**Лабораторная работа №1**

**«Реализация одного из порождающих паттернов проектирования»**

**Цель работы:** научиться применять порождающие паттерны проектирования.

**Продолжительность работы** - 4 часа.

**Содержание**

[**Порождающие паттерны** 1](#_Toc100605461)

[**Паттерн Фабричный метод** 1](#_Toc100605462)

[**Паттерн Абстрактная фабрика** 4](#_Toc100605463)

[**Нюансы Factory Method и Abstract Factory** 12](#_Toc100605464)

# **Порождающие паттерны**

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

Список порождающих паттернов (GoF):

* Фабричный метод (*Factory method*)
* Абстрактная фабрика (*Abstract Factory*)
* Строитель (*Builder*)
* Прототип (*Prototype*)
* Одиночка (*Singleton*)

# **Паттерн Фабричный метод**

**Назначение**

Фабричный метод — это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, но позволяет подклассам выбрать класс создаваемых объектов. Таким образом, Фабричный метод делегирует операцию создания экземпляра подкласса.

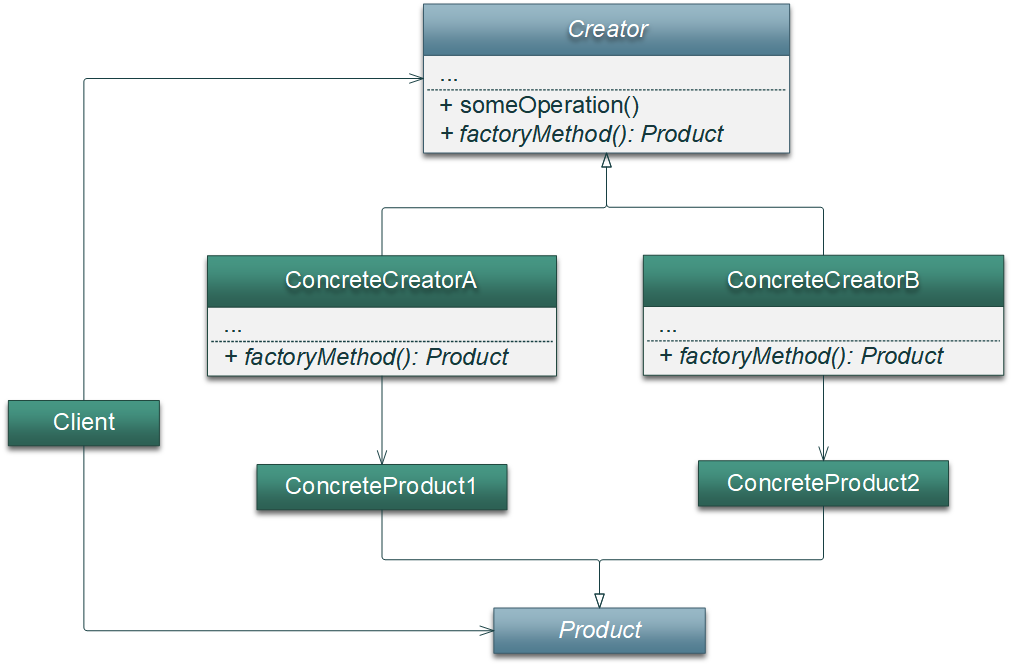
Нужно отметить, что это паттерн не просто позволяет подклассам выбирать класс создаваемого объекта, а даёт возможность суперклассу работать по интерфейсу абстрактного объекта, не вдаваясь в подробности реализации конкретного объекта.

**Применимость**

Использование паттерна Factory Method (фабричный метод) целесообразно если:

* Заранее нельзя знать тип и зависимости объектов, с которыми будут работать другие объекты или пользователь.
* Нужно расширять код производства, не трогая основной. В таком случае используем наследование и переопределение суперкласса в подклассе

**Структура**



Участники

* ***Creator*** – абстрактный создатель: содержит реализацию всех методов, выполняющих операции с продуктами и содержит абстрактный фабричный метод, который должен быть реализован всеми подклассами.
* **ConcreteCreator** (ConcreteCreatorA, ConcreteCreatorB) – конкретные создатели: реализует операции, создающие конкретные объекты-продукты
* ***Product*** - абстрактный продукт: объявляет интерфейс для типа объекта-продукта
* **ConcreteProduct** (Product1, Product2) - конкретный продукт: определяет продукт, создаваемый соответствующей конкретной - реализует интерфейс *Product*
* **Client** - клиент: пользуется исключительно интерфейсами, которые объявлены в классах *Creator* и *Product*

**Результаты применения**

Достоинства:

* ***Убирает привязку класса к конкретным продуктам***.

Поскольку создатель инкапсулирует ответственность за процесс создания, то она изолирует клиента от деталей реализации классов.

* ***Переносит код производства продуктов в одно место***.

Такой подход упрощает поддержку кода.

* ***Упрощает добавление новых продуктов****.*

При добавлении нового конкретного продукта в нем достаточно реализовать интерфейс абстрактного продукта.

* ***Реализует принцип открытости/закрытости*** *(Open-closed Principle, OCP).* Классы и методы должны быть открыты для расширения, но закрыты для модификации
* ***Реализует принцип инверсии зависимостей*** *(Dependency Inversion Principle, DIP).* Код должен зависеть от абстракций, а не реализаций.

Недостатки:

* **Появляется большая иерархия параллельных классов.**

При подходе создания подкласса создателя продуктов на один подкласс продукта, появляется большая иерархия параллельных классов. Но не обязательно, одним фабричным методом в подклассе создателя возвращать лишь один тип подкласса продукта.

# **Паттерн Абстрактная фабрика**

**Назначение**

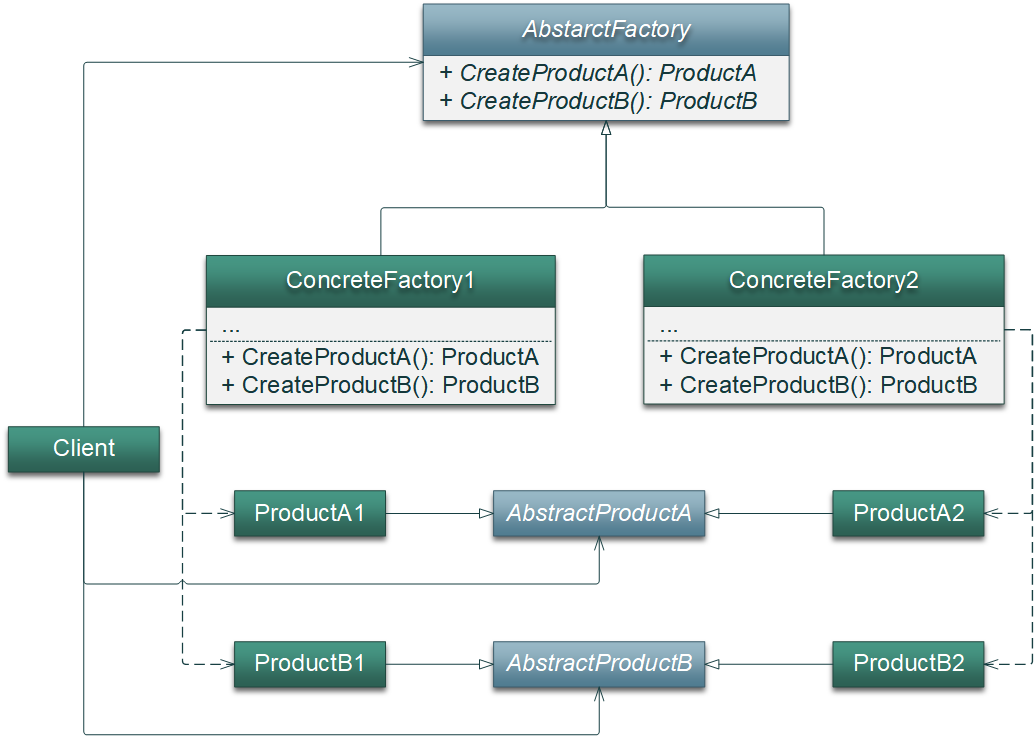
Абстрактная фабрика – это порождающий паттерн проектирования, позволяет работать с разными видами связанных друг с другом продуктов, не завися от конкретных продуктов. Он скрывает от клиентского кода подробности того, какие конкретные объекты будут созданы. Клиент будет работать со всеми типами конкретными продуктов через заранее определённый для них общих интерфейс.

**Применимость**

Использование паттерна Abstract Factory (абстрактная фабрика) целесообразно если:

* Система не должна зависеть от того, как создаются, компонуются и представляются входящие в нее объекты
* Входящие в семейство взаимосвязанные объекты должны использоваться вместе и вам необходимо обеспечить выполнение этого ограничения.
* Система должна конфигурироваться одним из семейств составляющих ее объектов, а вы хотите предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию.
* Когда класс содержит много фабричных методов, чтобы он не потерял свою основную цель, рекомендуется перенести логику создания продуктов в отдельную иерархию классов и применить Абстрактную фабрику. *Один класс должен решать только какую-то одну задачу, такой принцип программирования называется принципом единственной ответственности (Single Responsibility Principle, SRP)*

**Структура**



Участники

* ***AbstractFactory*** - абстрактная фабрика: объявляет интерфейс для операций, создающих абстрактные объекты-продукты
* **ConcreteFactory** (ConcreteFactory1, ConcreteFactory2) - конкретная фабрика: реализует операции, создающие конкретные объекты-продукты
* ***AbstractProduct*** (*AbstractProductА*, *AbstractProductВ*) - абстрактный продукт: объявляет интерфейс для типа объекта-продукта
* **ConcreteProduct** (ProductА, ProductВ) - конкретный продукт: определяет объектпродукт, создаваемый соответствующей конкретной - реализует интерфейс *AbstractProduct*
* **Client** - клиент: пользуется исключительно интерфейсами, которые объявлены в классах *AbstractFactory* и *AbstractProduct*

Отношения

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

**Результаты применения**

Достоинства:

* ***Изолирует конкретные классы***. Помогает контролировать классы объектов, создаваемых приложением. Поскольку фабрика инкапсулирует ответственность за создание классов и сам процесс их создания, то она изолирует клиента от деталей реализации классов.
* ***Упрощает замену семейств продуктов****.* Класс конкретной фабрики появляется в приложении только один раз: при инстанцировании. Это облегчает замену используемой приложением конкретной фабрики.
* ***Гарантирует сочетаемость продуктов****.* Если продукты некоторого семейства спроектированы для совместного использования, то важно, чтобы приложение в каждый момент времени работало только с продуктами единственного семейства. Класс AbstractFactory позволяет легко соблюсти это ограничение;
* **Реализует принцип *единственной ответственности*** *(Single Responsibility Principle, SRP)*. Для каждого класса должно быть определено единственное назначение.
* **Реализует принцип *открытости/закрытости*** *(Open-closed Principle, OCP).* Классы и методы должны быть открыты для расширения, но закрыты для модификации
* **Реализует принцип инверсии зависимостей** *(Dependency Inversion Principle, DIP).* Код должен зависеть от абстракций, а не реализаций.

Недостатки:

* ***Требует наличие всех типов продуктов в каждой вариации****.*

При наследовании абстрактной фабрики, нужно возвращать конкретный продукт, который подходит под семейство реализуемой фабрики

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

**Примеры Abstract Factory**

* Нужно выпускать машины с разным типом кузова, конкретными фабриками будут производители Toyota, Kia и Lada, а их конкретные продукты – седан, купе и универсал.
* Нужно выпускать одежду разного типа, конкретными фабриками будут производители ZARA, H&M и Bershka, а их конкретные продукты – верхняя одежда, нижняя и головные уборы.
* Для правильного отображения UI элементов на разных устройствах нужно использовать стиль и настройки, которые подходят именно этой платформе. Фабрика – платформа, а продукты – UI элементы.

**Пример кода для Abstract Factory**

Приведем реализацию паттерна Abstract Factory для работы с разными базами данных. Предполагается, что у нас будет два семейства БД (SQL Server и SQLite), каждая БД может предоставлять список запросов и таблиц специфичных только для неё, например таблицы в нашей реализации MS SQL содержат имя владельца, а запросы сохраняют время выполнения. У SQLite же не будет ничего такого, только немного другой вывод в консоль содержимого таблицы.

public abstract class AbsractDatabase

{

private string \_connectionString;

public AbsractDatabase(string connectionString)

{

\_connectionString = connectionString;

Console.WriteLine($"Connetion to {\_connectionString}");

}

public abstract List<AbstractTable> GetTables();

public abstract List<AbstractQuery> GetQueries();

}

public abstract class AbstractTable

{

protected string \_tableName;

protected string[] \_columnNames;

protected int \_numberRecords;

public AbstractTable(string tableName, string[] columnNames,

int numberRecords)

{

\_tableName = tableName;

\_columnNames = columnNames;

\_numberRecords = numberRecords;

}

public abstract void PrintInfo();

}

public abstract class AbstractQuery

{

protected string \_name;

protected string \_query;

public AbstractQuery(string name, string query)

{

\_name = name;

\_query = query;

}

public abstract void Execute();

}

public class SqlServerTable : AbstractTable

{

public readonly string Owner;

public SqlServerTable(string tableName, string[] columnNames,

int numberRecords, string owner)

: base(tableName, columnNames, numberRecords)

{

Owner = owner;

}

public override void PrintInfo()

{

Console.WriteLine($"I'm MS SQL Server table," +

$" {\_tableName}," +

$" and my owner is {Owner}" +

$"\nMy columns:");

foreach (string column in \_columnNames)

{

Console.Write(column + " / ");

}

Console.WriteLine();

Console.WriteLine($"Number records - {\_numberRecords}");

}

}

public class SqliteTable : AbstractTable

{

public SqliteTable(string tableName, string[] columnNames,

int numberRecords)

: base(tableName, columnNames, numberRecords) { }

public override void PrintInfo()

{

Console.WriteLine($"I'm SQLite table, {\_tableName}" +

$"\nMy columns:");

foreach(string column in \_columnNames)

{

Console.Write(column + " | ");

}

Console.WriteLine();

Console.WriteLine($"Number records - {\_numberRecords}");

}

}

public class SqlServerQuery : AbstractQuery

{

private int \_executionTime;

public SqlServerQuery(string name, string query)

: base(name, query) { }

public override void Execute()

{

\_executionTime = 300;

Thread.Sleep(300); // Имитация выполнения запроса

Console.WriteLine($"SqlServer Query {\_name} was executed," +

$" Execution time = {\_executionTime}");

}

}

public class SqliteQuery : AbstractQuery

{

public SqliteQuery(string name, string query)

: base(name, query) { }

public override void Execute()

{

Thread.Sleep(500); // Имитация выполнения запроса

Console.WriteLine($"Sqlite Query {\_name} was executed");

}

}

public SqlServerDatabase(string connectionString)

: base(connectionString) { }

public override List<AbstractQuery> GetQueries()

{

List<AbstractQuery> queries = new List<AbstractQuery>();

queries.Add(new SqlServerQuery("Get animals",

"SELECT \* FROM Animals"));

queries.Add(new SqlServerQuery("Get animal with Id = 25",

"SELECT \* FROM Animals" +

"WHERE Id = 1"));

queries.Add(new SqlServerQuery(@"Get bunnies",

"SELECT \* FROM Animals" +

"WHERE type = 'bunny'"));

return queries;

}

public override List<AbstractTable> GetTables()

{

List<AbstractTable> tables = new List<AbstractTable>();

tables.Add(new SqlServerTable("Animals",

new string[] { "Id", "Name", "Type" }, 8, "Ivan")

);

return tables;

}

public SqliteDatabase(string connectionString)

: base(connectionString) { }

public override List<AbstractQuery> GetQueries()

{

List<AbstractQuery> queries = new List<AbstractQuery>();

queries.Add(new SqliteQuery("Get users",

"SELECT \* FROM Users"));

queries.Add(new SqliteQuery("Delete user with Id = 1",

"DELETE FROM Users" +

"WHERE Id = 1"));

return queries;

}

public override List<AbstractTable> GetTables()

{

List<AbstractTable> tables = new List<AbstractTable>();

tables.Add(new SqliteTable("Users",

new string[] { "Id", "First name", "Last name"}, 15)

);

tables.Add(new SqliteTable("Countries",

new string[] { "Id", "Name", "Сapital" }, 60)

);

return tables;

}

static void Main(string[] args)

{

string connectionString = "Server=(localdb)\\" +

"mssqllocaldb;Database=helloappdb";

AbsractDatabase db =

new SqlServerDatabase(connectionString);

List<AbstractTable> tables = db.GetTables();

Console.WriteLine("Tables");

foreach (var table in tables)

{

table.PrintInfo();

}

List<AbstractQuery> queries = db.GetQueries();

Console.WriteLine("Execute queries");

foreach (var query in queries)

{

query.Execute();

}

//--------------------------------------------------

Console.WriteLine();

connectionString = "Data Source=Another.db";

db = new SqliteDatabase(connectionString);

tables = db.GetTables();

Console.WriteLine("Tables");

foreach (var table in tables)

{

table.PrintInfo();

}

queries = db.GetQueries();

Console.WriteLine("Execute queries");

foreach (var query in queries)

{

query.Execute();

}

}

}

# **Нюансы Factory Method и Abstract Factory**

* ***Можно использовать Фабричный Метод даже при наличии одного конкретного создателя***.

Паттерн отделяет реализацию продукта от его использования. Если позднее добавятся другие продукты или изменится реализация продукта, то это не отразится на работе класса-создателя.

* ***В реализации подклассом создателя Фабричного Метода, метод может возвращать разные подклассы продуктов***.

Например, подкласс создателя при передаче в него перечисляемого типа может через конструкцию switch возвращать нужный подкласс продукта

* ***Фабричный метод и класс-создатель не обязательно должны быть абстрактными.***

Фабричный метод может по умолчанию создавать конкретный продукт, что позволит создавать продукты, если подкласс создателя не захочет переопределить этот метод.

* ***Не точное понимание Фабричного метода.***

На практике Фабричный метод позволяет подклассам создателя самостоятельно принимать решение о тип создаваемого продукта во время выполнения. Но более строгий смысл паттерна в том, что класс создателя не должен знать о фактическом типе создаваемых продуктов.

* ***Схожесть Фабричного Метода и Абстрактной фабрики***

Оба паттерна помогают отделить приложение от реализации. Методы Абстрактной Фабрики часто реализованы с помощью Фабричного метода.

* ***Различия Фабричного Метода и Абстрактной фабрики***

Для создания Фабричный Метод использует классы, а Абстрактная Фабрика – объекты. Например, Фабричный метод может создавать продукты, которые могут потребоваться другому методу внутри этого класса. Когда внутри класса становится много фабричных методов, то стоит перенести их в отдельных класс, Абстрактную Фабрику. Абстрактная фабрика нужна для создания **семейства** продуктов. Полезность фабричного метода в отделении клиента от создания конкретных классов , и когда заранее неизвестен состав создателей.