# GoF паттерны на платформе .NET

У нашей индустрии есть одна интересная черта: когда появляется новый инструмент или технология, то часто ею настолько увлекаются, что начинают забывать про старые и проверенные вещи. Возможно это связано с надеждой найти в конце концов серебряную пулю и каждый в душе надеется, что вот, благодаря "этой штуке", хорошая система родится сама собой, а не, как обычно, благодаря крови, поту и опыту команды разработчиков.

Так, когда в начале 90-х на арене мира разработки программного обеспечения появились паттерны проектирования, многие начали мечтать о том, что благодаря им даже бизнес-пользователи и ~~1С программисты~~ смогут собирать приложение из готовых кирпичиков. Довольно быстро стало понятным, что планы были через чур оптимистичными, и начали искать другие подходы. Так появилось "программирование через конфигурацию", пламенно воспетую Хантом и Томасом в их "Программисте-прагматике" (\*). Затем появились IoC- или DI-контейнеры (IoC - Inversion of Control, или DI - Dependency Injection), и начался новый этап создания "слабосвязанных приложений". У многих появилась привычка выделять интерфейсы (\*\*) не задумываясь, а количество зависимостей у класса легко начало переваливать за 5-6. В результате, разбираться в приложении стало еще сложнее, поскольку прямые связи между классами стали заменяться косвенными.

(\*) Сноска: "Программист-прагматик. Путь от подмастерья к мастеру" Энди Хант и Дейв Томас.

(\*\*) Сноска: речь идет о интерфейсах в таких языках программирования, как C# или Java.

Поиск идеального инструмента, языка, принципа или методологии разработки – это Святой Грааль в разработке ПО. Все хотят найти идеальный инструмент, который позволит справиться со сложностью современных систем и навести порядок в том хаосе, который творится в мире программирования. Но может быть, вместо того, чтобы каждый раз хвататься за что-то новое, как за спасительную соломинку, стоит понять, что за этой соломинкой скрыто? Ведь если присмотреться, то новый инструмент очень часто оказывается лишь новой оберткой, в которой завернуты старые идеи.

## Отношение к паттернам проектирования

Большинство разработчиков ПО сходятся в мысли, что паттерны проектирования – вещь интересная, но далеко не все считают их полезными. Почему так вышло? Когда молодой разработчик сталкивается с новым инструментом, то он изо всех сил старается воспользоваться им по максимуму. В результате, он проходит определенные стадии развития, которые в случае паттернов проектирования выглядят так:

1-я стадия: Ух-ты-Ух-ты-Ух-ты! Я узнал, что такое паттерны! Класс! Когда и где я смогу ими воспользоваться?

2-я стадия: Ух-ты! Я отрефакторил (\*) старый код и вместо десяти строк кода воспользовался 7 паттернов! Вот как здорово!

3-я стадия: ух-ты. Ну, паттерны - это классная штука, но через пару месяцев мой прошлый рефакторинг уже не кажется таким уж полезным. Что-то я и сам начал путаться за всеми этими абстрактными фасадированными декораторами, завернутыми в синглтон.

4-я стадия: нет, паттерны – это хорошо, но нужно отталкиваться не от паттернов, а от решаемой задачи и уже исходя из проблемы выбирать подходящие решения. Паттерны – хорошо, но своя голова – на порядок лучше!

(\*) Сноска: речь идет о рефакторинге кода - изменении структуры программы без изменения ее поведения. Подрбонее об этом можно почитать в Сети, или в книге Мартина Фаулера "Рефакторинг. Улучшение существующего кода".

Есть разработчики, которые успешно прошли все четыре стадии и достигли "просветления", набивая шишки в разных проектах, на себе оценивая последствия использования тех или иных паттернов. Но ведь есть и те, кто пришел на проект, в котором царствовала вторая стадия использования паттернов и увидели решения простых задач невероятно изощренным способом.

Какое будет у вас отношение к паттернам при виде классов вроде AbstractSingletonProxyFactoryBean (\*) и приложений "Hello, World" следующего вида?

public class HelloWorld  
{  
 public static void Main(String[] args)  
 {  
 MessageBody mb = new MessageBody();  
 mb.Configure("Hello World!");  
 AbstractStrategyFactory asf = DefaultFactory.Instance;  
 MessageStrategy strategy = asf.CreateStrategy(mb);  
 mb.Send(strategy);  
 }  
}

Листинг 0.1 - Гипертрофированное использование паттернов проектирования

(\*)Сноска: Пример кода взят из обсуждения паттернов проектирования на shashdot: http://developers.slashdot.org/comments.pl?sid=33602&cid=3636102.

Подливает масла в огонь наше через чур наивное отношение к паттернам проектирования, которое отлично описал Джошуа Кериевски в своей книге "Рефакторинг с использованием шаблонов". "К сожалению, когда программисты смотрят на единственную диаграмму, сопровождающую каждый шаблон в книге Design Patterns (\*), они часто приходят к выводу, что приведенная диаграмма и есть способ реализации шаблона. Они бы гораздо лучше разобрались в ситуации, если бы внимательно прочитали самое интересное - примечание к реализации. Многие программисты берут в руки книгу Design Patterns, всматриваются в структурную диаграмму шаблона и начинают кодировать. Полученный код в точности отражает диаграмму, а не реализацию шаблона, наиболее полно соответствующую решаемой задаче."

(\*) Сноска: речь идет о самой знаменитой книге по паттернам проектирования - "Приемы объектно-ориентированного проектирования. Паттерны проектирования", написанная "бандой четырех" - Эрихом Гаммой, Ричардом Хелмом, Ральфом Джонсоном и Джоном Влисидесом.

Большинство экспертов в области разработки ПО сходятся на мысли, что главным инструментом разработчика является его голова и собственный опыт. Если в "умной книге" нарисована некоторая диаграмма классов, то это не значит, что нужно бросать все, и пробовать "втулить" ее в свой проект. Чтобы использовать паттерны проектированbя по максимуму, нужно вникнуть в его суть, понять, какую проблему он призван решить и каким образом он это делает. Понимание целей паттерна и контекста его использования позволит варьировать реализацию паттерна и лучше адаптировать его под свои нужды.

## Фреймворки паттернов

Описанные выше 4 стадии изучения паттернов относятся к молодым специалистам, но не меньшие проблемы ждут команду, если о паттернах внезапно узнает авторитетный член команды (\*). Что делает опытный специалист, когда знакомится с новыми принципами проектирования? Правильно, он старается их обобщить и поделится своим новым опытом с менее опытными бойцами. В результате появляются библиотеки или фреймворки паттернов проектирования.

(\*) Сноска: да, сегодня такая ситуация может показаться маловероятной, но она возможна. Многие матерые гуру набили шишки, решая практические задачи, не особенно задумываясь о теории. И если такой авторитет решит воспользоваться паттернами, то ждите их во всех будущих проектах.

Я не говорю, что это абсолютно неверная идея, но в большинстве случаев такой подход противоречит самому понятию паттернов проектирования. Есть редкие исключения, такие как библиотека Loki (\*) Андрея Александреску, когда библиотека представляет набор базовых решений для упрощения реализации основных паттернов проектирования в конкретном языке программирования. Но в подавляющем большинстве случаев выгоды от повторного использования таких библиотек будет очень мало, их качество будет невысоким, а решения, полученные на их основе будут чрезмерно сложными.

(\*) СНОСКА: дать ссылку на книгу Александреску и библиотеку Loki.

## Гибкость vs. Конкретность

У большинства паттернов проектирования есть некоторая каноническая реализация, а есть упрощенные/усложненные вариации. Как вы, наверное, знаете, у гибкости есть своя цена, которая может быть оправдана в одном случае и не иметь смысла в другом. Хотя при описании паттернов проектирования дается контекст, в котором его применение будет наиболее актуальным, лишь разработчик конкретного приложения может сказать, где проходит грань между сложностью и гибкостью, подходящая для его конкретного случая.

Большинство паттернов проектирования предназначены для получения расширяемости системы в определенной плоскости. При этом эта «плоскость» может быть полезной для одного приложения и вредной - для другого. Наличие иерархии наследование может быть просто добавлять сложности в простое приложение, но в случае библиотеки, может сделать решение через чур сложным.

Наследование является чрезвычайно полезным инструментом для расширения функционала, но оно приводит к сильной связи между базовым классом и наследниками. Для некоторых паттернов наследование является неотъемлемой частью реализации, для других - обузой, которая сделает решение громоздким. Канонические примеры большинства паттернов, приведенные "бандой четырех" содержат наследование. Но это не значит, что вы должны слепо этому следовать.

Во многих случаях, наследование должно применяться осознанно, и лишь тогда, когда обеспечиваемая гибкость является действительно необходимой.

### Для чего нужна еще одна книга о паттернах?

Паттерны не привязаны к платформе, но их типовая реализация несколько отличается от языка к языку, иногда из-за технических различий, иногда – из-за культурных. Я не хочу здесь поднимать вопрос о полезности паттернов проектирования. Это все равно, что поднимать вопрос о пользе исключений или многопоточности в .NET приложениях: хотите вы того или нет, но вам без них не обойтись. Так и с паттернами. Код любого .NET приложения просто пропитан паттернами в явном или неявном виде, и игнорирование этого факта вряд ли пойдет вам на пользу.

В этой книге я хочу вернуться к стандартным паттернам проектирования, посмотреть на то, что с ними случилось за последние двадцать лет и показать, в каком виде они применяются в современных .NET приложениях и в самом .NET Framework.

Я собираюсь рассмотреть использование паттернов проектирования на примере простого приложения импорта лог-файлов для последующего полнотекстового поиска (full text search). Подавляющее число паттернов естественным образом ложится на эту задачу, и они, в том или ином виде, были использованы в полноценной версии такого приложения под названием Application Insights, над которым я сейчас работаю в Майкрософт.