Factory Patterns - простой заводской шаблон

Снеш Праджапати

25 сен 2016 CPOL

В этой серии статей мы узнаем о различных шаблонах проектирования фабрики. Существует три вида заводских шаблонов проектирования, а именно: простой заводской шаблон, заводской шаблон метода и абстрактный заводской шаблон.

Скачать исходный код - 12,2 КБ

Вступление

В этой серии статей мы узнаем о различных шаблонах проектирования фабрики. Существует три вида заводских шаблонов проектирования, а именно: простой заводской шаблон, заводской шаблон метода и абстрактный заводской шаблон. Мы подробно разберемся в этих трех шаблонах, узнав, как реализовать, когда реализовать, и впоследствии мы поймем различия между ними. Простой фабричный шаблон не является частью книги Gang of Four (GoF), но фабричный шаблон метода и абстрактные фабричные шаблоны являются частью этой стандартной книги.

Чтобы сохранить разумный размер статей, эта серия статей разделена на три части.

Контуры

Часть 1 - Простой заводской шаблон

- Что такое шаблоны дизайна
- Как шаблоны дизайна помогают нам
- Простой заводской шаблон
- Проблемы без простого заводского шаблона
- Разрешение с простым заводским шаблоном
- Проблема с простым заводским шаблоном

Часть 2 - Шаблон заводского метода

- Что такое шаблон фабричного метода
- Понять определение на примере
- Когда использовать шаблон фабричного метода
- Преимущества паттерна фабричного метода

Часть 3 - Абстрактный узор фабрики

- Что такое абстрактный узор фабрики
- Когда использовать абстрактный заводской паттерн
- Понять определение на примере
- Проблема с шаблоном абстрактной фабрики

Что такое шаблоны дизайна

Как говорится в Википедии : «Шаблон проектирования - это универсальное многократно используемое решение часто встречающейся проблемы в заданном контексте при разработке программного обеспечения».

Другими словами, мы можем сказать, что Design Pattern предоставляет проверенное решение для решения часто встречающихся проблем в программных (как правило, больших) приложениях. Реализация определенного шаблона проектирования в любом приложении увеличивает гибкость, ремонтопригодность и удобочитаемость приложения. *Прежде чем реализовывать какой-либо шаблон проектирования в приложении, сначала мы должны уяснить проблему, которую можно решить с помощью конкретного шаблона проектирования.*

Как шаблоны дизайна помогают нам

Приложение можно разрабатывать без реализации шаблонов проектирования. Без использования шаблона проектирования первоначально время разработки приложения будет меньше, но по мере роста функциональных возможностей приложения вы поймете, что очень сложно изменить, поддерживать или понимать дизайн. Вот где шаблон дизайна помогает нам определить общую проблему и решить ее наилучшим образом.

Первая и самая важная часть при работе с шаблонами проектирования - это понимание контекста и точной проблемы в приложении. Как только проблема идентифицирована, легко определить, какой шаблон проектирования следует использовать для решения идентифицированной проблемы.

Что такое простой заводской шаблон

Согласно определению из википедии , Factory Pattern - это «Фабрика - это объект для создания других объектов». Simple Factory Pattern - это `Factory`класс в его простейшей форме (по сравнению с Factory Method Pattern или Abstract Factory Pattern). Другими словами, мы можем сказать: ***в простом шаблоне фабрики у нас есть фабричный класс, у которого есть метод, который возвращает различные типы объектов на основе заданных входных данных.***

Давайте разберемся на примере:

Обзор примера

Чтобы понять на практике Simple Factory Pattern, мы возьмем пример компании Electric, которая производит различные типы вентиляторов, мы назовем это `FanFactory`. Но сначала мы реализуем этот сценарий без использования простого заводского шаблона, а затем увидим проблемы и способы их решения с помощью этого шаблона.

Приведенная ниже программа представляет собой простое консольное приложение, в этой программе `Main`метод используется как клиент, который создает файл `TableFan`. В простейшей реализации мы видим, что клиент может создать файл `TableFan`напрямую в соответствии с потребностями (без фабрики).

C # Копировать код

```
class Program
{
   static void Main(string[] args)
    {
           TableFan fan = new TableFan();
           fan.SwitchOn();
     }
```

```
}

class TableFan { }
```

Проблемы без простого заводского шаблона

В приведенном выше примере мы не использовали какой-либо шаблон, и приложение работает нормально. Но если мы подумаем о будущих возможных изменениях и внимательно присмотримся, мы можем предвидеть следующие проблемы с текущей реализацией:

1. В текущем приложении, где нужен конкретный вентилятор, он создается с использованием конкретного класса. В будущем, если будет внесено какое-либо изменение в название класса / будет предложен другой конкретный класс, вы должны будете внести изменения во всем приложении. Например, вместо `TableFan`класса нам нужно ввести `BasicTableFan`(который теперь следует использовать вместо старого `TableFan`класса) и `PowerTableFan`классы. Таким образом, любые изменения в отношении `Fan`классов затруднят поддержку кода и потребуют множества изменений во многих местах.

2. В настоящее время, когда клиент создает `TableFan`объект, конструктор `TableFan`класса не принимает никаких аргументов, поэтому `Fan`процесс создания прост. Но позже `TableFan`процесс создания объекта изменится, и что, если конструктор `Fan`класса ожидает два объекта в качестве аргументов. Затем каждое место в коде клиента должно вносить изменения независимо от того, где `Fan`был создан объект. Например:

   C # Копировать код

   ```
   TableFan fan = new TableFan(Moter moter, BladType bladType);
   ```

3. В `TableFan fan = new TableFan(Moter moter, BladType bladType);`коде , когда `Fan`создается объект, она принимает два типа `Moter`, `BladType`. В приведенном выше коде клиент знает о классах, а также знает процесс создания объекта, который должен быть внутренней деталью `FanFactory`. Мы должны избегать раскрытия таких деталей создания объектов и внутренних классов клиентскому приложению.

4. В некоторых случаях управление жизненным циклом созданных объектов должно быть централизованным, чтобы обеспечить согласованное поведение в приложении. Этого нельзя сделать, если клиент волен создавать конкретный объект так, как он хочет. Такой сценарий часто встречается в фреймворках Cache Management и DI.

5. В C # и Java конструктор класса должен иметь то же имя, что и имя этого класса. Допустим, для конструктора необходимо иметь описательные имена, например `CreateTableFanByModelNumber(int modelNumber)`. В `TableFan`классе, в лучшем случае, у нас может быть такой конструктор, `Fan(int modelNumber)`но имя не так информативно, как его предназначение. Другой аналогичный пример по проблеме описательного имени доступен также на этой странице википедии .

Разрешение с простым заводским шаблоном

Чтобы решить вышеупомянутые проблемы, мы можем использовать Simple Factory Pattern, потому что этот шаблон подходит для решения вышеупомянутых проблем. Далее мы продолжим с тем же примером и изменим существующий код. Когда мы реализуем Simple Factory Pattern, необходимо создать класс, который будет иметь метод для возврата запрошенного экземпляра объекта. Давайте создадим класс с именем " `FanFactory`", который будет реализовывать интерфейс с именем " `IFanFactory`". В этом интерфейсе есть метод, `IFan CreateFan(FanType type);`который принимает `enum FanType`и возвращает соответствующий экземпляр `Fan`.

Теперь `Client`не будем знать о конкретных классах вроде `TableFan`или `CeilingFan`. Клиент будет использовать `FanType enum`и `IFan interface`. На основе переданного `enum`аргумента при вызове `CreateFan`метода " " `FanFactory`будет возвращен экземпляр желаемого вентилятора. Ниже приведен модифицированный код приложения:

C # Сжать ▲ Копировать код

```
enum FanType
{
    TableFan,
    CeilingFan,
```

```
    ExhaustFan
}

interface IFan
{
    void SwitchOn();
    void SwitchOff();
}

class TableFan : IFan {.... }

class CeilingFan : IFan {.... }

class ExhaustFan : IFan {..... }

interface IFanFactory
{
    IFan CreateFan(FanType type);
}

class FanFactory : IFanFactory
{
    public IFan CreateFan(FanType type)
    {
        switch (type)
        {
            case FanType.TableFan:
                return new TableFan();
            case FanType.CeilingFan:
                return new CeilingFan();
            case FanType.ExhaustFan:
                return new ExhaustFan();
            default:
                return new TableFan();
        }
    }
}

//The client code is as follows:

static void Main(string[] args)
{
        IFanFactory simpleFactory = new FanFactory();
        // Creation of a Fan using Simple Factory
        IFan fan = simpleFactory.CreateFan(FanType.TableFan);
        // Use created object
        fan.SwitchOn();

        Console.ReadLine();
 }
```

Теперь мы **подтвердим, как Simple Factory Pattern решил все вышеупомянутые проблемы в** разделе «Проблема без реализации Simple Factory Pattern»:

1. Теперь клиент использует интерфейсы, и `FanFactory`поэтому, если мы изменим имя конкретного класса, `Fan`который создается для данного `enum`значения, нам нужно изменить только одно место, то есть внутри `CreateFan`метода " " объекта `FanFactory`. Клиентский код не затрагивается.

2. Если позже `Fan`процесс создания объекта изменится и конструктор `Fan`класса ожидает два или более объекта в качестве аргументов, нам нужно будет изменить только " `CreateFan`" внутри метода `FanFactory`. Клиентский код не затрагивается.

3. При использовании `FanFactory`все внутренние детали будут скрыты от клиента. Так что это хорошо с точки зрения абстракции и безопасности.

4. При необходимости мы можем написать логику для управления временем жизни вместе с централизованным созданием объектов `FanFactory`.

5. При использовании `FanFactory` мы можем просто иметь методы с разными и более описательными именами, которые возвращали бы объект `IFan`. В нашем примере приложения `FanFactory` может быть открыт общедоступный метод `CreateTableFanByModelNumber(int modelNumber)` для клиента.

Проблема с простым заводским шаблоном

Скажем, в будущем, если `FanFactory` нужно будет сделать новый тип вентилятора так `WallFan` же. Чтобы принять это новое требование, мы должны изменить `CreateFan` метод и добавить один `switch` регистр для `WallFan` типа. Если снова `Fan` вводится новый вид , то снова нужно добавить еще один случай. Это будет нарушением принципа открытия и закрытия SOLID . В следующей статье мы увидим, как мы можем решить эту проблему с нарушением с помощью Factory Method Pattern.

Заключение

В этой статье у нас было пошаговое руководство по изучению Simple Factory Pattern и его использованию. Мы поняли контекст Simple Factory Pattern и то, как его использовать для повышения удобства сопровождения приложения. Но это нарушает принцип Open Close Principle, поэтому в следующей статье мы увидим, как можно улучшить дизайн с помощью Factory Method Pattern . Спасибо за прочтение! Мы будем рады вашим комментариям и предложениям по улучшению.

использованная литература

- Обсуждение мотивации Simple Factory
- Блог Кори Бродерика
- Блог компьютерщиков-программистов
- Статья о заводских выкройках

Лицензия

Делиться

об авторе

Снеш Праджапати

Разработчик программного обеспечения

Индия

Я разработчик программного обеспечения, работающий над технологиями Microsoft. Меня интересует изучение и распространение новейших технологий.

Комментарии и обсуждения

Вы должны войти в систему, чтобы использовать эту доску сообщений.

Search Comments

Первая Предыдущая Следующая

По-прежнему одна из лучших статей о фабрике
_D корпус 1-фев-19 2:31

Сжать ▲ Копировать код

```
Still one of the best article on factory
```

«Если вы не можете объяснить это просто, вы недостаточно хорошо это понимаете»,

- Альберт Эйнштейн.

Войти · Просмотр темы

Плохой пример
Член 13833966 18.05.18 7:09

Спасибо, Снеш, у меня есть вопросы
Член 13601376 31.12.17 9:36

Спасибо
Дарин Сиз 13 октября 16 17:58

Re: Спасибо
Снеш Праджапати 13 октября 16 18:22

Спасибо
Реджеп СЕЛЛИ 7-окт-16 1:11

Re: Спасибо
Снеш Праджапати 9 октября 16 19:00

Ваш код на шаг впереди
Stefan_Lang 28-сен-16 2:35

Re: Ваш код на шаг впереди
Snesh Prajapati 28-Sep-16 6:56

Re: Ваш код на шаг впереди
Stefan_Lang 29-Sep-16 0:20

Re: Ваш код на шаг впереди
Snesh Prajapati 29-Sep-16 7:27

Re: Ваш код на шаг впереди
Vaso Elias 3-Oct-16 4:55

Ожидание шаблона фабричного метода и абстрактного фабричного шаблона
DeyChandan 27-Sep-16 23:52

Re: Ожидание шаблона фабричного метода и абстрактного фабричного шаблона
Snesh Prajapati 28-Sep-16 6:58

Re: Ожидание шаблона фабричного метода и абстрактного фабричного шаблона
Snesh Prajapati 2-Oct-16 16:12

Re: Ожидание шаблона фабричного метода и абстрактного фабричного шаблона
DeyChandan 2-Oct-16 21:41

Re: Ожидание шаблона фабричного метода и абстрактного фабричного шаблона
Snesh Prajapati 2-Oct-16 22:38

Re: Ожидание шаблона фабричного метода и абстрактного фабричного шаблона
Snesh Prajapati 9-Oct-16 18:54

Под вопросом
Member 12692295 26-Sep-16 11:28

Re: под вопросом
Snesh Prajapati 26-Sep-16 16:49

Мой голос 5
AlessandroF73 26-Sep-16 5:23

Re: Мои 5 голосов
Snesh Prajapati 26-Sep-16 6:56

Мой голос 5
TinTinTiTin 25-Sep-16 21:15

Re: Мои 5 голосов
Snesh Prajapati 25-Sep-16 21:42

Обновить 1

Общие Новости Предложение Вопрос Ошибка Ответ Шутка Похвала Rant Администратор

Используйте Ctrl + Left / Right для переключения сообщений, Ctrl + Up / Down для переключения цепочек, Ctrl + Shift + Left / Right для переключения страниц.