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.

Лабораторная работа № 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

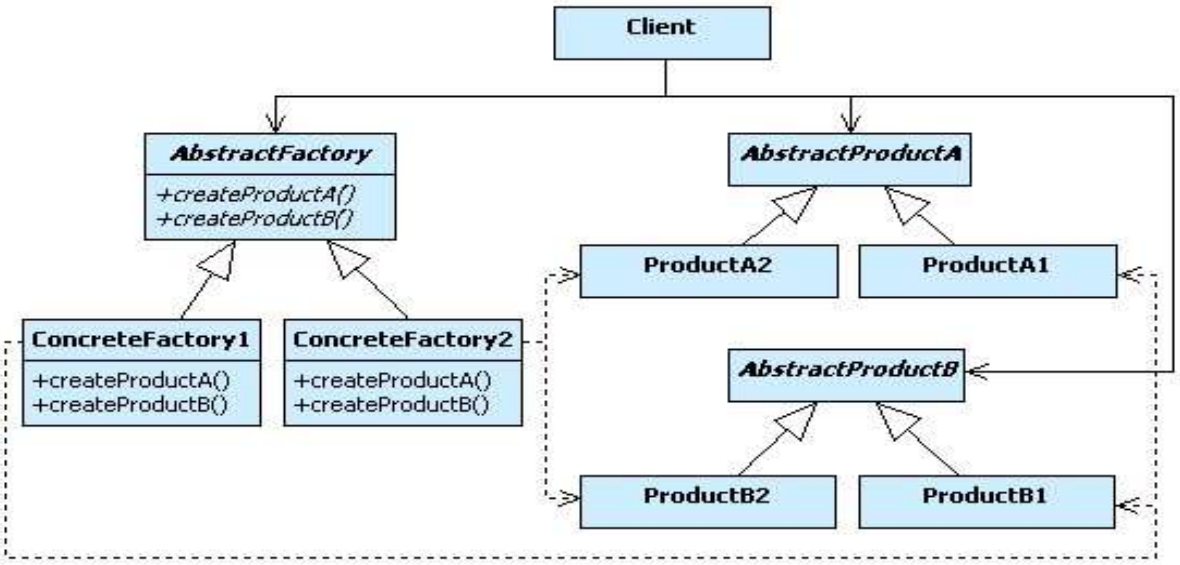
Использование паттерна Abstract Factory (абстрактная фабрика) целесообразно если:

- 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.

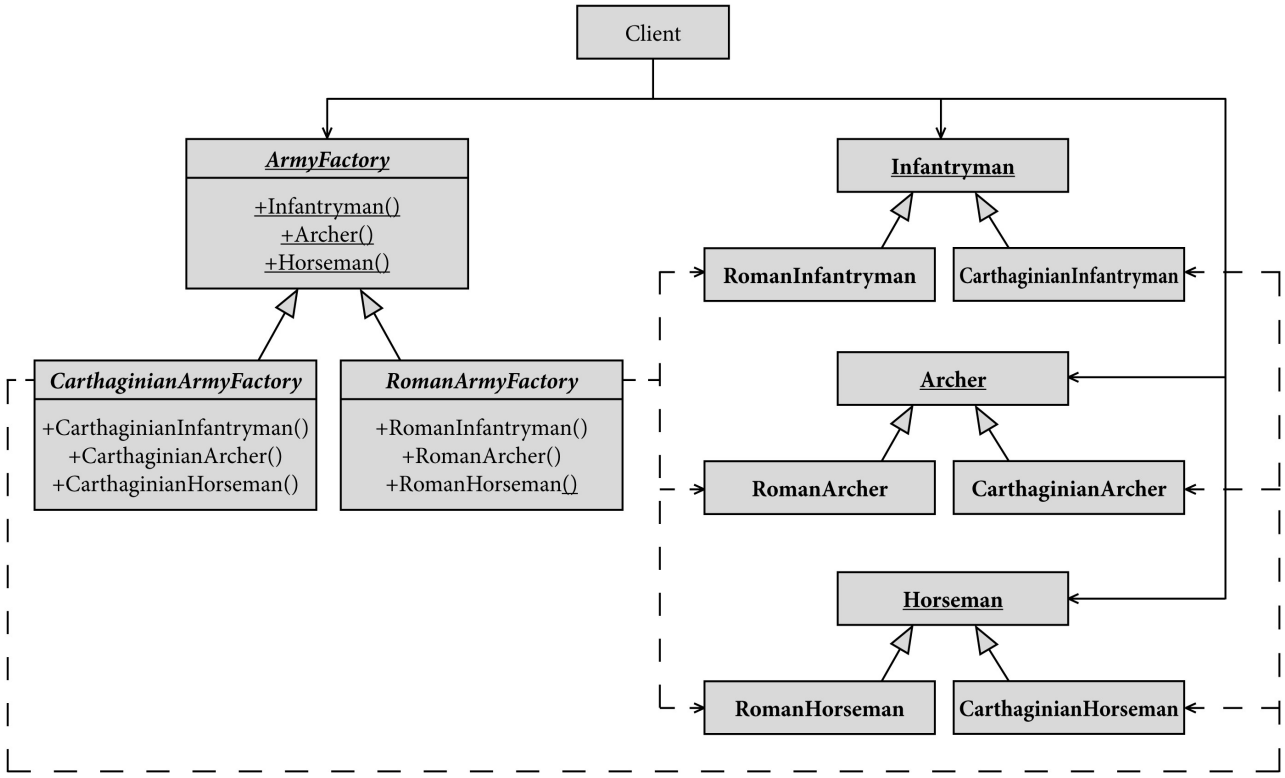


Рисунок 1. UML-диаграмма классов для военной стратегии "Пунические войны"

```
#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() {}
};


// К л а с с ы  в с е х  в и д о в  в о и н о в  Р и м с к о й  а р м и и
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;
    }
};


// К л а с с ы  в с е х  в и д о в  в о и н о в  а р м и и
К а р ф а г е н а
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.

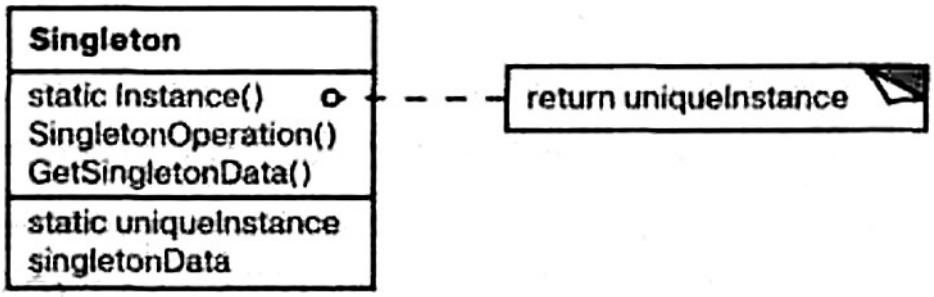
Применимость

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;
    // К о н с т р у к т о р ы  и  о п е р а т о р  п р и с в а и в а н и я
н е д о с т у п н ы  к л и е н т а м
    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

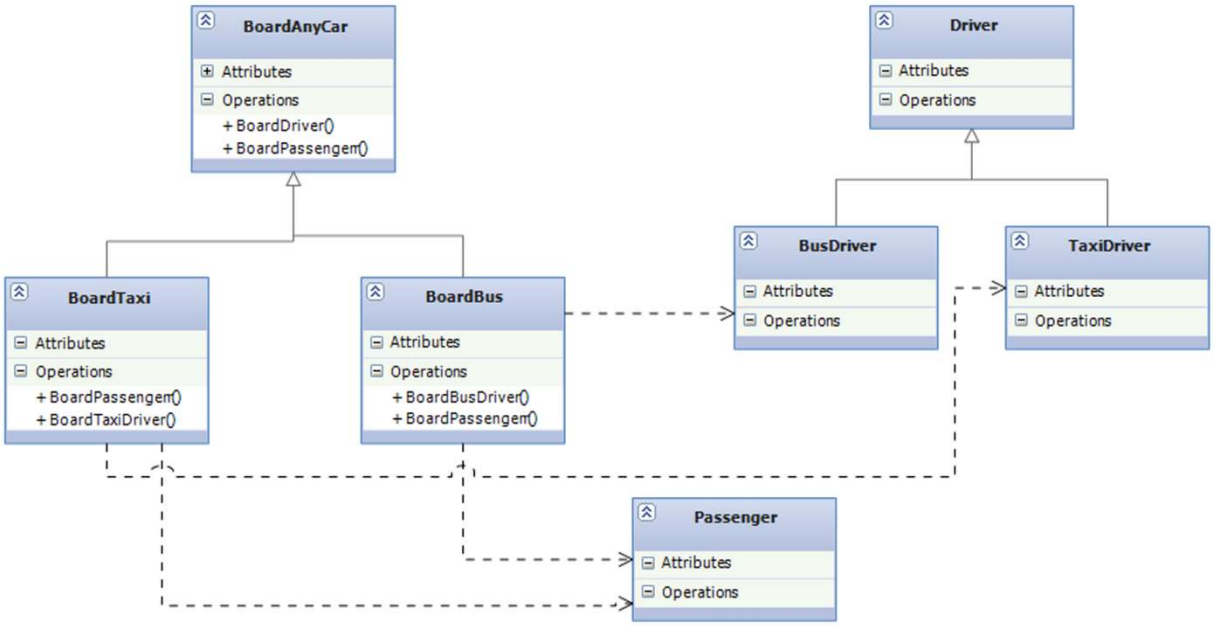
В случае использования нескольких взаимозависимых одиночек их реализация может резко усложниться.

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.



Лабораторная работа № 2
«Реализация одного из порождающих паттерны проектирования»

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

Содержание