Шаблон проектирования

Шаблон проектирования — это способ представления в общем виде как условия задачи, которую необходимо решить, так и правильных подходов к ее решению

- Многократно применять высококачественное решение для повторяющихся задач
- Ввести общую терминологию для расширения взаимопонимания в пределах группы разработчиков
- Поднять уровень, на котором проблема решается, и избежать нежелательного углубления в детали реализации уже на ранних этапах разработки
- Оценить, что было создано — именно то, что нужно, или же просто некоторое работоспособное решение
- Ускорить профессиональное развитие как всей группы разработчиков в целом, так и отдельных ее членов
- Повысить модифицируемость кода
- Обеспечить выбор лучших вариантов реализации проекта, даже если сами шаблоны проектирования в нем не используются
- Найти альтернативное решение для исключения громоздких иерархий наследования классов

Шаблон проектирования

- Структурные шаблоны – показывают, как объекты и классы объединяются для образования сложных структур.
- Порождающие шаблоны – управляют и контролируют процесс создания и жизненный цикл объектов.
- Шаблоны поведения – используются для организации, управления и объединения различных вариантов поведения объектов.

Шаблон проектирования Factory Method

Тип: порождающий шаблон проектирования (Creational)

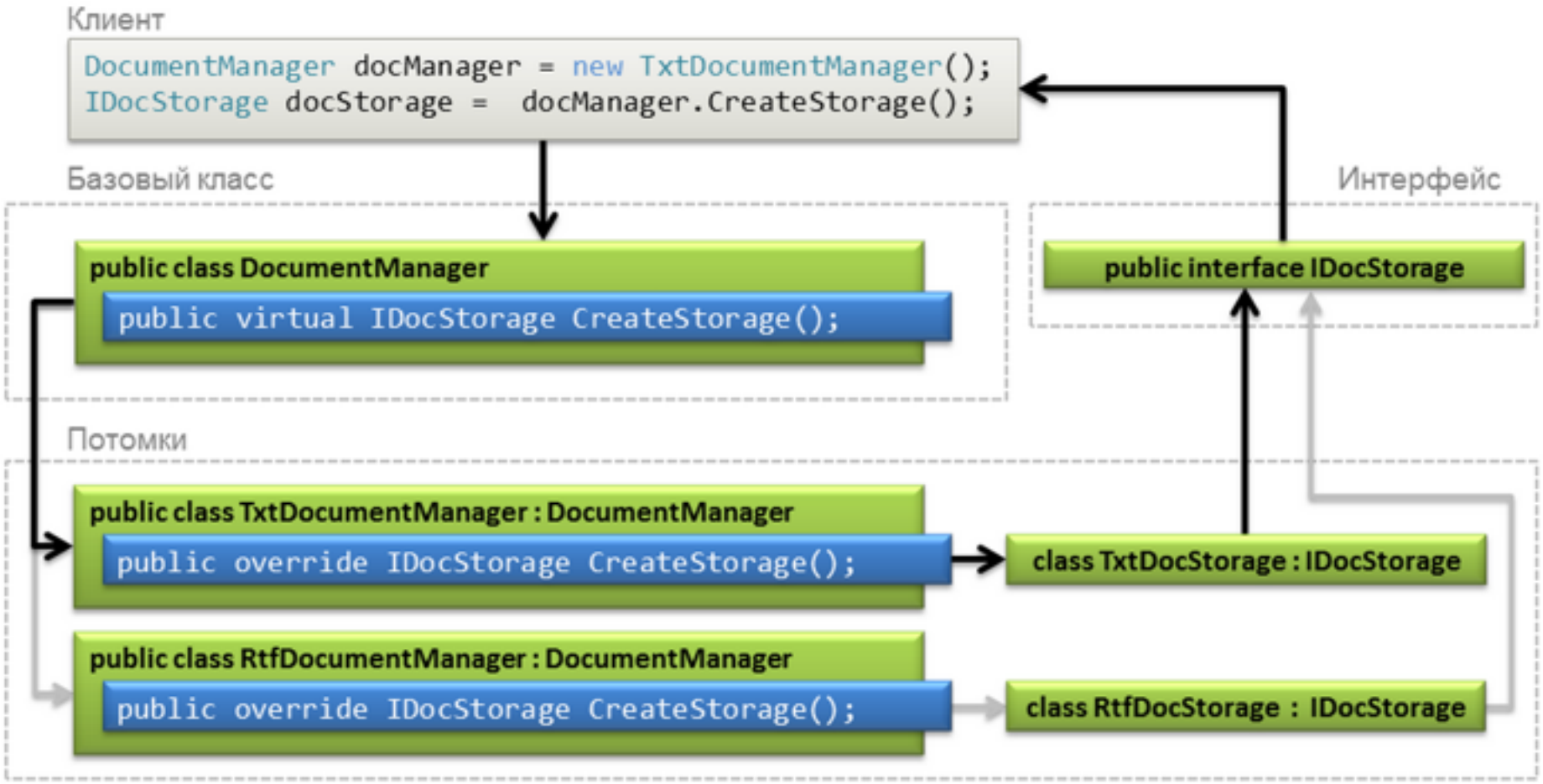
Описание:

- применяется для создания объектов с определенным интерфейсом, реализации которого предоставляются потомками

Шаблон используется в случаях если:

- класс заранее не знает, какие объекты необходимо будет создавать, т.к. возможны варианты реализации
- (или) класс спроектирован так, что спецификация порождаемого объекта определяется только в наследниках.
- (или) класс выделяет и делегирует часть своих функций вспомогательному классу, при этом необходимо скрыть его реализацию для достижения большей гибкости или возможности расширения функциональности

Шаблон проектирования Factory Method



Шаблон проектирования Factory Method

Реализация шаблона в общем виде

- определяется интерфейс порождаемых объектов **IProduct**
- базовый класс описывает метод public **IProduct FabricMethod()** для их создания
- наследники переопределяют его, порождая свои реализации **IProduct**
- базовый класс и клиентский код используют в работе только интерфейс **IProduct**, не обращаясь к конкретным реализациям самостоятельно

Example

Шаблон проектирования Abstract factory (Абстрактная фабрика)

Тип: порождающий шаблон проектирования (Creational)

Описание:

- предоставляет интерфейс, позволяющий порождать семейства объектов с заданными интерфейсами, при этом их реализации могут варьироваться

Шаблон используется в случаях если:

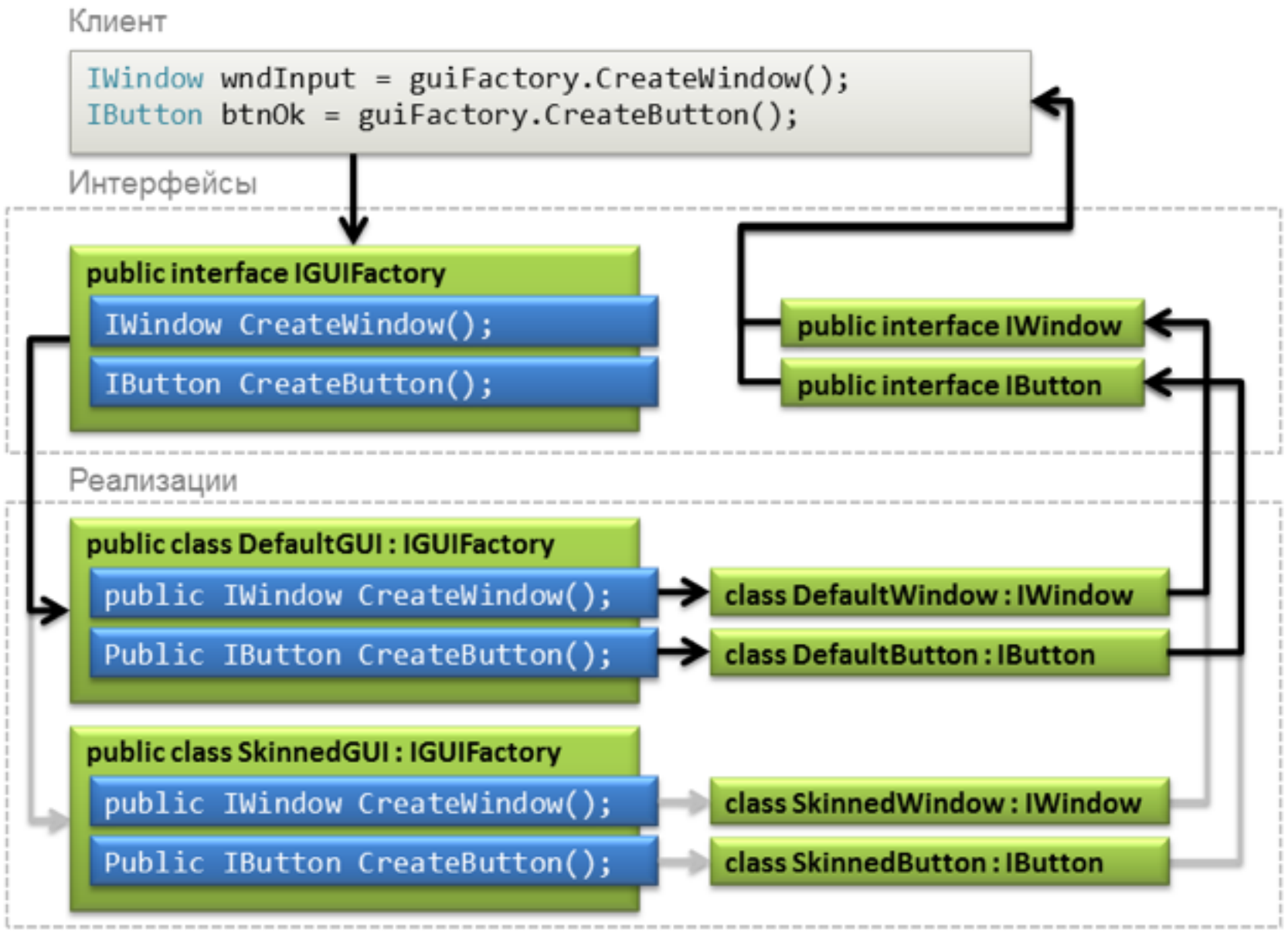
- система не должна зависеть от способа создания и реализации входящих в нее объектов;
- (и) система работает с семействами объектов;
- (и) входящие в семейство объекты должны использоваться совместно

Часто можно увидеть совместное использование Абстрактной фабрики с другими шаблонами:

- Одиночка – если не требуется больше одного ее экземпляра
- Фабричный метод – для создания ее экземпляров

Клиентский код использует в работе только интерфейсы. Реализации Абстрактной фабрики и порождаемых ею объектов скрыты. Такой подход уменьшает зависимости между объектами и повышает гибкость, за счет возможности изменения реализаций

Шаблон проектирования Abstract factory



Шаблон проектирования Abstract factory

Реализация шаблона в общем виде

- разрабатываем интерфейсы объектов семейства и Абстрактной фабрики
- создаем семейства объектов и реализации Абстрактной фабрики для них
- в программе, например, в зависимости от версии ОС, конфигурации или другого параметра, порождается необходимая реализация Абстрактной фабрики
- в дальнейшем используются только интерфейсы как Абстрактной фабрики, так и порождаемых ей объектов

Factory Method vs Abstract factory

Абстрактная фабрика	Фабричный метод
Порождает семейство объектов с определенными интерфейсами	Порождает один объект с определенным интерфейсом
Интерфейс, реализуемый классами	Метод класса, который переопределяется потомками
Скрывает реализацию семейства объектов	Скрывает реализацию объекта

Шаблон проектирования Строитель

Тип: порождающий шаблон проектирования (Creational)

Описание:

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

Шаблон используется в случаях 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

- данный шаблон не скрывает реализацию порождаемых объектов, а создает то, что требуется;
- как следствие, результатом работы могут быть объекты, не связанные явно между собой. Как правило, у них единые цели, но не обязательно есть общие интерфейсы, базовые классы и т.д.

Шаблон проектирования Строитель

Шаблон Строитель включает двух участников процесса:

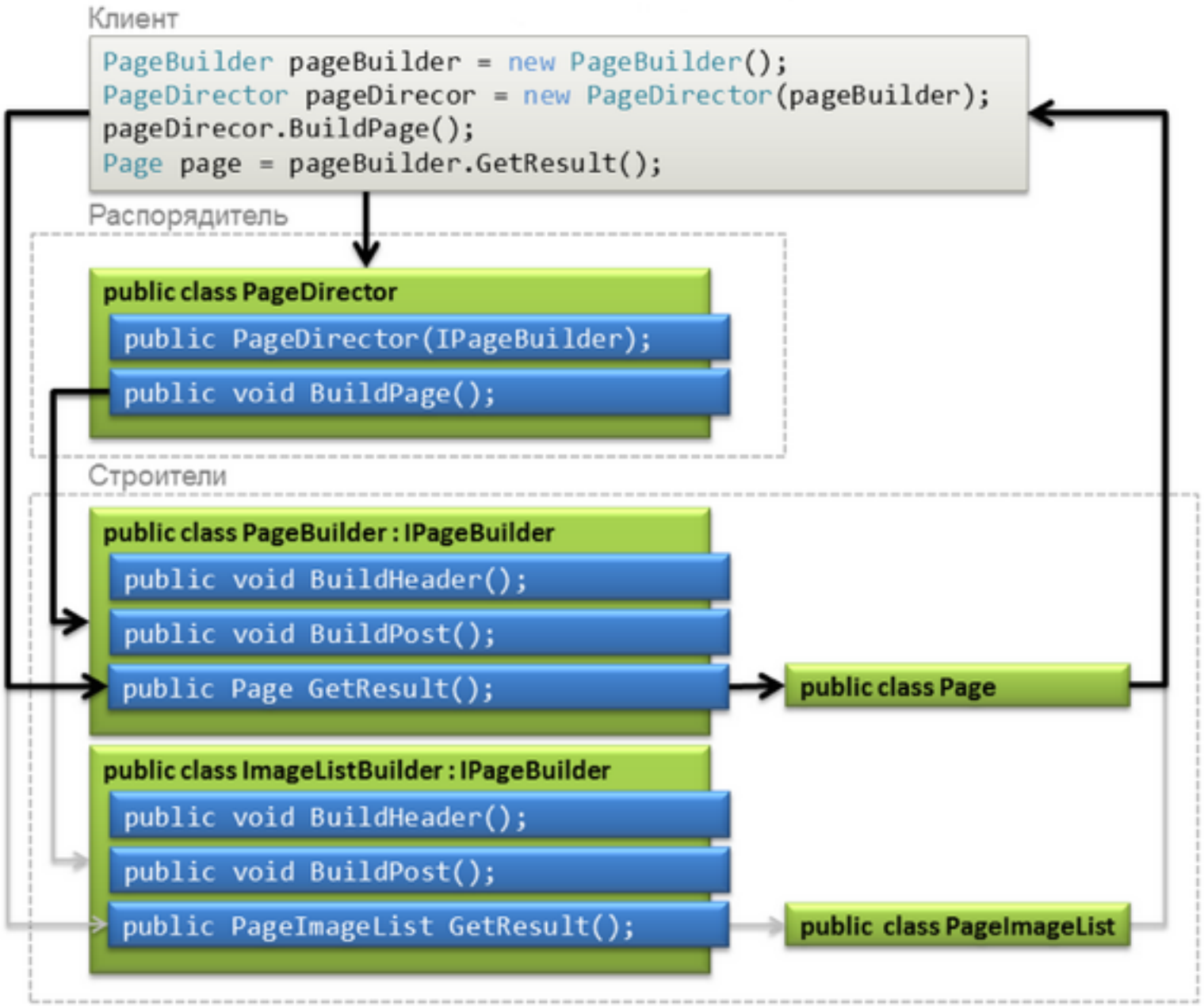
- Строитель (Builder) – предоставляет методы для сборки частей объекта, при необходимости преобразовывает исходные данные в нужный вид, создает и выдает объект;
- Распорядитель (Director) – определяет стратегию сборки: собирает данные и определяет порядок вызовов методов Строителя.

Шаблон проектирования Строитель

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 *Director* (реализующий *IDirector*), работающий со Строителями через интерфейс *IBuilder*;
- создаем класс Строителя *Builder*, реализующий интерфейс *IBuilder* и метод получения результата;
- в клиентском коде экземпляру *Director* передаем интерфейс *IBuilder* экземпляра *Builder*;
- запускаем процесс сборки, вызвав метод Распорядителя;
- получаем созданный экземпляр *Product* у используемой реализации Строителя *Builder*.

Шаблон проектирования Строитель



Factory Method vs Abstract factory vs Builder

Абстрактная фабрика	Фабричный метод	Строитель
Порождает семейство объектов с определенными интерфейсами	Порождает один объект с определенным интерфейсом	Создает в несколько шагов один сложный (составной) объект
Интерфейс, реализуемый классами	Метод класса, который переопределяется потомками	Интерфейс строителя, реализуемый классами, и класс для управления процессом
Скрывает реализацию семейства объектов	Скрывает реализацию объекта	Скрывает процесс создания объекта, порождает требуемую реализацию

.

Порождающие шаблоны

- Фабричный метод (Factory Method)
- Одиночка (Singleton)
- Абстрактная фабрика (Abstract factory)
- Строитель (Builder)
- Прототип (Prototype)
- Пул объектов (Object pool)
- Инициализация при получении ресурса (RAII)
- Отложенная инициализация
- Пул одиночек (Multiton) + Улучшенный Пул одиночек

Шаблон проектирования Wrapper / Decorator (декоратор, обертка)

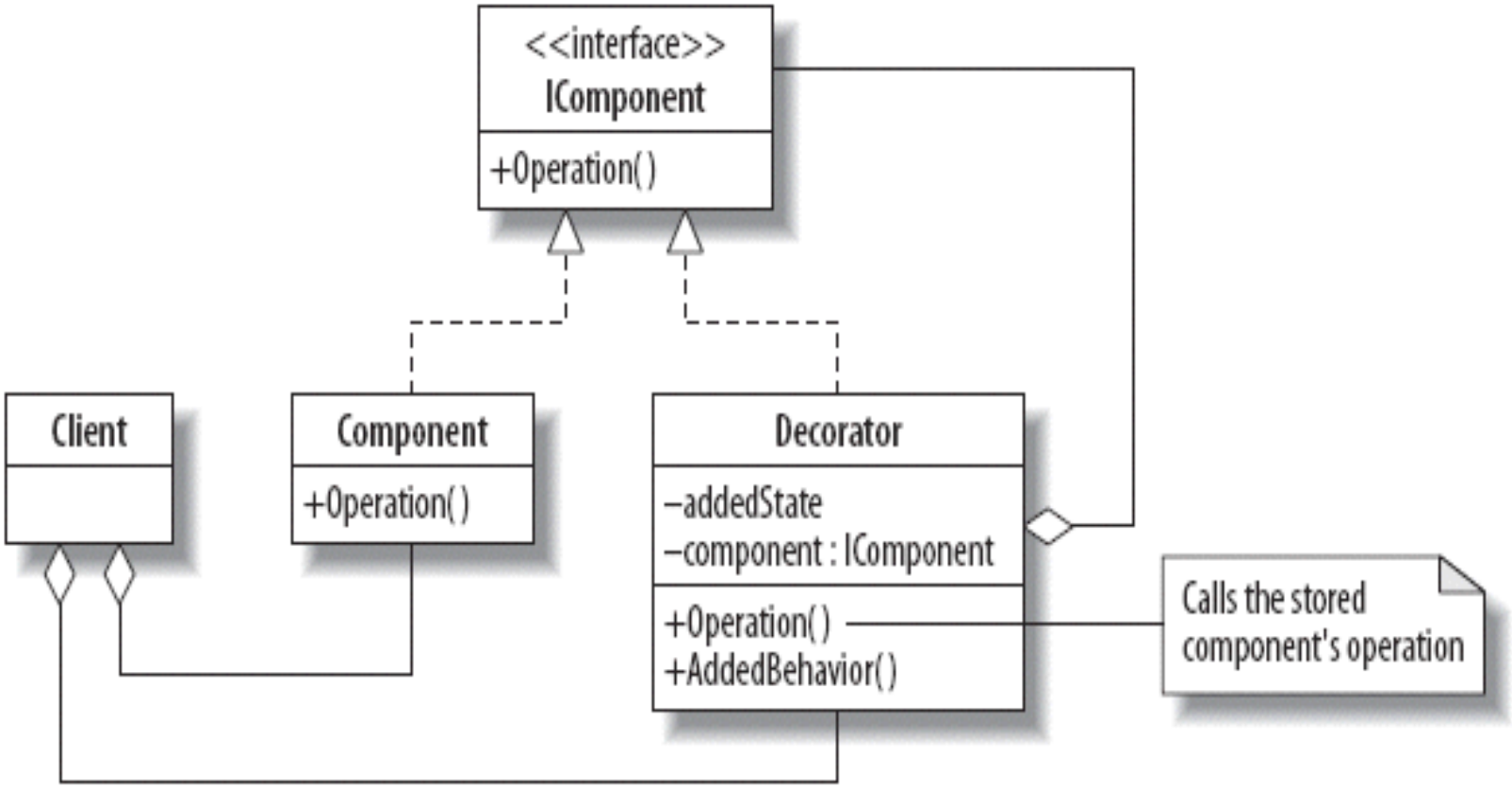
Тип: структурный шаблон проектирования ((Structural
Описание:

- предназначен для динамического добавления объекту новой функциональности, является гибкой альтернативой механизму наследования, в том числе и множественного

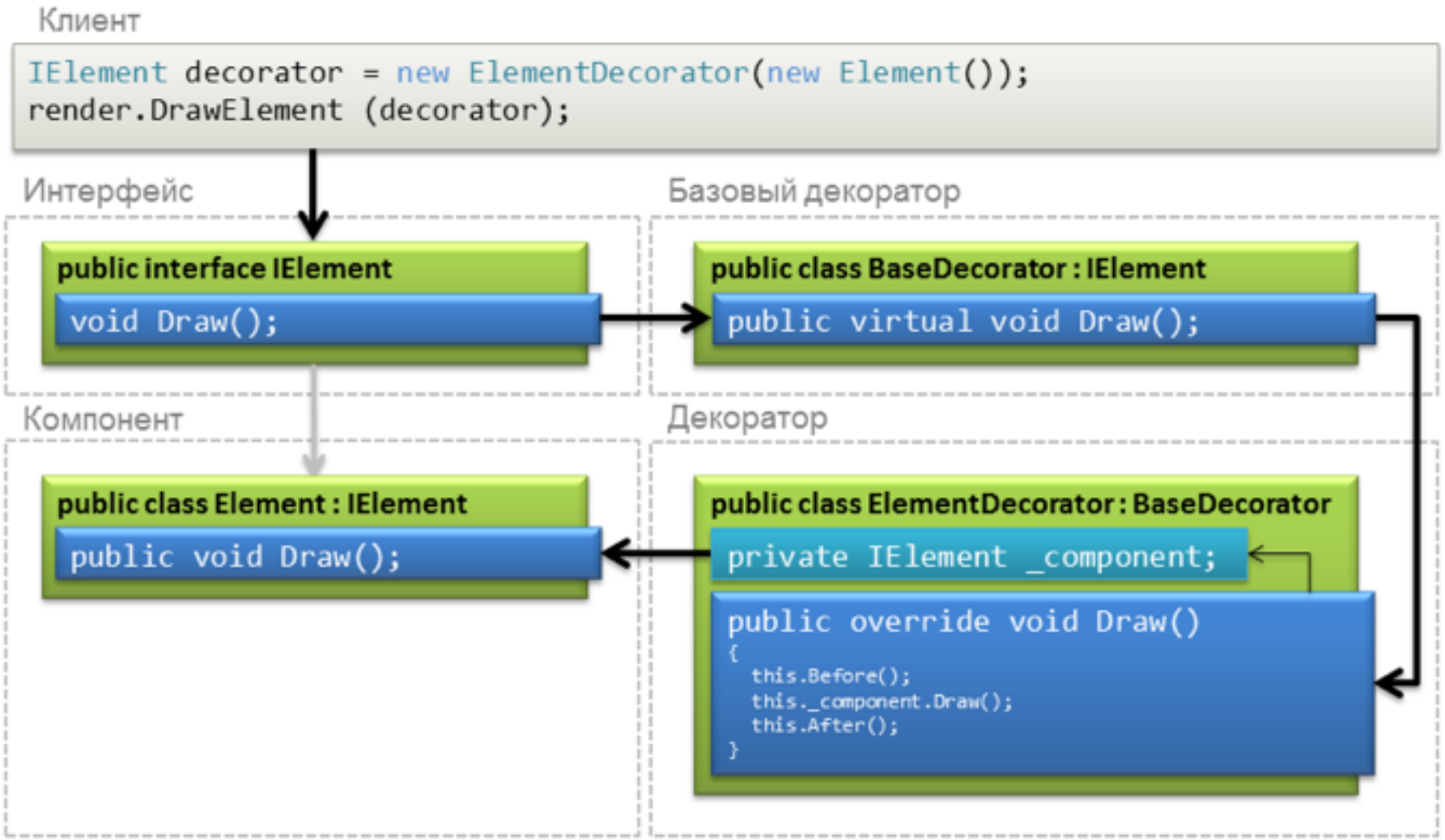
Шаблон используется в случаях если:

- динамически и прозрачно для клиента изменять функциональность объекта
- (или) реализовать небольшую функциональность, которая в дальнейшем может быть исключена
- (или) уменьшить число классов, получающихся в результате использования наследования
- (или) добавить функциональность классу, от которого невозможно наследоваться
- (или) реализовать аналог множественного наследования, в языках его не поддерживающих

Шаблон проектирования Decorator



Шаблон проектирования Decorator



Шаблон проектирования Decorator

Декоратор является оберткой над исходным компонентом. Он реализует тот же самый интерфейс, поэтому может замещать этот компонент. Однако, цель шаблона не просто в переадресации запросов. Он добавляет свой код до и/или после вызовов исходных методов, в крайнем случае замещая их полностью. **Это приводит к изменению исходного поведения и появлению новых возможностей**.

Реализация шаблона в общем виде

- определяем общий интерфейс (*IComponent*) и его реализации;
- разрабатываем базовый Декоратор (*DecoratorBase*), реализующий общий интерфейс (*IComponent*):
  - создаем механизм подключения и хранения компонента;
  - реализуем переадресацию всех методов и свойств;
- создаем конкретные Декораторы (*Decorator*), используя наследование от базового;
- в клиентском коде используем Декоратор вместо конкретного компонента;
- при необходимости создаем цепочки Декораторов, передавая один из них в другой. Это позволяет добавить несколько новых возможностей компоненту;
- если функции декоратора становятся не нужны, то используем вновь исходный объект

Шаблон проектирования Decorator

- Декоратор прозрачен для использования, т.к. клиент не заметит подмены компонента за счет реализации одного и того же интерфейса с ним.
- Добавление новой функциональности осуществляется подменой экземпляра оригинального компонента. Исключить ее так же легко – нужно использовать оригинальный объект.
- Поскольку шаблон реализует интерфейс исходного компонента, то ничего не мешает вкладывать один Декоратор в другой, создавая их цепочки. Данный подход является альтернативой множественному наследованию.
- В шаблоне нет ограничения на добавления новых свойств и методов. Такой объект может использоваться как вместо декорируемого компонента, так и самостоятельно.
- Декоратор способен работать как с самим исходным компонентом, так и его наследниками.

Шаблон проектирования Adapter (Адаптер)

Тип: структурный шаблон проектирования Structural

Описание:

- предназначен для приведения интерфейса объекта к требуемому виду

Шаблон используется в случаях если:

- существующий объект, называемый адаптируемым, предоставляет необходимые функции, но не поддерживает нужного интерфейса;
- (или) неизвестно заранее, с каким интерфейсами придется работать адаптируемому объекту;
- (или) формат входных или выходных данных метода не совпадает с требуемым.

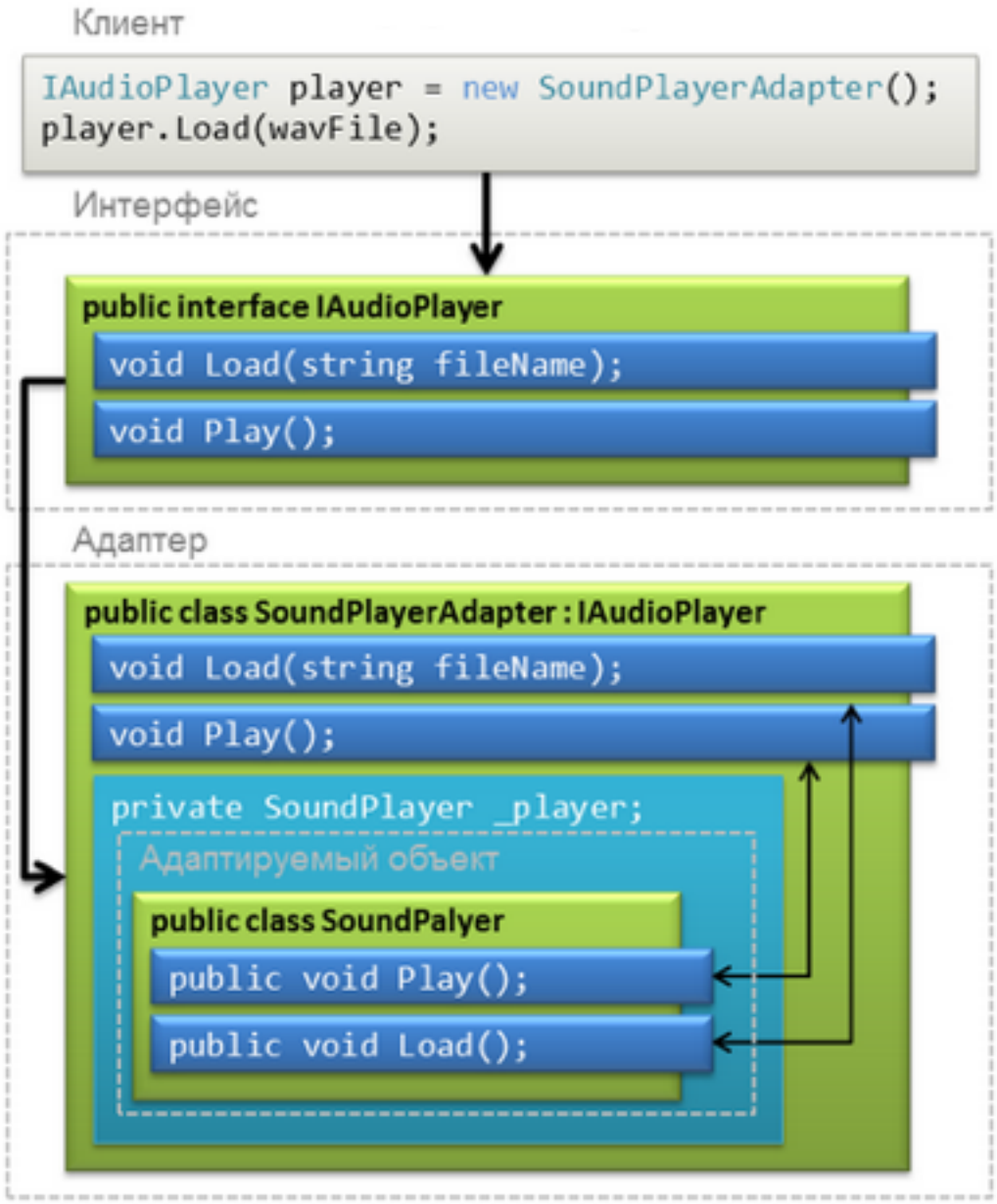
Шаблон проектирования Adapter

Главная задача Адаптера – реализация требуемого интерфейса и трансляция его вызовов адаптируемому объекту. Подобную ситуацию можно встретить при использовании сторонних библиотек. Они далеко не всегда предоставляют интерфейсы, которые необходимы в для связи с другими объектами. При этом изменить код или добавить поддержку интерфейса не предоставляется возможным.

Реализация шаблона в общем виде

- Адаптер объекта – использует композицию, т.е. содержит экземпляр адаптируемого объекта
- Адаптер класса – использует наследование от адаптируемого объекта для получения его функциональности

Шаблон проектирования Adapter



Структурные шаблоны

- Адаптер (Adapter)
- Фасад (Facade)
- Мост (Bridge)
- Декоратор (Decorator)
- Прокси (Proxy)
- Компоновщик (Composite)
- Приспособленец (Flyweight)

Схожие шаблоны и их отличия

Адаптер	**Изменяет интерфейс объекта не изменяя его функциональности. Может адаптировать несколько объектов к одному интерфейсу.**	**Позволяет повторно использовать уже существующий код.**	**Содержит или наследует адаптируемый объект.**
Фасад	**Объединяет группу объектов под одним специализированным интерфейсом.**	**Упрощает работу с группой объектов, вносит новый уровень абстракции.**	**Содержит или ссылается на объекты, необходимые для реализации специализированного интерфейса.**
Мост	**Разделяет объект на абстракцию и реализацию. Используется для иерархии объектов.**	**Позволяет отдельно изменять (наследовать) абстракцию и реализацию, повышая гибкость системы.**	**Содержит объект(реализацию), который предоставляет методы для заданной абстракций и ее уточнений (наследников).**
Декоратор	**Расширяет возможности объекта, изменяет его поведение. Поддерживает интерфейс декорируемого объекта, но может добавлять новые методы и свойства.**	**Дает возможность динамически менять функциональность объекта. Является альтернативой наследованию (в том числе множественному).**	**Содержит декорируемый объект. Возможна цепочка объектов, вызываемых последовательно.**
Прокси	**Прозрачно замещает объект и управляет доступом к нему. Не изменяет интерфейс или поведение.**	**Упрощает и оптимизирует работу с объектом. Может добавлять свою функциональность, скрывая ее**	**Содержит объект или ссылку на него, может управлять существованием замещенного объекта.**
Компоновщик	**Предоставляет единый интерфейс для взаимодействия с составными объектами и их частями.**	**Упрощает работу клиента, позволяет легко добавлять новые варианты составных объектов и их частей.**	**Включается в виде интерфейса в составные объекты и их части.**
Приспособленец	**Не ставит целью изменение интерфейса объекта. Но это может потребоваться для получения обратно данных из вынесенной части состояния.**	**Позволяет уменьшить число экземпляров объекта в приложении и тем самым сэкономить его ресурсы.**	**Выносит контекстно-зависимую часть состояния объекта вовне, заменяя несколько его экземпляров одним.**

Поведенческие шаблоны

- Цепочка ответственностей (Chain of Responsibility)
- Итератор (Iterator)

Шаблон проектирования Стратегия

Стратегия — поведенческий шаблон проектирования, который определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Стратегия позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются

Стратегия является фундаментальным паттерном, поскольку она проявляется в большинстве других классических паттернов, которые поддерживают специализацию за счет наследования. Абстрактная фабрика – это стратегия создания семейства объектов; фабричный метод – стратегия создания одного объекта; строитель – стратегия построения объекта; итератор – стратегия перебора элементов и т.д.

Мотивация использования Стратегии: выделение поведения или алгоритма с возможностью его замены во время исполнения

Шаблон проектирования Стратегия

Стратегия является невероятно распространенным паттерном в .NET Framework. Весь LINQ (Language Integrated Query) – это набор методов расширения, принимающих "стратегии" фильтрации, получения проекции и т.д. Коллекции принимают стратегии сравнения элементов, в результате любой класс, который принимает *IComparer<T>* или *IEqualityComparer<T>* используют стратегию.

Когда выделять интерфейс (.NET interface) у класса:

Класс является реализацией некоторой стратегии и будет использовать полиморфным образом.

Реализация класса работает с внешним окружением (файлами, сокетами, конфигурацией и т.п.).

Класс находится на стыке модулей.

Когда не нужно выделять интерфейс класса:

Класс является неизменяемым «объектом-значением» (Value Object) или объектом-данных (Data Object).

Класс обладает стабильным поведением (не работает с внешним окружением).