Метод фабрика (Factory Method)

Цел на шаблона

„Метод Фабрика“ (Factory Method) е шаблон, който дефинира интерфейс за създаване на обекти, но точния тип на създадения обект се решава от наследниците на фабриката.

Приложимост

Най-често шаблонът „Метод Фабрика“ се използва в следните ситуации:

- Когато искаме да предоставим точка на разширение за наследниците на даден клас при инстанциране на обект.
  - Виртуален конструктор - дава възможност наследниците на фабриката да решат какъв тип обект да създадат.
- Когато не искаме да използваме обикновен конструктор и искаме да енкапсулираме създаването на конкретен обект.
- Когато искаме да създаваме йерархии от обекти. (Като част от по-обхватния шаблон Абстрактна Фабрика.)
- Когато искаме да предоставим по-добро и по-говорящо име на конструктора на даден клас.
  - Дава възможност за различни имена на конструкторите.
- Когато искаме да контролираме броя на създадените обекти:
  - Може да връща една и съща инстанция многократно.
  - Да забавим създаването на даден обект до първото му поискване (Lazy-loading).
- Когато искаме да избегнем оператора new, който е опасен и може да има странични ефекти.
  - Понякога създаването на обект не е сигурно, че ще е успешно - може да хвърли изключение, което не е добре да се случва в конструктор.
  - Използва се за създаване на обекти зависещи от интернет услуги, файлова система, потоци или други обекти на операционната система.
- Много често се използва като „Шаблонен Метод“ (Template Method) за създаване на обекти.
  - Използва се за създаване на сложни обекти, които имат нужда от повече конфигуриране.

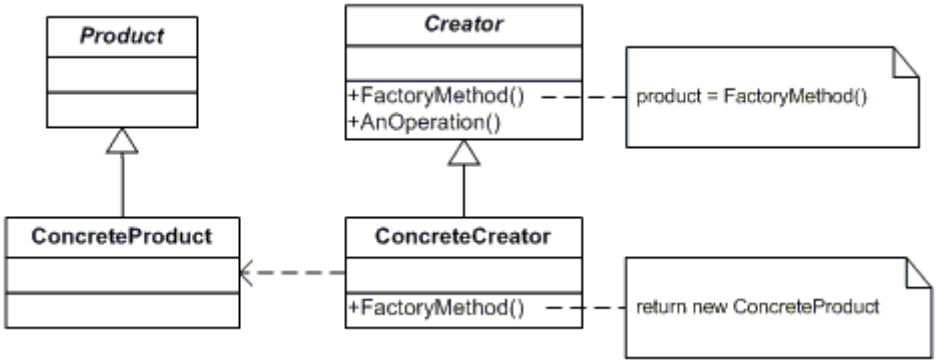
Оценка на шаблона

1. Шаблонът „Метод Фабрика“ дава предимство пред простото използване на контруктори, енкапсулирайки създаването на конкретен обект. Много често се налага, когато се създаде един обект да се конфигурира по специален начин. Това конфигуриране заедно със самото създаване може много лесно да се скрие (енкапсулира) в един метод, вместо да бъде копирано навсякъде, където създаваме обекти.

2. Използването на шаблона води до подобряване качеството на кода, защото се програмира срещу общ интерфейс на продуктите, а не конкретни техни имплементации.
3. Предоставя се точка на разширение за наследниците на даден клас при създаването на даден продукт. Наследниците на обекта създадел могат да предефинират метода фабрика и по-този начин да променят типа на връщания обект (конкретен продукт), запазвайки базовата функционалност. (Виртуален конструктор).
4. Може да служи като средство за по-добра абстракция, когато работим с различни йерархии от класове.

Един от основните недостатъци е, че Фабриките работят с йерархии от класове и ако продуктите не споделят общ интерфейс (базов клас), то не може да се използва шаблона „Метод Фабрика“.

UML class diagram:



Примерен код:

```
// Factory Method pattern -- Structural example

using System;


namespace DoFactory.GangOfFour.Factory.Structural
{
  /// <summary>
  /// MainApp startup class for Structural
  /// Factory Method Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      // An array of creators
```

```
    Creator[] creators = new Creator[2];

    creators[0] = new ConcreteCreatorA();
    creators[1] = new ConcreteCreatorB();

    // Iterate over creators and create products
    foreach (Creator creator in creators)
    {
      Product product = creator.FactoryMethod();
      Console.WriteLine("Created {0}",
        product.GetType().Name);
    }

    // Wait for user
    Console.ReadKey();
  }
}

/// <summary>
/// The 'Product' abstract class
/// </summary>
abstract class Product
{
}

/// <summary>
/// A 'ConcreteProduct' class
/// </summary>
class ConcreteProductA : Product
{
}

/// <summary>
/// A 'ConcreteProduct' class
/// </summary>
class ConcreteProductB : Product
{
```

```
  }

  /// <summary>
  /// The 'Creator' abstract class
  /// </summary>
  abstract class Creator
  {
    public abstract Product FactoryMethod();
  }

  /// <summary>
  /// A 'ConcreteCreator' class
  /// </summary>
  class ConcreteCreatorA : Creator
  {
    public override Product FactoryMethod()
    {
      return new ConcreteProductA();
    }
  }

  /// <summary>
  /// A 'ConcreteCreator' class
  /// </summary>
  class ConcreteCreatorB : Creator
  {
    public override Product FactoryMethod()
    {
      return new ConcreteProductB();
    }
  }
}
```

Output

```
Created ConcreteProductA
Created ConcreteProductB
```

Адаптер (Adapter)

Цел на шаблона

Целта на този шаблон е да адаптира един клас към интерфейс, който той не притежава по време на изпълнението на програмата, без да се променя дефиницията на класа.

Приложимост

Шаблонът се използва в следните случаи:

– нужда от клас, чийто интерфейс не съвпада с очаквания (например случая, дефиниран в предната секция)

– нужда от преизползваем клас, опериращ с несвързани или непредвидени интерфейси. Когато създаваме библиотека, която трябва да работи с различни имплементации на дадена абстракция, се налага тези имплементации да бъдат адаптирани към очакван от библиотеката интерфейс. Вж. „вградими адаптери“ по-долу.

– (само за адаптер на обекти) нужда от използването на няколко съществуващи класове наследници. Понякога имаме нужда да адаптираме общият интерфейс на цяла йерархия от наследници и това може да стане с един единствен адаптер на обекти.

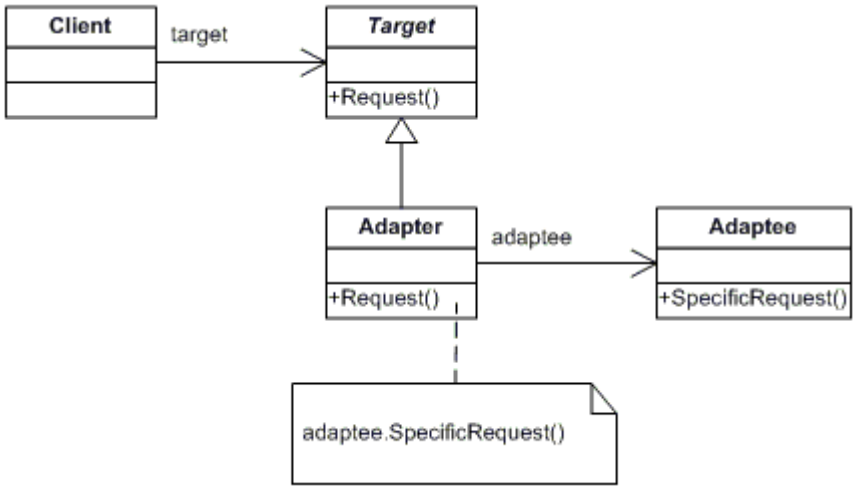
Важно уточнение е, че под „интерфейс“ нямаме предвид понятието от езика - **`interface`**, а по-широкия смисъл – набор от операции, с които обект се представя на заобикалящата го среда. В този смисъл адаптерът може да адаптира конкретен клас към конкретен клас, без те да имплементират **`interface`**.

Оценка на шаблона

Ето няколко заключения за приложимостта на шаблона, както и забележки за някои по-специфични случаи:

1. Адаптерите на обекти имат предимството, че могат да адаптират цяла йерархия, а не само един конкретен клас. Използването на адаптери на класове е ограничено в езици, в които липсва множествено наследяване.
2. Понякога интерфейсът, който трябва да бъде адаптиран има твърде много методи. Това само по себе си и сигнал за лош дизайн и можем да опитаме да разделим интерфейса на няколко отделни такива, като адаптираме само нужната част. Друга опция е да не предоставяме имплементации на всеки метод, а само на тези, които ще са нужни на клиента. Трябва да сме внимателни с тази опция, защото в бъдеще клиентът може да започне да работи и с други методи, а адаптерът да не ги предоставя.
3. Степента на адаптиране зависи от това доколко са сходни двата интерфейса. При по-съществени разлики се налага конвертиране на параметри, подаване на параметри по подразбиране или неизползване на някои параметри.
4. Шаблонът е изключително полезен при работа с остарели библиотеки, като позволява техните класове да бъдат използвани съвместно с по-съвременни такива.
5. Вградимите адаптери („pluggable adapters”) могат да адаптират динамично няколко различни класа, които не са известни по време на компилация, и които нямат общ интерфейс.

UML class diagram:



Примерен код:

```
// Adapter pattern -- Structural example

using System;


namespace DoFactory.GangOfFour.Adapter.Structural
{
  /// <summary>
  /// MainApp startup class for Structural
  /// Adapter Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      // Create adapter and place a request
      Target target = new Adapter();
      target.Request();


      // Wait for user
```

```
    Console.ReadKey();
  }
}

/// <summary>
/// The 'Target' class
/// </summary>
class Target
{
  public virtual void Request()
  {
    Console.WriteLine("Called Target Request()");
  }
}

/// <summary>
/// The 'Adapter' class
/// </summary>
class Adapter : Target
{
  private Adaptee _adaptee = new Adaptee();

  public override void Request()
  {
    // Possibly do some other work
    //  and then call SpecificRequest
    _adaptee.SpecificRequest();
  }
}

/// <summary>
/// The 'Adaptee' class
/// </summary>
class Adaptee
{
  public void SpecificRequest()
  {
```

```
      Console.WriteLine("Called SpecificRequest()");
    }
  }
}
```

Output

```
Called SpecificRequest()
```

Стратегия (Strategy)

Цел на шаблона

Да се дефинират взаимно заменяеми алгоритми, които решават един и същ проблем, но по различни начини, позволявайки да променяме поведението на обекта, който ги използва, без да променяме самия него.

Приложимост

Преди да започнем да разглеждаме как точно шаблонът *"Стратегия" (Strategy)* решава гореспоменатите проблеми, ще изброим още няколко случая, при които е добре да вземем предвид възможната употреба на този шаблон:

– Ако имаме много класове, които решават един и същ проблем, но по различни начини. *"Стратегия" (Strategy)* ни дава възможност да ги използваме от един клас, като променяме поведението му по време на изпълнение на програмата.

– Имаме клас, който променя поведението си в зависимост от даден параметър, чрез употребата на много условни конструкции. Можем да заместим всяка такава конструкция със стратегия.

– В даден клас имаме прекалено много структури и логика, свързани с алгоритъм, изпълняващ се в класа. *"Стратегия" (Strategy)* ни позволява да отделим и скрием тези подробности в отделен клас.

Имаме клас, който има няколко начина на поведение и не ни е известно по време на компилация кой от тях трябва да използваме. Чрез *"Стратегия" (Strategy)* можем да вземем това решение по време на изпълнение на програмата.

Оценка на шаблона

Има случаи, в които шаблонът *"Стратегия" (Strategy)* е подходящ, но има и такива, в които не е. Имайки предвид следващите няколко точки, може да се прецени дали е удачно да се приложи в конкретна ситуация:

1. Опасност от голям брой класове, имплементиращи базовия интерфейс **`Strategy`**. Ако броят на стратегиите е голям, се получават големи йерархии с много класове. Тези йерархии могат да се организират и подредят, като всички стратегии, чиито алгоритми имат общи части, наследят общ базов клас.

2. Опция за наследяване на контекста. Можем да направим класове, наследници директно на контекста, като всеки наследник дефинира различен алгоритъм. Това обаче обвързва всеки контекст с даден алгоритъм, правейки го по-труден за поддръжка и разбиране. Освен това, губим възможността да сменяме алгоритъма по време на изпълнение на програмата. Като резултат от това решение, получаваме йерархия, класовете в която се различават само по алгоритъма, реализиран във всеки от тях. Поставяйки всяка стратегия в отделен клас, я правим независима от контекста, по-лесна за разбиране и променяне.

3. Възможност за премахване на много условни изрази. Когато имаме клас, който променя поведението си спрямо различни параметри, е добре да се замислим дали не ни е нужен шаблонът *"Стратегия" (Strategy)*. Прекаленото използване на **`if…else`** конструкции води до сложен, неразбираем и най-вече труден за проследяване код. Прекомерното използване на условни изрази често може да се замести със стратегии.

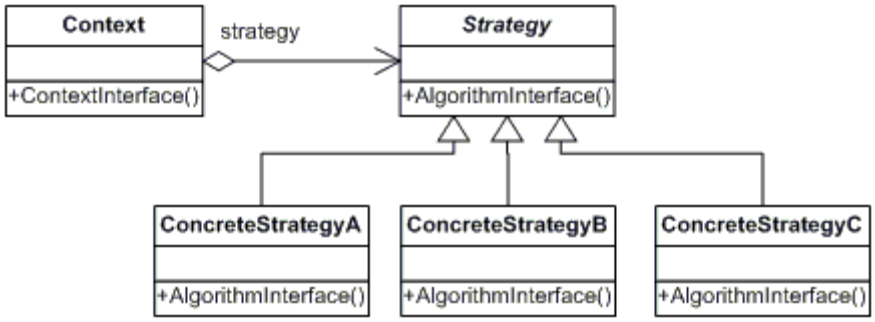
4. Възможност за превключване между бързина и ефективност. Например, ако имаме един и същ алгоритъм с две различни импементации, едната от които работи бързо, но консумира много памет и друга, която пести памет, но изисква много време, за да приключи. Можем да избираме коя от двете имплементации да използваме по време на изпълнение на програмата, благодарение на шаблона *"Стратегия" (Strategy)*.

5. Опасност от подаване на прекалено много информация към стратегията. Трябва внимателно да се подбира информацията, която контекстът подава на съответната стратегия. Тъй като всички стратегии имплементират един и същ интерфейс, всички конкретни стратегии ще получават една и съща информация. Някои от по-простите алгоритми може да не използват цялата подавана им информация (или дори никаква), което би довело до неоптимално използване на ресурси.

6. Опасност от голям брой обекти. Тъй като всяка стратегия се съдържа в отделен обект, в зависимост от броя нужни стратегии, броят на обектите може да нарасне драматично. Този проблем е възможно да бъде превъзмогнат, като се използва шаблонът *"Миниобект" (Flyweight)*. Така всеки различен вид стратегия ще има само една инстанция и всеки контекст ще предава специфичната за него информация на тази инстанция.

7. *"Стратегия" (Strategy)* срещу *"Мост" (Bridge)*. И двата шаблона изглеждат доста подобни, но това е само на пръв поглед. *"Стратегия" (Strategy)* позволява да се променя начинът, по който обекта изпълнява дадено действие, или по точно какъв алгоритъм използва. За да постигне това, шаблонът *"Стратегия" (Strategy)* използва базов интерфейс **`Strategy`** и няколко класа, които имплементират този интерфейс. Контекстът държи референция към обект от тип въпросния базов интерфейс и според изискванията си, използва различни наследници. *"Мост" (Bridge)* ви позволява да разделите интерфейса от имплементацията. Тя варира по същия начин както при *"Стратегия" (Strategy)*. Тук базовият клас, който варира, ще наречем **`Implementation`**. Разликата е, че *"Мост" (Bridge)* дефинира интерфейс (на контекста), към който се обръщат всички негови наследници, без да го предефинират (тоест контекстът има собствена имплементация). *"Мост" (Bridge)* използва наследниците на **`Implementation`** по същия начин, по който *"Стратегия" (Strategy)* ипзолвза наследниците на **`Strategy`**. Така, когато се промени класът, наследник на **`Implementation`**, който контекстът (мост) използва по време на изпълнение на програмата, наследниците на контекст (мост) няма да разберат, тъй като имплементацията (на контекста) в базовия клас остава непроменена.

UML class diagram:



Примерен код:

```
// Strategy pattern -- Structural example

using System;


namespace DoFactory.GangOfFour.Strategy.Structural
{
  /// <summary>
  /// MainApp startup class for Structural
  /// Strategy Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      Context context;


      // Three contexts following different strategies
      context = new Context(new ConcreteStrategyA());
      context.ContextInterface();


      context = new Context(new ConcreteStrategyB());
```

```
    context.ContextInterface();

    context = new Context(new ConcreteStrategyC());
    context.ContextInterface();

    // Wait for user
    Console.ReadKey();
  }
}

/// <summary>
/// The 'Strategy' abstract class
/// </summary>
abstract class Strategy
{
  public abstract void AlgorithmInterface();
}

/// <summary>
/// A 'ConcreteStrategy' class
/// </summary>
class ConcreteStrategyA : Strategy
{
  public override void AlgorithmInterface()
  {
    Console.WriteLine(
      "Called ConcreteStrategyA.AlgorithmInterface()");
  }
}

/// <summary>
/// A 'ConcreteStrategy' class
/// </summary>
class ConcreteStrategyB : Strategy
{
  public override void AlgorithmInterface()
  {
```

```
      Console.WriteLine(
        "Called ConcreteStrategyB.AlgorithmInterface()");
    }
  }

  /// <summary>
  /// A 'ConcreteStrategy' class
  /// </summary>
  class ConcreteStrategyC : Strategy
  {
    public override void AlgorithmInterface()
    {
      Console.WriteLine(
        "Called ConcreteStrategyC.AlgorithmInterface()");
    }
  }

  /// <summary>
  /// The 'Context' class
  /// </summary>
  class Context
  {
    private Strategy _strategy;

    // Constructor
    public Context(Strategy strategy)
    {
      this._strategy = strategy;
    }

    public void ContextInterface()
    {
      _strategy.AlgorithmInterface();
    }
  }
}
```

Output

```
Called ConcreteStrategyA.AlgorithmInterface()
Called ConcreteStrategyB.AlgorithmInterface()
Called ConcreteStrategyC.AlgorithmInterface()
```