Что такое паттерны проектирования

Паттерн (шаблон) проектирования – описание взаимодействия **объектов** и **классов**, адаптированных для решения общей задачи проектирования в конкретном контексте. Другими словами, паттерны – это некоторые обобщенные решения задач, часто встречающихся при проектировании информационных систем.

Паттерны не являются готовыми решениями для конкретной задачи. Они – обобщение опыта разработчиков при решении наиболее часто встречающихся проблем проектирования.

Паттерн состоит из следующих основных элементов:

Имя – ключевое слово (словосочетание), с помощью которого можно быстро описать ситуацию («проблема – решение – последствия»). Присваивание паттернам имен позволяет проектировать на более высоком уровне абстракции. С помощью словаря паттернов можно вести обсуждение с коллегами, упоминать паттерны в документации, в тонкостях представлять дизайн системы.

Задача (проблема) – описание того, когда следует применять паттерн. При этом может описываться конкретная проблема проектирования, например способ представления алгоритмов в виде объектов. Иногда отмечается, какие структуры классов или объектов свидетельствуют о негибком дизайне. Также может включаться перечень условий, при выполнении которых имеет смысл применять данный паттерн.

Решение – описание того, как проблема может быть решена в обобщенных терминах. Описание элементов дизайна, отношений между ними, функций каждого элемента. Конкретный дизайн или реализация не имеются в виду, поскольку паттерн – это шаблон, применимый в самых разных ситуациях. Просто дается абстрактное описание задачи проектирования и того, как она может быть решена с помощью некоего весьма обобщенного сочетания элементов (классов и объектов).

Результаты (последствия) – хорошее и плохое, типично возникающее в результате применения паттерна, возможные компромиссы. Знать о последствиях принимаемого решения необходимо, чтобы можно было выбрать между различными вариантами и оценить преимущества и недостатки данного паттерна.

Паттерны GoF

Классификация паттернов

В настоящем пособии представлена основная группа паттернов объектно-ориентированного проектирования, сформулированная в работе авторов Э. Гамма, Р. Хелма, Р. Джонсона, Дж. Влиссидес и известная как «паттерны GoF» (GoF – сокр. от Gang of Four – «Союз Четырех»). Паттерны GoF классифицируются по двум параметрам:

1. Цель – назначение паттерна. По этому признаку различают следующие виды паттернов:

- порождающие – решают вопросы создания объектов;
- структурные – описывают проблемы и способы **композиции** объектов;
- паттерны поведения – решают проблемы взаимодействия объектов.

2. Уровень применимости – указывает, к чему применяется паттерн: к **классам** или **объектам**. Паттерны уровня классов описывают отношения между классами и их **подклассами**. Такие отношения выражаются с помощью **наследования**, поэтому они статичны, то есть зафиксированы на этапе компиляции. Паттерны уровня объектов описывают отношения между объектами, которые могут изменяться во время выполнения, и потому более динамичны. Почти все паттерны в какой-то мере используют наследование, поэтому к категории «Ппаттерны классов» отнесены только те, что сфокусированы лишь на отношениях между классами.

Классификацию паттернов GoF можно представить в следующем виде:

Цель / Уровень	Порождающие	Структурные	Поведения
Класс	**Фабричный Метод**	**Адаптер**	**Интерпретатор** **Шаблонный Метод**
Объект	**Абстрактная Фабрика** **Одиночка** **Прототип** **Строитель**	**Адаптер** **Декоратор** **Заместитель** **Компоновщик** **Мост** **Приспособленец** **Фасад**	**Итератор** **Команда** **Наблюдатель** **Посетитель** **Посредник** **Состояние** **Стратегия** **Хранитель** **Цепочка Обязанностей**

Выбор и использование паттерна

Как выбрать паттерн: выделите проблему из контекста решаемой задачи, локализуйте ее, определите тип паттерна. Рассмотрите все паттерны, которые могут решать эту задачу, их достоинства и недостатки. Рассмотрите влияние последствий на связанные с данной частью системы элементы архитектуры.

Как использовать паттерн: разберитесь в механизме паттерна (обязанности классов и объектов, действия участников). Определите конкретные элементы (классы, **интерфейсы**, операции) Вашей системы, соответствующие участникам паттерна, задайте их имена (используйте суффиксы/префиксы участника/операции описания паттерна). Выявите влияние применения паттерна на другие элементы системы, модифицируйте их в случае необходимости. Реализуйте операции, которые выполняют обязанности и отвечают за отношения, определенные в паттерне.

Когда не надо применять паттерны: когда не потребуется повторного использования и изменений системы.

Для выбора подходящего паттерна можно воспользоваться следующей таблицей. Здесь изложены конкретные проблемы, которые приводят к ухудшению гибкости создаваемых программных систем и, в частности, уменьшают степень повторной используемости и затрудняют будущие изменения.

Проблема	Паттерны
При создании объекта явно указывается класс	**Абстрактная Фабрика** **Прототип** **Фабричный Метод**
Зависимость от конкретных операций	**Команда** **Цепочка Обязанностей**
Зависимость от аппаратной и программной платформ	**Абстрактная Фабрика** **Мост**
Зависимость от представления или **реализации**	**Абстрактная Фабрика** **Заместитель** **Мост** **Строитель** **Хранитель**
Зависимость от алгоритмов	**Итератор** **Мост** **Посетитель** **Стратегия** **Шаблонный Метод**
Сильная связанность	**Абстрактная Фабрика** **Команда** **Мост** **Наблюдатель** **Посредник** **Фасад** **Цепочка Обязанностей**
Злоупотребление наследованием классов	**Адаптер** **Декоратор** **Компоновщик** **Мост** **Наблюдатель** **Посетитель** **Приспособленец** **Состояние** **Стратегия** **Цепочка Обязанностей**

Паттерны, не попавшие в таблицу: **Одиночка** – поддерживает реализацию других паттернов (например, **Прототип**); **Интерпретатор** – решает комплекс задач для узкой области применения.

В настоящем пособии каждая группа паттернов рассматривается в целом, делается сравнительный анализ применения соответствующих паттернов. Из каждой группы описание

нескольких наиболее важных паттернов представлено в полном объеме. Для остальных представлено краткое описание, достаточное для понимания принципов их работы.

Порождающие паттерны

Порождающие паттерны проектирования абстрагируют процесс инстанцирования. Они помогают делать систему независимой от способа создания, **композиции** и представления объектов. Паттерн, порождающий классы, использует **наследование**, чтобы варьировать инстанцируемый класс, а паттерн, порождающий объекты, делегирует инстанцирование другому объекту. Эти паттерны оказываются важны, когда система больше зависит от композиции объектов, чем от наследования классов. Получается так, что основной упор делается не на жестком кодировании фиксированного набора поведений, а на определении небольшого набора фундаментальных поведений, с помощью композиции которых можно получать любое число более сложных. Таким образом, для создания объектов с конкретным поведением требуется нечто большее, чем простое инстанцирование класса.

Для порождающих паттернов актуальны две темы. Во-первых, эти паттерны инкапсулируют знания о конкретных классах, которые применяются в системе. Во-вторых, скрывают детали того, как эти классы создаются и стыкуются. Единственная информация об объектах, известная системе, – это их интерфейсы, определенные с помощью **абстрактных классов**. Следовательно, порождающие паттерны обеспечивают большую гибкость при решении вопроса о том, что создается, кто это создает, как и когда. Можно собрать систему из готовых объектов с самой различной структурой и функциональностью статически (на этапе компиляции) или динамически (во время выполнения). Порождающие паттерны показывают, как сделать дизайн более гибким, хотя и необязательно меньшим по размеру.

Иногда допустимо выбирать между тем или иным порождающим паттерном. Например, есть случаи, когда с пользой для дела можно использовать как **Прототип** *(ссылка на кадр 39)*, так и **Абстрактную Фабрику** *(ссылка на кадр 37)*. В других ситуациях порождающие паттерны дополняют друг друга. Так, применяя **Строитель** *(ссылка на кадр 38)*, можно использовать другие паттерны для решения вопроса о том, какие компоненты нужно строить, а Прототип часто реализуется вместе с **Одиночка** *(ссылка на кадр 40)*.

Фабричный метод (Factory Method)

Другое название: Virtual Constructor (Виртуальный Конструктор).

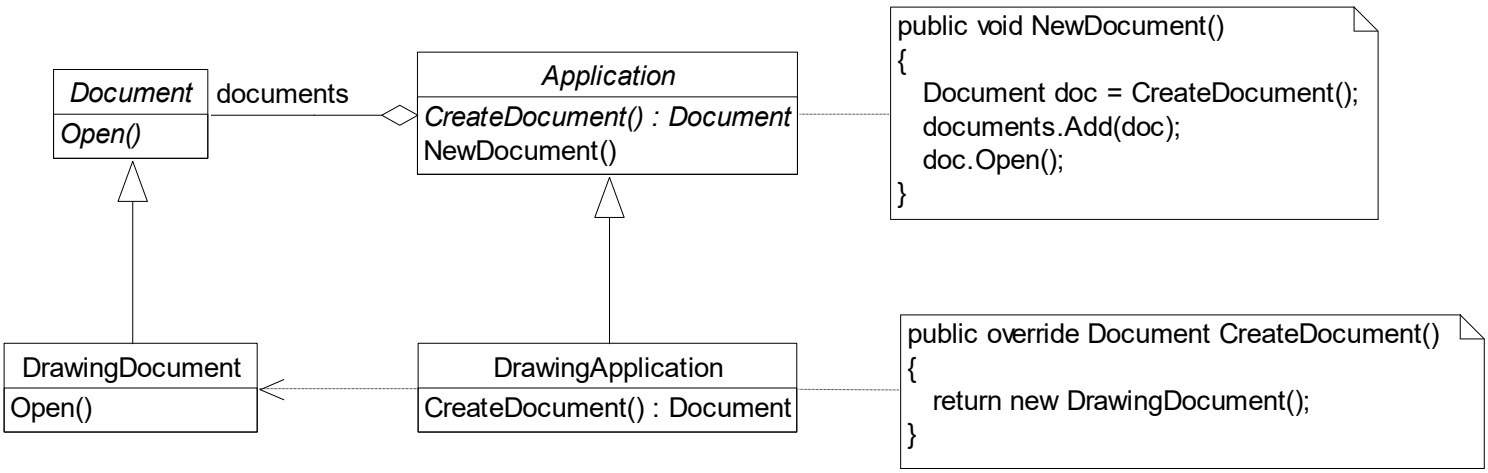
Назначение: определяет интерфейс для создания объекта, но оставляет **подклассам** возможность принять решение о том, какой класс инстанцировать.

Задача. Каркасы пользуются **абстрактными классами** для определения и поддержания отношений между объектами. Кроме того, каркас часто отвечает за создание самих объектов.

Рассмотрим каркас для приложений, способных представлять пользователю сразу несколько документов. Две основных абстракции в таком каркасе – это классы `Application` и `Document`. Оба класса абстрактные, поэтому клиенты должны порождать от них подклассы для создания специфичных для приложения **реализаций**. Например, чтобы создать приложение для рисования, мы определим подклассы `DrawingApplication` и `DrawingDocument`. Класс `Application` отвечает за управление документами и создает их по мере необходимости, например когда пользователь выбирает из меню пункт `Open` (открыть) или `New` (создать).

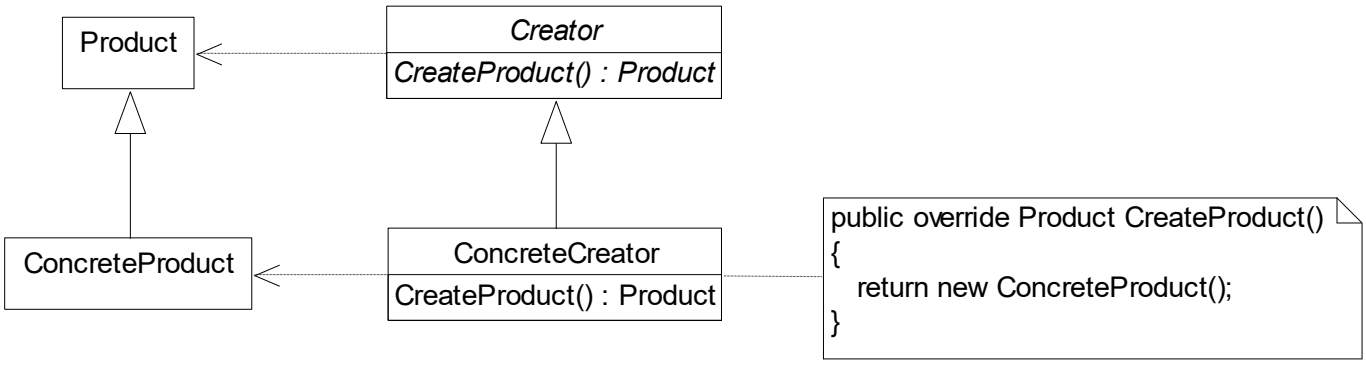
Поскольку решение о том, какой подкласс класса `Document` инстанцировать, зависит от приложения, то `Application` не может знать заранее, что именно понадобится. Этому классу известно лишь, когда нужно инстанцировать новый документ, а не какой документ создать. Возникает дилемма: каркас должен инстанцировать классы, но знает он лишь об **абстрактных классах**, которые инстанцировать нельзя.

Решение. Паттерн Фабричный Метод предлагает решение проблемы путем инкапсуляции информации о том, какой подкласс класса `Document` создать, внутри специальной операции `CreateDocument`, которая и называется фабричным методом.



Подклассы класса `Application` переопределяют абстрактную операцию `CreateDocument` таким образом, чтобы она возвращала подходящий подкласс класса `Document`. Как только подкласс `Application` инстанцирован, он может инстанцировать специфические для приложения документы, ничего не зная об их классах.

Общая структура решения.



`Product` (`Document`) – продукт: определяет **интерфейс** объектов, создаваемых фабричным методом.

`ConcreteProduct` (`DrawingDocument`) – конкретный продукт: реализует интерфейс `Product`.

`Creator` (`Application`) – создатель: объявляет фабричный метод `CreateProduct`, возвращающий объект типа `Product`, может вызывать этот метод. Также может определять **реализацию** по умолчанию фабричного метода, например в случае, если класс `Product` не абстрактный.

`ConcreteCreator` (`DrawingApplication`) – конкретный создатель: замещает фабричный метод таким образом, что возвращается экземпляр класса `ConcreteProduct`.

Создатель полагается на свои подклассы в определении фабричного метода, который будет возвращать экземпляр подходящего конкретного продукта.

Применимость. Используйте паттерн Фабричный Метод, когда:

- классу заранее неизвестно, объекты каких классов ему нужно создавать;
- класс спроектирован так, чтобы объекты, которые он создает, специфицировались подклассами;
- класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и вы планируете локализовать знание о том, какой класс выполняет эти обязанности.

Результаты. Фабричные методы избавляют проектировщика от необходимости встраивать в код зависящие от приложения классы. Код имеет дело только с **интерфейсом** класса `Product`, поэтому он может работать с любыми определенными пользователями классами конкретных продуктов.

Потенциальный недостаток фабричного метода состоит в том, что клиентам, возможно, придется создавать подкласс класса `Creator` для создания лишь одного объекта `ConcreteProduct`. Порождение подклассов оправданно, если клиенту так или иначе приходится создавать подклассы `Creator`, в противном случае клиенту придется иметь дело с дополнительным уровнем подклассов.

Создание объектов внутри класса с помощью фабричного метода всегда оказывается более гибким решением, чем непосредственное создание. Фабричный метод создает в подклассах операции-зацепки (hooks) для предоставления расширенной версии объекта.

В примере с документом класс `Document` мог бы определить фабричный метод `CreateFileDialog`, который создает диалоговое окно для выбора файла существующего документа. Подкласс этого класса мог бы определить специализированное для приложения диалоговое окно, заместив этот фабричный метод. В данном случае фабричный метод не является абстрактным, а содержит разумную реализацию по умолчанию.

В примере, который мы рассмотрели, фабричный метод вызывался только создателем. Но это совершенно необязательно: клиенты тоже могут вызывать фабричные методы, особенно при наличии параллельных иерархий классов. Параллельные иерархии возникают в случае, когда класс делегирует часть своих обязанностей другому классу, не являющемуся производным от него.

Особенности реализации. Существуют две основных разновидности паттерна. Во-первых, это случай, когда класс `Creator` является абстрактным и не содержит реализации объявленного в нем фабричного метода. Вторая возможность: `Creator` – конкретный класс, в котором по умолчанию есть реализация фабричного метода.

В первом случае для определения реализации необходимы подклассы, поскольку никакого разумного умолчания не существует. Во втором случае **конкретный класс** `Creator` использует фабричный метод, главным образом, ради повышения гибкости. Выполняется правило: «Создавай объекты в отдельной операции, чтобы подклассы могли подменить способ их создания». Соблюдение этого правила гарантирует, что авторы подклассов смогут при необходимости изменить класс объектов, инстанцируемых их родителем.

Параметризованные фабричные методы – это еще один вариант паттерна, который позволяет фабричному методу создавать разные виды продуктов. Фабричному методу передается параметр, который идентифицирует вид создаваемого объекта. Все объекты, получающиеся с помощью фабричного метода, разделяют общий **интерфейс** `Product`. В примере с документами класс `Application` может поддерживать разные виды документов. Вы передаете методу `CreateDocument` лишний параметр, который и определяет, документ какого вида нужно создать.

Соглашения об именовании: рекомендуется применять такие соглашения об именах, которые дают ясно понять, что вы пользуетесь фабричными методами – это слова «Create», «Make» и т.п. внутри имен соответствующих методов.

Родственные паттерны.

Абстрактня Фабрика реализуется с помощью фабричных методов.

Фабричные методы очень часто вызываются внутри **Шаблонных Методов**. В примере с документами: `NewDocument` – это типичный шаблонный метод.

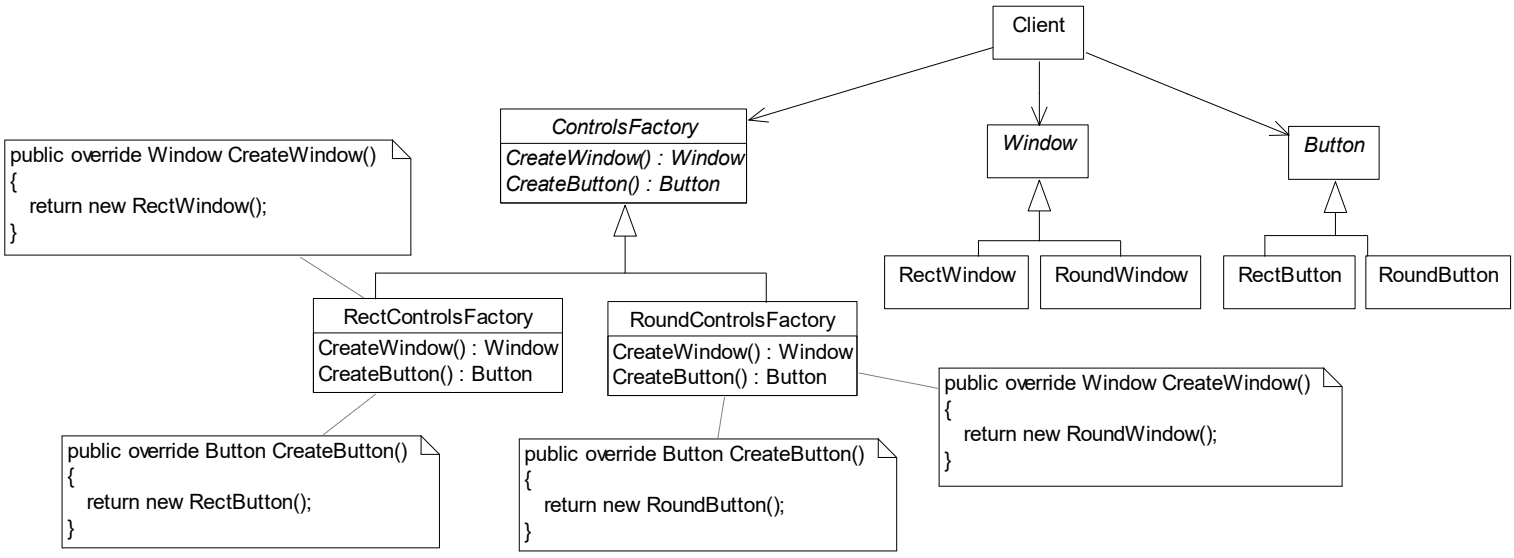
Абстрактная фабрика (Abstract Factory)

Другое название: Kit (инструментарий).

Назначение: предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.

Задача. Рассмотрим инструментальную программу для создания пользовательского интерфейса, поддерживающего разные стандарты внешнего облика, например обычные прямоугольные элементы управления (Rect) и округлые (Round). Чтобы приложение можно было перенести на другой стандарт, в нем не должен быть жестко закодирован внешний облик элементов управления. Если инстанцирование классов для конкретного внешнего облика разбросано по всему приложению, то изменить облик впоследствии будет нелегко.

Решение. Паттерн Абстрактная Фабрика предлагает решить эту проблему, определив **абстрактный класс** `ControlsFactory`, в котором объявлен **интерфейс** для создания всех основных видов элементов управления (англ. – controls): окна (window), кнопки (button) и т.д. Есть также абстрактные классы для каждого отдельного вида и конкретные подклассы, реализующие элементы управления с определенным внешним обликом. В интерфейсе `ControlsFactory` имеется операция, возвращающая новый объект для каждого вида элемента. Клиенты вызывают эти операции для получения экземпляров элементов управления, но при этом ничего не знают о том, какие именно классы используют. Стало быть, клиенты остаются независимыми от выбранного стандарта внешнего облика.

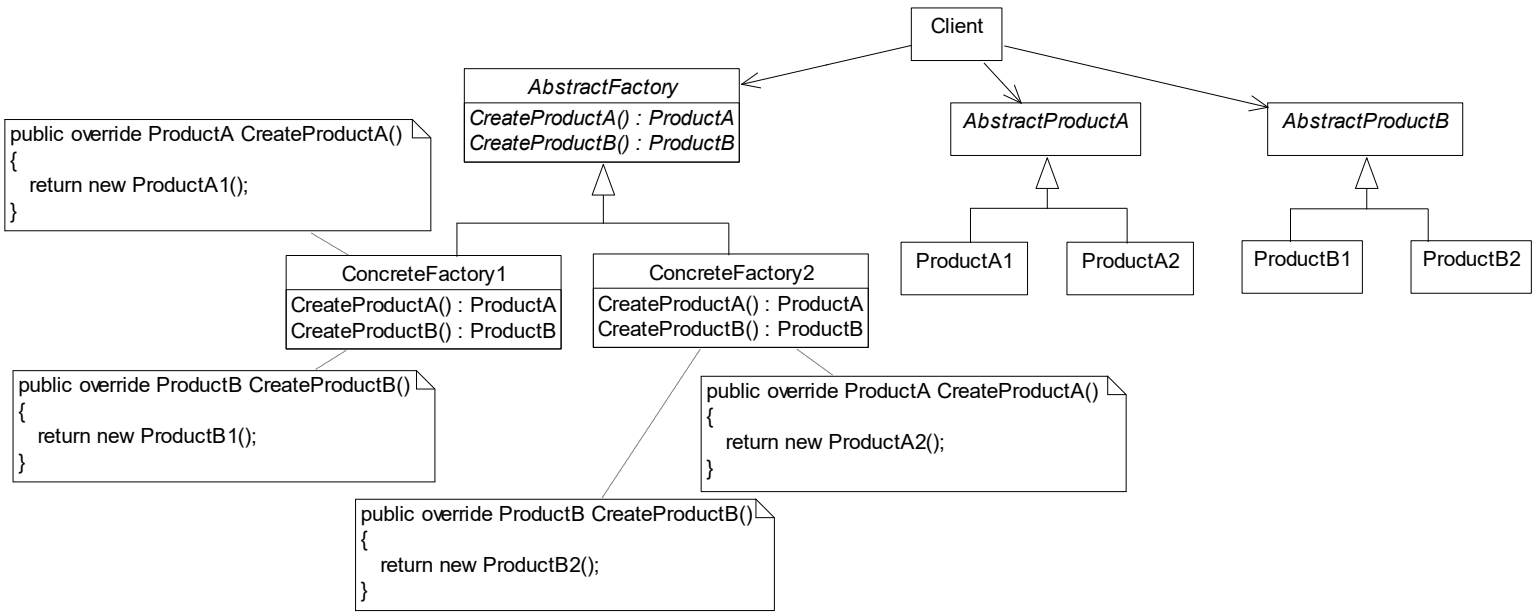


(Для простоты понимания опущены зависимости, отраженные в комментариях)

Для каждого стандарта внешнего облика существует определенный подкласс `ControlsFactory`. Каждый такой подкласс реализует операции, необходимые для создания соответствующего стандарту элемента управления. Например, операция `CreateButton` в классе `RectControlsFactory` инстанцирует и возвращает стандартную прямоугольную кнопку, тогда как соответствующая операция в классе `RoundControlsFactory` возвращает округлую кнопку. Клиенты создают элементы управления, пользуясь исключительно **интерфейсами** `ControlsFactory`, `Window` и `Button`, и им ничего не известно о классах, реализующих элементы управления для конкретного стандарта. Другими словами, клиенты должны лишь придерживаться интерфейса, определенного абстрактным, а не конкретным классом.

Класс `ControlsFactory` также устанавливает зависимости между конкретными классами элементов управления. Округлое окно должно использовать округлые кнопки, текстовые поля и т.д., и это ограничение поддерживается автоматически, как следствие использования класса `RoundControlsFactory`.

Общая структура решения.



(Для простоты понимания опущены зависимости, отраженные в комментариях)

`AbstractFactory` (`ControlsFactory`) – абстрактная фабрика: объявляет **интерфейс** для операций, создающих абстрактные объекты-продукты.

`CocreteFactoryY` (`RectControlsFactory`, `RoundControlsFactory`) – конкретная фабрика: реализует операции, создающие конкретные объекты-продукты типа Y.

`AbstractProductX` (`Window`, `Button`) – абстрактные продукты: объявляют интерфейс для типов объектов-продуктов.

`ConcreteProductXY` (`RectWindow`, `RoundButton` и др.) – конкретные продукты: определяют объекты-продукты, создаваемые конкретной фабрикой Y, реализуя интерфейс `AbstractProductX`.

`Client` – клиент: пользуется исключительно интерфейсами, которые объявлены в классах `AbstractFactory` и `AbstractProductX`.

Обычно во время выполнения создается единственный экземпляр класса `ConcreteFactoryY`. Эта конкретная фабрика создает объекты-продукты, имеющие вполне определенную реализацию. Для создания других видов объектов клиент должен воспользоваться другой конкретной фабрикой.

`AbstractFactory` передоверяет создание объектов-продуктов своим подклассам `ConcreteFactoryY`.

Применимость. Используйте паттерн Абстрактная Фабрика, когда:

- система не должна зависеть от того, как создаются, компонуются и представляются входящие в нее объекты;
- входящие в семейство взаимосвязанные объекты должны использоваться вместе, и вам необходимо обеспечить выполнение этого ограничения;
- система должна конфигурироваться одним из семейств составляющих ее объектов;
- вы хотите предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию.

Результаты. Паттерн Абстрактная Фабрика обладает следующими достоинствами и недостатками:

- изолирует конкретные классы. Помогает контролировать классы объектов, создаваемых приложением. Поскольку фабрика инкапсулирует ответственность за создание классов и сам процесс их создания, то она изолирует клиента от деталей реализации классов. Клиенты манипулируют экземплярами через их абстрактные **интерфейсы**. Имена изготавливаемых классов известны только конкретной фабрике, в коде клиента они не упоминаются;
- упрощает замену семейств продуктов. Класс конкретной фабрики появляется в приложении только один раз – при иистанцировании. Это облегчает замену используемой приложением конкретной фабрики. Приложение может изменить конфигурацию продуктов, просто подставив новую конкретную фабрику. Поскольку абстрактная фабрика создает все семейство продуктов, то и заменяется сразу все семейство. В нашем примере пользовательского интерфейса перейти от прямоугольных элементов управления к округлым можно, просто переключившись на продукты соответствующей фабрики и заново создав интерфейс;
- гарантирует сочетаемость продуктов. Если продукты некоторого семейства спроектированы для совместного использования, то важно, чтобы приложение в каждый момент времени работало только с продуктами единственного семейства. Класс `AbstractFactory` позволяет легко соблюсти это ограничение;
- трудно поддерживать новые виды продуктов. Расширение абстрактной фабрики для изготовления новых видов продуктов – непростая задача. Интерфейс `AbstractFactory` фиксирует набор продуктов, которые можно создать. Для

поддержки новых продуктов необходимо расширить **интерфейс** фабрики, то есть изменить класс `AbstractFactory` и все его подклассы.

Особенности реализации. Вот некоторые полезные приемы реализации паттерна Абстрактная Фабрика.

- фабрики как объекты существующие в единственном экземпляре: как правило, приложению нужен только один экземпляр класса `ConcreteFactory` на каждое семейство продуктов. Поэтому для реализации лучше всего применить паттерн **Одиночка**;
- создание продуктов: класс `AbstractFactory` объявляет только интерфейс для создания продуктов. Фактическое их создание – дело подклассов `ConcreteFactory`. Чаще всего для этой цели определяется фабричный метод для каждого продукта (паттерн **Фабричный Метод**). Конкретная фабрика специфицирует свои продукты путем замещения фабричного метода для каждого из них. Хотя такая реализация проста, она требует создавать новый подкласс конкретной фабрики для каждого семейства продуктов, даже если они почти ничем не отличаются;
- если семейств продуктов может быть много, то конкретную фабрику удастся реализовать с помощью паттерна **Прототип**. В этом случае она инициализируется экземпляром-прототипом каждого продукта в семействе и создает новый продукт путем клонирования этого прототипа. Подход на основе прототипов устраняет необходимость создавать новый класс конкретной фабрики для каждого нового семейства продуктов;
- определение расширяемых фабрик: класс `AbstractFactory` обычно определяет разные операции для каждого вида изготавливаемых продуктов. Виды продуктов кодируются в **сигнатуре** операции. Для добавления нового вида продуктов нужно изменить интерфейс класса `AbstractFactory` и всех зависящих от него классов. Более гибкий, но не такой безопасный способ – добавить параметр к операциям, создающим объекты. Данный параметр определяет вид создаваемого объекта. Это может быть любой идентификатор, однозначно описывающий вид продукта. При таком подходе классу `AbstractFactory` нужна только одна операция `CreateProduct` с параметром, указывающим тип создаваемого объекта. Такой вариант удобно использовать в динамически типизированных языках типа Smalltalk, нежели в статически типизированных, каким является, например, C++. Воспользоваться данным подходом в C++ можно только, если у всех объектов имеется общий абстрактный базовый класс или если объекты-продукты могут быть безопасно приведены к корректному типу клиентом, который их запросил. Но даже если приведение типов не нужно, остается принципиальная проблема: все продукты возвращаются клиенту одним и тем же абстрактным интерфейсом с уже определенным типом возвращаемого значения. Клиент не может ни различить классы продуктов, ни сделать какие-нибудь предположения о них. Если клиенту нужно выполнить операцию, зависящую от подкласса, то она будет недоступна через абстрактный интерфейс. Хотя клиент может выполнить динамическое приведение типа, это однозначно небезопасно. Здесь мы имеем классический пример компромисса между высокой степенью гибкости и расширяемостью интерфейса.

Родственные паттерны.

Классы `AbstractFactory` почти всегда используют **Фабричные Методы**, но могут быть реализованы и с помощью паттерна **Прототип**.

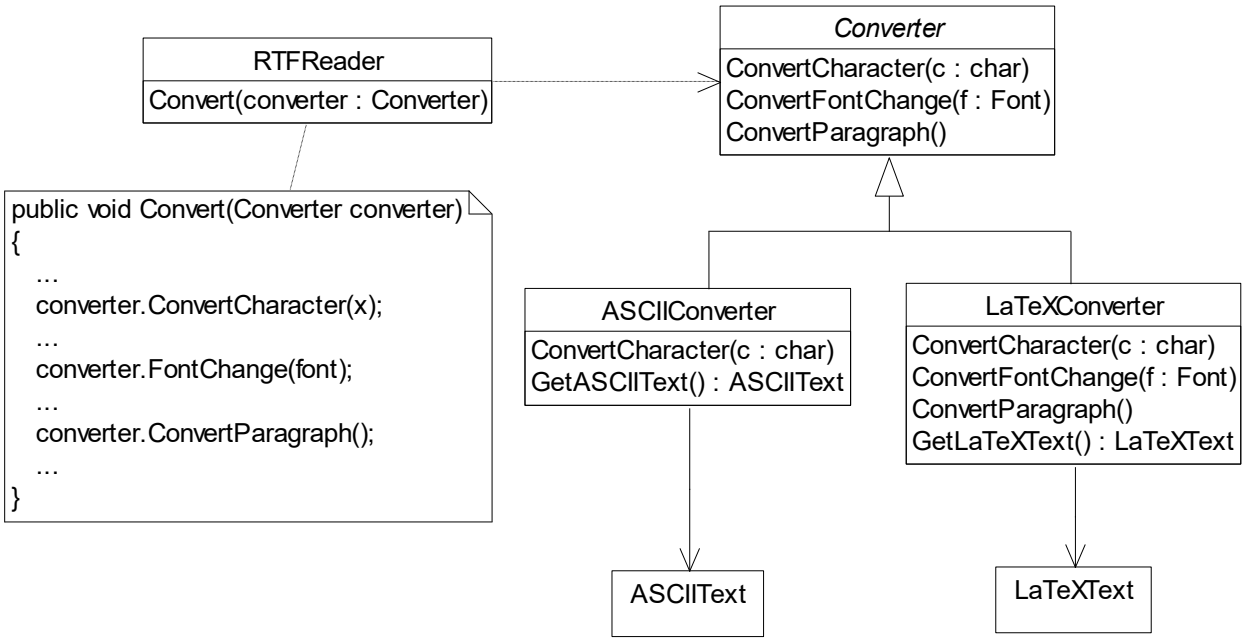
Конкретная фабрика часто описывается паттерном **Одиночка**.

Строитель (Builder)

Назначение: отделяет конструирование сложного объекта от его представления, так что в результате одного и того же процесса конструирования могут получаться разные представления.

Задача. Программа, в которую заложена возможность распознавания и чтения документа в формате RTF (Rich Text Format), должна также уметь преобразовывать его во многие другие форматы, например в простой ASCII-текст пли в формат LaTeX. Однако число вероятных преобразований заранее неизвестно. Поэтому должна быть обеспечена возможность без труда добавлять новый конвертор.

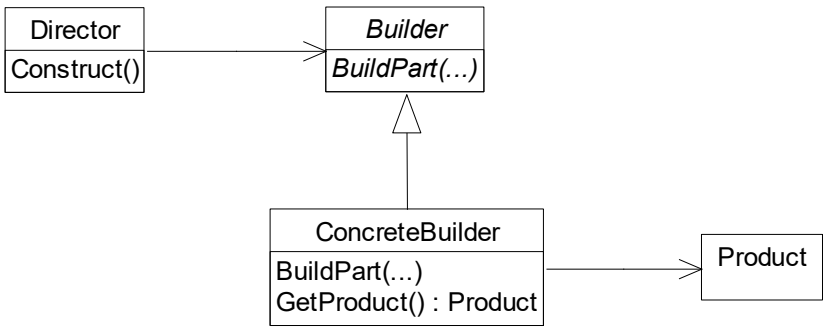
Решение. Паттерн Строитель предлагает конфигурировать класс `RTFReader`, отвечающий непосредственно за работу с документами в формате RTF, специальным объектом `Converter`, который будет преобразовывать RTF в другой текстовый формат. При разборе документа класс `RTFReader` вызывает `Converter` для выполнения преобразования. Всякий раз, когда `RTFReader` распознает лексему RTF (простой текст или управляющее слово), для ее преобразования объекту `Converter` посылается запрос. Объекты `Converter` отвечают как за преобразование данных, так и за представление лексемы в конкретном формате.



Подклассы `Converter` специализируются на различных преобразованиях и форматах. Например, `ASCIIConverter` игнорирует запросы на преобразование чего бы то ни было, кроме простого текста. С другой стороны, `LaTeXConverter` будет реализовывать все запросы для получения представления в формате LaTeX, собирая по ходу необходимую информацию о стилях.

Класс каждого конвертора принимает механизм создания и сборки сложного объекта и скрывает его за абстрактным **интерфейсом**. Конвертор отделен от загрузчика, который отвечает за синтаксический разбор RTF-документа. В паттерне Строитель абстрагированы все эти отношения. В нем любой класс конвертора называется строителем, а загрузчик – распорядителем. В применении к рассмотренному примеру Строитель отделяет алгоритм интерпретации формата текста (то есть анализатор RTF-документов) от того, как создается и представляется документ в преобразованном формате. Это позволяет повторно использовать алгоритм разбора, реализованный в `RTFReader`, для создания разных текстовых представлений RTF-документов; достаточно передать в `RTFReader` различные подклассы класса `Converter`.

Общая структура решения.



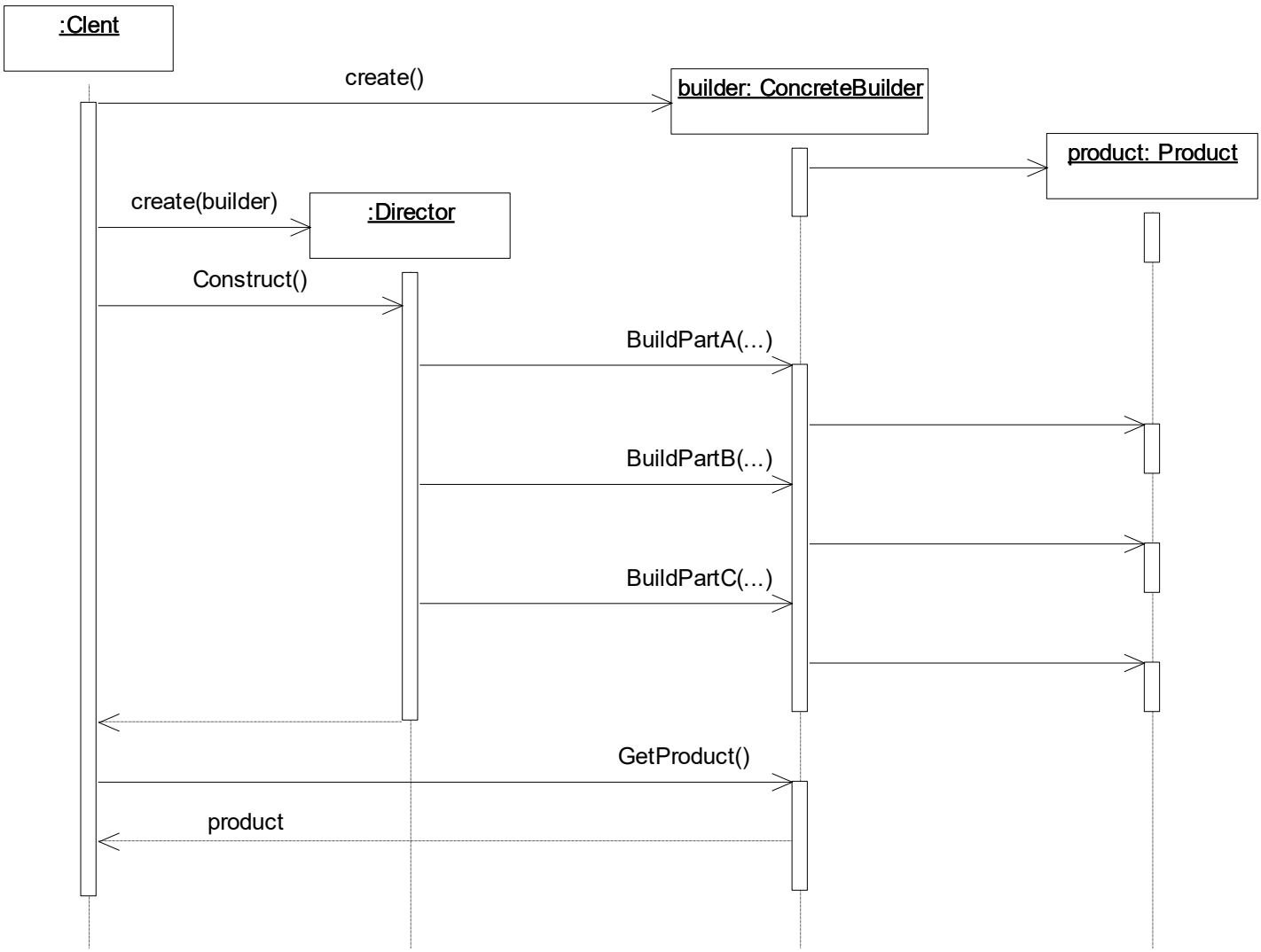
`Builder` (`Converter`) – строитель: задает абстрактный интерфейс для создания частей объекта `Product`.

`ConcreteBuilder` (`ASCIIConverter`, `LaTeXConverter`) – конкретный строитель: конструирует и собирает вместе части продукта посредством реализации интерфейса `Builder`; определяет создаваемое представление и следит за ним; предоставляет интерфейс для доступа к продукту (например, `GetASCIIText`, `GetLaTeXText`).

`Director` (`RTFReader`) – распорядитель: конструирует объект, пользуясь интерфейсом `Builder`.

`Product` (`ASCIIText`, `LaTeXText`) – продукт: представляет сложный конструируемый объект. `ConcreteBuilder` строит внутреннее представление продукта и определяет процесс его сборки; включает классы, которые определяют составные части, в том числе интерфейсы для сборки конечного результата из частей.

Клиент создает объект-распорядитель `Director` и конфигурирует его нужным объектом-строителем `Builder`. Распорядитель уведомляет строителя о том, что нужно построить очередную часть продукта. Строитель обрабатывает запросы распорядителя и добавляет новые части к продукту. Клиент забирает продукт у строителя. Диаграмма взаимодействий иллюстрирует взаимоотношения строителя и распорядителя с клиентом.



Применимость. Используйте паттерн Строитель, когда:

- алгоритм создания сложного объекта не должен зависеть от того, из каких частей состоит объект и как они стыкуются между собой;
- процесс конструирования должен обеспечивать различные представления конструируемого объекта.

Результаты:

- позволяет изменять внутреннее представление продукта. Объект `Builder` предоставляет распорядителю абстрактный **интерфейс** для конструирования продукта, за которым он может скрыть представление и внутреннюю структуру продукта, а также процесс его сборки. Поскольку продукт конструируется через абстрактный интерфейс, то для изменения внутреннего представления достаточно всего лишь определить новый вид строителя;
- изолирует код, реализующий конструирование и представление. Паттерн Строитель улучшает модульность, инкапсулируя способ конструирования и представления сложного объекта. Клиентам ничего не надо знать о классах, определяющих внутреннюю структуру продукта, они отсутствуют в интерфейсе строителя. Каждый конкретный строитель `ConcreteBuilder` содержит весь код, необходимый для создания и сборки конкретного вида продукта. Код пишется только один раз, после чего разные распорядители могут использовать его повторно для построения вариантов продукта из одних и тех же частей. В примере с RTF-документом мы могли бы

определить загрузчик для формата, отличного от RTF, скажем, `HTMLReader`, и воспользоваться теми же самыми классами Converter для генерирования представлений HTML-документов в виде ASCII-текста или LaTeX-текста;

- дает более тонкий контроль над процессом конструирования. В отличие от порождающих паттернов, которые сразу конструируют весь объект целиком, строитель делает это шаг за шагом под управлением распорядителя. И лишь когда продукт завершен, распорядитель забирает его у строителя. Поэтому интерфейс строителя в большей степени отражает процесс конструирования продукта, нежели другие порождающие паттерны. Это и позволяет обеспечить лучший контроль над процессом конструирования, а значит, и над внутренней структурой готового продукта.

Особенности реализации. Обычно существует **абстрактный класс** `Builder`, в котором определены операции для каждого компонента, который распорядитель может «попросить» создать. По умолчанию эти операции ничего не делают. Но в классе конкретного строителя `ConcreteBuilder` они замещены для тех компонентов, в создании которых он принимает участие.

Вот еще некоторые моменты, связанные с реализацией:

- интерфейс сборки и конструирования: строители конструируют свои продукты шаг за шагом. Поэтому интерфейс класса `Builder` должен быть достаточно общим, чтобы обеспечить конструирование при любом виде конкретного строителя. Ключевой вопрос проектирования связан с выбором модели процесса конструирования и сборки. Обычно бывает достаточно модели, в которой результаты выполнения запросов на конструирование просто добавляются к продукту. В примере с RTF-документом строитель преобразует и добавляет очередную лексему к уже конвертированному тексту. Но иногда может потребоваться доступ к частям сконструированного к данному моменту продукта. Примером являются древовидные структуры, скажем, деревья синтаксического разбора, которые строятся снизу вверх. В этом случае строитель должен был бы вернуть узлы-потомки распорядителю, который затем передал бы их назад строителю, чтобы тот мог построить родительские узлы;
- почему нет **абстрактного класса** для продуктов: в типичном случае продукты, изготавливаемые различными строителями, имеют настолько разные представления, что изобретение для них общего родительского класса ничего не дает. Поскольку клиент обычно конфигурирует распорядителя подходящим конкретным строителем, то, надо полагать, ему известно, какой именно подкласс класса `Builder` используется и как нужно обращаться с произведенными продуктами;
- пустые методы класса `Builder` по умолчанию: методы строителя рекомендуется не объявлять как абстрактные. Вместо этого лучше определить их как пустые функции, что позволяет подклассу замещать только те операции, в которых он заинтересован.

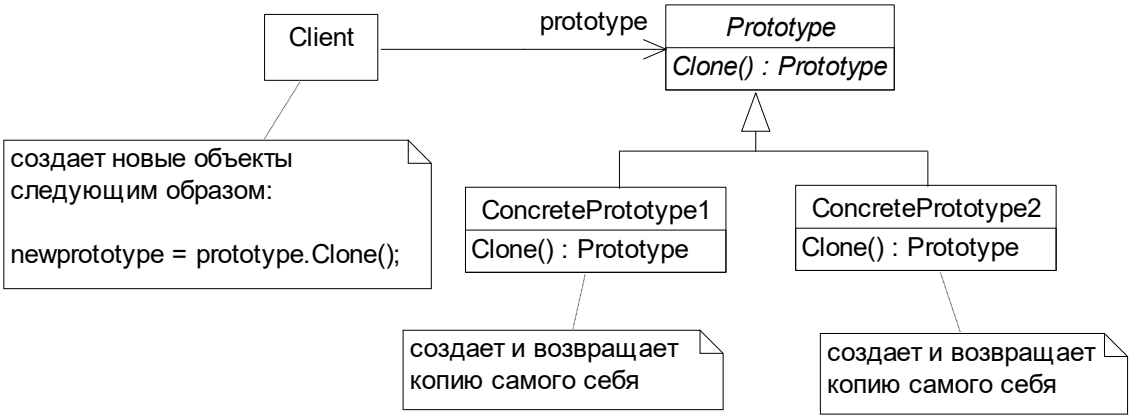
Родственные паттерны.

Абстрактная Фабрика похожа на Строитель в том смысле, что может конструировать сложные объекты. Основное различие между ними в том, что Строитель делает акцент на пошаговом конструировании объекта, а Абстрактная Фабрика – на создании семейств объектов. Строитель возвращает продукт на последнем шаге, тогда как с точки зрения Абстрактной Фабрики продукт возвращается немедленно.

Прототип (Prototype)

Назначение: задает виды создаваемых объектов с помощью экземпляра-прототипа и создает новые объекты путем копирования этого прототипа.

Общая структура решения.



`Prototype` – прототип: объявляет **интерфейс** для клонирования самого себя.

`ConcretePrototype` – конкретные прототипы: реализуют операцию клонирования себя.

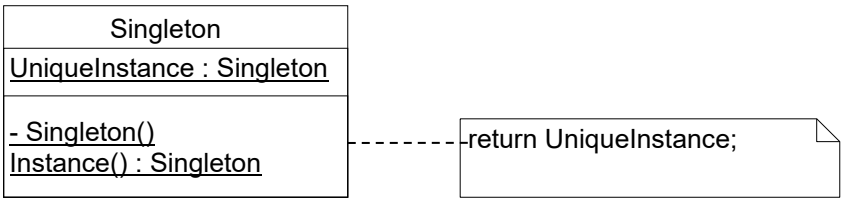
`Client` – клиент: создает новый объект, обращаясь к прототипу с запросом клонировать себя.

Клиент обращается к прототипу, чтобы тот создал свою копию.

Одиночка (Singleton)

Назначение: гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

Общая структура решения.



`Singleton` – одиночка: определяет операцию уровня класса `Instance`, которая позволяет клиентам получать доступ к единственному экземпляру; может нести ответственность за создание собственного уникального экземпляра.

Резюме

Есть два наиболее распространенных способа параметризовать систему классами создаваемых ей объектов. Первый способ – порождение **подклассов** от класса, создающего объекты. Он соответствует паттерну **Фабричный Метод**. Основной недостаток этого метода: требуется создавать новый подкласс лишь для того, чтобы изменить класс продукта. И таких изменений может быть очень много. Например, если создатель продукта сам создается фабричным методом, то придется замещать и создателя тоже.

Другой способ параметризации системы в большей степени основан на **композиции** объектов. Вы определяете объект, которому известно о классах объектов-продуктов, и делаете его параметром системы. Это ключевой аспект таких паттернов, как **Абстрактная Фабрика**, **Строитель** и **Прототип**. Для всех трех характерно создание фабричного объекта, который изготавливает продукты.

В Абстрактной Фабрике фабричный объект производит объекты разных классов. Фабричный объект Строителя постепенно создает сложный продукт, следуя специальному протоколу. Фабричный объект Прототипа изготавливает продукт путем копирования объекта-прототипа. В последнем случае фабричный объект и прототип – это одно и то же, поскольку именно прототип отвечает за возврат продукта.

Паттерн **Фабричный Метод** достаточно прост в применении. Другие паттерны нуждаются в создании новых классов, а Фабричный Метод – только в создании одной новой операции. Часто этот паттерн рассматривается как стандартный способ создания объектов, но его не рекомендуется применять в ситуации, когда инстанцируемый класс никогда не изменяется или когда инстанцирование выполняется внутри операции, которую легко можно заместить в подклассах (например, во время инициализации).

Проекты, в которых используются паттерны Абстрактная Фабрика, Прототип или Строитель, оказываются еще более гибкими, чем те, где применяется Фабричный Метод, но за это приходится платить повышенной сложностью. Часто в начале работы над проектом за основу берется Фабричный Метод, а позже, когда становится ясно, что решение получается недостаточно гибким, выбираются другие паттерны. Владение разными паттернами проектирования открывает широкий выбор при оценке различных критериев.

Структурные паттерны

В структурных паттернах рассматривается вопрос о том, как из классов и объектов образуются более крупные структуры. Структурные паттерны уровня класса используют **наследование** для составления **композиций** из **интерфейсов** и **реализаций**. Например, паттерн **Адаптер** делает интерфейс одного класса (адаптируемого) совместимым с интерфейсом другого, обеспечивая тем самым унифицированную абстракцию разнородных интерфейсов. Это достигается за счет закрытого наследования адаптируемому классу. После этого адаптер выражает свой интерфейс в терминах операций адаптируемого класса.

Вместо композиции интерфейсов или реализаций структурные паттерны уровня объекта компонуют объекты для получения новой функциональности. Дополнительная гибкость в этом случае связана с возможностью изменить композицию объектов во время выполнения, что недопустимо для статической композиции классов.

Примером структурного паттерна уровня объектов является **Компоновщик**. Он описывает построение иерархии классов для двух видов объектов: примитивных и составных. Последние позволяют создавать произвольно сложные структуры из примитивных и других составных объектов. В паттерне **Заместитель** объект берет на себя функции другого объекта. У Заместителя есть много применений. Он может действовать как локальный представитель объекта, находящегося в удаленном адресном пространстве. Или представлять большой объект, загружаемый по требованию. Или ограничивать доступ к критически важному объекту. Заместитель вводит дополнительный косвенный уровень доступа к отдельным свойствам объекта, поэтому он может ограничивать, расширять или изменять эти свойства.

Паттерн **Приспособленец** определяет структуру для совместного использования объектов. Владельцы разделяют объекты, по меньшей мере, по двум причинам: для достижения эффективности и непротиворечивости. Приспособленец акцентирует внимание на эффективности использования памяти. В приложениях, в которых участвует очень много объектов, должны снижаться накладные расходы на хранение. Значительной экономии можно добиться за счет разделения объектов вместо их дублирования. Но объект может быть разделяемым, только если его состояние не зависит от контекста. У объектов-приспособленцев такой зависимости нет. Любая дополнительная информация передается им по мере необходимости. В отсутствие контекстных зависимостей объекты-приспособленцы могут легко разделяться.

Если паттерн Приспособленец дает способ работы с большим числом мелких объектов, то **Фасад** показывает, как один объект может представлять целую **подсистему**. Фасад представляет набор объектов и выполняет свои функции, перенаправляя **сообщения** объектам, которых он представляет.

Паттерн Мост отделяет абстракцию объекта от его реализации, так что их можно изменять независимо.

Паттерн **Декоратор** описывает динамическое добавление объектам новых обязанностей. Это структурный паттерн, который рекурсивно компонует объекты с целью реализации заранее неизвестного числа дополнительных функций. Например, объект-декоратор, содержащий некоторый элемент пользовательского интерфейса, может добавить к нему оформление в виде рамки или тени либо новую функциональность, например возможность прокрутки или изменения масштаба. Два разных оформления прибавляются путем простого вкладывания одного декоратора в другой. Для достижения этой цели каждый объект-декоратор должен соблюдать интерфейс своего компонента и перенаправлять ему сообщения. Свои функции (скажем, рисование рамки вокруг компонента) декоратор может выполнять как до, так и после перенаправления сообщения.

Многие структурные паттерны в той или иной мере связаны друг с другом.

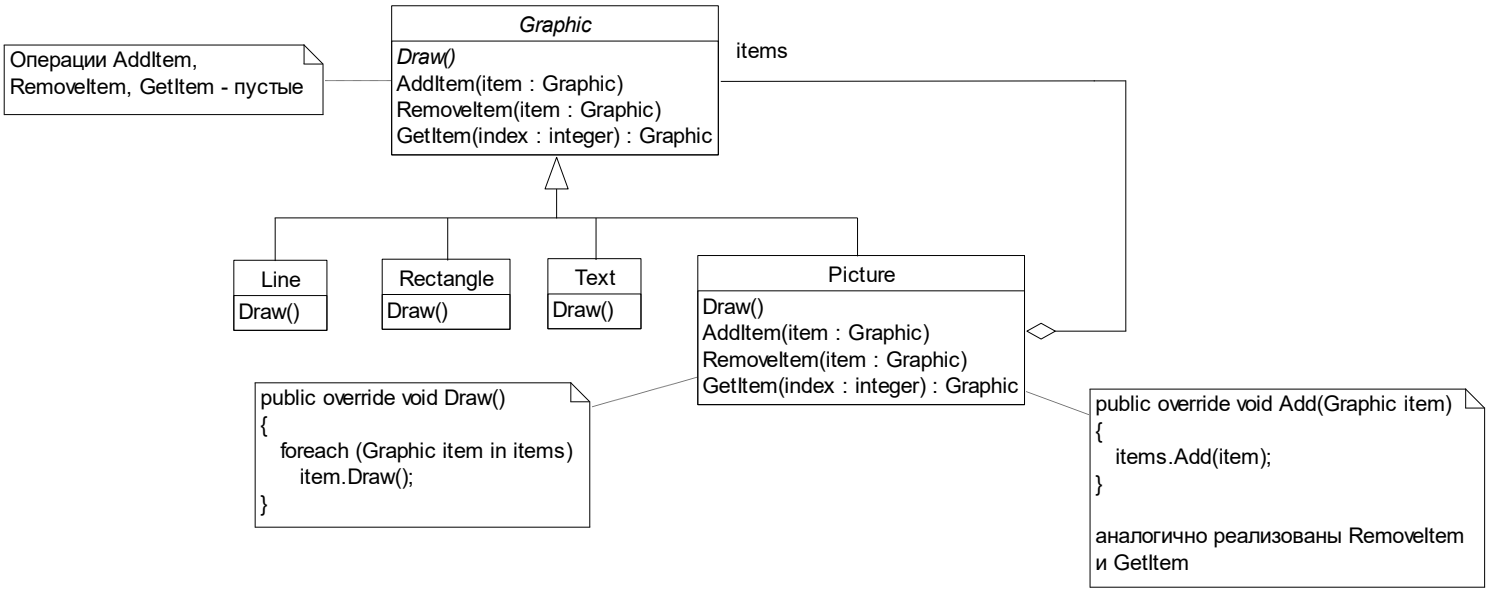
Компоновщик (Composite)

Назначение: компонует объекты в древовидные структуры для представления иерархии часть-целое. Позволяет клиентам единообразно трактовать индивидуальные и составные объекты.

Задача. Такие приложения, как графические редакторы, позволяют пользователям строить сложные документы из более простых компонентов. Проектировщик может сгруппировать мелкие компоненты для формирования более крупных, которые, в свою очередь, могут стать основой для создания еще более крупных. В простой реализации допустимо было бы определить классы графических примитивов, например текста и линий, а также классы, выступающие в роли контейнеров для этих примитивов. Но у такого решения есть существенный недостаток. Программа, в которой эти классы используются, должна по-разному обращаться с примитивами и контейнерами, хотя пользователь чаще всего работает с ними единообразно. Необходимость различать эти объекты усложняет приложение.

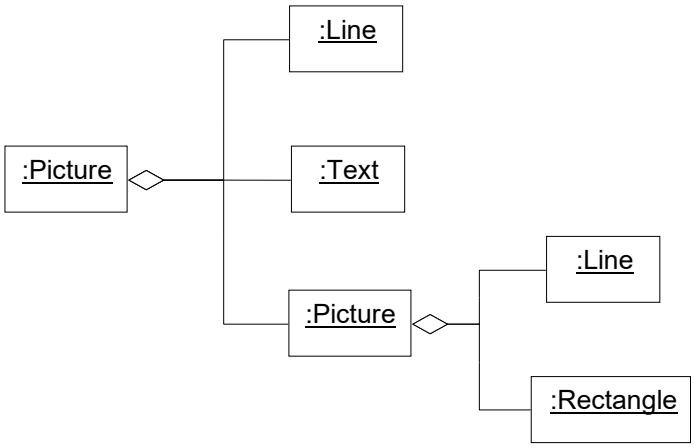
Решение. Паттерн Компоновщик предлагает применить рекурсивную композицию таким образом, что клиенту не придется проводить различие между простыми и составными объектами.

Ключевым моментом здесь является **абстрактный класс**, который представляет одновременно и примитивы, и контейнеры. В графической системе этот класс может называться `Graphic`. В нем объявлены операции, специфичные для каждого вида графического объекта (такие как `Draw`) и общие для всех составных объектов, например операции для доступа и управления внутренними элементами.

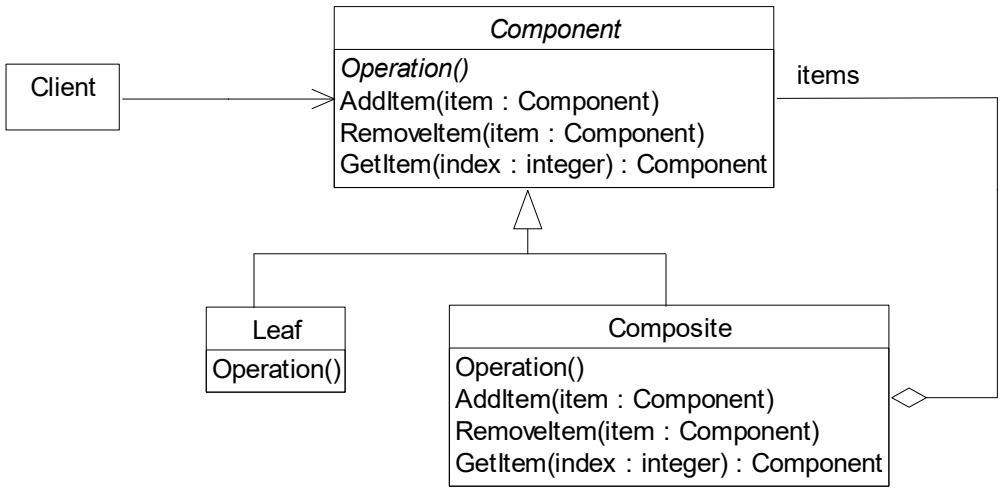


Подклассы `Line`, `Rectangle` и `Text` определяют примитивные графические объекты. В них операция `Draw` реализована соответственно для рисования прямых, прямоугольников и текста. Поскольку у примитивных объектов нет вложенных объектов, то ни один из этих подклассов не реализует операции, относящиеся к управлению контейнером.

Класс `Picture` определяет агрегат, состоящий из объектов `Graphic`. Реализованная в нем операция `Draw` вызывает одноименную функцию для каждого вложенного объекта, которая и выводит их на экран. Поскольку интерфейс класса `Picture` соответствует интерфейсу `Graphic`, то в состав объекта `Picture` могут входить и другие такие же объекты:



Общая структура решения.



`Component` (`Graphic`) – компонент: объявляет **интерфейс** для компонуемых объектов; предоставляет подходящую реализацию операций по умолчанию, общую для всех классов; объявляет интерфейс для управления внутренними объектами; при необходимости реализует интерфейс для доступа к владельцу компонента в рекурсивной структуре.

`Leaf` (`Rectangle`, `Line`, `Text`, и т.п.) – лист: представляет листовые узлы композиции и не имеет внутренних объектов; определяет поведение примитивных объектов в композиции.

`Composite` (`Picture`) – составной объект: определяет поведение компонентов, у которых есть вложенные объекты; хранит внутренние объекты и реализует операции интерфейса класса `Component`, относящиеся к управлению этими объектами.

`Client` – клиент: манипулирует объектами композиции через интерфейс `Component`.

Клиенты используют интерфейс класса `Component` для взаимодействия с объектами в составной структуре. Если получателем запроса является листовый объект `Leaf`, то он и обрабатывает запрос. Когда же получателем является составной объект `Composite`, то обычно он перенаправляет запрос вложенным в него объектам, возможно, выполняя некоторые дополнительные операции до или после перенаправления.

Применимость: используйте этот паттерн, если нужно представить иерархию объектов вида часть-целое, а также когда необходимо, чтобы клиенты единообразно трактовали составные и примитивные объекты.

Результаты. Паттерн Компоновщик:

- определяет иерархии классов, состоящие из примитивных и составных объектов. Из примитивных объектов можно составлять более сложные, которые, в свою очередь, участвуют в более сложных композициях и так далее. Любой клиент, ожидающий примитивный объект, может работать и с составным;
- упрощает архитектуру клиента. Клиенты могут единообразно работать с индивидуальными и объектами и с составными структурами. Обычно клиенту неизвестно, взаимодействует ли он с листовым или составным объектом. Это упрощает

код клиента, поскольку нет необходимости писать функции, ветвящиеся в зависимости от того, с объектом какого класса они работают;

- облегчает добавление новых видов компонентов. Новые подклассы классов `Composite` или `Leaf` будут автоматически работать с уже существующими структурами и клиентским кодом. Изменять клиента при добавлении новых компонентов не нужно;
- способствует созданию общего дизайна компонентов. Однако простота добавления новых компонентов имеет и свои отрицательные стороны – становится трудно наложить ограничения на то, какие объекты могут входить в состав композиции. Иногда желательно, чтобы составной объект мог включать только определенные виды компонентов. Паттерн Компоновщик не позволяет воспользоваться для реализации таких ограничений статической системой типов. Вместо этого следует проводить проверки во время выполнения.

Особенности реализации. При реализации паттерна Компоновщик приходится рассматривать много вопросов:

- явные ссылки на владельцев: хранение в компоненте ссылки на своего владельца может упростить обход структуры и управление ею. Наличие такой ссылки облегчает передвижение вверх по структуре и удаление компонента. Кроме того, ссылки на владельцев помогают поддерживать паттерн **Цепочка Обязанностей**. Обычно ссылку на владельца определяют в классе `Component`. Классы `Leaf` и `Composite` могут унаследовать саму ссылку и операции с ней. При этом важно поддерживать следующий инвариант: если некоторый объект в составной структуре ссылается на другой составной объект как на своего владельца, то для последнего первый является вложенным объектом. Простейший способ гарантировать соблюдение этого условия – изменять владельца компонента только тогда, когда он добавляется или удаляется из составного объекта. Если это удается один раз реализовать в операциях `AddItem` и `RemoveItem`, то **реализация** будет унаследована всеми подклассами, и, значит, инвариант будет поддерживаться автоматически;
- разделение компонентов: часто бывает полезно разделять компоненты, например для уменьшения объема занимаемой памяти. Но если у компонента может быть более одного владельца, то разделение становится проблемой. Возможное решение – позволить компонентам хранить ссылки на нескольких владельцев. Однако в таком случае при распространении запроса по структуре могут возникнуть неоднозначности. Паттерн **Приспособленец** показывает, как следует изменить дизайн, чтобы вовсе отказаться от хранения подобных указателей. Работает он в тех случаях, когда вложенные объекты могут и не посылать **сообщений** своим владельцам, вынеся за свои границы часть внутреннего состояния;
- максимизация интерфейса класса `Component`: одна из целей паттерна Компоновщик – избавить клиентов от необходимости знать, работают ли они с листовым или составным объектом. Для достижения этой цели класс `Component` должен сделать как можно больше операций общими для классов `Composite` и `Leaf`. Обычно класс `Component` предоставляет для этих операций реализации по умолчанию, а подклассы `Composite` и `Leaf` замещают их. Однако иногда эта цель вступает в конфликт с принципом проектирования иерархии классов, согласно которому класс должен определять только логичные для всех его подклассов операции. Класс `Component` поддерживает много операций, не имеющих смысла для класса `Leaf`. Однако можно рассматривать `Leaf` как `Component`, у которого никогда не бывает вложенных объектов. Тогда мы можем определить в классе `Component` операцию доступа к вложенным объектам таким образом, что она никогда не будет возвращать объекты. Подклассы `Leaf` могут использовать эту реализацию по умолчанию, а в подклассах

`Composite` она будет переопределена, чтобы возвращать реальные внутренние объекты составного компонента.

- объявление операций для управления вложенными объектами: хотя в классе `Composite` реализованы операции `AddItem` и `RemoveItem` для добавления и удаления внутренних объектов, но для паттерна Компоновщик важно, в каких классах эти операции объявлены. Надо ли объявлять их в классе `Component` и тем самым делать доступными в `Leaf`, или их следует объявить и определить только в классе `Composite` и его подклассах? Решая этот вопрос, мы должны выбирать между безопасностью и прозрачностью:

    1) если определить **интерфейс** для управления потомками в корне иерархии классов, то мы добиваемся прозрачности, так как все компоненты удается трактовать единообразно. Однако расплачиваться приходится безопасностью, поскольку клиент может попытаться выполнить бессмысленное действие, например добавить или удалить объект из листового узла;
    2) если управление вложенными объектами сделать частью класса Composite, то безопасность удастся обеспечить, ведь любая попытка добавить или удалить объекты из листьев в статически типизированном языке будет перехвачена на этапе компиляции. Но прозрачность мы утрачиваем, ибо у листовых и составных объектов оказываются разные интерфейсы.

  В паттерне Компоновщик особое значение придается прозрачности, а не безопасности. Если для вас важнее безопасность, будьте готовы к тому, что иногда вы можете потерять информацию о типе и придется преобразовывать компонент к типу составного объекта. Как это сделать, не прибегая к небезопасным приведениям типов? Можно, например, объявить в классе `Component` операцию `GetComposite(): Composite`. Класс `Component` реализует ее по умолчанию, возвращая нулевой указатель. А в классе `Composite` эта операция переопределена и возвращает указатель `this` на сам объект. Благодаря этой операции `GetComposite` можно спросить у компонента, является ли он составным. К возвращаемому этой операцией составному объекту допустимо безопасно применять операции `AddItem` и `RemoveItem`.

  Аналогичные проверки на принадлежность классу `Composite` можно выполнить, просто проверив соответствие типу. Разумеется, при таком подходе мы не обращаемся со всеми компонентами единообразно, что плохо. Снова приходится проверять тип, перед тем как предпринять то или иное действие.

  Единственный способ обеспечить прозрачность – это включить в класс `Component` реализации операции `AddItem` и `RemoveItem` по умолчанию. Но появится новая проблема: нельзя реализовать `Component.AddItem()` так, чтобы она никогда не приводила к ошибке. Можно, конечно, сделать данную операцию пустой, но тогда нарушается важное проектное ограничение: попытка добавить что-то в листовый объект, скорее всего, свидетельствует об ошибке.

  Обычно лучшим решением является такая реализация `AddItem` и `RemoveItem` по умолчанию, при которой они завершаются с ошибкой (возможно, возбуждая исключение), если компоненту не разрешено иметь вложенные объекты (для `AddItem`) или аргумент не является вложенным объектом (для `RemoveItem`);
- должен ли `Component` реализовывать список компонентов: может возникнуть желание определить множество вложенных объектов на уровне класса `Component`, в котором объявлены операции доступа и управления внутренними объектами. Но размещение указателя на внутренние объекты в базовом классе приводит к непроизводительному расходу памяти во всех листовых узлах, так как у листа вложенных объектов быть не может. Такой прием можно применить, только если в структуре не слишком много вложенных листовых объектов;

- упорядочение внутренних объектов: во многих случаях порядок следования вложенных объектов важен. В рассмотренном выше примере класса `Graphic` под порядком может пониматься Z-порядок расположения внутренних объектов. В составных объектах, описывающих деревья синтаксического разбора, составные операторы могут быть экземплярами класса `Composite`, порядок следования внутренних объектов которых отражает семантику программы. Если порядок их следования важен, необходимо учитывать его при проектировании интерфейсов доступа и управления вложенными объектами. В этом может помочь паттерн **Итератор**;
- кэширование для повышения производительности: если приходится часто обходить композицию или производить в ней поиск, то класс `Composite` может кэшировать информацию об обходе и поиске. Кэшировать разрешается либо полученные результаты, либо только информацию, достаточную для ускорения обхода или поиска. Например, класс `Picture` из приведенного в начале примера, мог бы кэшировать охватывающие прямоугольники своих вложенных объектов. При рисовании или выборе эта информация позволила бы пропускать те объекты, которые невидимы в текущем окне. Любое изменение компонента должно делать кэши всех его владельцев недействительными. Поэтому, если вы решите воспользоваться кэшированием, необходимо определить интерфейс, позволяющий уведомить составные объекты о недействительности их кэшей;
- какая структура данных лучше всего подходит для хранения компонентов: составные объекты могут хранить свои вложенные объекты в самых разных структурах данных, включая связанные списки, деревья, массивы и хэш-таблицы. Выбор структуры данных определяется, как всегда, эффективностью. Собственно говоря, вовсе не обязательно пользоваться какой-либо из универсальных структур. Может быть, будет удобно каждый внутренний объект представить в виде отдельной переменной. Правда, для этого каждый подкласс `Composite` должен будет реализовывать свой собственный интерфейс управления памятью.

Родственные паттерны.

Отношение «компонент–родитель» иногда используется в паттерне **Цепочка Обязанностей**.

Паттерн **Декоратор** часто применяется совместно с Компоновщиком. Когда декораторы и компоновщики используются вместе, у них обычно бывает общий родительский класс. Поэтому декораторам придется поддержать **интерфейс** компонентов такими операциями, как `AddItem`, `RemoveItem` и `GetItem`.

Паттерн **Приспособленец** позволяет разделять компоненты, но ссылаться на своих владельцев они уже не могут.

Итератор можно использовать для обхода составных объектов.

Посетитель локализует операции и поведение, которые в противном случае пришлось бы распределять между классами `Composite` и `Leaf`.

Декоратор (Decorator)

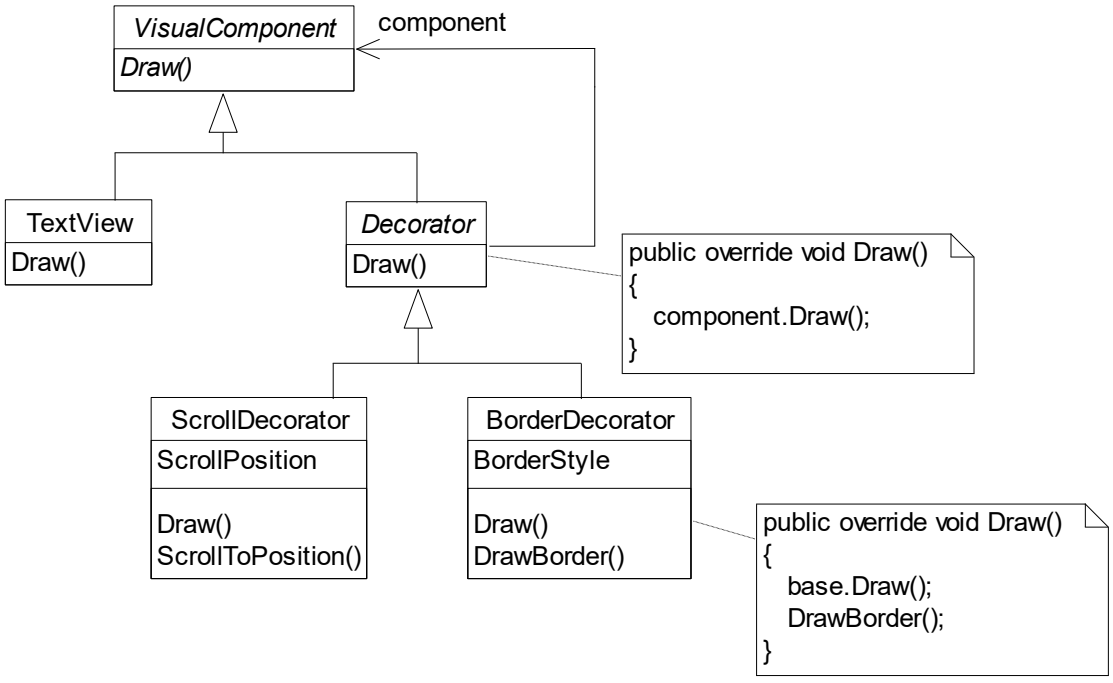
Другое название: Wrapper (Обертка).

Назначение: динамически добавляет объекту новые обязанности, является гибкой альтернативой порождению подклассов с целью расширения функциональности.

Задача. Иногда бывает нужно возложить дополнительные обязанности на отдельный объект, а не на класс в целом. Так, библиотека для построения графических интерфейсов пользователя должна уметь добавлять новое свойство, скажем, рамку или новое поведение (например, возможность прокрутки к любому элементу интерфейса).

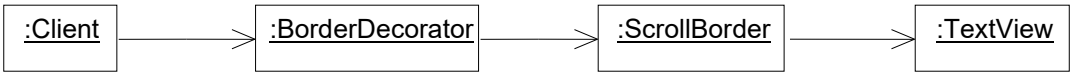
Добавить новые обязанности допустимо с помощью **наследования**. При наследовании класса с рамкой вокруг каждого экземпляра подкласса будет рисоваться рамка. Однако это решение статическое, а значит, недостаточно гибкое. Клиент не может управлять оформлением компонента рамкой.

Решение. Более гибким является другой подход: поместить компонент в другой объект, называемый декоратором, который как раз и добавляет рамку. Декоратор следует интерфейсу декорируемого объекта, поэтому его присутствие прозрачно для клиентов компонента. Декоратор переадресует запросы внутреннему компоненту, но может выполнять и дополнительные действия (например, рисовать рамку) до или после переадресации. Поскольку декораторы прозрачны, они могут вкладываться друг в друга, добавляя тем самым любое число новых обязанностей.



Предположим, что имеется объект класса `TextView`, который отображает текст в окне. По умолчанию `TextView` не имеет полос прокрутки, поскольку они не всегда нужны. Но при необходимости их удастся добавить с помощью декоратора `ScrollDecorator`. Допустим, что еще мы хотим добавить жирную сплошную рамку вокруг объекта `TextView`. Здесь может помочь декоратор `BorderDecorator`. Мы просто компонуем оба декоратора и получаем искомый результат.

На диаграмме объектов показано, как композиция объекта `TextView` с объектами `BorderDecorator` и `ScrollDecorator` порождает элемент для ввода текста, окруженный рамкой и снабженный полосой прокрутки.

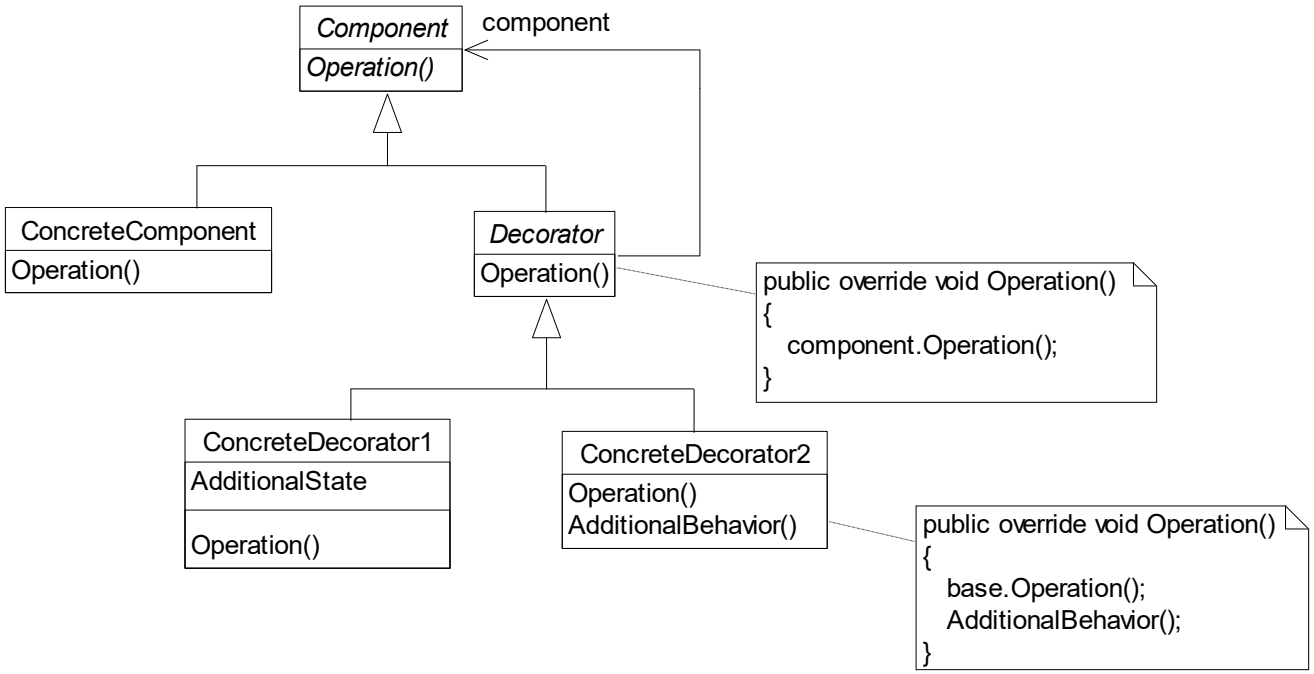


Классы `ScrollDecorator` и `BorderDecorator` являются подклассами `Decorator` – **абстрактного класса**, который представляет визуальные компоненты, применяемые для оформления других визуальных компонентов.

`VisualComponent` – это абстрактный класс для представления визуальных объектов. В нем определен интерфейс для рисования и обработки событий. Отметим, что класс `Decorator` просто переадресует запросы на рисование своему компоненту, а его подклассы могут расширять эту операцию.

Подклассы `Decorator` могут добавлять любые операции для обеспечения необходимой функциональности. Так, операция `ScrollTo` объекта `ScrollDecorator` позволяет другим объектам выполнять прокрутку, если им известно о присутствии объекта `ScrollDecorator`. Важная особенность этого паттерна состоит в том, что декораторы могут употребляться везде, где возможно появление самого объекта `VisualComponent`. Поэтому клиент не может отличить декорированный объект от недекорированного, а значит, никоим образом не зависит от наличия или отсутствия оформлений.

Общая структура решения.



`Component` (`VisualComponent`) – компонент: определяет интерфейс для объектов, на которые могут быть динамически возложены дополнительные обязанности.

`ConcreteComponent` (`TextView`) – конкретный компонент: определяет объект, на который возлагаются дополнительные обязанности.

`Decorator` – декоратор: хранит ссылку на объект `Component` и определяет интерфейс, соответствующий интерфейсу `Component`.

`ConcreteDecorator` (`BorderDecorator`, `ScrollDecorator`) – конкретные декораторы: реализуют дополнительные обязанности, возлагаемые на компонент.

`Decorator` переадресует запросы объекту `Component`. Также может выполнять и дополнительные операции до и после переадресации.

Применимость. Используйте паттерн Декоратор:

- для динамического, прозрачного для клиентов добавления обязанностей объектам;
- для реализации обязанностей, которые могут быть сняты с объекта;
- когда расширение путем порождения подклассов по каким-то причинам неудобно или невозможно – иногда приходится реализовывать много независимых расширений, так что порождение подклассов для поддержки всех возможных комбинаций приведет к комбинаторному росту их числа. В других случаях определение класса может быть скрыто или почему-либо еще недоступно, так что породить от него подкласс просто нельзя.

Результаты:

- большая гибкость, нежели у статического **наследования**: паттерн Декоратор позволяет более гибко добавлять объекту новые обязанности, чем было бы возможно в случае статического наследования. Декоратор может добавлять и удалять обязанности во время выполнения программы. При использовании же наследования требуется создавать новый класс для каждой дополнительной обязанности (для нашего примера

это могли бы быть `BorderedTextView` или даже `BorderedScrollableTextView`), что ведет к увеличению числа классов и, как следствие, к возрастанию сложности системы. Кроме того, применение нескольких декораторов к одному компоненту позволяет произвольным образом сочетать обязанности. Декораторы позволяют легко добавить одно и то же свойство дважды. Например, чтобы окружить объект `TextView` двойной рамкой, нужно просто добавить два декоратора `BorderDecorator`;

- позволяет избежать перегруженных функциями классов на верхних уровнях иерархии: Декоратор разрешает добавлять новые обязанности по мере необходимости. Вместо того чтобы пытаться поддержать все мыслимые возможности в одном сложном, допускающем разностороннюю настройку классе, вы можете определить простой класс и постепенно наращивать его функциональность с помощью декораторов. В результате приложение уже не платит за неиспользуемые функции. Нетрудно также определять новые виды декораторов независимо от классов, которые они расширяют, даже если первоначально такие расширения не планировались. При расширении же сложного класса обычно приходится вникать в детали, не имеющие отношения к добавляемой функции;
- декоратор и его компонент не идентичны: декоратор действует как прозрачное обрамление, но при этом декорированный компонент все же не идентичен исходному. При использовании декораторов это следует иметь в виду;
- множество мелких объектов: при использовании в проекте паттерна Декоратор нередко получается система, составленная из большого числа мелких объектов, которые похожи друг на друга и различаются только способом взаимосвязи, а не классом и не значениями своих внутренних переменных. Хотя проектировщик, разбирающийся в устройстве такой системы, может легко настроить ее, но изучать и отлаживать ее очень тяжело.

Особенности реализации. Применение паттерна Декоратор требует рассмотрения нескольких вопросов:

- соответствие интерфейсов: интерфейс декоратора должен соответствовать интерфейсу декорируемого компонента, поэтому классы `ConcreteDecorator` должны наследовать общему классу или реализовывать общий интерфейс;
- отсутствие абстрактного класса `Decorator`: нет необходимости определять абстрактный класс `Decorator`, если планируется добавить всего одну обязанность. Так часто происходит, когда вы работаете с уже существующей иерархией классов, а не проектируете новую. В таком случае ответственность за переадресацию запросов, которую обычно несет класс `Decorator`, можно возложить непосредственно на `ConcreteDecorator`;
- облегченные классы `Component`: чтобы можно было гарантировать соответствие интерфейсов, компоненты и декораторы должны наследовать общему классу `Component`. Важно, чтобы этот класс был настолько легким, насколько возможно. Иными словами, он должен определять интерфейс, а не хранить данные. В противном случае декораторы могут стать весьма тяжеловесными, и применять их в большом количестве будет накладно. Включение большого числа функций в класс `Component` также увеличивает вероятность, что конкретным подклассам придется платить за то, что им не нужно;
- изменение облика, а не внутреннего устройства объекта: декоратор можно рассматривать как появившуюся у объекта оболочку, которая изменяет его поведение. Альтернатива – изменение внутреннего устройства объекта, хорошим примером чего может служить паттерн **Стратегия**. Стратегии лучше подходят в ситуациях, когда

класс `Component` уже достаточно тяжел, так что применение паттерна Декоратор обходится слишком дорого. В паттерне Стратегия компоненты передают часть своей функциональности отдельному объекту-стратегии, поэтому изменить или расширить поведение компонента допустимо, заменив этот объект. Например, мы можем поддержать разные стили рамок, поручив рисование рамки специальному объекту `Border`. Объект `Border` является примером обьекта-стратегии: В данном случае он инкапсулирует стратегию рисования рамки. Число стратегий может быть любым, поэтому эффект такой же, как от рекурсивной вложенности декораторов.

Родственные паттерны.

Адаптер: если Декоратор изменяет только обязанности объекта, но не его интерфейс, то адаптер придает объекту совершенно новый интерфейс.

Компоновщик: Декоратор можно считать вырожденным случаем составного объекта, у которого есть только один компонент. Однако Декоратор добавляет новые обязанности, **агрегирование** объектов не является его целью.

Стратегия: Декоратор позволяет изменить внешний облик объекта, Стратегия – его внутреннее содержание. Это два взаимодополняющих способа изменения объекта.

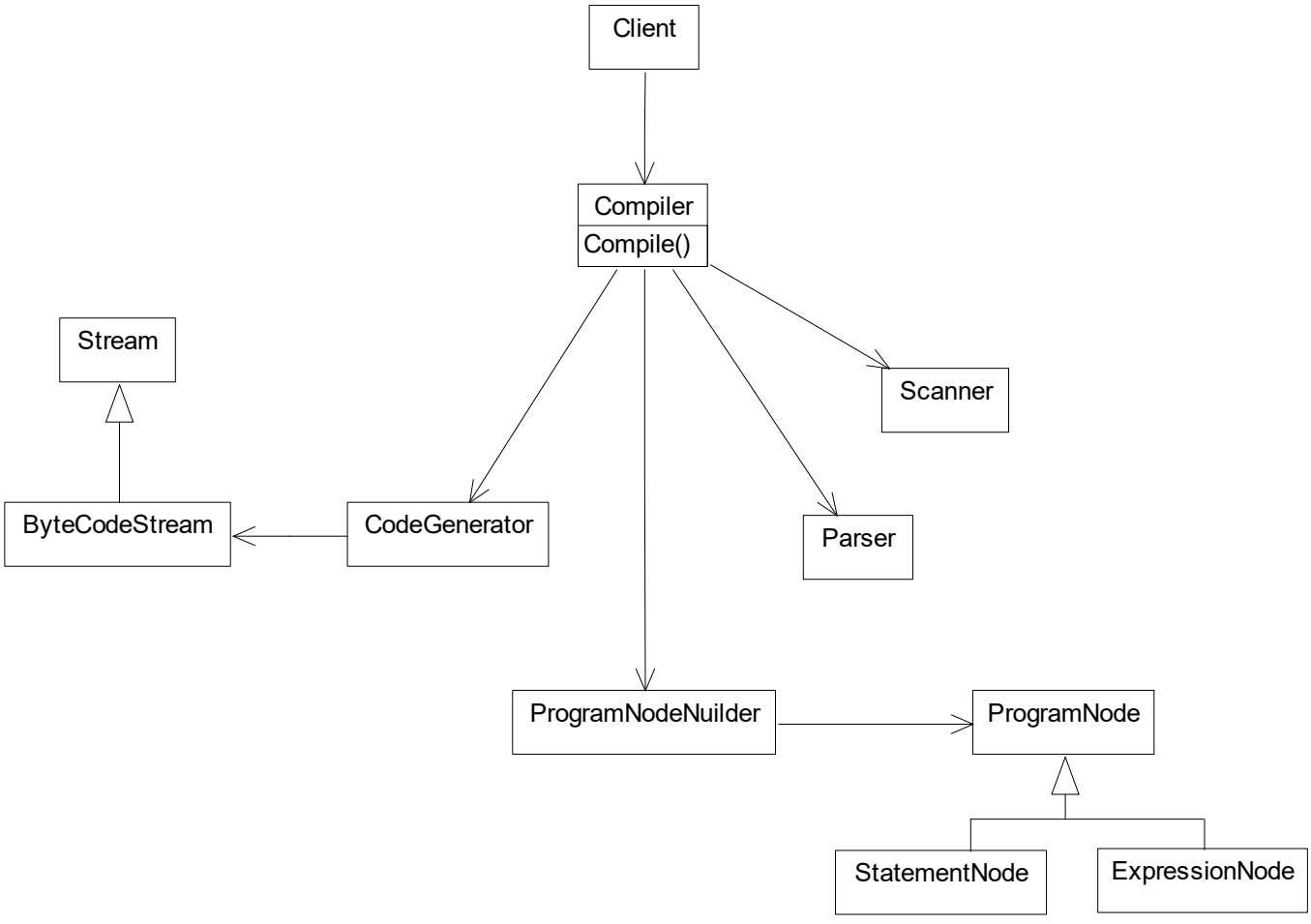
Фасад (Facade)

Назначение: предоставляет унифицированный **интерфейс** вместо набора интерфейсов некоторой подсистемы, определяет интерфейс более высокого уровня, который упрощает использование подсистемы.

Задача. Разбиение на подсистемы облегчает проектирование сложной системы в целом. Цель проектирования – свести к минимуму зависимость подсистем друг от друга и обмен информацией между ними.

Решение. Один из способов, сопутствующих решению этой задачи – введение объекта фасад, предоставляющего единый упрощенный интерфейс к более сложным системным средствам.

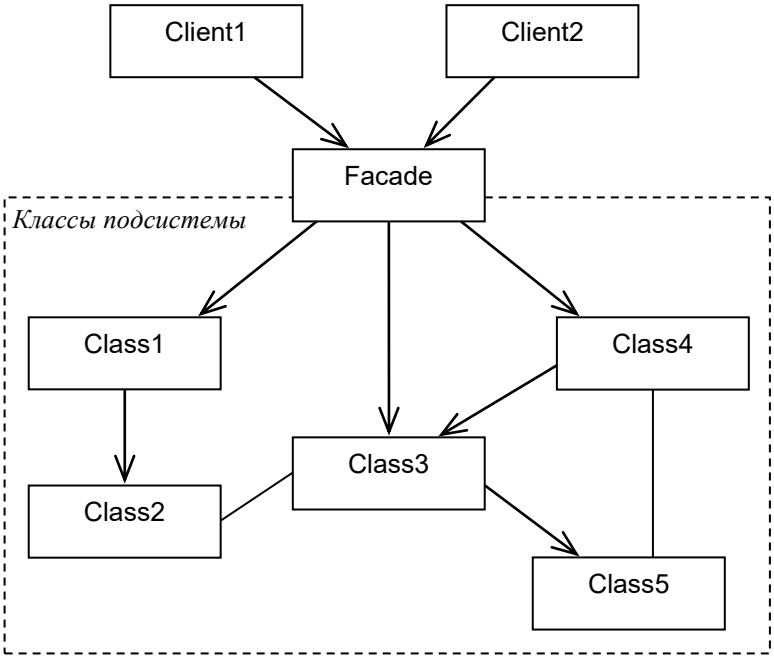
Рассмотрим, например, среду программирования, которая дает приложениям доступ к подсистеме компиляции. В этой подсистеме имеются такие классы, как `Scanner` (лексический анализатор), `Parser` (синтаксический анализатор), `ProgramNode` (узел программы), `ByteCodeStream` (поток байтовых кодов) и `ProgramNodeBuilder` (строитель узла программы). Все вместе они составляют компилятор. Некоторым специализированным приложениям, возможно, понадобится прямой доступ к этим классам. Но для большинства клиентов компилятора такие детали, как синтаксический разбор и генерация кода, обычно не нужны; им просто требуется откомпилировать некоторую программу. Для таких клиентов применение мощного, но низкоуровневого интерфейса подсистемы компиляции только усложняет задачу.



Чтобы предоставить интерфейс более высокого уровня, изолирующий клиента от этих классов, в подсистему компиляции включен класс `Compiler` (компилятор). Он определяет унифицированный интерфейс ко всем возможностям компилятора. Класс `Compiler` выступает в роли фасада: предлагает простой интерфейс к более сложной подсистеме. Он «склеивает» классы, реализующие функциональность компилятора, но не скрывает их полностью. Благодаря

фасаду компилятора работа большинства программистов облегчается. При этом те, кому нужен доступ к средствам низкого уровня, не лишаются его.

Общая структура решения.



`Facade` (`Compiler`) – фасад: знает, каким классам подсистемы адресовать запрос; переадресует запросы клиентов подходящим объектам внутри подсистемы.

`Class1`, `Class2`, … (`Scanner`, `Parser`, `ProgramNode` и т.д.) – классы подсистемы: реализуют функциональность подсистемы; выполняют работу, порученную объектом `Facade`; ничего не знают о существовании фасада, то есть не хранят ссылок на него.

Клиенты `Client1`, `Client2`, … не имеют прямого доступа к объектам подсистемы. Они общаются с подсистемой, посылая запросы объекту-фасаду. Он переадресует их подходящим объектам внутри подсистемы.

Применимость. Используйте паттерн Фасад, когда:

- хотите предоставить простой **интерфейс** к сложной подсистеме. Часто подсистемы усложняются по мере развития. Применение большинства паттернов приводит к появлению меньших классов, но в большем количестве. Такую подсистему проще повторно использовать и настраивать под конкретные нужды, но вместе с тем применять подсистему без настройки становится труднее. Фасад предлагает некоторый вид системы по умолчанию, устраивающий большинство клиентов. И лишь те объекты, которым нужны более широкие возможности настройки, могут обратиться напрямую к тому, что находится за фасадом;
- между клиентами и классами реализации абстракции существует много зависимостей. Фасад позволит отделить подсистему как от клиентов, так и от других подсистем, что, в свою очередь, способствует повышению степени независимости и переносимости;
- вы хотите разложить подсистему на отдельные слои. Используйте Фасад для определения точки входа на каждый уровень подсистем. Если подсистемы зависят друг

от друга, то зависимость можно упростить, разрешив подсистемам обмениваться информацией только через фасады.

Результаты. У паттерна Фасад есть следующие преимущества:

- изолирует клиентов от компонентов подсистемы, уменьшая тем самым число объектов, с которыми клиентам приходится иметь дело, и упрощая работу с подсистемой;
- позволяет ослабить связанность между подсистемой и ее клиентами. Зачастую компоненты подсистемы сильно связаны. Слабая связанность позволяет видоизменять компоненты, не затрагивая при этом клиентов. Фасады помогают разложить систему на слои и структурировать зависимости между объектами, а также избежать сложных и циклических зависимостей. Это может оказаться важным, если клиент и подсистема реализуются независимо Уменьшение числа зависимостей на стадии компиляции чрезвычайно важно в больших системах. Фасад может также упростить процесс переноса системы на другие платформы, поскольку уменьшается вероятность того, что в результате изменения одной подсистемы понадобится изменять и все остальные;
- фасад не препятствует приложениям напрямую обращаться к классам подсистемы, если это необходимо. Таким образом, у вас есть выбор между простотой и общностью.

Особенности реализации. При реализации паттерна Фасад следует обратить внимание на следующие вопросы:

- уменьшение степени связанности клиента с подсистемой: степень связанности можно значительно уменьшить, если сделать класс `Facade` абстрактным. Его конкретные подклассы будут соответствовать различным реализациям подсистемы. Тогда клиенты смогут взаимодействовать с подсистемой через **интерфейс** абстрактного класса `Facade`. Это изолирует клиентов от информации о том, какая реализация подсистемы используется. Вместо порождения подклассов можно сконфигурировать объект `Facade` различными объектами подсистем. Для настройки фасада достаточно заменить один или несколько таких объектов;
- открытые и закрытые классы подсистем: подсистема похожа на класс в том отношении, что у обоих есть интерфейсы и оба что-то инкапсулируют. Класс инкапсулирует состояние и операции, а подсистема – классы. И если полезно различать открытый и закрытый интерфейсы класса, то не менее разумно говорить об открытом и закрытом интерфейсах подсистемы. Открытый интерфейс подсистемы состоит из классов, к которым имеют доступ все клиенты; закрытый интерфейс доступен только для расширения подсистемы. Класс `Facade`, конечно же, является частью открытого интерфейса, но это не единственная часть. Другие классы подсистемы также могут быть открытыми.

Родственные паттерны.

Паттерн **Абстрактная Фабрика** допустимо использовать вместе с Фасадом, чтобы предоставить интерфейс для создания объектов подсистем способом, не зависимым от этих подсистем. Абстрактная Фабрика может выступать и как альтернатива Фасаду, чтобы скрыть платформенно-зависимые классы.

Паттерн **Посредник** аналогичен Фасаду в том смысле, что абстрагирует функциональность существующих классов. Однако назначение посредника – абстрагировать произвольное взаимодействие между «сотрудничающими» объектами. Часто он централизует функциональность, не присущую ни одному из них. Коллеги посредника обмениваются информацией именно с ним, а не напрямую между собой. Напротив, фасад просто абстрагирует

интерфейс объектов подсистемы, чтобы ими было проще пользоваться. Он не определяет новой функциональности, и классам подсистемы ничего не известно о его существовании.

Обычно требуется только один фасад. Поэтому объекты фасадов часто бывают **Одиночками**.

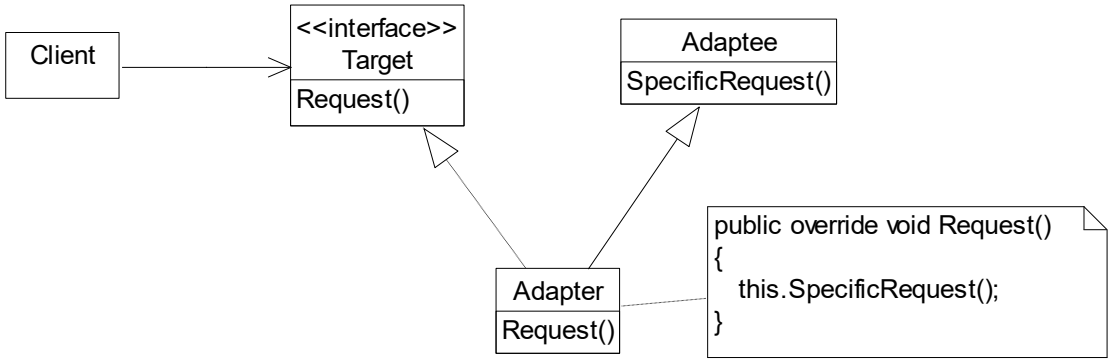
Адаптер (Adapter)

Другое название: Wrapper (Обертка).

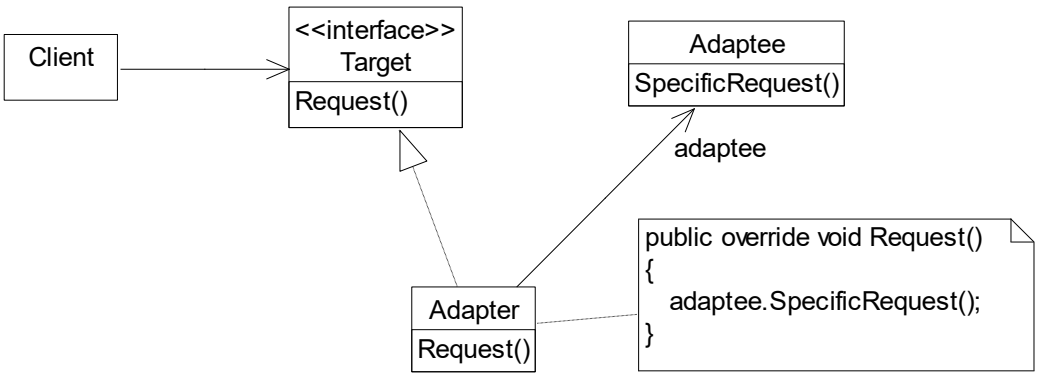
Назначение: преобразует **интерфейс** одного класса в интерфейс другого, который ожидают клиенты, обеспечивает совместную работу классов с несовместимыми интерфейсами, которая без него была бы невозможна.

Общая структура решения.

Адаптер класса применяет реализацию одного интерфейса одновременно с наследованием от другого класса, чтобы адаптировать этот класс к заданному интерфейсу:



Адаптер объекта применяет реализацию интерфейса и композицию объектов:



`Target` – целевой объект: определяет зависящий от предметной области **интерфейс**, которым пользуется `Client`.

`Client` – клиент: вступает во взаимоотношения с объектами, удовлетворяющими интерфейсу `Target`.

`Adaptee` – адаптируемый объект: определяет существующий интерфейс, который нуждается в адаптации.

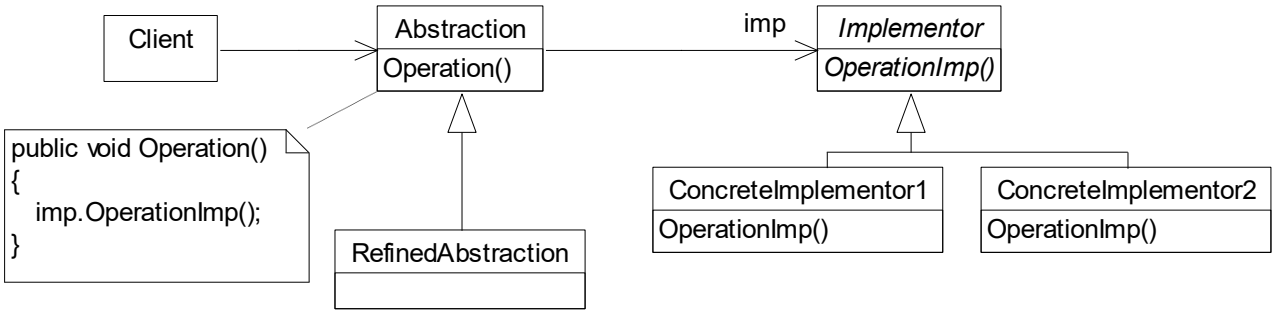
`Adapter` – адаптер: адаптирует интерфейс `Adaptee` к интерфейсу `Target`.

Клиенты вызывают операции экземпляра адаптера `Adapter`. В свою очередь, адаптер вызывает операции адаптируемого объекта или класса `Adaptee`, который и выполняет запрос.

Мост (Bridge)

Назначение: отделить абстракцию от ее **реализации** так, чтобы то и другое можно было изменять независимо.

Общая структура решения.



`Abstraction` – абстракция: определяет **интерфейс** абстракции; хранит ссылку на объект типа `Implementor`.

`RefinedAbstraction` – уточненная абстракция: расширяет интерфейс, определенный абстракцией `Abstraction`.

`Implementor` – реализатор: определяет интерфейс для классов реализации, не обязан точно соответствовать интерфейсу класса `Abstraction`, т.е. оба интерфейса могут быть совершенно различны. Обычно интерфейс класса `Implementor` предоставляет только примитивные операции, а класс `Abstraction` определяет операции более высокого уровня, базирующиеся на этих примитивах.

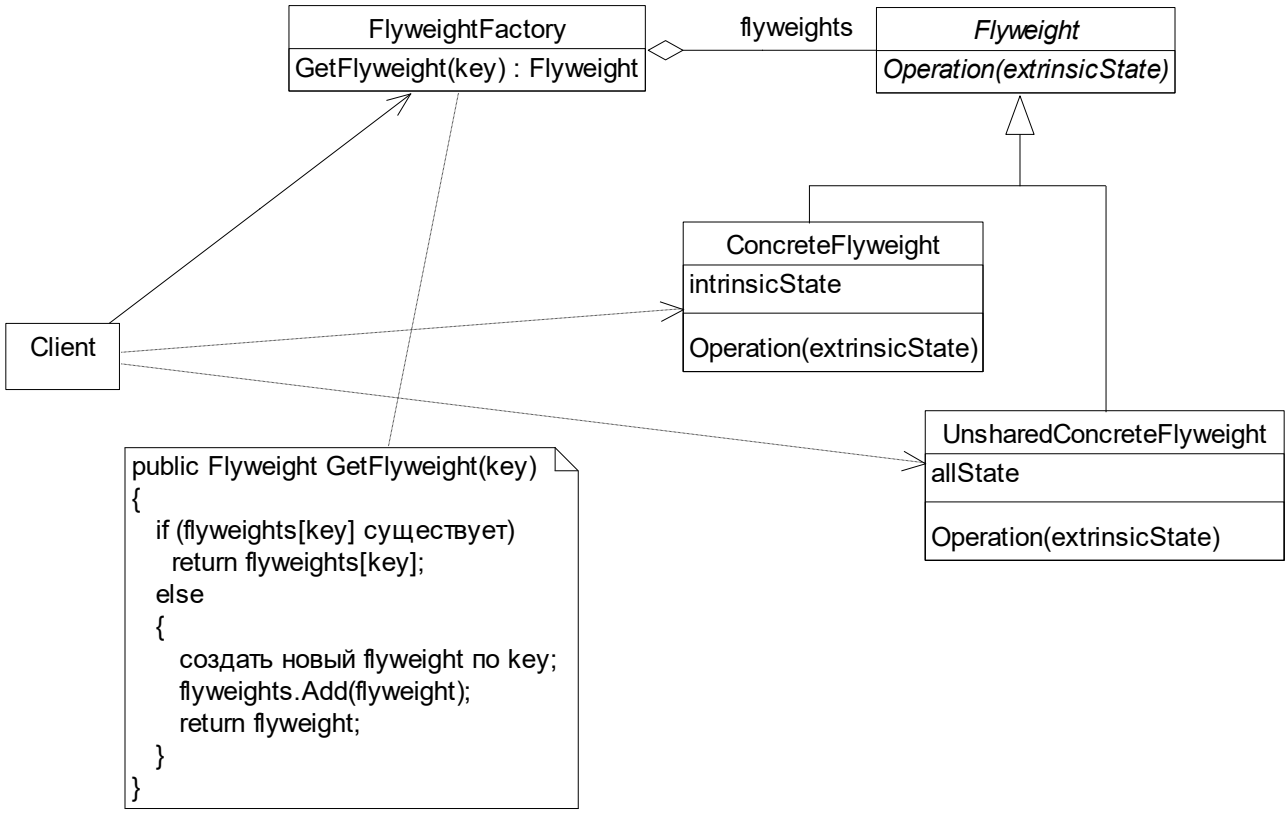
`ConcreteImplementor` – конкретные реализаторы: содержат конкретные реализации интерфейса класса `Implementor`.

Объект `Abstraction` перенаправляет запросы клиента своему объекту `Implementor`.

Приспособленец (Flyweight)

Назначение: использует разделение для эффективной поддержки множества мелких объектов.

Общая структура решения.



`Flyweight` – приспособленец: объявляет интерфейс, с помощью которого конкретные приспособленцы могут получать внешнее состояние или как-то воздействовать на него.

`ConcreteFlyweight` – конкретный приспособленец: реализует интерфейс класса `Flyweight` и добавляет при необходимости внутреннее состояние. Объект класса `ConcreteFlyweight` должен быть разделяемым. Любое сохраняемое им состояние должно быть внутренним, то есть не зависящим от контекста.

`UnsharedConcreteFlyweight` – неразделяемый конкретный приспособленец: не все подклассы `Flyweight` обязательно должны быть разделяемыми – интерфейс `Flyweight` допускает разделение, но не навязывает его. Обычно у объектов `UnsharedConcreteFlyweight` на некотором уровне структуры есть вложенные объекты в виде объектов класса `ConcreteFlyweight`.

`FlyweightFactory` – фабрика приспособленцев: создает объекты-приспособленцы и управляет ими; обеспечивает должное разделение приспособленцев. Когда клиент запрашивает приспособленца, объект `FlyweightFactory` предоставляет существующий экземпляр или создает новый, если готового еще нет.

`Client` – клиент: хранит ссылки на одного или нескольких приспособленцев; вычисляет или хранит внешнее состояние приспособленцев.

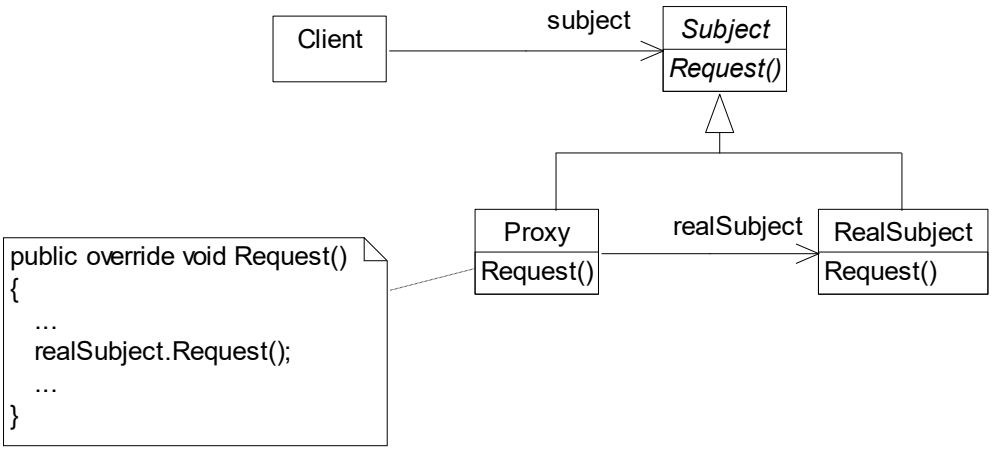
Состояние, необходимое приспособленцу для нормальной работы, можно охарактеризовать как внутреннее или внешнее. Первое хранится в самом объекте

ConcreteFlyweight. Внешнее состояние хранится или вычисляется клиентами. Клиент передает его приспособленцу при вызове операций. Клиенты не должны создавать экземпляры класса ConcreteFlyweight напрямую, а могут получать их только от объекта FlyweightFactory. Это позволит гарантировать корректное разделение.

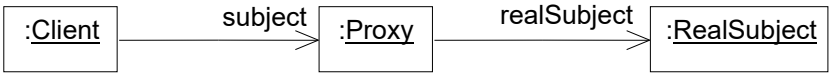
Заместитель (Proxy)

Назначение: является суррогатом другого объекта и контролирует доступ к нему.

Общая структура решения.



Пример диаграммы объектов для структуры с заместителем во время выполнения:



Proxy – заместитель: хранит ссылку, которая позволяет ему обратиться к реальному субъекту, при этом может решать следующие задачи:

- предоставляет **интерфейс**, идентичный интерфейсу Subject, так что заместитель всегда может быть подставлен вместо реального субъекта;
- может контролировать доступ к реальному субъекту и отвечать за его создание и удаление;
- может отвечать за кодирование запроса и его аргументов и отправление закодированного запроса реальному субъекту в другом адресном пространстве («Удаленный Заместитель»);
- может кэшировать дополнительную информацию о реальном субъекте, чтобы отложить его создание («Виртуальный Заместитель»);
- может проверять, имеет ли вызывающий объект необходимые для выполнения запроса права («Защищающий Заместитель»);

Subject – субъект: определяет общий для RealSubject и Proxy интерфейс, так что класс Proxy можно использовать везде, где ожидается RealSubject;

`RealSubject` – реальный субъект: определяет реальный объект, представленный заместителем.

Клиент владеет указателем на субъект, в который фактически записан заместитель. Заместитель при необходимости переадресует запросы объекту `RealSubject`. Детали зависят от вида заместителя.

Резюме

Структурные паттерны похожи между собой, особенно когда речь идет об их участниках и взаимодействиях. Вероятное объяснение такому явлению: все структурные паттерны основаны на небольшом множестве языковых механизмов структурирования кода и объектов (**наследовании** для паттернов уровня класса и **композиции** для паттернов уровня объектов). Но имеющееся сходство может быть обманчиво, ибо с помощью разных паттернов можно решать совершенно разные задачи. В этом разделе сопоставлены группы структурных паттернов для того, чтобы ярче продемонстрировать их сравнительные достоинства и недостатки.

Адаптер и Мост. У паттернов **Адаптер** и **Мост** есть несколько общих атрибутов. Тот и другой повышают гибкость, вводя дополнительный посреднический уровень при обращении к другому объекту. Оба перенаправляют запросы другому объекту, используя иной интерфейс.

Основное различие между Адаптером и Мостом в их назначении. Цель первого – устранить несовместимость между двумя существующими интерфейсами. При разработке Адаптера не учитывается, как эти интерфейсы реализованы и то, как они могут независимо развиваться в будущем. Он должен лишь обеспечить совместную работу двух независимо разработанных классов, так чтобы ни один из них не пришлось переделывать. С другой стороны, мост связывает абстракцию с ее, возможно, многочисленными **реализациями**. Данный паттерн предоставляет клиентам стабильный интерфейс, позволяя в то же время изменять классы, которые его реализуют. Мост также подстраивается под новые реализации, появляющиеся в процессе развития системы.

В связи с описанными различиями Адаптер и Мост часто используются в разные моменты жизненного цикла системы. Когда выясняется, что два несовместимых класса должны работать вместе, следует обратиться к адаптеру. Тем самым удастся избежать дублирования кода. Заранее такую ситуацию предвидеть нельзя. Наоборот, пользователь Моста с самого начала понимает, что у абстракции может быть несколько реализаций, и развитие того и другого будет идти независимо.

Адаптер обеспечивает работу после того, как нечто спроектировано; Мост – до того.

Фасад можно представлять себе как адаптер к набору других объектов. Но при такой интерпретации легко не заметить такой нюанс: Фасад определяет новый интерфейс, тогда как

Адаптер повторно использует уже имеющийся. Подчеркнем, что Адаптер заставляет работать вместе два существующих интерфейса, а не определяет новый.

Компоновщик, Декоратор и Заместитель. У **Компоновщика** и **Декоратора** аналогичные структурные диаграммы, свидетельствующие о том, что оба паттерна основаны на рекурсивной композиции и предназначены для организации заранее неопределенного числа объектов. При обнаружении данного сходства может возникнуть искушение посчитать объект-декоратор вырожденным случаем компоновщика, но при этом будет искажен сам смысл паттерна Декоратор.

Назначение Декоратора – добавить новые обязанности объекта без порождения **подклассов**. Этот паттерн позволяет избежать комбинаторного роста числа подклассов, если проектировщик пытается статически определить все возможные комбинации.

У Компоновщика другие задачи. Он должен так структурировать классы, чтобы различные взаимосвязанные объекты удавалось трактовать единообразно, а несколько объектов рассматривать как один. Акцент здесь делается не на оформлении, а на представлении.

Указанные цели различны, но дополняют друг друга. Поэтому Компоновщик и Декоратор часто используются совместно. Оба паттерна позволяют спроектировать систему так, что приложения можно будет создавать, просто соединяя объекты между собой, без определения новых классов. Появится некий **абстрактный класс**, одни подклассы которого – компоновщики, другие – декораторы, а третьи – реализации фундаментальных строительных блоков системы. В таком случае у компоновщиков и декораторов будет общий интерфейс. Для декоратора компоновщик является конкретным компонентом. А для компоновщика декоратор – это листовой узел.

Заместитель – еще один паттерн, структура которого напоминает Декоратор. Оба они описывают, как можно предоставить косвенный доступ к объекту, к тому же, в реализации и объектов-декораторов и заместителей хранится ссылка на другой объект, которому переадресуются запросы. Но и здесь цели различаются. Как и Декоратор, Заместитель предоставляет клиенту интерфейс, совпадающий с интерфейсом замещаемого объекта. Но в отличие от Декоратора Заместителю не нужно динамически добавлять и отбирать свойства, он не предназначен для рекурсивной композиции. Заместитель должен предоставить стандартную замену субъекту, когда прямой доступ к нему неудобен или нежелателен, например, потому что он находится на удаленной машине, хранится на диске или доступен лишь ограниченному кругу клиентов.

В паттерне Заместитель субъект определяет основную функциональность, а заместитель разрешает или запрещает доступ к ней. В Декораторе компонент обладает лишь частью функциональности, а остальное привносят один или несколько декораторов. Декоратор позволяет справиться с ситуацией, когда полную функциональность объекта нельзя определить

на этапе компиляции или это по тем или иным причинам неудобно. Такая неопределенность делает рекурсивную композицию неотъемлемой частью декоратора. Для Заместителя дело обстоит не так, ибо ему важно лишь одно отношение – между собой и своим субъектом, а данное отношение можно выразить статически.

Указанные различия существенны, поскольку в них абстрагированы решения конкретных проблем, снова и снова возникающих при объектно-ориентированном проектировании. Но это не означает, что сочетание разных паттернов невозможно. Можно представить себе заместителя-декоратора, который добавляет новую функциональность заместителю, или декоратора-заместителя, который оформляет удаленный объект.

Паттерны поведения

Паттерны поведения связаны с алгоритмами и распределением обязанностей между объектами. Речь в них идет не только о самих объектах и классах, но и о типичных способах взаимодействия. Паттерны поведения характеризуют сложный поток управления, который трудно проследить во время выполнения программы. Внимание акцентировано не на потоке управления как таковом, а на связях между объектами.

В паттернах поведения уровня класса используется **наследование** – чтобы распределить поведение между разными классами. Наиболее простым и широко распространенным из них является **Шаблонный Метод**, который представляет собой абстрактное определение алгоритма. Алгоритм здесь определяется пошагово. На каждом шаге вызывается либо примитивная, либо абстрактная операция. Алгоритм приобретает конкретную **реализацию** за счет **подклассов**, где определены абстрактные операции. Другой паттерн поведения уровня класса – **Интерпретатор**, который представляет грамматику языка в виде иерархии классов и реализует интерпретатор как последовательность операций над экземплярами этих классов.

В паттернах поведения уровня объектов используется не наследование, а композиция. Некоторые из них описывают, как с помощью **кооперации** множество равноправных объектов справляется с задачей, которая ни одному из них не под силу. Важно здесь то, как объекты получают информацию о существовании друг друга. Объекты-коллеги могут хранить ссылки друг на друга, но это увеличит степень связанности системы. При максимальной степени связанности каждому объекту пришлось бы иметь информацию обо всех остальных. Эту проблему решает паттерн **Посредник**. Посредник, находящийся между объектами-коллегами, обеспечивает косвенность ссылок, необходимую для разрывания лишних связей.

Паттерн **Цепочка Обязанностей** позволяет и дальше уменьшать степень связанности. Он дает возможность посылать запросы объекту не напрямую, а по цепочке объектов-кандидатов. Запрос может выполнить любой кандидат, если это допустимо в текущем состоянии

выполнения программы. Число кандидатов заранее не определено, а подбирать участников можно во время выполнения.

Паттерн **Наблюдатель** определяет и отвечает за зависимости между объектами. Классическим примером наблюдателя можно считать сложные документы (например, электронные таблицы) – здесь все объекты документа уведомляются о любых изменениях состояния всех объектов, от которых они зависят, и таким образом происходит автоматическое обновление всего документа.

Прочие паттерны поведения связаны с инкапсуляцией поведения в объекте и делегированием ему запросов. Паттерн **Стратегия** инкапсулирует алгоритм объекта, упрощая его спецификацию и замену. Паттерн **Команда** инкапсулирует запрос в виде объекта, который можно передавать как параметр, хранить в списке истории или использовать как-то иначе. Паттерн **Состояние** инкапсулирует состояние объекта таким образом, что при изменении состояния объект может изменять поведение. Паттерн **Посетитель** инкапсулирует поведение, которое в противном случае пришлось бы распределять между классами, а паттерн **Итератор** абстрагирует способ доступа и обхода объектов из некоторого агрегата.

Шаблонный метод (Template Method)

Назначение: определяет основу алгоритма и позволяет подклассам переопределить некоторые шаги алгоритма, не изменяя его структуру в целом.

Задача. Рассмотрим каркас приложения, в котором имеются классы `Application` и `Document`. Класс `Application` отвечает за открытие существующих документов, хранящихся во внешнем формате, например в виде файла. Объект класса `Document` представляет информацию документа после его прочтения из файла. Приложения, построенные на базе этого каркаса, могут порождать подклассы от классов `Application` и `Document`, отвечающие конкретным потребностям. Например, графический редактор определит подклассы `DrawApplication` и `DrawDocument`, а электронная таблица – подклассы `SpreadsheetApplication` и `SpreadsheetDocument`.

Очевидно, что логика алгоритма открытия документов в любом приложении достаточно однотипна:

1) проверяется, можно ли в текущем состоянии приложения открыть документ;

2) производится выбор имени файла (например, предлагается пользователю сделать это в диалоговом окне);

3) проверяется сам факт существования указанного файла, корректность его формата;

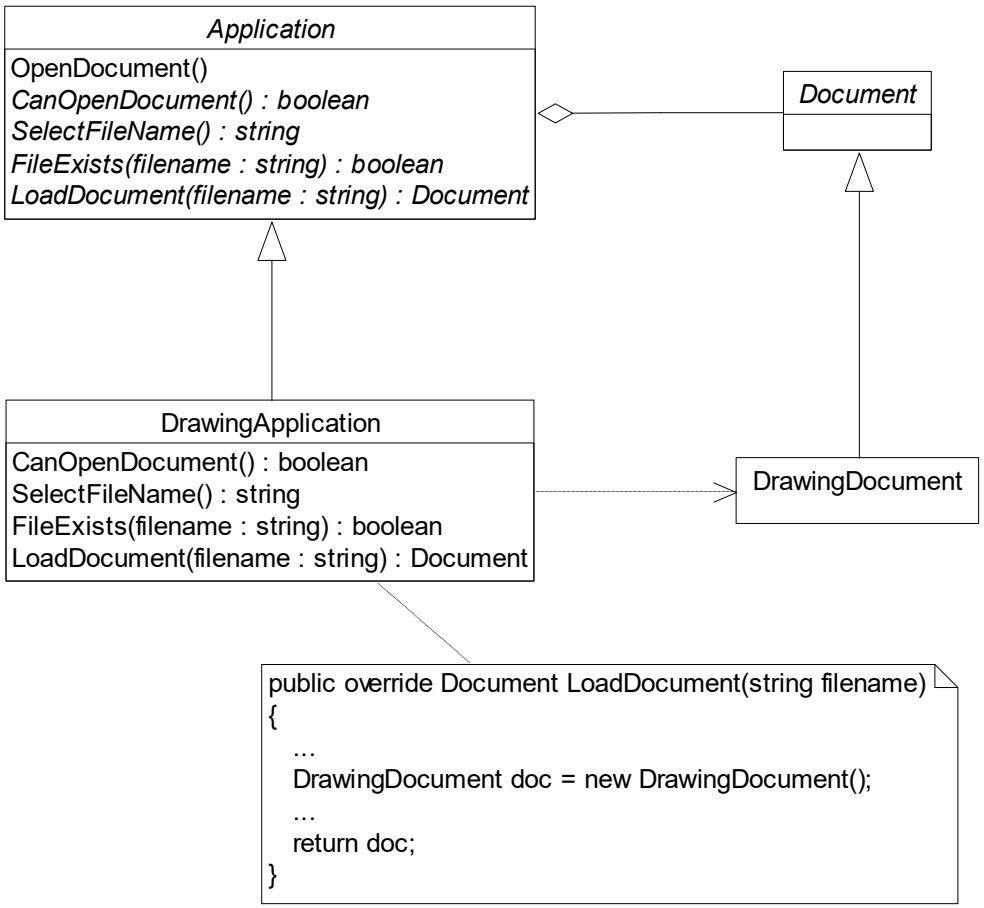
4) выполняется специализированное для каждого приложения считывание внутренней информации файла и создается объект-документ;

5) документ добавляется к списку документов приложения.

Таким образом, возникает задача исключения избыточности алгоритмов операций создания документов в классах иерархии `Application`.

Решение. Паттерн Шаблонный Метод предлагает определить в **абстрактном классе** `Application` общий алгоритм открытия и считывания документа в виде операции `OpenDocument`. Программный код такой операции может выглядеть так:

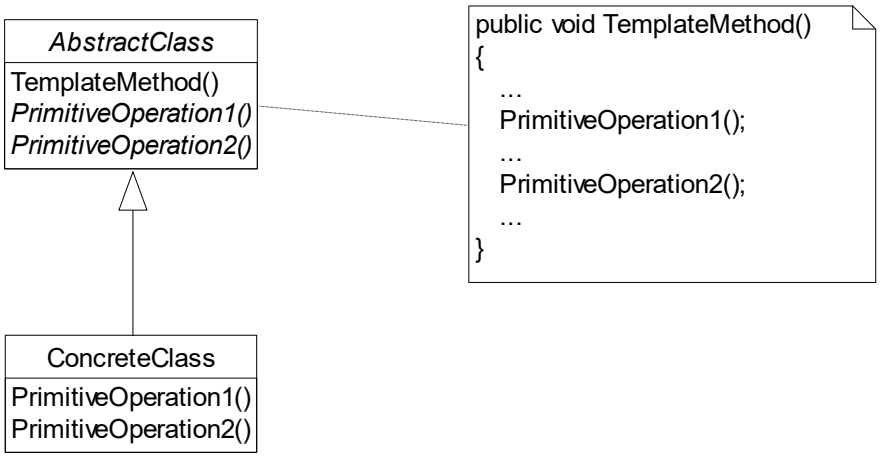
```
class Application
{
    ...
    Public void OpenDocument()
    {
        if (CanOpenDocument())
        {
            string filename = SelectFileName();
            if (FileExists(filename))
            {
                Document doc = LoadDocument(filename);
                documents.Add(doc);
            }
        }
    }
```



Операции, подобные `OpenDocument`, называют шаблонными методами. Они описывают алгоритм в терминах абстрактных операций, которые замещены в подклассах для получения нужного поведения. Подклассы класса `Application` самостоятельно определяют, как выполнить проверку возможности открытия (`CanOpenDocument`), как выбрать файл (`SelectFileName`), корректность выбранного файла (`FileExists`), а также сам процесс конструирования документа на основе информации из файла (`LoadDocument`).

Определяя некоторые шаги алгоритма с помощью абстрактных операций, шаблонный метод фиксирует их последовательность, но позволяет реализовать их в подклассах. При этом сохраняется общий ход алгоритма и конкретика некоторых шагов, которые неизменны для потомков.

Общая структура решения.



`AbstractClass` (`Application`) – **абстрактный класс**: определяет абстрактные примитивные операции, замещаемые в конкретных подклассах для реализации шагов алгоритма; реализует шаблонный метод `TemplateMethod`, определяющий скелет алгоритма. Шаблонный метод вызывает примитивные операции `PrimitiveOperation`, а также операции, определенные в классе `AbstractClass` или в других объектах.

`ConcreteClass` (`DrawApplication`) – конкретный класс: реализует примитивные операции, выполняющие шаги алгоритма способом, специфическим для данного подкласса.

`ConcreteClass` предполагает, что инвариантные шаги алгоритма будут выполнены в `AbstractClass`.

Применимость. Используйте паттерн Шаблонный Метод:

- чтобы однократно использовать инвариантные части алгоритма, оставляя реализацию изменяющегося поведения на усмотрение подклассов;
- когда нужно вычленить и локализовать в одном классе поведение, общее для всех подклассов, дабы избежать дублирования кода. Это хороший пример техники «вынесения за скобки с целью обобщения»: сначала идентифицируются различия в существующем коде, а затем они выносятся в отдельные операции. В конечном итоге

различающиеся фрагменты кода заменяются шаблонным методом, из которого вызываются новые операции;

- для управления расширениями подклассов: можно определить шаблонный метод так, что он будет вызывать операции-зацепки (hooks) в определенных точках, разрешив тем самым расширение поведения только в этих точках.

Результаты. Шаблонные методы – один из фундаментальных приемов повторного использования кода. Они особенно важны в библиотеках классов, поскольку предоставляют возможность вынести общее поведение в библиотечные классы.

Шаблонные методы приводят к инвертированной структуре кода, которую иногда называют принципом Голливуда, подразумевая часто употребляемую в этой киноимперии фразу «Не звоните нам, мы сами позвоним». В данном случае это означает, что родительский класс вызывает операции подкласса, а не как обычно – наоборот.

Особенности реализации:

- использование контроля доступа: примитивные операции, которые вызывает шаблонный метод, можно объявить защищенными членами. Тогда гарантируется, что вызывать их сможет только сам шаблонный метод. Примитивные операции, которые обязательно нужно замещать, объявляются как абстрактные. Сам шаблонный метод замещать не надо, так что его не нужно делать виртуальным;
- сокращение числа примитивных операций: важной целью при проектировании шаблонных методов является всемерное сокращение числа примитивных операций, которые должны быть замещены в подклассах. Чем больше операций нужно замещать, тем утомительнее становится программирование клиента.

Родственные паттерны.

Фабричные Методы часто вызываются из Шаблонных Методов (в примере шаблонный метод `OpenDocument` вызывал фабричный метод `LoadDocument`).

Стратегия: Шаблонные Методы применяют **наследование** для модификации части алгоритма. Стратегии используют делегирование для модификации алгоритма в целом.

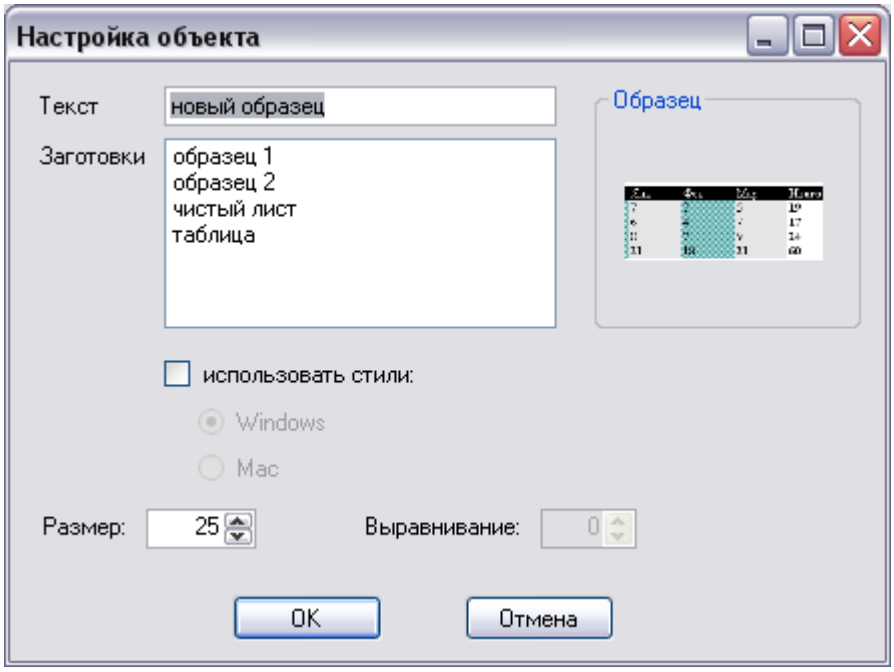
Посредник (Mediator)

Назначение: определяет объект, инкапсулирующий способ взаимодействия множества объектов, обеспечивает слабую связанность системы, избавляя объекты от необходимости явно ссылаться друг на друга и позволяя тем самым независимо изменять взаимодействия между ними.

Задача. Объектно-ориентированное проектирование способствует распределению некоторого поведения между объектами. Но при этом в получившейся структуре объектов может возникнуть много связей или (в худшем случае) каждому объекту придется иметь информацию обо всех остальных. Несмотря на то, что разбиение системы на множество объектов в общем случае повышает степень повторного использования, изобилие взаимосвязей приводит к обратному эффекту. Если взаимосвязей слишком много, тогда система подобна

монолиту, и маловероятно, что объект сможет работать без поддержки других объектов. Более того, существенно изменить поведение системы практически невозможно, поскольку оно распределено между многими объектами. Если вы предпримете подобную попытку, то для настройки поведения системы вам придется определять множество подклассов.

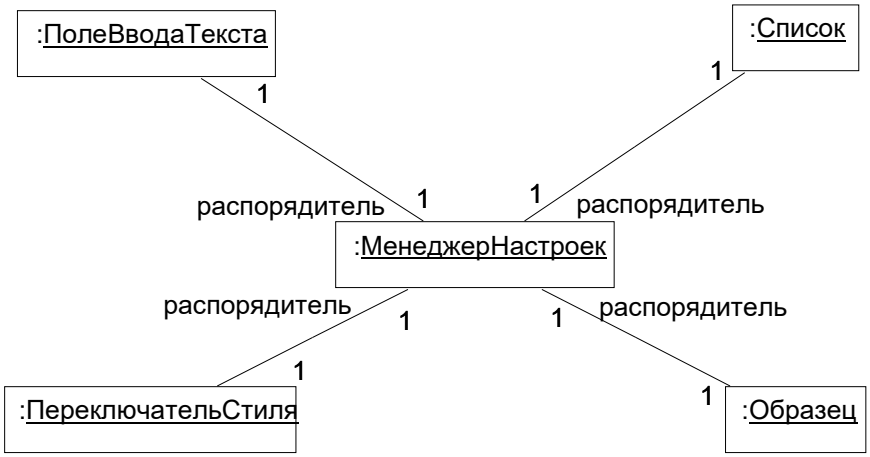
Рассмотрим реализацию диалоговых окон в графическом интерфейсе пользователя. Здесь располагается ряд элементов управления: кнопки, меню, поля ввода и т.д., как показано на рисунке.



Часто между разными элементами управления в диалоговом окне существуют зависимости. Например, если одно из полей ввода пустое, то определенная кнопка недоступна. При выборе из списка может измениться содержимое поля ввода. И наоборот, ввод текста в некоторое поле может автоматически привести к выбору одного или нескольких элементов списка. Если в поле ввода присутствует какой-то текст, то могут быть активизированы кнопки, позволяющие произвести определенное действие над этим текстом, например изменить либо удалить его. В разных диалоговых окнах зависимости между элементами управления могут быть различными. Поэтому, несмотря на то, что во всех окнах встречаются однотипные элементы, просто взять и повторно использовать их готовые классы не удастся – придется производить настройку с целью учета зависимостей. Индивидуальная настройка каждого элемента управления – утомительное занятие, ибо участвующих классов слишком много.

Решение. Всех этих проблем в большой степени можно избежать, если инкапсулировать коллективное поведение в отдельном объекте-посреднике. Посредник отвечает за координацию взаимодействий между группой объектов. Он избавляет входящие в группу объекты от необходимости явно ссылаться друг на друга. Все объекты располагают информацией только о посреднике, поэтому количество взаимосвязей сокращается.

Так, класс МенеджерНастроек может служить посредником между элементами управления в диалоговом окне. Объект этого класса знает обо всех элементах в окне и координирует взаимодействие между ними, то есть выполняет функции центра коммуникаций.



На диаграмме взаимодействий показано, как объекты кооперируются друг с другом, реагируя на изменение выбранного элемента списка:

Последовательность событий, в результате которых информация о выбранном элементе списка передается в поле ввода, следующая:

1) список информирует распорядителя о происшедших в нем изменениях;

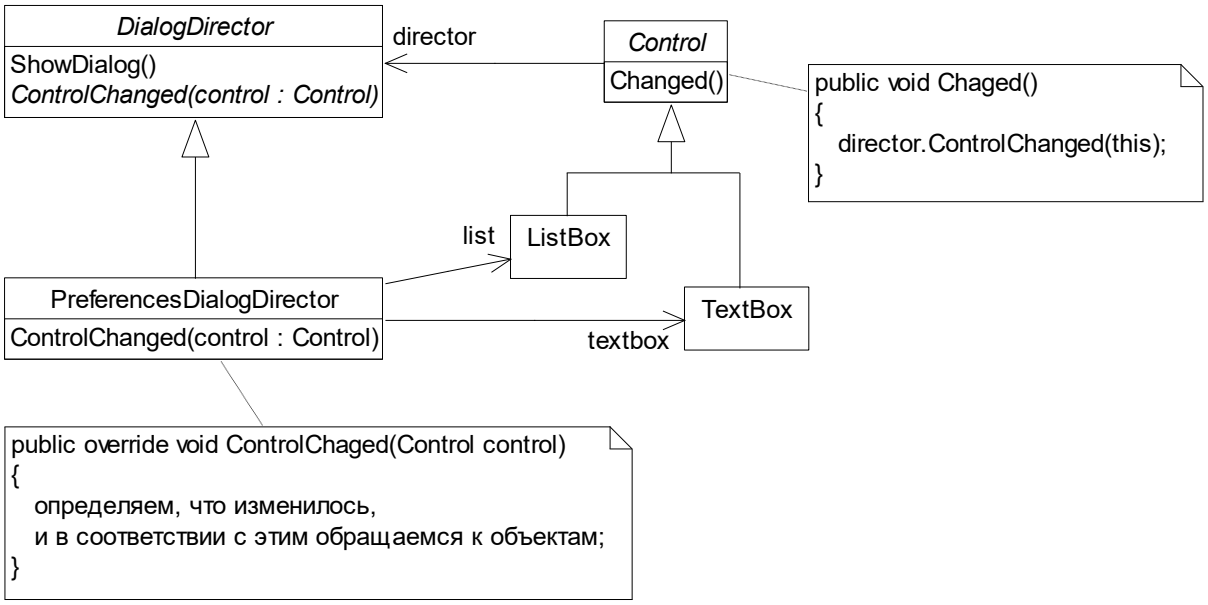
2) распорядитель получает от списка данные о выбранном элементе;

3) распорядитель извлекает из этих данных текст и передает полю ввода;

4) распорядитель формирует на основе данных эскиз для образца и передает ему этот эскиз;

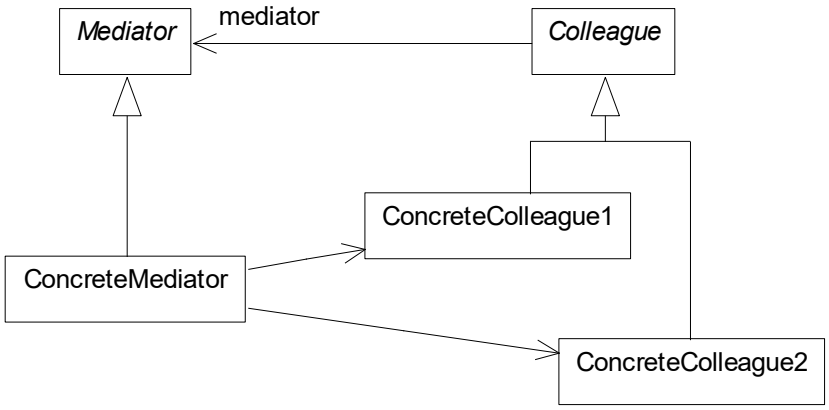
5) распорядитель может выполнить и другие действия, связанные, например, с изменением доступности кнопок и других элементов управления.

Обратите внимание на то, как распорядитель осуществляет посредничество между списком и полем ввода. Элементы управления общаются друг с другом не напрямую, а через распорядителя. Им вообще не нужно владеть информацией друг о друге, они осведомлены лишь о существовании распорядителя. Поскольку поведение локализовано в одном классе, то его несложно модифицировать или сделать совершенно другим путем расширения или замены этого класса. Абстракцию `МенеджерНастроек` можно было бы интегрировать в библиотеку классов так, как показано на диаграмме (здесь ей присвоено имя `PreferencesDialogDirector`):

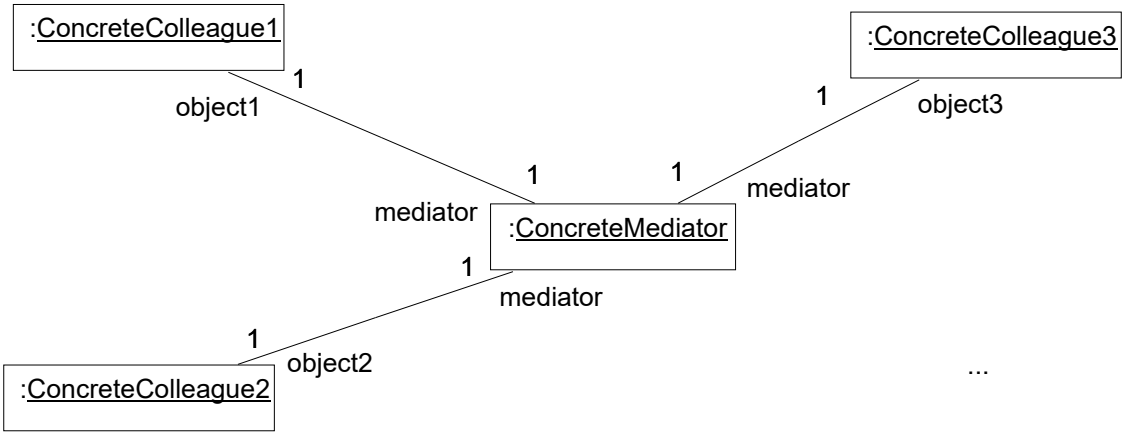


`DialogDirector` – это **абстрактный класс**, который определяет поведение диалогового окна в целом. Клиенты вызывают его операцию `ShowDialog` для отображения окна на экране. Для оповещения распорядителя об изменениях элементы управления вызывают его операцию `ControlChanged`, передавая себя в качестве параметра. Подклассы `DialogDirector` замещают операцию `ControlChanged` для обработки этих вызовов.

Общая структура решения.



Типичная структура объектов:



`Mediator` (`DialogDirector`) – посредник: определяет **интерфейс** для обмена информацией с объектами `Colleague`.

`ConcreteMediator` (`PreferencesDialogDirector`) – конкретный посредник: реализует кооперативное поведение, координируя действия объектов `Colleague`; владеет информацией о конкретных коллегах.

`Colleague` (`Control`) – класс-коллега: объявляет интерфейс, характерный для сотрудничающих объектов, поддерживает указатель на объект-посредник.

`ConcreteColleague` (`ListBox`, `TextBox`) – конкретные объекты-коллеги: каждый из них знает о своем объекте `Mediator`, благодаря указателю, поддерживаемому предком; все коллеги обмениваются информацией только с Посредником, не обращаясь напрямую друг к другу.

Коллеги посылают запросы посреднику и получают запросы от него. Посредник реализует кооперативное поведение путем переадресации каждого запроса подходящему коллеге или нескольким коллегам.

Применимость. Используйте паттерн Посредник, когда:

- имеются объекты, связи между которыми сложны и четко определены, получающиеся при этом взаимозависимости не структурированы и трудны для понимания;
- нельзя повторно использовать объект, поскольку он обменивается информацией со многими другими объектами;
- поведение, распределенное между несколькими классами, должно поддаваться настройке без порождения множества подклассов.

Результаты. У паттерна Посредник есть следующие достоинства и недостатки:

- уменьшает число порождаемых подклассов: посредник локализует поведение, которое в противном случае пришлось бы распределять между несколькими объектами. Для изменения поведения нужно породить подклассы только от класса посредника `Mediator`, классы коллег можно использовать повторно без каких бы то ни было изменений;
- устраняет связанность между коллегами: посредник обеспечивает слабую связанность коллег. Изменять классы `Colleague` и `Mediator` можно независимо друг от друга;
- упрощает протоколы взаимодействия объектов: посредник заменяет дисциплину взаимодействия «все со всеми» дисциплиной «один со всеми / все с одним», то есть один посредник взаимодействует со всеми коллегами. Отношения вида «один ко многим» проще для понимания, сопровождения и расширения, чем «многие ко многим»;
- абстрагирует способ кооперирования объектов: выделение механизма посредничества в отдельную концепцию и инкапсуляция ее в одном объекте позволяет сосредоточиться именно на взаимодействии объектов, а не на их индивидуальном поведении. Это дает возможность прояснить имеющиеся в системе взаимодействия;
- централизует управление: паттерн Посредник переносит сложность взаимодействия в класс-посредник. Поскольку посредник инкапсулирует протоколы, то он может быть сложнее отдельных коллег. В результате сам посредник становится классом с широким кругом обязанностей, который трудно сопровождать.

Особенности реализации:

- избавление от **абстрактного класса** `Mediator`: если коллеги работают только с одним посредником, то нет необходимости определять **абстрактный класс** `Mediator`. Обеспечиваемая классом `Mediator` абстракция позволяет коллегам работать с разными подклассами класса `Mediator` и наоборот;

- обмен информацией между коллегами и посредником: коллеги должны обмениваться информацией со своим посредником только тогда, когда возникает представляющее интерес событие. Одним из подходов к реализации посредника является применение паттерна **Наблюдатель**. Тогда классы коллег действуют как субъекты, посылающие извещения посреднику о любом изменении своего состояния. Посредник реагирует на них, сообщая об этом другим коллегам. Другой подход: в классе `Mediator` определяется специализированный интерфейс уведомления, который позволяет коллегам обмениваться информацией более свободно.

Родственные паттерны.

Фасад отличается от Посредника тем, что абстрагирует некоторую **подсистему** объектов для предоставления более удобного интерфейса. Его протокол однонаправленный, то есть объект-фасад направляет запросы классам подсистемы, но не наоборот. Посредник же обеспечивает совместное поведение, которое объекты-коллеги не могут или не хотят реализовывать, и его протокол двунаправленный.

Коллеги могут обмениваться информацией с посредником посредством паттерна Наблюдатель.

Команда (Command)

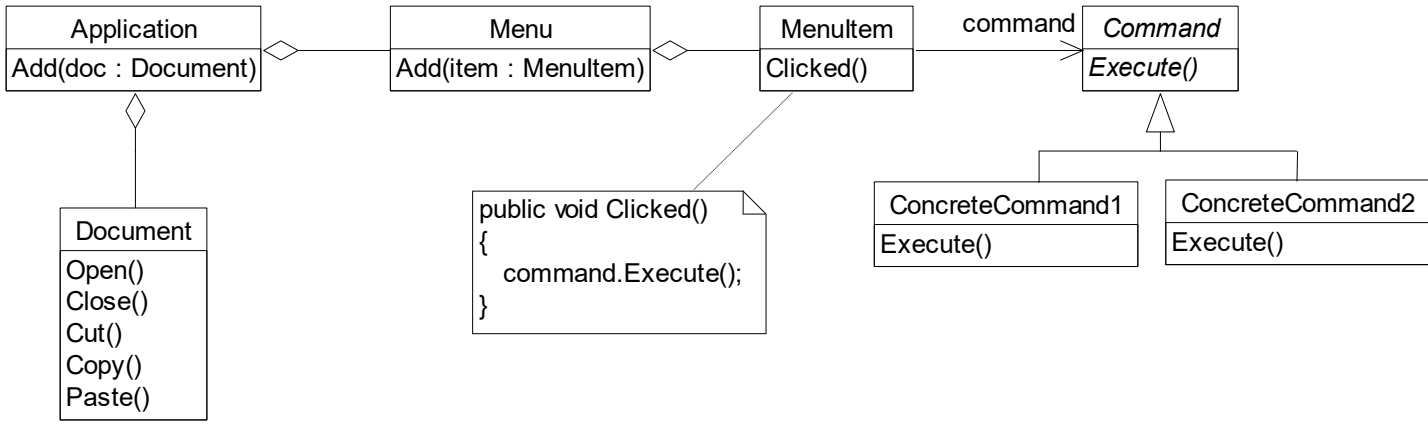
Другие названия: Action (Действие), Transaction (Транзакция).

Назначение: инкапсулирует запрос в объекты, позволяя передавать их в качестве параметров для обработки, ставить в очередь, протоколировать, поддерживать отмену операций и т.п.

Задача. Иногда необходимо посылать объектам запросы, ничего не зная о том, выполнение какой операции запрошено и кто является получателем. Например, в библиотеках для построения пользовательских интерфейсов встречаются такие объекты, как кнопки и меню, которые посылают запрос в ответ на действие пользователя. Но в саму библиотеку не заложена возможность обрабатывать этот запрос, так как только приложение, использующее ее, располагает информацией о том, что следует сделать. Проектировщик библиотеки не владеет никакой информацией о получателе запроса и о том, какие операции тот должен выполнить.

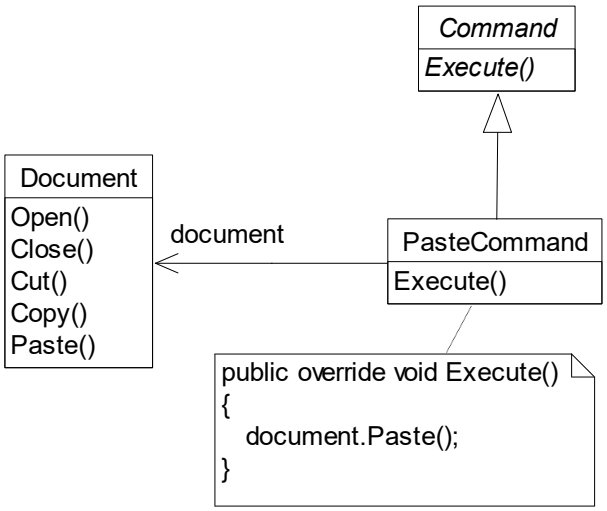
Решение. Паттерн Команда предлагает отправлять запросы неизвестным объектам приложения, преобразовав сам запрос в объект. Этот объект можно хранить и передавать, как и любой другой. В основе данного паттерна лежит **абстрактный класс** `Command`, в котором объявлен **интерфейс** для выполнения операций. В простейшей своей форме этот интерфейс состоит из одной абстрактной операции `Execute`. Конкретные подклассы `Command` определяют пару «получатель-действие», сохраняя получателя в переменной экземпляра, и реализуют операцию `Execute` так, чтобы она посылала запрос. У получателя же есть информация, необходимая для выполнения запроса.

С помощью объектов `Command` легко реализуется меню. Каждый пункт меню – это экземпляр класса `MenuItem`. Сами меню и все их пункты создает класс `Application` наряду со всеми остальными элементами пользовательского интерфейса. Класс `Application` отслеживает также открытые пользователем документы. Приложение конфигурирует каждый объект `MenuItem` экземпляром конкретного подкласса `Command`. Когда пользователь выбирает некоторый пункт меню, ассоциированный с ним объект `MenuItem` вызывает операцию `Execute` для своего объекта-команды, а `Execute` выполняет нужные действия.

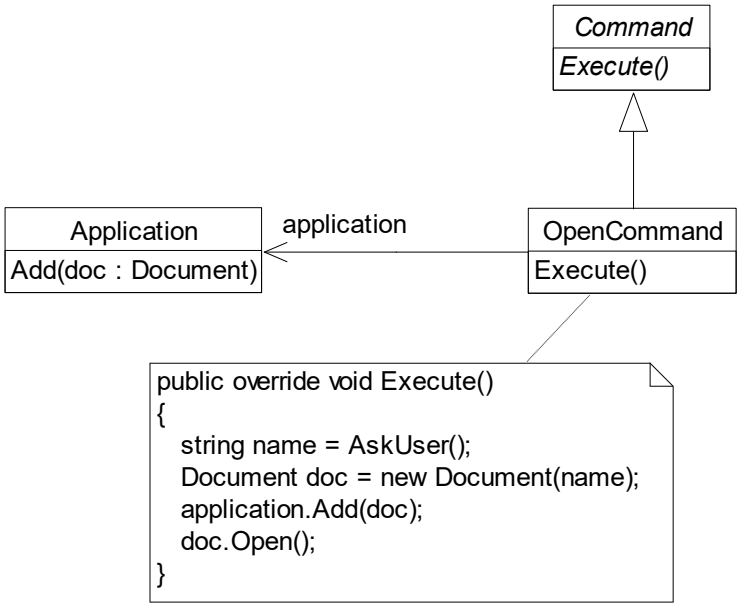


Объекты `MenuItem` не имеют информации о том, какой подкласс класса `Command` они используют. Подклассы `Command` хранят информацию о получателе запроса и вызывают одну или несколько операций этого получателя.

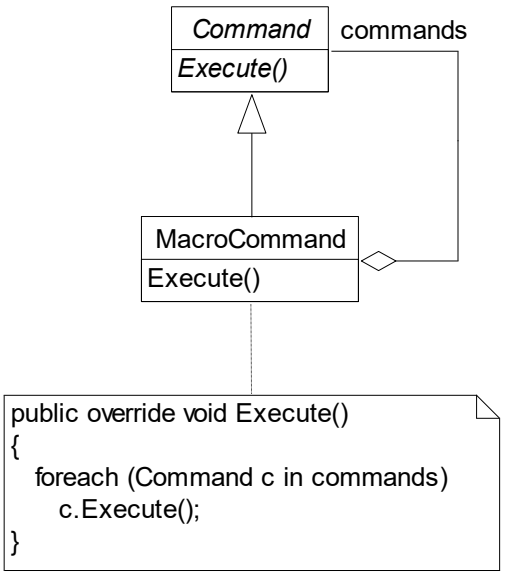
Например, подкласс `PasteCommand` может поддерживать вставку текста из буфера обмена в документ. Получателем для `PasteCommand` является `Document`, который был передан при создании объекта. Операция `Execute` вызывает операцию `Paste` документа-получателя.



Для подкласса `OpenCornmand` операция `Execute` ведет себя по-другому: она запрашивает у пользователя имя документа, создает соответствующий объект `Document`, извещает о новом документе приложение-получатель и открывает этот документ.



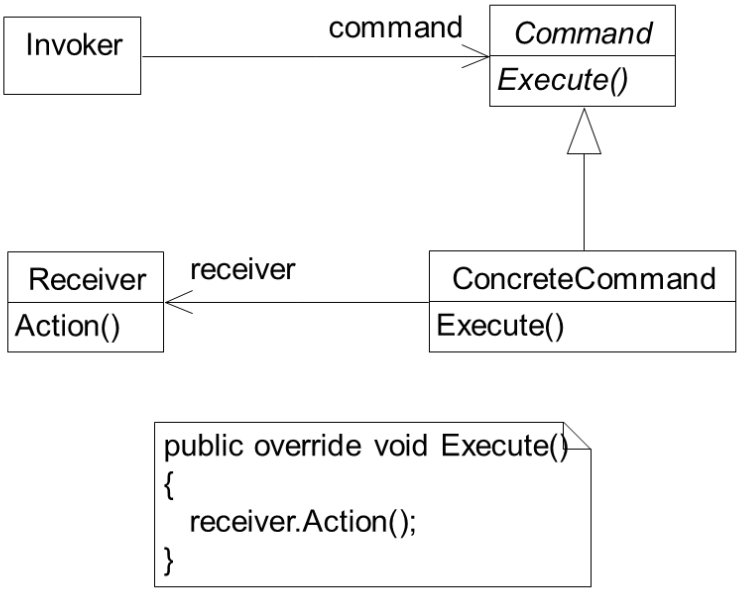
Иногда объект `MenuItem` должен выполнить последовательность команд. Например, пункт меню для центрирования страницы стандартного размера можно было бы сконструировать сразу из двух объектов: `CenterDocumentCommand` и `NormalsizeCommand`. Поскольку такое комбинирование команд – явление обычное, то мы можем определить класс `MacroCommand`, позволяющий объекту `MenuItem` выполнять произвольное число команд. `MacroCommand` – это конкретный подкласс класса `Command`, который просто выполняет последовательность команд. У него нет явного получателя, поскольку для каждой команды определен свой собственный исполнитель.



В каждом из приведенных примеров паттерн Команда отделяет объект, инициирующий операцию, от объекта, который может ее выполнить. Это позволяет добиться высокой гибкости при проектировании пользовательского интерфейса. Пункт меню и кнопка одновременно могут

быть ассоциированы в приложении с некоторой функцией, для этого достаточно приписать обоим элементам один и тот же экземпляр конкретного подкласса класса Command. Мы можем динамически подменять команды, что очень полезно для реализации контекстно-зависимых меню. Можно также поддерживать сценарии, если компоновать простые команды в более сложные. Все это выполнимо, потому что объект, инициирующий запрос, должен располагать информацией лишь о том, как его отправить, а не о том, как его выполнить.

Общая структура решения.



Command – команда: объявляет **интерфейс** для выполнения операции.

ConcreteCommand (PasteCommand, OpenCommand) – конкретная команда: определяет связь между объектом-получателем Receiver и действием; реализует операцию Execute путем вызова соответствующих операций объекта Receiver.

Invoker (MenuItem) – инициатор: обращается к команде для выполнения запроса.

Receiver (Document, Application) – получатель: располагает информацией о способах выполнения операций, необходимых для удовлетворения запроса.

Приложение создает объект ConcreteCommand и устанавливает для него получателя, а также передает указатель на эту команду соответствующему инициатору. Инициатор Invoker отправляет запрос, вызывая операцию команды Execute. Если требуется, например, поддерживать отмену выполненных действий, то ConcreteCommand перед вызовом Execute сохраняет информацию о состоянии, достаточную для выполнения отката.

Применимость. Используйте паттерн Команда, когда:

- хотите параметризовать объекты выполняемым действием, как в случае с пунктами меню `MenuItem`;
- нужно определять, ставить в очередь и выполнять запросы в разное время. Время жизни объекта `Command` необязательно должно зависеть от времени жизни исходного запроса. Если получателя запроса удается реализовать так, чтобы он не зависел от адресного пространства, то объект-команду можно передать другому процессу, который займется его выполнением;
- хотите поддерживать отмену операций: операция `Execute` объекта `Command` может сохранить состояние, необходимое для отката действий, выполненных командой. В этом случае в интерфейсе класса `Command` должна быть дополнительная операция `Unexecute`, которая отменяет действия, выполненные предшествующим обращением к `Execute`. Выполненные команды хранятся в списке истории. Для реализации произвольного числа уровней отмены и повтора команд нужно обходить этот список соответственно в обратном и прямом направлениях, вызывая при посещении каждого элемента команду `Unexecute` или `Execute`;
- хотите поддержать протоколирование изменений, чтобы их можно было выполнить повторно после аварийной остановки системы. Дополнив интерфейс класса `Command` операциями сохранения и загрузки, вы сможете вести протокол изменений во внешней памяти. Для восстановления после сбоя нужно будет загрузить сохраненные команды с диска и повторно выполнить их с помощью операции `Execute`;
- необходимо структурировать систему на основе высокоуровневых операций, построенных из примитивных. Такая структура типична для информационных систем, поддерживающих транзакции. Транзакция инкапсулирует набор изменений данных. Паттерн Команда позволяет моделировать транзакции. У всех команд есть общий интерфейс, что дает возможность работать одинаково с любыми транзакциями. С помощью этого паттерна можно легко добавлять в систему новые виды транзакций.

Результаты:

- команда разрывает связь между объектом, инициирующим операцию, и объектом, имеющим информацию о том, как ее выполнить;

- команды – это самые настоящие объекты, допускается манипулировать ими и расширять их точно так же, как в случае с любыми другими объектами;

- из простых команд можно собирать составные, для этого можно применить паттерн **Компоновщик**;

- добавлять новые команды легко, поскольку никакие существующие классы изменять не нужно.

Особенности реализации. При реализации паттерна Команда следует обратить внимание на следующие аспекты:

- насколько «умной» должна быть команда: у команды может быть широкий круг обязанностей. На одном полюсе стоит простое определение связи между получателем и действиями, которые нужно выполнить для удовлетворения запроса. На другом – реализация всего самостоятельно, без обращения за помощью к получателю. Последний вариант полезен, когда вы хотите определить команды, не зависящие от существующих классов, когда подходящего получателя не существует или когда получатель команде точно не известен. Например, команда, создающая новое окно

приложения, может не понимать, что именно она создает, а трактовать окно, как любой другой объект. Где-то посередине между двумя крайностями находятся команды, обладающие достаточной информацией для динамического обнаружения своего получателя;

- поддержка отмены и повтора операций: команды могут поддерживать отмену и повтор операций, если имеется возможность отменить результаты выполнения. В классе `ConcreteCommand` может сохраняться необходимая для этого дополнительная информация, в том числе: объект-получатель `Receiver`, который выполняет операции в ответ на запрос, аргументы операции, выполненной получателем, исходные значения различных атрибутов получателя, которые могли измениться в результате обработки запроса. Получатель должен предоставить операции, позволяющие команде вернуться в исходное состояние. Для поддержки всего одного уровня отмены приложению достаточно сохранять только последнюю выполненную команду. Если же нужны многоуровневые отмена и повтор операций, то придется вести список истории выполненных команд. Проход по списку в обратном направлении и откат результатов всех встретившихся по пути команд отменяет их действие; проход в прямом направлении и выполнение встретившихся команд приводит к повтору действий. Команду, допускающую отмену, возможно, придется скопировать перед помещением в список истории. Дело в том, что объект команды, использованный для доставки запроса, скажем, от пункта меню `MenuItem`, позже мог быть использован для других запросов. Поэтому копирование необходимо, чтобы определить разные вызовы одной и той же команды, если ее состояние при любом вызове может изменяться. Например, команда `DeleteCommand`, которая удаляет выбранные объекты, при каждом вызове должна сохранять разные наборы объектов. Поэтому объект `DeleteCommand` необходимо скопировать после выполнения, а копию поместить в список истории. Если в результате выполнения состояние команды никогда не изменяется, то копировать не нужно – в список достаточно поместить лишь ссылку на команду Команды, которые обязательно нужно копировать перед помещением в список истории, ведут себя подобно прототипам;
- как избежать накопления ошибок в процессе отмены: при обеспечении надежного, сохраняющего семантику механизма отмены и повтора может возникнуть проблема накопления ошибок, в результате чего состояние приложения оказывается отличным от первоначального. Поэтому порой необходимо сохранять в команде больше информации, дабы гарантировать, что объекты будут целиком восстановлены. Чтобы предоставить команде доступ к этой информации, не раскрывая внутреннего устройства объектов, можно воспользоваться паттерном **Хранитель**.

Родственные паттерны.

Паттерн **Компоновщик** можно использовать для реализации макрокоманд.

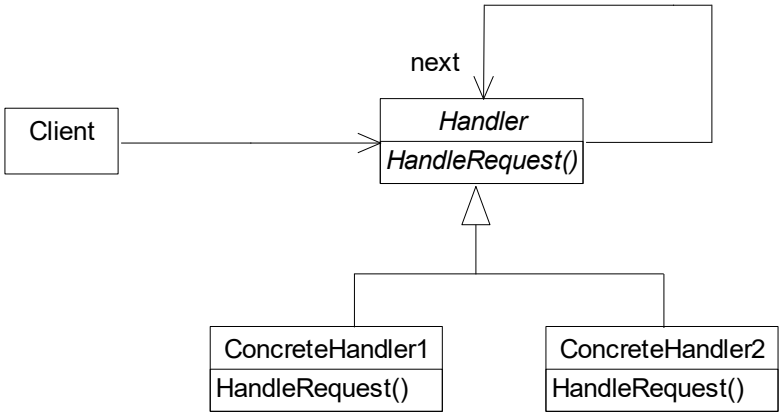
Паттерн Хранитель может помочь сохранять состояние команды, необходимое для отмены ее действия.

Команда, которую нужно копировать перед помещением в список истории, ведет себя, как **Прототип**.

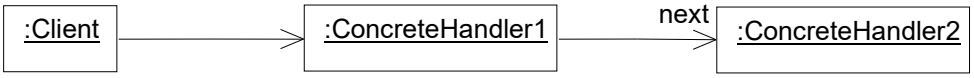
Цепочка обязанностей (Chain Of Responsibility)

Назначение: позволяет избежать привязки отправителя запроса к его получателю, давая шанс обработать запрос нескольким объектам. Связывает объекты-получатели в цепочку и передает запрос вдоль этой цепочки, пока его не обработают.

Общая структура решения.



Типичная структура объектов:



`Handler` – обработчик: определяет **интерфейс** для обработки запросов (`HandleRequest`); может реализовывать связь с преемником (указатель `next`).

`ConcreteHandler` – конкретные обработчики: обрабатывают запрос, за который отвечают; имеют доступ к своему преемнику. Если `ConcreteHandler` способен обработать запрос, то делает это, если не может – то направляет его своему преемнику.

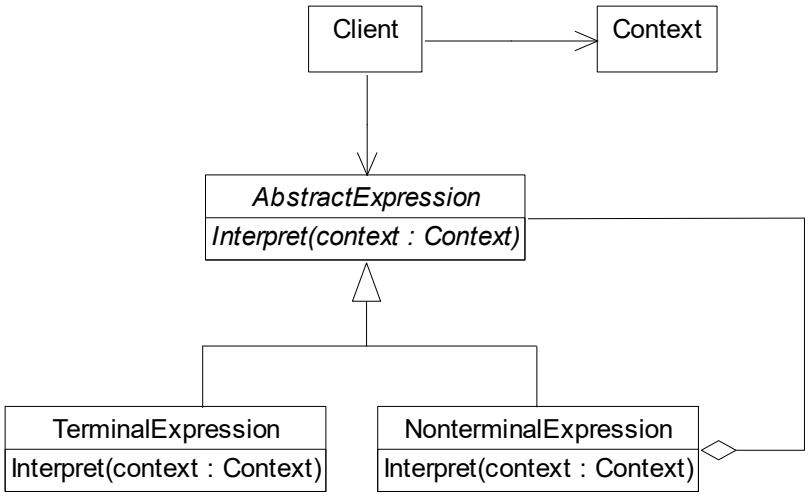
`Client` – клиент: отправляет запрос некоторому объекту `ConcreteHandler` в цепочке.

Когда клиент инициирует запрос, этот запрос продвигается по цепочке, пока некоторый объект `ConcreteHandler` не возьмет на себя ответственность за его обработку.

Интерпретатор (Interpreter)

Назначение: для заданного языка определяет представление его грамматики, а также интерпретатор предложений этого языка.

Общая структура решения.



`AbstractExpression` – абстрактное выражение: объявляет абстрактную операцию Interpret, общую для всех узлов в абстрактном синтаксическом дереве.

`TerminalExpression` – терминальное выражение: реализует операцию `Interpret` для терминальных символов грамматики; необходим отдельный экземпляр этого класса для каждого терминального символа в предложении.

`NonterminalExpression` – нетерминальное выражение: по одному такому классу требуется для каждого грамматического правила $R::=R_1R_2...R_n$; хранит переменные экземпляра типа `AbstractExpression` для каждого символа от R_1 до R_n; реализует операцию `Interpret` для нетерминальных символов грамматики. Эта операция вызывает операцию `Interpret` для переменных, представляющих R_1, ..., R_n.

`Context` – контекст: содержит информацию, глобальную по отношению к интерпретатору.

`Client` – клиент: строит или получает в готовом виде абстрактное синтаксическое дерево, представляющее отдельное предложение на языке с данной грамматикой.

Клиент строит или получает в готовом виде предложение в виде абстрактного синтаксического дерева, в узлах которого находятся объекты классов `NonterminalExpression` и `TerminalExpression`. Затем клиент инициализирует контекст и вызывает операцию `Interpret`.

В каждом узле вида `NonterminalExpression` через операцию `Interpret` этого узла вызывается операция `Interpret` для каждого подвыражения. Для класса `TerminalExpression` операция `Interpret` определяет базу рекурсии.

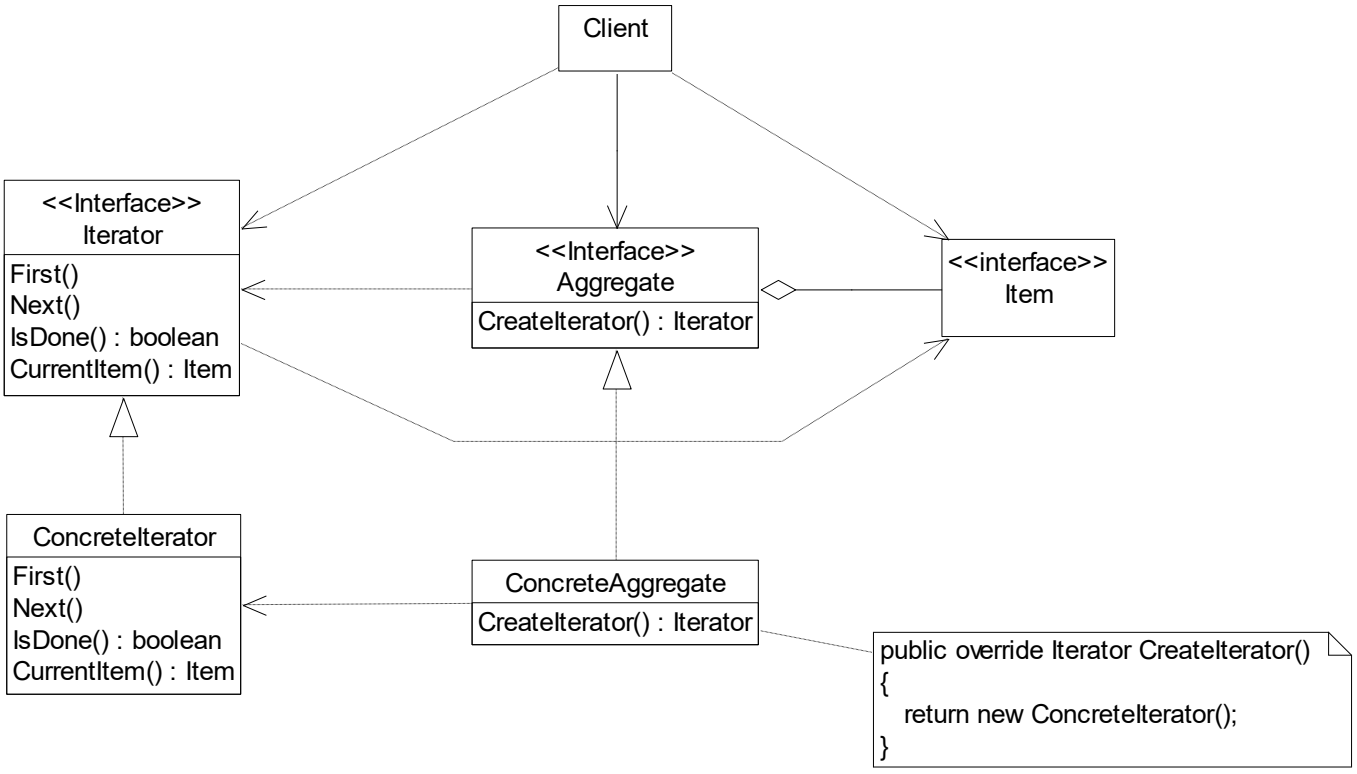
Операции `Interpret` в каждом узле используют контекст для сохранения и доступа к состоянию интерпретатора.

Итератор (Iterator)

Другое название: Cursor (Курсор).

Назначение: предоставляет способ последовательного доступа ко всем элементам составного объекта, не раскрывая его внутреннего устройства.

Общая структура решения.



`Iterator` – итератор: определяет **интерфейс** для доступа и обхода элементов.

`ConcreteIterator` – конкретный итератор: реализует интерфейс `Iterator`; следит за текущей позицией при обходе агрегата.

`Aggregate` – агрегат: определяет интерфейс для создания объекта-итератора.

`ConcreteAggregate` – конкретный агрегат: реализует интерфейс создания итератора и возвращает экземпляр подходящего класса `ConcreteIterator`.

`Item` – элемент: интерфейс внутренних объектов агрегата (требуется, если на уровне `Iterator` объявлена операция `CurrentItem`).

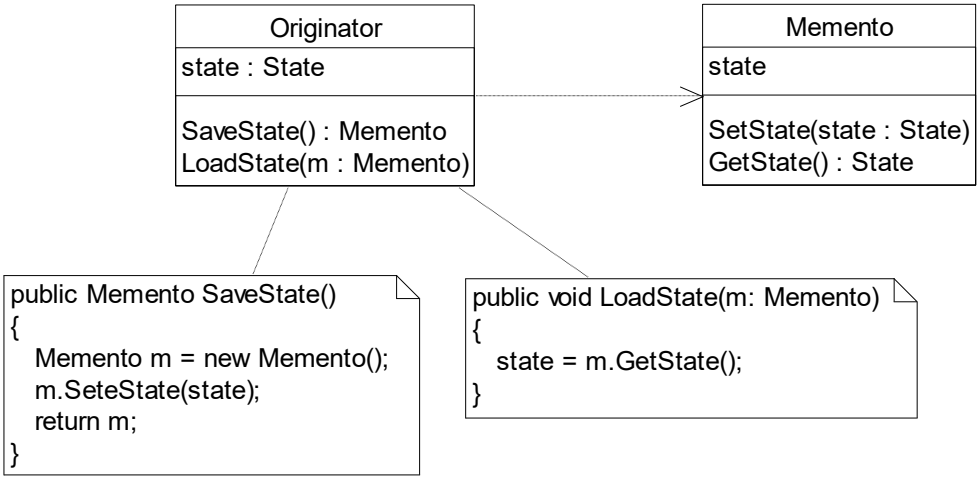
`ConcreteIterator` отслеживает текущий объект в агрегате и может определить следующий элемент, а также завершение обхода (`IsDone`).

Хранитель (Memento)

Другое название: Token (Лексема).

Назначение: фиксирует и выносит за пределы объекта его внутреннее состояние так, чтобы позднее можно было восстановить в нем объект, не нарушая при этом принципа инкапсуляции.

Общая структура решения.



`Memento` – хранитель: сохраняет внутреннее состояние объекта `Originator`, объем сохраняемой информации при этом определяется потребностями хозяина; запрещает доступ к состоянию всем другим объектам, кроме хозяина. Внутренняя структура состояния в хранителе может отличаться от структуры состояния хозяина, главное – она должна однозначно быть воспроизведена при вызове операции `GetState`. Хранители пассивны. Только хозяин, создавший хранитель, имеет доступ к информации о состоянии.

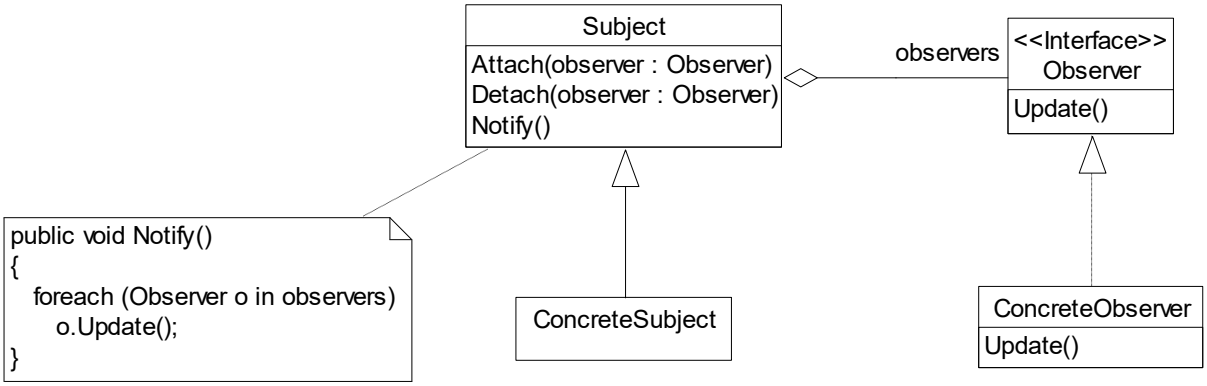
`Originator` – хозяин: создает хранителя, содержащего снимок текущего внутреннего состояния; использует его в будущем для восстановления этого состояния. Между сохранением и восстановлением состояния объект-хранитель может содержаться в другом объекте.

Наблюдатель (Observer)

Другое название: Publish-Subscribe (Издатель-Подписчик).

Назначение: определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом и автоматически обновляются.

Общая структура решения.



`Subject` – субъект: располагает информацией о своих наблюдателях – за субъектом может следить любое число наблюдателей; предоставляет **интерфейс** для присоединения и отделения наблюдателей; реализует операцию оповещения всех наблюдателей об изменении своего состояния.

`Observer` – наблюдатель: определяет интерфейс обновления для объектов, которые должны быть уведомлены об изменении субъекта.

`ConcreteSubject` – конкретный субъект: в случае изменения состояния вызывает операцию `Notify`, оповещая всех своих наблюдателей об изменении состояния.

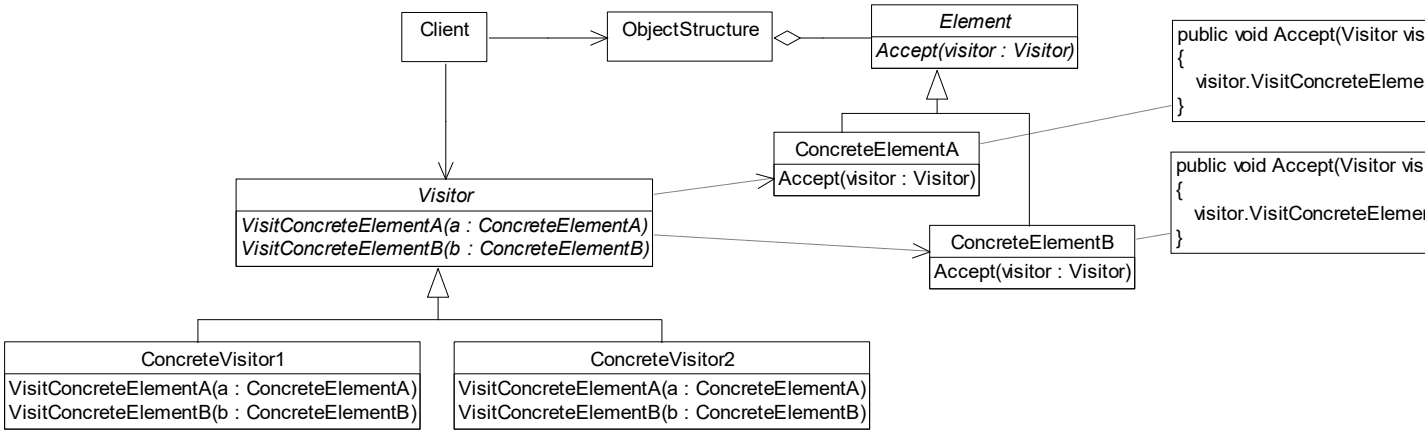
`ConcreteObserver` – конкретный наблюдатель: реализует интерфейс обновления, определенный в `Observer`, производя свое обновление при получении оповещения.

Объект `ConcreteSubject` уведомляет своих наблюдателей о любом изменении, которое могло бы привести к рассогласованности состояний наблюдателя и субъекта. После получения от конкретного субъекта уведомления об изменении объект `ConcreteObserver` может запросить у субъекта дополнительную информацию, которую использует для того, чтобы оказаться в состоянии, согласованном с состоянием субъекта. Это можно сделать, если наблюдатель располагает указателем на субъект, а субъект реализует некий интерфейс для передачи необходимой информации.

Посетитель (Visitor)

Назначение: описывает операцию, выполняемую с каждым объектом из некоторой структуры, при этом определение новой операции не изменяет классы этих объектов.

Общая структура решения.



`Visitor` – посетитель: объявляет операцию `Visit` для каждого класса `ConcreteElement` в структуре объектов. **Сигнатура** этой операции идентифицирует класс, который посылает посетителю запрос `Visit`. Это позволяет посетителю определить, элемент какого конкретного класса он посещает. Владея такой информацией, посетитель может обращаться к элементу напрямую через его **интерфейс**.

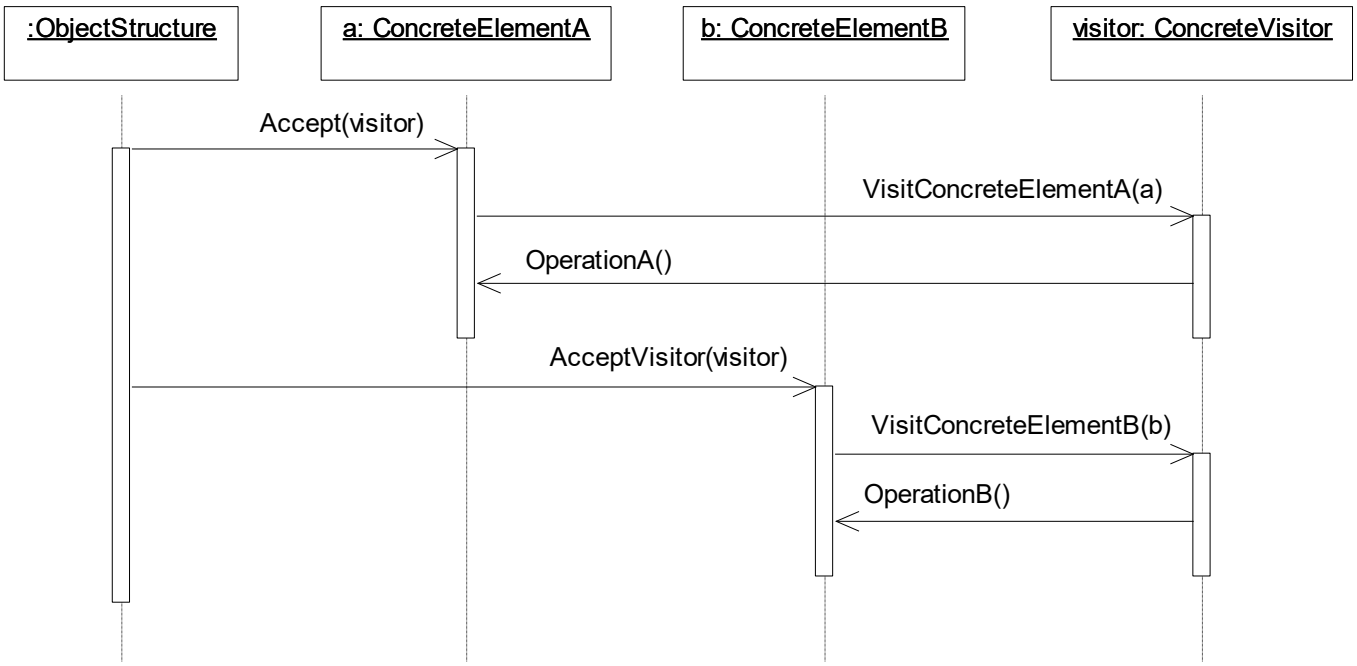
`ConcreteVisitor` – конкретные посетители: реализуют все операции, объявленные в классе `Visitor`. Каждая операция реализует фрагмент алгоритма, определенного для класса соответствующего объекта в структуре. Класс `ConcreteVisitor` предоставляет контекст для этого алгоритма и сохраняет его локальное состояние. Часто в этом состоянии аккумулируются результаты, полученные в процессе обхода структуры.

`Element` – элемент: определяет операцию `Accept`, которая принимает посетителя в качестве аргумента.

`ConcreteElement` – конкретные элементы: реализуют операцию `Accept`, принимающую посетителя как аргумент.

`ObjectStructure` – структура объектов: может перечислить свои элементы и предоставить посетителю высокоуровневый интерфейс для посещения своих элементов. Может быть как составным объектом (паттерн **Компоновщик** *(ссылка на кадр 43)*), так и коллекцией, например списком или множеством.

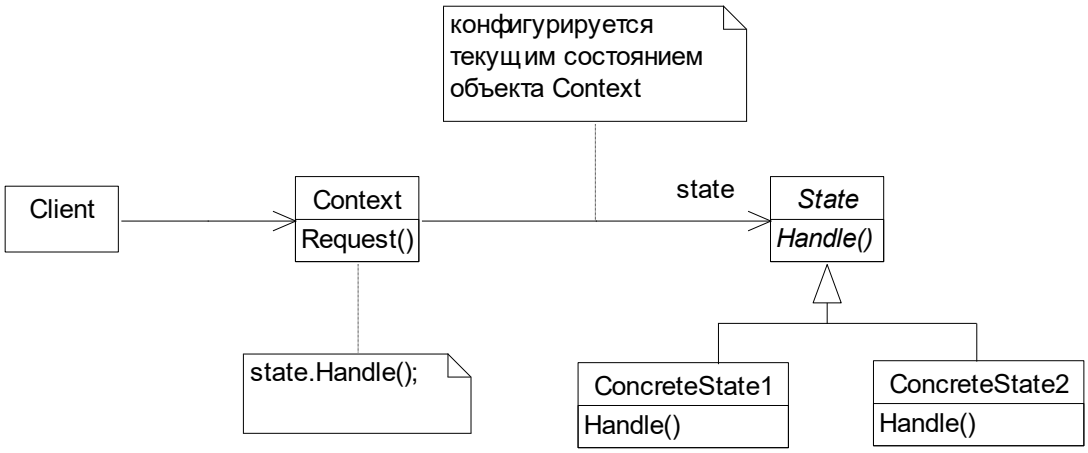
Клиент, использующий паттерн Посетитель, должен создать объект класса `ConcreteVisitor`, а затем обойти всю структуру, посетив каждый ее элемент. При посещении конкретного элемента этот элемент вызывает операцию посетителя, соответствующую своему классу. Элемент передает этой операции себя в качестве аргумента, чтобы посетитель мог при необходимости получить доступ к его состоянию.



Состояние (State)

Назначение: позволяет объекту варьировать свое поведение в зависимости от внутреннего состояния, извне создается впечатление, что изменился класс объекта.

Общая структура решения.



`Context` – контекст: определяет **интерфейс**, представляющий интерес для клиентов; хранит указатель на экземпляр подкласса `State`, которым определяется текущее состояние.

`State` – состояние: определяет интерфейс для инкапсуляции поведения, ассоциированного с конкретным состоянием контекста `Context`.

`ConcreteState` – конкретные состояния: каждый класс `ConcreteState` реализует поведение, ассоциированное с некоторым состоянием контекста `Context`.

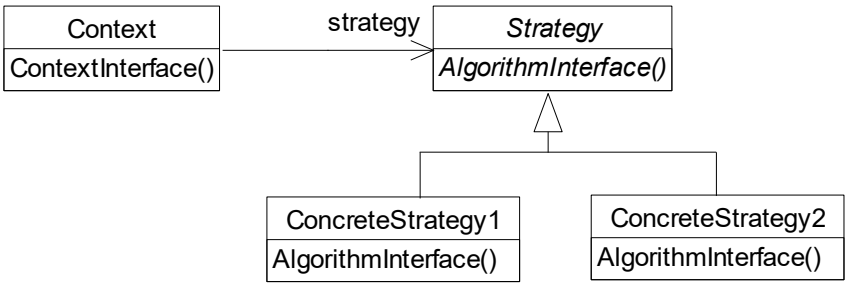
Класс Context делегирует зависящие от состояния запросы текущему объекту ConcreteState. Контекст может передать себя в качестве аргумента объекту State, который будет обрабатывать запрос. Это дает возможность объекту-состоянию при необходимости получить доступ к контексту.

Context – это основной интерфейс для клиентов. Клиенты могут конфигурировать контекст объектами состояния State. Один раз сконфигурировав контекст, клиенты уже не должны напрямую связываться с объектами состояния.

Стратегия (Strategy)

Назначение: определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми, позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются.

Общая структура решения.



Strategy – стратегия: объявляет общий для всех поддерживаемых алгоритмов интерфейс. Класс Context пользуется этим интерфейсом для вызова конкретного алгоритма, определенного в классе ConcreteStrategy.

ConcreteStrategy – конкретные стратегии: реализуют алгоритмы, использующие интерфейс, объявленный в классе Strategy.

Context – контекст: хранит ссылку на объект класса Strategy, которая конфигурируется конкретным объектом. Может определять интерфейс, который позволяет объекту Strategy получить доступ к данным контекста.

Классы Strategy и Context взаимодействуют для реализации выбранного алгоритма. Контекст может передать стратегии все необходимые алгоритму данные в момент его вызова. Либо вместо этого контекст может позволить стратегии обращаться к своим операциям в нужные моменты, передав ей ссылку на самого себя.

Контекст переадресует запросы своих клиентов объекту-стратегии. Обычно клиент создает объект ConcreteStrategy и передает его контексту, после чего клиент общается исключительно с контекстом. При этом в распоряжении клиента находится несколько классов ConcreteStrategy, которые он может выбирать.

Резюме

Инкапсуляция вариаций – элемент многих паттернов поведения. Если определенная часть программы подвержена периодическим изменениям, эти паттерны позволяют определить объект для инкапсуляции такого аспекта. Другие части программы, зависящие от данного аспекта, могут кооперироваться с ним. Обычно паттерны поведения определяют **абстрактный класс**, с помощью которого описывается инкапсулирующий объект. Своим названием паттерн обычно как раз и обязан этому объекту. Например:

- объект **Стратегия** инкапсулирует алгоритм;
- объект **Состояние** инкапсулирует поведение, зависящее от состояния;
- объект **Посредник** инкапсулирует протокол общения между объектами;
- объект **Итератор** инкапсулирует способ доступа и обхода компонентов составного объекта.

На самом деле, это характерно не только для паттернов поведения. Например, паттерны **Абстрактная Фабрика**, **Строитель** и **Прототип** инкапсулируют знание о том, как создаются объекты. **Декоратор** инкапсулирует обязанности, которые могут быть добавлены к объекту. **Мост** отделяет абстракцию от ее **реализации**, позволяя изменять их независимо друг от друга.

Перечисленные паттерны описывают подверженные изменениям аспекты программы. В большинстве паттернов фигурируют два вида объектов: новый объект (или объекты), который инкапсулирует аспект, и существующий объект (или объекты), который пользуется новыми объектами. Если бы не паттерн, то функциональность новых объектов пришлось бы делать неотъемлемой частью существующих. Например, код объектов-стратегий, вероятно, был бы зашит в контекст клиента, а код объектов-состояний был бы реализован непосредственно в контексте состояния.

Но не все паттерны поведения разбивают функциональность таким образом. Например, паттерн **Цепочка Обязанностей** связан с произвольным числом объектов (то есть цепочкой), причем все они могут уже существовать в системе. Цепочка Обязанностей иллюстрирует еще одно различие между паттернами поведения: не все они определяют статические отношения взаимосвязи между классами. В частности, Цепочка Обязанностей показывает, как организовать обмен информацией между заранее не известным числом объектов. В других паттернах участвуют объекты, передаваемые в качестве аргументов.

Объекты как аргументы. В нескольких паттернах участвует объект, который всегда используется только как аргумент. Одним из них является **Посетитель**. Объект-посетитель – это аргумент полиморфной операции `Accept`, принадлежащей посещаемому объекту. Посетитель никогда не рассматривается как часть посещаемых объектов, хотя традиционным альтернативным вариантом этому паттерну служит распределение кода посетителя между классами объектов, входящих в структуру.

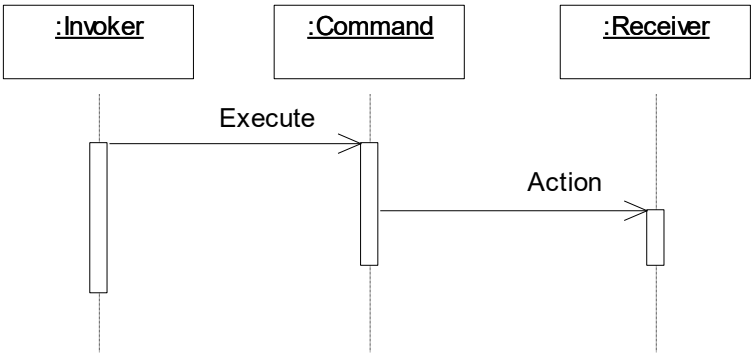
Другие паттерны определяют объекты, которые передаются от одного владельца к другому и активизируются в будущем. К этой категории относятся **Команда** и **Хранитель**. В паттерне Команда таким объектом является запрос, а в Хранителе – внутреннее состояние объекта в определенный момент. И там, и там объект может иметь сложную внутреннюю структуру, но клиент об этом ничего не знает. Но даже здесь есть различия. В паттерне Команда важную роль играет полиморфизм, поскольку выполнение объекта-команды – полиморфная операция. Напротив, интерфейс хранителя настолько узок, что его можно передавать лишь как значение. Поэтому вполне вероятно, что хранитель не предоставляет полиморфных операций своим клиентам.

Должен ли обмен информацией быть инкапсулированным или распределенным? Паттерны **Посредник** и **Наблюдатель** конкурируют между собой. Различие между ними в том, что Наблюдатель распределяет обмен информацией за счет объектов наблюдатель и субъект, а Посредник, наоборот, инкапсулирует взаимодействие между другими объектами. В паттерне Наблюдатель участники наблюдатель и субъект должны кооперироваться, чтобы поддержать функциональность.

Паттерны обмена информацией определяются тем, как связаны между собой наблюдатели и субъекты; у одного субъекта обычно бывает много наблюдателей, а иногда наблюдатель субъекта сам является субъектом наблюдения со стороны другого объекта. В паттерне Посредник ответственность за поддержание функциональности возлагается исключительно на посредника. Чаще всего повторное использование наблюдателей и субъектов реализовать проще, чем повторное использование посредников. Паттерн Наблюдатель способствует разделению и ослаблению связей между наблюдателем и субъектом, что приводит к появлению сравнительно мелких классов, более приспособленных для повторного использования.

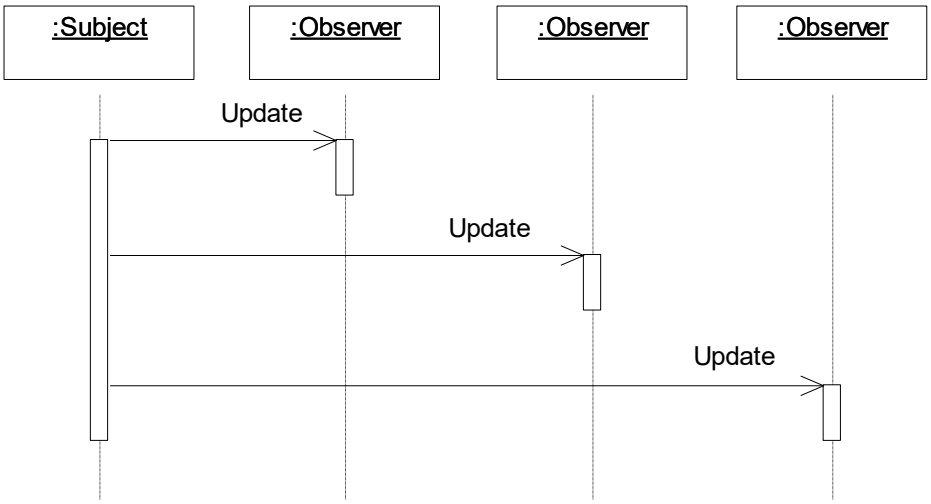
С другой стороны, потоки информации в Посреднике проще для понимания, нежели в Наблюдателе. Наблюдатели и субъекты обычно связываются вскоре после создания, и понять, каким же образом организована их связь, в последующих частях программы довольно трудно.

Разделение получателей и отправителей. Когда взаимодействующие объекты напрямую ссылаются друг на друга, они становятся зависимыми, а это может отрицательно сказаться на повторном использовании системы и разбиении ее на уровни. Паттерны **Команда**, **Наблюдатель**, **Посредник** и **Цепочка Обязанностей** указывают разные способы разделения получателей и отправителей запросов. Каждый способ имеет свои достоинства и недостатки.



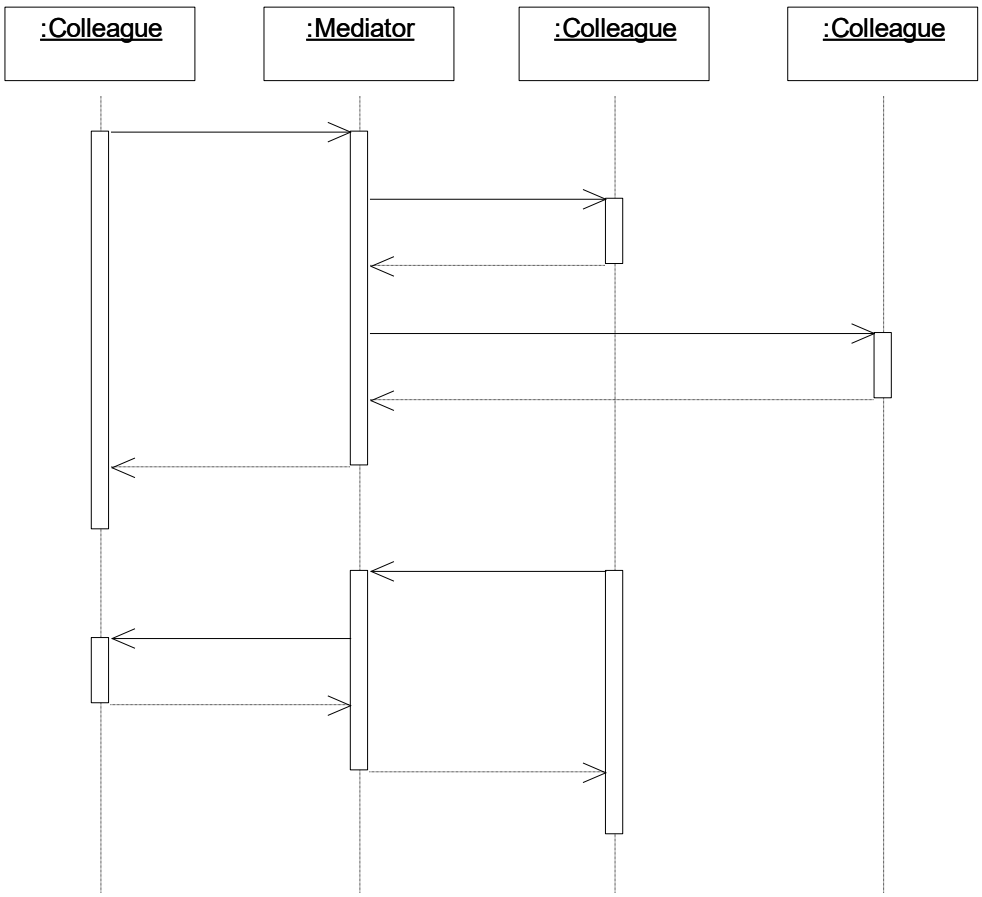
Паттерн Команда поддерживает разделение за счет объекта-команды, который определяет привязку отправителя к получателю. Паттерн Команда предоставляет простой интерфейс для выдачи запроса (операцию `Execute`). Заключение связи между отправителем и получателем в самостоятельный объект позволяет отправителю работать с разными получателями. Он отделяет отправителя от получателей, облегчая тем самым повторное использование. Кроме того, объект-команду можно повторно использовать для параметризации получателя различными отправителями. Номинально паттерн Команда требует определения **подкласса** для каждой связи отправитель-получатель, хотя имеются способы реализации, при которых удается избежать порождения подклассов.

Паттерн **Наблюдатель** отделяет отправителей (субъектов) от получателей (наблюдателей) путем определения интерфейса для извещения о произошедших с субъектом изменениях. По сравнению с Командой в Наблюдателе связь между отправителем и получателем слабее, поскольку у субъекта может быть много наблюдателей и их число даже может меняться во время выполнения.



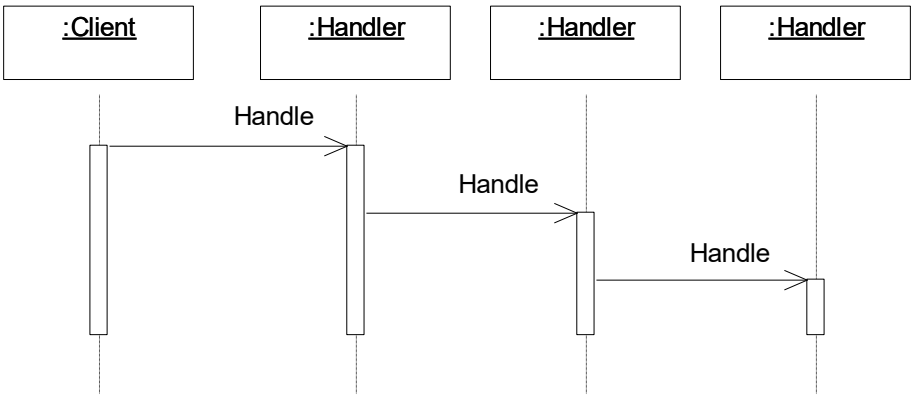
Интерфейсы субъекта и наблюдателя в паттерне Наблюдатель предназначены для передачи информации об изменениях. Стало быть, этот паттерн лучше всего подходит для разделения объектов в случае, когда между ними есть зависимость по данным.

Паттерн **Посредник** разделяет объекты, заставляя их ссылаться друг на друга косвенно, через объект-посредник.



Объект-посредник распределяет запросы между объектами-коллегами и централизует обмен информацией между ними. Таким образом, коллеги могут общаться между собой только с помощью интерфейса посредника. Поскольку этот интерфейс фиксирован, посредник может реализовать собственную схему диспетчеризации для большей гибкости. Разрешается кодировать запросы и упаковывать аргументы так, что коллеги смогут запрашивать выполнение операций из заранее неизвестного множества. Паттерн Посредник часто способствует уменьшению числа подклассов в системе, поскольку централизует весь обмен информацией в одном классе, вместо того чтобы распределять его по подклассам.

Наконец, паттерн **Цепочка Обязанностей** отделяет отправителя от получателя за счет передачи запроса по цепочке потенциальных получателей.



Поскольку интерфейс между отправителями и получателями фиксирован, то Цепочка Обязанностей также может нуждаться в специализированной схеме диспетчеризации. Поэтому она обладает теми же недостатками с точки зрения безопасности типов, что и Посредник. Цепочка Обязанностей – это хороший способ разделить отправителя и получателя в случае, если она уже является частью структуры системы, а один объект из группы может принять на себя обязанность обработать запрос. Данный паттерн повышает гибкость и за счет того, что цепочку можно легко изменить или расширить.