ГОУВПО

ДОНЕЦКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

МЕТОДИЧЕСКИЕ УКАЗАНИЯ И ЗАДАНИЯ

к индивидуальной работе по дисциплине

«Конструирование программного обеспечения»

(для студентов направления подготовки

09.03.04 “Программная инженерия”)

ДОНЕЦК-ДОННТУ-2016

ГОУВПО

ДОНЕЦКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

МЕТОДИЧЕСКИЕ УКАЗАНИЯ И ЗАДАНИЯ
к индивидуальной работе по дисциплине

«Конструирование программного обеспечения»

(для студентов направления подготовки 09.03.04 “Программная инженерия”)

Рассмотрено на заседании кафедры

программной инженерии

Протокол № 1 от 30.08.2016

Утверждено на заседании

учебно-издательского Совета ДонНТУ

протокол № от

ДОНЕЦК –2016

УДК 681.3

Методические указания и задания к индивидуальной работе по дисциплине «Конструирование программного обеспечения» для студентов направления подготовки 09.03.04 «Программная инженерия», Сост.: Чернышова А.В., Донецк, ДонНТУ, 2016 - 85 стр.

Приведены методические указания и задания к выполнению индивидуальной работы по дисциплине «Конструирование программного обеспечения» для студентов направления подготовки 09.03.04 «Программная инженерия». Излагаются вопросы, связанные с изучением паттернов проектирования, в частности, порождающие паттерны (Abstract Factory (Абстрактная Фабрика), Builder (Строитель), Factory Method (Фабричный Метод), Prototype (Прототип), Singleton (Одиночка)). Студентам предлагается изучить вопросы, связанные с паттернами проектирования и использовать на практике полученные знания в ходе процесса проектирования программного продукта. Также в рамках индивидуальной работы студентам предлагается самостоятельно изучить назначение и виды поведенческих паттернов проектирования.

Методические указания предназначены для усвоения теоретических основ и формирования практических навыков по курсу «Конструирование программного обеспечения» по разделу паттерны проектирования (порождающие паттерны, поведенческие паттерны).

Составители: ст. преп. каф. ПИ Чернышова А.В.

Методические указания к индивидуальной работе

Понятие паттерна проектирования

Технически, паттерны (шаблоны) проектирования - это всего лишь абстрактные примеры правильного использования небольшого числа комбинаций простейших техник ООП. Паттерны проектирования - это простые примеры, показывающие правильные способы организации взаимодействий между классами или объектами.

Паттерны (шаблоны) проектирования – это 23 примера, которые описывают:

- правильные способы формирования внутреннего состояния (полей) и поведения (методов) объекта или класса;
- правильные способы создания объекта (через вызов конструктора или другим способом);
- правильные способы объединения объектов в группы;
- правильные способы организации информационных потоков (вызовов методов и очередности вызовов) позволяющих наладить гармоничное взаимодействие между объектами и группами этих объектов в объектно-ориентированных системах.

Паттерны проектирования помогают представить объектно-ориентированную систему формально, отображая результаты мышления проектировщика в комбинациях двадцати трех точных понятий и утверждений. Например, каталог паттернов можно было бы представить универсальной алгеброй, а каждый отдельный паттерн – конгруэнцией этой алгебры. Но традиционно принято использовать визуальный формализм представления паттернов с применением диаграмм классов и диаграмм последовательностей языка UML.

Для того, чтобы лучше понять, что такое паттерны, предлагается воспользоваться метафорой и провести проекцию элементов объективной реальности на виртуальную, то есть на мир программирования. Проекция модели реального мира на модель программной системы - это та проекция, которая послужила толчком к развитию объектно-ориентированного программирования.

Рассмотрим в качестве метафоры фигурное катание. В фигурном катании можно выделить четыре основных категории базовых элементов: *шаги*, *спирали*, *вращения* и *прыжки*.

\- *Шаги:*

Основной шаг, Шассе, Кроссролл, Подсечка, Беговой шаг, Моухог, Чоктау и др.

\- *Спирали:*

Спираль в арабеске, спираль Шарлотты, Y-спираль, Fan-спираль, Спираль-Керриган и др.

- *Вращения:*

Вращение стоя назад, Вращение-заклон, Вращение бильман, Вращение в ласточке (либела) и др.

- *Прыжки:*

Лутц, Тулуп, Флип, Аксель, Риттбергер, Сальхов, Сальто и др.

Базовые элементы фигурного катания, можно назвать шаблонами или паттернами фигурного катания. В итоге полный танец состоит из комбинации базовых элементов в определенной последовательности. И все танцы в фигурном катании отличаются набором и последовательностью исполнения базовых элементов (шаблонов).

Формат описания паттернов проектирования

При рассмотрении паттернов проектирования используется единый формат описания. Описание каждого шаблона состоит из следующих разделов:

Название

Название паттерна (на Русском языке) отражающее его назначение.

Также известен как

Альтернативное название паттерна (если такое название имеется).

Классификация

Классификация паттернов производится:

- По цели (порождающий, структурный или поведенческий)
- По применимости (к объектам и/или к классам)

Частота использования

Низкая - **1** 2 3 4 5

Ниже средней - 1 **2** 3 4 5

Средняя - 1 2 **3** 4 5

Выше средней - 1 2 3 **4** 5

Высокая - 1 2 3 4 **5**

Назначение

Краткое описание назначения паттерна и задачи проектирования, решаемые с его использованием.

Введение

Описание паттерна с использованием метафор, позволяющих лучше понять идею, лежащую в основе паттерна, в общем виде охарактеризовать специфические аспекты использования паттерна проводя ассоциации с другими знакомыми процессами, для формирования ясного представления механизма работы паттерна.

Структура паттерна на языке UML

Графическое представление паттерна с использованием диаграмм классов языка UML. На диаграммах показаны основные участники (классы) и связи отношений между участниками.

Структура паттерна на языке C#

Программная реализация паттерна с использованием языка C#.

Участники

Имена участников (классы которые входят в состав паттерна) и описание их назначения.

Отношения между участниками

Описание отношений (взаимодействий) между участниками (классами и/или объектами).

Мотивация

Определение потребности в использовании паттерна. Рассмотрение способов применения паттерна.

Применимость паттерна

Рекомендации по применению паттерна.

Результаты

Особенности и варианты использования паттерна. Результаты применения.

Реализация

Описание вариантов и способов реализации паттерна.

Пример кода

Дополнительные примеры, иллюстрирующие использование паттерна.

Известные применения паттерна в .Net

Использование паттерна в .Net Framework и/или его выражение в языке C#.

Каталог паттернов проектирования

Каталог состоит из 23 паттернов. Все паттерны разделены на три группы:

Порождающие

1. Abstract Factory (Абстрактная Фабрика)
2. Builder (Строитель)
3. Factory Method (Фабричный Метод)
4. Prototype (Прототип)
5. Singleton (Одиночка)

Структурные

1. Adapter (Адаптер)
2. Bridge (Мост)
3. Composite (Компоновщик)
4. Decorator (Декоратор)
5. Facade (Фасад)
6. Flyweight (Приспособленец)

7. Proxy (Заместитель)

Поведенческие

1. Chain of Responsibility (Цепочка Обязанностей)
2. Command (Команда)
3. Interpreter (Интерпретатор)
4. Iterator (Итератор)
5. Mediator (Посредник)
6. Memento (Хранитель)
7. Observer (Наблюдатель)
8. State (Состояние)
9. Strategy (Стратегия)
10. Template Method (Шаблонный Метод)
11. Visitor (Посетитель)

Техники ООП

В основу категоризации каталога паттернов легли три простейшие объектно-ориентированные техники. Это техника использования объектов-фабрик, порождающих объекты-продукты, техника использования объекта-фасада и техника диспетчеризации.

Фабрика - Продукт

Техника использования объекта-фабрики для порождения объектов-продуктов, была положена в основу всех порождающих паттернов. Методы, принадлежащие объекту-фабрике, которые порождают и возвращают объекты-продукты, принято называть фабричными-методами (или виртуальными конструкторами).

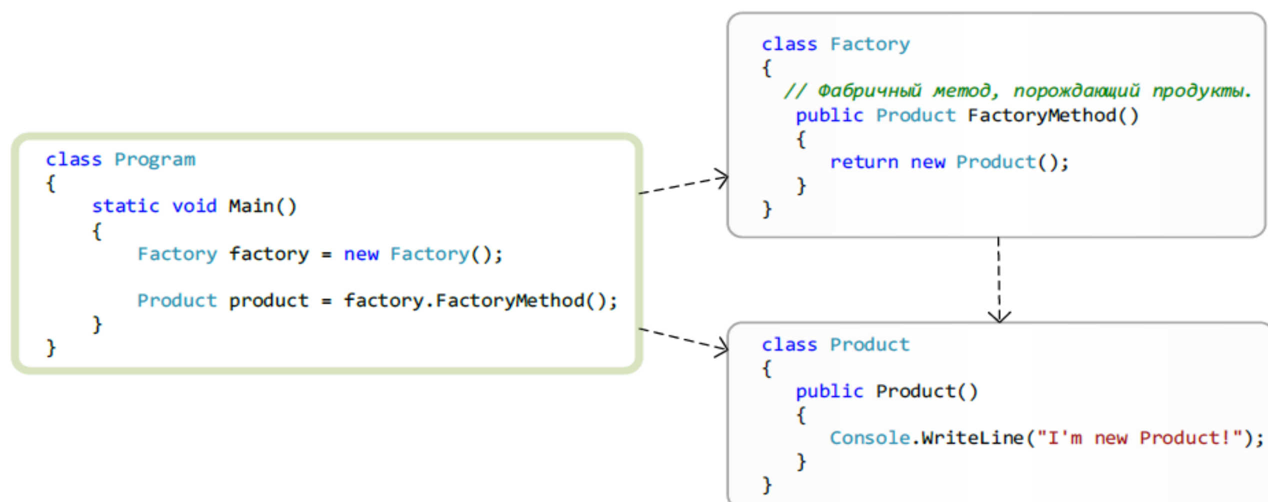


Рисунок 1 – Фабрика - Продукт

На диаграмме последовательностей можно отследить работу фабричной техники.

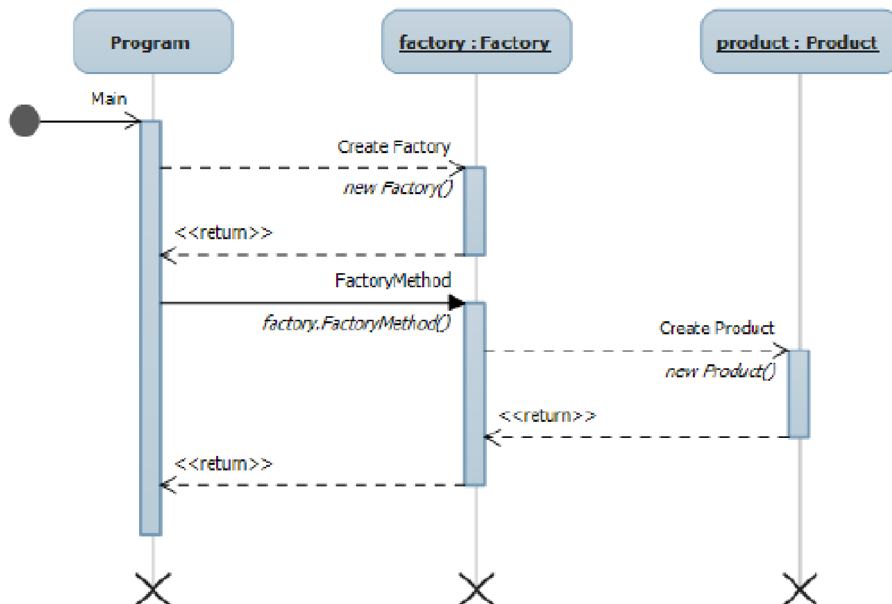


Рисунок 2 - Диаграмма последовательностей фабричной техники

Фасад – Подсистема

<u>Техника использования объекта-фасада, скрывающего за собой работу с неким подмножеством объектов, легла в основу структурных паттернов.</u>

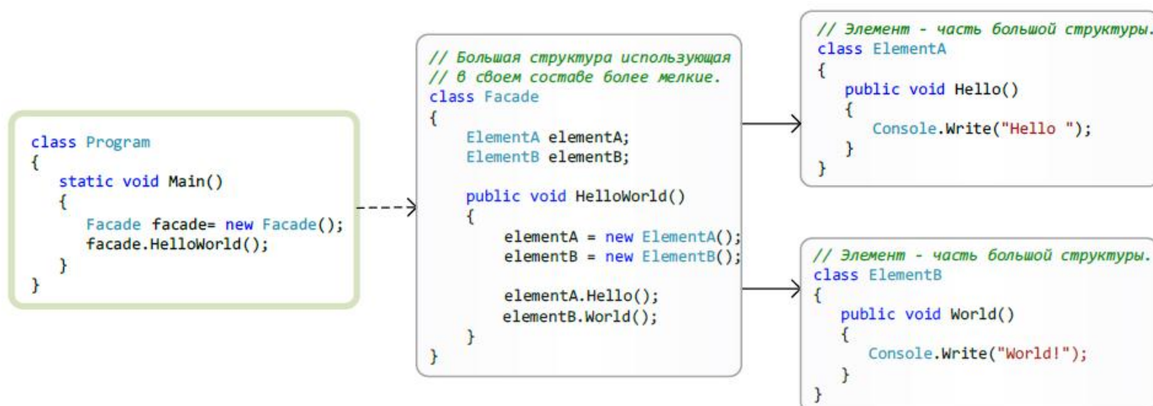


Рисунок 3 – Пример фасадной техники

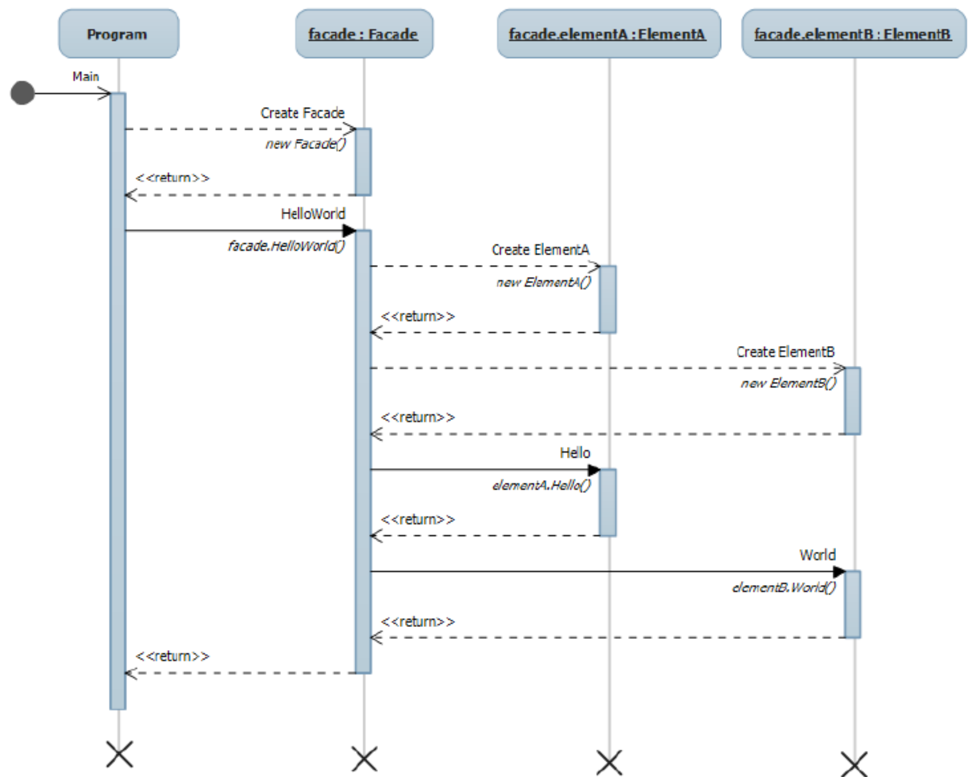


Рисунок 4 - Диаграмма последовательностей фасадной техники

Диспетчеризация

Техника использования диспетчеризации имеет две формы: «Цепочка объектов» и «Издатель- Подписчик». Эти техники были положены в основу поведенческих паттернов.

Цепочка объектов

При использовании техники «Цепочка объектов» - объекты связываются в цепочку, вдоль которой происходит серия вызовов методов (посылка сообщений).

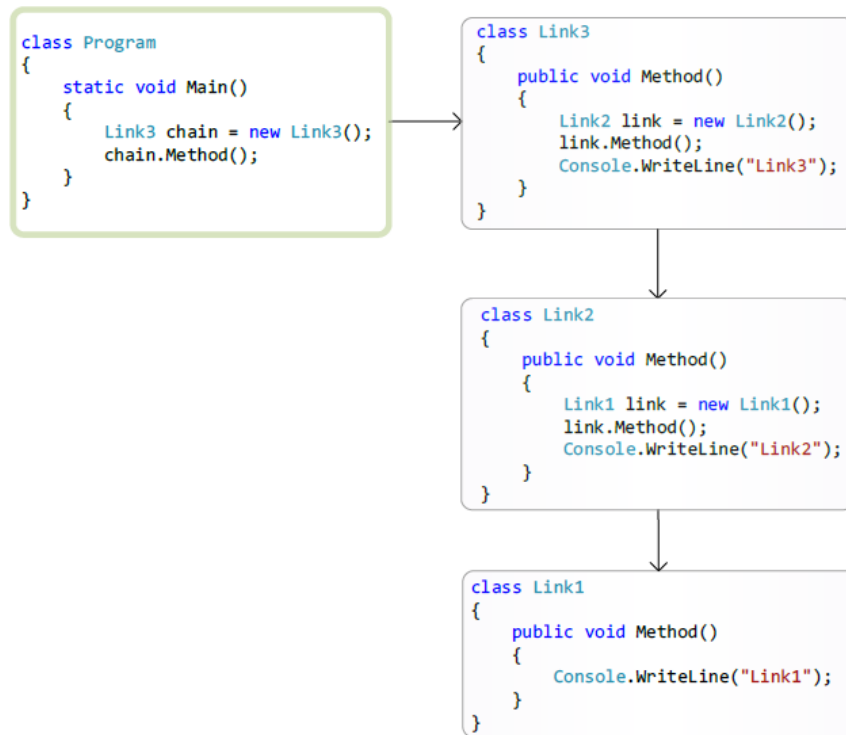


Рисунок 5 - техника «Цепочка объектов»

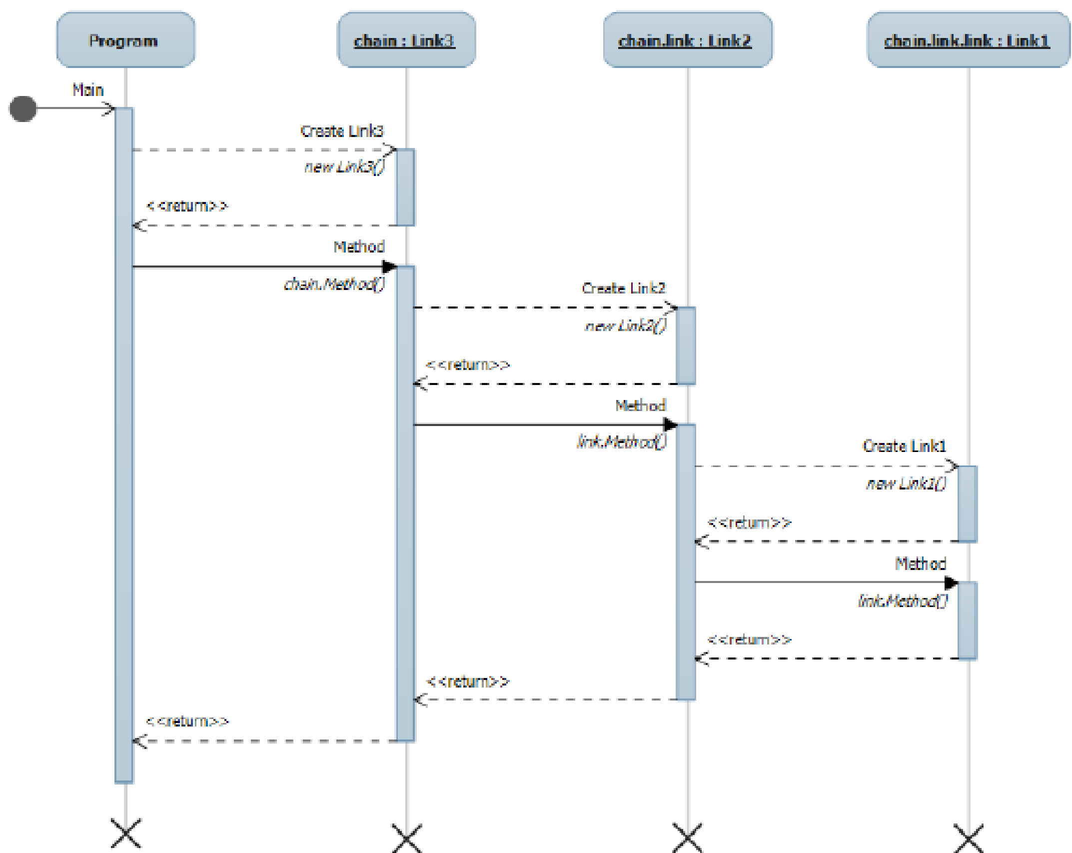


Рисунок 6 - Диаграмма последовательностей техники "Цепочка объектов».

Издатель-Подписчик

При использовании техники «Издатель-Подписчик» - объект-издатель вызывает метод на объекте- подписчике, а объект-подписчик после этого вызывает метод на объекте-издателе. Таким образом объект-издатель уведомляет объекта-подписчика о наступлении некоторого события.

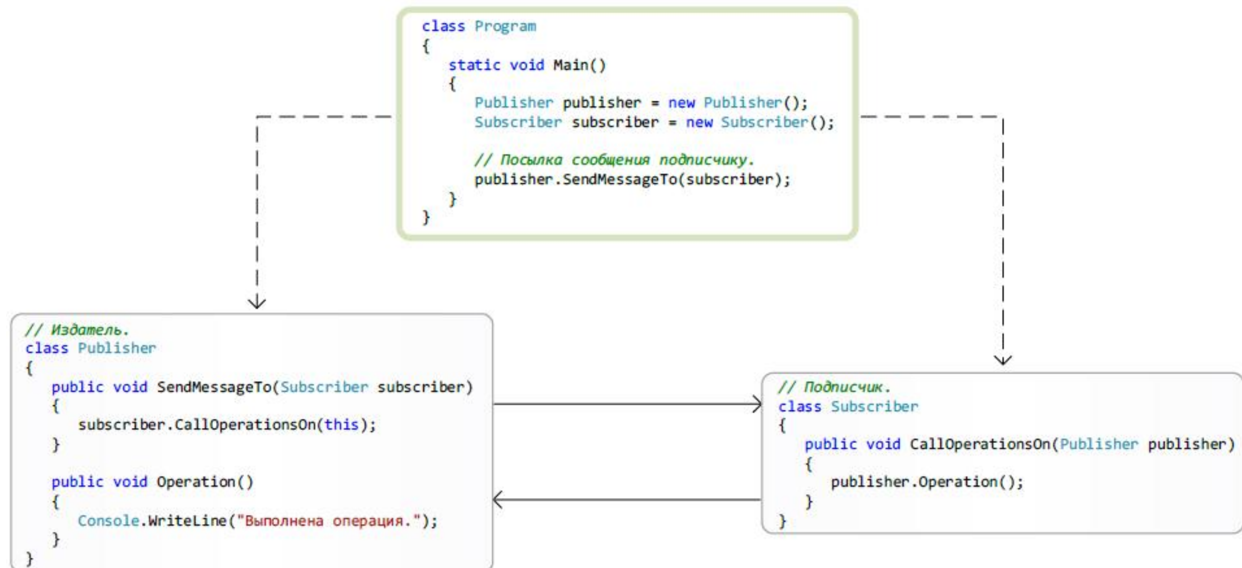


Рисунок 7 – Техника Издатель-Подписчик

На диаграмме последовательностей можно отследить работу техники «Издатель-Подписчик».

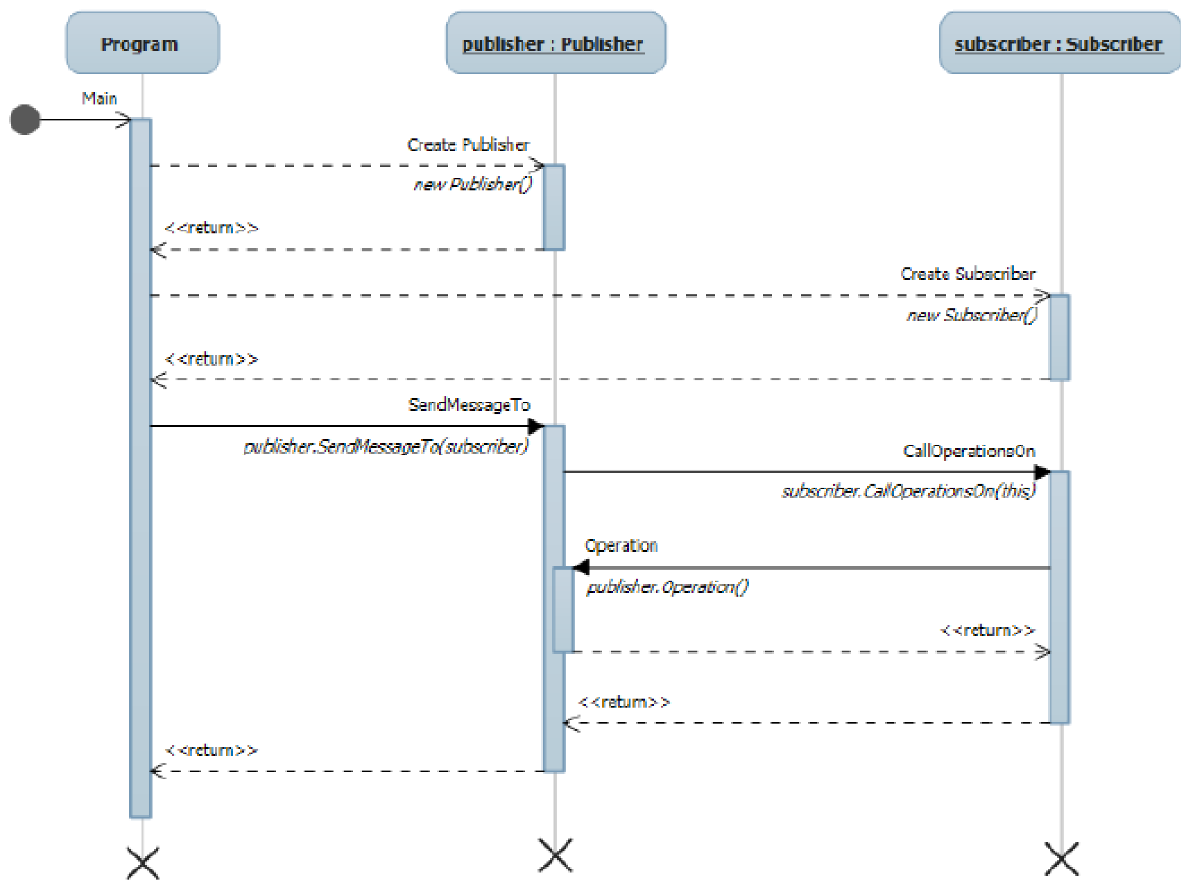


Рисунок 8 - Диаграмма последовательностей техники «Издатель-Подписчик»

Принципы организации каталога

Все 23 паттерна классифицируются по двум критериям – цель и применимость (уровень).

Применимость (уровень)	Цель		
	Порождающие	Структурные	Поведенческие
К классам	Фабричный метод	Адаптер (класса)	Интерпретатор Шаблонный метод
К объектам	Абстрактная фабрика Одиночка Прототип Строитель	Адаптер (объекта) Декоратор Заместитель Компоновщик Мост Приспособленец Фасад	Итератор Команда Наблюдатель Посетитель Посредник Состояние Стратегия Хранитель Цепочка обязанностей

Рисунок 9 – Принципы организации каталога паттернов

Цель паттерна

<u>Цель паттерна – показывает его назначение.</u>

<u>Целью порождающих паттернов, является организация процесса создания объектов.</u>

<u>Целью структурных паттернов, является составление правильно организованных структур из объектов и классов.</u>

<u>Целью поведенческих паттернов, является организация устойчивого (робастного) взаимодействия между классами или объектами, через правильное формирование информационных потоков.</u>

Уровень паттерна

<u>Уровень паттерна - показывает область применения паттерна: к классам или к объектам.</u>

Паттерны уровня классов описывают отношения между классами и их подклассами. Такие отношения выражаются при помощи статических связей отношений – наследования и реализации.

Паттерны уровня объектов описывают взаимодействия между объектами. Такие отношения выражаются при помощи динамических связей отношений – ассоциации, агрегации и композиции.

Рекомендации по изучению паттернов

Существует две категории программистов, которые решили приступить к изучению паттернов проектирования. Первая категория – начинающие программисты или программисты с небольшим опытом разработки, которые только слышали о таком понятии как паттерны и об их полезности от старших и более опытных коллег. Вторая категория – программисты с опытом разработки, с хорошим пониманием ООП, но по ряду причин не применявшие в своей практике паттерны, при этом осознающие полезность их использования.

Начинающим программистам понадобится немного больше времени для изучения и хорошего понимания паттернов. Им рекомендуется сначала приступить к ознакомлению с диаграммой классов и сравнению классов и связей отношений, изображенных на диаграмме с реализацией паттерна в примере на языке C#. Такой подход позволит закрепить понимание простейших техник ООП, которые используются при построении паттерна. Для достижения большего эффекта есть смысл при изучении кода паттерна параллельно рисовать диаграмму объектов (овалы - символизирующие объекты в памяти и стрелки – показывающие как одни объекты ссылаются на другие).

Начать рассмотрение паттерна лучше всего с тела метода Main, сперва ознакомившись с интерфейсом взаимодействия используемых объектов. Далее есть смысл переходить к знакомству с классами используемых объектов.

Требуется понять все объектно-ориентированные техники, используемые в коде, мысленно выстроить схему паттерна, а именно запомнить основных участников и связи отношений между ними, а также осознать объектную модель паттерна. И только после этого есть смысл перейти к чтению главы описывающей паттерн и рассмотрению примеров его использования. Таким образом с пониманием абстрактной техники построения паттерна начнет ассоциироваться смысл примеров использования этого паттерна.

Важно понимать, что паттерн - это формула, а пример использования паттерна - это пример применения этой формулы. Формула – первична, ее применение – вторично. Для создателей каталога паттернов формула была вторична. В основу этого каталога была положена докторская диссертация Эриха Гаммы – а это значит сначала исследования в области построения объектно-ориентированных систем, затем формализация результатов

исследований и представление их в виде 23 паттернов (формул). Разработчикам исследовать ничего не нужно, им не нужно порождать новых знаний и делать открытий, им просто требуется использовать готовые паттерны (формулы) в повседневной работе для решения проектных задач.

Рекомендации по применению паттернов

Использовать паттерны просто для тех, кто знает наизусть все 23 паттерна (всех участников и связи отношений между ними). 23 паттерна – это «таблица умножения» проектировщика. Как трудно производить расчеты без знания таблицы умножения, также трудно проектировать приложения без знания паттернов.

Нет надобности искать в каком месте и когда применить тот или иной паттерн. Выбор паттерна – это выбор способа решения задачи. Нет задачи – нет и решения. Поставленная задача – причина. Паттерн - путь к следствию. Решенная задача – следствие.

Примеры, приводимые в книге приближены к несложным проектным требованиям и их рассмотрение окажется полезным для понимания использования паттернов. Важно понимать, что пример - это образец чего-либо, как правило – самый яркий и лучший образец (пример для подражания).

Умение увязывать между собой абстрактные примеры и реальные системы или их части, то есть применять знания, полученные во время обучения, к решению проектных задач — есть признак профессионализма. Неумение делать это — основное свойство неопытности. Поиск аналогий — есть перенесение опыта из одной ситуации в другую. Начинающим специалистам не всегда быстро удается перенести опыт из одной проектной ситуации в другую. В этом нет ничего страшного. Потратив определенное количество времени на обучение и рассмотрение примеров, вполне возможно достигнуть желаемых результатов.

Порождающие паттерны

Порождающие паттерны проектирования, не просто описывают процесс создания объектов- продуктов определенных классов, они при этом делают абстрактным сам процесс создания (другими словами абстрагируют процесс инстанцирования). Абстрагирование процесса инстанцирования в ООП, это еще одна простая техника, которая часто используется при порождении объектов-продуктов.

Понятие абстрагирования процесса инстанцирования и есть ничто иное, как абстрактное описание процесса порождения экземпляра типа, через использование абстрактных фабричных методов.

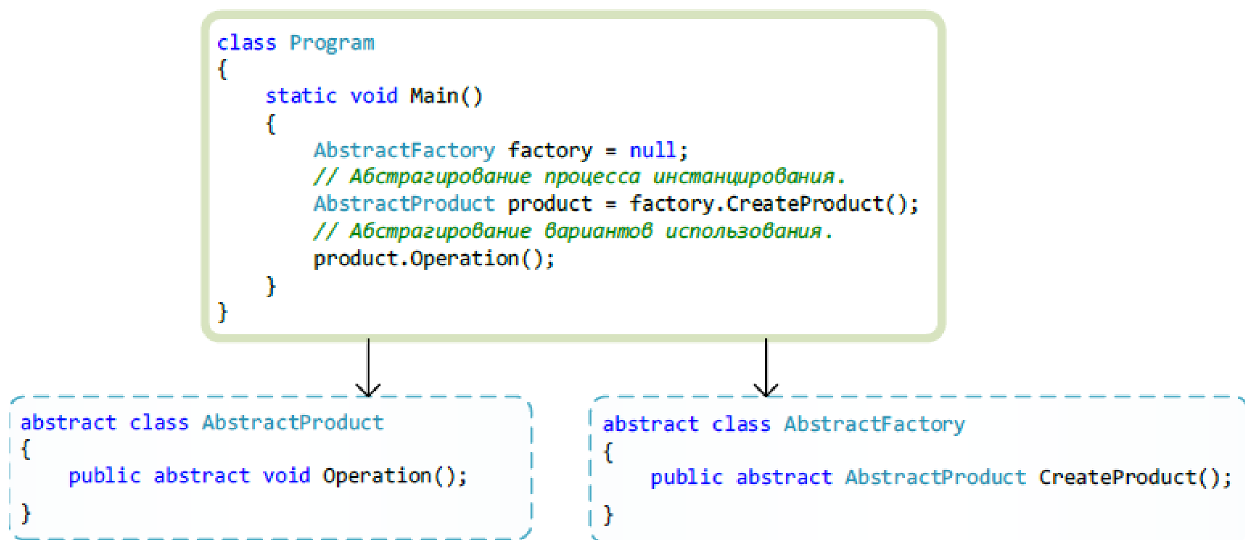


Рисунок 10 - Абстрагирование процесса инстанцирования

Сперва создаются абстрактные классы. С их помощью легче выявить и задать нужные типы, выразить типы как собирательные понятия других типов. Описать интерфейсы взаимодействия с каждым типом. Абстрактно (без реализации) описать процессы порождения экземпляров этих типов и варианты использования этих типов, через имеющиеся у них интерфейсы.

После того, как заданы нужные типы, описаны интерфейсы взаимодействия, абстрагированы процессы инстанцирования и варианты использования, можно приступить к выбору структур данных и алгоритмов для их обработки, а также подойти к выбору деталей реализации конкретных классов.

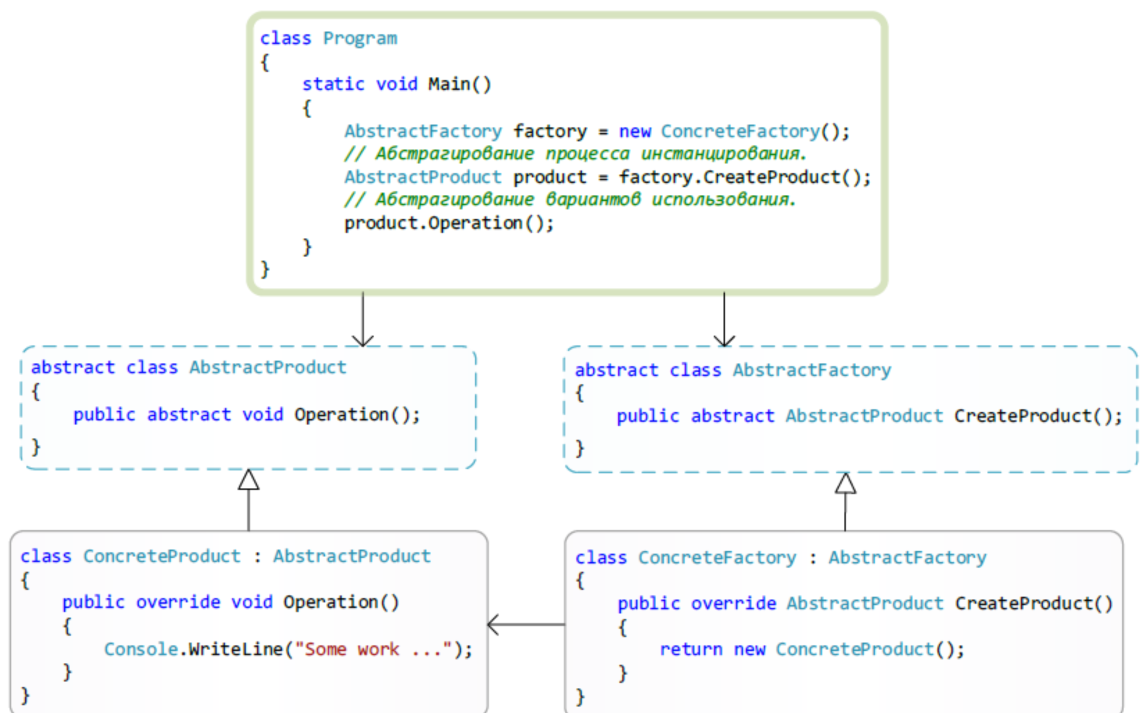


Рисунок 11 – Абстрагирование (Шаг 2 выбор структур данных и алгоритм их обработки)

Объектно-ориентированная программная система (программа-целое, составленная из соединенных объектов-частей) – состоит из множества объектов, находящихся в определенных отношениях и связях друг с другом. Связанные между собой объекты образуют логическую целостность (единство) системы. Порождающие паттерны проектирования используя технику абстрагирования процесса инстанцирования, помогают построить программную систему так, что эта система не зависит от способа создания в ней объектов, композиции (составления и соединения) объектов и представления (внутреннего состояния) объектов.

Порождающий паттерн уровня классов, использует наследование, чтобы варьировать (видоизменять) инстанцируемый класс, а порождающий паттерн уровня объектов, использует композицию делегируя (передавая ответственность) инстанцирование другому объекту.

Использование порождающих паттернов оказываются полезным, когда в программной системе чаще используется композиция объектов чем наследование классов. При использовании наследования основной акцент делается на жестком кодировании фиксированного набора поведений (методов). В случае использования композиции можно получать любое число новых поведений (методов) составленных из уже существующих поведений.

Порождающие паттерны позволяют скрыть работу с конкретными классами и детали того как эти классы создаются и стыкуются. Единственная известная информация об объектах - это интерфейсы этих объектов, заданные через абстрактные классы. Таким образом, упрощается понимание процессов создания объектов: когда, как, кто и что создает.

Рассмотрим пример. Игра – Лабиринт

Изучение всех пяти порождающих паттернов, будет сопровождаться примерами построения лабиринта для компьютерной игры. Реализация игры-лабиринта будет изменяться при применении разных паттернов. Для того что бы упростить представление использования паттернов в игре-лабиринте, будут игнорироваться все соображения относительно взаимодействия игры с пользовательским интерфейсом.

В реализации данной игры внутри лабиринта сможет находиться только один игрок. Лабиринт представляет собой множество комнат. Каждая комната лабиринта является ассоциативным объектом класса Room, который содержит в себе ссылки на составляющие ее элементы - стены класса Wall или двери в другую комнату класса Door.

Комната имеет четыре стороны. Для задания северной, южной, восточной и западной сторон используются элементы перечисления Direction.

```
enum Direction
{
    North,
    South,
    East,
    West
}
```

Абстрактный класс MapSite - является базовым классом для всех классов компонентов лабиринта (Room, Door и Wall). В классе MapSite создается абстрактная операция Enter.

```
public abstract class MapSite
{
    public abstract void Enter();
}
```

Реализованные в производных классах операции Enter позволят игроку перемещаться из комнаты в комнату или оставаться в той же комнате. Например, если игрок находится в комнате и говорит «Иду на восток», то игрой определяется, какой объект типа MapSite находится к востоку от игрока, и на этом объекте вызывается операция Enter. Если на востоке был объект-дверь (Door) то игрок перейдет в другую комнату, а если был объект-стена (Wall), то останется в этой же комнате.

Класс Room - производный от класса MapSite, содержит ссылки на другие объекты ти типа MapSite, а также хранит номер комнаты. Каждая из комнат в лабиринте имеет свой уникальный номер.

```
class Room : MapSite
{
    int roomNumber = 0;
    Dictionary<Direction, MapSite> sides;

    public Room(int roomNo)
    {
        this.roomNumber = roomNo;
        sides = new Dictionary<Direction, MapSite>(4);
    }

    public override void Enter()
    {
        Console.WriteLine("Room");
    }

    public MapSite GetSide(Direction direction)
    {
        return sides[direction];
    }

    public void SetSide(Direction direction, MapSite mapSide)
    {
        this.sides.Add(direction, mapSide);
    }

    public int RoomNumber
    {
        get { return roomNumber; }
        set { roomNumber = value; }
    }
}
```

COMODO Antivirus - Мастер

Классы Wall и Door описывают стены и двери, из которых состоит комната.

```
class Wall : MapSite
{
    public Wall()
    {
    }

    public override void Enter()
    {
        Console.WriteLine("Wall");
    }
}
```

```
class Door : MapSite
{
    Room room1 = null;
    Room room2 = null;
    bool isOpen;


    public Door(Room room1, Room room2)
    {
        this.room1 = room1;
        this.room2 = room2;
    }

    public override void Enter()
    {
        Console.WriteLine("Door");
    }


    public Room OtherSideFrom(Room room)
    {
        if (room == room1)
            return room2;
        else
            return room1;
    }
}
```

Класс Maze используется для представления лабиринта, как набора комнат. В классе Maze имеется операция RoomNo(int number) для получения ссылки на экземпляр комнаты по ее номеру.

```
class Maze
{
    Dictionary<int, Room> rooms = null;

    public Maze()
    {
        this.rooms = new Dictionary<int, Room>();
    }

    public void AddRoom(Room room)
    {
        rooms.Add(room.RoomNumber, room);
    }
    public Room RoomNo(int number)
    {
        return rooms[number];
    }
}
```

Класс MazeGame – содержит в себе метод CreateMaze, который создает лабиринт состоящий из двух комнат с одной дверью между ними и возвращает ссылку на экземпляр созданного лабиринта.

В методе CreateMaze жестко «закодирована» структура лабиринта. Для изменения структуры лабиринта, потребуется внести изменения в тело самого метода или полностью его заместить, что может стать причиной возникновения ошибок и не способствует повторному использованию.

```
class MazeGame
{
    public Maze CreateMaze()
    {
        Maze aMaze = new Maze();
        Room r1 = new Room(1);


        Room r2 = new Room(2);
        Door theDoor = new Door(r1, r2);

        aMaze.AddRoom(r1);
        aMaze.AddRoom(r2);

        r1.SetSide(Direction.North, new Wall());
        r1.SetSide(Direction.East, theDoor);
        r1.SetSide(Direction.South, new Wall());
        r1.SetSide(Direction.West, new Wall());

        r2.SetSide(Direction.North, new Wall());
        r2.SetSide(Direction.East, new Wall());
        r2.SetSide(Direction.South, new Wall());
        r2.SetSide(Direction.West, theDoor);

        return aMaze;
    }
}
```

На диаграмме ниже показаны отношения между классами, которые использовались при построении игры-лабиринта.

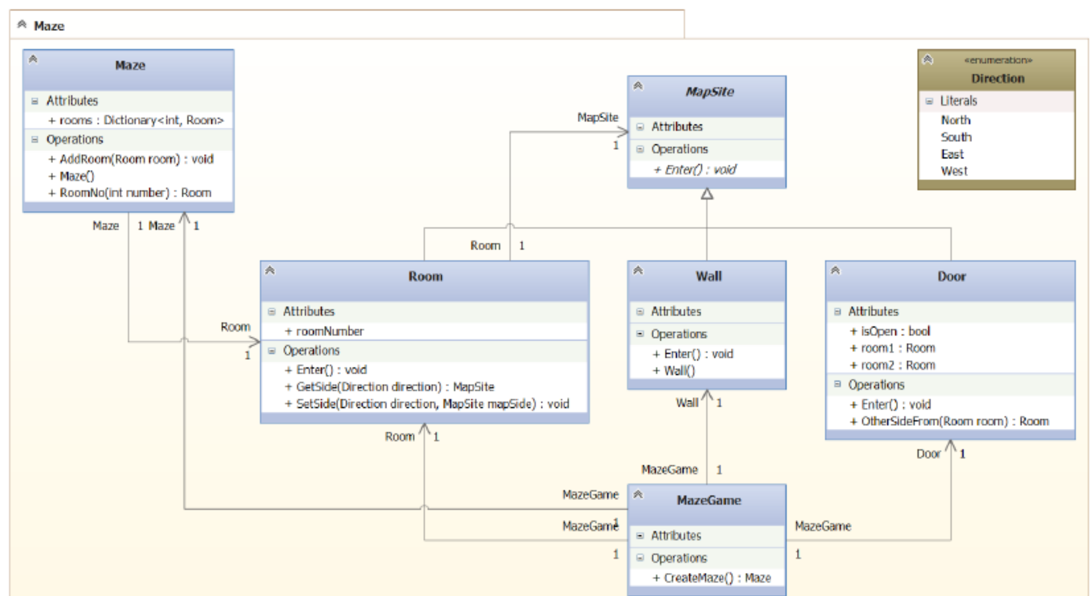


Рисунок 12 – Диаграмма классов для игры «Лабиринт»

Что можно сказать в общем о дизайне этой программы? Ее можно охарактеризовать как слабо спроектированную и не объектно-ориентированную. Хотя даже в таком виде эта программа работает. Но важно понимать, что в процессе разработки программ участвуют люди, которым будет сложно сопровождать и модифицировать такого вида программу.

Порождающие паттерны, которые будут использоваться в игре-лабиринте, помогут сделать дизайн программы более гибким, хотя и необязательно меньшим по размеру. Применение паттернов позволит легко изменять классы компонентов лабиринта.

Паттерн Abstract Factory

Название: Абстрактная фабрика

Также известен как: Kit (Набор инструментов)

Классификация

По цели: порождающий

По применимости: к объектам

Частота использования

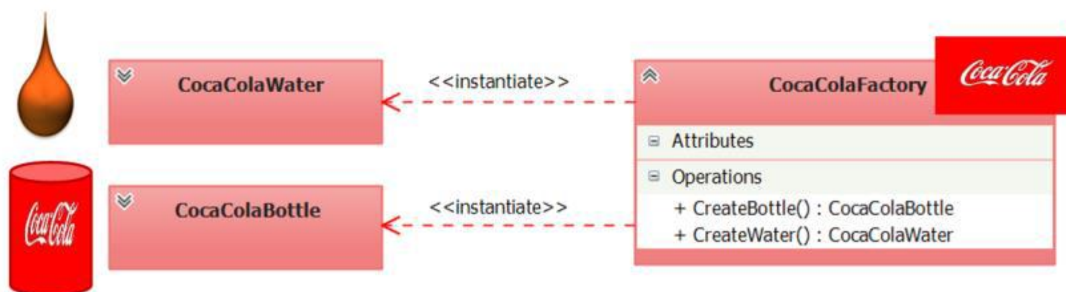
Высокая - 1 2 3 4 **5**

Назначение

Паттерн Abstract Factory - предоставляет клиенту интерфейс (набор методов) для создания семейств взаимосвязанных или взаимозависимых объектов-продуктов, при этом скрывает от клиента информацию о конкретных классах создаваемых объектов-продуктов.

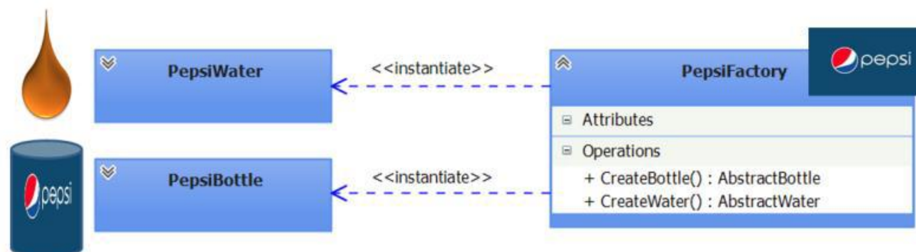
Введение

Что такое фабрика в объективной реальности? Фабрика – это объект имеющий станки (методы), производящие продукты. Например, фабрика компании Coca-Cola производит сладкую газированную воду, разлитую в жестяные банки. Предположим, что в помещении фабрики стоит два станка. Один станок размешивает и газирует сладкую воду, а другой станок формирует жестяные банки. После того как сладкая вода и жестяная банка произведены, требуется воду влить в банку, если сказать другими словами, то требуется организовать взаимодействие между двумя продуктами: водой и банкой. Опишем данный процесс с использованием диаграмм классов языка UML.



На диаграмме видно, что фабрика Coca-Cola порождает два продукта: воду и банку. Эти продукты должны обязательно взаимодействовать друг с другом.

Фабрика компании Pepsi также порождает свое собственное семейство взаимодействующих и взаимозависимых продуктов (вода и банка).



Важно заметить, что не логично пытаться наладить взаимодействие продуктов из разных семейств (например, вливать воду Coca-Cola в банку Pepsi или воду Pepsi в банку Coca-Cola). Скорее всего оба производителя будут против такого взаимодействия. Такой подход представляет собой пример антипатерна.

Представим рассмотренные фабрики и порождаемые ими семейства продуктов в контексте одной программы.

Сперва требуется создать абстрактные классы для задания типов продуктов (AbstractWater и AbstractBottle) и типа фабрик (AbstractFactory). Описать интерфейсы взаимодействия с каждым типом продукта и фабрики.

Далее требуется создать конкретный класс Client в котором абстрактно (без реализации) описать процессы порождения экземпляров типов продуктов и варианты использования этих типов продуктов, через имеющиеся у них абстрактные интерфейсы. Также класс Client реализует идею инкапсуляции вариаций (сокрытие частей программной системы).

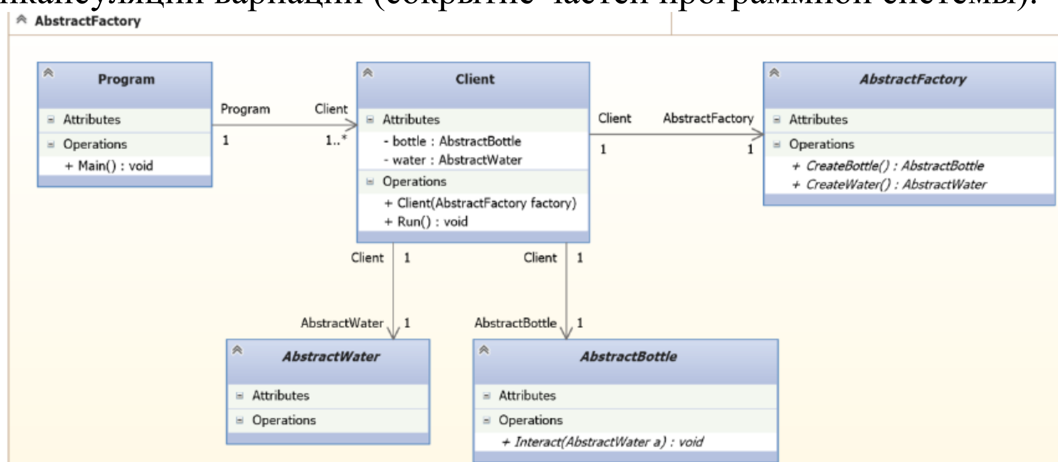


Рисунок 13 - Абстрактные классы

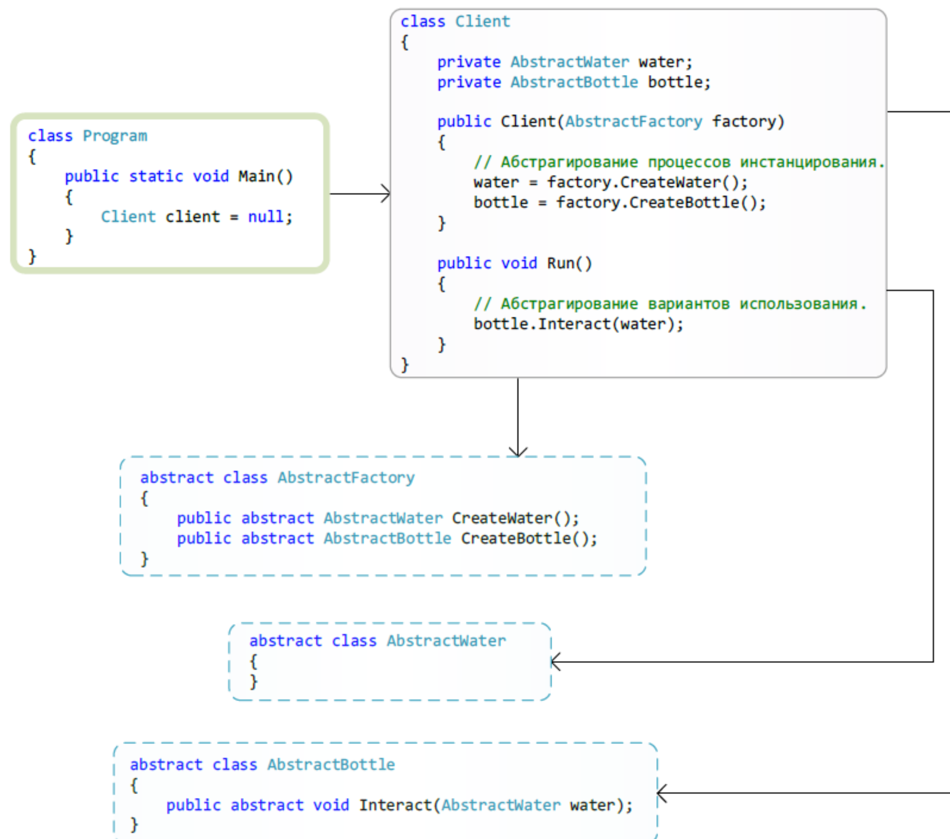


Рисунок 14 - Абстрактные классы

После того, как заданы нужные типы продуктов и фабрик, описаны интерфейсы взаимодействия между продуктами, абстрагированы процессы инстанцирования продуктов и варианты использования продуктов, можно приступить к реализации конкретных классов продуктов и фабрик.

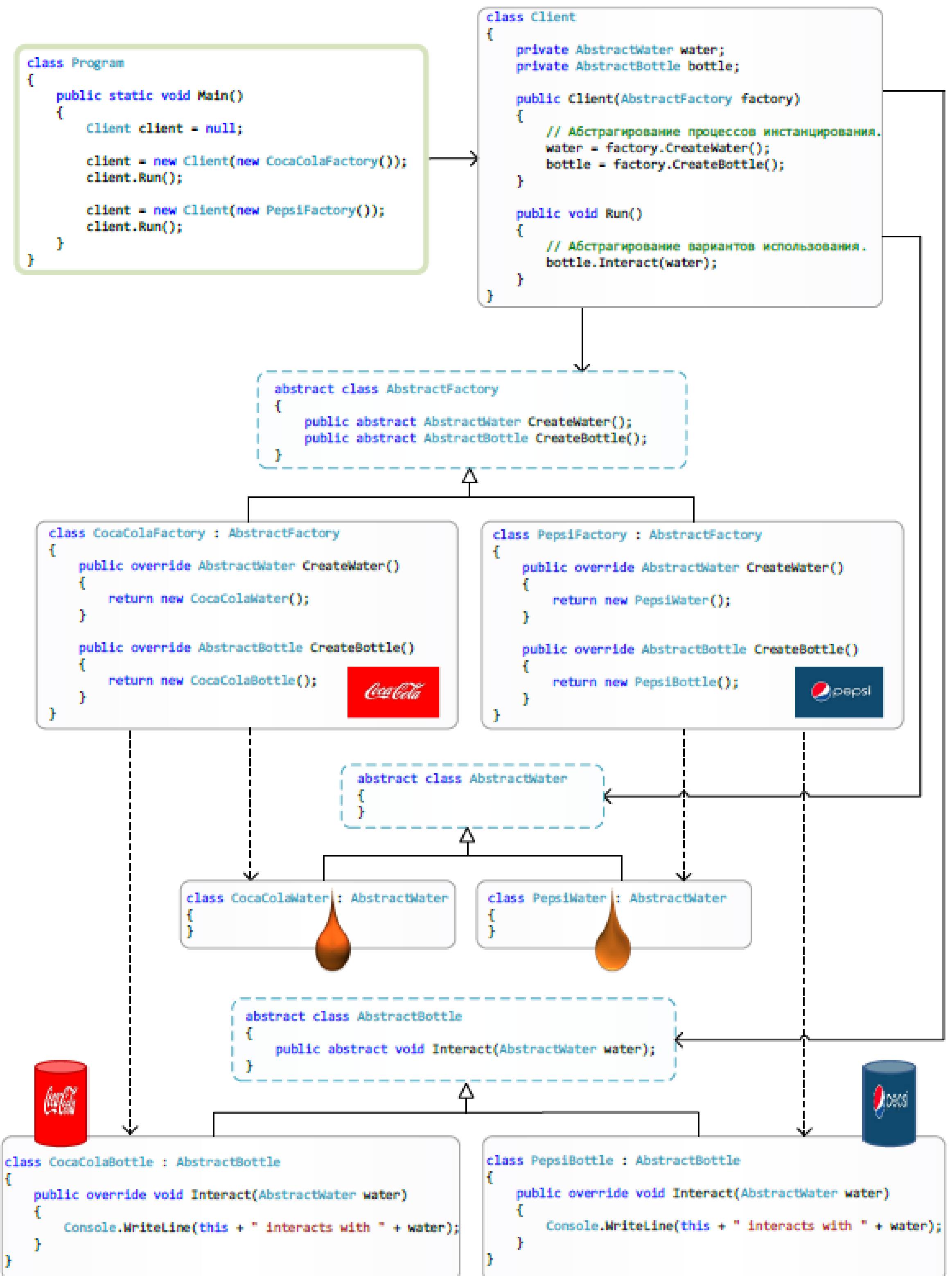


Рисунок 15 - Реализации конкретных классов продуктов и фабрик

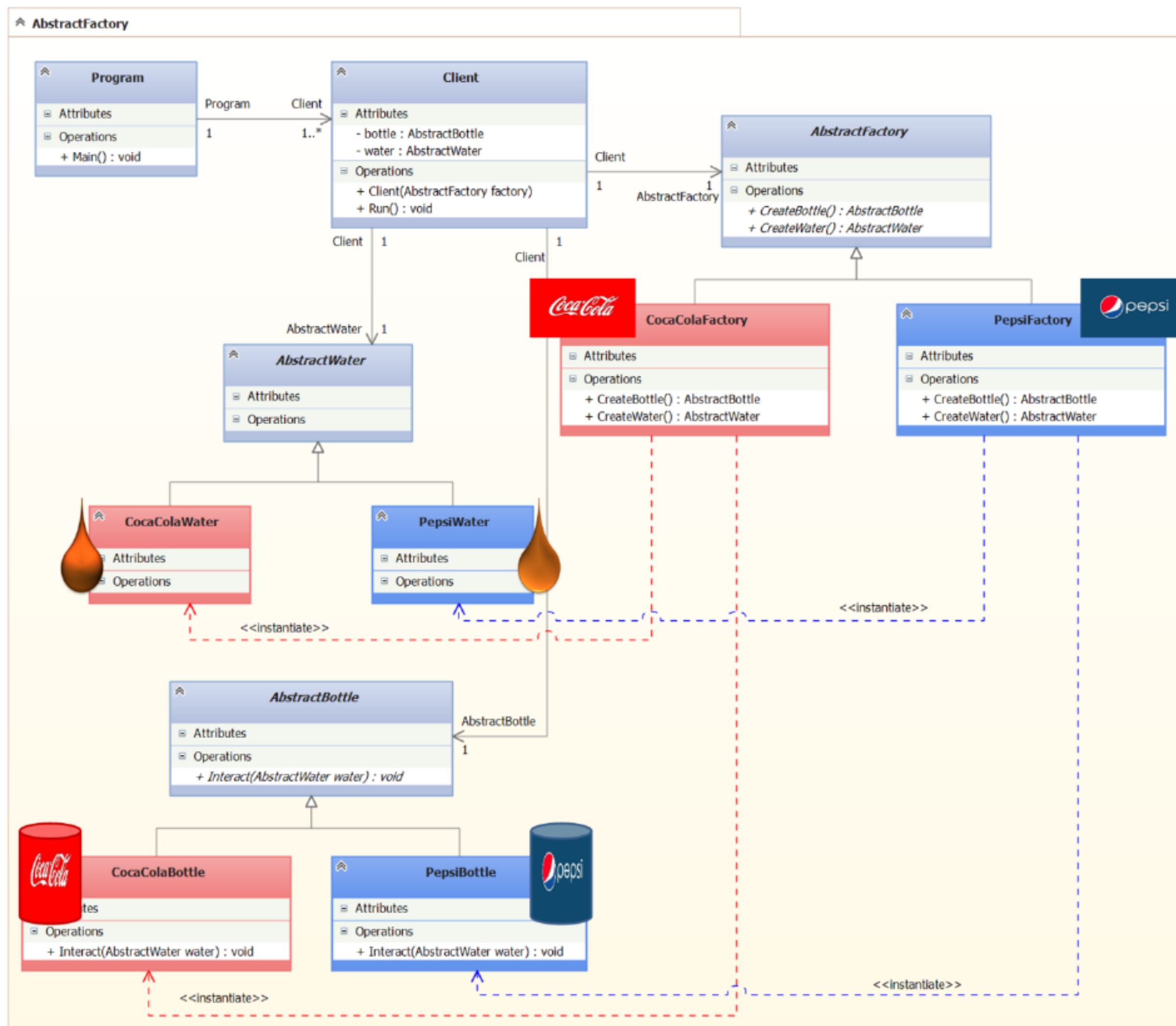
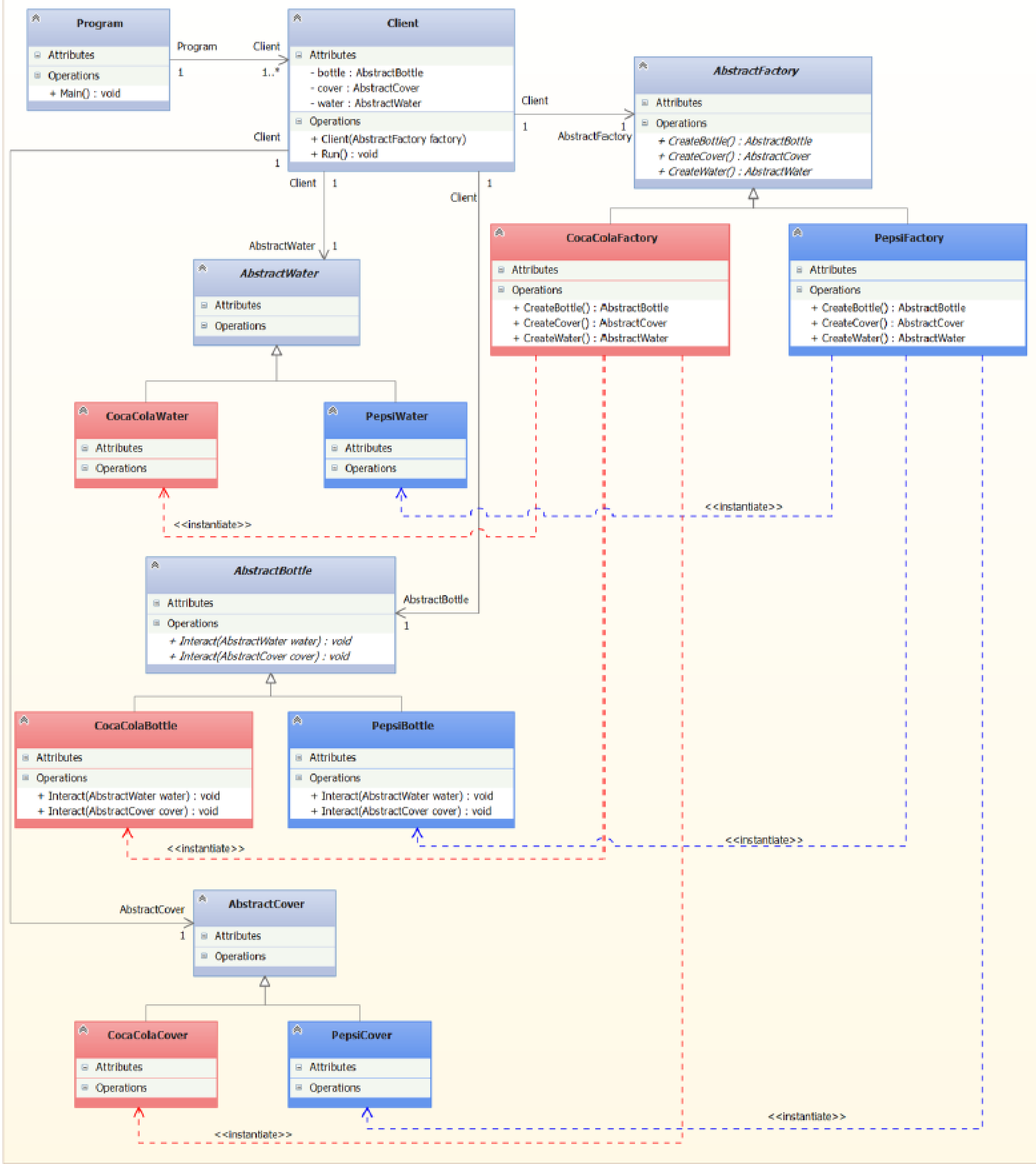


Рисунок 15 - Реализации конкретных классов продуктов и фабрик

Используя такой подход к порождению продуктов, теперь не составит труда добавлять новые виды продуктов в систему (например, крышку для закрытия банки с водой).


Program
Attributes
Operations
+ Main() : void
Program
Client
1
1..*
Client
Attributes
- bottle : AbstractBottle
- cover : AbstractCover
- water : AbstractWater
Operations
+ Client(AbstractFactory factory)
+ Run() : void
AbstractFactory
Attributes
Operations
+ CreateBottle() : AbstractBottle
+ CreateCover() : AbstractCover
+ CreateWater() : AbstractWater
CocaColaFactory
Attributes
Operations
+ CreateBottle() : AbstractBottle
+ CreateCover() : AbstractCover
+ CreateWater() : AbstractWater
PepsiFactory
Attributes
Operations
+ CreateBottle() : AbstractBottle
+ CreateCover() : AbstractCover
+ CreateWater() : AbstractWater
AbstractWater
Attributes
Operations
CocaColaWater
Attributes
Operations
PepsiWater
Attributes
Operations
<<instantiate>>
AbstractBottle
Attributes
Operations
+ Interact(AbstractWater water) : void
+ Interact(AbstractCover cover) : void
CocaColaBottle
Attributes
Operations
+ Interact(AbstractWater water) : void
+ Interact(AbstractCover cover) : void
PepsiBottle
Attributes
Operations
+ Interact(AbstractWater water) : void
+ Interact(AbstractCover cover) : void
AbstractCover
Attributes
Operations
CocaColaCover
Attributes
Operations
PepsiCover
Attributes
Operations

Структура паттерна на языке UML

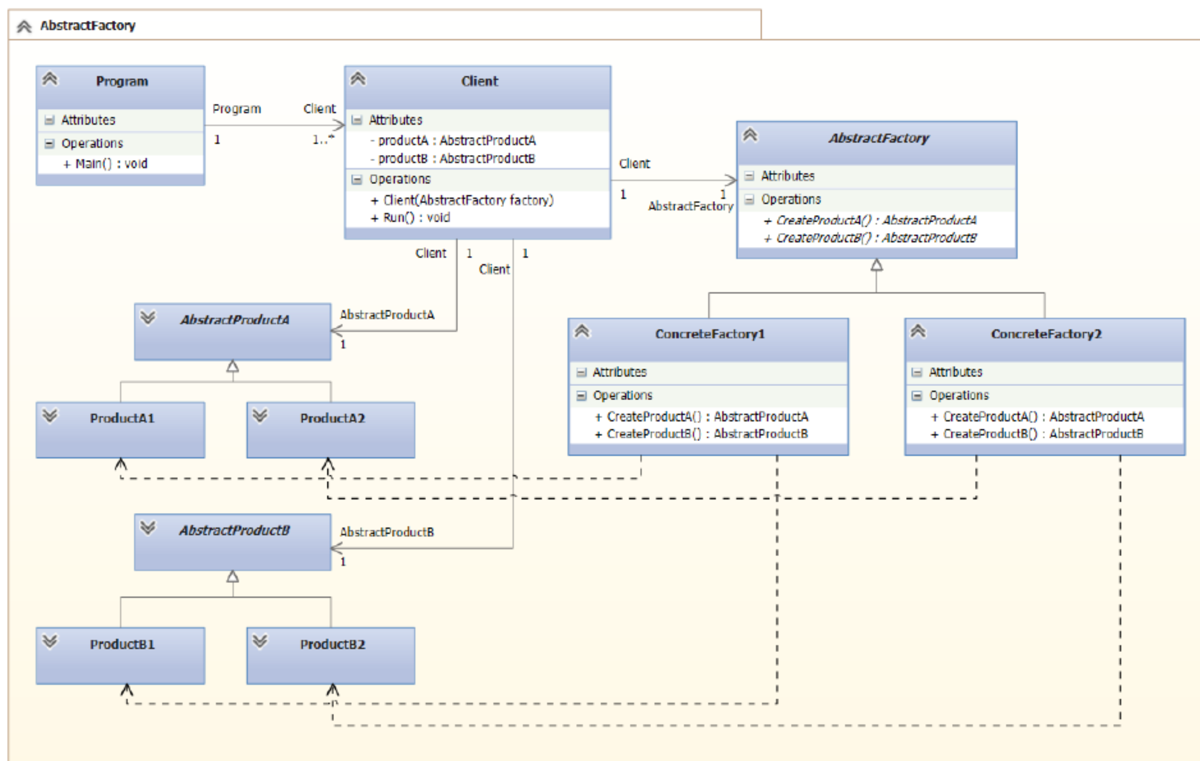
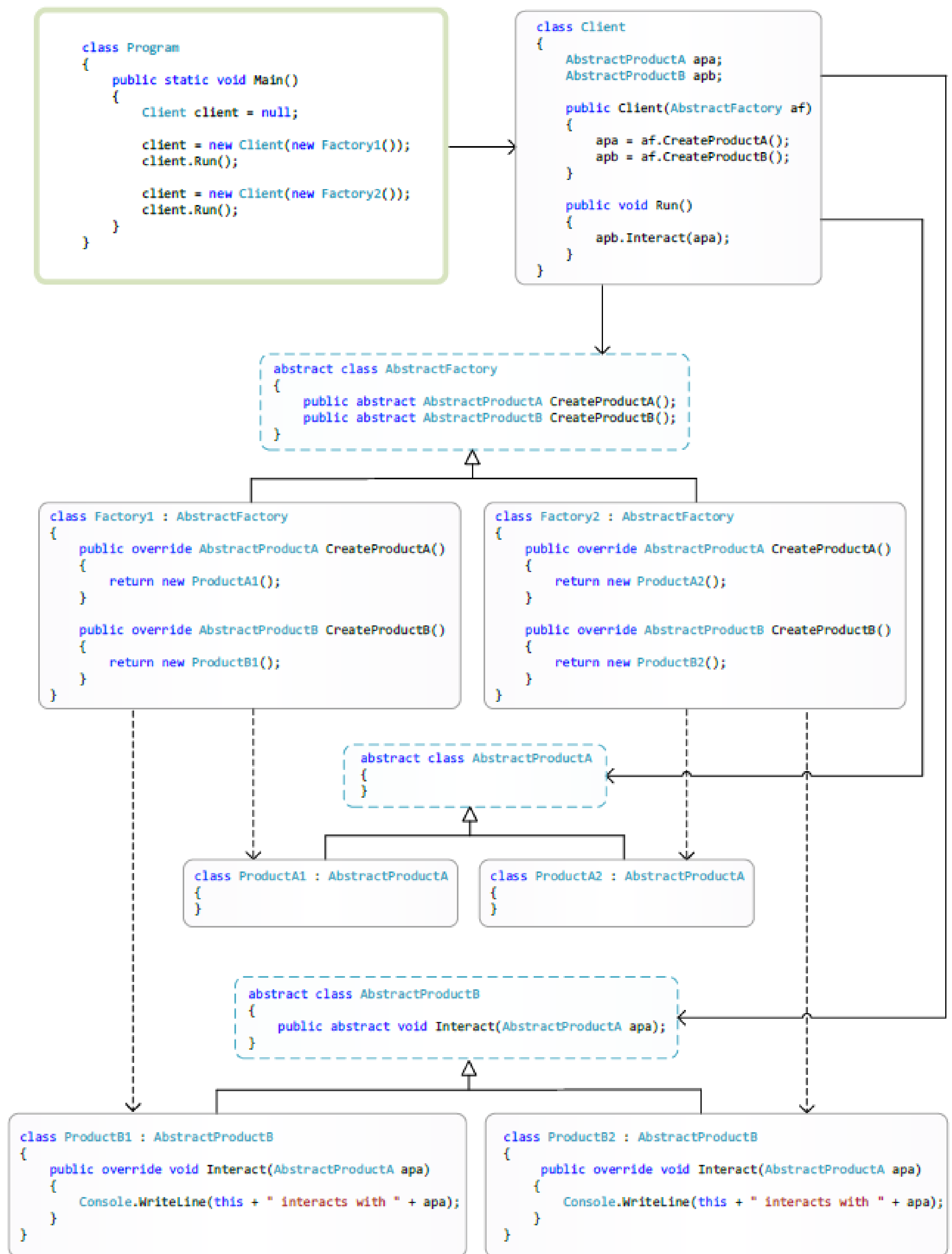


Рисунок 16 - Структура паттерна на языке UML



Участники

AbstractProduct - **Абстрактный продукт**:

Абстрактные классы продуктов предоставляют абстрактные интерфейсы взаимодействия с объектами-продуктами производных конкретных классов.

AbstractFactory - **Абстрактная фабрика**:

Класс AbstractFactory содержит в себе набор абстрактных фабричных методов. Эти абстрактные методы описывают интерфейс взаимодействия с объектами-фабриками и имеют возвращаемые значения типа абстрактных-продуктов, тем самым предоставляя возможность применять технику абстрагирования процесса инстанцирования. Класс AbstractFactory не занимается созданием объектов-продуктов, ответственность за их создание ложится на производный класс ConcreteFactory.

Client - **Клиент**:

Класс Client создает и использует продукты, пользуясь исключительно интерфейсом абстрактных классов AbstractFactory и AbstractProduct и ему ничего не известно о конкретных классах фабрик и продуктов.

 ConcreteProduct - **Конкретный продукт**:

Конкретные классы продукты, наследуются от абстрактных классов продуктов. Объекты-продукты конкретных классов предполагается создавать в телах фабричных методов реализаций соответствующих фабрик.

 ConcreteFactory - **Конкретная фабрика**:

Классы конкретных фабрик, наследуются от абстрактной фабрики и реализуют фабричные методы порождающие объекты-продукты.

Отношения между участниками

Отношения между классами

Класс Client связан связями отношения ассоциации с классами абстрактных продуктов и классом абстрактной фабрики.

Все конкретные классы продуктов связаны связями отношения наследования с абстрактными классами продуктов.

Все конкретные классы фабрик связаны связями отношения наследования с классом абстрактной фабрики и связями отношения зависимости (стереотипа - instantiate) с конкретными классами порождаемых продуктов.

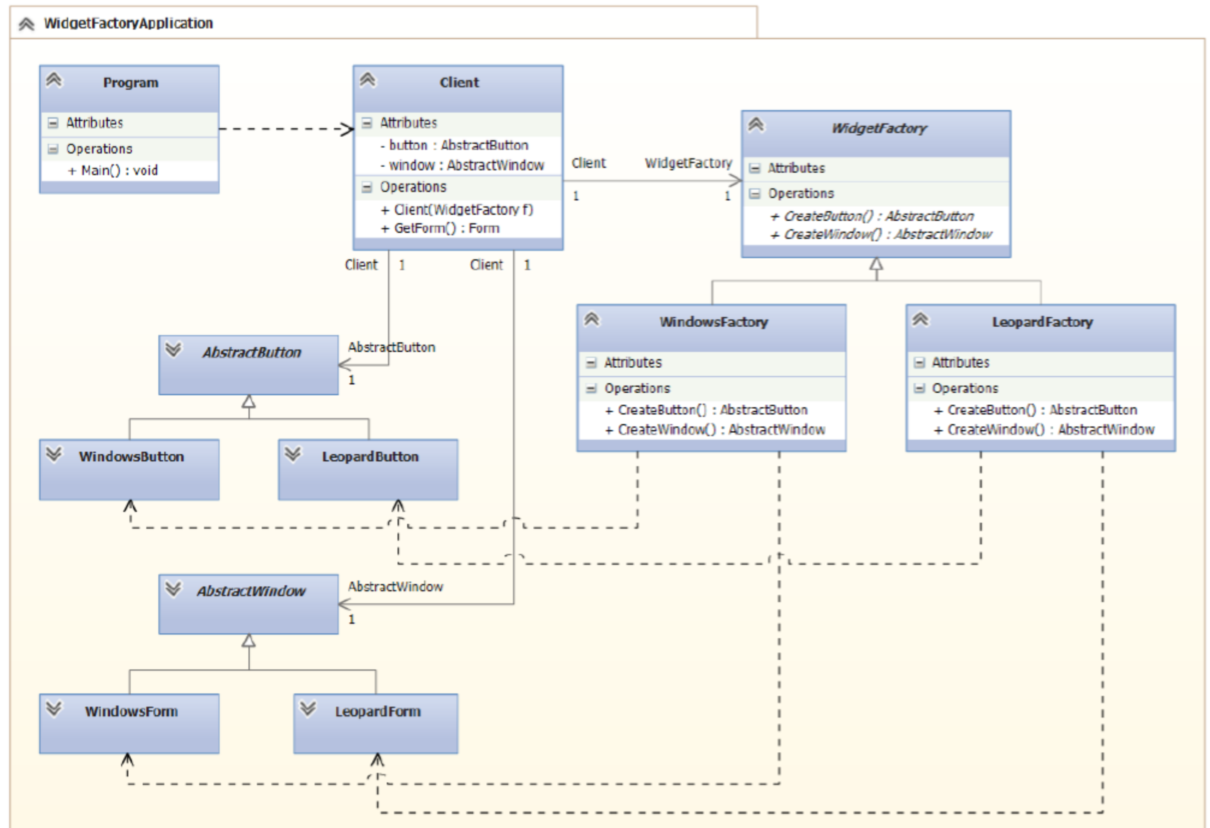
Отношения между объектами

В системе создается (чаще всего) только один экземпляр конкретной фабрики. Задачей конкретной фабрики является создание объектов продуктов, входящих в определенное семейство.

При создании экземпляра клиента, клиент конфигурируется экземпляром конкретной фабрики (ссылка на экземпляр фабрики передается в качестве аргумента конструктора клиента).

Абстрактный класс AbstractFactory передает ответственность за создание объектов-продуктов производным конкретным фабрикам.

Мотивация



Применимость паттерна Abstract Factory

Паттерн Abstract Factory рекомендуется использовать, когда:

- Требуется создавать объекты-продукты разных типов и налаживать между ними взаимодействие, при этом образуя семейства из этих объектов-продуктов. Входящие в семейство объекты-продукты обязательно должны использоваться вместе.

- Требуется построить подсистему (модуль или компонент) таким образом, чтобы ее внутреннее устройство (состояние и/или поведение) настраивалось при ее создании. При этом чтобы ни процесс, ни результат построения подсистемы не был зависим от способа создания в ней объектов, их композиции (составления и соединения объектов) и представления (настройки внутреннего состояния объектов).

- Подсистема или система должна настраиваться (конфигурироваться) через использование одного из семейств объектов-продуктов, порождаемых одним объектом-фабрикой;

Результаты

Паттерн Abstract Factory обладает следующими преимуществами:

Сокрытие работы с конкретными классами продуктов.

Фабрика скрывает от клиента детали реализации конкретных классов и процесс создания экземпляров этих классов. Конкретные классы-продуктов известны только конкретным фабрикам и в коде клиента они не используются. Клиент управляет экземплярами конкретных классов только через их абстрактные интерфейсы.

Позволяет легко заменять семейства используемых продуктов.

Экземпляр класса конкретной фабрики создается в приложении в одном месте и только один раз, что позволяет в дальнейшем проще подменять фабрики. Для того чтобы изменить семейство используемых продуктов, нужно просто создать новый экземпляр класса-фабрики, тогда заменится сразу все семейство.

Обеспечение совместного использования продуктов.

Позволяет легко контролировать взаимодействие между объектами-продуктами, которые спроектированы для совместного использования и входят в одно семейство.

Паттерн Abstract Factory обладает следующим недостатком:

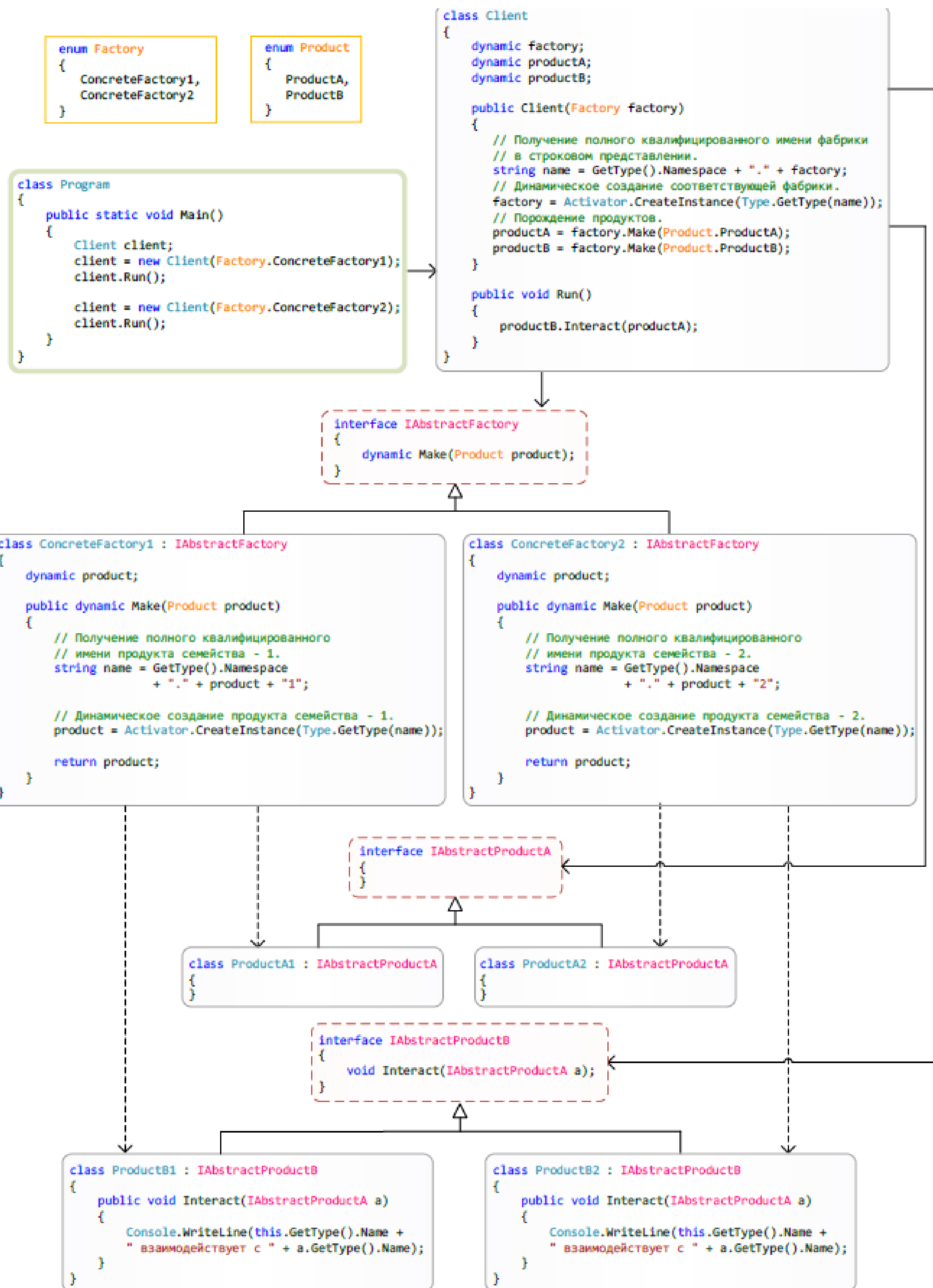
Имеется небольшое неудобство добавления нового вида продуктов.

Для создания нового вида продуктов потребуется создать новые классы продуктов (абстрактные и конкретные), добавить новый абстрактный фабричный метод в абстрактный класс фабрики и реализовать этот абстрактный метод в производных конкретных классах фабриках, а также изменить код класса Client.

Реализация

Полезные приемы реализации паттерна Abstract Factory:

Объекты-фабрики существуют в единственном экземпляре.
Создание объектов-продуктов.
Определение расширяемых фабрик.



Известные применения паттерна в .Net

Microsoft.Build.Tasks.CodeTaskFactory
http://msdn.microsoft.com/ru-ru/library/microsoft.build.tasks.codetaskfactory.aspx
Microsoft.Build.Tasks.XamlTaskFactory
http://msdn.microsoft.com/ru-ru/library/microsoft.build.tasks.xamltaskfactory.aspx
Microsoft.IE.SecureFactory
http://msdn.microsoft.com/ru-ru/library/microsoft.ie.securefactory(v=vs.90).aspx
System.Activities.Presentation.Model.ModelFactory

http://msdn.microsoft.com/ru-ru/library/system.activities.presentation.model.modelfactory.aspx
System.Data.Common.DbProviderFactory
http://msdn.microsoft.com/ru-ru/library/system.data.common.dbproviderfactory.aspx
System.Data.EntityClient.EntityProviderFactory
http://msdn.microsoft.com/ru-ru/library/system.data.entityclient.entityproviderfactory.aspx
System.Data.Odbc.OdbcFactory
http://msdn.microsoft.com/ru-ru/library/system.data.odbc.odbcfactory.aspx
System.Data.OleDb.OleDbFactory
http://msdn.microsoft.com/ru-ru/library/system.data.oledb.oledbfactory.aspx
System.Data.OracleClient.OracleClientFactory
http://msdn.microsoft.com/ru-ru/library/system.data.oracleclient.oracleclientfactory.aspx
System.Data.Services.DataServiceHostFactory
http://msdn.microsoft.com/ru-ru/library/system.data.services.dataservicehostfactory.aspx
System.Data.SqlClient.SqlClientFactory
http://msdn.microsoft.com/ru-ru/library/system.data.sqlclient.sqlclientfactory.aspx
System.ServiceModel.ChannelFactory
http://msdn.microsoft.com/ru-ru/library/system.servicemodel.channelfactory.aspx
System.Threading.Tasks.TaskFactory
http://msdn.microsoft.com/ru-ru/library/system.threading.tasks.taskfactory(v=vs.110).aspx
System.Web.Compilation.ResourceProviderFactory
http://msdn.microsoft.com/ru-ru/library/system.web.compilation.resourceproviderfactory.aspx
System.Web.Hosting.AppDomainFactory
http://msdn.microsoft.com/ru-ru/library/system.web.hosting.appdomainfactory(v=vs.90).aspx
System.Xml.Serialization.XmlSerializerFactory
http://msdn.microsoft.com/ru-ru/library/system.xml.serialization.xmlserializerfactory(v=vs.90).aspx

Паттерн Builder

Название - Строитель

Также известен как

-

Классификация

По цели: порождающий

По применимости: к объектам

Частота использования

Ниже средней - 1 **2** 3 4 5

Назначение

Паттерн Builder – помогает организовать пошаговое построение сложного объекта-продукта так, что клиенту не требуется понимать последовательность шагов и внутреннее устройство строящегося объекта-продукта, при этом в результате одного и того же процесса конструирования могут получаться объекты- продукты с различным представлением (внутренним устройством).

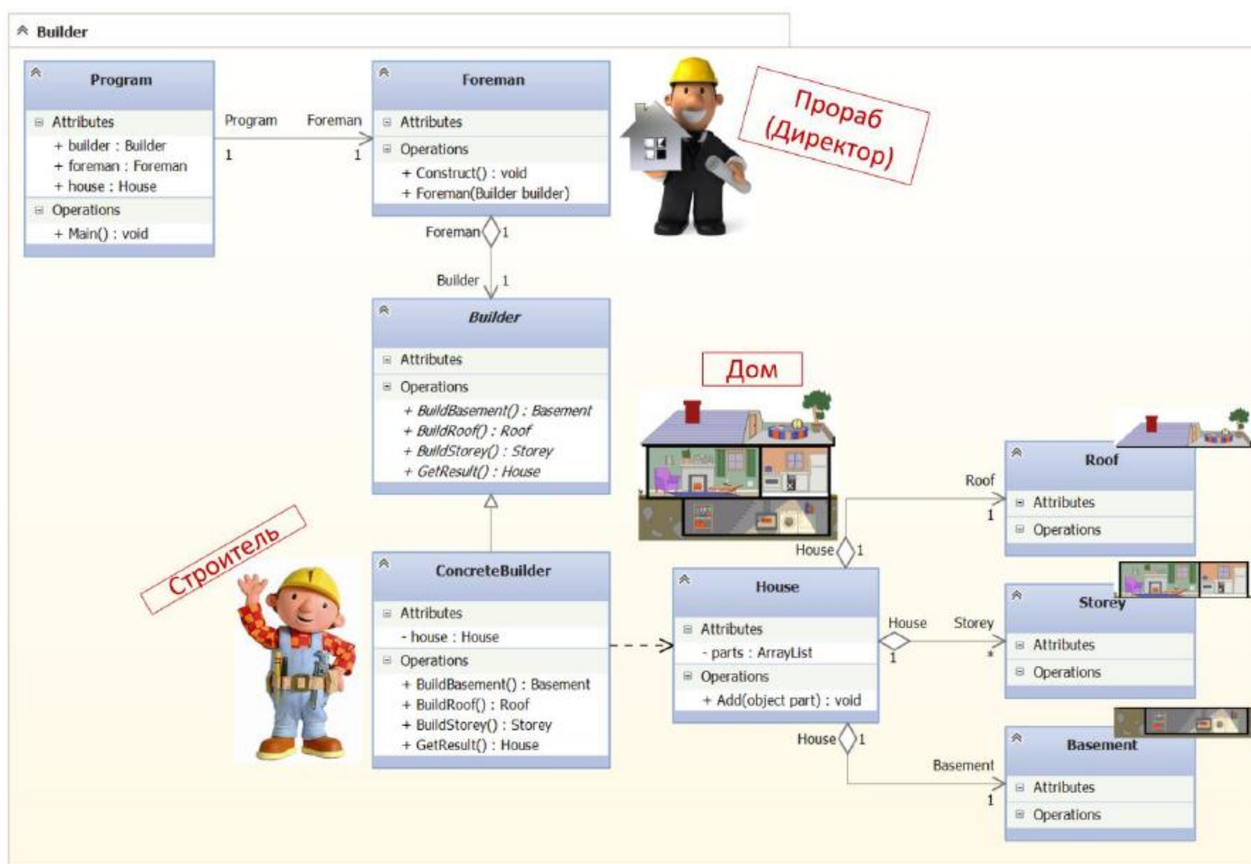
Введение

Кто такой строитель в объективной реальности? Строитель - это человек, который занимается возведением зданий и сооружений (мостов, плотин, туннелей и пр.). Результатом строительства считается возведённое здание (сооружение). Для того чтобы здание было построено по правилам и соответствовало проектным нормам, строителями нужно руководить. Должность руководителя на стройке называется прораб (сокращение от «производитель работ»). Прораб дает указания строителю, как и в каком порядке проводить строительные работы. Паттерн Builder, построен на подобной метафоре.

Прораб, должен давать строителю инструкции по построению частей дома в определенной последовательности. Например,

1. «Построй подвал»,
2. «Построй этаж»,
3. «Построй крышу».

Способ построения дома определяет тип конкретного строителя. Строитель-каменщик, который строит дом из кирпича, будет строить дом отличным способом, от строителя-плотника который будет строить сруб (деревянный дом) из бревен. Таким образом, согласно проекту, прораб должен вызвать соответствующего строителя и давать ему соответствующие инструкции в определенном порядке. Сначала построить подвал, потом этаж и в последнюю очередь крышу.



Структура паттерна на языке UML

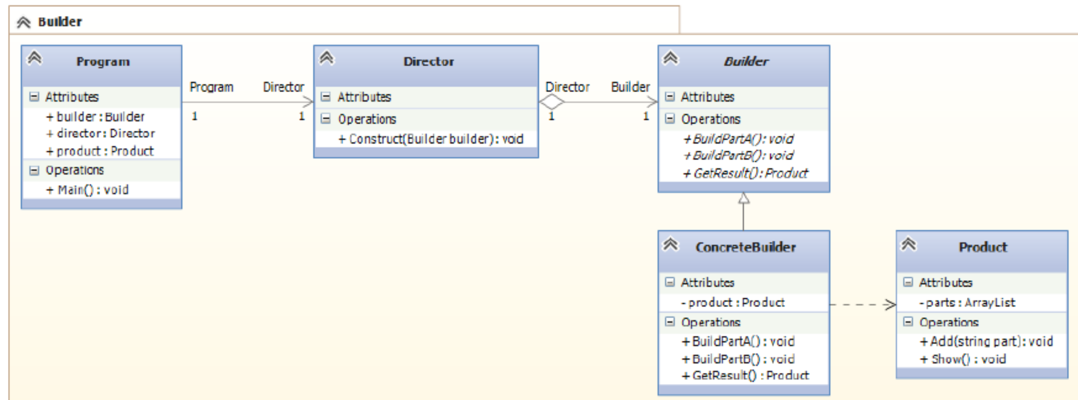


Рисунок 17 - Структура паттерна на языке UML

```
class Program
{
    public static void Main()
    {
        Builder builder = new ConcreteBuilder();
        Director director = new Director(builder);
        director.Construct();
        Product product = builder.GetResult();
        product.Show();
    }
}
```

```
class Director
{
    Builder builder;

    public Director(Builder builder)
    {
        this.builder = builder;
    }

    public void Construct()
    {
        builder.BuildPartA();
        builder.BuildPartB();
        builder.BuildPartC();
    }
}
```

```
abstract class Builder
{
    public abstract void BuildPartA();
    public abstract void BuildPartB();
    public abstract void BuildPartC();
    public abstract Product GetResult();
}
```

```
class ConcreteBuilder : Builder
{
    Product product = new Product();

    public override void BuildPartA()
    {
        product.Add("Part A");
    }

    public override void BuildPartB()
    {
        product.Add("Part B");
    }

    public override void BuildPartC()
    {
        product.Add("Part C");
    }

    public override Product GetResult()
    {
        return product;
    }
}
```

```
class Product
{
    ArrayList parts = new ArrayList();

    public void Add(string part)
    {
        parts.Add(part);
    }

    public void Show()
    {
        foreach (string part in parts)
            Console.WriteLine(part);
    }
}
```

Рисунок 18 - Структура паттерна на языке C#

Участники

Product - **Продукт**:

Представляет собой класс сложно-конструируемого объекта-продукта и содержит в себе набор методов для сборки конечного результата-продукта из частей. Класс продукта может быть связан связями отношений агрегации, с классами которые описывают составные части создаваемого продукта.

Builder - **Абстрактный строитель**:

Предоставляет набор абстрактных методов (интерфейс) для создания объекта-продукта из частей и получения готового результата.

ConcreteBuilder - **Конкретный строитель**:

Конструирует объект-продукт собирая его из частей, реализуя интерфейс, заданный абстрактным строителем (Builder). Предоставляет доступ к готовому продукту (возвращает продукт клиенту или в частном случае директору (Director)).

Director – **Директор (Распорядитель)**:

Пользуясь интерфейсом строителя (Builder), директор дает строителю указание построить продукт.

Отношения между участниками

Отношения между классам

Класс Director связан связью отношения агрегации с абстрактным классом Builder.

Класс ConcreteBuilder связан связью отношения наследования с абстрактным классом Builder и связью отношения зависимости с классом Product.

Класс Product может быть связан связями отношения агрегации с классами частей (Part).

Отношения между объектами

Клиент создает экземпляр класса ConcreteBuilder.

Клиент создает экземпляр класса Director при этом в качестве аргумента конструктора передает ссылку на ранее созданный экземпляр класса ConcreteBuilder.

Директор (Director) вызывает на строителе (ConcreteBuilder) методы, тем самым уведомляя строителя о том, что требуется построить определенную часть продукта.

Строитель выполняет операции по построению продукта, добавляя к продукту те части, которые указывает директор (Director).

Клиент получает от строителя ссылку на экземпляр построенного продукта.

На диаграмме последовательностей показаны отношения между объектами (директором и строителем).

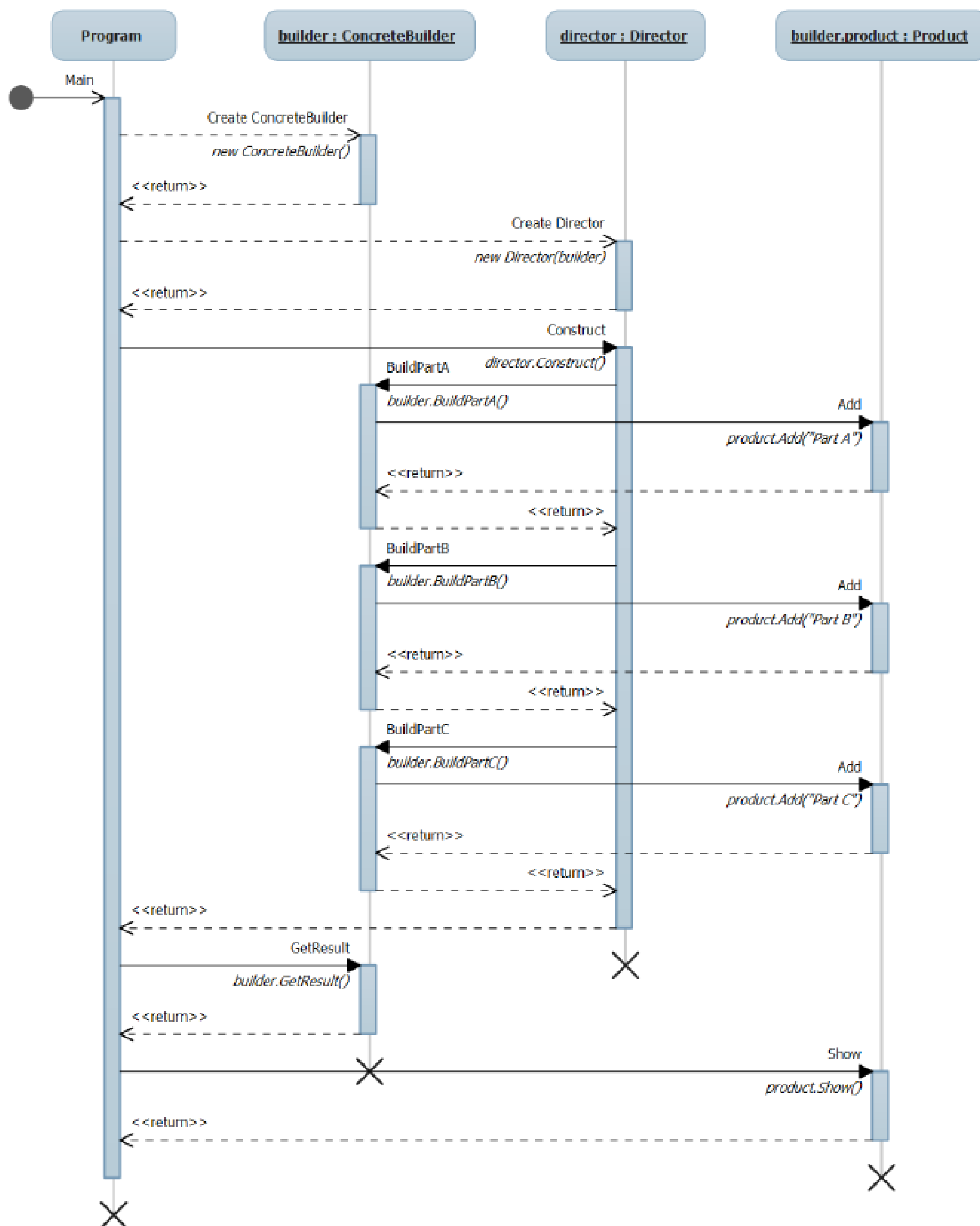
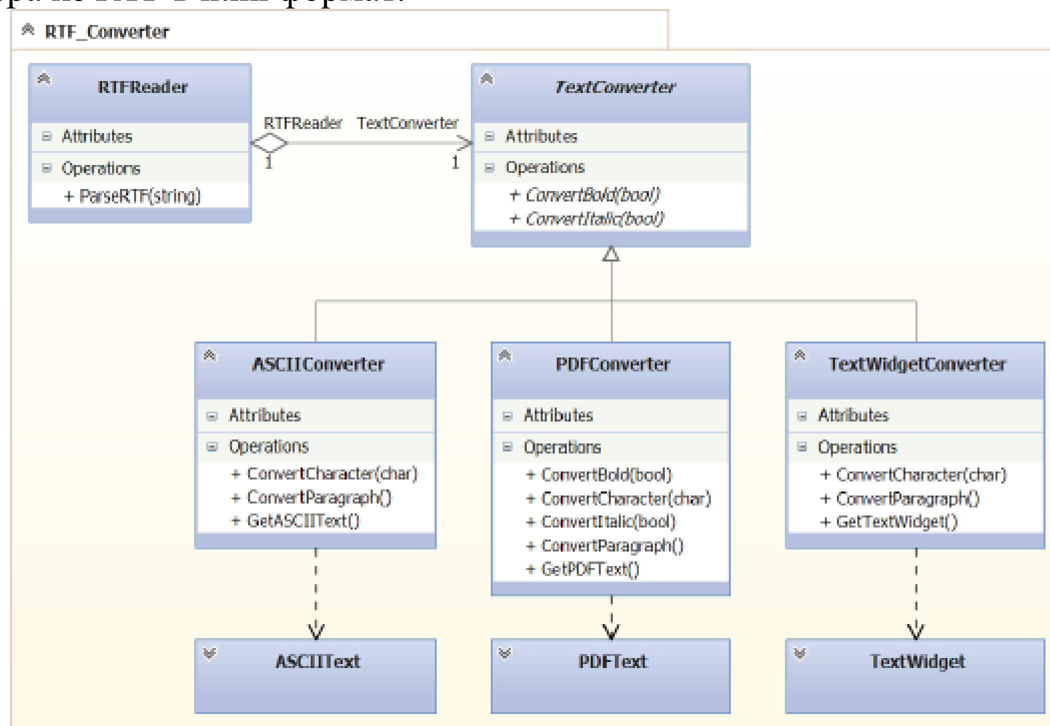


Рисунок 19 - отношения между объектами (директором и строителем)

Мотивация

Предлагается написать программу для преобразования RTF документа в другие форматы: ASCII, PDF и в представление для отображения на форме в элементе управления. Требуется предусмотреть возможность легкого добавления новых механизмов преобразований в другие форматы.

Возможное число форматов, в которые необходимо будет преобразовать документ заранее неизвестно. Поэтому должна быть предусмотрена (обеспечена) возможность легкого добавления нового конвертера, например, конвертера из RTF в html формат.



Таким образом, нужно сконфигурировать экземпляр класса RTFReader (Директор) объектом типа TextConverter(абстрактный Builder), который мог бы преобразовывать RTF формат в другие форматы. При разборе (анализе)документа в формате RTF, объект класса RTFReader дает команду объекту типа TextConverter выполнить преобразование формата. При этом каждый раз при распознавании «лексемы RTF» (т.е. простого текста или управляющего слова) RTFReader вызывает необходимый метод объекта типа TextConverter. Подклассы класса TextConverter (конкретные конвертеры) отвечают, как за преобразование данных (текста), так и за представление лексемы в новом формате.

Каждый подкласс класса TextConverter (конкретный строитель) позволяет преобразовать RTF формат только в один определенный формат. Например, ASCIIConverter преобразовывает RTF формат в простейший ASCII формат, при этом игнорирует преобразование таблиц, изображений и шрифтов. С другой стороны, PDFConverter будет преобразовывать все содержимое включая таблицы, изображения, шрифты, стили и пр. А TextWidgetConverter построит объект пользовательского интерфейса (контрол), и преобразование формата будет зависеть от возможностей используемого элемента управления (контрола). Например, TextBoxсможет отображать только простой текст, тогда как RichTextBox сможет полноценно отобразить все составляющие документа в RTF формате.

Класс каждого конкретного конвертера (строителя) использует механизм создания и сборки сложного объекта-продукта и скрывает этот механизм за реализацией интерфейса, предоставленного классом

TextConverter. Конвертер отделен отобъекта класса RTFReader(директора), который отвечает за синтаксический разбор RTF документа.

В паттерне Builder абстрагированы все отношения между директором и строителями и любой объект типа TextConverter будет называться Строителем (Builder), а RTFReader-Директором (Director). Применение паттерна Builder в данном примере позволяет отделить алгоритм интерпретации текста в RTF формате (анализатор RTF документов) от алгоритма конвертации документа в новый формат. Это позволяет повторно использовать алгоритм интерпретации текста в RTF, реализованный в RTFReader(Директоре), в связкес различными конвертерами в другие форматы (Строителями). Для этого достаточно сконфигурировать RTFReader необходимым конвертером(подклассом класса TextConverter).

Применимость паттерна

Паттерн Строитель рекомендуется использовать, когда:

- Алгоритм пошагового создания сложного объекта-продукта не должен зависеть от того, из каких частей состоит объект-продукт и как эти части стыкуются между собой;

- Процесс создания продукта должен обеспечивать возможность получения различных вариаций создаваемого продукта.

Результаты

Паттерн Builder обладает следующими преимуществами:

Позволяет изменять состав продукта.

Абстрактный класс Builder предоставляет директору набор абстрактных методов (абстрактный интерфейс) для управления построением продукта. За абстрактным интерфейсом Builder скрывает внутреннюю структуру создаваемого продукта и процесс его построения. Поскольку построение продукта производится согласно абстрактному интерфейсу, то для изменения структуры продукта достаточно создать новую разновидность строителя;

Скрывает код, реализующий конструирование и представление.

Паттерн Builder улучшает модульность, скрывая способы построения и представления сложных объектов-продуктов. Клиенты ничего не знают о классах, входящих в состав внутренней структуры продукта, использование этих классов отсутствует в интерфейсе строителя.

Конкретные строители ConcreteBuilder содержат код, необходимый для создания и сборки конкретного вида продукта. Код пишется только один раз и разные директоры могут использовать его повторно для построения различных вариантов продукта из одних и тех же частей комбинируя эти части.

Предоставляет полный контроль над процессом построения продукта.

В отличие от других порождающих паттернов, которые сразу конструируют весь объект-продукт полностью, строитель строит продукт шаг за шагом под управлением директора.

И только тогда, когда построение продукта завершено, директор или клиент забирают его у строителя. Поэтому интерфейс строителя в большей степени отражает процесс пошагового конструирования продукта, нежели другие порождающие паттерны. Это позволяет обеспечить полный контроль над процессом конструирования, а значит, и над внутренней структурой (комбинацией частей) готового продукта.

Реализация

Обычно используется абстрактный класс Builder, предоставляющий интерфейс для построения каждой отдельной части продукта, который директор может «попросить» создать. В классах конкретных строителей ConcreteBuilder реализуются абстрактные операции абстрактного класса Builder.

Полезные приемы реализации паттерна строитель:

Интерфейс строителя.

Строители конструируют продукты шаг за шагом. Интерфейс класса Builder должен быть достаточно общим, чтобы обеспечить создание продуктов при любой реализации конкретных строителей. Иногда может потребоваться доступ к частям уже сконструированного, готового продукта и такую возможность желательно предусмотреть.

Отсутствие общего базового абстрактного класса для продуктов.

Чаще всего продукты имеют настолько разный состав, что создание для них общего базового класса ничего не дает.

Пример кода игры «Лабиринт»

Класс MazeBuilder предоставляет абстрактный интерфейс для построения лабиринта:

```
abstract class MazeBuilder
{
	public abstract void BuildMaze();
	public abstract void BuildRoom(int roomNo);
	public abstract void BuildDoor(int roomFrom, int roomTo);
	public abstract Maze GetMaze();
}
```

Этот интерфейс позволяет создавать три типа объектов: целый лабиринт, комнату с номером и двери между пронумерованными комнатами. Реализация метода GetMaze в подклассах MazeBuilder создает и возвращает лабиринт клиенту.

Класс MazeGame предоставляет собой объектно-ориентированное представление всей игры. Метод CreateMaze класса MazeGame, принимает в качестве аргумента ссылку на экземпляр конкретного строителя типа

MazeBuilder и возвращает построенный лабиринт (ссылку на экземпляр класса Maze).

```
public Maze CreateMaze(MazeBuilder builder)
{
	builder.BuildMaze();
	builder.BuildRoom(1);
	builder.BuildRoom(2);
	builder.BuildDoor(1, 2);
	// Возвращает готовый продукт (Лабиринт)
	return builder.GetMaze();
}
```

Эта версия метода CreateMaze показывает, что строитель скрывает внутреннее устройство лабиринта, то есть конкретные классы комнат, дверей и стен.

Как и все другие порождающие паттерны, паттерн строитель позволяет скрывать способы создания объектов. В данном примере сокрытие организуется при помощи интерфейса, предоставляемого классом MazeBuilder. Это означает, что MazeBuilder можно использовать для построения лабиринтов любых разновидностей. В качестве примера для построения альтернативного лабиринта рассмотрим метод CreateComplexMaze принадлежащий классу MazeGame:

```
public Maze CreateComplexMaze(MazeBuilder builder)
{
	// Построение 1001-й комнаты.
	for (int i = 0; i < 1001; i++)
	{
		builder.BuildRoom(i + 1);
	}
	return builder.GetMaze();
}
```

Важно понимать, что MazeBuilder не создает лабиринты напрямую, его главная задача –предоставить абстрактный интерфейс, описывающий создание лабиринта. Реальную работу по построению лабиринта выполняют конкретные подклассы класса MazeBuilder.

Класс StandardMazeBuilder реализует логику построения простых лабиринтов.

```
    // Подкласс StandardMazeBuilder - содержит реализацию построения простых
лабиринтов.
    class StandardMazeBuilder : MazeBuilder
    {

        Maze currentMaze = null;

        // Конструктор.
        public StandardMazeBuilder()
        {
            this.currentMaze = null;
        }


        // Инстанцирует экземпляр класса Maze, который будет собираться другими
        // операциями.
        public override void BuildMaze()
        {
            this.currentMaze = new Maze();
        }


        // Создает комнату и строит вокруг нее стены.
        public override void BuildRoom(int roomNo)
        {
            //if (currentMaze.RoomNo(roomNo) == null)
            {
                Room room = new Room(roomNo);
                currentMaze.AddRoom(room);

                room.SetSide(Direction.North, new Wall());
                room.SetSide(Direction.South, new Wall());
                room.SetSide(Direction.East, new Wall());
                room.SetSide(Direction.West, new Wall());
            }
        }


        // Чтобы построить дверь между двумя комнатами, требуется найти обе комнаты в
        // лабиринте и их общую стену.
        public override void BuildDoor(int roomFrom, int roomTo)
        {
            Room room1 = currentMaze.RoomNo(roomFrom);
            Room room2 = currentMaze.RoomNo(roomTo);
            Door door = new Door(room1, room2);

            room1.SetSide(CommonWall(room1, room2), door);
            room2.SetSide(CommonWall(room2, room1), door);
        }
```

```
        // Возвращает клиенту собранный продукт т.е., лабиринт.
        public override Maze GetMaze()
        {
            return this.currentMaze;
        }
```

```
        // CommonWall - Общая стена.
        // Это вспомогательная операция, которая определяет направление общей для
двух
        // комнат стены.
        private Direction CommonWall(Room room1, Room room2)
        {
            if (room1.GetSide(Direction.North) is Wall &&
                room1.GetSide(Direction.South) is Wall &&
                room1.GetSide(Direction.East) is Wall &&
                room1.GetSide(Direction.West) is Wall &&
                room2.GetSide(Direction.North) is Wall &&
                room2.GetSide(Direction.South) is Wall &&
                room2.GetSide(Direction.East) is Wall &&
                room2.GetSide(Direction.West) is Wall)
            {
                return Direction.East;
            }
            else
            {
                return Direction.West;
            }
        }
    }
```

Известные применения паттерна в .Net

`System.Data.Common.DbCommandBuilder`
http://msdn.microsoft.com/ru-ru/library/system.data.common.dbcommandbuilder(v=vs.90).aspx
`System.Data.Common.DbConnectionStringBuilder`
http://msdn.microsoft.com/ru-ru/library/system.data.common.dbconnectionstringbuilder.aspx
`System.Data.Odbc.OdbcCommandBuilder`
http://msdn.microsoft.com/ru-ru/library/system.data.odbc.odbccommandbuilder.aspx
`System.Data.Odbc.OdbcConnectionStringBuilder`
http://msdn.microsoft.com/ru-ru/library/system.data.odbc.odbcconnectionstringbuilder.aspx
`System.Data.OleDb.OleDbCommandBuilder`
http://msdn.microsoft.com/ru-ru/library/system.data.oledb.oledbcommandbuilder(v=vs.90).aspx
`System.Data.OleDb.OleDbConnectionStringBuilder`
http://msdn.microsoft.com/ru-ru/library/system.data.oledb.oledbconnectionstringbuilder.aspx
`System.Data.SqlClient.SqlCommandBuilder`
http://msdn.microsoft.com/ru-ru/library/system.data.sqlclient.sqlcommandbuilder.aspx
`System.Data.SqlClient.SqlConnectionStringBuilder`
http://msdn.microsoft.com/ru-ru/library/system.data.sqlclient.sqlconnectionstringbuilder.aspx
`System.Reflection.Emit.ConstructorBuilder`
http://msdn.microsoft.com/ru-ru/library/system.reflection.emit.constructorbuilder.aspx
`System.Reflection.Emit.EnumBuilder`
http://msdn.microsoft.com/ru-ru/library/system.reflection.emit.enumbuilder.aspx
`System.Reflection.Emit.EventBuilder`
http://msdn.microsoft.com/ru-ru/library/system.reflection.emit.eventbuilder.aspx
`System.Reflection.Emit.FieldBuilder`
http://msdn.microsoft.com/ru-ru/library/system.reflection.emit.fieldbuilder.aspx
`System.Reflection.Emit.MethodBuilder`
http://msdn.microsoft.com/ru-ru/library/system.reflection.emit.methodbuilder.aspx
`System.Reflection.Emit.ParameterBuilder`
http://msdn.microsoft.com/ru-ru/library/system.reflection.emit.parameterbuilder(v=vs.100).aspx
`System.Reflection.Emit.PropertyBuilder`
http://msdn.microsoft.com/ru-ru/library/system.reflection.emit.propertybuilder.aspx
`System.Reflection.Emit.TypeBuilder`
http://msdn.microsoft.com/ru-ru/library/system.reflection.emit.typebuilder(v=vs.110).aspx
`System.Text.StringBuilder`
http://msdn.microsoft.com/ru-ru/library/system.text.stringbuilder.aspx

Паттерн Factory Method

Название - Фабричный Метод

Также известен как Virtual Constructor (Виртуальный Конструктор)

Классификация

По цели: порождающий

По применимости: к классам

Частота использования

Высокая - 1 2 3 4 **5**

Назначение

Паттерн Factory Method – предоставляет абстрактный интерфейс (набор методов) для создания объекта-продукта, но оставляет возможность, разработчикам классов, реализующих этот интерфейс самостоятельно принять решение о том, экземпляр какого конкретного класса-продукта создать. Паттерн Factory Method позволяет базовым абстрактным классам передать ответственность за создание объектов-продуктов своим производным классам.

Введение

Паттерн Factory Method лежит в основе всех порождающих паттернов, организовывая процесс порождения объектов-продуктов. Если проектировщик на этапе проектирования системы не может сразу определиться с выбором подходящего паттерна для организации процесса порождения продукта в конкретной ситуации, то сперва следует воспользоваться паттерном Factory Method.

Например, если проектировщик, не определился со сложностью продукта или с необходимостью и способом организации взаимодействия между несколькими продуктами, тогда есть смысл сперва воспользоваться паттерном Factory Method. Позднее, когда требования будут сформулированы более четко, можно будет произвести быструю подмену паттерна Factory Method на другой порождающий паттерн, более соответствующий проектной ситуации.

Важно помнить, что Factory Method является паттерном уровня классов, и он сфокусирован только на отношениях между классами. Основной задачей паттерна Factory Method является организация техники делегирования ответственности за создание объектов продуктов одним классом (часто абстрактным) другому классу (производному конкретному классу). Другими словами – абстрактный класс содержащий в себе абстрактный фабричный метод, говорит своему производному конкретному классу: «Конкретный класс, я поручаю твоему разработчику самостоятельно выбрать конкретный класс порождаемого объекта-продукта при реализации моего абстрактного фабричного метода».

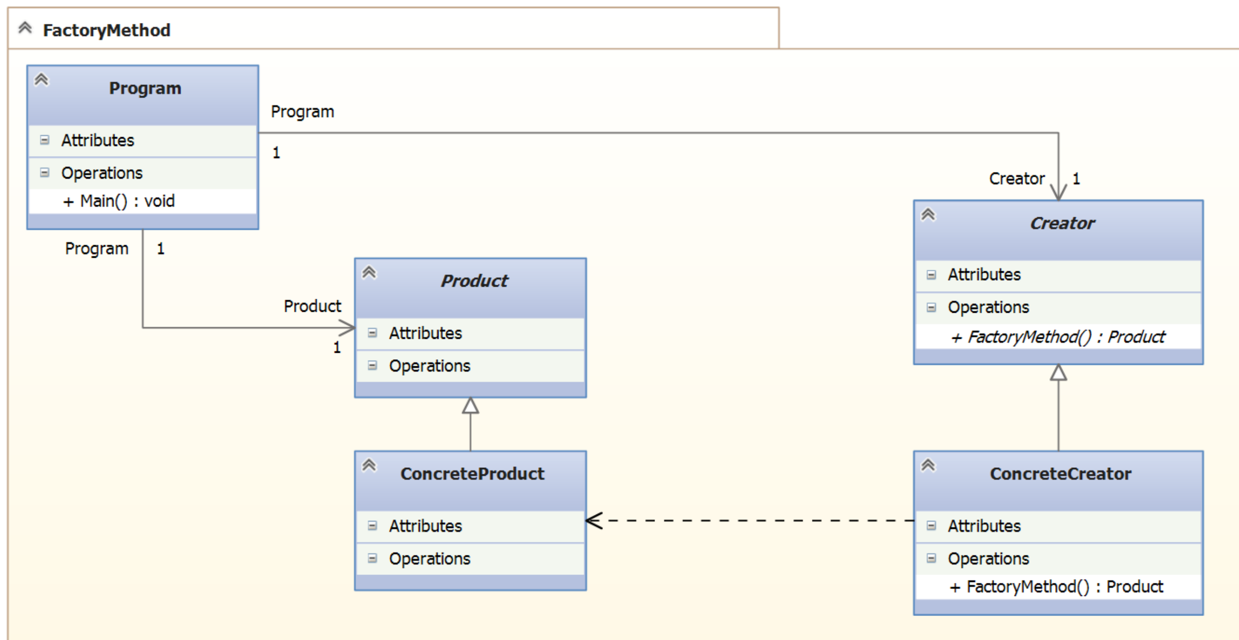


Рисунок 20 - Структура паттерна на языке UML

Структура паттерна на языке C#

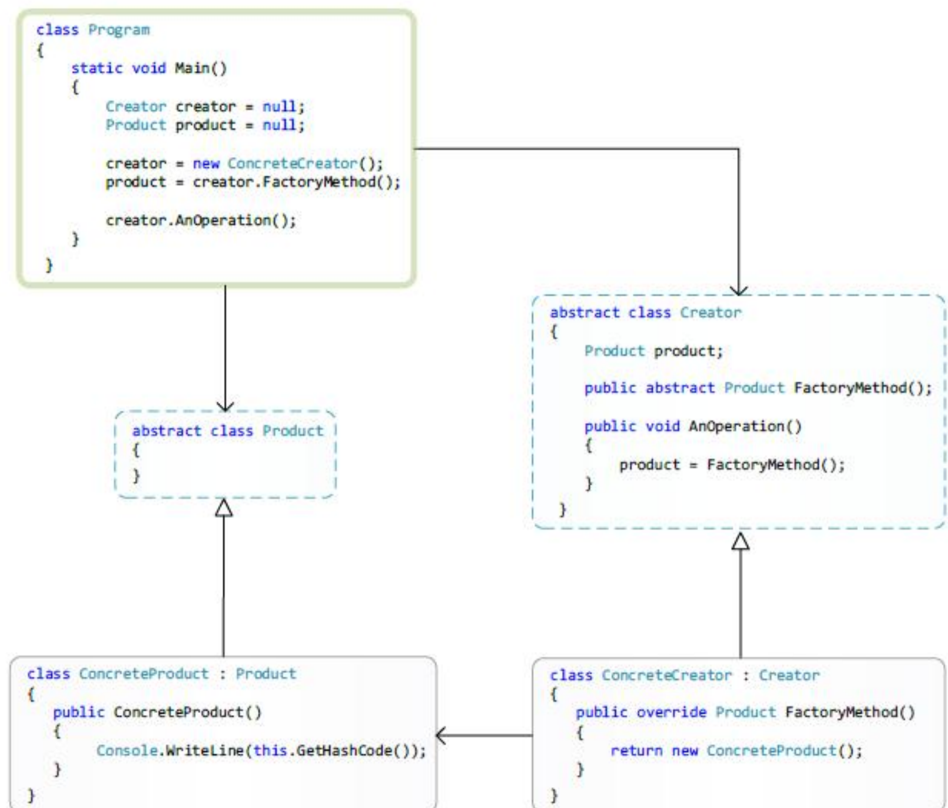


Рисунок 21- Структура паттерна на языке C#

Участники

Product - **Продукт**:

Предоставляет интерфейс для взаимодействия с продуктами.
Creator - **Создатель**:

Предоставляет интерфейс (абстрактные фабричные методы) для порождения продуктов. В частных случаях класс Creator может предоставлять реализацию фабричных методов, которые возвращают экземпляры продуктов (ConcreteProduct).

ConcreteProduct - **Конкретный продукт**:

Реализует интерфейс предоставляемый базовым классом Product.

ConcreteCreator - **Конкретная фабрика**:

Реализует интерфейс (фабричные методы) предоставляемый базовым классом Creator.

Отношения между участниками

Отношения между классами

Класс ConcreteProduct связан связью отношения наследования с абстрактным классом Product.

Класс ConcreteCreator связан связью отношения наследования с абстрактным классом Creator и связью отношения зависимости с классом порождаемого продукта ConcreteProduct.

Отношения между объектами

Класс Creator предоставляет своим производным классам ConcreteCreator возможность самостоятельно выбрать вид создаваемого продукта, посредством реализации метода FactoryMethod.

Мотивация

Предлагается рассмотреть простейший каркас (Framework) предоставляющий набор функциональности для построения приложений способных работать с несколькими документами. В таком фреймворке есть смысл выделить две основных абстракции – абстрактные классы Document и Application.

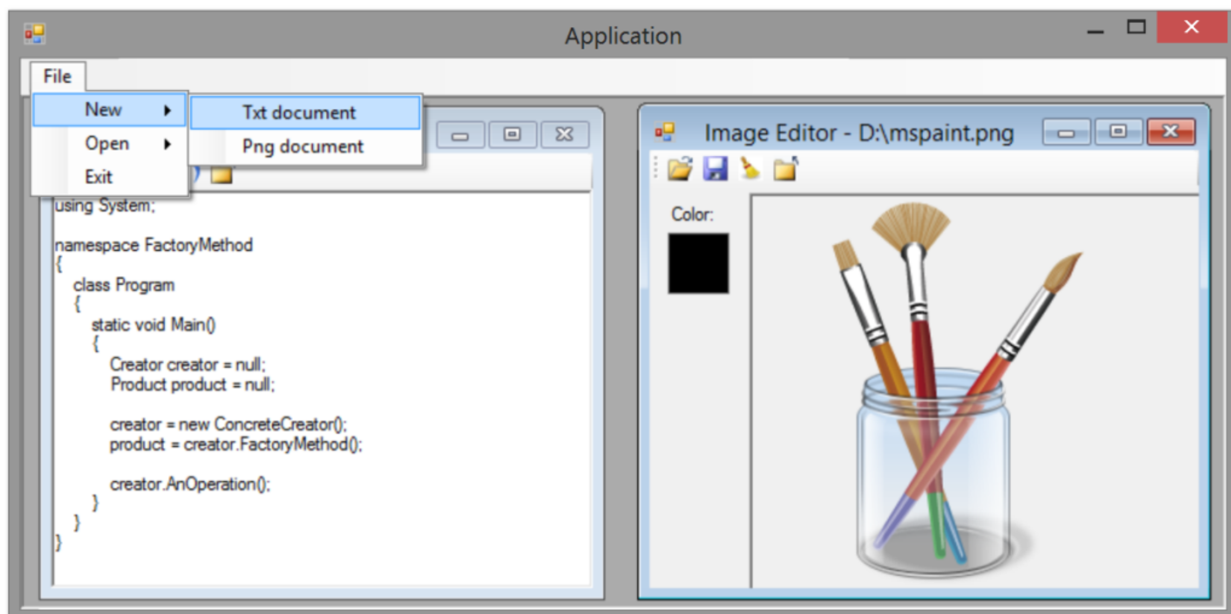
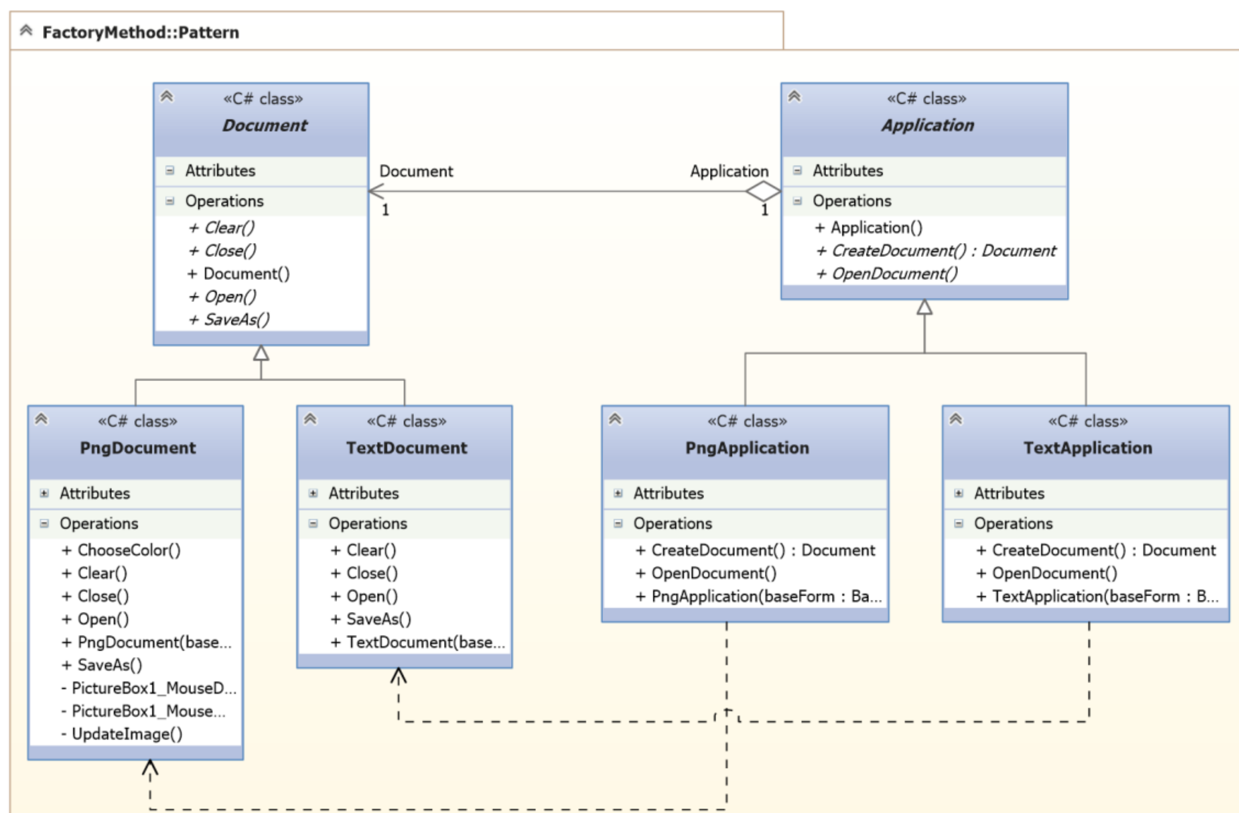


Рисунок 22- Пример приложения

Чтобы создать приложение, которое могло бы работать как с графикой, так и с текстом, предлагается создать классы Application и Document. Класс Application отвечает за создание и открытие документов, а то с каким документом будет производиться работа, определяется его подклассами PngApplication и TextApplication.

Таким образом, можно предоставить пользователю возможность пользоваться пунктами меню: **File.Open** и **File.New** для открытия и создания файлов с расширениями ***.png** и ***.txt**. Файлы, с которыми сможет работать приложение, представлены подклассами PngDocument и TextDocument класса Document. Решение с каким документом будет производиться работа принимается пользователем и абстрактный класс Application не может заранее спрогнозировать, экземпляр какого конкретного класса-документа (PngDocument или TextDocument) потребуется создать. Паттерн Factory Method предлагает элегантное решение такой задачи. Подклассы класса Application реализуют абстрактный фабричный метод CreateDocument, таким образом, чтобы он возвращал ссылку на экземпляр класса требуемого документа (PngDocument или TextDocument). Как только создан экземпляр одного из классов PngApplication или TextApplication, этот экземпляр можно использовать для создания документа определенного типа через вызов на этом экземпляре реализации фабричного метода CreateDocument. Метод CreateDocument называется «фабричным методом» или «виртуальным конструктором», так как он отвечает за непосредственное изготовление объекта-продукта.



Паттерн Фабричный Метод рекомендуется использовать, когда:

- Абстрактному базовому классу Creator заранее неизвестно, экземпляры каких конкретных классов-продуктов потребуется создать.

- Абстрактный класс Creator спроектирован таким образом, чтобы объекты-продукты, которые потребуется создать, описывались производными от него классами (ConcreteCreator).

- Процесс создания продукта должен обеспечивать возможность получения различных вариаций создаваемого продукта.

- Абстрактный класс Creator планирует передать ответственность за создание объектов-продуктов одному из своих подклассов.

- Требуется собрать в одном месте (в группе наследников) всех ConcreteCreators ответственных за создание объектов-продуктов определенного типа.

Результаты

Использование фабричных методов избавляет от необходимости использования конструкторов экземпляров для создания объектов-продуктов непосредственно в месте использования этих объектов-продуктов. Таким образом имеется возможность работать с абстрактным интерфейсом класса Product, что в свою очередь позволяет работать с любыми продуктами конкретных классов ConcreteProduct производных от Product.

Особенности применения паттерна Factory Method:

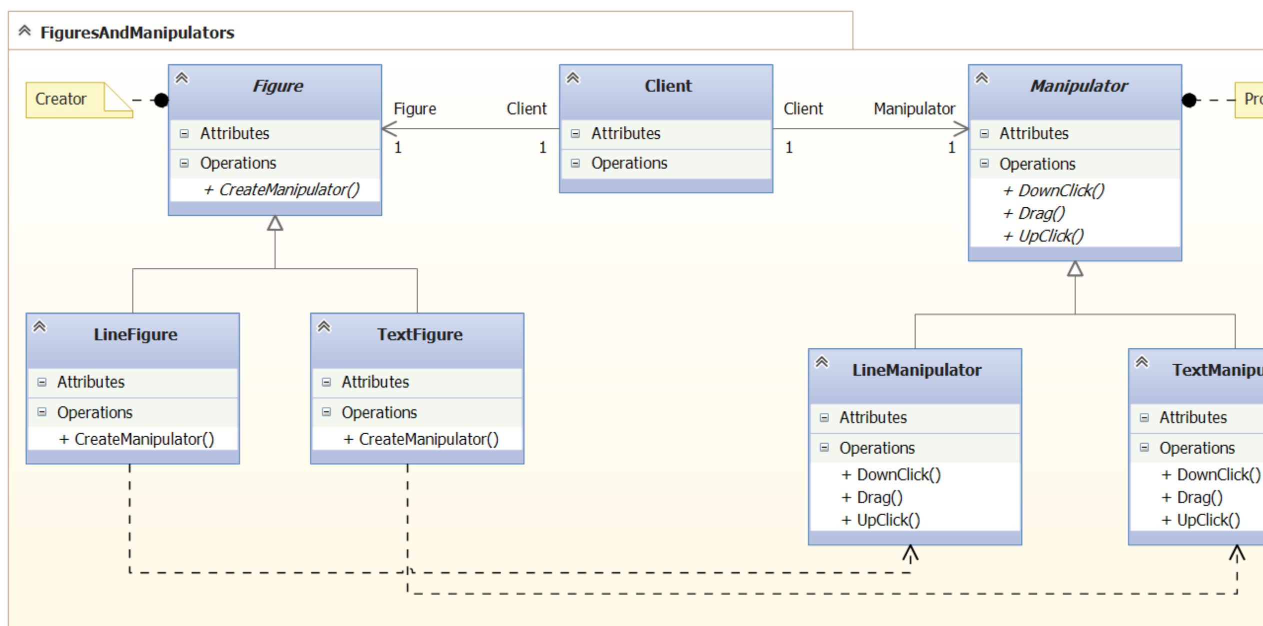
- Предоставление производным классам ConcreteCreator **операций-зацепок (hooks).**

Создание объектов через использование фабричных методов даже внутри класса оказывается более гибким решением, чем через вызов конструкторов непосредственно. Паттерн Factory Method позволяет использовать в производных классах ConcreteProduct операции-зацепки для построения расширенных версий объектов-продуктов.

- Соединение параллельных иерархий и контроль над ними.
Принципы рефакторинга рекомендуют избегать наличия в программном коде параллельных иерархий, но зачастую использование параллельных иерархий может быть хорошим решением. Из этого следует правило: параллельные иерархии в программном коде вполне допустимы в том и только в том случае, если они находятся в контексте паттерна проектирования и их использование регулируется содержащим их паттерном. Неконтролируемое (свободное) использование параллельных иерархий является запахом *(smell)* и требует рефакторинга.

Параллельные иерархии организуются тогда, когда классу «A» требуется возложить часть своих обязанностей на другой класс «B» не являющимся производным от класса «A».

Например, имеется приложение, в котором требуется создавать и перемещать графические фигуры при помощи мыши. Приходится сохранять и обновлять информацию о текущем состоянии манипуляций. Но такая информация требуется только в момент перемещения фигуры, поэтому помещать информацию о перемещении в саму фигуру иррационально. В таком случае для каждого типа фигур правильно использовать отдельный объект класса Manipulator (манипулятор фигурой). Разные фигуры, будут иметь разные манипуляторы, являющиеся подклассами класса Manipulator. Соответствия *«фигура A - манипулятор A»*, *«фигура B - манипулятор B»* и т.д., - представляют собой пример параллельной иерархии. Соответствия *«фигура A - манипулятор A»*, *«фигура B – манипулятор отсутствует»* и т.д., - представляют собой пример частично параллельной иерархии. На диаграмме классов видно, что иерархия классов фигур Figure параллельна иерархии классов манипуляторов Manipulator.



Важно понимать, что именно фабричные методы (в том числе и в данном примере) определяют соответствия между объектами входящими в параллельные иерархии. Именно фабричные методы содержат механизм порождения необходимых объектов, из которых будет составлена параллельная иерархия. Такой подход является надежным и безопасным при эксплуатации параллельных иерархий.

Реализация

Полезные приемы реализации паттерна Фабричный Метод:

Три основных разновидности паттерна.

1. Класс Creator является абстрактным и содержит только абстрактные фабричные методы. В этом случае требуется создание производных классов в которых будет реализован абстрактный фабричный метод из базового класса.

2. Класс Creator является конкретным классом и содержит реализацию фабричного метода по умолчанию. В этом случае фабричный метод используется главным образом для повышения гибкости. Выполняется правило, которое требует создавать объекты-продукты в фабричном методе, чтобы в производных классах была возможность заместить или переопределить способ создания объектов-продуктов. Такой подход гарантирует, что производным классам будет предоставлена возможность порождения объектов-продуктов требуемых классов.

3. Класс Creator является абстрактным и содержит реализацию фабричного метода по умолчанию.

Фабричные методы с параметрами.

Допускается создавать фабричные методы принимающие аргументы. Аргумент фабричного метода определяет вид создаваемого объекта-

продукта. Переопределение фабричного метода с аргументами, позволит изменять и конфигурировать изготавливаемые продукты.

Языково-зависимые особенности.

Разные языки программирования могут иметь в своем составе свои уникальные конструкции и техники, с использованием которых можно интересным образом выразить идеи использования паттерна – Фабричный Метод.

В языке C#, фабричные методы могут быть виртуальными или абстрактными (исключительно виртуальными). Нужно осторожно подходить к вызову виртуальных методов в конструкторе класса Creator.
Следует помнить, что в C# невозможно реализовать абстрактный метод базового абстрактного класса как виртуальный в производном классе. Абстрактные методы интерфейсов (interface) допустимо реализовывать как виртуальные.

После переопределения (override) виртуального метода в производном классе ConcreteCreator, виртуальные методы базового класса Creator становятся недоступными для их вызова на экземпляре класса ConcreteCreator (неважно, было привидение к базовому типу или нет). Если виртуальный метод вызывается в конструкторе класса Creator, а переопределенный (override) метод в конструкторе ConcreteCreator, то при создании экземпляра класса ConcreteCreator, в первую очередь отработает конструктор базового класса Creator, в котором произойдет вызов переопределенного метода из производного класса, а невиртуальный метод базового класса Creator. В случае замещения виртуального метода такой эффект отсутствует.

Обойти эту особенность возможно через использование функции доступа GetProduct, которая создает продукт по запросу, а с конструктора снять обязанность по созданию продуктов. Функция доступа возвращает продукт, но сперва проверяет его существование. Если продукт еще не создан, то функция доступа его создает (через вызов фабричного метода). Такую технику часто называют отложенной (или ленивой) инициализацией.

Использование обобщений (Generics).

Иногда приходится создавать конкретные классы создателей ConcreteCreator производные от базового класса Creator только для того чтобы создавать объекты-продукты определенного вида. Чтобы изменить подход порождения продуктов, в языке C#, можно воспользоваться такими конструкциями языка, как обобщения (Generics). Для организации процесса порождения продукта можно использовать технику – Service Locator.

При использовании обобщений (Generics), порождать несколько подклассов ConcreteCreator от класса Creator не потребуется, достаточно при создании экземпляра продукта в качестве параметра-типа фабричного метода CreateProduct указать желаемый тип порождаемого продукта.

```
ICreator creator = new StandardCreator();
IProduct productA = creator.CreateProduct<ProductA>();
IProduct productB = creator.CreateProduct<ProductB>();
IProduct productC = creator.CreateProduct<ProductC>();
```

Соглашения об именовании.

На практике рекомендуется давать такие имена фабричным методам, чтобы можно было легко понять, что используется именно фабричный метод. Например, фабричный метод порождающий документы мог бы иметь имя CreateDocument, где в имя метода входит название производимого действия Create и название того Document что создается.

Пример кода игры «Лабиринт»

Рассмотрим класс MazeGame, который использует фабричные методы. Фабричные методы создают объекты лабиринта: комнаты, стены, двери. В отличие от работы с абстрактной фабрикой, методы: MakeMaze, MakeRoom, MakeWall, MakeDoor – содержатся непосредственно в классе MazeGame.

```
class MazeGame
{
    // Использование Фабричных методов.
    public Maze CreateMaze()
    {
        Maze aMaze = this.MakeMaze();

        Room r1 = MakeRoom(1);
        Room r2 = MakeRoom(2);
        Door theDoor = MakeDoor(r1, r2);

        aMaze.AddRoom(r1);
        aMaze.AddRoom(r2);

        r1.SetSide(Direction.North, MakeWall());
        r1.SetSide(Direction.East, theDoor);
        r1.SetSide(Direction.South, MakeWall());
        r1.SetSide(Direction.West, MakeWall());

        r2.SetSide(Direction.North, MakeWall());
        r2.SetSide(Direction.East, MakeWall());
        r2.SetSide(Direction.South, MakeWall());
        r2.SetSide(Direction.West, theDoor);

        return aMaze;
    }

    public virtual Maze MakeMaze()
    {
        return new Maze();
    }

    public virtual Room MakeRoom(int number)
    {
        return new Room(number);
    }

    public virtual Wall MakeWall()
    {
        return new Wall();
    }

    public virtual Door MakeDoor(Room r1, Room r2)
    {
        return new Door(r1, r2);
    }
}
```

Для того чтобы сделать игру более разнообразной можно ввести специальные варианты частей лабиринта (EnchantedRoom – волшебная комната, DoorNeedingSpell – дверь, требующая заклинания, RoomWithBomb – комната с бомбой, BombedWall – взорванная стена).

```
// Класс заклинания необходимый для
// функционирования лабиринта с заклинаниями.
class Spell
{
    public Spell()
    {
        Console.WriteLine("Заклинание...");
    }
}

// Класс волшебная комната.
class EnchantedRoom : Room
{
    // Поля.
    private Spell spell = null;

    // Конструкторы.

    public EnchantedRoom(int roomNo)
        : base(roomNo)
    {
    }

    public EnchantedRoom(int number, Spell spell)
        : base(number)
    {
        this.spell = spell;
    }
}
```

```csharp
// Класс заклинания необходимый для
// функционирования лабиринта с заклинаниями.
class Spell
{
    public Spell()
    {
        Console.WriteLine("Заклинание...");
    }
}

// Класс волшебная комната.
class EnchantedRoom : Room
{
    // Поля.
    private Spell spell = null;

    // Конструкторы.

    public EnchantedRoom(int roomNo)
        : base(roomNo)
    {
    }

    public EnchantedRoom(int number, Spell spell)
        : base(number)
    {
        this.spell = spell;
    }
}
```

```
// Класс двери для которой требуется заклинание.
class DoorNeedingSpell : Door
{
    // Конструктор.
    public DoorNeedingSpell(Room room1, Room room2)
        : base(room1, room2)
    {
    }
}

// Класс комнаты с бомбой.
class RoomWithBomb : Room
{
    // Конструктор.
    public RoomWithBomb(int roomNo)
        : base(roomNo)
    {

    }
}

// Класс взорванной стены.
class BombedWall : Wall
{
}
```

Подклассы EnchantedMazeGame и BombedMazeGame класса MazeGame, скрывают работу с такими специфическими классами как: EnchantedRoom, DoorNeedingSpell, RoomWithBomb, BombedWall. Использование фабричных методов позволяет в подклассах класса MazeGame:

EnchantedMazeGame, BombedMazeGame –выбирать различные варианты объектов-продуктов для построения лабиринта.

```
class EnchantedMazeGame : MazeGame
{
    // Конструктор лабиринта с заклинаниями.
    public EnchantedMazeGame()
    {
    }

    // Методы.
    public override Room MakeRoom(int number)
    {
        return new EnchantedRoom(number, this.CastSpell());
    }

    public override Door MakeDoor(Room r1, Room r2)
    {
        return new DoorNeedingSpell(r1, r2);
    }

    // Метод создания заклинания.
    protected Spell CastSpell()
    {
        return new Spell();
    }
}

class BombedMazeGame : MazeGame
{
    // Конструктор лабиринта с бомбами.
    public BombedMazeGame()
    {
    }

    // Методы.
    public override Wall MakeWall()
    {
        return new BombedWall();
    }

    public override Room MakeRoom(int number)
    {
        return new RoomWithBomb(number);
    }
}
```

Известные применения паттерна в .Net

Паттерн Factory Method лежит в основе всех порождающих паттернов, соответственно он используется везде где можно увидеть применение порождающих паттернов.

См. пункты «Известные применения паттерна в .Net» других порождающих паттернов.

Паттерн Prototype

Название Прототип

Также известен как

-

Классификация

По цели: порождающий

По применимости: к объектам

Частота использования

Средняя - 1 2 **3** 4 5

Назначение

Паттерн Prototype – предоставляет возможность создания новых объектов-продуктов (клонов), используя технику клонирования (копирования) созданного ранее объекта-оригинала-продукта (прототипа). Паттерн Prototype – позволяет задать различные виды (классы-виды) объектов-продуктов (клонов), через настройку состояния каждого нового созданного клона. Классификация клонов-продуктов производится на основании различия их состояний.

Введение

Паттерн Prototype описывает процесс правильного создания объектов-клонов на основе имеющегося объекта-прототипа, другими словами, паттерн Prototype описывает правильные способы организации процесса клонирования.

Что такое клонирование в объективной реальности? В биологии термин «клонирование» - обозначает процессы копирования любых живых существ. Но, в информатике термин «клонирование» имеет свое специфическое значение, отличное от его значения в биологии. В биологии процесс клонирования происходит путем создания не готовой копии взрослого организма, а путем выращивания взрослого клона из младенца. Изначально клон-младенец порождается на основе генетического материала организма-прототипа, и развивается до взрослого состояния поэтапно, согласно заложенной в нем генетической программы развития. В информатике же клонирование происходит «мгновенно». В результате клонирования прототипа в информатике, получается сразу «взрослый» клон полностью идентичный прототипу.

В жизни программному клонированию аналогии не найти, так как биологический организм-клон никогда не будет идентичен своему организму-прототипу. Нарушение идентичности происходит за счет вмешательства в развитие клона внешних факторов среды его обитания. С потерей идентичности у биологического клона развивается ярко выраженная индивидуальность. В информатике же, легко добиться идентичности клона и прототипа за счет возможности копирования всего состояния прототипа в клон.

Но можно отойти от подхода к клонированию со строгим контролем идентичности и представить клона, как систему, созданную по образцу другой. Клон должен сохранять основные свойства исходной системы-

прототипа не разрушая абстракции (генотипа) прототипа, то есть собирательного понятия исходной системы. Индивидуальные черты (фенотип), которые в процессе развития приобретает клон, не должны разрушать абстракцию (генотип) прототипа и такими индивидуальными чертами клона можно пренебречь.

Например, при клонировании овцы Долли, сама Долли и овца-донор имели стопроцентную идентичность генотипа, но не имели стопроцентной идентичности фенотипа. Другими словами, и донор-овца и клон-овца были овцами-близнецами одной породы (сохранение генотипа), а не овцами разных пород, но у них мог отличаться вес, рост, оттенок шерсти, а также приобретенный опыт определяющий индивидуальное поведение (разность фенотипов).

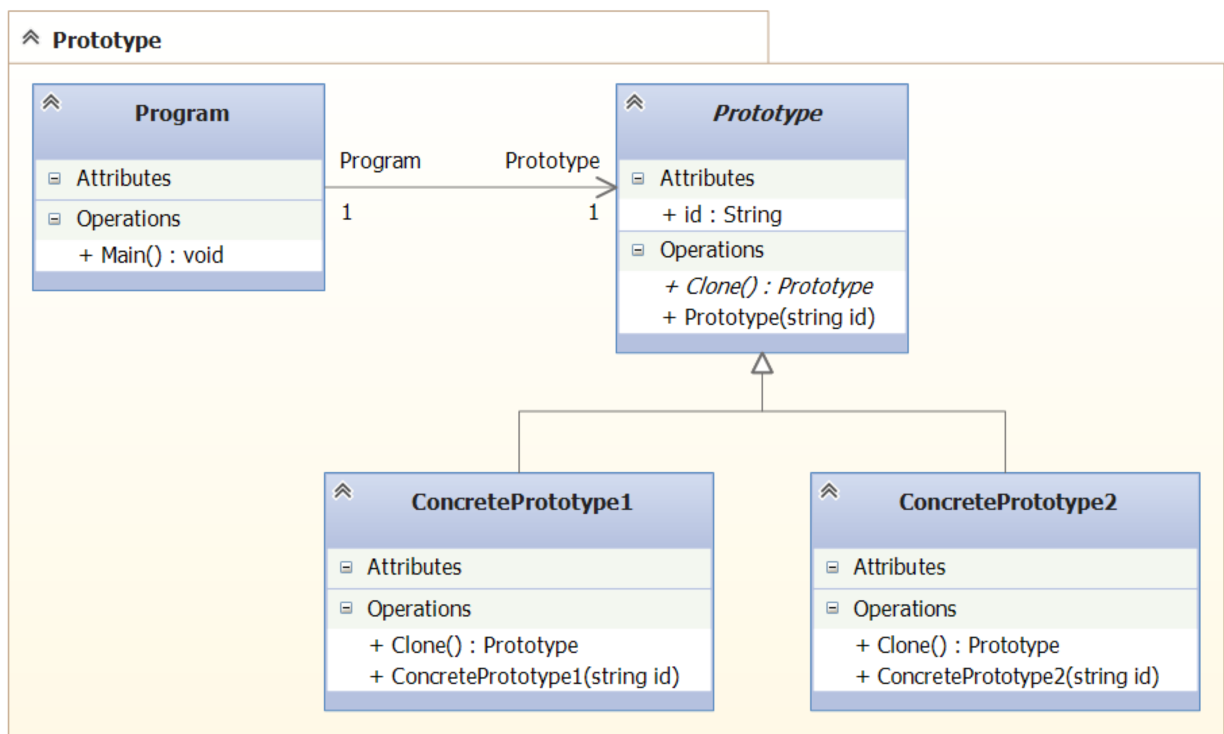


Рисунок 23 - Паттерн Prototype

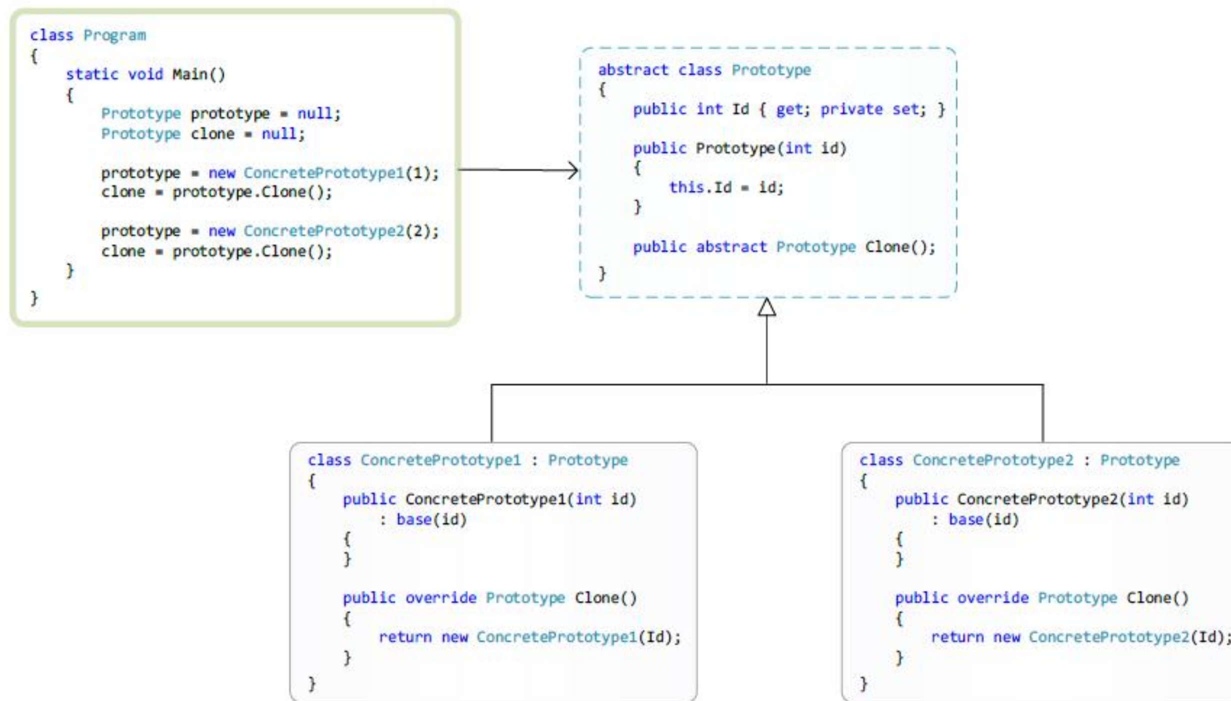


Рисунок 24 - Паттерн Prototype на языке C#

Участники

Prototype - **Прототип**:

Предоставляет интерфейс для клонирования себя.

ConcretePrototype - **Конкретный прототип**:

Реализует операцию клонирования себя.

Client - **Клиент**:

Клиент создает экземпляр прототипа. Вызывает на прототипе метод клонирования.

Отношения между участниками

Отношения между классами

Класс ConcretePrototype связан связью отношения наследования с абстрактным классом Prototype.

Отношения между объектами

Клиент вызывает на экземпляре-прототипе метод Clone и этот метод создает экземпляр-клон прототипа и возвращает ссылку на него.

Мотивация

Предлагается построить простейший нотный редактор, позволяющий набирать, редактировать и проигрывать нотный текст. Редактор должен содержать партитуру, нотные станы, ноты и другие музыкальные объекты.

Ввод нот может происходить при помощи мыши путем выбора ноты в палитре нот и перемещения выбранной ноты на нотный стан.

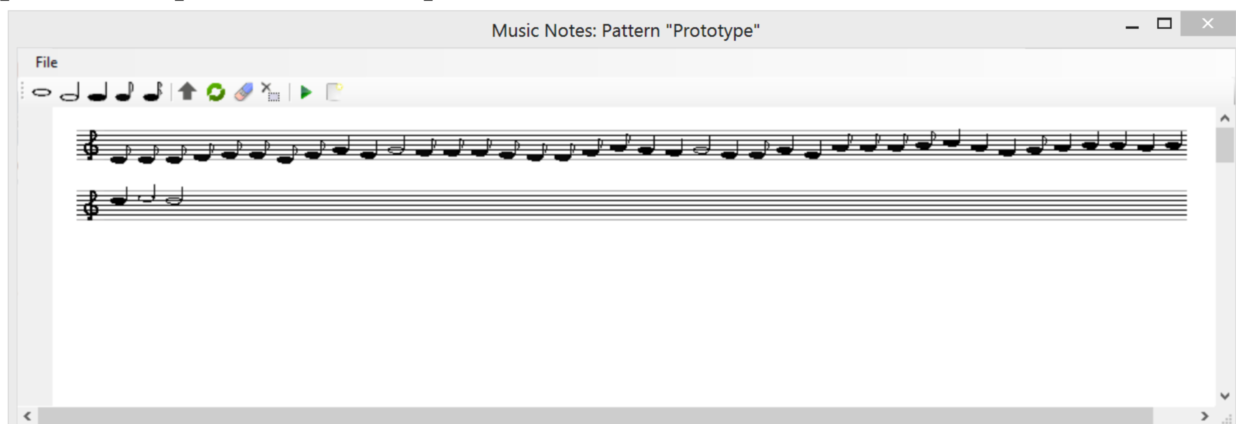
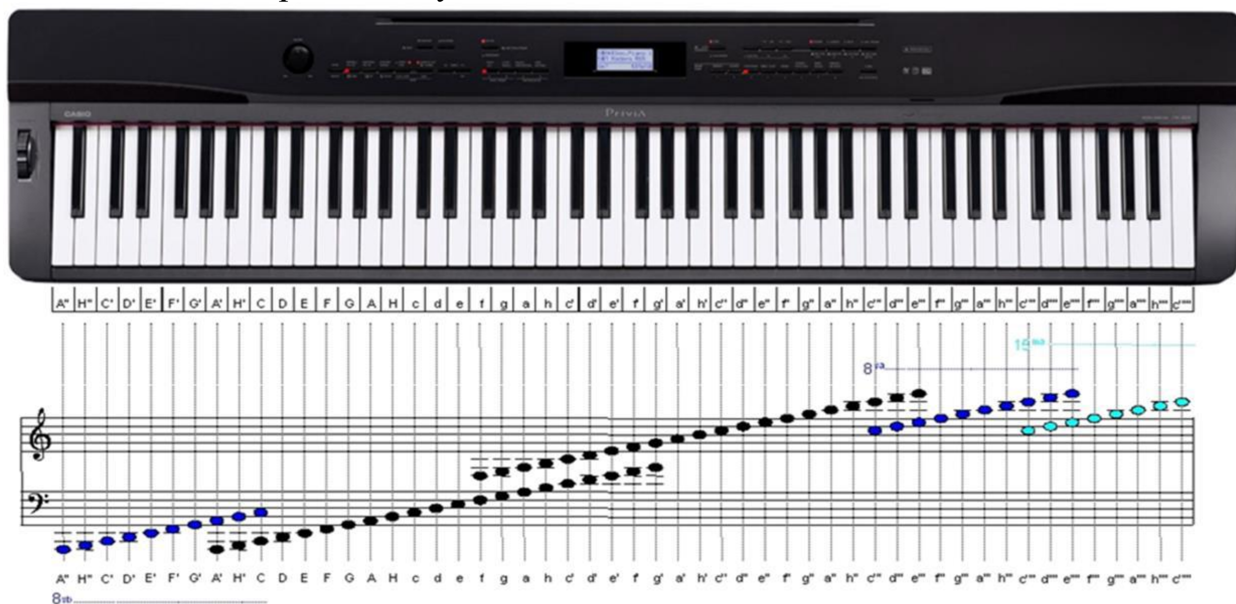


Рисунок 25 – Пример приложения с использованием паттерна Prototype

Нота представляет собой графическое изображение звука. Нотная гамма содержит семь нот: «до», «ре», «ми», «фа», «соль», «ля», «си». Ноты распределяются по октавам.

Например, у пианино девять октав: семь полных и две неполных. В каждой октаве семь белых клавиш и пять черных, итого двенадцать клавиш в октаве. Каждая нота имеет длительность звучания: 1, 1/2, 1/4, 1/8, 1/16, 1/32 и 1/64. Соответственно с использованием пианино можно воспроизвести очень большое количество вариаций звуков.



Таким образом, может показаться, что потребуется создать по одному классу (class) для каждой вариации воспроизводимого звука. Но такой подход является нерациональным и неприемлемым. Более правильным решением проблемы - будет создание ноты-прототипа для ее последующего клонирования с возможностью настройки звучания (свойств) каждой ноты-клона.

Для того чтобы создать приложение с графическим интерфейсом, которое позволяло бы создавать и редактировать музыкальные композиции в наглядной форме, предлагается создать абстрактные классы Graphic и Tool. Класс Graphic служит для графического представления нот и нотных станов на форме приложения, а класс Tool определяет функциональные возможности для работы с графическими представлениями, такие как создание графического представления каждой ноты в зависимости от характеристик ноты (реализуется конкретным подклассом GraphicTool класса Tool), размещение его на нотном стане (реализуется конкретным подклассом RelocationTool класса Tool) и ориентирование штиля ноты (реализуется подклассом RotateTool класса Tool).

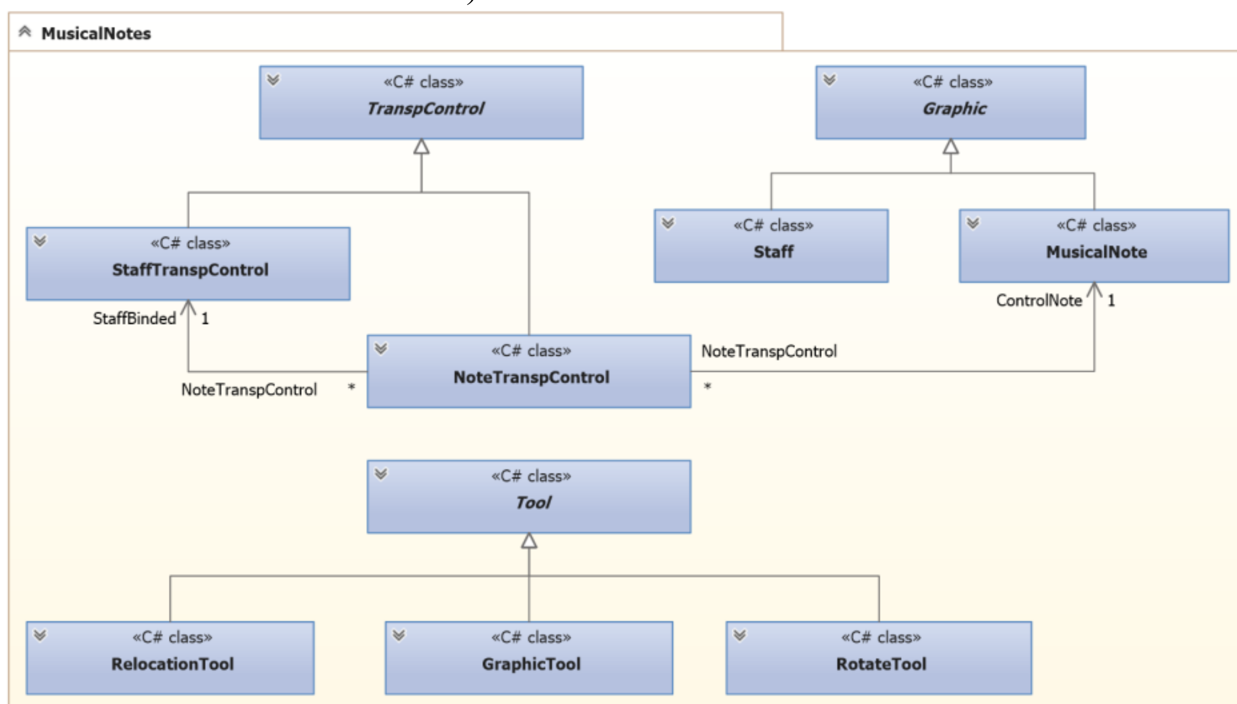


Рисунок 26 –Возможные классы приложения

Чтобы обеспечить гибкость и рациональность разработки приложения, класс GraphicTool будет создавать экземпляры классов MusicalNote (класс ноты) и Staff (класс музыкального стана), которые являются наследниками абстрактного класса Graphic и реализуют метод Clone базового класса путем клонирования прототипов экземпляров этих классов, переданных в качестве аргумента конструктора класса, и затем добавлять экземпляры-клоны в партитуру.

Применимость паттерна

Паттерн Prototype рекомендуется использовать, когда:

- Программист не должен знать, как в системе создаются, компонуются и представляются объекты-продукты.

- Классы, экземпляры которых требуется создать, определяются во время выполнения программы, например, с использованием техники позднего связывания (динамическая загрузка DLL).

- Требуется избежать построения параллельных иерархий классов (фабрик или продуктов).

- Экземпляры определенного класса могут иметь небольшое количество состояний. Тогда может оказаться удобнее установить прототип в соответствие каждому уникальному состоянию и в дальнейшем клонировать прототипы, а не создавать экземпляры вручную и не настраивать состояние этих экземпляров каждый раз заново.

- Требуется создать некоторое количество экземпляров с одинаковым долго-вычисляемым состоянием. В этом случае для повышения производительности удобнее создать прототип и его клонировать, причем с использованием метода MemberwiseClone класса System.Object входящего в поставку Microsoft .NET Framework. Клонирование с использованием конструктора повышения производительности не даст.

Результаты

Многие особенности применения паттерна Prototype совпадают с особенностями применения паттернов Abstract Factory и Builder:

- Сокрытие работы с конкретными классами продуктов.
- Возможность легкой замены семейств используемых продуктов.
- Обеспечение совместного использования продуктов.
- Имеется небольшое неудобство добавления нового вида продуктов.
- Возможность изменения состава продукта.
- Сокрытие кода, реализующего конструирование и представление.
- Контроль над процессом построения продукта.

Дополнительные особенности результатов применения паттерна Prototype:

- **Возможность добавления и удаления продуктов во время выполнения.**

Паттерн Prototype позволяет добавлять новую разновидность «классов-продуктов» во время выполнения программы за счет конфигурирования состояния объекта-клона полученного в результате клонирования объекта-прототипа. «Класс-продукт» следует понимать не как конструкцию языка C# (class), а как «вид-продукт» получившийся за счет конфигурации состояния объекта-клона.

- Описание новых типов объектов (объектов-классов) путем изменения состояния клонов.

Программные системы позволяют формировать состояние и поведение сложных объектов-клонов за счет композиции состояний и поведений более простых объектов. При этом для представления сложного состояния объекта, не требуется создавать новых классов (class).

- **Уменьшение числа производных классов.**

Большинство порождающих паттернов используют иерархии классов продуктов параллельные иерархиям классов фабрик. Паттерн Prototype через клонирование и настройку объектов-продуктов позволяет избавиться от наличия иерархии фабрик, порождающих продукты.

Динамическое добавление классов во время выполнения программы.

Платформа .Net позволяет динамически подключать новые классы к исполняющемуся приложению и создавать экземпляры этих классов (late binding - позднее связывание). Приложение, создающее экземпляры динамически загружаемого класса, не имеет возможности вызывать конструктор напрямую. Вместо этого на классе-объекте Activator вызывается метод CreateInstance, которому в качестве аргумента передается строковое представление полного квалифицированного имени класса, экземпляр которого требуется создать, например, Activator.CreateInstance("MyNamespace.MyClass"). Паттерн Prototype в определенных случаях может стать гибкой альтернативой позднему связыванию избавляя от многократного инстанцирования классов, подключаемых динамически. Достаточно создать один экземпляр-прототип и в дальнейшем его клонировать. Такой подход может дать выигрыш в производительности при необходимости создания большого количества однотипных экземпляров классов.

При клонировании могут возникнуть проблемы корректного клонирования ассоциаций. Особое внимание требуется обращать на клонирование двухсторонних ассоциаций.

Реализация:

Использование паттерна Prototype может оказаться полезным в статически типизированных языках вроде C#, где классы не являются объектами. Меньше всего паттерн Prototype может оказаться полезным в прототипно-ориентированных языках вроде JavaScript, так как в основе таких языков уже заложена конструкция эквивалентная прототипу, это – «объект-класс». В прототипно-ориентированных языках создание любого объекта производится путем клонирования прототипа и такие языки базируются именно на идеях использования паттерна Prototype.

Вопросы, которые могут возникнуть при реализации прототипов:

Использование диспетчера прототипов.

Если в программе требуется создавать большое число прототипов и при этом заранее не известно сколько прототипов может понадобиться, то есть смысл использовать вспомогательный объект для создания прототипов и контроля над ними. Такой объект будет представлять собой реестр прототипов или его можно назвать - «диспетчер прототипов». Реестр прототипов – представляет собой ассоциативную коллекцию типа хеш-

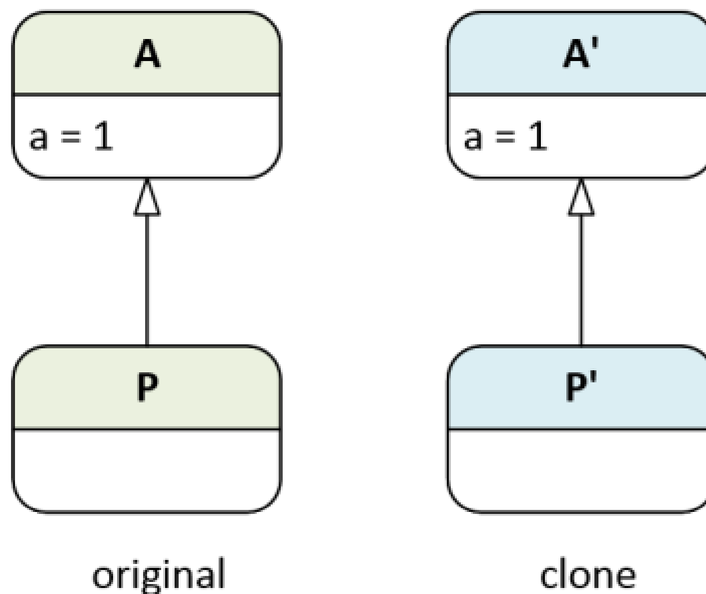
таблицы или словаря, в которой в соответствие ключам поставлены объекты-прототипы. Реестр может возвращать ссылку на прототип или непосредственно на построенный клон прототипа, это зависит от желаемой логики.

Реализация метода Clone**.**

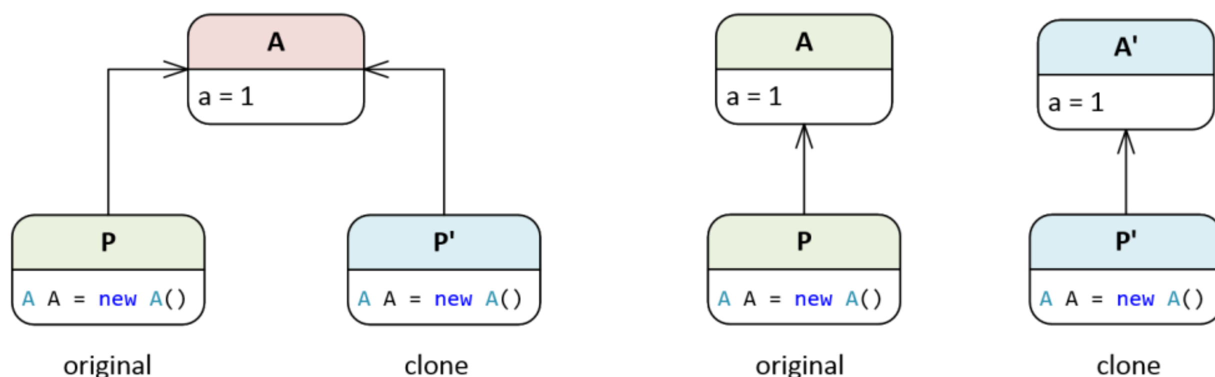
Самый ответственный момент при использовании паттерна Prototype – реализация метода Clone. Особое внимание потребуется при наличии ассоциаций в клонируемом прототипе.

Паттерн Prototype, выражен в платформе .Net в виде техники клонирования, через использование реализации интерфейса ICloneable или метода MemberwiseClone класса System.Object. Но метод MemberwiseClone не решает проблему «глубокого и поверхностного клонирования». При клонировании с использованием метода MemberwiseClone следует учитывать следующие особенности:

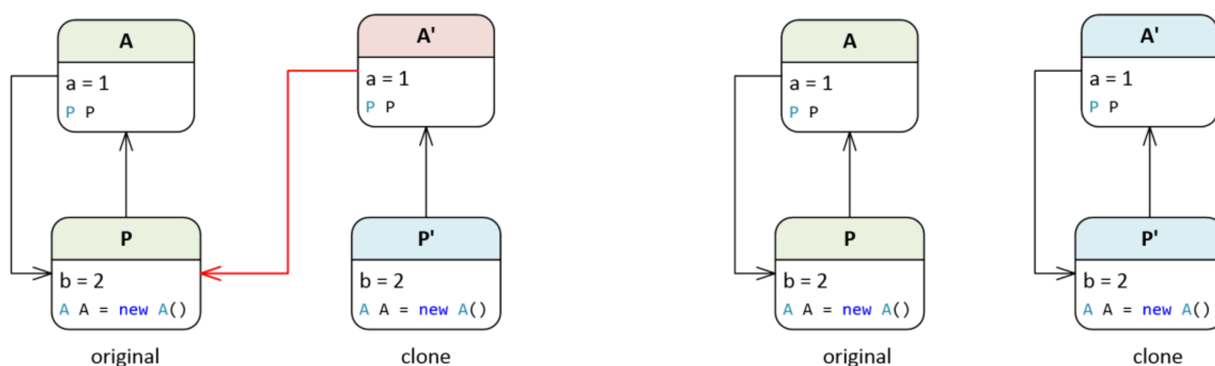
Граф наследования всегда клонируется глубоко.



Могут возникнуть проблемы корректного клонирования ассоциаций. Так как ассоциации клонируются поверхностно. Перед клонированием следует ответить на вопрос: должны ли при клонировании объекта клонироваться также и другие объекты на которые ссылаются переменные клонируемого объекта, или клон будет использовать эти объекты совместно с оригиналом?



Особо внимание требуется обратить на клонирование двухсторонних ассоциаций.



Инициализация клонов.

Часто, сразу после клонирования, требуется производить настройку (изменение) состояния объекта-клона (полностью или частично). Можно задуматься над тем, чтобы передавать параметры в метод Clone перед началом клонирования, но такой подход считается неудачным, так как мешает построению единообразного интерфейса клонирования. Для таких случаев, будет целесообразно ввести набор методов для установки или очистки важных элементов состояния, или ввести один метод Initialize, который будет принимать параметры для установки состояния клона.

Пример кода игры «Лабиринт»

Определим класс MazePrototypeFactory, производный от класса MazeFactory, который был создан при рассмотрении паттерна Abstract Factory. Конструктор класса MazePrototypeFactory в качестве параметров будет принимать прототипы объектовклассов Maze, Wall, Room и Door,из которых состоитлабиринт.

Класс MazePrototypeFactory будет создавать готовые объекты на основе соответствующих прототипов путем клонирования этих прототипов, для этого переопределим в нем фабричные методы MakeMaze, MakeRoom, MakeWall и MakeDoor из базового класса MazeFactory.

```
class MazePrototypeFactory : MazeFactory
{
    // Поля.
    Maze prototypeMaze = null;
    Room prototypeRoom = null;
    Wall prototypeWall = null;
    Door prototypeDoor = null;

    // Конструктор.
    public MazePrototypeFactory(Maze maze, Wall wall, Room room, Door door)
    {
        this.prototypeMaze = maze;
        this.prototypeWall = wall;
        this.prototypeRoom = room;
        this.prototypeDoor = door;
    }

    // Методы.

    public override Maze MakeMaze()
    {
        return prototypeMaze.Clone();
    }

    public override Room MakeRoom(int number)
    {
        Room room = prototypeRoom.Clone();
        room.Initialize(number);

        return room;
    }

    public override Wall MakeWall()
    {
        // Клонирование.
        return prototypeWall.Clone();
    }

    public override Door MakeDoor(Room room1, Room room2)
    {
        Door door = prototypeDoor.Clone();
        door.Initialize(room1, room2);

        return door;
    }
}
```

Для создания экземпляра фабрики, позволяющей построить простой лабиринт, достаточно сконфигурировать объект класса MazePrototypeFactory объектами классов Maze, Wall, Room и Door, составляющими простой лабиринт. Для создания лабиринта требуется фабрику класса MazePrototypeFactory, передать в качестве аргумента метода CreateMaze класса MazeGame, который был создан при рассмотрении паттерна Abstract Factory.

```
// Создаем генератор лабиринта.
MazeGame game = new MazeGame();

//Конфигурируем фабрику базовыми элементами лабиринта
MazePrototypeFactory simpleMazeFactory =
     new MazePrototypeFactory(new Maze(), new Wall(), new Room(), new Door());

// Создаем лабиринт из двух комнат используя фабричный метод - CreateMaze().
Maze maze = game.CreateMaze(simpleMazeFactory);
```

Поскольку экземпляр класса MazePrototypeFactory может быть сконфигурирован экземплярами классов производных от классов Maze, Wall, Room и Door, то отсутствует необходимость порождать новые классы производные от класса MazePrototypeFactory для создания специальных видов лабиринтов.

Для того чтобы изменить тип простого лабиринта на лабиринт с бомбами требуется организовать следующий вызов:

```
// Конфигурируем фабрику специальными элементами лабиринта
MazePrototypeFactory bombedMazeFactory = new MazePrototypeFactory(new Maze(),
new BombedWall(), new RoomWithBomb(), new Door());
```

Объекты классов типов Maze, Wall, Room и Door, которые предполагается использовать в качестве прототипов, должны содержать в себе реализацию метода Clone.

```
class Door : MapSite
{
    // Поля.
    Room room1 = null;
    Room room2 = null;
    bool isOpen;

    // Конструкторы.
    public Door(){}

    public Door(Door other)
    {
        this.room1 = other.room1;
        this.room2 = other.room2;
    }

    public Door(Room room1, Room room2)
    {
        this.room1 = room1;
        this.room2 = room2;
    }

    // Методы.
    // Отображает дверь.
    public override void Enter()
    {
        Console.WriteLine("Door");
    }


    // Метод возвращает ссылку на другую комнату.

    public Room OtherSideFrom(Room room)
    {
        if (room == room1)  // Если идем из r1 в r2, то возвращаем ссылку на r2.
            return room2;
        else            // Иначе: Если идем из r2 в r1, то возвращаем ссылку на r1.
            return room1;
    }

    // Клонирование.
    public virtual Door Clone()
    {
        Door door = new Door(this.room1, this.room2);
        // При обращении к закрытым членам экземпляра, в пределах того-же класса
        // инкапсуляция не работает.
        door.isOpen = this.isOpen;
        return door;
    }

    // Метод инициализации...
    public virtual void Initialize(Room room1, Room room2)
    {
        this.room1 = room1;
        this.room2 = room2;
    }
}
```

В подклассах классов Maze, Wall, Room и Door, нужно переопределить метод Clone.

```
class BombedWall : Wall
{
    // Поля.
    bool bomb;

    // Конструкторы.
    public BombedWall(){}

    public BombedWall(BombedWall other)
    {
        this.bomb = other.bomb;
    }

    // Переопределение базовой операции клонирования.
    public override Wall Clone()
    {
        return new BombedWall(this);
    }

    public bool HasBomb()
    {
        return this.bomb;
    }
}
```

При таком определении метода Clone в базовых классах Maze, Wall, Room и Door, клиент избавляется от необходимости знать о конкретных реализациях операции клонирования в подклассах этих классов, т.к. клиенту никогда не придется приводить значение, возвращаемое Clone, к базовому типу.

Известные применения паттерна в .Net

System.Data.DataSet
http://msdn.microsoft.com/ru-ru/library/system.data.dataset.aspx
System.Array
http://msdn.microsoft.com/ru-ru/library/system.array.aspx
System.Collections.ArrayList
http://msdn.microsoft.com/ru-ru/library/system.collections.arraylist.aspx
System.Collections.BitArray
http://msdn.microsoft.com/ru-ru/library/system.collections.bitarray.aspx
System.Collections.Hashtable
http://msdn.microsoft.com/ru-ru/library/system.collections.hashtable.aspx
System.Collections.Queue
http://msdn.microsoft.com/ru-ru/library/system.collections.queue.aspx
System.Collections.SortedList
http://msdn.microsoft.com/ru-ru/library/system.collections.sortedlist.aspx
System.Collections.Stack
http://msdn.microsoft.com/ru-ru/library/system.collections.stack.aspx
System.Data.DataTable
http://msdn.microsoft.com/ru-ru/library/system.data.datatable.aspx

`System.Data.SqlClient.SqlCommand`
http://msdn.microsoft.com/ru-ru/library/system.data.sqlclient.sqlcommand(v=vs.100).aspx
`System.Delegate`
http://msdn.microsoft.com/ru-ru/library/system.delegate.aspx
`System.Reflection.AssemblyName`
http://msdn.microsoft.com/ru-ru/library/system.reflection.assemblyname.aspx
`System.Text.Encoding`
http://msdn.microsoft.com/ru-ru/library/system.text.encoding(v=vs.90).aspx
`System.Xml.XmlNode`
http://msdn.microsoft.com/ru-ru/library/system.xml.xmlnode.aspx
`System.Globalization.Calendar`
http://msdn.microsoft.com/ru-ru/library/system.globalization.calendar.aspx
`System.Globalization.DateTimeFormatInfo`
http://msdn.microsoft.com/ru-ru/library/system.globalization.datetimeformatinfo(v=vs.110).aspx
`System.Globalization.TextInfo`
http://msdn.microsoft.com/ru-ru/library/system.globalization.textinfo.aspx
`System.String`
http://msdn.microsoft.com/ru-ru/library/system.string.aspx

Паттерн Singleton

Название - Одиночка

Также известен как

Solitaire (Холостяк)

Классификация

По цели: порождающий

По применимости: к объектам

Частота использования

Выше средней - 1 2 3 **4** 5

Назначение

Паттерн Singleton - гарантирует, что у класса может быть только один экземпляр. В частном случае предоставляется возможность наличия, заранее определенного числа экземпляров.

Введение

В реальной жизни аналогией объекта в единственном экземпляре может служить Солнце. Как бы мы не смотрели на него, мы всегда будем видеть одно и то же Солнце – звезду, вокруг которой вращается планета Земля, так как не может быть другого экземпляра Солнца, по крайней мере, для нашей планеты.

Важно заметить, что аналогия с Солнцем не является полной в отношении паттерна Singleton. Так как ни один объект в реальной жизни не может

контролировать процесс своего порождения, а, следовательно, и существования себя в единственном экземпляре. Такая возможность появляется только у виртуальных объектов программных систем.

При разработке приложений бывает необходимо, чтобы существовал только один экземпляр определенного класса. Например, в играх от первого лица персонаж-анимат, которым управляет игрок, должен быть одним единственным, так как такой анимат, является образом-проекцией, живого игрока на игровой мир.

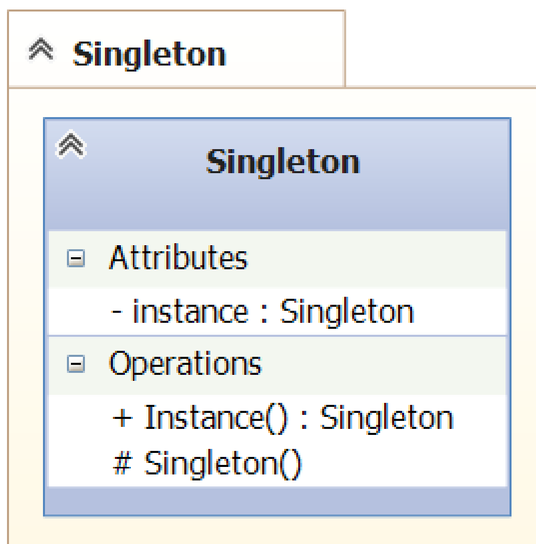


Рисунок 27 – Структура паттерна Singleton на языке UML

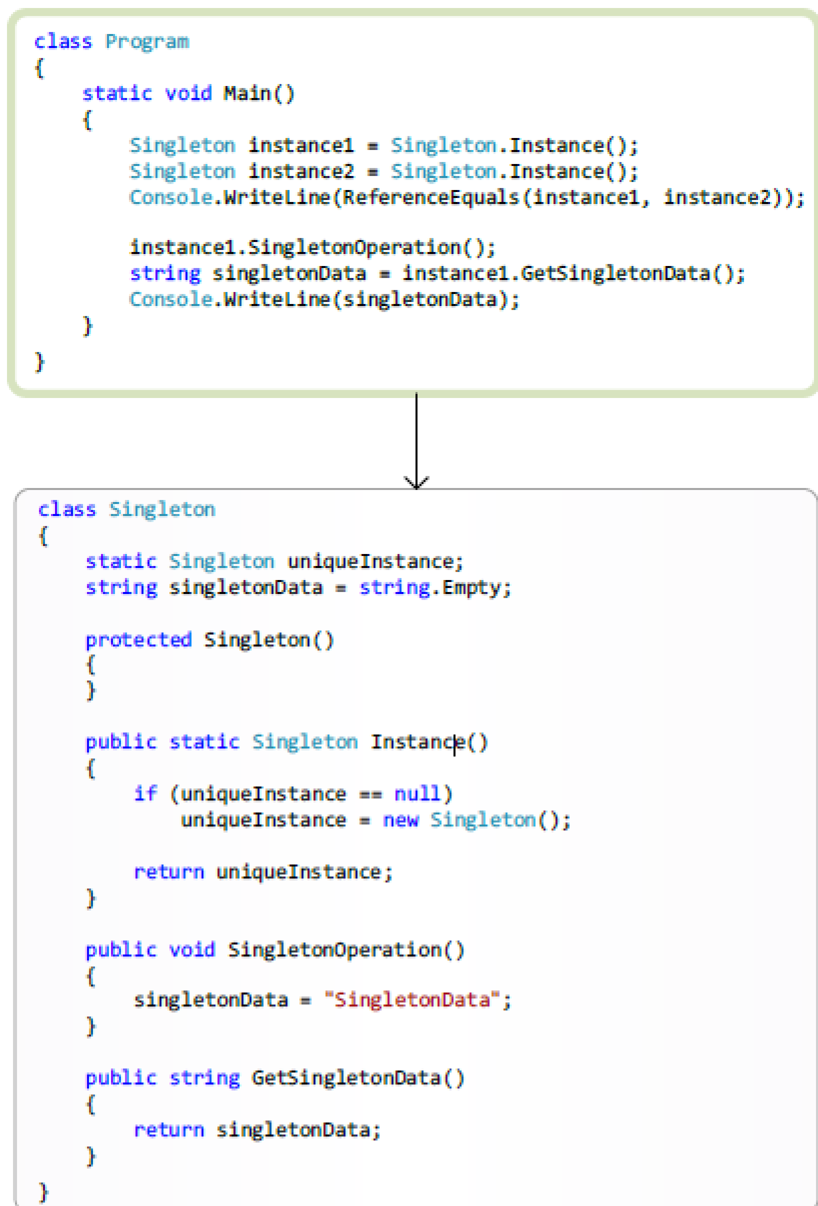


Рисунок 28 – Структура паттерна Singleton на языке C#

Участники

- Singleton - **Одиночка**:

Предоставляет интерфейс (метод Instance) для получения доступа к единственному экземпляру.

Отношения между участниками

Отношения между классами

- Класс Singleton не имеет обязательных связей отношений.

Отношения между объектами

- Клиент получает доступ к экземпляру через вызов метода Instance на классе-объекте Singleton.

Мотивация

Для некоторых классов требуется, чтобы у них мог существовать только один экземпляр их класса. Например, класс Console, который представляет собой объектно-ориентированное представление сущности – консоль, из объективной реальности. В объективной реальности консоль, это собирательное понятие, объединяющее два устройства – клавиатуру и экран монитора (устройства ввода/вывода информации).

Класс Console, является статическим классом, а это значит, что невозможно создать переменное количество его экземпляров. Поэтому можно сказать, что статические классы в C# являются одним из возможных выражений паттерна Singleton.

Применимость паттерна

Паттерн Singleton рекомендуется использовать, когда:

- В системе должен быть только один экземпляр некоторого класса, или в частном случае заранее определенное пользователем количество экземпляров (два, три и т.д.).

- Требуется организовать расширение класса единственного экземпляра через использование механизма наследования.

Результаты

Особенности применения паттерна Singleton:

- **Контроль доступа к единственному экземпляру.**

Процесс создания единственного экземпляра скрыт в классе Singleton, поэтому класс Singleton полностью контролирует доступ к экземпляру через использование метода Instance, который всегда возвращает ссылку на один и тот же экземпляр.

- **Возможность расширения через наследование.**

Если класс Singleton не является статическим или герметизированным / запечатанным (sealed), то от него возможно наследование, что позволит расширить существующую функциональность.

- **Возможность наличия переменного числа экземпляров.**

Паттерн Singleton позволяет создавать фиксированное число экземпляров класса Singleton, например, только два или три и т.д.

- **Большая гибкость чем у статических классов.**

Одним из вариантов реализации паттерна Singleton в C#, является использование статических классов. Но такой подход может в дальнейшем препятствовать изменению дизайна в том случае, если понадобится использование нескольких экземпляров класса Singleton. Кроме того, статические классы не сопрягаются с механизмами наследования и статические методы не могут быть виртуальными, что не допускает полиморфных отношений.

Реализация

Особенности, которые следует учесть при реализации паттерна Singleton:

- **Гарантия наличия единственного экземпляра.**

Паттерн Singleton устроен так, что при его применении не удастся создать более одного экземпляра определенного класса. Чаще всего для получения экземпляра используют статический фабричный метод Instance, который гарантирует создание не более одного экземпляра. Если метод Instance будет вызван впервые, то он создаст экземпляр при помощи защищенного конструктора и вернет ссылку на него, при этом сохранив эту ссылку в одном из своих полей. При всех последующих вызовах метода Instance, будет возвращаться ранее сохраненная ссылка на существующий экземпляр.

Наследование от Singleton.

Процесс создания производного класса SingletonDerived довольно прост, для этого достаточно заместить в нем статический метод Instance, в котором унаследованной из базового класса переменной instance присвоить ссылку на экземпляр производного класса SingletonDerived.

```
public new static DerivedSingleton Instance()
{
    if (uniqueInstance == null)
        uniqueInstance = new DerivedSingleton();

    return uniqueInstance as DerivedSingleton;
}
```

Если в программе имеется несколько производных классов-одиночек (DerivedSingleton1, DerivedSingleton2, …) и далее потребуется организовать

выбор использования определенного производного класса-одиночки, то для выбора нужного экземпляра-одиночки удобно будет воспользоваться *реестром одиночек*, который представляет собой ассоциативный массив (ключ-значение). В качестве ключа может выступать имя одиночки, а в качестве значения – ссылка на экземпляр определенного одиночки.

Потокобезопасный Singleton

Может возникнуть ситуация, когда из разных потоков происходит обращение к свойству Singleton Instance, которое должно вернуть ссылку на экземпляр-одиночку и при этом экземпляр-одиночка еще не был создан и его поле instance равно null. Таким образом, существует риск, что в каждом отдельном потоке будет создан свой экземпляр-одиночка, а это противоречит идеологии паттерна Singleton. Избежать такого эффекта возможно через инстанцирование класса Singleton в теле критической секции (конструкции lock), или через использование «объектов уровня ядра операционной системы типа WaitHandle для синхронизации доступа к разделяемым ресурсам».

```
class Singleton
{
    private static volatile Singleton instance = null;
    private static object syncRoot = new Object();

    private Singleton() { }

    public static Singleton Instance
    {
        get
        {
            Thread.Sleep(500);

            if (instance == null)
            {
                lock (syncRoot) // Закомментировать lock для проверки.
                {
                    if (instance == null)
                        instance = new Singleton();
                }
            }

            return instance;
        }
    }
}
```

Сериализация Singleton.

В редких случаях может потребоваться сериализовать и десериализовать экземпляр класса Singleton. Например, для сериализации и десериализации можно воспользоваться экземпляром класса BinaryFormatter. Сам процесс сериализации и десериализации не вызывает технических

трудностей. Единственный момент, на который следует обратить внимание — это возможность множественной десериализации, когда сериализованный ранее экземпляр-одиночка десериализуется несколько раз в одной программе. В таком случае, может получиться несколько различных экземпляров одиночек, что противоречит идеологии паттерна Singleton. Чтобы предотвратить возможность дублирования экземпляра-одиночки при каждой десериализации, рекомендуется использовать подход, при котором сериализуется и десериализуется не сам экземпляр-одиночка, а его суррогат (объект-заместитель).

```
BinaryFormatter formatter = new BinaryFormatter();
formatter.SurrogateSelector = Singleton.SurrogateSelector;
```

Использование объекта-суррогата дает возможность организовать тонкий контроль над десериализацией экземпляра-одиночки.

```
// Класс Singleton не должен сериализовываться напрямую
// и не должен иметь атрибута [Serializable]!
public sealed class Singleton
{
    private static Singleton instance = null;
    public String field = "Some value";

    public static Singleton Instance
    {
        get { return (instance == null) ?
                     instance = new Singleton() : instance; }
    }

    public static SurrogateSelector SurrogateSelector
    {
        get
        {
            var selector = new SurrogateSelector();
            var singleton = typeof(Singleton);
            var context = new StreamingContext(StreamingContextStates.All);
            var surrogate = new SerializationSurrogate();

            selector.AddSurrogate(singleton, context, surrogate);
            return selector;
        }
    }
```

```
        // Nested class
        private sealed class SerializationSurrogate : ISerializationSurrogate
        {
            // Метод вызывается для сериализации объекта типа Singleton
            void ISerializationSurrogate.GetObjectData(Object obj,
                                   SerializationInfo info, StreamingContext context)
            {
                Singleton singleton = Singleton.Instance;
                info.AddValue("field", singleton.field);
            }

            // Метод вызывается для десериализации объекта типа Singleton
            Object ISerializationSurrogate.SetObjectData(Object obj,
                                                   SerializationInfo info,
                                                   StreamingContext context,
                                                   ISurrogateSelector selector)
            {
                Singleton singleton = Singleton.Instance;
                singleton.field = info.GetString("field");
                return singleton;
            }
        }
    }
```

Singleton с отложенной инициализацией.

Чаще всего метод Instance использует отложенную (ленивую) инициализацию, т.е., экземпляр не создается и не хранится вплоть до первого вызова метода Instance. Для реализации техники отложенной инициализации в C# рекомендуется воспользоваться классом Lazy<T>, причем по умолчанию экземпляры класса Lazy<T> являются потокобезопасными.

```
class Singleton
{
    static Lazy<Singleton> instance = new Lazy<Singleton>();

    public static Singleton Instance
    {
        get
        {
            return instance.Value;
        }
    }
}
```

Пример кода игры «Лабиринт»

В игре Лабиринт классом с единственным экземпляром может быть класс MazeFactory, который строит лабиринт. Легко понять, что для расширения лабиринта путем строительства большего числа комнат со стенами и дверьми не нужно каждый раз строить новую фабрику, всегда можно использовать одну и ту же уже имеющуюся фабрику. Таким образом, сам класс MazeFactory будет контролировать наличие одного единственного своего экземпляра mazeFactory, и будет предоставлять доступ к этому экземпляру путем использования метода Instance.

```
public static MazeFactory Instance()
{
    if (instance == null)
    {
        // Берем значение свойства MAZESTYLE из файла окружения
        string mazeStyle = GetEnv("MAZESTYLE");

        // 0 - совпадают, 1 - не совпадают
        if (string.Compare(mazeStyle, "bombed") == 0)
        {
            Console.WriteLine("Фабрика для лабиринта с бомбами");
            instance = new BombedMazeFactory();
        }
        else if (string.Compare(mazeStyle, "enchanted") == 0)
        {
            Console.WriteLine("Фабрика для лабиринта с заклинаниями");
            instance = new EnchantedMazeFactory();
        }
        else // По умолчанию.
        {
            Console.WriteLine("Фабрика для обычного лабиринта");
            instance = new MazeFactory();
        }
    }
    return instance;
}
```

Следует заметить, что в данной реализации, метод Instance нужно модифицировать при определении каждого нового подкласса MazeFactory. Такой подход является неприемлемым, поскольку не обеспечивает необходимую гибкость разработки. Вариантами решения данной проблемы являются использование принципов «реестра одиночек» и отложенной (ленивой) инициализации, тогда приложению не нужно будет загружать все неиспользуемые подклассы.

Известные применения паттерна в .Net

System.ServiceModel.ServiceHost
http://msdn.microsoft.com/ru-RU/library/system.servicemodel.servicehost.singletoninstance(v=vs.100).aspx
System.Data.DataRowComparer
http://msdn.microsoft.com/ru-ru/library/system.data.datarowcomparer.aspx

Задание к индивидуальной работе:

В рамках выполнения индивидуальной работы студенту предлагается:

1) Изучить порождающие паттерны (Abstract Factory (Абстрактная Фабрика), Builder (Строитель), Factory Method (Фабричный Метод), Prototype (Прототип), Singleton (Одиночка)). Сделать краткий сравнительный анализ о целесообразности использования порождающих паттернов на этапе проектировании программного обеспечения.
2) Посмотреть примеры использования порождающих паттернов с целью изучения целесообразности использования (см. ссылки в метод. указаниях).
3) Выбрать предметную область для проектируемого программного обеспечения (можно взять один из курсовых проектов и модифицировать его на этапе проектирования, используя паттерн по варианту задания).
4) Представить краткие теоретические сведения по выбранному паттерну проектирования.
5) Представить старую диаграмму классов и диаграмму последовательностей (без использования паттернов).
6) Представить диаграмму последовательностей с использованием новой техники. Привести ее краткое описание.
7) Представить диаграмму классов с использованием новой техники проектирования. Описать назначение модифицированных классов и их содержимое.
8) Обосновать, что используемая техника проектирования является эффективной/неэффективной по отношению к Вашему программному продукту.
9) Изучить самостоятельно один из поведенческих паттернов (Chain of Responsibility (Цепочка Обязанностей), Command (Команда), Interpreter (Интерпретатор), Iterator (Итератор), Mediator (Посредник), Memento (Хранитель), Observer (Наблюдатель), State (Состояние), Strategy (Стратегия), Template Method (Шаблонный Метод), Visitor (Посетитель)). Представить краткое описание паттерна и обосновать возможность использования поведенческих паттернов (см. вариант) при проектировании своего программного продукта.

Варианты задания к индивидуальной работе:

Вариант по журналу	Порождающие паттерны (практическое использование относительно своего ПО на этапе проектирования)	Поведенческие паттерны (теоретическое описание, изучение возможностей техники)
1	Abstract Factory (Абстрактная Фабрика)	Chain of Responsibility (Цепочка Обязанностей)
2	Builder (Строитель)	Command (Команда),
3	Factory Method (Фабричный Метод)	Interpreter (Интерпретатор),
4	Prototype (Прототип)	Iterator (Итератор),
5	Singleton (Одиночка)	Mediator (Посредник)
6	Abstract Factory (Абстрактная Фабрика)	Memento (Хранитель)
7	Builder (Строитель)	Observer (Наблюдатель)
8	Factory Method (Фабричный Метод)	State (Состояние)
9	Prototype (Прототип)	Strategy (Стратегия)
10	Singleton (Одиночка)	Template Method (Шаблонный Метод)
11	Abstract Factory (Абстрактная Фабрика)	Visitor (Посетитель)

12	Builder (Строитель)	Visitor (Посетитель)
13	Factory Method (Фабричный Метод)	Chain of Responsibility (Цепочка Обязанностей)
14	Prototype (Прототип)	Command (Команда),
15	Singleton (Одиночка)	Interpreter (Интерпретатор),
16	Abstract Factory (Абстрактная Фабрика)	Iterator (Итератор)
17	Builder (Строитель)	Mediator (Посредник)
18	Factory Method (Фабричный Метод)	Memento (Хранитель)
19	Prototype (Прототип)	Observer (Наблюдатель)
20	Singleton (Одиночка)	State (Состояние)
21	Abstract Factory (Абстрактная Фабрика)	Strategy (Стратегия)
22	Builder (Строитель)	Template Method (Шаблонный Метод)
23	Factory Method (Фабричный Метод)	Template Method (Шаблонный Метод)
24	Prototype (Прототип)	Memento (Хранитель)
25	Singleton (Одиночка)	Iterator (Итератор)

Требования к отчету по индивидуальной работе

1) Привести краткое теоретическое описание одного из порождающих паттернов (см. вариант задания).
2) Представить теоретические сведения по выбранному поведенческому паттерну проектирования. (см вариант задания и перечень рекомендованной литературы).
3) Описать целесообразность применения к выбранной предметной области на этапе проектирования рассмотренных паттернов проектирования.
4) Представить диаграмму классов и диаграмму последовательностей спроектированного ранее программного продукта (без использования паттернов).
5) Представить диаграмму последовательностей для своего программного продукта с использованием новой техники (порождающие паттерны проектирования, см. вариант). Привести ее краткое описание.
6) Представить диаграмму классов с использованием новой техники (порождающие паттерны проектирования, см. вариант). Описать назначение модифицированных классов и их содержимое.
7) Обосновать, является ли используемая техника эффективной/неэффективной по отношению к Вашему программному продукту.

Требования к отчету по индивидуальной работе: формат А4, не более 15 страниц.

Контрольные вопросы:

1) Что такое паттерн?
2) Какие виды паттернов Вы знаете?
3) Какие порождающие паттерны Вы знаете?
4) Какие поведенческие паттерны Вы знаете?
5) При выборе той или иной техники проектирования, нужно ли обращать внимание на дальнейший выбор средств разработки?

Рекомендуемая литература:

1. Тепляков С. Паттерны проектирования на платформе .NET. — СПб.: Питер, 2015. — 320 с.: ил. ISBN 978-5-496-01649-0
2. А. Шевчук, Д. Охрименко, А. Касьянов, «Design Patterns via C#. Приемы объектно-ориентированного проектирования», Itvdn, 2015 г. (электронный вариант), 2015. – 288 с.
3. Эрих Гамма, Ричард Хелм, Ральф Джонсон и Джон Влиссидес. «Приемы объектно-ориентированного проектирования. Паттерны проектирования».: - Библиотека программиста, СПб.: ДМК, 2010. - 366 с.

МЕТОДИЧЕСКИЕ УКАЗАНИЯ И ЗАДАНИЯ
к индивидуальной работе по дисциплине

«Конструирование программного обеспечения»

(для студентов направления подготовки 09.03.04 “Программная инженерия”)

Составители: Алла Викторовна Чернышова