№ 17-18 Применение порождающих паттернов проектирования

Задание

1) На основе лабораторной №16 разработайте типы (могут быть выделены жирным в вариантах задания, а также могут быть дополнительные типы) и интерфейсы, определите нужные отношения между ними и функции. Напишите консольное демо.

   Добавление шаблонов:

2) Создание и изменение объектов приложения сделайте на основе **Abstaract Factory и Builder.** Можно добавить функцию генерации *n* случайно заполненных объектов через запрос.
3) Создайте пользовательский тип, который хранит настройки приложения (цвет фона, шрифт, размер и т.п.) и сделайте его **Singleton**. Вместо настроек **Singleton** можно сделать тип, который есть в вашем проекте. Например, **Администратор**, **Тьютер** и т.п.
4) Добавьте функцию создания клона объекта на основе шаблона **Prototype**.

Вопросы

1. Какие типы паттернов бывают?
2. Нарисуете диаграмму классов и поясните принцип работы паттерна Singleton. Назовите условия применения.
3. Как сделать потокобезопасную, с отложенной реализацией версию Singleton.
4. Нарисуете диаграмму классов и поясните принцип работы паттерна Factory Method. Назовите условия применения
5. Нарисуете диаграмму классов и поясните принцип работы паттерна Abstract Factory. Назовите условия применения
6. Нарисуете диаграмму классов и поясните принцип работы паттерна Prototype. Назовите условия применения
7. Нарисуете диаграмму классов и поясните принцип работы паттерна Builder. Назовите условия применения
8. Что такое Пул объектов (Object pool), в чем суть паттерна?
9. Какое основное назначение паттерна отложенная инициализация (Lazy initialization)?

https://proglib.io/p/creational-patterns/

https://sourcemaking.com/design_patterns/prototype

https://live13.livejournal.com/462582.html

http://design-pattern.ru/

https://www.oodesign.com/abstract-factory-pattern.html

https://tproger.ru/translations/design-patterns-simple-words-1/

Строитель (Builder)

Строитель позволяет отделить процесс создания сложного объекта от его реализации. При этом, результатом одних и тех же операций могут быть различные объекты.

Данный шаблон используется в случае, если:

- процесс создания объекта можно разделить на части (шаги);
- (и) алгоритм этого процесса не должен зависеть от того, из каких частей состоит объект;
- (и) конструирование должно обеспечивать возможность создавать различные объекты.

Шаблон Строитель включает двух участников процесса:

- Строитель (Builder) – предоставляет методы для сборки частей объекта, при необходимости преобразовывает исходные данные в нужный вид, создает и выдает объект;
- Распорядитель (Director) – определяет стратегию сборки: собирает данные и определяет порядок вызовов методов Строителя.

Задача распорядителя – сокрытие стратегии сборки. Это позволит, при необходимости, модифицировать или даже полностью менять ее, не затрагивая остальной код.

Так же Распорядитель, как правило, отвечает за получение данных для конструирования. И уже потом, Строитель преобразовывает их в вид, необходимый для порождаемого объекта. Такое разделение связано с тем, что создаваемый объект скрыт от Распорядителя и, кроме того, может не уметь работать с форматом исходных данных.

Реализация шаблона в общем виде

- определяем шаги конструирования сложного объекта, и на их основе разрабатываем интерфейс Строителя *IBuilder*;

- если планируется несколько стратегий сборки, то создаем интерфейс Распорядителя *IDirector*;
- разрабатываем класс Распорядителя *MyDirector* (реализующий *IDirector*), работающий со Строителями через интерфейс *IBuilder*;
- создаем класс Строителя *MyBuilder*, реализующий интерфейс *IBuilder* и метод получения результата;
- в клиентском коде экземпляру *MyDirector* передаем интерфейс *IBuilder* экземпляра *MyBuilder*;
- запускаем процесс сборки, вызвав метод Распорядителя;
- получаем созданный экземпляр *MyProduct* у используемой реализации Строителя *MyBuilder*.

Возможно возникнет вопрос о необходимости создания интерфейсов. Такой подход позволит в дальнейшем, изменяя реализации Строителя и Распорядителя, влиять результат конструирования.

Примеры реализации. Создание различных конфигураций одного объекта

Рассмотрим ситуацию, когда Строитель создает различные конфигурации объекта.

Для примера, возьмем часть кода сайта, ответственную за генерацию страниц. Необходимо создать объект *Page*, который содержит HTML код для выбранной страницы.

Определим шаги конструирования страницы: создаем шапку (Header), добавляем элементы меню (MenuItems), выводим публикации (Post) и завершаем страницу кодом подвала (Footer). Эти четыре шага и будут определять интерфейс Строителя:

```
public interface IPageBuilder
{
    void BuildHeader(HeaderData header);
    void BuildMenu(MenuItems menuItems);
    void BuildPost(PostData post);
    void BuildFooter(FooterData footer);
}
```

Поскольку стратегия сборки будет одна, приступим к реализации Распорядителя. Его использование не имеет смысла без экземпляра Строителя. Поэтому конструктор будет требовать передачи ему интерфейса *IPageBuilder*.

Создадим метод *BuildPage()*, определяющий стратегию сборки страницы. В нем получим данные для выбранной страницы (экземпляр класса *PageData*) и по шагам вызовем методы Строителя.

```
public class PageDirector
```

```
{
    private readonly IPageBuilder _builder;

    private HeaderData GetHeader(int pageId) { /* SKIPPED */ }
    private MenuItems GetMenuItems(int pageId) { /* SKIPPED */ }
    private IEnumerable<PostData> GetPosts(int pageId) { /* SKIPPED */ }
    private FooterData GetFooter(int pageId) { /* SKIPPED */ }

    public PageDirector(IPageBuilder builder)
    {
        this._builder = builder;
    }

    public void BuildPage(int pageId)
    {
        this._builder.BuildHeader(this.GetHeader(pageId));
        this._builder.BuildMenu(this.GetMenuItems(pageId));

        foreach (PostData post in this.GetPosts(pageId))
        {
            this._builder.BuildPost(post);
        }

        this._builder.BuildFooter(this.GetFooter(pageId));
    }
}
```

Осталось создать класс Строителя. Реализуем интерфейса *IPageBuilder* и напишем метод*GetResult()*, возвращающий результат сборки. Для упрощения примера, будем просто передавать данные в создаваемый экземпляр класса *Page*. Обратите внимание на его объявление с использованием readonly. Это гарантирует, что ни один из шагов не пересоздаст объект.

```
public class PageBuilder : IPageBuilder
{
    private readonly Page _page = new Page();
    public void BuildHeader(HeaderData header) { this._page.AddHeader(header); }
    public void BuildMenu(MenuItems menuItems) { this._page.SetMenuItems(menuItems); }
    public void BuildPost(PostData post) { this._page.AddPost(post); }
    public void BuildFooter(FooterData footer) { this._page.AddFooter(footer); }
    public Page GetResult() { return this._page; }
}
```

Все готово к использованию, например вот так:

```
public void PostPage(int pageId)
{
    PageBuilder pageBuilder = new PageBuilder();
    PageDirector pageDirector = new PageDirector(pageBuilder);
    pageDirector.BuildPage(pageId);
    Page page = pageBuilder.GetResult();
    this.Post(page);
}
```

Создали Строителя и Распорядителя, приказали создать объект и забрали результат и отправили его на вывод.