

# Pattern “Specification”

---

*Прочитать статью на сайте*

## Disclaimer

Эта статья по большей части адресована новичкам в программировании, которые знакомы лишь с некоторыми шаблонами проектирования или не знакомы с ними вовсе.

## О шаблонах проектирования

Возьмем определение шаблона проектирования из википедии.

**Шаблоны проектирования, паттерны проектирования** (англ. *design pattern*) — это многократно применяемая архитектурная конструкция, предоставляющая решение общей проблемы проектирования в рамках конкретного контекста и описывающая значимость этого решения. Паттерн не является законченным образцом проекта, который может быть прямо преобразован в код.

Если передать это с помощью примера, то можно представить, что вы решаете задачу «по аналогии». Или, например, решение того же самого уравнения, но с другими конкретными числами.

Шаблон «Спецификация» – является шаблоном поведения приложения. Результатом выполнения будет являться переменная булевого типа, подавая которую на вход оператора условного перехода вы управляете поведением программы.

С помощью описанных ниже приемов вы сможете:

- Сделать свой код более читабельным и кратким;
- Избежать дубликации кода;
- Легче вносить изменения в реализацию.

Как всегда, лучше всего работу сложного механизма показывает конкретный пример. Начнем с более простых вещей, постепенно переходя к сложным. Постараюсь избежать ситуации, когда раскрыт только простейший пример, а как это применять в реальной жизни – непонятно.

## Область применения

Шаблон «Спецификация» нужно применять при ветвистости алгоритма, когда происходит множество проверок. Когда вы осуществляете однотипные проверки условий в разных местах программы. Я думаю, на дальнейших примерах, вы сможете в полной мере прочувствовать, где лучше всего использовать данный паттерн.

Итак, допустим у нас есть класс для товаров **Ware**.

```
public class Ware {
    public string Articul { get; set; }
    public string Name { get; set; }
    public DateTime ProduceDate { get; set; }
    public DateTime BestBefore { get; set; }
}
```

и есть доменный сервис **WareCalculation**, который каким-либо образом обрабатывает списки товаров. Допустим, у нас есть метод который позволяет подсчитать количество товаров, у которых подходит к концу срок годности. В самом простом случае это будет реализованно следующим образом:

```
public class WareCalculation {
    public int CountExpiredWares(List<Ware> wares) {
        return wares.FindAll(w => w.BestBefore < DateTime.Now.AddMonths(4)).Count;
    }
}
```

Для контроля корректности применения последующего рефакторинга и паттернов, должны быть тесты. В нашем случае достаточно следующего теста:

```
[TestMethod]
public void ShouldCountExpiredWares() {
    var list = new List<Ware> {
        Create.Ware(w => w.BestBefore =
DateTime.Now.AddMonths(6)),
        Create.Ware(),
        Create.Ware()
    };

    var expiredWare = new WareCalculation().CountExpiredWares(list);
    Assert.AreEqual(2, expiredWare);
}
```

Это у нас простой вырожденный пример, на котором я покажу саму технику использования. В тесте использован DSL (Domain Specific Language) про который можно прочитать в более ранних статьях. В исходном коде можно будет найти его реализацию.

В доменном методе у нас есть условие, по которому находятся нужные товары. Это условие можно заменить спецификацией. Общий вид класса спецификации выглядит так:

```
public abstract class Specification<T>{
    public abstract bool IsSatisfiedBy(T item);
}
```

Как вы видите, класс Specification является параметризованным, а иначе как проверять условия в методе IsSatisfiedBy? Ведь мы не знаем, какой тип объекта придет к нам в качестве параметра. В своей программе вы не ограничитесь только одной спецификацией, но их

использование должно быть унифицированным, т.е. чтобы было легко передавать в качестве параметра. Именно для этого нам потребовалось объявить класс абстрактным.

Сейчас можно заняться рефакторингом метода `CountExpiredWares`. Для этого мы создадим новый класс спецификации **WareExpirationSpecification**.

```
public class WareNearExpirationSpec: Specification<Ware> {
    public bool IsSatisfiedBy(Ware item) {
        return false;
    }
}
```

Помним, что «нет теста – нет кода» и поэтому метод у нас всегда возвращает `false`, пока не напишем тесты.

```
[TestMethod]
public void WareNearExpirationSpecificationTest() {
    Assert.IsTrue(new WareNearExpirationSpec().IsSatisfiedBy (
        Create.Ware(w => w.BestBefore = DateTime.Now)));

    Assert.IsFalse(new WareNearExpirationSpec().IsSatisfiedBy (
        Create.Ware(w => w.BestBefore =
DateTime.Now.AddMonths(10))));

    Assert.IsTrue(new WareNearExpirationSpec().IsSatisfiedBy (
        Create.Ware(w => w.BestBefore = DateTime.Now.AddMonths(-
10))));
}
```

И реализация спецификации:

```
public class WareNearExpirationSpec : Specification<Ware> {
    public bool IsSatisfied(Ware item) {
        return item.BestBefore < DateTime.Now.AddMonths(4);
    }
}
```

В теле доменного метода теперь можно использовать спецификацию.

```
public int CountExpiredWares(List<Ware> wares) {
    var wareNearExpirationSpec = new WareNearExpirationSpec();
    return wares.FindAll(wareNearExpirationSpec.IsSatisfiedBy).Count;
}
```

На мой взгляд читабельность метода улучшилась. Но мы на этом не остановимся и продолжим улучшать спецификацию.

## Улучшение спецификации

На данном этапе может казаться, что в целом все хорошо, какого-то избытка кода нет, даже наоборот просматривается явное его увеличение, но это временно. Количество месяцев в проверке жестко (явно) прописано и при таком подходе будет сложно выбирать товары которые

лучше послать в соседний супермаркет для быстрой реализации, а какие отправить в дальнее путешествие.

В таком случае, эталон для проверки передается в конструктор спецификации.

```
public class WareNearExpireDateSpec : ISpecification<Ware> {
    private readonly int monthsLeft;

    public WareNearExpireDateSpec(int monthsLeft) {
        this.monthsLeft = monthsLeft;
    }

    public bool IsSatisfied(Ware item) {
        return item.BestBefore < DateTime.Now.AddMonths(monthsLeft);
    }
}
```

## Композиция спецификаций

Предположим, что у нас есть метод в доменном сервисе, который осуществляет подсчет товаров по каким-то критериям для специальной акции.

```
public int CountSpecialWares(List<Ware> wares) {
    return wares.FindAll(w =>
        w.BestBefore < DateTime.Now.AddMonths(4) &&
        w.BestBefore > DateTime.Now.AddMonths(2) &&
        w.Articul.StartsWith("Special") &&
        w.ProduceDate > DateTime.Now.AddMonths(-2)
    ).Count;
}
```

Акций может быть сколько угодно и по разным условиям, так же может быть и различное число условий на товар. Неужели придется писать бесчисленное количество методов с разным входным набором параметров или спецификаций? Все нет. Шаблон «Спецификация» позволит нам написать примерно следующий код метода:

```
public int CountSpecialWares(List<Ware> wares, ISpecification<Ware>
specification) {
    return wares.FindAll(specification.IsSatisfiedBy).Count;
}
```

Вторым параметром будет идти любая комбинация спецификаций, которая может быть применима для класса Ware.

Для комбинирования спецификаций предлагаю сделать 2 класса,

- AndSpecification
- OrSpecification

которые будут являться контейнером для конкретных реализаций паттерна. Они во многом будут схожи, так что сразу можно выделить базовый класс для них CompositeSpecification<T>. Этот класс так же пометим как абстрактный. В нем будет содержаться коллекция спецификаций, методы для

их добавления и удаления, а так же свойство, которое будет возвращать содержимое в виде не редактируемой коллекции. В конструктор можно подавать любое количество спецификаций.

```
public abstract class CompositeSpecification<T> : Specification<T> {
    protected readonly List<Specification<T>> specifications;

    protected CompositeSpecification(params Specification<T>[]
specifications) {
        this.specifications = new List<Specification<T>>(specifications);
    }

    public ReadOnlyCollection<Specification<T>> Specifications {
        get { return specifications.AsReadOnly(); }
    }

    public CompositeSpecification<T> Add(Specification<T> specification)
{
        specifications.Add(specification);
        return this;
    }

    public CompositeSpecification<T> Remove(Specification<T>
specification) {
        specifications.Remove(specification);
        return this;
    }
}
```

После этого классы AndSpecification и OrSpecification будут достаточно простыми. В первом из классов, необходимо будет перебрать все спецификации и убедиться, что переданный элемент удовлетворяет одновременно всем спецификациям.

```
public class AndSpecification<T> : CompositeSpecification<T> {
    public AndSpecification(params Specification<T>[] specifications)
        : base(specifications) {
    }

    public override bool IsSatisfiedBy(T obj) {
        return specifications.All(specification =>
specification.IsSatisfiedBy(obj));
    }
}
```

Для класса спецификации OrSpecification проверяем до первого успешно пройденного условия проверки.

```
public class OrSpecification<T> : CompositeSpecification<T> {
    public OrSpecification(params Specification<T>[] specifications) :
base(specifications) { }

    public override bool IsSatisfiedBy(T obj) {
        return specifications.Any(specification =>
specification.IsSatisfiedBy(obj));
    }
}
```

```
}
```

Я предположу что у вас уже написаны отдельные спецификации на условия из теста в начале этого раздела. В любом случае эти спецификации есть в приложении к статье. Примером использования может послужить тест:

```
[TestMethod]
public void CountSpecialWaresSpec() {
    var list = new List<Ware> {
        Create.Ware(w => {
            w.BestBefore =
                DateTime.Now.AddMonths(9);
            w.Articul =
                "Specia...";
            w.ProduceDate =
                DateTime.Now.AddMonths(-2);
        }
    ),
    Create.Ware(),
    Create.Ware()
};

    var actionSpec = new AndSpecification<Ware>(
        new WareArticulStartsWithSpec("Spec"),
        new WareExpireDateBetweenSpec(8, 10),
        new WareProduceDateMoreSpec(1));

    var expiredWare = new WareCalculation().CountSpecialWares(list,
actionSpec);
    Assert.AreEqual(1, expiredWare);
}
```

При таком подходе делать гибкие пользовательские настройки, по результатам которых формировался бы необходимый набор спецификаций. При чтении кода вы сразу сможете определить смысл проверки, не вникая в ветвистые условия проверки через множества полей. К тому же, спецификации в реальных программах осуществляют и более комплексные проверки которые писать каждый раз очень утомительно. Еще более скучно будет в них вникать через месяц другой.

Еще один пример, который может быть полезен в реальности – поиск по шаблону. Допустим, у вас есть форма для поиска товара, где заполняются все или только некоторые поля. По нажатию на кнопку «Поиск», вы можете сформировать новый товар с информацией введенной пользователем и передать его параметром в конструктор спецификации.

При поиске по базе или иному источнику данных, вам надо лишь в условие поиска добавить спецификацию.

Plusom такого подхода является то, что при изменении доменного объекта, вам надо будет изменить только метод `IsSatisfiedBy` в паттерне, а не выискивать по всему коду использование поиска и его редактирование. Если вы пропустите какое-то место, компилятор вам не подскажет, ведь весь код правилен, все указанные поля существуют, а то, что появились новые и они существенны для поиска его не волнует.