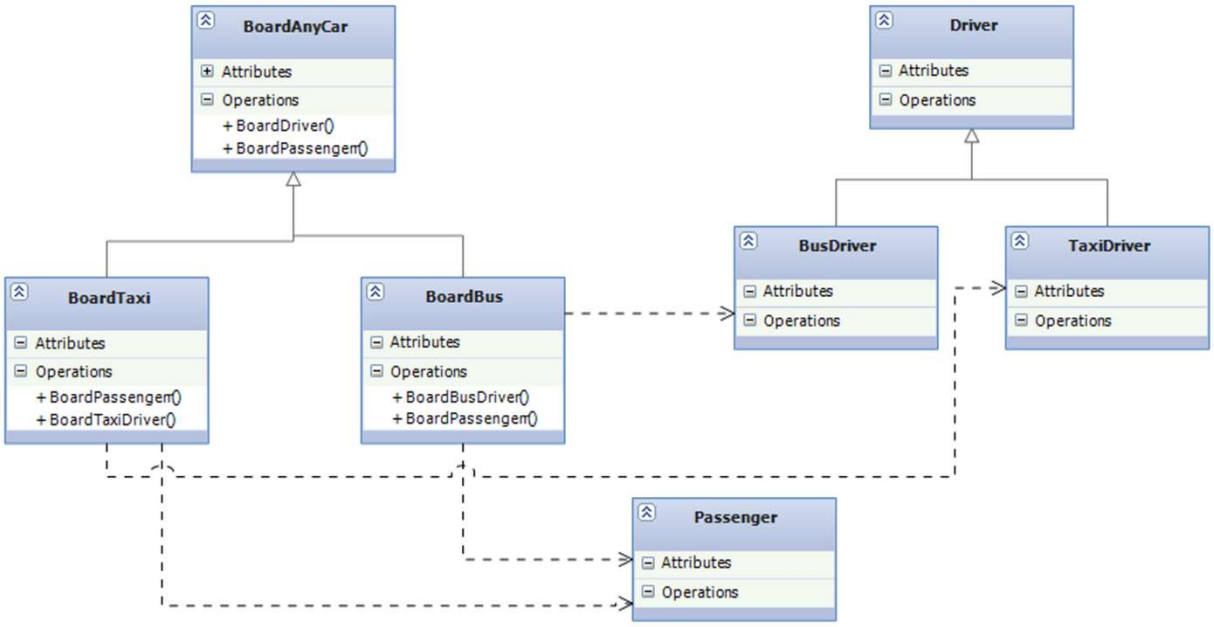
В случае использования нескольких взаимозависимых одиночек их реализация может резко усложниться.

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

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

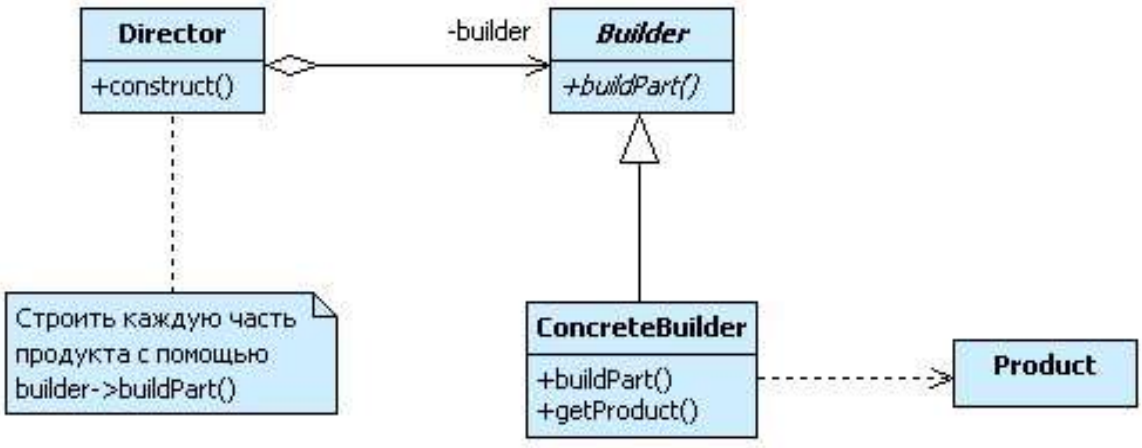


Рисунок 2. UML-диаграмма классов паттерна Builder

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.

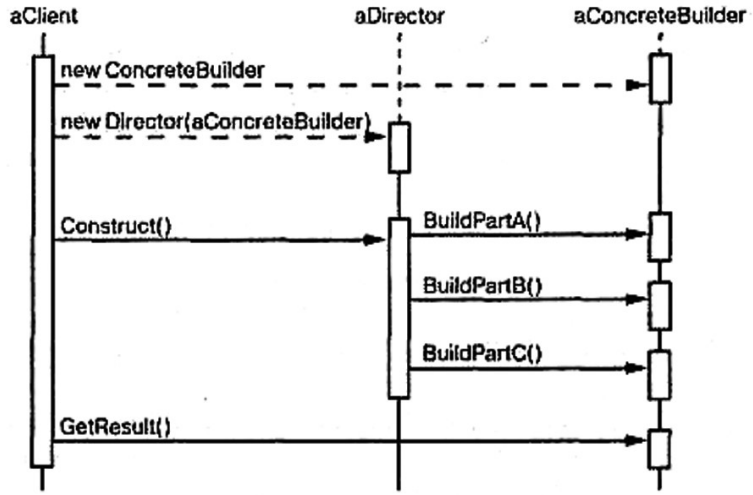


Рисунок 3. UML-диаграмма последовательностей паттерна Builder

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>

// К л а с с ы  в с е х  в о з м о ж н ы х  р о д о в  в о й с к
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».