1. **Singleton**

Рассказать причины использования в C++, C# и Java (помогите, кто знает Java!!)

Давайте поговорим о C++. Бывает, в программе есть, например, какой—то глобальный объект, который не может быть создан при старте, ибо неизвестен порядок инициализации объектов. Ситуация: у вас есть глобальный (статический, static) объект **A**, который в конструкторе обращается к глобальному объекту **B** (показать код). Согласно стандарту С++, все static переменные инициализируются при старте приложения, однако стандарт не гарантирует, что сначала создастся объект **B**, а только потом объект **A**, и как следствие, есть большая вероятность, что объект **A** обратится к еще несозданному объекту **B**. В результате приходится делать какую-то функцию, которая создает объект (показать код). Мы обращаемся к этой функции, и она возвращает объект или создает и возвращает, если его еще нет. Все о чем я рассказал, это решение создавать объект, называется Singleton.

Стоит заметить, что этот прием, кроме решения вышеприведенной проблемы, обеспечивает еще и, так называемый, lazy loading, т.е. отложенное создание, создание по требованию (англ. on-demand).

Перенесемся в мир .Net. По спецификации, статические (static) члены класса инициализируются при первом (любом) обращении к классу, т.е. инициализируются отложено. Таким образом, возвращаясь к нашему примеру, объект **A**, пытаясь обратится в конструкторе к объекту **B**, гарантированно получит его, ибо если даже объект **B** еще не был создан, то CLR создаст его для нас (как и было оговорено, CLR инициализирует static’и отложено, по первому требованию, а не при старте программы, как это делается в С++) (показать код). Так в .Net нет проблем, скажете вы, зачем нам этот прием Singleton, который использует дополнительную функцию для создания? Ан нет. Здесь есть свои подводные камни. Дело в том, что CLR инициализирует статические члены класса лишь единожды, поэтому если ваш конструктор содержит какую-нибудь логику, которая потенциально может сгенерировать исключение, то это конец в работе вашего приложения (показать код). Итак, CLR при первом обращении к классу начинает инициализировать статическую переменную Instance (показать в коде), вызывает конструктор, а он генерирует исключение. Переменная Instance будет содержать null, и сколько бы вы к ней больше не обращались, этот null там так и останется навсегда. А при использовании функции, создающей глобальный объект класса, даже если и будет сгенерировано исключение, то при следующем вызове этой функции, будет еще одна попытка создать объект.

Итак, примерно таким путем пришли к приему программирования, который назвали Singleton (показать конечный вариант кода).

Стоит заметить, что у Singleton’а есть проблемы с многопоточностью и пути их решения. Однако это не затрагивает Singleton как понятие. Ключевые моменты мы разобрали. Сказано это лишь с той целью, чтобы мы не забывали об этом.

1. **Observer**

Представьте себе ситуацию, у вас есть список (лист), в который набиваются элементы, и вам нужно, чтобы по мере добавления элементов в этот лист обновлялся UI.

Рассказать что актуально в Java и C++, а в C# есть события (events). Рассказать что такое event – указатель на объект + указатель на функцию, а observer – реализация событий через интерфейс. Если тренинг для С++-ников показать реализацию делегатов на С++ (C++ delegates (Anton Rodionov).zip)

1. **Adapter**

История 1. Представьте себе ситуацию, у вас есть интерфейс IStream (поток) и реализация этого интерфейса для чтения файла, а вам нужно сделать так, чтобы очередная реализация IStream читала из массива byte[]. Но массив byte[] не наследуется от IStream, так что же делать?

История 2. Представьте себе ситуацию, у вас есть интерфейс IConfig с методом string GetValue(string key), и у вас есть реализация IConfig, которая считывает по ключу значения из системного реестра. А вам для тестирования нужно сделать так, чтобы очередная реализация IConfig забирала значения из вашего Dictionary<string, string>. Но Dictionary не наследуется от IConfig, так что же делать?

1. **Decorator и Proxy**

История 1. Представьте себе ситуацию, у вас есть интерфейс IStream (поток) и стандартная реализация этого интерфейса для работы с файлами (чтения и записи), которую вы используете для сохранения пользовательских настроек. А теперь пришло требование, что вам нужно сделать так, чтобы данные записывались в файл в сжатом виде.

Начать с размышления с того, что можно вставить логику сжатия данных в место, где используется IStream (непосредственно до вызова метода Write и сразу после вызова метода Read). Подвести слушателей к пониманию, почему это плохо. Затем уже показать, как написать этот Decorator (Wrapper).

История 2. Представьте себе ситуацию, у вас есть интерфейс IBookStorage и стандартная реализация этого интерфейса для работы с веб-сервисом, которую вы используете для создания, получения, обновления и удаления (CRUD) книг в вашей электронной библиотеке. А теперь вы решили, что хорошо было бы кэшировать полученные данные на какое-то время. Как добавить кэш?

Начать с размышления с того, что можно модифицировать существующий код для работы с кэшем. А что если он должен быть опциональным? А что если другой команде нужен ваш компонент, но без всяких намеков на кэш? Подвести публику к тому, что это не лучший вариант, что нарушается single responsibility и т.п. и подойти уже к приему Proxy (Wrapper).

Только ради того, чтобы слушатели поняли, почему decorator и proxy – разные pattern’ы, рассказать, чем они отличаются – смыслом применения. Коротко говоря, если модифицируется результат работы класса, то это decorator, а если налагаются дополнительные проверки на выполнение функций, или же что-то делается отложено, или забирается из кэша – это proxy. Ни в коем случае не требовать запоминать разницу!!

1. **Bridge и Strategy**

Очень простые приемы, которые заключаются в использовании интерфейсов, чтобы скрыть конкретную реализацию (на этом запад накрутил целую концепцию SOA).

Только ради того, чтобы слушатели поняли, почему bridge и strategy – разные pattern’ы, рассказать, чем они отличаются – способом применения. Коротко говоря, если метод класса полностью делегирует свои обязанности интерфейсу (какому-либо объекту через интерфейс) – это bridge (якобы просто мост к тому, кто будет делать работу), а если вы просто используете интерфейс в середине метода для сокрытия конкретной реализации – это strategy. Ни в коем случае не требовать запоминать разницу!!

1. **Visitor**

Посмотрим на реализацию большинства фреймворков для работы с UI. Основным классом в них выступает Control. Любая диаграмма, таблица или просто текст – это Control. У класса Control есть метод Draw, в который передается DrawContext, в котором как минимум указана позиция, в которой Control должен быть нарисован. Если посмотреть со стороны фреймворка, то вы знаете где нарисовать контрол, но не знаете как. Для этого вы как-бы «посещаете» каждый конкретный контрол и просите его нарисовать себя. Вот в этом и суть приема, который назвали Visitor.

1. **Factory**

Прием достаточно простой. Цель Factory – создать объект. Почему не создать его просто через new, вот в чем вопрос. И здесь у каждого своя причина. Вот некоторые их них:

1. Проблема расширяемости конструктора. Вы хотите расширить класс, но для этого ему нужно передать в конструктор еще один интерфейс. Проблема в том, что этот класс инстанциируется во многих местах, и теперь вам придется «пофиксить» все эти места. Решение – использовать фабрику, которая будет создавать объекты этого типа. Современный подход – использование IoC контейнеров для данных целей.
2. Поговорим о С++. Проблема в том, что вы не можете создать объект класса просто потому, что он находится где-то в другой dll, и класс не экспортируется через \_\_declspec(dllexport). Это обычно делается для расширяемого программирования (для плагинов). Обычно экспортируют лишь одну функцию, например GetFactory, которая возвращает IFactory, через которую вы можете создавать объекты тех или иных классов.

В C# эту проблему можно обойти рефлексией, инстанциируя классы из другой dll через Activator.CreateInstance (очень распространенный подход).

1. Иногда Factory инкапсулирует какую-то логику создания объекта, возвращая объект того или иного типа (типы реализуют один и тот же интерфейс). Например, возвращаясь к истории о BookStorage, Factory может обратиться к конфигурационному файлу (косвенно) и вернуть просто BookStorage или BookCachedStorage (прокси над IBookStorage, накладывающая кэш).

\_\_\_\_\_\_\_

Повторюсь, главная цель повествования, чтобы люди не пугались слов Adapter, Proxy и т.п., а понимали о чем идет речь.