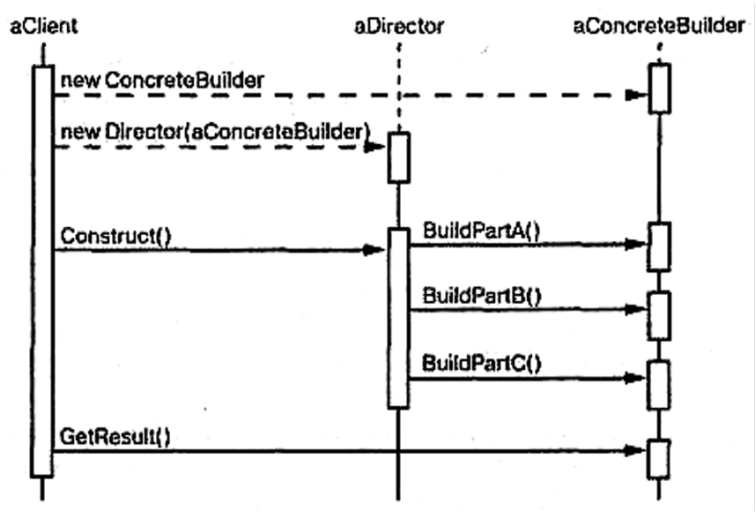


Рисунок 3. UML-диаграмма последовательностей паттерна Builder

Результаты

Плюсы и минусы паттерна строитель и его применения:

- *позволяет изменять внутреннее представление продукта.* Объект Builder предоставляет распорядителю абстрактный интерфейс для конструирования продукта, за которым он может скрыть представление и внутреннюю структуру продукта, а также процесс его сборки. Поскольку продукт конструируется через абстрактный интерфейс, то для изменения внутреннего представления достаточно всего лишь определить новый вид строителя;

- *изолирует код, реализующий конструирование и представление.* Паттерн строитель улучшает модульность, инкапсулируя способ конструирования и представления сложного объекта. Клиентам ничего не надо знать о классах, определяющих внутреннюю структуру продукта, они отсутствуют в интерфейсе строителя. Каждый конкретный строитель ConcreteBuilder содержит весь код, необходимый для создания и сборки конкретного вида продукта. Код пишется только один раз, после чего разные распорядители могут использовать его повторно для построения вариантов продукта из одних и тех же частей.

- *дает более тонкий контроль над процессом конструирования.* В отличие от порождающих паттернов, которые сразу конструируют весь объект целиком, строитель делает это шаг за шагом под управлением распорядителя. И лишь когда продукт завершен, распорядитель забирает его у строителя. Поэтому интерфейс строителя в большей степени отражает процесс конструирования продукта, нежели другие порождающие паттерны. Это позволяет обеспечить более тонкий контроль над процессом конструирования, а значит, и над внутренней структурой готового продукта.

Реализация

Обычно существует абстрактный класс Builder, в котором определены операции для каждого компонента, который распорядитель может ≪попросить≫ создать. По умолчанию эти операции ничего не делают. Но в классе конкретного строителя ConcreteBuilder они замещены для тех компонентов, в создании которых он принимает участие.

Вот еще некоторые достойные внимания вопросы реализации:

- *интерфейс сборки и конструирования.* Интерфейс класса Builder должен быть достаточно общим, чтобы обеспечить конструирование при любом виде конкретного строителя. Ключевой вопрос проектирования связан с выбором модели процесса конструирования и сборки. Обычно бывает достаточно модели, в которой результаты выполнения запросов на конструирование просто добавляются к продукту. Но иногда может потребоваться доступ к частям сконструированного к данному моменту продукта. Например, деревья синтаксического разбора строятся снизу вверх.

- *почему нет абстрактного класса для продуктов.* В типичном случае продукты, изготавливаемые различными строителями, имеют настолько разные представления, что изобретение для них общего родительского класса ничего не дает. Поскольку клиент обычно конфигурирует распорядителя подходящим конкретным строителем, то, надо полагать, ему известно, какой именно подкласс класса Builder используется и как нужно обращаться с произведенными продуктами;

- *пустые методы класса Builder по умолчанию.* В C++ методы строителя намеренно не объявлены чисто виртуальными функциями-членами. Вместо этого они определены как пустые функции, что позволяет подклассу замещать только те операции, в которых он заинтересован.

Пример кода

Приведем реализацию паттерна Builder на примере построения армий для военной стратегии "Пунические войны". Такие рода войск как пехота, лучники и конница для обеих армий идентичны. С целью демонстрации возможностей паттерна Builder введем новые виды боевых единиц:

- Катапульты для армии Рима.
- Боевые слоны для армии Карфагена.

```
#include <iostream>
#include <vector>

// Классы всех возможных родов войск
class Infantryman
{
  public:
    void info() {
      cout << "Infantryman" << endl;
    }
```

```
};

class Archer
{
  public:
    void info() {
      cout << "Archer" << endl;
    }
};

class Horseman
{
  public:
    void info() {
    cout << "Horseman" << endl;
    }
};

class Catapult
{
  public:
    void info() {
      cout << "Catapult" << endl;
    }
};

class Elephant
{
  public:
    void info() {
      cout << "Elephant" << endl;
    }
};


// Класс "Армия", содержащий все типы боевых единиц
class Army
{
  public:
    vector<Infantryman> vi;
    vector<Archer>      va;
    vector<Horseman>    vh;
    vector<Catapult>    vc;
    vector<Elephant>    ve;
    void info() {
      int i;
      for(i=0; i<vi.size(); ++i)  vi[i].info();
      for(i=0; i<va.size(); ++i)  va[i].info();
      for(i=0; i<vh.size(); ++i)  vh[i].info();
      for(i=0; i<vc.size(); ++i)  vc[i].info();
```

```
    for(i=0; i<ve.size(); ++i)  ve[i].info();
  }
};


// Базовый класс ArmyBuilder объявляет интерфейс для поэтапного
// построения армии и предусматривает его реализацию по умолчанию

class ArmyBuilder
{
 protected:
  Army* p;
 public:
  ArmyBuilder(): p(0) {}
  virtual ~ArmyBuilder() {}
  virtual void createArmy() {}
  virtual void buildInfantryman() {}
  virtual void buildArcher() {}
  virtual void buildHorseman() {}
  virtual void buildCatapult() {}
  virtual void buildElephant() {}
  virtual Army* getArmy() { return p; }
};


// Римская армия имеет все типы боевых единиц кроме боевых слонов
class RomanArmyBuilder: public ArmyBuilder
{
 public:
  void createArmy() { p = new Army; }
  void buildInfantryman() { p->vi.push_back( Infantryman()); }
  void buildArcher() { p->va.push_back( Archer()); }
  void buildHorseman() { p->vh.push_back( Horseman()); }
  void buildCatapult() { p->vc.push_back( Catapult()); }
};


// Армия Карфагена имеет все типы боевых единиц кроме катапульт
class CarthaginianArmyBuilder: public ArmyBuilder
{
 public:
  void createArmy() { p = new Army; }
  void buildInfantryman() { p->vi.push_back( Infantryman()); }
  void buildArcher() { p->va.push_back( Archer()); }
  void buildHorseman() { p->vh.push_back( Horseman()); }
  void buildElephant() { p->ve.push_back( Elephant()); }
};
```

```
// Класс-распорядитель, поэтапно создающий армию той или иной стороны.
// Именно здесь определен алгоритм построения армии.
class Director
{
  public:
    Army* createArmy( ArmyBuilder & builder )
    {
        builder.createArmy();
        builder.buildInfantryman();
        builder.buildArcher();
        builder.buildHorseman();
        builder.buildCatapult();
        builder.buildElephant();
        return( builder.getArmy());
    }
};


int main()
{
    Director dir;
    RomanArmyBuilder ra_builder;
    CarthaginianArmyBuilder ca_builder;

    Army * ra = dir.createArmy( ra_builder);
    Army * ca = dir.createArmy( ca_builder);
    cout << "Roman army:" << endl;
    ra->info();
    cout << "\nCarthaginian army:" << endl;
    ca->info();
    // ...

    return 0;
}
```

Вывод программы будет следующим:

```
Roman army:
Infantryman
Archer
Horseman
Catapult

Carthaginian army:
Infantryman
Archer
Horseman
Elephant
```

Очень часто базовый класс *строителя* (в коде выше это *ArmyBuilder*) не только объявляет интерфейс для построения частей продукта, но и определяет ничего не делающую реализацию по умолчанию. Тогда соответствующие подклассы (*RomanArmyBuilder, CarthaginianArmyBuilder)* переопределяют только те методы, которые участвуют в построении текущего объекта. Так класс *RomanArmyBuilder* не определяет метод *buildElephant()*, поэтому Римская армия не может иметь слонов. А в классе *CarthaginianArmyBuilder* не определен *buildCatapult()*, поэтому армия Карфагена не может иметь катапульты.

Достоинства паттерна Builder

- Возможность контролировать процесс создания сложного продукта.
- Возможность получения разных представлений некоторых данных.

Недостатки паттерна Builder

ConcreteBuilder и создаваемый им продукт жестко связаны между собой, поэтому при внесеннии изменений в класс продукта скорее всего придется соотвествующим образом изменять и класс *ConcreteBuilder*.

Родственные паттерны

Абстрактная фабрика похожа на *строитель* в том смысле, что может конструировать сложные объекты. Основное различие между ними в том, что *строитель* делает акцент на пошаговом конструировании объекта, а *абстрактная фабрика* - на создании семейств объектов (простых или сложных). *Строитель* возвращает продукт на последнем шаге, тогда как с точки зрения *абстрактной фабрики* продукт возвращается немедленно.

Паттерн *компоновщик* - это то, что часто создает строитель.

4. Задание

3. Разработать UML-диаграммы (диаграмму классов и диаграмму последовательности) и с помощью паттерна «Строитель» решить следующую задачу.

Обеспечить контроль загрузки и готовности к отправлению автобусов и такси. Водитель такси и автобуса имеют права разной категории. Без водителя машина не поедет. Два водителя в одну машину сесть не могут. Без пассажиров машины не поедут. Есть лимит загрузки машин. Для автобуса 30 чел. Для такси -4 чел.

Есть разница между пассажирами автобуса и такси.

Для автобуса: три категории пассажиров - взрослый, льготный, ребенок - разная стоимость билета.

Для такси: взрослый и ребенок. Необходимо детское кресло.

5. Требования к отчету

Отчет к лабораторной работе должен содержать текст работающей программы на языке программирования C++ или C# и результат выполнения программы.

6. Вопросы.

1. В чем заключается разница между паттерном проектирования «Абстрактная фабрика» и «Строитель».

2. Достоинства и недостатки паттернов проектирования «Абстрактная фабрика» и «Строитель».

Л а б о р а т о р н а я р а б о т а № 3
«**Реализация одного из структурных паттернов проектирования**»

Цель работы: Применение паттерна проектирования **Composite (компоновщик)**

Продолжительность работы - 4 часа.

Содержание

Структурные паттерны

В структурных паттернах рассматривается вопрос о том, как из классов и объектов образуются более крупные структуры. Структурные паттерны уровня *класса* используют наследование для составления композиций из *интерфейсов* и *реализаций*. Простой пример - использование множественного наследования для объединения нескольких классов в один. В результате получается класс, обладающий свойствами всех своих родителей.

Вместо композиции *интерфейсов* или *реализаций* структурные паттерны уровня *объекта* компонуют объекты для получения новой функциональности. Дополнительная гибкость в этом случае связана с возможностью изменить композицию объектов во время выполнения, что недопустимо для статической композиции классов.

При этом могут использоваться следующие механизмы:

- *Наследование*, когда базовый класс определяет интерфейс, а подклассы - реализацию. Структуры на основе наследования получаются статичными.
- *Композиция*, когда структуры строятся путем объединения объектов некоторых классов. Композиция позволяет получать структуры, которые можно изменять во время выполнения.

Кратко рассмотрим особенности структрурных паттернов.

Паттерн Adapter представляет собой программную обертку над уже существующими классами и предназначен для преобразования их интерфейсов к виду, пригодному для последующего использования в новом программном проекте.

Паттерн Bridge отделяет абстракцию от реализации так, что то и другое можно изменять независимо.

Паттерн Composite группирует схожие объекты в древовидные структуры. Рассматривает единообразно простые и сложные объекты.

Паттерн Decorator используется для расширения функциональности объектов. Являясь гибкой альтернативой порождению классов, паттерн Decorator динамически добавляет объекту новые обязанности.

Паттерн Facade предоставляет высокоуровневый унифицированный интерфейс к набору интерфейсов некоторой подсистемы, что облегчает ее использование.

Паттерн Flyweight использует разделение для эффективной поддержки множества объектов.

Паттерн Proxy замещает другой объект для контроля доступа к нему.

Паттерн компоновщик (Composite)

К*лассификация паттерна*

Компоновщик - паттерн, структурирующий объекты.

Назначение

Компонует объекты в древовидные структуры для представления иерархий часть-целое. Позволяет клиентам единообразно трактовать индивидуальные и составные объекты.

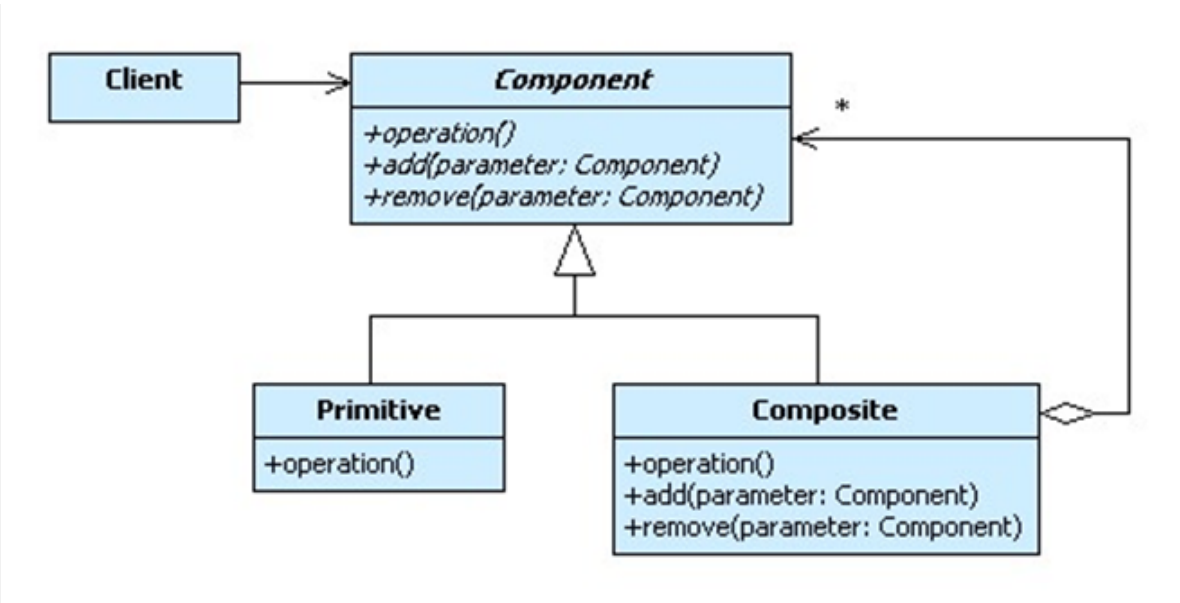
Используйте паттерн Composite если:

- Необходимо объединять группы схожих объектов и управлять ими.
- Объекты могут быть как примитивными (элементарными), так и составными (сложными). Составной объект может включать в себя коллекции других объектов, образуя сложные древовидные структуры. Пример: директория файловой системы состоит из элементов, каждый их которых также может быть директорией.
- Код клиента работает с примитивными и составными объектами единообразно.

Описание паттерна Composite

Управление группами объектов может быть непростой задачей, особенно, если эти объекты содержат собственные объекты. Паттерн компоновщик описывает, как можно применить рекурсивную композицию таким образом, что клиенту не придется проводить различие между простыми и составными объектами.

UML-диаграмма классов паттерна Composite



Участники

- **Component** (Graphic) - компонент: - объявляет интерфейс для

компонуемых объектов; предоставляет подходящую реализацию операций по умолчанию, общую для всех классов; объявляет интерфейс для доступа к потомкам и управления ими; определяет интерфейс для доступа к родителю компонента в рекурсивной структуре и при необходимости реализует его. Описанная возможность необязательна;

- Primitive (Rectangle, Line, Text, и т.п.) - примитив: - представляет листовые узлы композиции и не имеет потомков; определяет поведение примитивных объектов в композиции;

- **Composite** (Picture) - составной объект: определяет поведение компонентов, у которых есть потомки; хранит компоненты-потомки; реализует относящиеся к управлению потомками операции в интерфейсе клдсса Component;

- Client - клиент: - манипулирует объектами композиции через интерфейс Component.

Отношения

Клиенты используют интерфейс класса Component для взаимодействия с объектами в составной структуре. Если получателем запроса является листовый объект Leaf, то он и обрабатывает запрос. Когда же получателем является составной объект Composite, то обычно он перенаправляет запрос своим потомкам, возможно, выполняя некоторые дополнительные операции до или после перенаправления.

Результаты

Паттерн компоновщик:

- *определяет иерархии классов, состоящие из примитивных и составных объектов.* Из примитивных объектов можно составлять более сложные, которые, в свою очередь, участвуют в более сложных композициях и так далее. Любой клиент, ожидающий примитивного объекта, может работать и с составным;
- *упрощает архитектуру клиента.* Клиенты могут единообразно работать с индивидуальными и объектами и с составными структурами. Обычно клиенту неизвестно, взаимодействует ли он с листовым или составным объектом. Это упрощает код клиента, поскольку нет необходимости писать функции, ветвящиеся в зависимости от того, с объектом какого класса они работают;
- *облегчает добавление новых видов компонентов.* Новые подклассы классов Composite или Leaf будут автоматически работать с уже существующими структурами и клиентским кодом. Изменять клиента при добавлении новых компонентов не нужно;
- *способствует созданию общего дизайна.* Однако такая простота добавления новых компонентов имеет и свои отрицательные стороны: становится трудно наложить ограничения на то, какие объекты могут входить в состав композиции. Иногда желательно, чтобы составной объект мог включать только определенные

виды компонентов.

Реализация паттерна Composite

Применим паттерн Composite для нашей стратегической игры. Сначала сформируем различные военные соединения римской армии, а затем рассчитаем разрушающую силу.

Каждая боевая единица (всадник, лучник, пехотинец) обладает своей собственной разрушающей силой. Эти единицы могут объединяться в группы для образования более сложных военных подразделений, например, римские легионы, которые, в свою очередь, объединяясь, образуют армию. Как спроектировать такие иерархические соединения и рассчитать их боевые возможности?

Паттерн Composite вводит абстрактный базовый класс Component с поведением, общим для всех примитивных и составных объектов. Для случая стратегической игры - это метод getStrength() для подсчета разрушающей силы. Подклассы Primitive and Composite являются производными от класса Component. Составной объект Composite хранит компоненты-потомки абстрактного типа Component, каждый из которых может быть также Composite.

```
#include <iostream>
#include <vector>
#include <assert.h>

// Component
class Unit
{
  public:
    virtual int getStrength() = 0;
    virtual void addUnit(Unit* p) {
      assert( false);
    }
    virtual ~Unit() {}
};

// Primitives
class Archer: public Unit
{
  public:
    virtual int getStrength() {
      return 1;
    }
};

class Infantryman: public Unit
{
  public:
    virtual int getStrength() {
      return 2;
    }
};

class Horseman: public Unit
```

```
{
  public:
    virtual int getStrength() {
      return 3;
    }
};


// Composite
class CompositeUnit: public Unit
{
  public:
    int getStrength() {
      int total = 0;
      for(int i=0; i<c.size(); ++i)
        total += c[i]->getStrength();
      return total;
    }
    void addUnit(Unit* p) {
        c.push_back( p);
    }
    ~CompositeUnit() {
      for(int i=0; i<c.size(); ++i)
        delete c[i];
    }
  private:
   std::vector<Unit*> c;
};


// Вспомогательная функция для создания легиона
CompositeUnit* createLegion()
{
  // Римский легион содержит:
  CompositeUnit* legion = new CompositeUnit;
  // 3000 тяжелых пехотинцев
  for (int i=0; i<3000; ++i)
    legion->addUnit(new Infantryman);
  // 1200 легких пехотинцев
  for (int i=0; i<1200; ++i)
    legion->addUnit(new Archer);
  // 300 всадников
  for (int i=0; i<300; ++i)
    legion->addUnit(new Horseman);

  return legion;
}

int main()
{
```

```
// Римская армия состоит из 4-х легионов
CompositeUnit* army = new CompositeUnit;
for (int i=0; i<4; ++i)
  army->addUnit( createLegion());

cout << "Roman army damaging strength is "
    << army->getStrength() << endl;
// …
delete army;
return 0;
```

Следует обратить внимание на один важный момент. Абстрактный базовый класс Unit объявляет интерфейс для добавления новых боевых единиц addUnit(), несмотря на то, что объектам примитивных типов (Archer, Infantryman, Horseman) подобная операция не нужна. Сделано это в угоду прозрачности системы в ущерб ее безопасности. Клиент знает, что объект типа Unit всегда будет иметь метод addUnit(). Однако его вызов для примитивных объектов считается ошибочным и небезопасным.

Достоинства паттерна Composite

- В систему легко добавлять новые примитивные или составные объекты, так как паттерн Composite использует общий базовый класс Component;
- Код клиента имеет простую структуру – примитивные и составные объекты обрабатываются одинаковым образом;
- Паттерн Composite позволяет легко обойти все узлы древовидной структуры.

Недостатки паттерна Composite

Неудобно осуществить запрет на добавление в составной объект Composite объектов определенных типов. Так, например, в состав римской армии не могут входить боевые слоны.

Родственные паттерны

Отношение компонент-родитель используется в паттерне *цепочка обязанностей*.

Паттерн *декоратор* часто применяется совместно с *компоновщиком*. Когда *декораторы* и *компоновщики* используются вместе, у них обычно бывает общий родительский класс. Поэтому *декораторам* придется поддержать интерфейс компонентов такими операциями, как Add, Remove и GetChild.

Паттерн *приспособленец* позволяет разделять компоненты, но ссылаться на своих родителей они уже не могут.

Итератор можно использовать для обхода составных объектов.

Посетитель локализует операции и поведение, которые в противном случае пришлось бы распределять между классами Composite и Leaf.

1. 4. Задание

1. Разработать UML-диаграммы (диаграмму классов и диаграмму последовательности) и с помощью паттерна **Composit** решить следующую задачу.

Обеспечить контроль загрузки и готовности к отправлению самолета.

В самолете присутствуют пилоты(2), стюардессы(6), пассажиры первого(макс. 10 чел), бизнес (макс. 20 чел) и эконом (150 чел) классов. Пассажиры имеют багаж (от 5 до 60 кг). Бесплатно к провозу багажа допускается 35 кг - бизнес класс, 20 кг - эконом и без ограничения - первый класс.

Примитивный объект – пассажир, пилот стюардесса. Составной объект – Первый, бизнес и эконом классы.

Пилоты и стюардессы не могут иметь багажа.

Есть максимальная допустимая загрузка самолета багажом. При превышении – багаж снимается с рейса. Снять багаж можно только у пассажиров эконом класса.

В результате работы программ должна быть создана карта загрузки самолета с указанием перевеса багажа и информации о снятом с рейса багаже.

5. Требования к отчету

Отчет к лабораторной работе должен содержать текст работающей программы на языке программирования C++ или C# и результат выполнения программы.

6. Вопросы.

1. Объясните целесообразность применения паттерна для решения задачи лабораторной работы.

2. Каковы паттерны родственны паттерну **Composit**.

Л а б о р а т о р н а я р а б о т а № 4

«Реализация одного из структурных паттернов проектирования»

Цель работы: Применение паттерна проектирования Proxy (заместитель, surrogate, суррогат)

Продолжительность работы - 4 часа.

Содержание

Паттерн заместитель (Proxy)

Название и классификация паттерна

Заместитель - паттерн, структурирующий объекты.

Назначение

Вам нужно управлять ресурсоемкими объектами. Вы не хотите создавать экземпляры таких объектов до момента их реального использования. Proxy является суррогатом другого объекта и управляет доступом к нему.

Обсуждение паттерна Proxy

Дорожный чек - заместитель наличных денежных средств и может быть использован в путешествии вместо кошелька с деньгами. При необходимости деньги могут быть получены при предъявлении дорожного чека.

Суррогат или заместитель это объект, интерфейс которого идентичен интерфейсу реального объекта. При первом запросе клиента заместитель создает реальный объект, сохраняет его адрес и затем отправляет запрос этому реальному объекту. Все последующие запросы просто переадресуются инкапсулированному реальному объекту.

Существует четыре ситуации, когда можно использовать паттерн Proxy:

- Виртуальный proxy является заместителем объектов, создание которых обходится дорого. Реальный объект создается только при первом запросе/доступе клиента к объекту.
- Удаленный proxy предоставляет локального представителя для объекта, который находится в другом адресном пространстве ("заглушки" в RPC и CORBA).
- Защитный proxy контролирует доступ к основному объекту. "Суррогатный" объект предоставляет доступ к реальному объекту, только вызывающий объект имеет соответствующие права.
- Интеллектуальный proxy выполняет дополнительные действия при доступе к объекту.

Вот типичные области применения интеллектуальных proxy:

- Подсчет числа ссылок на реальный объект. При отсутствии ссылок память под объект автоматически освобождается (известен также как интеллектуальный указатель или smart pointer).
- Загрузка объекта в память при первом обращении к нему.
- Установка запрета на изменение реального объекта при обращении к нему других объектов.

Применимость

Паттерн заместитель применим во всех случаях, когда возникает необходимость сослаться на объект более изощренно, чем это возможно, если использовать простой указатель. Вот несколько типичных ситуаций, где заместитель оказывается полезным:

- *удаленный заместитель* предоставляет локального представителя вместо объекта, находящегося в другом адресном пространстве.

- *виртуальный заместитель* создает «тяжелые» объекты по требованию.

- *защищающий заместитель* контролирует доступ к исходному объекту. Такие заместители полезны, когда для разных объектов определены различные права доступа.

- «*умная*» *ссылка* - это замена обычного указателя. Она позволяет выполнить дополнительные действия при доступе к объекту. К типичным применениям такой ссылки можно отнести:

- подсчет числа ссылок на реальный объект, с тем чтобы занимаемую им память можно было освободить автоматически, когда не останется ни одной ссылки (такие ссылки называют еще «умными» указателями);

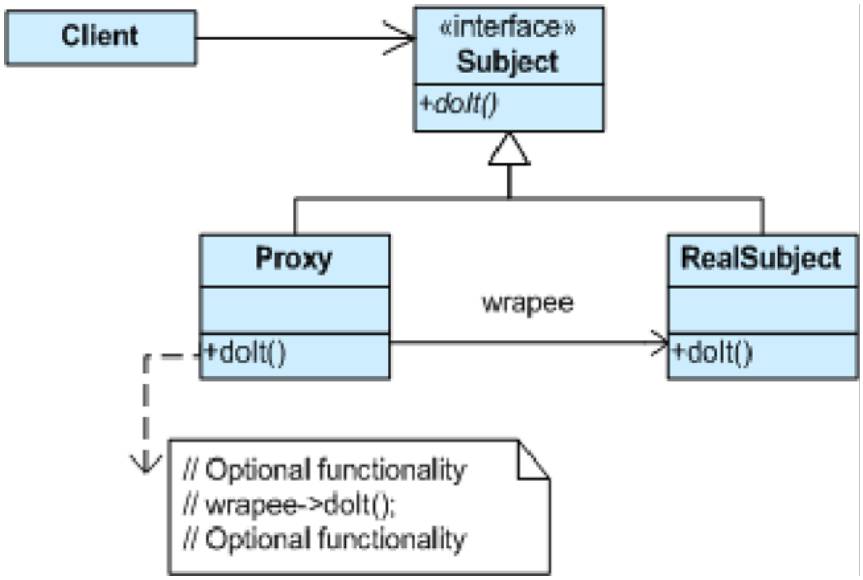
- загрузку объекта в память при первом обращении к нему;

- проверку и установку блокировки на реальный объект при обращении к нему, чтобы никакой другой объект не смог в это время изменить его.

Структура паттерна Proxy

Заместитель Proxy и реальный объект RealSubject имеют одинаковые интерфейсы класса Subject, поэтому заместитель может использоваться "прозрачно" для клиента вместо реального объекта.

UML-диаграмма классов паттерна Proxy



Участники

- **Proxy** - заместитель: - хранит ссылку, которая позволяет заместителю обратиться к реальному субъекту. Объект класса Proxy может обращаться к объекту класса Subj ect, если интерфейсы классов RealSubj ect и Subj ect одинаковы;

  - предоставляет интерфейс, идентичный интерфейсу Subj ect, так что заместитель всегда может быть подставлен вместо реального субъекта;

  - контролирует доступ к реальному субъекту и может отвечать за его создание и удаление;

  - прочие обязанности зависят от вида заместителя:

  - *удаленный заместитель* отвечает за кодирование запроса и его аргументов и отправление закодированного запроса реальному субъекту в другом адресном пространстве;

  - *виртуальный заместитель* может кэшировать дополнительную информацию о реальном субъекте, чтобы отложить его создание.

  - *защищающий заместитель* проверяет, имеет ли вызывающий объект необходимые для выполнения запроса права;

- Subject - субъект: - определяет общий для RealSubject и Proxy интерфейс, так что класс Proxy можно использовать везде, где ожидается RealSubject;

- RealSubject - реальный субъект: - определяет реальный объект, представленный заместителем.

Отношения

Proxy при необходимости переадресует запросы объекту RealSubject. Детали зависят от вида заместителя.

Результаты

С помощью паттерна заместитель при доступе к объекту вводится дополнительный уровень косвенности. У этого подхода есть много вариантов в зависимости от вида заместителя:

- удаленный заместитель может скрыть тот факт, что объект находится в другом адресном пространстве;
- виртуальный заместитель может выполнять оптимизацию, например создание объекта по требованию;
- защищающий заместитель и «умная» ссылка позволяют решать дополнительные задачи при доступе к объекту.

Есть еще одна оптимизация, которую паттерн заместитель иногда скрывает от клиента. Она называется *копированием при записи* (copy-on-write) и имеет много общего с созданием объекта по требованию. Копирование большого и сложного объекта - очень дорогая операция. Если копия не модифицировалась, то нет смысла эту цену платить. Если отложить процесс копирования, применив заместитель, то можно быть уверенным, что эта операция произойдет только тогда, когда он действительно был изменен. Чтобы во время записи можно было копировать, необходимо подсчитывать ссылки на субъект. Копирование заместителя просто увеличивает счетчик ссылок. И только тогда, когда клиент запрашивает операцию, изменяющую субъект, заместитель действительно выполняет копирование. Одновременно заместитель должен уменьшить счетчик ссылок. Когда счетчик ссылок становится равным нулю, субъект уничтожается.

Копирование при записи может существенно уменьшить плату за копирование «тяжелых» субъектов.

Особенности паттерна Proxy

- Adapter предоставляет своему объекту другой интерфейс . Proxy предоставляет тот же интерфейс. Decorator предоставляет расширенный интерфейс.
- Decorator и Proxy имеют разные цели, но схожие структуры. Оба вводят дополнительный уровень косвенности: их реализации хранят ссылку на объект, на который они отправляют запросы.

Реализация паттерна Proxy

Инициализация при первом использовании.

- Спроектируйте класс-обертку с "дополнительным уровнем косвенности".
- Этот класс содержит указатель на реальный класс.
- Этот указатель инициализируется нулевым значением.
- Реальный объект создается при поступлении запроса "на первом использовании" (отложенная инициализация или lazy intialization).
- Запрос всегда делегируется реальному объекту.
- **class RealImage**

```
{
   int m_id;
  public:
   RealImage(int i)
   {
      m_id = i;
      cout << "   $$ ctor: " << m_id << '\n';
   }
   ~RealImage()
   {
      cout << "   dtor: " << m_id << '\n';
   }
   void draw()
   {
      cout << "   drawing image " << m_id << '\n';
   }
};

// 1. Класс-обертка с "дополнительным уровнем косвенности"
class Image
{
   // 2. Класс-обертка содержит указатель на реальный класс
   RealImage *m_the_real_thing;
   int m_id;
   static int s_next;
  public:
   Image()
   {
      m_id = s_next++;
      // 3. Инициализируется нулевым значением
      m_the_real_thing = 0;
   }
   ~Image()
   {
      delete m_the_real_thing;
   }
   void draw()
   {
      // 4. Реальный объект создается при поступлении
      //    запроса "на первом использовании"
      if (!m_the_real_thing)
        m_the_real_thing = new RealImage(m_id);
      // 5. Запрос всегда делегируется реальному объекту
      m_the_real_thing->draw();
   }
};
int Image::s_next = 1;

int main()
{
```

```
  Image images[5];

  for (int i; true;)
  {
    cout << "Exit[0], Image[1-5]: ";
    cin >> i;
    if (i == 0)
      break;
    images[i - 1].draw();
  }
}
```

Вывод программы:

```
 Exit[0], Image[1-5]: 2
   $$ ctor: 2
   drawing image 2
Exit[0], Image[1-5]: 4
   $$ ctor: 4
   drawing image 4
Exit[0], Image[1-5]: 2
   drawing image 2
Exit[0], Image[1-5]: 0
   dtor: 4
   dtor: 2
```

Защитный proxy контролирует доступ к основному объекту

```
class Person
{
    string nameString;
    static string list[];
    static int next;
  public:
    Person()
    {
       nameString = list[next++];
    }
    string name()
    {
       return nameString;
    }
};
string Person::list[] =
{
  "Tom", "Dick", "Harry", "Bubba"
};
int Person::next = 0;

class PettyCashProtected
```

```
{
   int balance;
  public:
   PettyCashProtected()
   {
      balance = 500;
   }
   bool withdraw(int amount)
   {
      if (amount > balance)
        return false;
      balance -= amount;
      return true;
   }
   int getBalance()
   {
      return balance;
   }
};

class PettyCash
{
   PettyCashProtected realThing;
  public:
   bool withdraw(Person &p, int amount)
   {
      if (p.name() == "Tom" || p.name() == "Harry"
                        || p.name() == "Bubba")
        return realThing.withdraw(amount);
      else
        return false;
   }
   int getBalance()
   {
      return realThing.getBalance();
   }
};

int main()
{
  PettyCash pc;
  Person workers[4];
  for (int i = 0, amount = 100; i < 4; i++, amount += 100)
    if (!pc.withdraw(workers[i], amount))
      cout << "No money for " << workers[i].name() << '\n';
    else
      cout << amount << " dollars for " << workers[i].name() << '\n';
  cout << "Remaining balance is " << pc.getBalance() << '\n';
}
```

- Вывод программы:
- **100 dollars for Tom**
- **No money for Dick**
- **300 dollars for Harry**
- **No money for Bubba**
- **Remaining balance is 100**
-
- ***Родственные паттерны***
- Паттерн *адаптер* предоставляет другой интерфейс к адаптируемому объекту. Напротив, заместитель в точности повторяет интерфейс своего субъекта. Однако, если *заместитель* используется для ограничения доступа, он может отказаться выполнять операцию, которую субъект выполнил бы, поэтому на самом деле интерфейс *заместителя* может быть и подмножеством интерфейса субъекта.
- Реализация *декоратора* похожа на реализацию *заместителя*, но назначение совершенно иное. *Декоратор* добавляет объекту новые обязанности, а *заместитель* контролирует доступ к объекту.

4. Задание

4. Разработать UML-диаграммы (диаграмму классов и диаграмму последовательности) и с помощью паттерна «proxy» решить следующую задачу.

Создать простейшую модель фрагмента графического редактора, позволяющую нарисовать на экране монитора бокс, имеющий размеры реального изображения, хранящегося на диске под именем TestImage. Используя паттерн «proxy» обеспечить свободное перемещение бокса с помощью «мыши» по экрану. При двойном нажатии на правую клавишу «мыши» обеспечить загрузку реального изображения в нарисованный бокс.

5. Требования к отчету

Отчет к лабораторной работе должен содержать текст работающей программы на языке программирования C++ или C# и результат выполнения программы.

6. Вопросы.

1. Чем похожи и чем отличаются паттерны Proxy, Adapter и Decorator

Л а б о р а т о р н а я р а б о т а № 5

«Реализация одного из паттернов поведения»

Цель работы: Применение паттерна проектирования Interpreter (интерпетатор)
Продолжительность работы - 4 часа.

Содержание

Паттерны поведения рассматривают вопросы о связях между объектами и распределением обязанностей между ними. Для этого могут использоваться механизмы, основанные как на наследовании, так и на композиции.

Назначение паттерна Interpreter

2. Для заданного языка определяет представление его грамматики, а также интерпретатор предложений этого языка.
3. Отображает проблемную область в язык, язык – в грамматику, а грамматику – в иерархии объектно-ориентированного проектирования.

Решаемая проблема

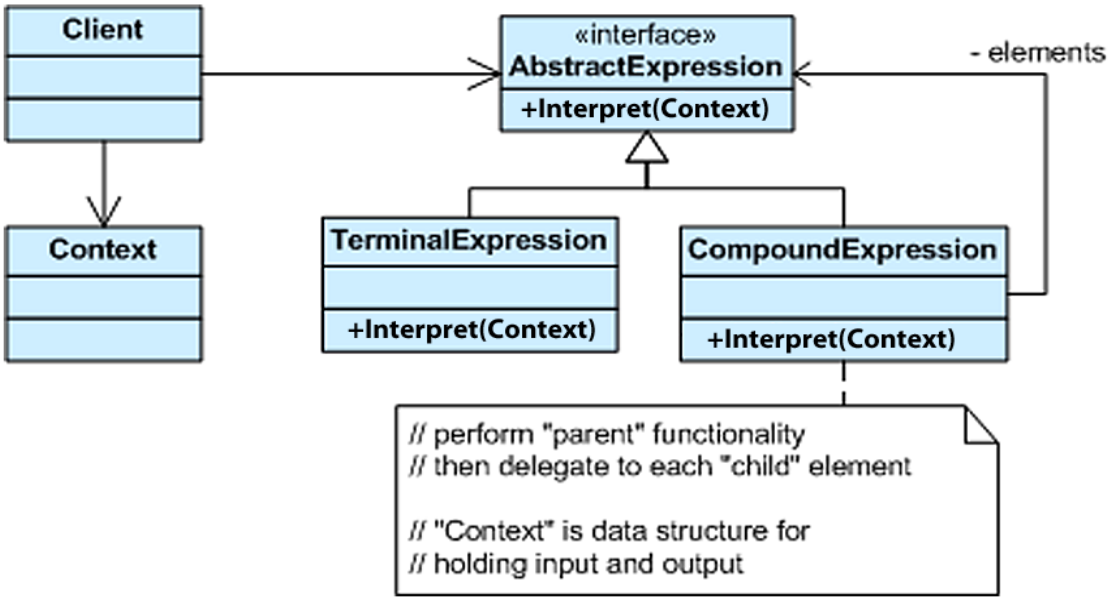
Пусть в некоторой, хорошо определенной области периодически случается некоторая проблема. Если эта область может быть описана некоторым “языком“, то проблема может быть легко решена с помощью “интерпретирующей машины“.

Обсуждение паттерна Interpreter

Паттерн Interpreter определяет грамматику простого языка для проблемной области, представляет грамматические правила в виде языковых предложений и интерпретирует их для решения задачи. Для представления каждого грамматического правила паттерн Interpreter использует отдельный класс. А так как грамматика, как правило, имеет иерархическую структуру, то иерархия наследования классов хорошо подходит для ее описания.

Абстрактный базовый класс определяет метод interpret(), принимающий (в качестве аргумента) текущее состояние языкового потока. Каждый конкретный подкласс реализует метод interpret(), добавляя свой вклад в процесс решения проблемы.

Структура паттерна Interpreter



Паттерн Interpreter моделирует проблемную область с помощью рекурсивной грамматики. Каждое грамматическое правило может быть либо составным (правило ссылается на другие правила) либо терминальным (листовой узел в структуре ”дерево”).

Для рекурсивного обхода ”предложений” при их интерпретации используется паттерн Composite.

UML-диаграмма классов паттерна Interpreter

Пример паттерна Interpreter

Паттерн Intepreter определяет грамматическое представление для языка и интерпретатор для интерпретации грамматики. Музыканты являются примерами интерпретаторов. Тональность и продолжительность звуков могут быть описаны нотами. Такое представление является музыкальным языком. Музыканты, используя ноты, способны воспроизвести оригинальные частоту и длительность каждого представленного звука.

Использование паттерна Interpreter

4. Определите “малый“ язык, “инвестиции” в который будут оправданными.
5. Разработайте грамматику для языка.
6. Для каждого грамматического правила (продукции) создайте свой класс.
7. Полученный набор классов организуйте в структуру с помощью паттерна Composite.
8. В полученной иерархии классов определите метод interpret(Context).
9. Объект Context инкапсулирует информацию, глобальную по отношению к интерпретатору. Используется классами во время процесса ”интерпретации”.

Особенности паттерна Interpreter

- Абстрактное синтаксическое дерево интерпретатора – пример паттерна Composite.
- Для обхода узлов дерева может применяться паттерн Iterator.
- Терминальные символы могут разделяться с помощью Flyweight.
- Паттерн Interpreter не рассматривает вопросы синтаксического разбора. Когда грамматика очень сложная, должны использоваться другие методики.

Реализация паттерна Interpreter

Совместное использование паттернов Interpreter и Template Method

Рассмотрим задачу интерпретирования (вычисления) значений строковых представлений римских чисел. Используем следующую грамматику.

```
romanNumeral ::= {thousands} {hundreds} {tens} {ones}
thousands,hundreds,tens,ones ::= nine | four | {five} {one}
{one} {one}
nine ::= "CM" | "XC" | "IX"
four ::= "CD" | "XL" | "IV"
five ::= 'D' | 'L' | 'V'
one  ::= 'M' | 'C' | 'X' | 'I'
```

Для проверки и интерпретации строки используется иерархия классов с общим базовым классом RNInterpreter, имеющим 4 под-интерпретатора. Каждый под-интерпретатор получает "контекст" (оставшуюся неразобранную часть строки и накопленное вычисленное значение разобранной части) и вносит свой вклад в процесс обработки. Под-переводчики просто определяют шаблонные методы, объявленные в базовом классе RNInterpreter.

```
#include <iostream.h>
```

```
#include <string.h>

class Thousand;
class Hundred;
class Ten;
class One;

class RNInterpreter
{
  public:
    RNInterpreter(); // ctor for client
    RNInterpreter(int){}
    // ctor for subclasses, avoids infinite loop
    int interpret(char*); // interpret() for client
    virtual void interpret(char *input, int &total)
    {
        // for internal use
        int index;
        index = 0;
        if (!strncmp(input, nine(), 2))
        {
            total += 9 * multiplier();
            index += 2;
        }
        else if (!strncmp(input, four(), 2))
        {
            total += 4 * multiplier();
            index += 2;
        }
        else
        {
            if (input[0] == five())
            {
                total += 5 * multiplier();
                index = 1;
            }
            else
              index = 0;
            for (int end = index + 3; index < end; index++)
              if (input[index] == one())
                total += 1 * multiplier();
              else
                break;
        }
        strcpy(input, &(input[index]));
    } // remove leading chars processed
  protected:
    // cannot be pure virtual because client asks for instance
    virtual char one(){}
    virtual char *four(){}
    virtual char five(){}
```

```
    virtual char *nine(){}
    virtual int multiplier(){}
  private:
    RNInterpreter *thousands;
    RNInterpreter *hundreds;
    RNInterpreter *tens;
    RNInterpreter *ones;
};

class Thousand: public RNInterpreter
{
  public:
    // provide 1-arg ctor to avoid infinite loop in base class
ctor
    Thousand(int): RNInterpreter(1){}
  protected:
    char one()
    {
        return 'M';
    }
    char *four()
    {
        return "";
    }
    char five()
    {
        return '\0';
    }
    char *nine()
    {
        return "";
    }
    int multiplier()
    {
        return 1000;
    }
};

class Hundred: public RNInterpreter
{
  public:
    Hundred(int): RNInterpreter(1){}
  protected:
    char one()
    {
        return 'C';
    }
    char *four()
    {
        return "CD";
    }
```

```
    char five()
    {
        return 'D';
    }
    char *nine()
    {
        return "CM";
    }
    int multiplier()
    {
        return 100;
    }
};

class Ten: public RNInterpreter
{
  public:
    Ten(int): RNInterpreter(1){}
  protected:
    char one()
    {
        return 'X';
    }
    char *four()
    {
        return "XL";
    }
    char five()
    {
        return 'L';
    }
    char *nine()
    {
        return "XC";
    }
    int multiplier()
    {
        return 10;
    }
};

class One: public RNInterpreter
{
  public:
    One(int): RNInterpreter(1){}
  protected:
    char one()
    {
        return 'I';
    }
    char *four()
```

```
    {
        return "IV";
    }
    char five()
    {
        return 'V';
    }
    char *nine()
    {
        return "IX";
    }
    int multiplier()
    {
        return 1;
    }
};

RNInterpreter::RNInterpreter()
{
  // use 1-arg ctor to avoid infinite loop
  thousands = new Thousand(1);
  hundreds = new Hundred(1);
  tens = new Ten(1);
  ones = new One(1);
}

int RNInterpreter::interpret(char *input)
{
  int total;
  total = 0;
  thousands->interpret(input, total);
  hundreds->interpret(input, total);
  tens->interpret(input, total);
  ones->interpret(input, total);
  if (strcmp(input, ""))
  // if input was invalid, return 0
    return 0;
  return total;
}

int main()
{
  RNInterpreter interpreter;
  char input[20];
  cout << "Enter Roman Numeral: ";
  while (cin >> input)
  {
    cout << "   interpretation is "
         << interpreter.interpret(input) << endl;
    cout << "Enter Roman Numeral: ";
  }
```

```
}
```

Вывод программы:

```
Enter Roman Numeral: MCMXCVI
   interpretation is 1996
Enter Roman Numeral: MMMCMXCIX
   interpretation is 3999
Enter Roman Numeral: MMMM
   interpretation is 0
Enter Roman Numeral: MDCLXVIIII
   interpretation is 0
Enter Roman Numeral: CXCX
   interpretation is 0
Enter Roman Numeral: MDCLXVI
   interpretation is 1666
Enter Roman Numeral: DCCCLXXXVIII
   interpretation is 888
```

4. Задание

5. Разработать UML-диаграммы (диаграмму классов и диаграмму последовательности) и с помощью паттерна « **Interpreter** » решить следующую задачу.

Создать простейшую интерпретатор текстового редактора, позволяющий исправлять стандартные ошибки, допускаемые при подготовке обычных текстов.

Как правило, человек при наборе текста в программе WORD не обращает внимания на соблюдение правил структурного оформления текстов, что вызывает некоторые трудности при чистовой верстке.

Типичные структурные ошибки:

1. Множественные пробелы;
2. Использование дефиса вместо тире;
3. Использование в качестве кавычек символов “”, тогда как стандартом является использование «»;
4. Неправильное использование табуляторов
5. Наличие «лишнего» пробела после открывающей скобки, перед закрывающей скобкой, перед запятой, перед точкой;
6. Наличие множественных символов перевода строки

Разработать грамматику и иерархию классов. Используя паттерн « **Interpreter** » провести синтаксический анализ текста и устранить перечисленные ошибки.

5. Требования к отчету

Отчет к лабораторной работе должен содержать текст работающей программы на языке программирования C++ или C# и результат выполнения программы.

6. Вопросы.

1. С помощью каких еще паттернов проектрования можно решить поставленную задачу?

Л а б о р а т о р н а я р а б о т а № 6

«Реализация одного из паттернов поведения»

Цель работы: Применение паттерна проектирования Observer (наблюдатель)
Продолжительность работы - 4 часа.

Содержание

Название и классификация паттерна

Наблюдатель - паттерн поведения объектов.

Назначение паттерна Observer

- Паттерн Observer определяет зависимость "один-ко-многим" между объектами так, что при изменении состояния одного объекта все зависящие от него объекты уведомляются и обновляются автоматически.
- Паттерн Observer инкапсулирует главный (независимый) компонент в абстракцию Subject и изменяемые (зависимые) компоненты в иерархию Observer.
- Паттерн Observer определяет часть "View" в модели Model-View-Controller (MVC) .

Решаемая проблема

Имеется система, состоящая из множества взаимодействующих классов. При этом взаимодействующие объекты должны находиться в согласованных состояниях. Вы хотите избежать монолитности такой системы, сделав классы слабо связанными (или повторно используемыми).

Обсуждение паттерна Observer

В результате разбиения системы на множество совместно работающих классов появляется необходимость поддерживать согласованное состояние взаимосвязанных

объектов. Но не хотелось бы, чтобы за согласованность надо было платить жесткой связанностью классов, так как это уменьшает возможности повторного использования.

Паттерн *наблюдатель* описывает, как устанавливать такие отношения. Паттерн *Observer* определяет объект *Subject*, хранящий данные (модель), а всю функциональность "представлений" делегирует слабосвязанным отдельным объектам *Observer*. У субъекта может быть сколько угодно зависимых от него наблюдателей. Все наблюдатели уведомляются об изменениях в состоянии субъекта. Получив уведомление, наблюдатель опрашивает субъекта, чтобы синхронизировать с ним свое состояние. Такого рода взаимодействие часто называется отношением издатель-подписчик. Субъект издает или публикует уведомления и рассылает их, даже не имея информации о том, какие объекты являются подписчиками. На получение уведомлений может подписаться неограниченное количество наблюдателей.

При создании наблюдатели *Observer* регистрируются у объекта *Subject*. Когда объект *Subject* изменяется, он извещает об этом всех зарегистрированных наблюдателей. После этого каждый обозреватель запрашивает у объекта *Subject* ту часть состояния, которая необходима для отображения данных.

Такая схема позволяет динамически настраивать количество и "типы" представлений объектов.

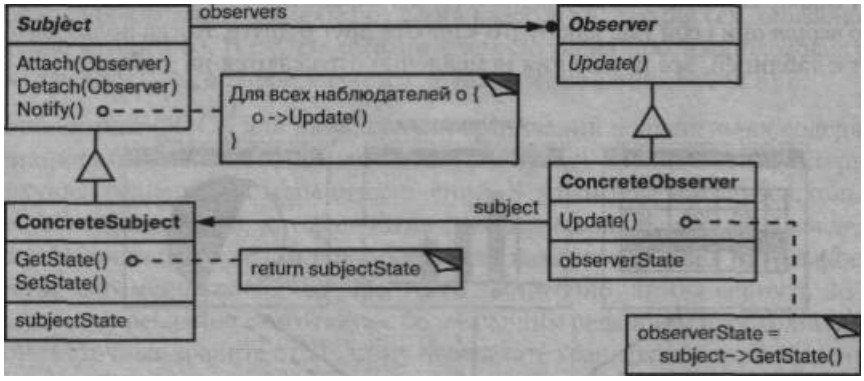
Применимость

- у абстракции есть два аспекта, один из которых зависит от другого. Инкапсуляции этих аспектов в разные объекты позволяют изменять и повторно использовать их независимо;
- при модификации одного объекта требуется изменить другие и заранее неизвестно, сколько именно объектов нужно изменить;
- один объект должен оповещать других, не делая предположений об уведомляемых объектах. Другими словами, вы не хотите, чтобы объекты были тесно связаны между собой.

Структура паттерна Observer

Subject представляет главную (независимую) абстракцию. Observer представляет изменяемую (зависимую) абстракцию. Субъект извещает наблюдателей о своем изменении, на что каждый наблюдатель может запросить состояние субъекта.

UML-диаграмма классов паттерна Observer



Участники

- **Subject** - субъект:

  - располагает информацией о своих наблюдателях. За субъектом может ≪следить≫ любое число наблюдателей;

  - предоставляет интерфейс для присоединения и отделения наблюдателей;

  Observer - наблюдатель: определяет интерфейс обновления для объектов, которые должны быть уведомлены об изменении субъекта;

- **ConcreteSubject** - конкретный субъект:

  - сохраняет состояние, представляющее интерес для конкретного наблюдателя *ConcreteObserver*;

  - посылает информацию своим наблюдателям, когда происходит изменение;

- **ConcreteObserver** - конкретный наблюдатель:

  - хранит ссылку на объект класса ConcreteSubj ect;

  - сохраняет данные, которые должны быть согласованы с данными субъекта;

  - реализует интерфейс обновления, определенный в классе Observer, чтобы поддерживать согласованность с субъектом.

Реализация паттерна Observer

Продемонстрируем применение паттерна на примере системы торгов на валютной бирже. В этой системе применение паттерна «Наблюдатель» придется очень кстати. Есть объект, за которым производится наблюдение – это валютная биржа, содержащая информацию о текущих курсах. Помимо биржи есть игроки, которым необходимо знать об изменениях курсов, чтобы на основании этой информации принимать решения о ставках. Реализуем биржу и двух игроков, представляющих банк и брокера.

Участники:

IObservable – интерфейс, представляющий наблюдаемый объект

IObserver – интерфейс, представляющий наблюдателя

Stock – реализация *IObservable*, эмулятор валютной биржи

Broker – реализация *IObserver*, представляющая брокера

Bank – реализация *IObserver*, представляющая банк

StockInfo – содержит текущую информацию о торгах

Пример работает следующим образом: Создаются 2 наблюдателя (биржевых игрока), которые подписываются на обновления состояния биржи(Stock хранит List<>,

содержащий всех подписчиков). Как только ситуация на рынке меняется (метод Stock.Market()), оба игрока, подписанных на обновления, получают об этом уведомление (через вызов метода Update()), и принимают решение о покупке или продаже валюты. После первой сделки, брокер выходит из торгов, отписываясь от Stock (через вызов метода StopTrade()), и далее торгует только банк.

Особенности паттерна Observer

Паттерны *Chain of Responsibility, Command, Mediator* и *Observer* показывают, как можно разделить отправителей и получателей запросов с учетом своих особенностей. *Chain of Responsibility* передает запрос отправителя по цепочке потенциальных получателей. *Command* определяет связь - "оправитель-получатель" с помощью подкласса. В *Mediator* отправитель и получатель ссылаются друг на друга косвенно, через объект-посредник. В паттерне *Observer* связь между отправителем и получателем получается слабой, при этом число получателей может конфигурироваться во время выполнения.

Mediator и *Observer* являются конкурирующими паттернами. Если *Observer* распределяет взаимодействие с помощью объектов "наблюдатель" и "субъект", то *Mediator* использует объект-посредник для инкапсуляции взаимодействия между другими объектами. Мы обнаружили, что легче сделать повторно используемыми Наблюдателей и Субъектов, чем Посредников.

Mediator может использовать *Observer* для динамической регистрации коллег и их взаимодействия с посредником.

Исходный код:

```
using System;
using System.Collections.Generic;
class Program
{
  static void Main(string[] args)
  {
    Stock stock = new Stock();
    Bank bank = new Bank("Сбербанк", stock);
    Broker broker = new Broker("Валерий Андреевич", stock);
    // изменение ситуации на бирже
    stock.Market();
    // брокер выходит из торгов
    broker.StopTrade();

    stock.Market();

    Console.Read();
  }
}
```

```
interface IObserver
{
   void Update(Object ob);
}

interface IObservable
{
   void RegisterObserver(IObserver o);
   void RemoveObserver(IObserver o);
   void NotifyObservers();
}

class Stock : IObservable
{
   StockInfo sInfo; // текущая информация о торгах

   List<IObserver> observers;
   public Stock()
   {
      observers = new List<IObserver>();
      sInfo= new StockInfo();
   }
   public void RegisterObserver(IObserver o)
   {
      observers.Add(o);
   }

   public void RemoveObserver(IObserver o)
   {
      observers.Remove(o);
   }

   public void NotifyObservers()
   {
      foreach(IObserver o in observers)
      {
         o.Update(sInfo);
      }
   }

   public void Market()
   {
      Random rnd = new Random();
```

```
        sInfo.USD = rnd.Next(20, 40);
        sInfo.Euro = rnd.Next(30, 50);
        NotifyObservers();
    }
}

class StockInfo
{
    public int USD { get; set; }
    public int Euro { get; set; }
}

class Broker : IObserver
{
    public string Name { get; set; }
    IObservable stock;
    public Broker(string name, IObservable obs)
    {
        this.Name = name;
        stock = obs;
        stock.RegisterObserver(this);
    }
    public void Update(object ob)
    {
        StockInfo sInfo = (StockInfo)ob;

        if(sInfo.USD>30)
            Console.WriteLine("Брокер {0} продает доллары;  Курс доллара:
{1}", this.Name, sInfo.USD);
        else
            Console.WriteLine("Брокер {0} покупает доллары;  Курс доллара:
{1}", this.Name, sInfo.USD);
    }
    public void StopTrade()
    {
        stock.RemoveObserver(this);
        stock=null;
    }
}

class Bank : IObserver
{
    public string Name { get; set; }
```

```
    IObservable stock;
    public Bank(string name, IObservable obs)
    {
        this.Name = name;
        stock = obs;
        stock.RegisterObserver(this);
    }
    public void Update(object ob)
    {
        StockInfo sInfo = (StockInfo)ob;

        if (sInfo.Euro > 40)
            Console.WriteLine("Банк {0} продает евро;  Курс евро: {1}", this.Name, sInfo.Euro);
        else
            Console.WriteLine("Банк {0} покупает евро;  Курс евро: {1}", this.Name, sInfo.Euro);
    }
}
```

Вывод программы:

Банк Сбербанк продает евро; Курс евро: 46

Брокер Валерий Андреевич покупает доллары; Курс доллара: 28

Банк Сбербанк продает евро; Курс евро: 46

4. Задание

Разработать UML-диаграммы (диаграмму классов и диаграмму последовательности) и с помощью паттерна « ***Observer*** » решить следующую задачу.

Деканат отслеживает текущую успеваемомсть в группах факультета по одной из дисциплин. Пеподаватели раз в неделю создают текущую успеваемость и размещают ее в базе данных. Если преподаватель вовремя не создал текущую успеваемость – деканат оповещает об этом кафедру.

5. Требования к отчету

Отчет к лабораторной работе должен содержать текст работающей программы на языке программирования C++ или C# и результат выполнения программы.

6. Вопросы.

1. С помощью каких еще паттернов проектрования можно решить поставленную задачу?

Л а б о р а т о р н а я р а б о т а № 7-8

«Самостоятельная работа по реализации компьютерной игры с применением паттернов проектирования»

Цель работы: Применение паттернов проектирования.
Продолжительность работы - 8 часа.

Содержание

Как решать задачи проектирования с помощью паттернов

Поиск подходящих объектов

Объектно-ориентированные программы состоят из объектов. *Объект* сочетает данные и процедуры для их обработки. Такие процедуры обычно называют *методами* или *операциями.* Объект выполняет операцию, когда получает *запрос* (или *сообщение)* от *клиента.*

Посылка запроса - это *единственный* способ заставить объект выполнить операцию. А выполнение операции - *единственный* способ изменить внутреннее состояние объекта.

Имея в виду эти два ограничения, говорят, что внутреннее состояние объекта *инкапсулировано:* к нему нельзя получить непосредственный доступ, то есть представление объекта закрыто от внешней программы.

Самая трудная задача в объектно-ориентированном проектировании - разложить систему на объекты. При ее решении приходится учитывать множество факторов: инкапсуляцию, глубину детализации, наличие зависимостей, гибкость, производительность, развитие, повторное использование и т.д. и т.п. Все это влияет на декомпозицию, причем часто противоречивым образом.

Можно построить модель реального мира или перенести выявленные при анализе системы объекты на свой дизайн. Но нередко появляются и классы, у которых нет прототипов в реальном мире. Это могут быть классы как низкого уровня, например массивы, так и высокого. Если придерживаться строгого моделирования и ориентироваться только на реальный мир, то получится система, отражающая сегодняшние потребности, но, возможно, не учитывающая будущего развития. Абстракции, возникающие в ходе проектирования, - ключ к гибкому дизайну.

Паттерны проектирования помогают выявить не вполне очевидные абстракции и объекты, которые могут их использовать. Например, объектов, представляющих процесс или алгоритм, в действительности нет, но они являются неотъемлемыми составляющими гибкого дизайна. Эти объекты редко появляются во время анализа и даже на ранних стадиях проектирования. Работа с ними начинается позже, при попытках сделать дизайн более гибким и пригодным для повторного использования.

Паттерны позволяют изменять отдельные составляющие системы независимо от прочих. Следствие: система менее подвержена влиянию изменений конкретного вида.

Специфицирование интерфейсов объекта

При объявлении объектом любой операции должны быть заданы: имя операции, объекты, передаваемые в качестве параметров, и значение, возвращаемое операцией. эту триаду называют *сигнатурой* операции. Множество сигнатур всех определенных для объекта операций называется *интерфейсом* этого объекта. Интерфейс описывает все множество запросов, которые можно отправить объекту. Любой запрос, сигнатура которого соответствует интерфейсу объекта, может быть ему послан.

В объектно-ориентированных системах интерфейсы фундаментальны. Об объектах известно только то, что они сообщают о себе через свои интерфейсы. Никакого способа получить информацию об объекте или заставить его что-то сделать в обход интерфейса не существует. Интерфейс объекта ничего не говорит о его реализации; разные объекты вправе реализовывать сходные запросы совершенно по-разному. Это означает, что два объекта с различными реализациями могут иметь одинаковые интерфейсы.

Когда объекту посылается запрос, то операция, которую он будет выполнять, зависит как от запроса, так и от объекта-адресата. Разные объекты, поддерживающие одинаковые интерфейсы, могут выполнять в ответ на такие запросы разные операции. Ассоциация запроса с объектом и одной из его операций во время выполнения называется *динамическим связыванием.*

Динамическое связывание означает, что отправка некоторого запроса не опре-деляет никакой конкретной реализации до момента выполнения. Следовательно, допустимо написать программу, которая ожидает объект с конкретным интерфейсом, точно зная, что любой объект с подходящим интерфейсом сможет принять этот запрос. Более того, динамическое связывание позволяет во время выполнения подставить вместо одного объекта другой, если он имеет точно такой же интерфейс. Такая взаимозаменяемость называется *полиморфизмом* и является важнейшей особенностью объектно-ориентированных систем. Она позволяет клиенту не делать почти никаких предположений об объектах, кроме того, что они поддерживают определенный интерфейс. Полиморфизм упрощает определение клиентов, позволяет отделить объекты друг от друга и дает объектам возможность изменять взаимоотношения во время выполнения.

Паттерны проектирования позволяют определять интерфейсы, задавая их основные элементы и то, какие данные можно передавать через интерфейс. Паттерн может также «сказать», что не должно проходить через интерфейс. Можно описать, как инкапсулировать и сохранить внутреннее состояние объекта таким образом, чтобы в будущем его можно было восстановить точно в таком же состоянии.

Паттерны проектирования специфицируют также отношения между интерфейсами. В частности, нередко они содержат требование, что некоторые классы должны иметь схожие интерфейсы, а иногда налагают ограничения на интерфейсы классов.

Проектирование с учетом будущих изменений

Системы необходимо проектировать с учетом их дальнейшего развития. Для проектирования системы, устойчивой к таким изменениям, следует предположить, как она будет изменяться на протяжении отведенного ей времени жизни.

Если при проектировании системы не принималась во внимание возможность изменений, то есть вероятность, что в будущем ее придется полностью перепроек-тировать. Перепроектирование отражается на многих частях системы, поэтому непредвиденные изменения всегда оказываются дорогостоящими.

Благодаря паттернам систему всегда можно модифицировать определенным образом. Каждый паттерн позволяет изменять некоторый аспект системы независимо от всех прочих, таким образом, она менее подвержена влиянию изменений конкретного вида. Вот

некоторые типичные причины перепроектирования, а также паттерны, которые позволяют этого избежать.

1. При создании объекта явно указывается класс. Задание имени класса привязывает нас к конкретной реализации, а не к конкретному интерфейсу. Это может осложнить изменение объекта в будущем. Чтобы уйти от такой проблемы, нужно создавать объекты косвенно.

Паттерны проектирования: абстрактная фабрика, фабричный метод, прототип.

2. Зависимость от конкретных операций. Задавая конкретную операцию, мы ограничиваем себя единственным способом выполнения запроса. Если не включать запросы в код, то будет проще изменить способ удовлетворения запроса как на этапе компиляции, так и на этапе выполнения.

Паттерны проектирования: цепочка обязанностей, команда.

3. Зависимость от аппаратной и программной платформ.

Внешние интерфейсы операционной системы и интерфейсы прикладных программ (API) различны на разных программных и аппаратных платформах. Если программа зависит от конкретной платформы, ее будет труднее перенести на другие. Поэтому при проектировании систем важно ограничивать платформенные зависимости.

Паттерны проектирования: абстрактная фабрика, мост.

4. Зависимость от представления или реализации объекта. Если клиент “знает”, как объект представлен, хранится или реализован, то при изменении объекта может оказаться необходимым изменить и клиента. Сокрытие этой информации от клиентов поможет уберечься от каскада изменений.

Паттерны проектирования: абстрактная фабрика, мост, хранитель, заместитель.

5. Зависимость от алгоритмов. Во время разработки и последующего использования алгоритмы часто расширяются, оптимизируются и заменяются. Зависящие от алгоритмов объекты придется переписывать при каждом изменении алгоритма. Поэтому алгоритмы, вероятность изменения которых высока, следует изолировать.

Паттерны проектирования: мост, итератор, стратегия, шаблонный метод, посетитель.

6. Сильная связанность. Сильно связанные между собой классы трудно использовать порознь, так как они зависят друг от друга. Сильная связанность приводит к появлению монолитных систем, в которых нельзя ни изменить, ни удалить класс без знания деталей и модификации других классов. Такую систему трудно изучать, переносить на другие платформы и сопровождать.

Слабая связанность повышает вероятность того, что класс можно будет повторно использовать сам по себе. При этом изучение, перенос, модификация и сопровождение системы намного упрощаются. Для поддержки слабо связанных систем в паттернах проектирования применяются такие методы, как абстрактные связи и разбиение на слои.

Паттерны проектирования: абстрактная фабрика, мост, цепочка обязанностей, команда, фасад, посредник, наблюдатель.

7. расширение функциональности за счет порождения подклассов. Специализация объекта путем создания подкласса часто оказывается непростым делом. С каждым новым подклассом связаны фиксированные издержки реализации (инициализация, очистка и т.д.). Для определения подкласса необходимо также ясно представлять себе устройство родительского класса. Например, для замещения одной операции может потребоваться заместить и другие. Замещение операции может оказаться необходимым для того, чтобы можно было вызвать унаследованную операцию. Кроме того, порождение подклассов ведет к комбинаторному росту числа классов, поскольку даже для реализации простого расширения может понадобиться много новых подклассов.

Композиция объектов и делегирование - гибкие альтернативы наследованию для комбинирования поведений. Приложению можно добавить новую функциональность, меняя способ композиции объектов, а не определяя новые подклассы уже имеющихся классов. С другой стороны, при интенсивном использовании композиции объектов проект может оказаться трудным для понимания. С помощью многих паттернов проектирования удается построить такое решение, где специализация достигается за счет определения одного подкласса и комбинирования его экземпляров с уже существующими.

Паттерны проектирования: мост, цепочка обязанностей, компоновщик,

декоратор, наблюдатель, стратегия.

8. неудобства при изменении классов. Иногда нужно модифицировать класс, но делать это неудобно. Допустим, вам нужен исходный код, а его нет (так обстоит дело с коммерческими библиотеками классов). Или любое изменение тянет за собой модификации множества существующих подклассов. Благодаря паттернам проектирования можно модифицировать классы и при таких условиях.

Паттерны проектирования: адаптер, декоратор, посетитель.

4. Задание

В группах по 4-5 человек разработать игру по выбору: пошаговую стратегию, логигу, «бродилку» с применением нескольких паттернов проектирование в комрлексе. Разработать UML-диаграммы (диаграмму классов и диаграмму последовательности) решаемой задачи.

Результатом лабораторной работы должна быть работающая игра.