

Паттерн Abstract Factory

Название

Абстрактная фабрика

Также известен как

Kit (Набор инструментов)

Классификация

По цели: порождающий

По применимости: к объектам


Частота использования

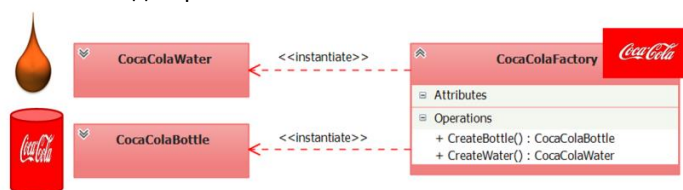
Высокая - 1 2 3 4 **5**

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.

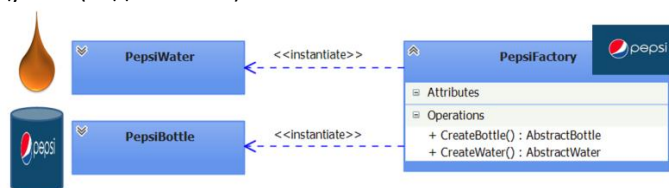
Введение

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 газировует 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, иначе воду будет проблематично поставить потребителю, равно как и пустая банка потребителю не нужна. Порождаемые фабрикой Coca-Cola взаимосвязанные продукты (вода и банка) образуют семейство продуктов фабрики Coca-Cola.

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

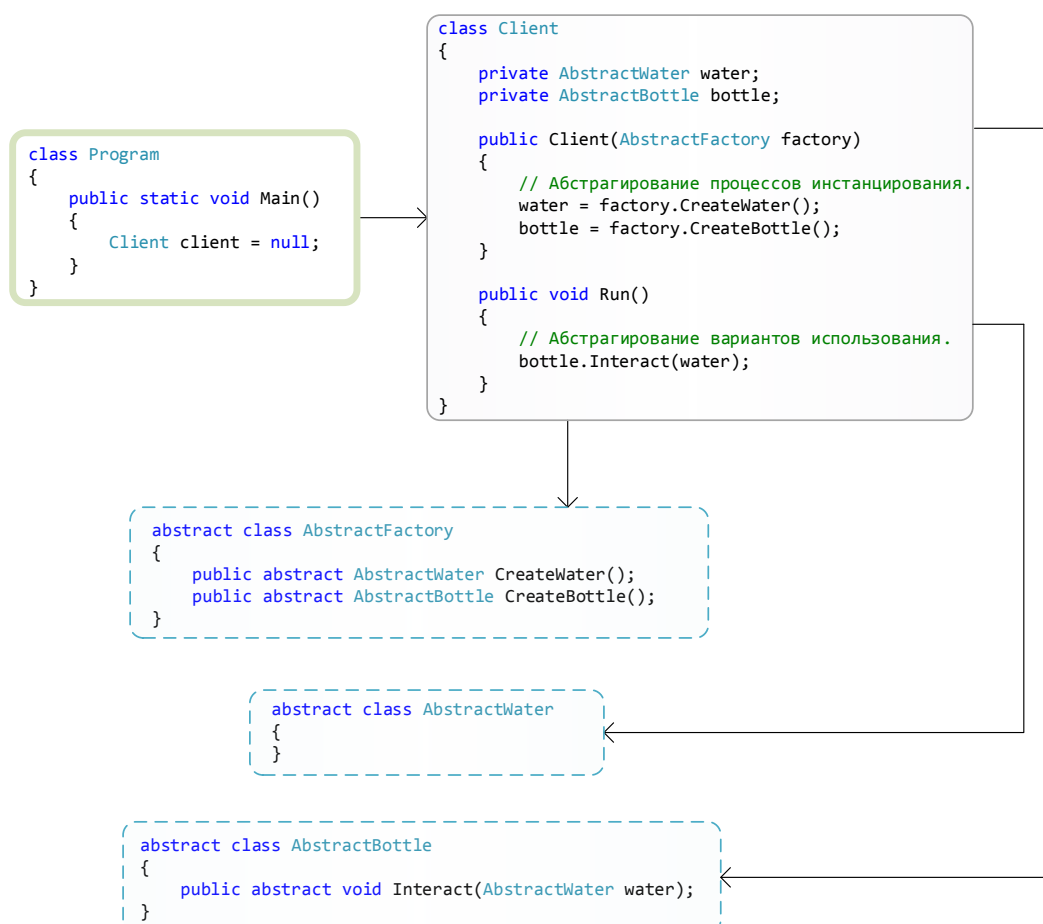
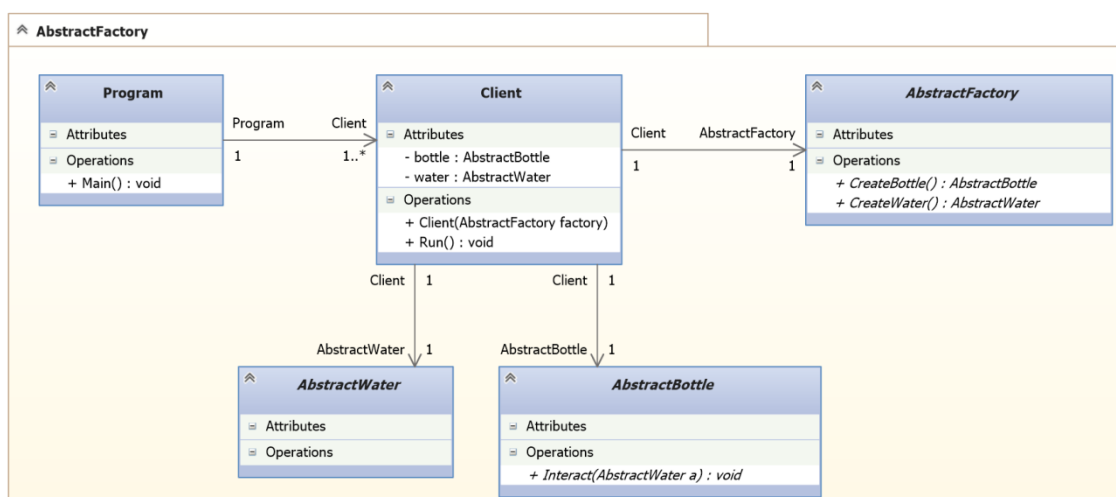


Важно заметить, что не логично пытаться наладить взаимодействие продуктов из разных семейств (например, вливать воду Coca-Cola в банку Pepsi или воду Pepsi в банку Coca-Cola). Скорее всего оба производителя будут против такого взаимодействия. Такой подход представляет собой пример антипатерна.

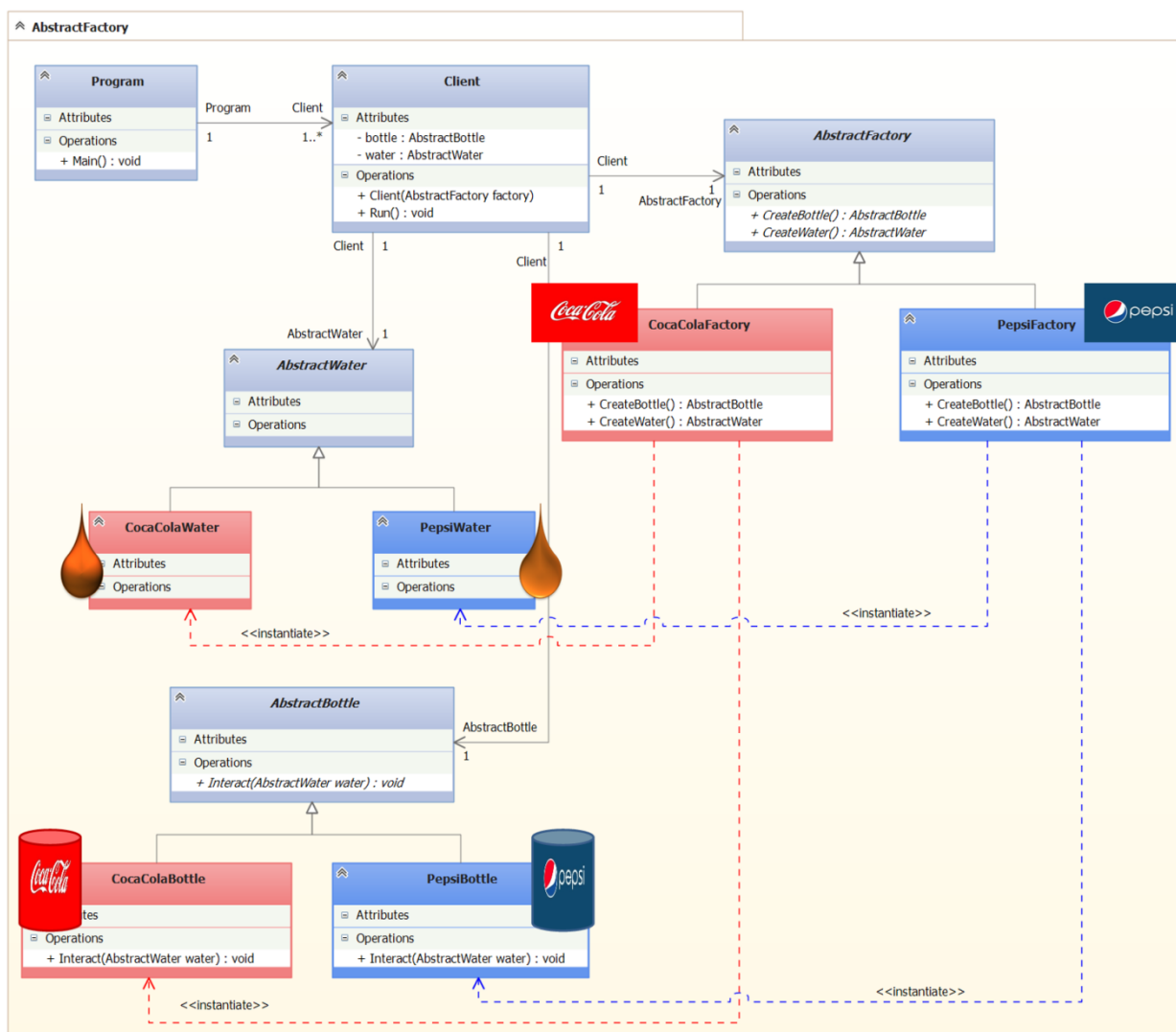
Представим рассмотренные фабрики и порождаемые ими семейства продуктов в контексте одной программы.

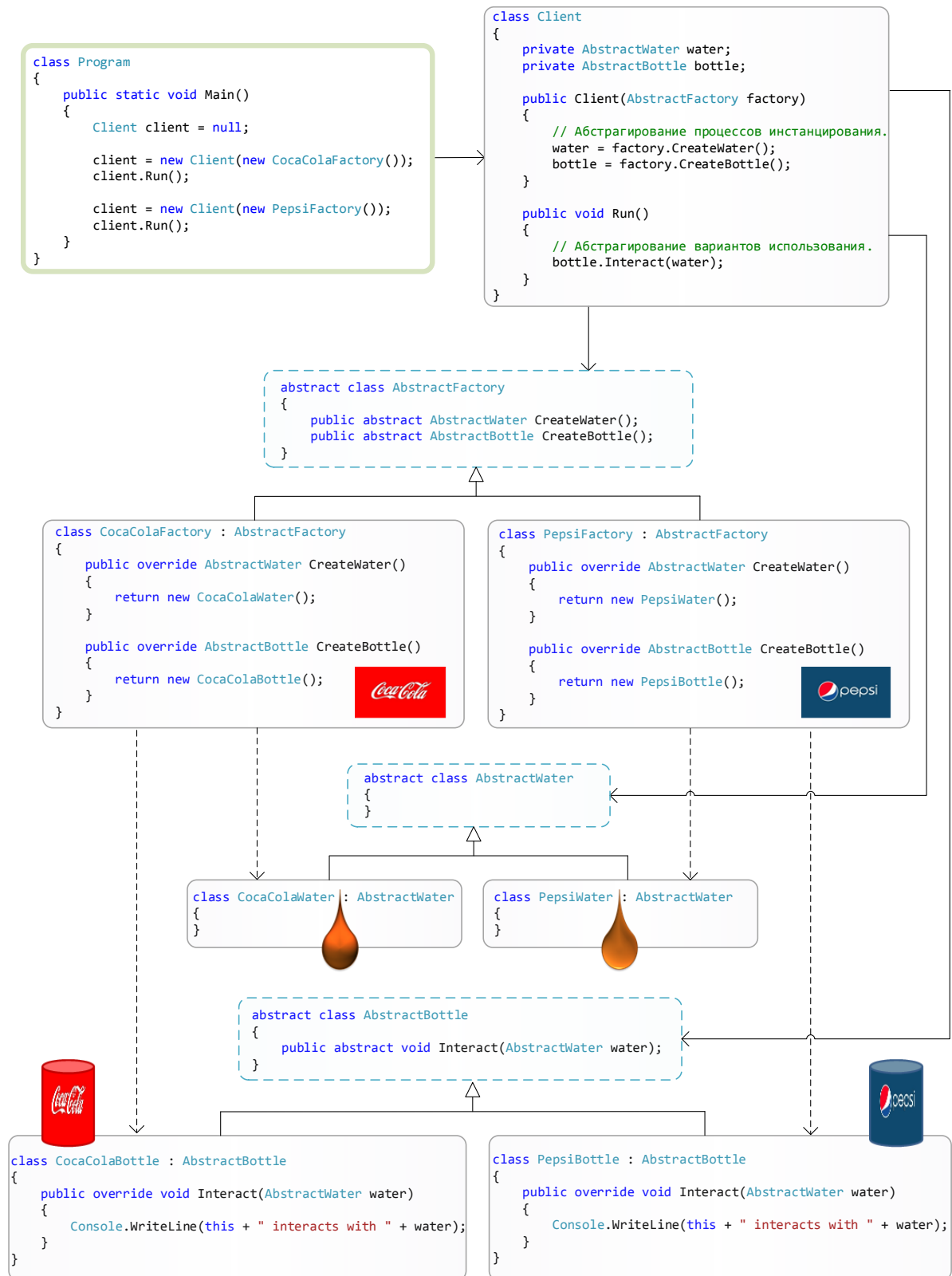
Сперва требуется создать абстрактные классы для задания типов продуктов (**AbstractWater** и **AbstractBottle**) и типа фабрик (**AbstractFactory**). Описать интерфейсы взаимодействия с каждым типом продукта и фабрики.

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).



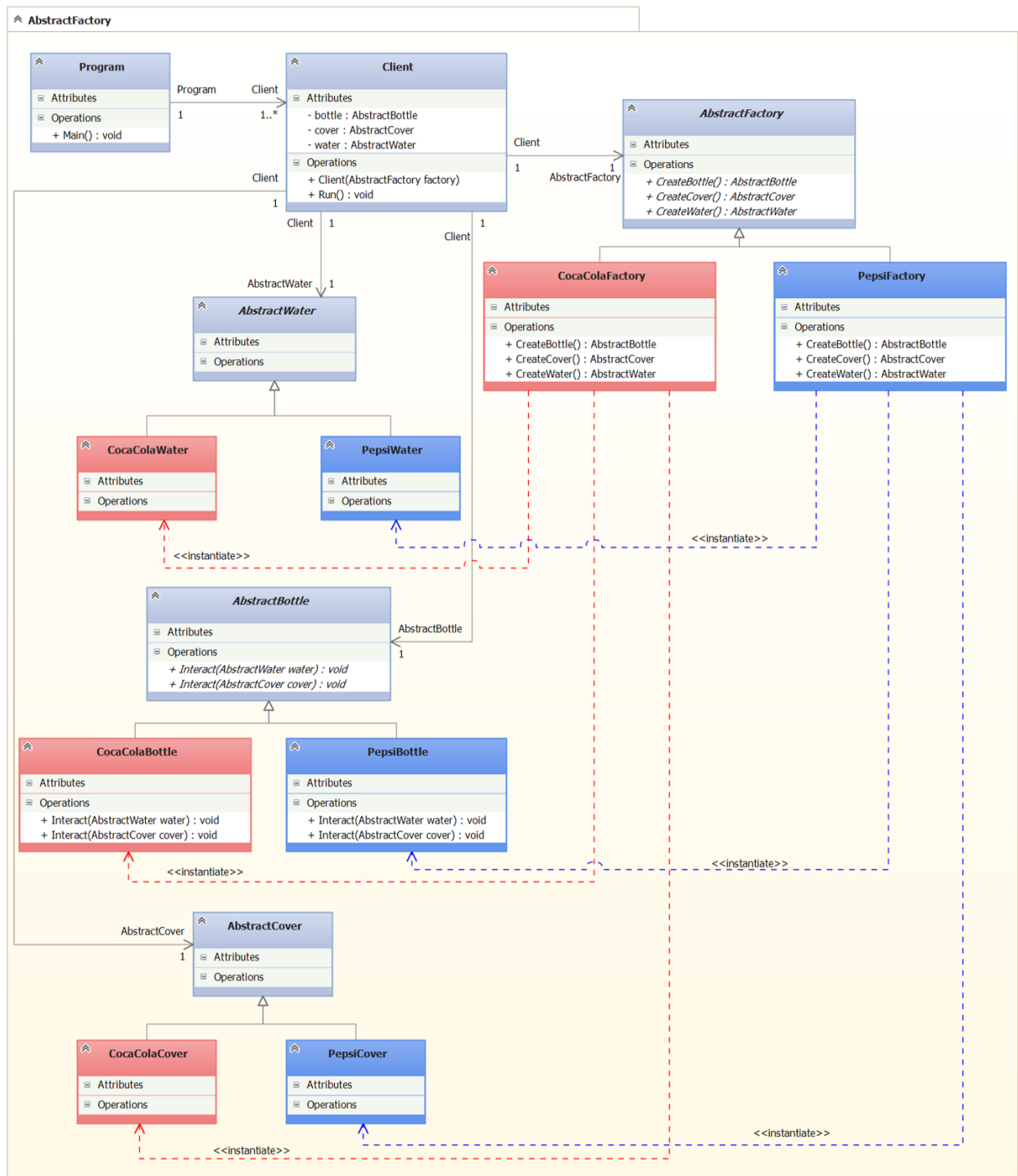
После того, как заданы нужные типы продуктов и фабрик, описаны интерфейсы взаимодействия между продуктами, абстрагированы процессы инстанцирования продуктов и варианты использования продуктов, можно приступить к реализации конкретных классов продуктов и фабрик.



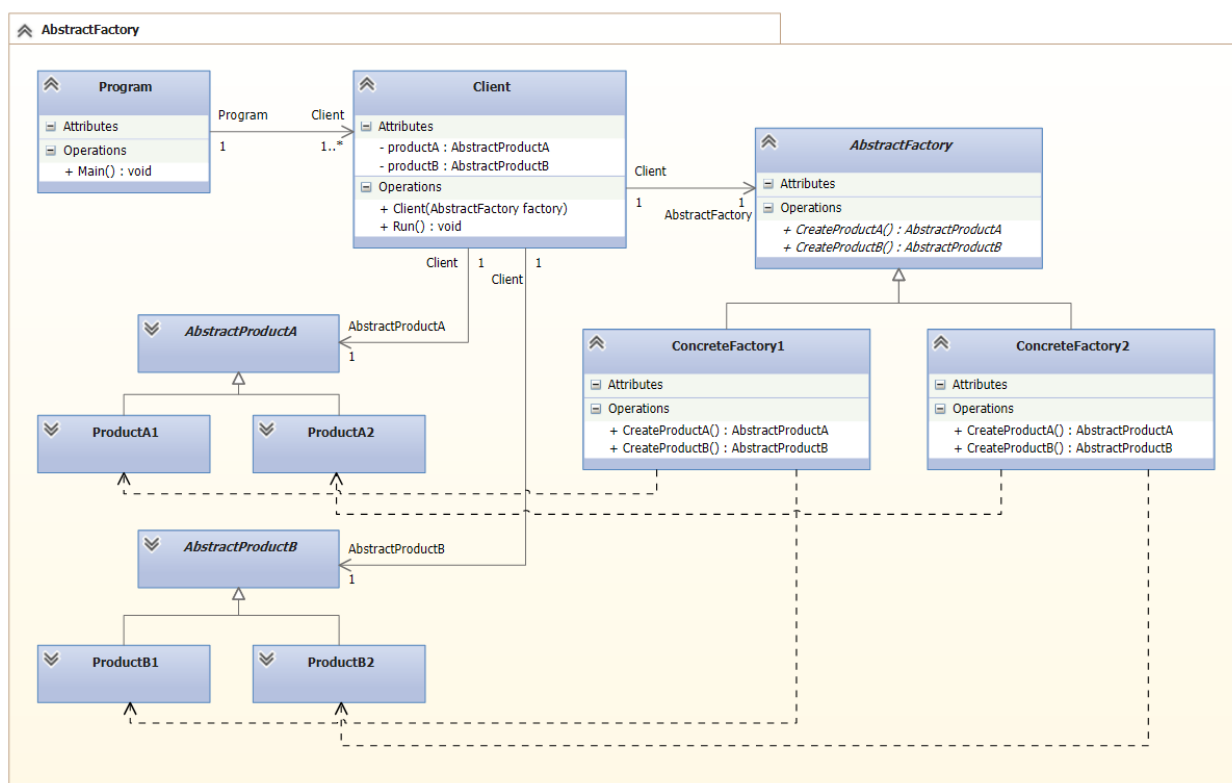


См. Пример к главе: \001_AbstractFactory\000_CocaCola_Pepsi

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

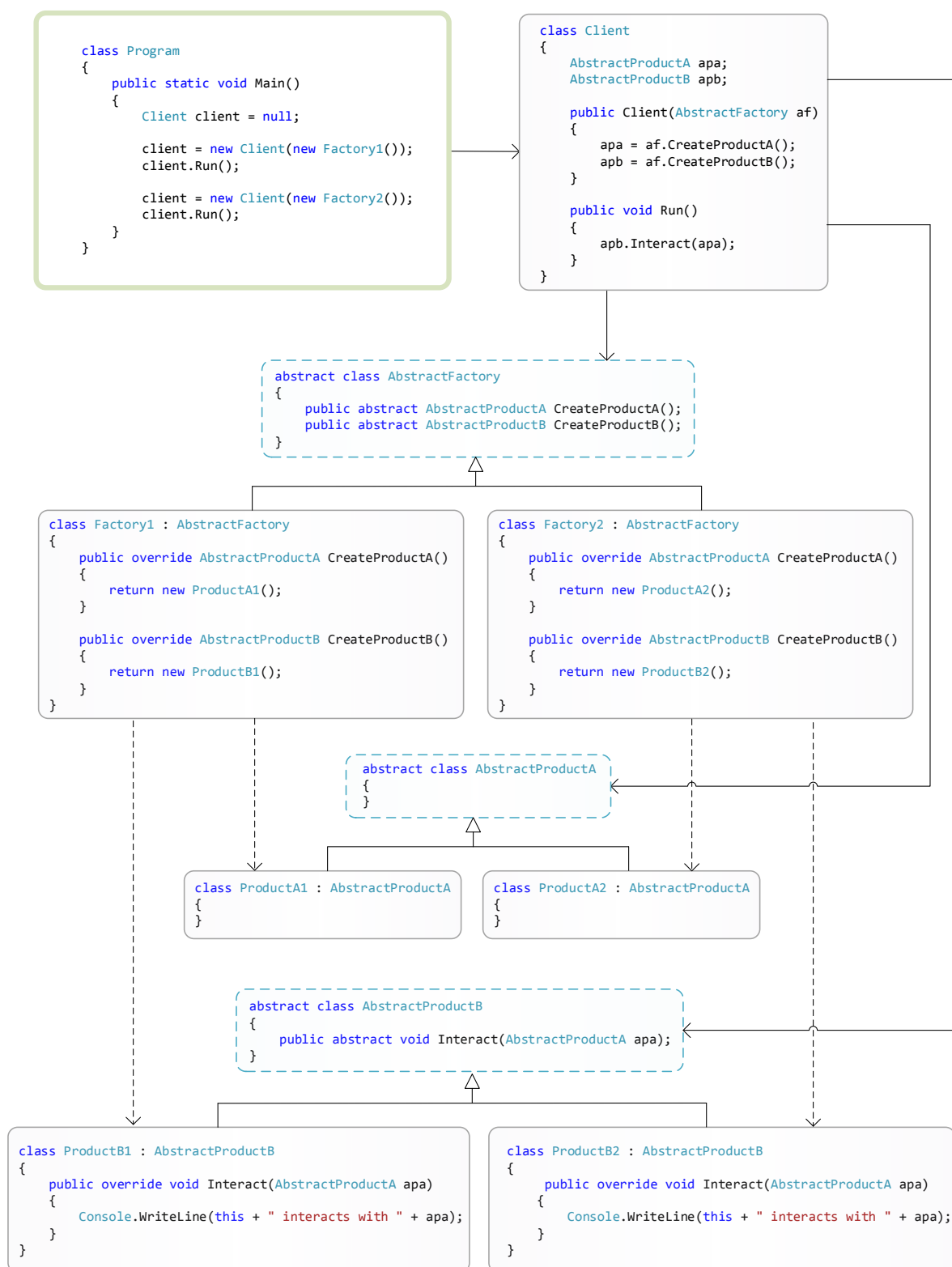


Структура паттерна на языке UML



См. Пример к главе: \001_AbstractFactory\001_AbstractFactory

Структура паттерна на языке C#



См. Пример к главе: \001_AbstractFactory\001_AbstractFactory

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

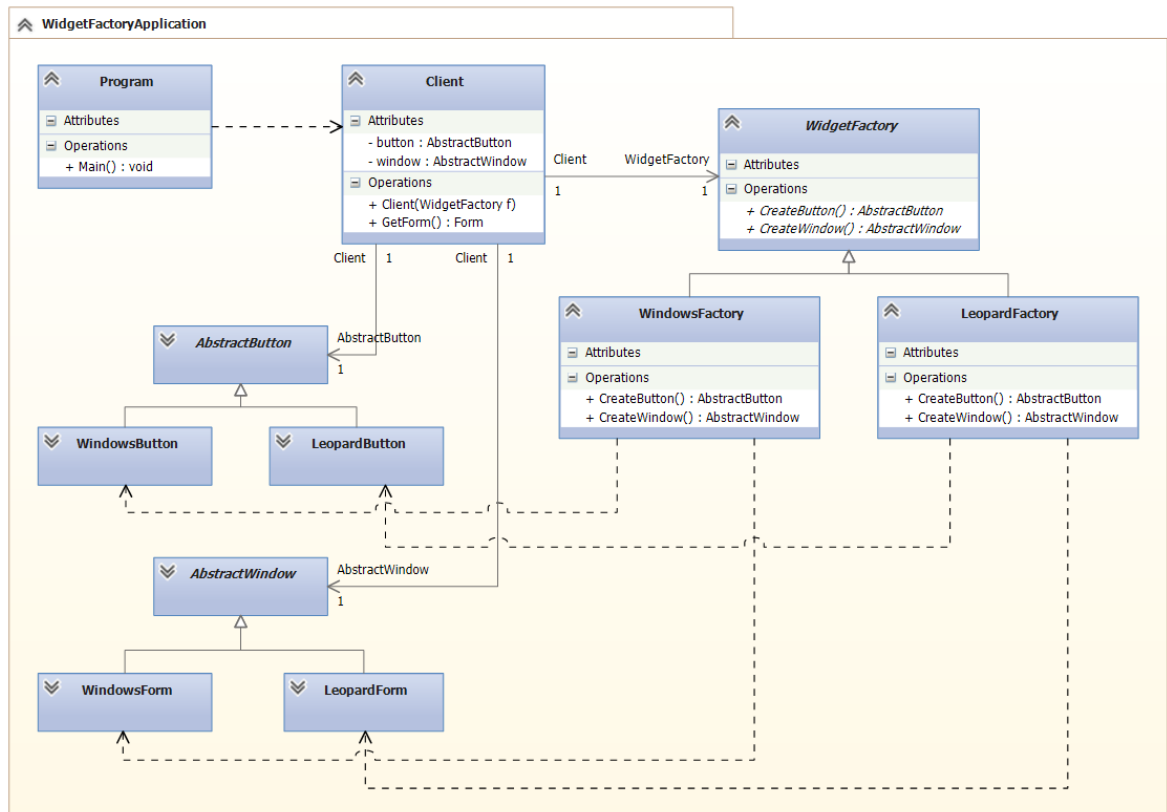
Отношения между классами

- 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



Рассмотрим простейшую программу, в которой поддерживается возможность создания и использования нескольких стилей пользовательского интерфейса, например, стиль Windows Explorer и стиль Mac OS Snow Leopard. В программе будут использоваться два элемента пользовательского интерфейса («controls - контролы» или иногда используется устаревшее название «widgets - виджеты»): Форма и Кнопка.

Хотелось бы предусмотреть в программе возможности быстрого изменения существующих стилей и легкого добавления новых стилей. Если создание элементов управления для каждого имеющегося стиля разбросано по многим участкам кода приложения, то изменять внешний вид такой программы будет неудобно.

Создадим абстрактный класс `WidgetFactory`, в котором имеется интерфейс (набор абстрактных фабричных методов) для создания элементов управления. Создадим абстрактные классы `AbstractWindow` и `AbstractButton` для описания каждого отдельного вида элемента управления и конкретные подклассы (`WindowsForm`, `LeopardForm` и `WindowsButton`, `LeopardButton`), реализующие элементы управления с определенным стилем. В абстрактном классе `WidgetFactory` имеются абстрактные операции (`CreateWindow` и `CreateButton`), возвращающие ссылки на новые экземпляры элементов управления для каждого абстрактного типа контролов. Клиент вызывает эти операции для получения экземпляров контролов, но при этом ничего не знает о том, какие именно конкретные классы используются для их построения. Соответственно клиент ничего не знает и о реализации выбранного стиля.

Для порождения контролов определенного стиля используются классы `WindowsFactory` и `LeopardFactory` производные от базового класса `WidgetFactory`, которые реализуют операции, необходимые для создания элемента управления определенного стиля. Например, операция `CreateButton` в классе `WindowsFactory` создает и возвращает кнопку в стиле Windows, тогда как операция `CreateButton` в классе `LeopardFactory` возвращает кнопку в стиле Snow Leopard. Клиент (`Client`) создает элементы управления, пользуясь исключительно интерфейсом, заданным в абстрактном классе `WidgetFactory`, и ему ничего не известно о классах, реализующих контролы для каждого конкретного стиля. Другими словами, клиент должен лишь придерживаться интерфейса, определенного абстрактным классом, а не конкретным классом.

Класс `WidgetFactory` устанавливает зависимости между конкретными классами контролов. Кнопка для Windows должна использоваться только с формой Windows, и это ограничение поддерживается автоматически, благодаря использованию класса `WindowsFactory`.

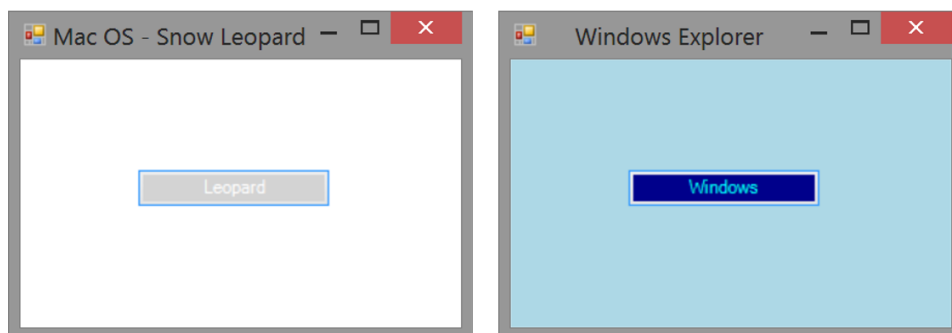


Рисунок. Результат подмены фабрик.

См. Пример к главе: \001_AbstractFactory\002_WidgetFactory

Применимость паттерна

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:

- **Скрытие работы с конкретными классами продуктов.**
Фабрика скрывает от клиента детали реализации конкретных классов и процесс создания экземпляров этих классов. Конкретные классы-продукты 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.**

В подсистеме, создается только один экземпляр класса `ConcreteFactory` для порождения, соответствующего семейства продуктов.

- **Создание объектов-продуктов.**

Класс `AbstractFactory` предоставляет только интерфейс (набор абстрактных методов) для создания продуктов. Фактически продукты создаются в фабричных методах производных конкретных классов-фабрик. Конкретная фабрика реализует фабричные методы, которые возвращают ссылки на создаваемые ими экземпляры продуктов.

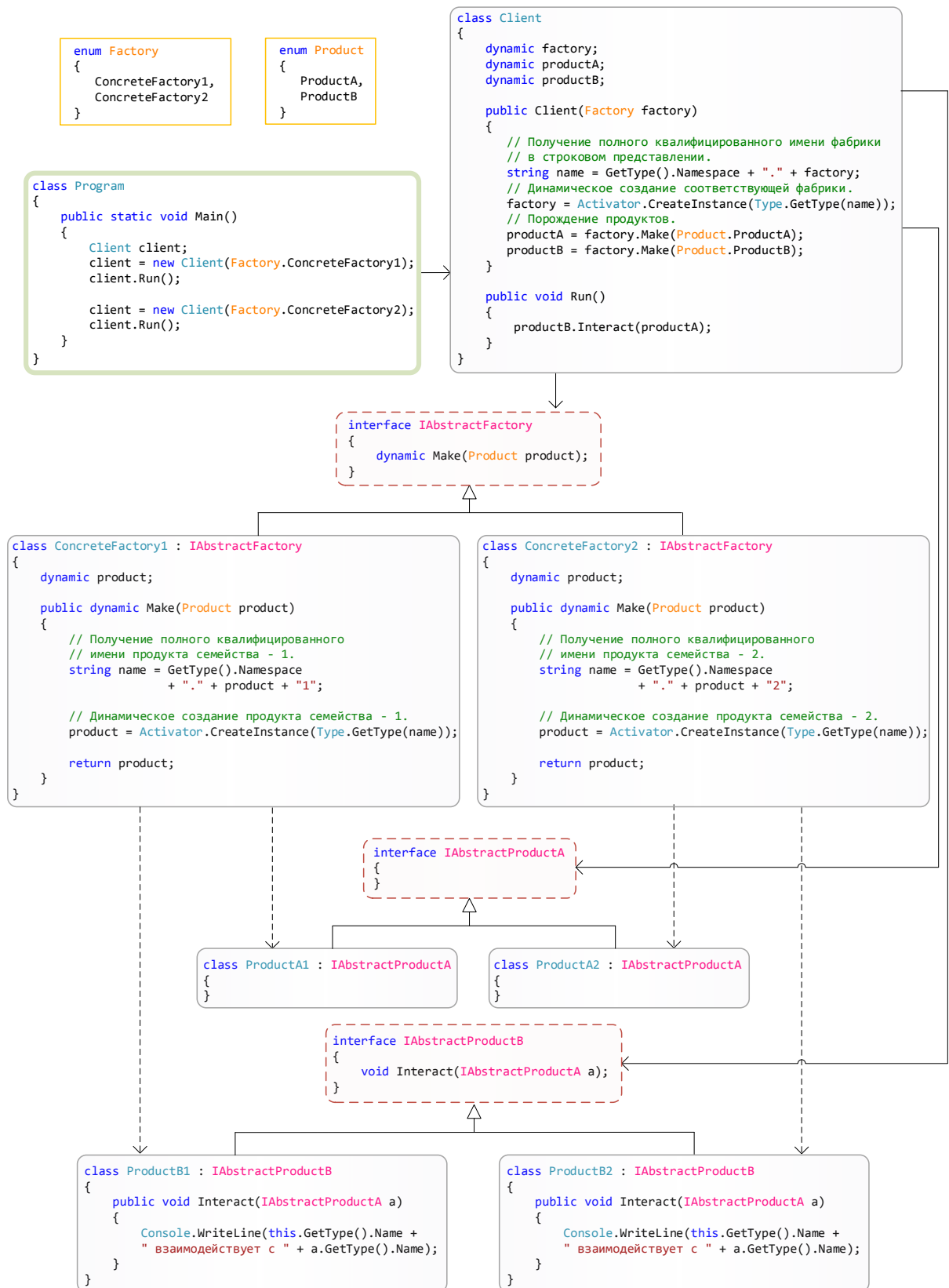
- **Определение расширяемых фабрик.**

Класс `AbstractFactory` содержит абстрактные операции для создания продуктов. Имена фабричных методов включают в себя название типа порождаемого продукта. Для добавления нового вида продуктов нужно добавить новый абстрактный фабричный метод в абстрактный класс `AbstractFactory` и реализовать этот метод в производных классах-фабриках.

Более гибкий способ – использовать фабричные методы с аргументами, описывающими виды создаваемых продуктов. Тип аргумента может быть числовой, строковой или типом перечисления (`enum`), однозначно описывающий тип порождаемого продукта. При таком подходе абстрактному классу `AbstractFactory` нужна только одна операция `Make` с аргументом, указывающим тип создаваемого продукта.

Интересные и гибкие варианты порождения можно организовать в динамически типизированных языках, каким и является C#. Языки с поддержкой динамической типизации позволяют создавать семейства продуктов без наличия общего абстрактного базового класса, а фабричные методы могут иметь возвращаемые значения динамического типа (`dynamic`). Также в языке C# абстрактный класс можно заменить конструкцией языка выражающей такой стереотип как «интерфейс» (`interface IAbstractFactory`).

Если в клиенте отказаться от приведения продуктов к базовому абстрактному типу то, можно было бы выполнить динамическое приведение типа (например, с помощью оператора `dynamic` в C#), но это не всегда безопасно и не всегда заканчивается успешно. Может возникнуть проблемная ситуация: все продукты будут возвращаться клиенту с интерфейсом, который не отображается в *intellisense* (*intellisense* – механизм автодополнения), клиенту будет сложно различать динамические типы продуктов, и могут возникать сложности с их использованием. Такие варианты представляют собой пример компромисса между гибкостью, расширяемостью интерфейса и производительностью.



См. Пример к главе: \001_AbstractFactory\003_AbstractFactory_Net

Пример кода игры «Лабиринт»

Реализацию игры-лабиринта, которая рассматривалась в начале этой главы можно изменить так, чтобы показать на ее примере возможность использования паттерна Abstract Factory.

Класс `MazeFactory` будет использоваться для создания компонентов лабиринта (комнат, стен и дверей между комнатами).

```
class MazeFactory
{
    public virtual Maze MakeMaze()
    {
        return new Maze();
    }

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

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

    public virtual Door MakeDoor(Room room1, Room room2)
    {
        return new Door(room1, room2);
    }
}
```

Метод `CreateMaze` класса `MazeGame`, принимает аргумент типа `MazeFactory` и возвращает построенный лабиринт (ссылку на экземпляр класса `Maze`).

```
class MazeGame
{
    MazeFactory factory = null;

    public Maze CreateMaze(MazeFactory factory)
    {
        this.factory = factory;
        Maze aMaze = this.factory.MakeMaze();
        Room r1 = this.factory.MakeRoom(1);
        Room r2 = this.factory.MakeRoom(2);
        Door aDoor = this.factory.MakeDoor(r1, r2);

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

        r1.SetSide(Direction.North, this.factory.MakeWall());
        r1.SetSide(Direction.East, aDoor);
        r1.SetSide(Direction.South, this.factory.MakeWall());
        r1.SetSide(Direction.West, this.factory.MakeWall());
        r2.SetSide(Direction.North, this.factory.MakeWall());
        r2.SetSide(Direction.East, this.factory.MakeWall());
        r2.SetSide(Direction.South, this.factory.MakeWall());
        r2.SetSide(Direction.West, aDoor);
        return aMaze;
    }
}
```

Эта версия метода CreateMaze лишена недостатка создания экземпляра лабиринта и его компонентов через прямой вызов конструкторов, теперь все компоненты создаются через использование фабричных методов, что позволит при создании варьировать типы создаваемых компонентов лабиринта.

Класс-фабрика `EnchantedMazeFactory` переопределяет фабричные методы базового класса-фабрики `MazeFactory`.

```
class EnchantedMazeFactory : MazeFactory
{
    public override Room MakeRoom(int number)
    {
        return new EnchantedRoom(number, CastSpell());
    }

    public override Door MakeDoor(Room room1, Room room2)
    {
        return new DoorNeedingSpell(room1, room2);
    }

    protected Spell CastSpell()
    {
        return null;
    }
}
```

Предположим, что в данном лабиринте в одной из комнат заложена бомба, когда бомба взрывается то в комнате обрушиваются стены. Для этого нужно создать классы `BombedWall` и `RoomWithABomb`. Класс `BombedWall` наследуется от класса `Wall`, а класс `RoomWithABomb` наследуется от класса `Room`.

```
class BombedWall : Wall
{
}

class RoomWithBomb : Room
{
    // Конструктор.
    public RoomWithBomb(int roomNo)
        : base(roomNo)
    {
    }
}
```

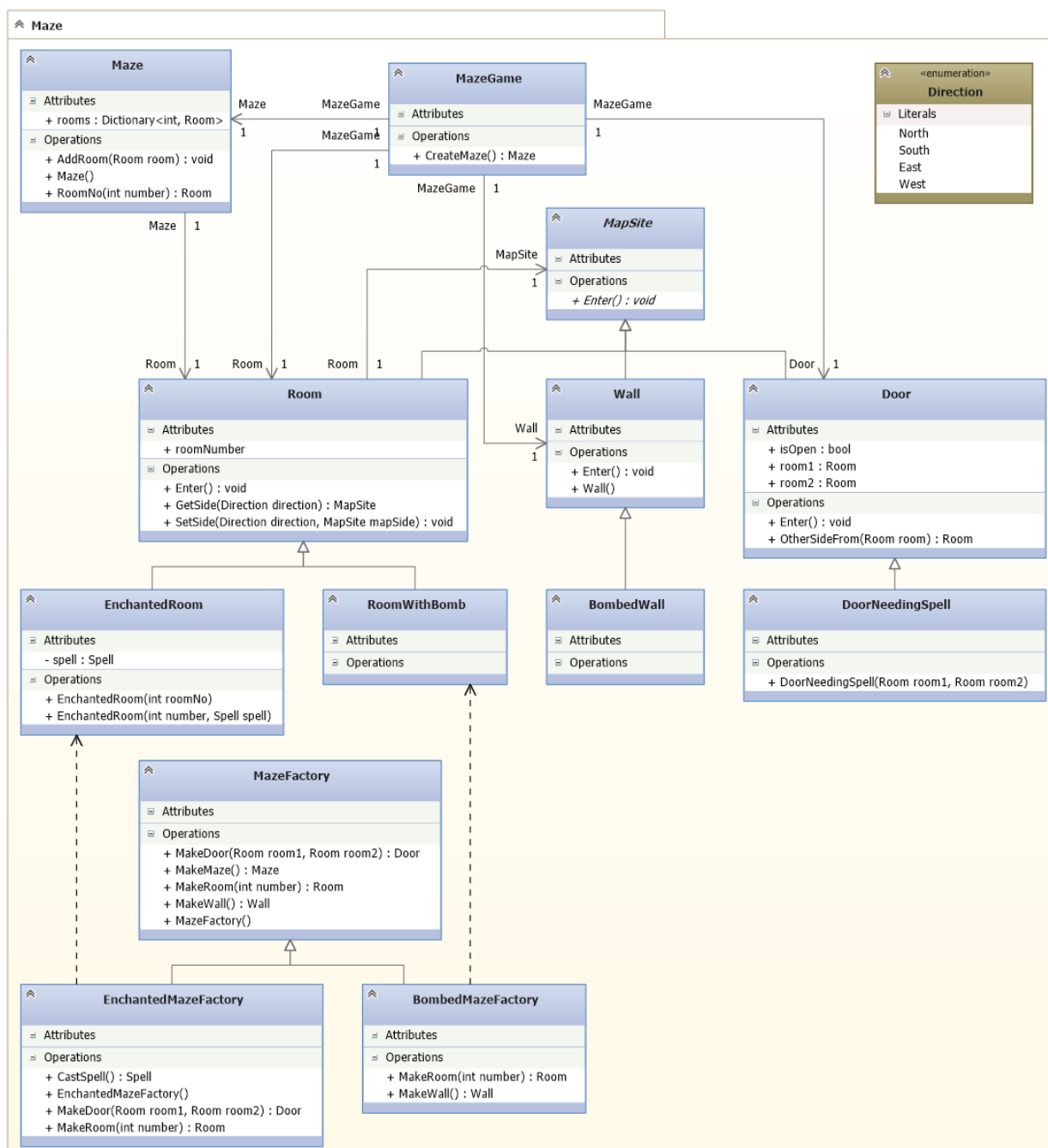
Класс фабрика `BombedMazeFactory` переопределяет фабричные методы базового класса фабрики `MazeFactory`

```
// Фабрика для создания комнат с бомбой.
class BombedMazeFactory : MazeFactory
{
    // Метод создает взорванные стены.
    public override Wall MakeWall()
    {
        return new BombedWall();
    }

    // Метод создает комнату с бомбой.
    public override Room MakeRoom(int number)
    {
        return new RoomWithBomb(number);
    }
}
```

Для построения лабиринта с бомбами вызывается метод `CreateMaze` класса `MazeGame`, которому в качестве аргумента передаётся ссылка на экземпляр класса `BombedMazeFactory`.

Важно заметить, что класс `MazeFactory` является конкретным, а не абстрактным классом, поэтому он используется и как абстрактная фабрика, так и как конкретная фабрика. Такой подход представляет собой еще одну разновидность реализации паттерна Abstract Factory. Так как `MazeFactory` является конкретным классом, хранящим в себе только фабричные методы, легко получить новую фабрику, для этого требуется просто создать новый подкласс класса `MazeFactory` и переопределить в нем фабричные методы.



См. Пример к главе: \MAZE\001_Maze_AF

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](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](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](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](http://msdn.microsoft.com/ru-ru/library/system.xml.serialization.xmlserializerfactory(v=vs.90).aspx)

И т.д.