**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.

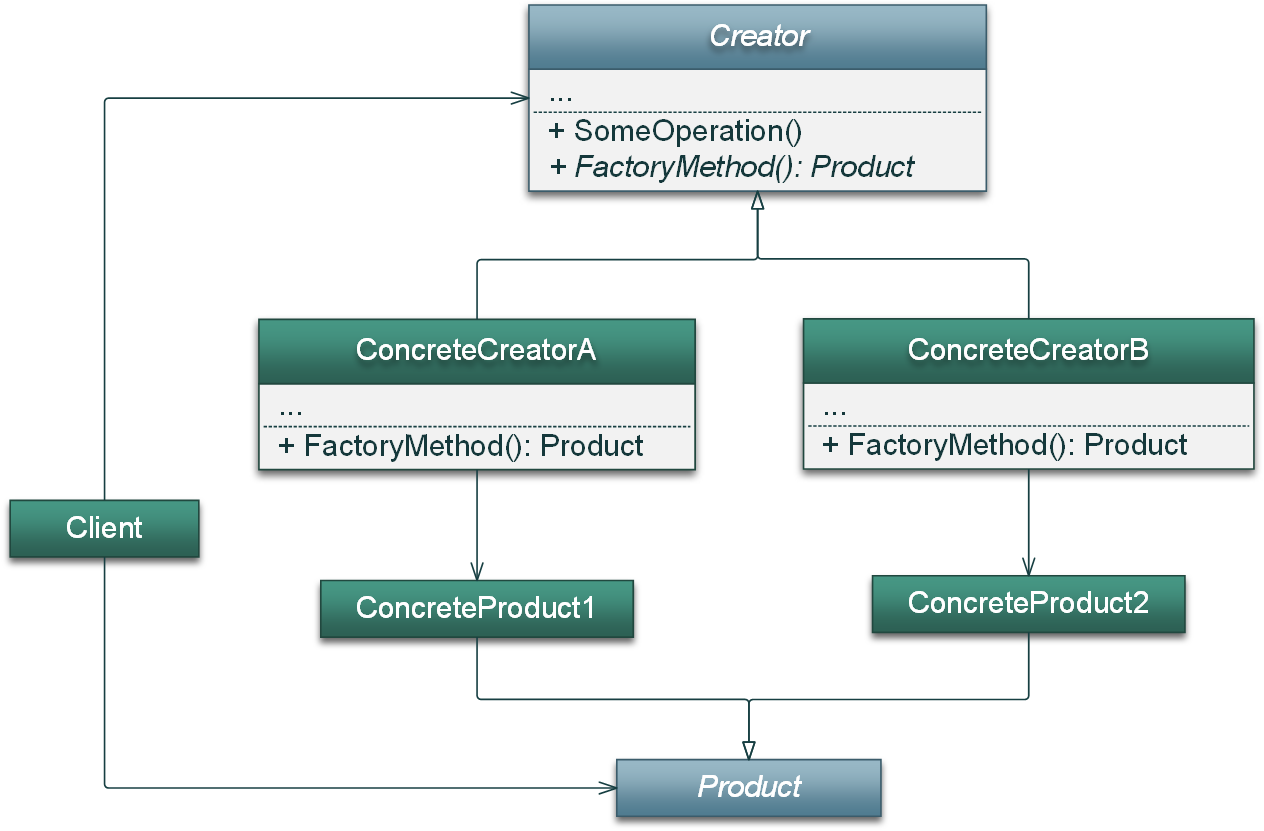
Нужно отметить, что это паттерн не просто позволяет подклассам выбирать класс создаваемого объекта, а даёт возможность суперклассу работать по интерфейсу абстрактного объекта, не вдаваясь в подробности реализации конкретного объекта.

**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**

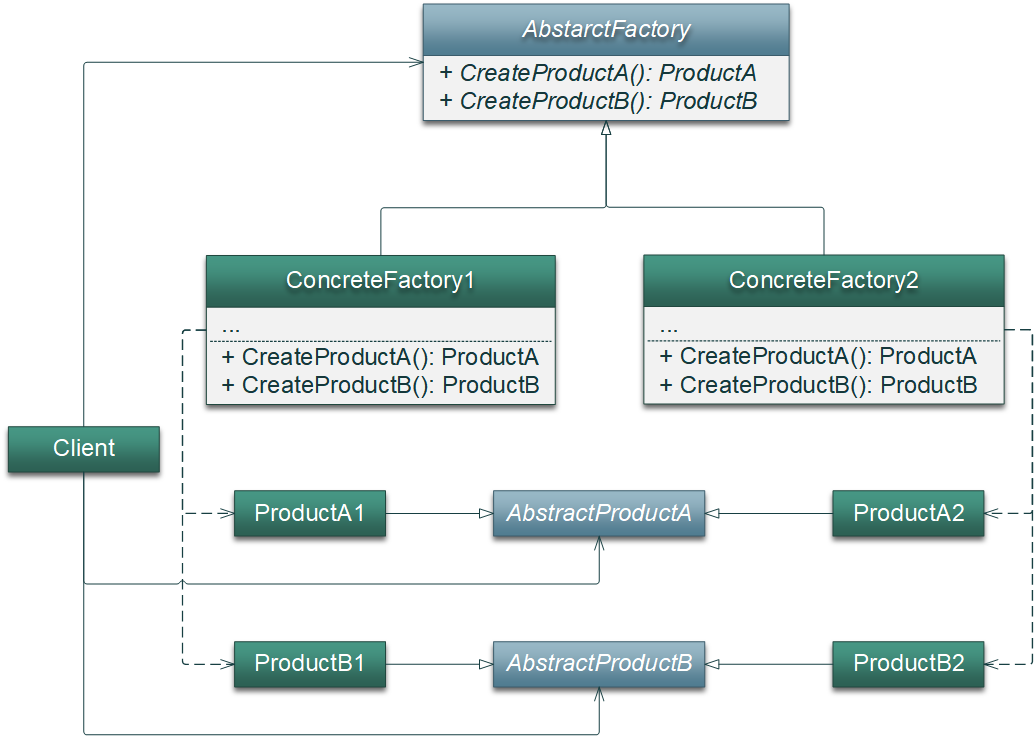
Абстрактная фабрика – это порождающий паттерн проектирования, позволяет работать с разными видами связанных друг с другом продуктов, не завися от конкретных продуктов. Он скрывает от клиентского кода подробности того, какие конкретные объекты будут созданы. Клиент будет работать со всеми типами конкретными 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 объект-продукт, 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.

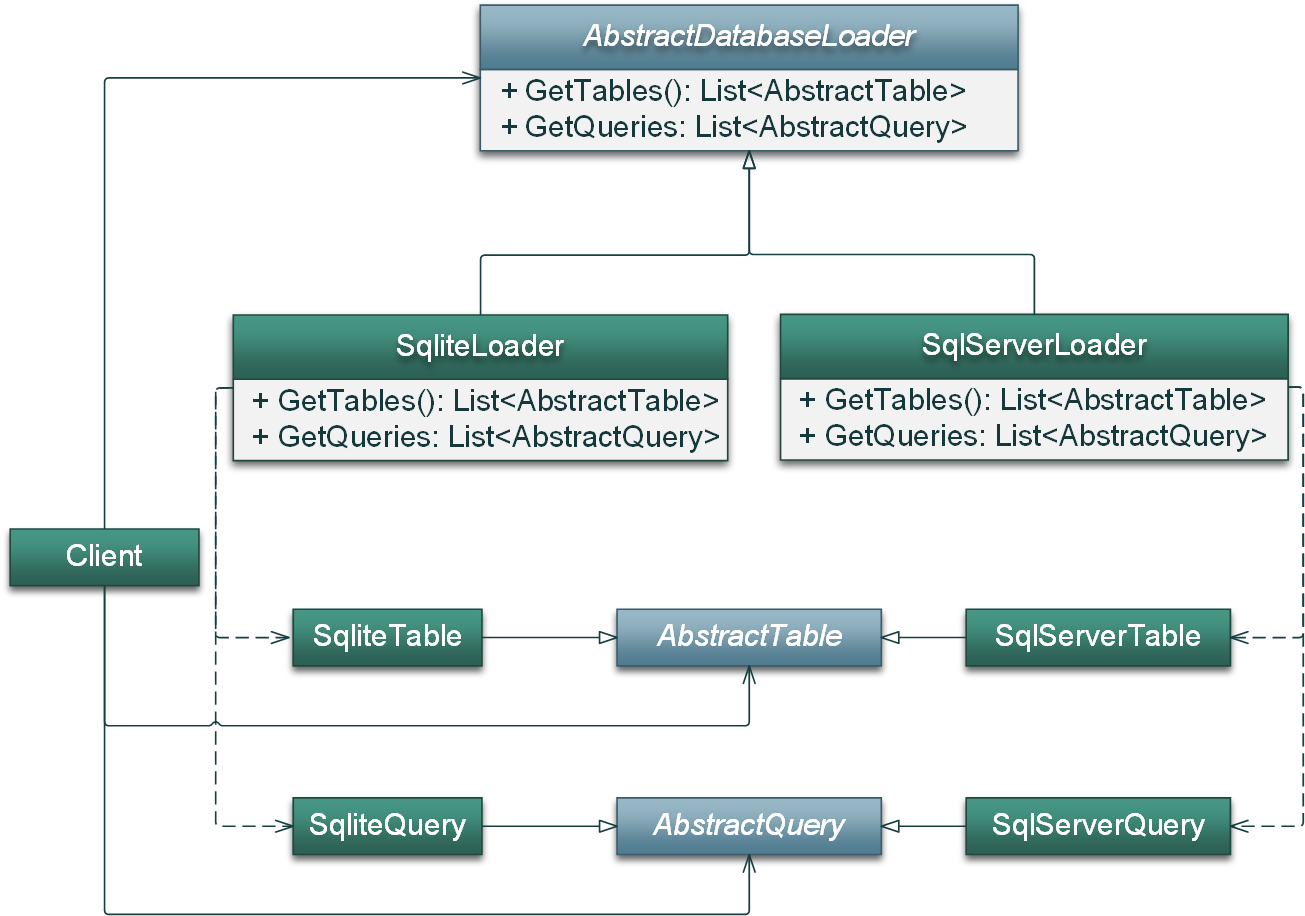
**Примеры Abstract Factory**

* Нужно выпускать 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 содержит список хранимых функций и таблиц специфичных только для неё. например таблицы в нашей реализации SQL Server отличаются от реализации тем, что содержат имя владельца, а запросы сохраняют время выполнения.

Структура паттерна для данного случая на UML диаграмме



// Наша абстрактная фабрика для работы с разными типами БД

public abstract class AbstractDatabaseLoader

{

// Строка для подключения к БД

public readonly string \_connectionString;

public AbstractDatabaseLoader(string connectionString)

{

\_connectionString = connectionString;

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

}

// Фабричные методы для загрузки таблиц и хранимых функций

public abstract List<AbstractTable> GetTables();

public abstract List<AbstractQuery> GetQueries();

}

// Абстрактная хранимая функция - продукт фабрики AbsracDatabaseLoader

public abstract class AbstractQuery

{

protected string \_name;

protected string \_query;

public AbstractQuery(string name, string query)

{

\_name = name;

\_query = query;

}

public abstract void Execute();

// Абстрактная таблица - ещё один продукт фабрики AbsracDatabaseLoader

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();

}

// Конкретная фабрика для получения данных из Sqlite

public class SqliteLoader : AbstractDatabaseLoader

{

public SqliteLoader(string connectionString)

: base(connectionString) { }

// Хранимая функция для БД типа Sqlite - конкретный продукт для SqliteLoader

public class SqliteQuery : AbstractQuery

{

public SqliteQuery(string name, string query)

: base(name, query) { }

// Реализует метод для выполнения функции

public override void Execute()

{

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

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

}

}

// Таблица для БД типа Sqlite - конкретный продукт для SqliteLoader

public class SqliteTable : AbstractTable

{

public SqliteTable(string tableName, string[] columnNames,

int numberRecords)

: base(tableName, columnNames, numberRecords) { }

// Реализует метод для вывода информации в консоль

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}");

}

}

// Конкретная фабрика для получения данных из SqlServer

// Выполним те же шаги реализации, что и в SqliteLoader

public class SqlServerLoader : AbstractDatabaseLoader

{

public SqlServerLoader(string connectionString)

: base(connectionString) { }

public override List<AbstractQuery> GetQueries()

{

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

// В этой реализации будем добавлять функции типа SqlServerQuery

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>();

// Также добавим таблицу типа SqlServerTable

tables.Add(new SqlServerTable("Animals",

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

);

return tables;

}

}

// Функции для БД типа SqlServer - конкретный продукт для SqlServerLoader

// Реализуем и расширим AbstractQuery, сохраняя время выполнения функции

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}");

}

}

// Таблицы для БД типа SqlServer - конкретный продукт для SqlServerLoader

// Реализуем и расширим AbstractTable, запоминая владельца таблицы

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}");

}

}

// Теперь проверим, что сделали

static void Main(string[] args)

{

// Объявим наши абстрактные объекты и строку подключения к БД

AbstractDatabaseLoader db;

List<AbstractTable> tables;

List<AbstractQuery> queries;

string connectionString;

// Создадим подключение для Sqlite и посмотрим содержимое такой БД

connectionString = "Data Source=foobar.db";

db = new SqliteLoader(connectionString);

tables = db.GetTables();

queries = db.GetQueries();

Console.WriteLine("Tables");

foreach (var table in tables)

{

table.PrintInfo();

}

Console.WriteLine("Execute queries");

foreach (var query in queries)

{

query.Execute();

}

//Выполним аналогично для SqlServer

Console.WriteLine();

connectionString = "Server=(localdb)\\mssqllocaldb;Database=helloappdb";

db = new SqlServerLoader(connectionString);

tables = db.GetTables();

queries = db.GetQueries();

Console.WriteLine("Tables");

foreach (var table in tables)

{

table.PrintInfo();

}

Console.WriteLine("Execute queries");

foreach (var query in queries)

{

query.Execute();

}

}

// Вывод в консоль

Conneсtion to Data Source=Another.db

Tables

I'm SQLite table, Users

My columns:

Id | First name | Last name |

Number records - 15

I'm SQLite table, Countries

My columns:

Id | Name | Сapital |

Number records - 60

Execute queries

Sqlite Query Get users was executed

Sqlite Query Delete user with Id = 1 was executed

Conneсtion to Server=(localdb)\mssqllocaldb;Database=helloappdb

Tables

I'm MS SQL Server table, Animals, and my owner is Ivan

My columns:

Id / Name / Type /

Number records - 8

Execute queries

SqlServer Query Get animals was executed, execution time = 300

SqlServer Query Get animal with Id = 25 was executed, execution time = 300

SqlServer Query Get bunnies was executed, execution time = 300

# 

# **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 приватному методу внутри этого же 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, а также когда заранее неизвестен состав создателей.

**Лабораторная работа №2**

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

**Цель работы:** научиться применять паттерн проектирования Builder (строитель)

**Продолжительность работы** - 4 часа.

**Содержание**

[**Паттерн Прототип** 1](#_Toc106830352)

[**Паттерн Одиночка** 2](#_Toc106830353)

[**Паттерн Строитель** 4](#_Toc106830354)

# 

# **Паттерн Прототип**

**Назначение**

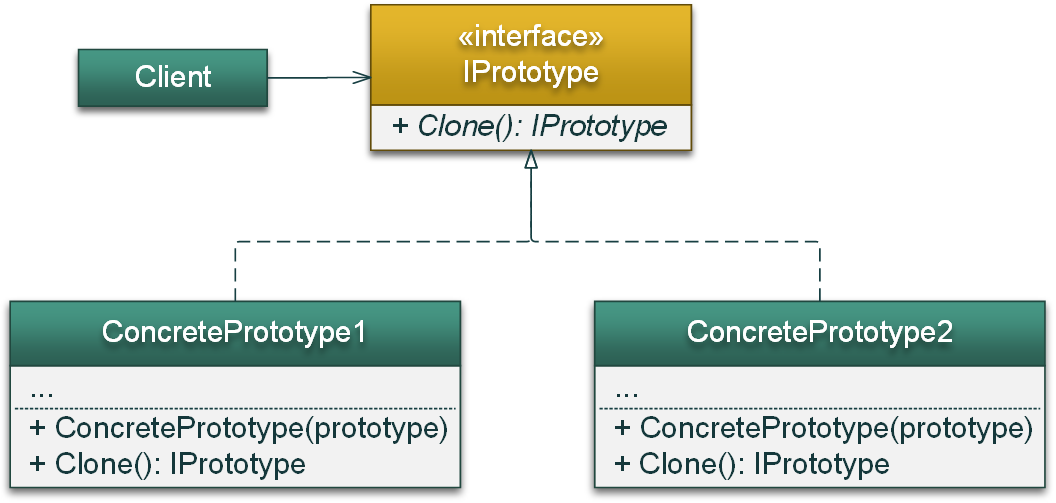
Прототип — это порождающий паттерн проектирования, который позволяет объектам создавать свои копии с помощью общего интерфейса.

**Применимость**

Использование паттерна Prototype (прототип) целесообразно если:

* Нужно сделать точную копию объекта, включая его внутреннее состояние, которое не может быть установлено конструктором.
* Не хочется увеличивать сложность системы, добавлением Фабричного метода или Абстрактной фабрики

**Структура**



Участники

* ***IPrototype*** – интерфейс прототипов: объявляет интерфейс для создания копий реализующего объекта
* ***ConcretePrototype*** – конкретный прототип: реализует операцию копирования себя
* **Client** - клиент: пользуется интерфейсом методом Clone интерфейса IPrototype для создания копии объекта

**Результаты применения**

Достоинства:

* **Увеличивает скорость клонирования объектов.**
* **Реализует*принцип открытости/закрытости.***
* **Реализует*принцип инверсии зависимостей.***

Недостатки:

* **Появляется проблема копирования объектов, имеющих вложенные объекты со ссылочным типом данных.**
* **Не реализует *принцип единственной ответственности.***

# **Паттерн Одиночка**

**Назначение**

Одиночка — это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет глобальную точку доступа к нему. У реализации этого паттерна нет ограничений, только важно, чтобы он удовлетворял своему определению. Существуют потокобезопасные версии, а также с отложенной инициализацией.

**Применимость**

Когда дело касается применимости, то возникает вопрос: «чем Singleton, лучше или хуже статического класса?». Действительно, оба этих решения предоставляет глобальную точку доступа к данным, но их отличие в том, что Singleton предоставляет именно *один* ***экземпляр*** класса, а статический класс свои ***поля****.*

Рассмотрим подробнее их свойства, что выбрать выявить применимость:

* **Количество точек доступа**

У Singleton только одна, статическое поле, а у статического класса зависит от публичных членов класса

* **Наследование классов**

в Singleton такое возможно, но невозможно в статическом классе

* **Наследование интерфейсов**

Аналогично наследованию классов

* **Возможность передачи в качестве параметров**

Singleton может предоставлять свой экземпляр, а статический класс передать как параметр невозможно, только его поля

* **Контроль времени жизни объекта**

Отложенная инициализация возможна для объекта Singleton, в статическом классе некоторые ЯП позволяют сделать отложенную инициализацию полей

* **Использование абстрактной фабрики для создания экземпляра класса**

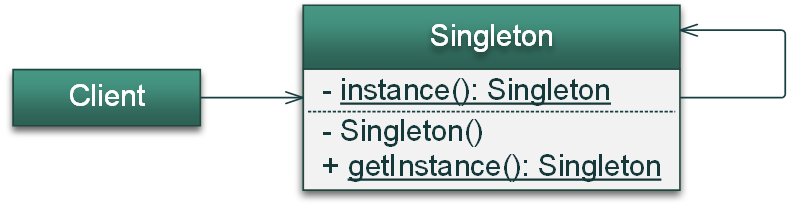
Действительно, объекты-одиночки может создавать Абстрактная фабрика, что невозможно для статических классов

* **Сериализация**

Намного удобнее сериализовать лишь объект-одиночку, чем каждое поле статического класса помещать в какую-либо структуру данных, а затем проводить сериализацию.

Поэтому выбор между этим паттерном и статическим классом зависит от того, нужен ли нам единственный глобальный экземпляр класса со своим состоянием или достаточно глобального набора методов и переменных

**Структура**



Участники

* **Singleton** – одиночка: содержит метод getInstance для получения своего экземпляра
* **Client** - клиент: пользуется экземпляром Singleton

**Пример кода для Singleton**

// Нашим Одиночкой будет профиль пользователя

public class Profile

{

public string Name { get; set; }

public Dictionary<string, string> Settings { get; private set; }

// Используем класс Lazy<T> для отложенной инициализации,

// для потокобезопасности добавим модификатор readonly

private static readonly Lazy<Profile> \_instance =

new Lazy<Profile>(() => new Profile());

private Profile()

{

Settings = new Dictionary<string, string>();

}

public static Profile Instance

{

get

{

return \_instance.Value;

}

}

}

public void Main(string[] args)

{

// Одиночка проинициализируется при первом вызове Instance

var userProfile = Profile.Instance;

// Изменим имя профиле и добавим настроек

userProfile.Name = "Ivan";

userProfile.Settings.Add("theme", "dark");

userProfile.Settings.Add("notifications", "false");

// Теперь мы может передать экземпляр одиночки в любой метод

UseProfile(userProfile);

// Мы также можем сериализовать Одиноку, чтобы сохранить его на диске или БД

var serializedUserProfile = JsonSerializer.Serialize(userProfile);

Console.WriteLine(serializedUserProfile);

// А десериализацию можно реализовать в конструкторе Одиночки

}

private static void UseProfile(Profile profile)

{

Console.WriteLine($"Profile name: {profile.Name}");

foreach (var setting in profile.Settings)

Console.WriteLine($"{setting.Key}: {setting.Value}");

}

**Результаты применения**

Достоинства:

* **Гарантирует наличие единственного экземпляра класса.**
* **Предоставляет глобальную точку доступа.**

Недостатки:

* **Проблемы обращения двух потоков к одному экземпляру.**
* **Не реализует *принцип единственной ответственности.***

**Multiton\***

Возможен вариант использования, когда нужно одновременно иметь доступ к нескольким состояниям экземпляра этого класса. В таком случае можно иметь одновременно несколько экземпляров этого класса, для этого существует реализация c названием Multitone.

# **Паттерн Строитель**

**Назначение**

Строитель — это порождающий паттерн проектирования, который отделяет конструирование сложного объекта от его представления, так что в результате одного и того же процесса конструирования могут получаться разные представления. Даёт возможность способ одновременно поэтапного и различного построения объектов.

**Применимость**

Использование паттерна Prototype (прототип) целесообразно если:

* Алгоритм создания сложного объекта не должен зависеть от того, из каких частей состоит объект и как они стыкуются между собой.
* Процесс конструирования должен обеспечивать различные представления конструируемого объекта.

**Результаты применения**

Достоинства:

* **Позволяет изменять внутреннее представление продукта.** Объект Builder предоставляет распорядителю абстрактный интерфейс для конструирования продукта, за которым он может скрыть представление и внутреннюю структуру продукта, а также процесс его сборки. Поскольку продукт конструируется через абстрактный интерфейс, то для изменения внутреннего представления достаточно всего лишь определить новый вид строителя.
* **Изолирует код, реализующий конструирование и представление.** Паттерн строитель улучшает модульность, инкапсулируя способ конструирования и представления сложного объекта. Клиентам ничего не надо знать о классах, определяющих внутреннюю структуру продукта, они отсутствуют в интерфейсе строителя. Каждый конкретный строитель ConcreteBuilder содержит весь код, необходимый для создания и сборки конкретного вида продукта.
* **Дает более тонкий контроль над процессом конструирования.** В отличие от порождающих паттернов, которые сразу конструируют весь объект целиком, строитель делает это шаг за шагом под управлением распорядителя. И лишь когда продукт завершен, распорядитель забирает его у строителя.

Недостатки:

* **Зависимость конкретного строителя от создаваемого продукта.**

При изменении класса продукта, вероятно, придётся корректировать класс ConcreteBuilder.

**Структура**

Картинка

Участники

* ***IPrototype*** – интерфейс прототипов: объявляет интерфейс для создания копий реализующего объекта
* ***ConcretePrototype*** – конкретный прототип: реализует операцию копирования себя
* **Client** - клиент: пользуется интерфейсом методом DeepCopy интерфейса IDeepCopy для создания копии объекта

Отношения

* Клиент создает объект-распорядитель Director и конфигурирует его нужным объектом строителем Builder
* Распорядитель уведомляет строителя о том, что нужно построить очередную часть продукта
* Строитель обрабатывает запросы распорядителя и добавляет новые части к продукту
* Клиент забирает продукт у строителя

**Реализация**

Обычно существует абстрактный класс *Builder*, в котором определены операции для каждого компонента, который распорядитель может «попросить» создать. По умолчанию эти операции ничего не делают. Но в классе конкретного строителя ConcreteBuilder они замещены для тех компонентов, в создании которых он принимает участие.

Вот еще некоторые достойные внимания вопросы реализации:

* **Интерфейс сборки и конструирования.** Интерфейс класса *Builder* должен быть достаточно общим, чтобы обеспечить конструирование при любом виде конкретного строителя. Ключевой вопрос проектирования связан с выбором модели процесса конструирования и сборки. Обычно бывает достаточно модели, в которой результаты выполнения запросов на конструирование просто добавляются к продукту. Но иногда может потребоваться доступ к частям сконструированного к данному моменту продукта. Например, деревья синтаксического разбора строятся снизу вверх.
* **Почему нет абстрактного класса для продуктов**. В типичном случае продукты, изготавливаемые различными строителями, имеют настолько разные представления, что изобретение для них общего родительского класса ничего не дает. Поскольку клиент обычно конфигурирует распорядителя подходящим конкретным строителем, то, надо полагать, ему известно, какой именно подкласс класса *Builder* используется и как нужно обращаться с произведенными продуктами
* **Пустые методы класса Builder пo умолчанию.** В C++ методы строителя намеренно не объявлены чисто виртуальными функциями-членами. Вместо этого они определены как пустые функции, что позволяет подклассу замещать только те операции, в которых он заинтересован.

**Пример кода для Builder**

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 содержит список хранимых функций и таблиц специфичных только для неё. например таблицы в нашей реализации SQL Server отличаются от реализации тем, что содержат имя владельца, а запросы сохраняют время выполнения.