Оглавление

Порождающие паттерны

Паттерн Abstract Factory

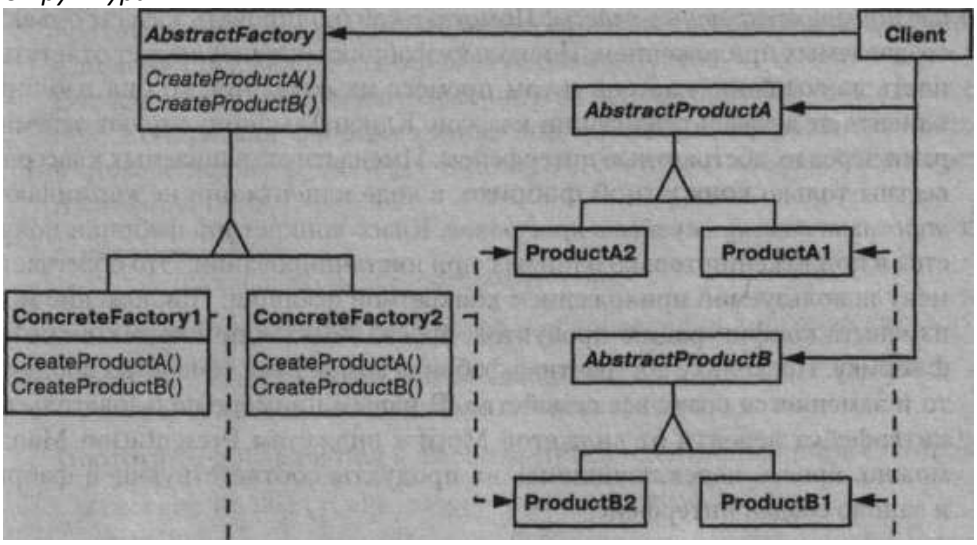
Абстрактная фабрика - паттерн, порождающий объекты.

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

Применимость:
Используйте паттерн абстрактная фабрика, когда:

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

Структура:



Участники:

- AbstractFactory - абстрактная фабрика: объявляет интерфейс для операций, создающих абстрактные объекты-продукты;
- ConcreteFactory - конкретная фабрика: реализует операции, создающие конкретные объекты-продукты;
- AbstractProduct - абстрактный продукт: объявляет интерфейс для типа объекта-продукта;
- ConcreteProduct - конкретный продукт: определяет объект - продукт, создаваемый соответствующей конкретной фабрикой, реализует интерфейс Abstract Product;

- Client - клиент: пользуется исключительно интерфейсами, которые объявлены в классах AbstractFactory и AbstractProduct.

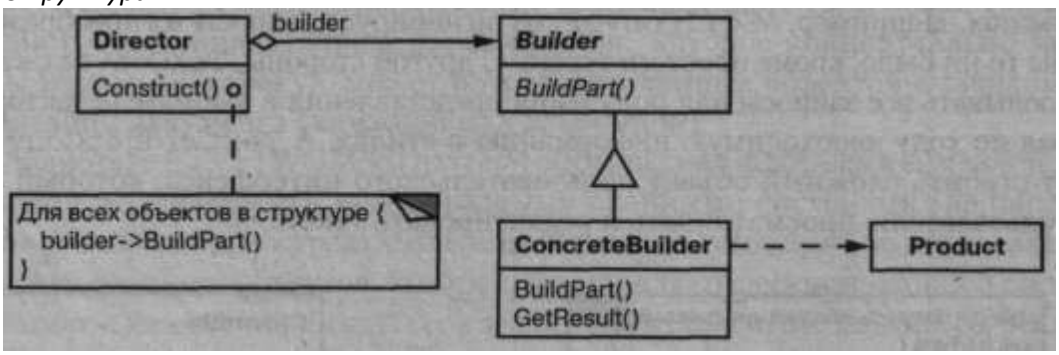
Результаты:
Паттерн абстрактная фабрика обладает следующими плюсами и минусами:
- Изолирует конкретные классы.
- Упрощает замену семейств продуктов.
- Гарантирует сочетаемость продуктов.
- Поддержать новый вид продуктов трудно.

Паттерн Builder

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

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.

Структура:



Участники:
- Builder - строитель: задает абстрактный интерфейс для создания частей объекта Product;
- ConcreteBuilder- 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;
- Director - распорядитель: конструирует объект, пользуясь интерфейсом Builder;
- Product - 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.

Результаты:
Плюсы и минусы паттерна строитель и его применения:
- Позволяет изменять внутреннее представление продукта;
- Изолирует код, реализующий конструирование и представление;
- Дает более тонкий контроль над процессом конструирования.

Паттерн Factory Method

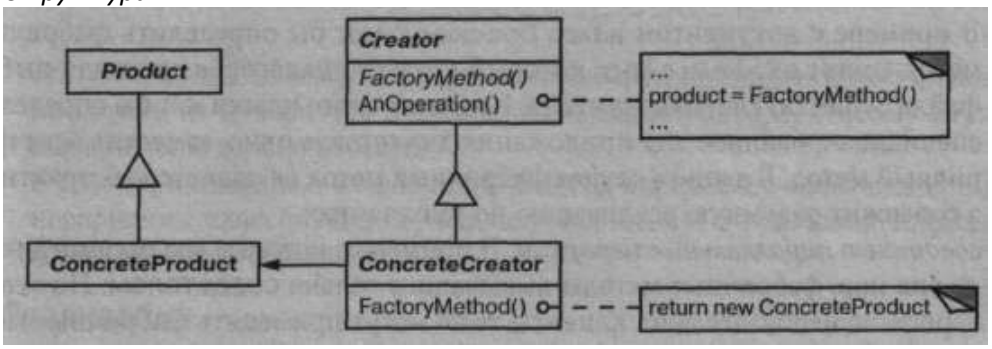
Фабричный метод - паттерн, порождающий классы.

Назначение:
Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать. Фабричный метод позволяет классу делегировать инстанцирование подклассам.

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 принимает эти обязанности на себя.

Структура:



Участники:

- Product (Document) - продукт: определяет интерфейс объектов, создаваемых фабричным методом;
- ConcreteProduct (MyDocument) - конкретный продукт: реализует интерфейс Product;
- Creator (Application) - создатель: объявляет фабричный метод, возвращающий объект типа Product. Creator может также определять реализацию по умолчанию фабричного метода, который возвращает объект ConcreteProduct; Может вызывать фабричный метод для создания объекта Product.
- ConcreteCreator (MyApplication) - конкретный создатель: замещает фабричный метод, возвращающий объект ConcreteProduct.

Достоинства:

- Позволяет сделать код создания объектов более универсальным, не привязываясь к конкретным классам (ConcreteProduct), а оперируя лишь общим интерфейсом (Product);
- Позволяет установить связь между параллельными иерархиями классов.

Недостатки:

- Необходимость создавать наследника Creator для каждого нового типа продукта (ConcreteProduct). Впрочем, современные языки программирования поддерживают конструкции, что позволяют реализовать фабричный метод без иерархии классов Creator.

Паттерн Prototype

Прототип - паттерн, порождающий объекты.

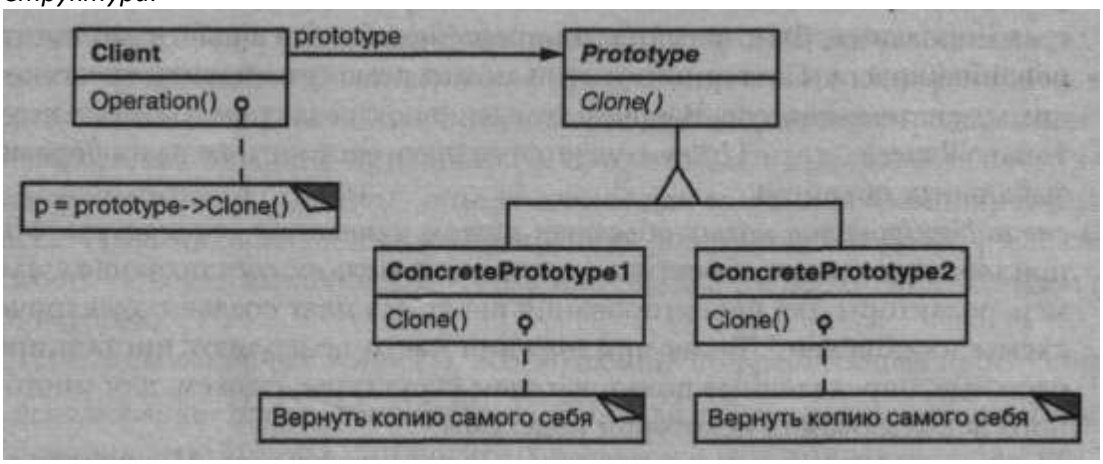
Назначение:

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

Применимость:

- Используйте паттерн прототип, когда система не должна зависеть от того, как в ней создаются, компонуются и представляются продукты:
- Инстанцируемые классы определяются во время выполнения, например с помощью динамической загрузки;
- Для того чтобы избежать построения иерархий классов или фабрик, параллельных иерархии классов продуктов;
- Экземпляры класса могут находиться в одном из не очень большого числа различных состояний. Может оказаться удобнее установить соответствующее число прототипов и клонировать их, а не инстанцировать каждый раз класс вручную в подходящем состоянии.

Структура:



Участники:

- Prototype - прототип: объявляет интерфейс для клонирования самого себя;
- ConcretePrototype - конкретный прототип: реализует операцию клонирования себя;
- Client - клиент: создает новый объект, обращаясь к прототипу с запросом клонировать себя.

Результаты

У прототипа те же самые результаты, что у абстрактной фабрики и строителя: он скрывает от клиента конкретные классы продуктов, уменьшая тем самым число известных клиенту имен. Кроме того, все эти паттерны позволяют клиентам работать со специфичными для приложения классами без модификаций.

Ниже перечислены дополнительные преимущества паттерна прототип:

- Добавление и удаление продуктов во время выполнения.
- Спецификация новых объектов путем изменения значений.
- Специфицирование новых объектов путем изменения структуры
- Уменьшение числа подклассов.
- Динамическое конфигурирование приложения классами.

Паттерн Singleton

Назначение

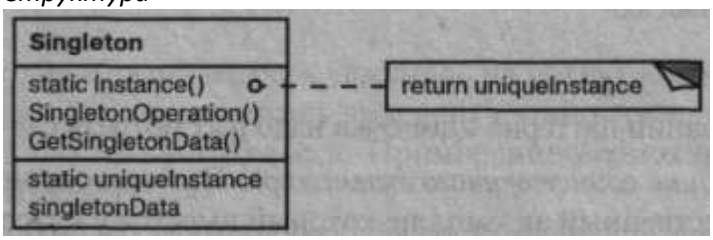
Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

Применимость

Используйте паттерн одиночка, когда:

- Должен быть ровно один экземпляр некоторого класса, легко доступный всем клиентам;
- Единственный экземпляр должен расширяться путем порождения подклассов, и клиентам нужно иметь возможность работать с расширенным экземпляром без модификации своего кода.

Структура



Участники

- Singleton - одиночка: определяет операцию Instance, которая позволяет клиентам получать доступ к единственному экземпляру. Instance – это метод класса и статическая функция-член в C++; Может нести ответственность за создание собственного уникального экземпляра.

Результаты

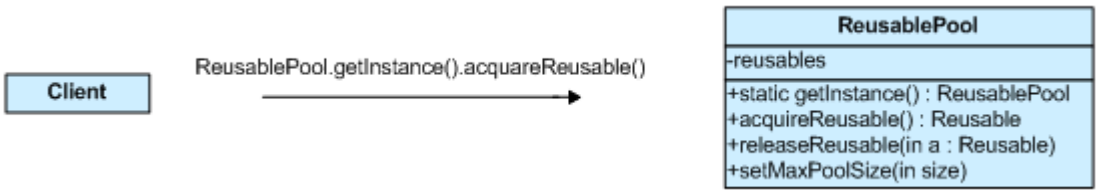
У паттерна одиночка есть определенные достоинства:

- Контролируемый доступ к единственному экземпляру.
- Уменьшение числа имен.
- Допускает уточнение операций и представления.
- Допускает переменное число экземпляров.
- Большая гибкость, чем у операций класса.

Паттерн Pool

Применимость

Объектный пул применяется для повышения производительности, когда создание объекта в начале работы и уничтожение его в конце приводит к большим затратам. Особенно заметно повышение производительности, когда объекты часто создаются-уничтожаются, но одновременно существует лишь небольшое их число.



Участники

- Reusable - экземпляры классов в этой роли взаимодействуют с другими объектами в течение ограниченного времени, а затем они больше не нужны для этого взаимодействия.
- Client - экземпляры классов в этой роли используют объекты Reusable.
- ReusablePool - экземпляры классов в этой роли управляют объектами Reusable для использования объектами Client.

Структурные паттерны

Паттерн Adapter

УТОЧНИТЬ КАКОЙ ИМЕННО АДАПТЕР: для объектов или интерфейса

Адаптер - паттерн, структурирующий классы и объекты.

Назначение

Преобразует интерфейс одного класса в интерфейс другого, который ожидают клиенты. Адаптер обеспечивает совместную работу классов с несовместимыми интерфейсами, которая без него была бы невозможна.

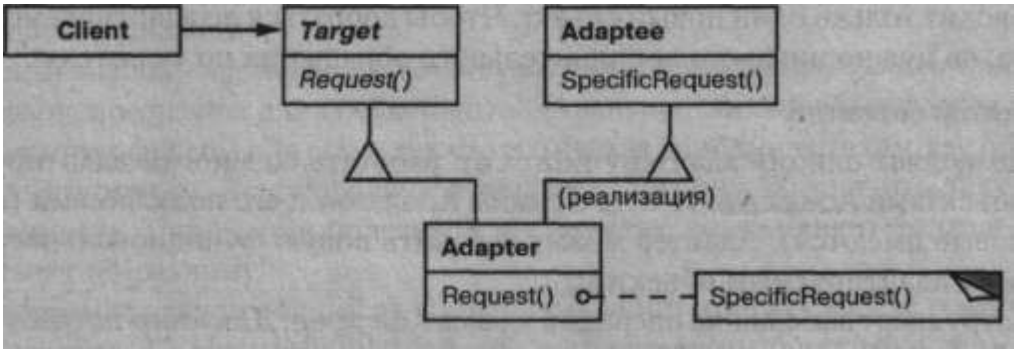
Применимость

Применяйте паттерн адаптер, когда:

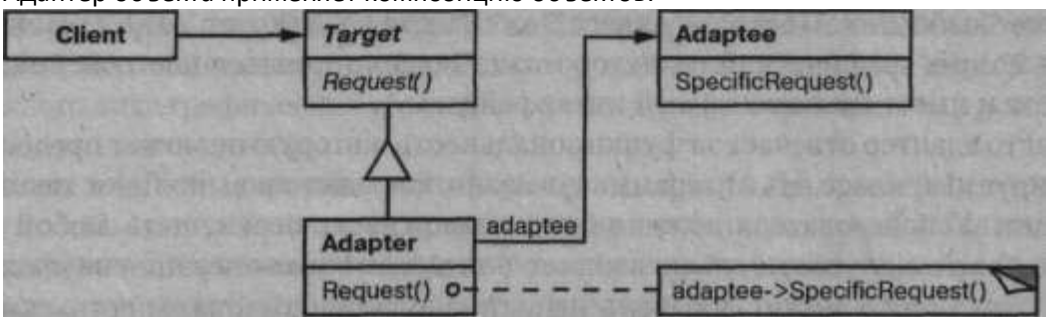
- Хотите использовать существующий класс, но его интерфейс не соответствует вашим потребностям;
- Собираетесь создать повторно используемый класс, который должен взаимодействовать с заранее неизвестными или не связанными с ним классами, имеющими несовместимые интерфейсы;
- (только для адаптера объектов!) Нужно использовать несколько существующих подклассов, но непрактично адаптировать их интерфейсы путем порождения новых подклассов от каждого. В этом случае адаптер объектов может приспосабливать интерфейс их общего родительского класса.

Структура

Адаптер класса использует множественное наследование для адаптации одного интерфейса к другому.



Адаптер объекта применяет композицию объектов.



Участники

- Target - целевой: определяет зависящий от предметной области интерфейс, которым пользуется Client;
- Client - клиент: вступает во взаимоотношения с объектами, удовлетворяющими интерфейсу Target;
- Adaptee - адаптируемый: определяет существующий интерфейс, который нуждается в адаптации;
- Adapter - адаптер: адаптирует интерфейс Adaptee к интерфейсу Target.

Результаты

Результаты применения адаптеров объектов и классов различны.

Адаптер класса:

- Адаптирует Adaptee к Target, перепоручая действия конкретному классу Adaptee. Поэтому данный паттерн не будет работать, если мы захотим одновременно адаптировать класс и его подклассы;
- Позволяет адаптеру Adapter заместить некоторые операции адаптируемого класса Adaptee, так как Adapter есть не что иное, как подкласс Adaptee;
- Вводит только один новый объект. Чтобы добраться до адаптируемого класса, не нужно никакого дополнительного обращения по указателю.

Адаптер объектов:

- Позволяет одному адаптеру Adapter работать со многим адаптируемыми объектами Adaptee, то есть с самим Adaptee и его подклассами (если таковые имеются). Адаптер может добавить новую функциональность сразу всем адаптируемым объектам;
- Затрудняет замещение операций класса Adaptee. Для этого потребуется породить от Adaptee подкласс и заставить Adapter ссылаться на этот подкласс, а не на сам Adaptee.

Паттерн Bridge

Назначение

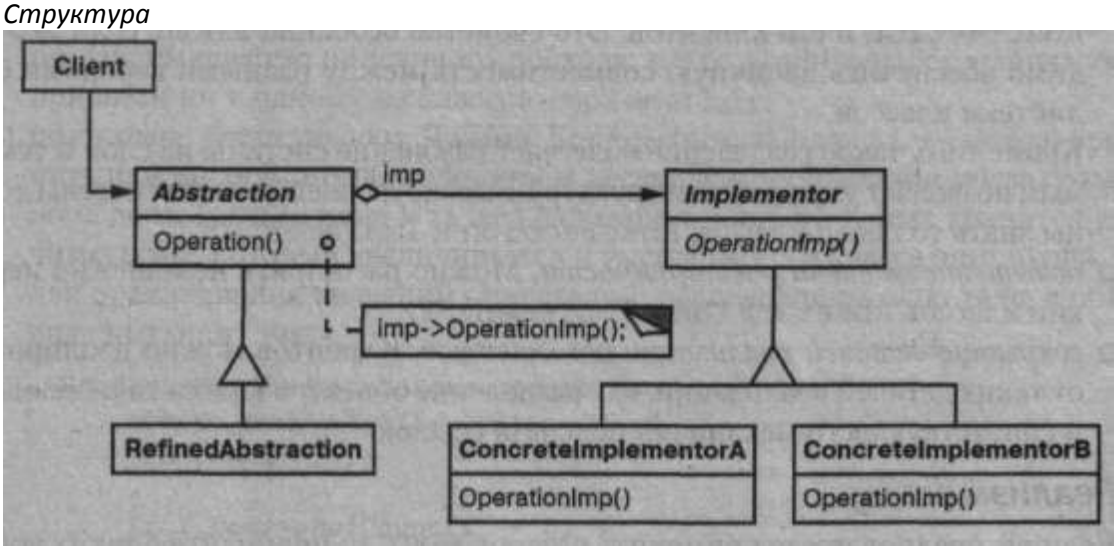
Отделить абстракцию от ее реализации так, чтобы то и другое можно было изменять независимо.

Применимость

- Используйте паттерн мост, когда:
- Хотите избежать постоянной привязки абстракции к реализации. Так, например, бывает, когда реализацию необходимо выбирать во время выполнения программы;
- И абстракции, и реализации должны расширяться новыми подклассами.
  В таком случае паттерн мост позволяет комбинировать разные абстракции и реализации и изменять их независимо;
- Изменения в реализации абстракции не должны сказываться на клиентах, то есть клиентский код не должен перекомпилироваться;

- Вы хотите полностью скрыть от клиентов реализацию абстракции. В C++ представление класса видимо через его интерфейс;

Структура



Участники

- 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. Он не обязан точно соответствовать интерфейсу класса Abstraction. На самом деле 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 - конкретный реализатор: содержит конкретную реализацию интерфейса класса Implementor.

Результаты

Результаты применения паттерна мост таковы:

- Отделение реализации от интерфейса.
- Повышение степени расширяемости.
- Сокрытие деталей реализации от клиентов.

Паттерн Composite

Назначение

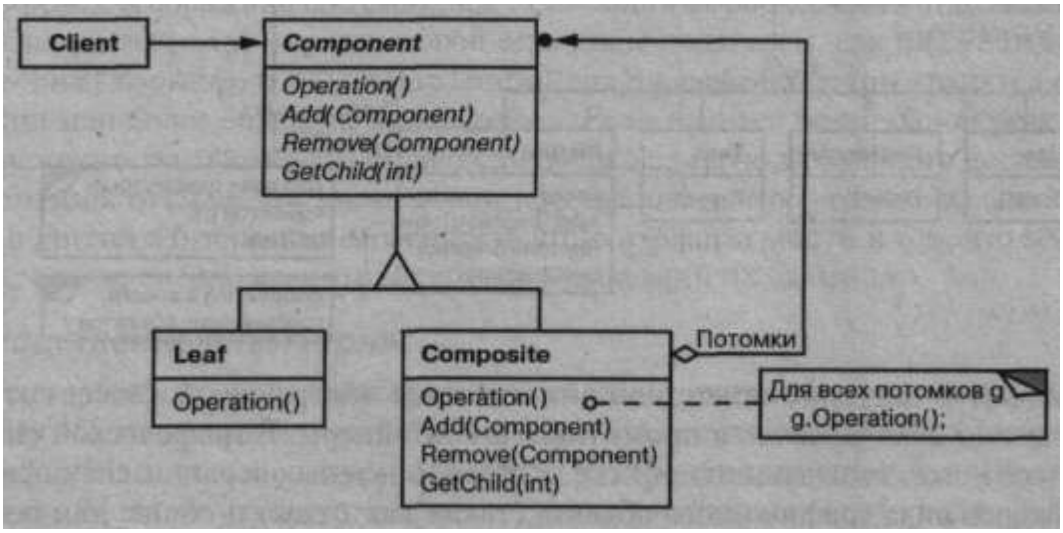
Компонует объекты в древовидные структуры для представления иерархий часть-целое. Позволяет клиентам единообразно трактовать индивидуальные и составные объекты.

Применимость

Используйте паттерн компоновщик, когда:

- Нужно представить иерархию объектов вида часть-целое;
- Хотите, чтобы клиенты единообразно трактовали составные и индивидуальные объекты.

Структура



Участники

- Component - компонент:объявляет интерфейс для компонуемых объектов; предоставляет подходящую реализацию операций по умолчанию, общую для всех классов; объявляет интерфейс для доступа к потомкам и управления ими; определяет интерфейс для доступа к родителю компонента в рекурсивной структуре и при необходимости реализует его. Описанная возможность необязательна;
- Leuf - лист: представляет листовые узлы композиции и не имеет потомков; определяет поведение примитивных объектов в композиции;
- Composite - составной объект: определяет поведение компонентов, у которых есть потомки; хранит компоненты-потомки; реализует относящиеся к управлению потомками операции в интерфейсе класса Component;
- Client - клиент: манипулирует объектами композиции через интерфейс Component.

Результаты

Паттерн компоновщик:

- Определяет иерархии классов, состоящие из примитивных и составных объектов.
- Упрощает архитектуру клиента.
- Облегчает добавление новых видов компонентов.
- Способствует созданию общего дизайна.

Паттерн Decorator

Назначение

Динамически добавляет объекту новые обязанности. Является гибкой альтернативой порождению подклассов с целью расширения функциональности.

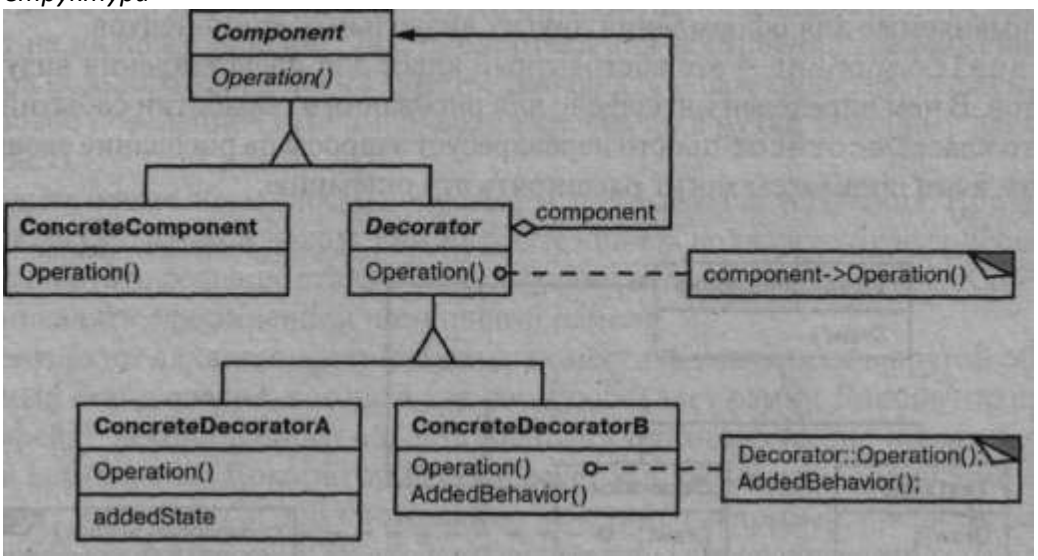
Применимость

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 нельзя.

Структура



Участники

- Component - компонент: определяет интерфейс для объектов, на которые могут быть динамически возложены дополнительные обязанности;
- ConcreteComponent - 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) - конкретный декоратор: возлагает дополнительные обязанности на компонент.

Результаты

У паттерна декоратор есть, по крайней мере, два плюса и два минуса:

- Большая гибкость, нежели у статического наследования.
- Позволяет избежать перегруженных функциями классов на верхних уровнях иерархии.
- Декоратор и его компонент не идентичны.
- Множество мелких объектов. (При использовании в проекте паттерна декоратор нередко получается система, составленная из большого числа мелких объектов, которые похожи друг на друга 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.)

Паттерн Facade

Назначение

Предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Фасад определяет интерфейс более высокого уровня, который упрощает использование подсистемы.

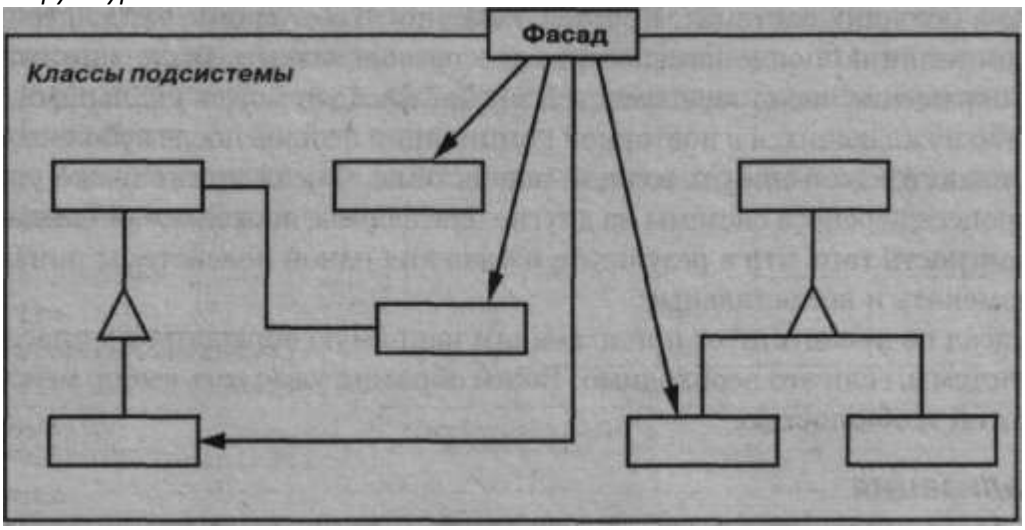
Применимость

Используйте паттерн фасад, когда:

- Хотите предоставить простой интерфейс к сложной подсистеме.
- Между клиентами и классами реализации абстракции существует много зависимостей.

- Вы хотите разложить подсистему на отдельные слои.

Структура



Участники

- Facade - фасад: «знает», каким классам подсистемы адресовать запрос; делегирует запросы клиентов подходящим объектам внутри подсистемы;
- Классы подсистемы: реализуют функциональность подсистемы; выполняют работу, порученную объектом Facade; ничего не «знают» о существовании фасада, то есть не хранят ссылок на него.

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;
- Фасад не препятствует приложениям напрямую обращаться к классам подсистемы, если это необходимо.

Паттерн Proxy

Назначение

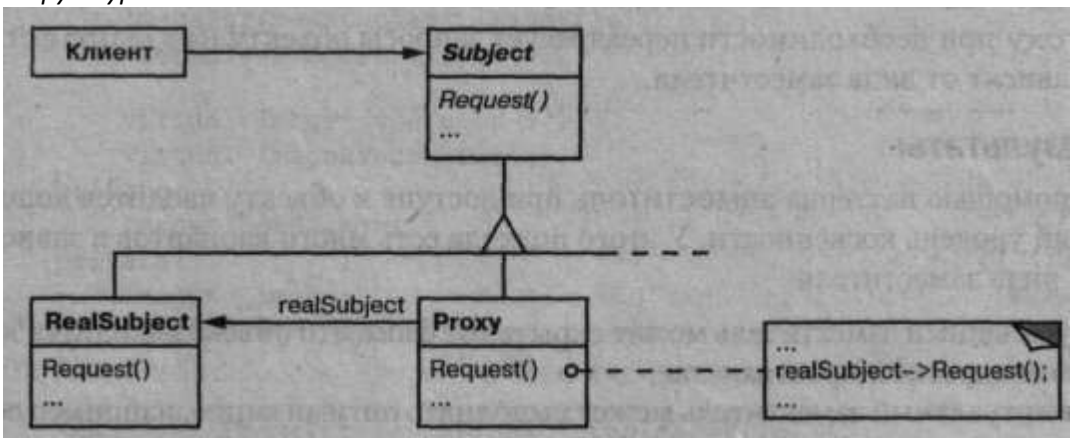
Является суррогатом другого объекта и контролирует доступ к нему.

Применимость

- Удаленный заместитель предоставляет локального представителя вместо объекта, находящегося в другом адресном пространстве;
- Виртуальный заместитель создает «тяжелые» объекты по требованию.
- Защищающий заместитель контролирует доступ к исходному объекту.

- «Умная» ссылка - это замена обычного указателя.

Структура



Участники

- Proxy - заместитель: хранит ссылку, которая позволяет заместителю обратиться к реальному субъекту. Объект класса Proxy может обращаться к объекту класса Subject, если интерфейсы классов RealSubject и Subject одинаковы; предоставляет интерфейс, идентичный интерфейсу Subject, так что заместитель всегда может быть подставлен вместо реального субъекта; контролирует доступ к реальному субъекту и может отвечать за его создание и удаление;
- 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.

Результаты

С помощью паттерна заместитель при доступе к объекту вводится дополнительный уровень косвенности. У этого подхода есть много вариантов в зависимости от вида заместителя:

- Удаленный заместитель может скрыть тот факт, что объект находится в другом адресном пространстве;
- Виртуальный заместитель может выполнять оптимизацию, например создание объекта по требованию;
- Защищающий заместитель и «умная» ссылка позволяют решать дополнительные задачи при доступе к объекту.

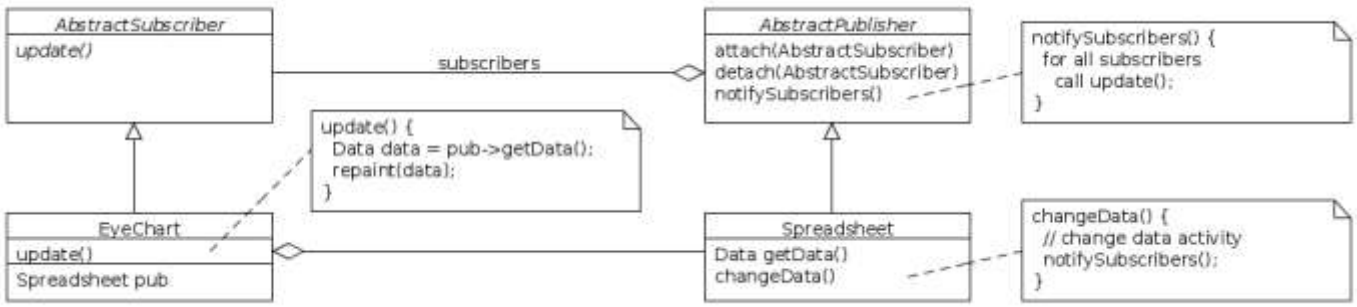
Паттерны поведения

Паттерн Publish-Subscribe

Назначение

Если один или несколько объектов в Вашем проекте должны отслеживать изменения другого объекта – гибким решением будет применение шаблона проектирования Publish-Subscribe.

Структура



Участники:

- AbstractSubscriber, задает интерфейс *подписчиков*;
- EyeChart, один из реальных подписчиков (легко могут быть добавлены другие виды диаграмм). Хранит ссылку на *издателя*, при помощи которой получает его обновленное состояние после получения сигнала об обновлении данных;
- AbstractPublisher, задает интерфейс издателей. Может реализовывать этот интерфейс (не являться абстрактным), т.к. все *издатели* должны одинаково добавлять/удалять *подписчика* и уведомлять их об обновлении;
- SpreadSheet, таблица с данными, уведомляющая *подписчиков* об изменении своих данных.

Результат:

- Ослабляется зависимость *издателя* от *подписчика.* Остается зависимость лишь от абстрактного класса, что позволяет без модификации кода *издателя* добавлять новые типы *подписчиков;*
- Появляется возможность добавлять и удалять *подписчиков* во время выполнения программы. *Издатель* не располагает информацией не только о конкретных типах *Подписчиков,* но и об их количестве. Так, **например,** во многих играх наряду с картой, отображающей состояние игрового мира присутствует мини-карта. Обе карты являются *подписчиками,* а игровой мир – *издателем,* при этом для отключения мини-карты достаточно удалить соответствующий объект и снять его с подписки.

Паттерн Chain of Responsibility

Назначение

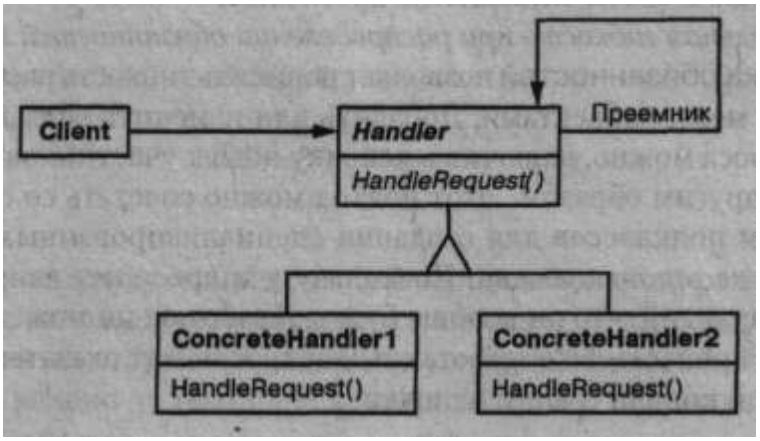
Позволяет избежать привязки отправителя запроса к его получателю, давая шанс обработать запрос нескольким объектам. Связывает объекты-получатели в цепочку и передает запрос вдоль этой цепочки, пока его не обработают.

Применимость

Используйте цепочку обязанностей, когда:

- Есть более одного объекта, способного обработать запрос, причем настоящий обработчик заранее неизвестен и должен быть найден автоматически;
- Вы хотите отправить запрос одному из нескольких объектов, не указывая явно, какому именно;
- Набор объектов, способных обработать запрос, должен задаваться динамически.

Структура



Участники

Handler - обработчик: определяет интерфейс для обработки запросов; (необязательно) реализует связь с преемником;
ConcreteHandler - конкретный обработчик: обрабатывает запрос, за который отвечает; имеет доступ к своему преемнику; если ConcreteHandler способен обработать запрос, то так и делает, если не может, то направляет его - его своему преемнику;
а Client - клиент: отправляет запрос некоторому объекту ConcreteHandler в цепочке.

Результаты
Паттерн цепочка обязанностей имеет следующие достоинства и недостатки:

- Ослабление связанности. Этот паттерн освобождает объект от необходимости «знать», кто конкретно обработает его запрос. Отправителю и получателю ничего неизвестно друг о друге, а включенному в цепочку объекту - о структуре цепочки. Таким образом, цепочка обязанностей помогает упростить взаимосвязи
- Дополнительная гибкость при распределении обязанностей между объектами.
- Получение не гарантировано. Поскольку у запроса нет явного получателя, то нет и гарантий, что он вообще будет обработан.

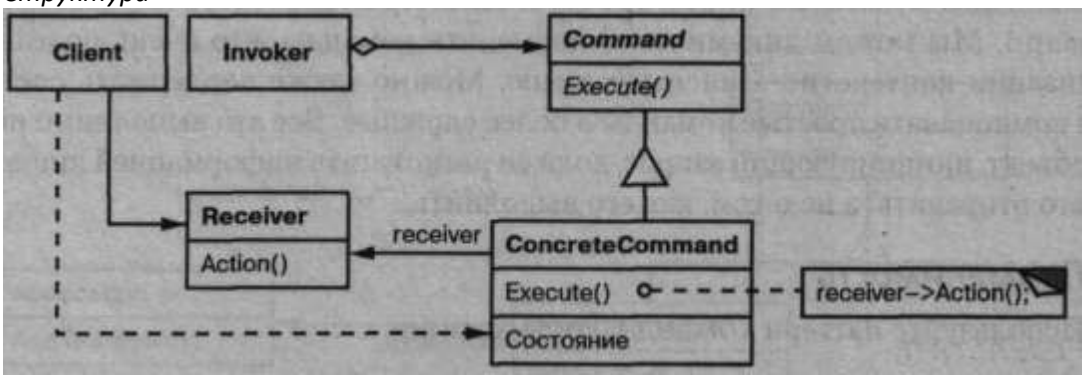
Паттерн Command

Назначение
Инкапсулирует запрос как объект, позволяя тем самым задавать параметры клиентов для обработки соответствующих запросов, ставить запросы в очередь или протоколировать их, а также поддерживать отмену операций.

Применимость
Используйте паттерн команда, когда хотите:

- Параметризовать объекты выполняемым действием, как в случае с пунктами меню.
- Определять, ставить в очередь и выполнять запросы в разное время.
- Поддержать отмену операций. Операция Execute объекта Command может сохранить состояние, необходимое для отката действий, выполненных командой.
- Поддержать протоколирование изменений, чтобы их можно было выполнить повторно после аварийной остановки системы.
- Структурировать систему на основе высокоуровневых операций, построенных из примитивных.

Структура



Участники

- Command - команда: объявляет интерфейс для выполнения операции;
- ConcreteCommand - конкретная команда: определяет связь между объектом-получателем Receiver и действием; реализует операцию Execute путем вызова соответствующих операций объекта Receiver;
- Client - клиент: создает объект класса ConcreteCommand и устанавливает его получателя;
- Invoker - инициатор: обращается к команде для выполнения запроса;

- Receiver (Document, Application) - получатель: располагает информацией о способах выполнения операций, необходимых для удовлетворения запроса. В роли получателя может выступать любой класс.

Результаты
Результаты применения паттерна команда таковы:
- 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. В общем случае составные команды описываются паттерном компоновщик;
- Добавлять новые команды легко, поскольку никакие существующие классы изменять не нужно.

Паттерн Mediator(Посредник)

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

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.

Структура

Участники
- Mediator (DialogDirector) - посредник; определяет интерфейс для обмена информацией с объектами Colleague;
- ConcreteMediator (FontDialogDirector) - конкретный посредник: реализует кооперативное поведение, координируя действия объектов Colleague; владеет информацией о коллегах и подсчитывает их;
- Классы Colleague (ListBox, EntryField) - коллеги: каждый класс Colleague «знает» о своем объекте Mediator; все коллеги обмениваются информацией только с посредником, так как при его отсутствии им пришлось бы общаться между собой напрямую.

Результаты
У паттерна посредник есть следующие достоинства и недостатки:
- Снижает число порождаемых подклассов.
- Устраняет связанность между коллегами.
- Упрощает протоколы взаимодействия объектов.
- Абстрагирует способ кооперирования объектов.
- Централизует управление.

Паттерн Memento(Хранитель)

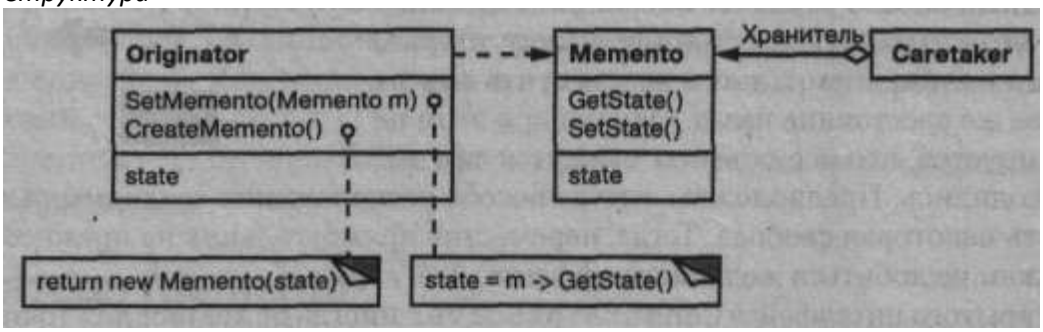
Назначение

Не нарушая инкапсуляции, фиксирует и выносит за пределы объекта его внутреннее состояние так, чтобы позднее можно было восстановить в нем объект.

Применимость

- Используйте паттерн хранитель, когда:
- Необходимо сохранить мгновенный снимок состояния объекта (или его части) чтобы впоследствии объект можно было восстановить в том же состоянии;
- Прямое получение этого состояния раскрывает детали реализации и нарушает инкапсуляцию объекта.

Структура



Участники

- Memento - хранитель: сохраняет внутреннее состояние объекта Originator. Объем сохраняемой информации может быть различным и определяется потребностями хозяина; запрещает доступ всем другим объектам, кроме хозяина.
- Originator - хозяин: создает хранитель, содержащего снимок текущего внутреннего состояния; использует хранитель для восстановления внутреннего состояния;
- Caretaker- посыльный: отвечает за сохранение хранителя; не производит никаких операций над хранителем и не исследует его внутреннее содержимое.

Результаты

Характерные особенности паттерна хранитель:

- Сохранение границ инкапсуляции.
- Упрощение структуры хозяина.
- Значительные издержки при использовании хранителей. С хранителями могут быть связаны заметные издержки, если хозяин должен копировать большой объем информации для занесения в память хранителя или если клиенты создают и возвращают хранителей достаточно часто.
- Скрытая плата за содержание хранителя. Посыльный отвечает за удаление хранителя, однако не располагает информацией о том, какой объем информации о состоянии скрыт в нем.

Паттерн Observer(наблюдатель)

Назначение

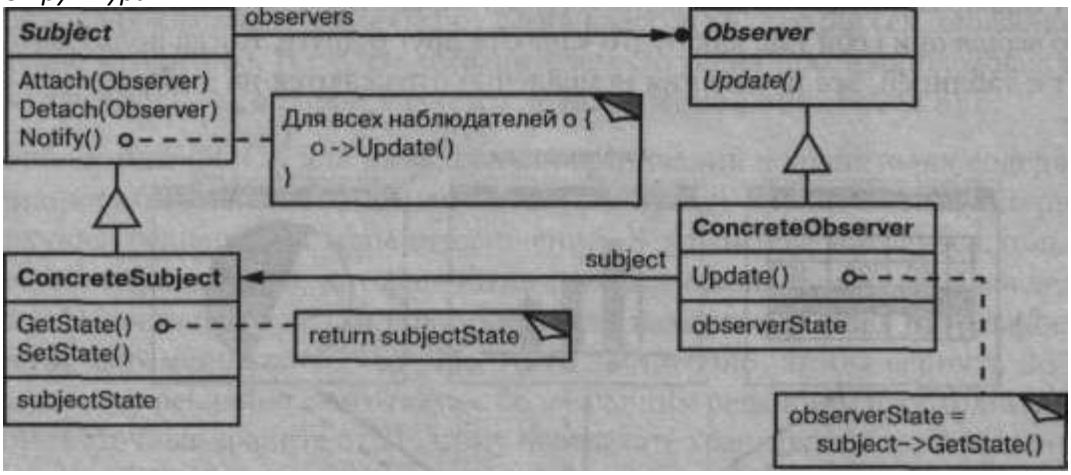
Определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом и автоматически обновляются.

Применимость

Используйте паттерн наблюдатель в следующих ситуациях:

- Когда у абстракции есть два аспекта, один из которых зависит от другого. Инкапсуляции этих аспектов в разные объекты позволяют изменять и повторно использовать их независимо;
- Когда при модификации одного объекта требуется изменить другие и вы не знаете, сколько именно объектов нужно изменить;
- Когда один объект должен оповещать других, не делая предположений об уведомляемых объектах. Другими словами, вы не хотите, чтобы объекты были тесно связаны между собой.

Структура



Участники

- *Subject - субъект: располагает информацией о своих наблюдателях. За субъектом может «следить» любое число наблюдателей; предоставляет интерфейс для присоединения и отделения наблюдателей;*
- *Observer - наблюдатель: определяет интерфейс обновления для объектов, которые должны быть уведомлены об изменении субъекта;*
- *ConcreteSubject - конкретный субъект: сохраняет состояние, представляющее интерес для конкретного наблюдателя ConcreteObserver; посылает информацию своим наблюдателям, когда происходит изменение;*
- *ConcreteObserver - конкретный наблюдатель: хранит ссылку на объект класса ConcreteSubject; сохраняет данные, которые должны быть согласованы с данными субъекта; реализует интерфейс обновления, определенный в классе Observer, чтобы поддерживать согласованность с субъектом.*

Результаты

Паттерн наблюдатель позволяет изменять субъекты и наблюдатели независимо друг от друга. Субъекты разрешается повторно использовать без участия наблюдателей, и наоборот. Это дает возможность добавлять новых наблюдателей без модификации субъекта или других наблюдателей.

Рассмотрим некоторые достоинства и недостатки паттерна наблюдатель:

- *Абстрактная связанность субъекта и наблюдателя. Субъект имеет информацию лишь о том, что у него есть ряд наблюдателей, каждый из которых подчиняется простому интерфейсу абстрактного класса Observer.*
- *Поддержка широковещательных коммуникаций. В отличие от обычного запроса для уведомления, посылаемого субъектом, не нужно задавать определенного получателя. Уведомление автоматически поступает всем подписавшимся на него объектам.*
- *Неожиданные обновления. Поскольку наблюдатели не располагают информацией друг о друге, им неизвестно и о том, во что обходится изменение субъекта.*

Паттерн State

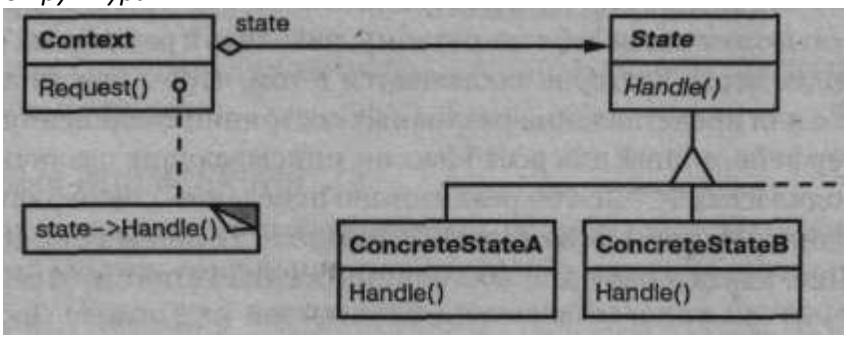
Назначение

Позволяет объекту варьировать свое поведение в зависимости от внутреннего состояния. Извне создается впечатление, что изменился класс объекта.

Применимость

- Используйте паттерн состояние в следующих случаях:
- Когда поведение объекта зависит от его состояния и должно изменяться во время выполнения;
- Когда в коде операций встречаются состоящие из многих ветвей условные операторы, в которых выбор ветви зависит от состояния. Обычно в таком случае состояние представлено перечисляемыми константами. Часто одна и та же структура условного оператора повторяется в нескольких операциях. Паттерн состояние предлагает поместить каждую ветвь в отдельный класс. Это позволяет трактовать состояние объекта как самостоятельный объект, который может изменяться независимо от других.

Структура



Участники

- Context - контекст:
  - определяет интерфейс, представляющий интерес для клиентов;
  - хранит экземпляр подкласса ConcreteState, которым определяется текущее состояние;
- State - состояние:
  - определяет интерфейс для инкапсуляции поведения, ассоциированного с конкретным состоянием контекста Context;
- Подклассы ConcreteState - конкретное состояние:
  - каждый подкласс реализует поведение, ассоциированное с некоторым состоянием контекста Context.

Результаты

- Результаты использования паттерна состояние:
- Локализует зависящее от состояния поведение и делит его на части, соответствующие состояниям.
- Делает явными переходы между состояниями.
- Объекты состояния можно разделять.

Паттерн Strategy

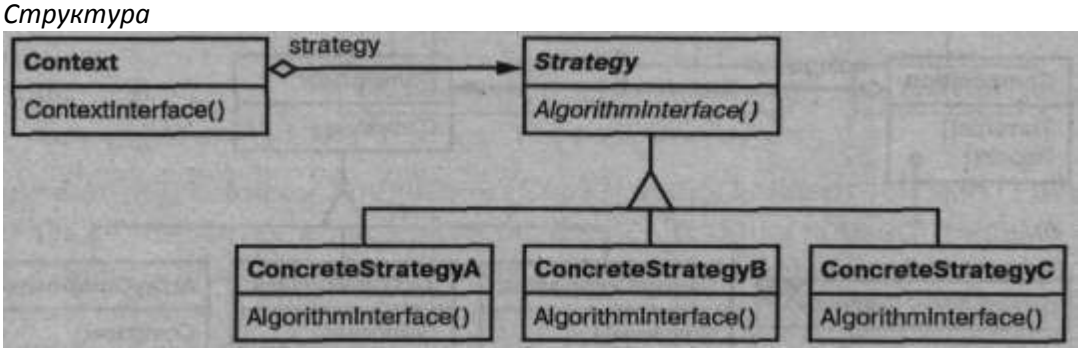
Назначение

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

Используйте паттерн стратегия, когда:

- Имеется много родственных классов, отличающихся только поведением.
- Вам нужно иметь несколько разных вариантов алгоритма.
- В алгоритме содержатся данные, о которых клиент не должен «знать». Используйте паттерн стратегия, чтобы не раскрывать сложные, специфичные для алгоритма структуры данных;
- В классе определено много поведений, что представлено разветвленными условными операторами. В этом случае проще перенести код из ветвей в отдельные классы стратегий.

Структура



Участники

- Strategy - стратегия: объявляет общий для всех поддерживаемых алгоритмов интерфейс. Класс Context пользуется этим интерфейсом для вызова конкретного алгоритма, определенного в классе ConcreteStrategy;
- ConcreteStrategy - конкретная стратегия: реализует алгоритм, использующий интерфейс, объявленный в классе Strategy;
- Context - контекст: конфигурируется объектом класса ConcreteStrategy; хранит ссылку на объект класса Strategy; может определять интерфейс, который позволяет объекту Strategy получить доступ к данным контекста.

Результаты
У паттерна стратегия есть следующие достоинства и недостатки:

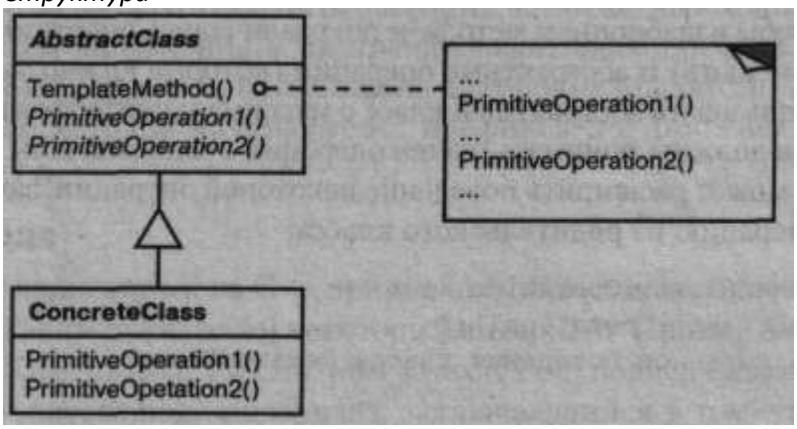
- Семейства родственных алгоритмов.
- Альтернатива порождению подклассов.
- С помощью стратегий можно избавиться от условных операторов. Благодаря паттерну стратегия удается отказаться от условных операторов при выборе нужного поведения.
- Выбор реализации. Стратегии могут предлагать различные реализации одного и того же поведения.
- Клиенты должны знать о различных стратегиях.
- Обмен информацией между стратегией и контекстом.
- Увеличение числа объектов.

Паттерн Template Method

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

Применимость
Паттерн шаблонный метод следует использовать:
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.
Для управления расширениями подклассов. Можно определить шаблонный метод так, что он будет вызывать операции-зацепки в определенных точках, разрешив тем самым расширение только в этих точках.

Структура



Участники

- AbstractClass (Application) - абстрактный класс: определяет абстрактные примитивные операции, замещаемые в конкретных подклассах для реализации шагов алгоритма; реализует шаблонный метод, определяющий скелет алгоритма. Шаблонный метод вызывает примитивные операции, а также операции, определенные в классе AbstractClass или в других объектах;
- ConcreteClass (MyApplication) - конкретный класс: реализует примитивные операции, выполняющие шаги алгоритма способом, который зависит от подкласса.

Паттерн Visitor(посетитель)

Назначение

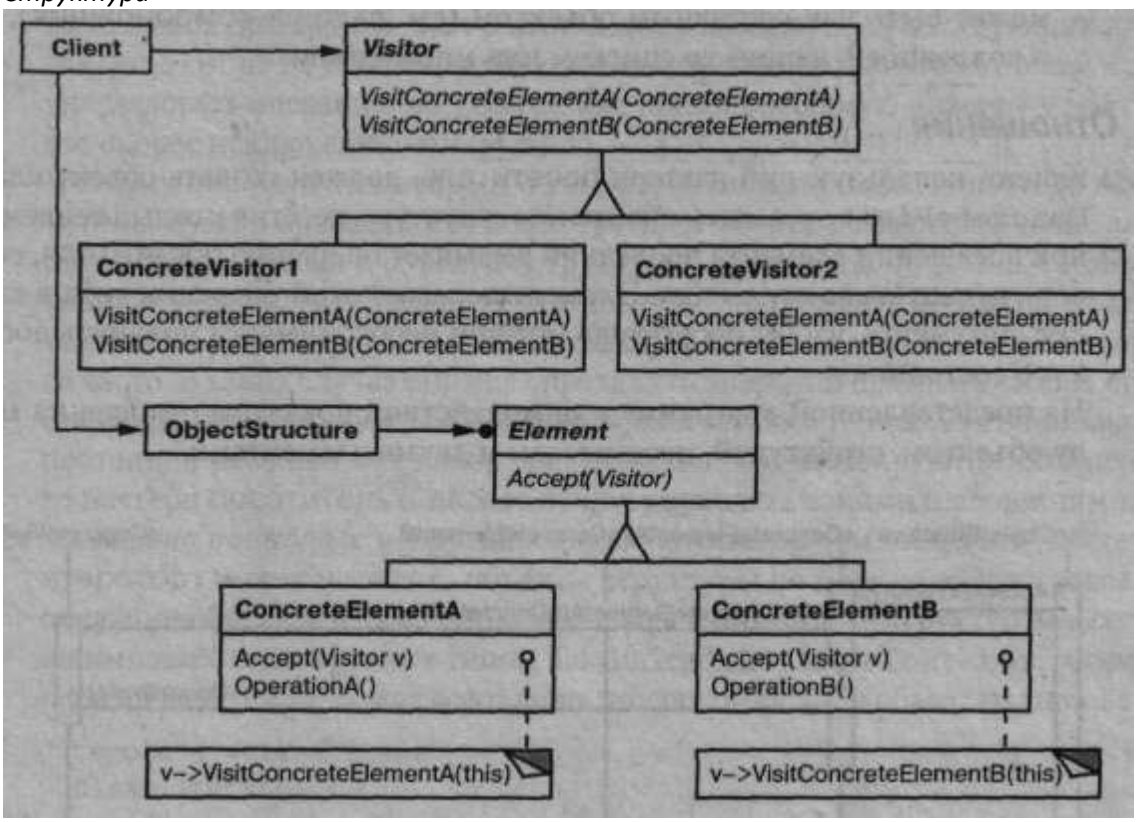
Описывает операцию, выполняемую с каждым объектом из некоторой структуры. Паттерн посетитель позволяет определить новую операцию, не изменяя классы этих объектов.

Применимость

Используйте паттерн посетитель, когда:

- В структуре присутствуют объекты многих классов с различными интерфейсами и вы хотите выполнять над ними операции, зависящие от конкретных классов;
- Над объектами, входящими в состав структуры, надо выполнять разнообразные, не связанные между собой операции и вы не хотите «засорять» классы такими операциями.
- Классы, устанавливающие структуру объектов, изменяются редко, но новые операции над этой структурой добавляются часто. При изменении классов, представленных в структуре, нужно будет переопределить интерфейсы всех посетителей, а это может вызвать затруднения. Поэтому если классы меняются достаточно часто, то, вероятно, лучше определить операции прямо в них.

Структура



Участники

- Visitor - посетитель:
  - объявляет операцию V i s i t для каждого класса ConcreteElement в структуре объектов. Имя и сигнатура этой операции идентифицируют 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;
- Concrete Visitor - конкретный посетитель: реализует 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 (Node) - элемент:
  - определяет операцию Accept, которая принимает посетителя в качестве аргумента;
- ConcreteElement (AssignmentNode, VariableRefNode) – конкретный элемент:

- реализует операцию Accept, принимающую посетителя как аргумент;
- ObjectStructure (Program) - структура объектов:
  - может перечислить свои элементы;
  - может предоставить посетителю высокоуровневый интерфейс для посещения своих элементов;
  - может быть, как составным объектом (см. паттерн компоновщик), так и коллекцией, например списком или множеством.

Результаты

- Некоторые достоинства и недостатки паттерна посетитель:
- Упрощает добавление новых операций.
- Объединяет родственные операции и отсекает те, которые не имеют к ним отношения.
- Добавление новых классов ConcreteElement затруднено.
- Посещение различных иерархий классов. Итератор может посещать объекты структуры по мере ее обхода, вызывая операции объектов.
- Аккумулирование состояния
- Нарушение инкапсуляции.

Реализация паттернов

Реализация Abstract Factory

```
#include <iostream>
#include <vector>

// Абстрактные базовые классы всех возможных видов воинов
class Infantryman
{
  public:
    virtual void info() = 0;
    virtual ~Infantryman() {}
};

class Archer
{
  public:
    virtual void info() = 0;
    virtual ~Archer() {}
};

class Horseman
{
  public:
    virtual void info() = 0;
    virtual ~Horseman() {}
};


// Классы всех видов воинов Римской армии
class RomanInfantryman: public Infantryman
{
  public:
    void info() {
      cout << "RomanInfantryman" << endl;
    }
};
```

```
class RomanArcher: public Archer
{
  public:
    void info() {
      cout << "RomanArcher" << endl;
    }
};

class RomanHorseman: public Horseman
{
  public:
    void info() {
      cout << "RomanHorseman" << endl;
    }
};


// Классы всех видов воинов армии Карфагена
class CarthaginianInfantryman: public Infantryman
{
  public:
    void info() {
      cout << "CarthaginianInfantryman" << endl;
    }
};

class CarthaginianArcher: public Archer
{
  public:
    void info() {
      cout << "CarthaginianArcher" << endl;
    }
};

class CarthaginianHorseman: public Horseman
{
  public:
    void info() {
      cout << "CarthaginianHorseman" << endl;
    }
};


// Абстрактная фабрика для производства воинов
class ArmyFactory
{
  public:
    virtual Infantryman* createInfantryman() = 0;
    virtual Archer* createArcher() = 0;
    virtual Horseman* createHorseman() = 0;
    virtual ~ArmyFactory() {}
};
```

```
// Фабрика для создания воинов Римской армии
class RomanArmyFactory: public ArmyFactory
{
  public:
    Infantryman* createInfantryman() {
      return new RomanInfantryman;
    }
    Archer* createArcher() {
      return new RomanArcher;
    }
    Horseman* createHorseman() {
      return new RomanHorseman;
    }
};


// Фабрика для создания воинов армии Карфагена
class CarthaginianArmyFactory: public ArmyFactory
{
  public:
    Infantryman* createInfantryman() {
      return new CarthaginianInfantryman;
    }
    Archer* createArcher() {
      return new CarthaginianArcher;
    }
    Horseman* createHorseman() {
      return new CarthaginianHorseman;
    }
};


// Класс, содержащий всех воинов той или иной армии
class Army
{
  public:
   ~Army() {
      int i;
      for(i=0; i<vi.size(); ++i)  delete vi[i];
      for(i=0; i<va.size(); ++i)  delete va[i];
      for(i=0; i<vh.size(); ++i)  delete vh[i];
    }
    void info() {
      int i;
      for(i=0; i<vi.size(); ++i)  vi[i]->info();
      for(i=0; i<va.size(); ++i)  va[i]->info();
      for(i=0; i<vh.size(); ++i)  vh[i]->info();
    }
    vector<Infantryman*> vi;
    vector<Archer*> va;
    vector<Horseman*> vh;
};
```

```
// Здесь создается армия той или иной стороны
class Game
{
  public:
    Army* createArmy( ArmyFactory& factory ) {
      Army* p = new Army;
      p->vi.push_back( factory.createInfantryman());
      p->va.push_back( factory.createArcher());
      p->vh.push_back( factory.createHorseman());
      return p;
    }
};


int main()
{
    Game game;
    RomanArmyFactory ra_factory;
    CarthaginianArmyFactory ca_factory;

    Army * ra = game.createArmy( ra_factory);
    Army * ca = game.createArmy( ca_factory);
    cout << "Roman army:" << endl;
    ra->info();
    cout << "\nCarthaginian army:" << endl;
    ca->info();
    // ...
}
```

Реализация Builder

```
// Реализация на C++.

#include <iostream>
#include <memory>
#include <string>

// Product
class Pizza
{
private:
    std::string dough;
    std::string sauce;
    std::string topping;

public:
    Pizza() { }
    ~Pizza() { }

    void SetDough(const std::string& d) { dough = d; }
    void SetSauce(const std::string& s) { sauce = s; }
    void SetTopping(const std::string& t) { topping = t; }

    void ShowPizza()
```

```
    {
        std::cout << " Yummy !!!" << std::endl
        << "Pizza with Dough as " << dough
        << ", Sauce as " << sauce
        << " and Topping as " << topping
        << " !!! " << std::endl;
    }
};

// Abstract Builder
class PizzaBuilder
{
protected:
    std::shared_ptr<Pizza> pizza;
public:
    PizzaBuilder() {}
    virtual ~PizzaBuilder() {}
    std::shared_ptr<Pizza> GetPizza() { return pizza; }

    void createNewPizzaProduct() { pizza.reset (new Pizza); }

    virtual void buildDough()=0;
    virtual void buildSauce()=0;
    virtual void buildTopping()=0;

};

// ConcreteBuilder
class HawaiianPizzaBuilder : public PizzaBuilder
{
public:
    HawaiianPizzaBuilder() : PizzaBuilder() {}
    ~HawaiianPizzaBuilder(){}

    void buildDough() { pizza->SetDough("cross"); }
    void buildSauce() { pizza->SetSauce("mild"); }
    void buildTopping() { pizza->SetTopping("ham and pineapple"); }
};

// ConcreteBuilder
class SpicyPizzaBuilder : public PizzaBuilder
{
public:
    SpicyPizzaBuilder() : PizzaBuilder() {}
    ~SpicyPizzaBuilder() {}

    void buildDough() { pizza->SetDough("pan baked"); }
    void buildSauce() { pizza->SetSauce("hot"); }
    void buildTopping() { pizza->SetTopping("pepperoni and salami"); }
};

// Director
class Waiter
{
```

```
private:
    PizzaBuilder* pizzaBuilder;
public:
    Waiter() : pizzaBuilder(NULL) {}
    ~Waiter() { }

    void SetPizzaBuilder(PizzaBuilder* b) { pizzaBuilder = b; }
    std::shared_ptr<Pizza> GetPizza() { return pizzaBuilder->GetPizza(); }
    void ConstructPizza()
    {
        pizzaBuilder->createNewPizzaProduct();
        pizzaBuilder->buildDough();
        pizzaBuilder->buildSauce();
        pizzaBuilder->buildTopping();
    }
};

// Клиент заказывает две пиццы.
int main()
{
    Waiter waiter;

    HawaiianPizzaBuilder hawaiianPizzaBuilder;
    waiter.SetPizzaBuilder (&hawaiianPizzaBuilder);
    waiter.ConstructPizza();
    std::shared_ptr<Pizza> pizza = waiter.GetPizza();
    pizza->ShowPizza();

    SpicyPizzaBuilder spicyPizzaBuilder;
    waiter.SetPizzaBuilder(&spicyPizzaBuilder);
    waiter.ConstructPizza();
    pizza = waiter.GetPizza();
    pizza->ShowPizza();

    return EXIT_SUCCESS;
}
```

Реализация Factory Method

```
#include <iostream>
#include <string>
using namespace std;

class Product{
public:
  virtual string getName() = 0;
  virtual ~Product() {}
};

class ConcreteProductA: public Product{
public:
  string getName() {return "ConcreteProductA";}
};
```

```
class ConcreteProductB: public Product{
public:
  string getName() {return "ConcreteProductB";}
};

class Creator{
public:
  virtual Product* factoryMethod() = 0;
};

class ConcreteCreatorA: public Creator{
public:
  Product* factoryMethod() {return new ConcreteProductA();}
};

class ConcreteCreatorB: public Creator{
public:
  Product* factoryMethod() {return new ConcreteProductB();}
};

int main()
{
  static const size_t count = 2;
  ConcreteCreatorA CreatorA;
  ConcreteCreatorB CreatorB;
  // An array of creators
  Creator*creators[count] = {&CreatorA, &CreatorB};
  // Iterate over creators and create products
  for(size_t i = 0; i<count; i++){
    Product* product=creators[i]->factoryMethod();
    cout << product->getName() << endl;
    delete product;
  }
  return 0;
}
```

Реализация Prototype

```
class Meal {
public:
        virtual ~Meal();
        virtual void eat() = 0;
        virtual Meal *clone() const = 0;
        //...
};
class Spaghetti : public Meal {
public:
        Spaghetti( const Spaghetti &);
        void eat();
        Spaghetti *clone() const { return new Spaghetti( *this ); }
        //...
};
```

Реализация Singleton

```
template<typename T>
```

```
class singleton
{
public:
    static T& instance()
    {
        if(!myInstance)
            myInstance = new T;
        return *myInstance;
    }
private:
    static T* myInstance;
    singleton (const singleton&){}
};
template<typename T>
T* singleton::myInstance = NULL;
```

Реализация Adapter

```
#include <iostream>

// Уже существующий класс температурного датчика окружающей среды
class FahrenheitSensor
{
  public:
    // Получить показания температуры в градусах Фаренгейта
    float getFahrenheitTemp() {
      float t = 32.0;
      // ... какой то код
      return t;
    }
};

class Sensor
{
  public:
    virtual ~Sensor() {}
    virtual float getTemperature() = 0;
};

class Adapter : public Sensor
{
  public:
    Adapter( FahrenheitSensor* p ) : p_fsensor(p) {
    }
   ~Adapter() {
      delete p_fsensor;
    }
    float getTemperature() {
      return (p_fsensor->getFahrenheitTemp()-32.0)*5.0/9.0;
    }
  private:
    FahrenheitSensor* p_fsensor;
};
```

```
int main()
{
  Sensor* p = new Adapter( new FahrenheitSensor);
  cout << "Celsius temperature = " << p->getTemperature() << endl;
  delete p;
  return 0;
}
```

Реализация Bridge

```
// Logger.h - Абстракция
#include <string>

// Опережающее объявление
class LoggerImpl;

class Logger
{
  public:
    Logger( LoggerImpl* p );
    virtual ~Logger( );
    virtual void log( string & str ) = 0;
  protected:
    LoggerImpl * pimpl;
};

class ConsoleLogger : public Logger
{
  public:
    ConsoleLogger();
    void log( string & str );
};

class FileLogger : public Logger
{
  public:
    FileLogger( string & file_name );
    void log( string & str );
  private:
    string file;
};

class SocketLogger : public Logger
{
  public:
    SocketLogger( string & remote_host, int remote_port );
    void log( string & str );
  private:
    string host;
    int    port;
};


// Logger.cpp - Абстракция
```

```
#include "Logger.h"
#include "LoggerImpl.h"

Logger::Logger( LoggerImpl* p ) : pimpl(p)
{ }

Logger::~Logger( )
{
    delete pimpl;
}

ConsoleLogger::ConsoleLogger() : Logger(
      #ifdef MT
        new MT_LoggerImpl()
      #else
        new ST_LoggerImpl()
      #endif
    )
{ }

void ConsoleLogger::log( string & str )
{
    pimpl->console_log( str);
}

FileLogger::FileLogger( string & file_name ) : Logger(
      #ifdef MT
        new MT_LoggerImpl()
      #else
        new ST_LoggerImpl()
      #endif
    ), file(file_name)
{ }

void FileLogger::log( string & str )
{
    pimpl->file_log( file, str);
}

SocketLogger::SocketLogger( string & remote_host,
                            int remote_port ) : Logger(
      #ifdef MT
        new MT_LoggerImpl()
      #else
        new ST_LoggerImpl()
      #endif
    ), host(remote_host), port(remote_port)
{ }

void SocketLogger::log( string & str )
{
    pimpl->socket_log( host, port, str);
}
```

```
// LoggerImpl.h - Реализация
#include <string>

class LoggerImpl
{
  public:
    virtual ~LoggerImpl( ) {}
    virtual void console_log( string & str ) = 0;
    virtual void file_log(
                    string & file, string & str ) = 0;
    virtual void socket_log(
           tring & host, int port, string & str ) = 0;
};

class ST_LoggerImpl : public LoggerImpl
{
  public:
    void console_log( string & str );
    void file_log   ( string & file, string & str );
    void socket_log (
            string & host, int port, string & str );
};

class MT_LoggerImpl : public LoggerImpl
{
  public:
    void console_log( string & str );
    void file_log   ( string & file, string & str );
    void socket_log (
            string & host, int port, string & str );
};


// LoggerImpl.cpp - Реализация
#include <iostream>
#include "LoggerImpl.h"


void ST_LoggerImpl::console_log( string & str )
{
  cout << "Single-threaded console logger" << endl;
}

void ST_LoggerImpl::file_log( string & file, string & str )
{
  cout << "Single-threaded file logger" << endl;
}

void ST_LoggerImpl::socket_log(
                     string & host, int port, string & str )
{
  cout << "Single-threaded socket logger" << endl;
};
```

```
void MT_LoggerImpl::console_log( string & str )
{
  cout << "Multithreaded console logger" << endl;
}

void MT_LoggerImpl::file_log( string & file, string & str )
{
  cout << "Multithreaded file logger" << endl;
}

void MT_LoggerImpl::socket_log(
                    string & host, int port, string & str )
{
  cout << "Multithreaded socket logger" << endl;
}

// Main.cpp
#include <string>
#include "Logger.h"

int main()
{
  Logger * p = new FileLogger( string("log.txt"));
  p->log( string("message"));
  delete p;
  return 0;
}
```

Реализация Composit

```
#include <iostream>
#include <list>
#include <algorithm>
#include <memory>

class IText{

public:
        typedef std::shared_ptr<IText> SPtr;

        virtual void draw() = 0;

        virtual void add(const SPtr&) {
                throw std::runtime_error("IText: Can't add to a leaf");
        }

        virtual void remove(const SPtr&){
                throw std::runtime_error("IText: Can't remove from a leaf");
        }
};

class CompositeText: public IText{
```

```
public:
        void add(const SPtr& sptr){
                children_.push_back(sptr);
        }

        void remove(const SPtr& sptr){
                children_.remove(sptr);
        }

        void replace(const SPtr& oldValue, const SPtr& newValue){
                std::replace(children_.begin(), children_.end(), oldValue, newValue);
        }

        virtual void draw(){
                for(SPtr& sptr : children_){
                        sptr->draw();
                }
        }

private:
        std::list<SPtr> children_;
};

class Letter: public IText{
public:
        Letter(char c):c_(c) {}

        virtual void draw(){
                std::cout<<c_;
        }

private:
        char c_;
};


int main(){

        CompositeText sentence;

        IText::SPtr lSpace(new Letter(' '));
        IText::SPtr lExcl(new Letter('!'));
        IText::SPtr lComma(new Letter(','));
        IText::SPtr lNewLine(new Letter('\n'));
        IText::SPtr lH(new Letter('H')); // letter 'H'
        IText::SPtr le(new Letter('e')); // letter 'e'
        IText::SPtr ll(new Letter('l')); // letter 'l'
        IText::SPtr lo(new Letter('o')); // letter 'o'
        IText::SPtr lW(new Letter('W')); // letter 'W'
        IText::SPtr lr(new Letter('r')); // letter 'r'
        IText::SPtr ld(new Letter('d')); // letter 'd'
        IText::SPtr li(new Letter('i')); // letter 'i'

        IText::SPtr wHello(new CompositeText);
```

```
        wHello->add(lH);
        wHello->add(le);
        wHello->add(ll);
        wHello->add(ll);
        wHello->add(lo);

        IText::SPtr wWorld(new CompositeText); // word "World"
        wWorld->add(lW);
        wWorld->add(lo);
        wWorld->add(lr);
        wWorld->add(ll);
        wWorld->add(ld);

        sentence.add(wHello);
        sentence.add(lComma);
        sentence.add(lSpace);
        sentence.add(wWorld);
        sentence.add(lExcl);
        sentence.add(lNewLine);

        sentence.draw();  // prints "Hello, World!\n"

        IText::SPtr wHi(new CompositeText); // word "Hi"
        wHi->add(lH);
        wHi->add(li);

        sentence.replace(wHello, wHi);
        sentence.draw();  // prints "Hi, World!\n"

        sentence.remove(wWorld);
        sentence.remove(lSpace);
        sentence.remove(lComma);
        sentence.draw();  // prints "Hi!\n"

        return 0;
}
```

Реализация Decorator

```
#include <iostream>
#include <memory>

class IComponent {
public:
        virtual void operation() = 0;
         virtual ~IComponent(){}
};

class Component : public IComponent {
public:
        virtual void operation() {
                std::cout<<"World!"<<std::endl;
        }
};
```

```
class DecoratorOne : public IComponent {
        std::shared_ptr<IComponent> m_component;

public:
        DecoratorOne(IComponent* component): m_component(component) {}

        virtual void operation() {
                std::cout << ", ";
                m_component->operation();
        }
};

class DecoratorTwo : public IComponent {
        std::shared_ptr<IComponent> m_component;

public:
        DecoratorTwo(IComponent* component): m_component(component) {}

        virtual void operation() {
                std::cout << "Hello";
                m_component->operation();
        }
};

int main() {
        DecoratorTwo obj(new DecoratorOne(new Component()));
        obj.operation(); // prints "Hello, World!\n"

        return 0;
}
```

Реализация Façade

```
#include <iostream>
#include <cstring>

/** Абстрактный музыкант - не является обязательной составляющей паттерна, введен для
упрощения кода */
class Musician {

    const char* name;

public:
    Musician(const char* name) {
        this->name = name;
    }

    virtual ~Musician() {}

protected:
    void output(const char* text) {
        std::cout << this->name << " " << text << "." << std::endl;
    }
```

```
};

/** Конкретные музыканты */
class Vocalist: public Musician {

public:
    Vocalist(const char* name): Musician(name) {}

    void singCouplet(const int coupletNumber) {
        char* text = strdup("спел куплет №");
        strncat(text, std::to_string(coupletNumber).c_str(), 15);
        output(text);
    }

    void singChorus() {
        output("спел припев");
    }
};

class Guitarist: public Musician {

public:
    Guitarist(const char* name): Musician(name) {}

    void playCoolOpening() {
        output("начинает с крутого вступления");
    }

    void playCoolRiffs() {
        output("играет крутые риффы");
    }

    void playAnotherCoolRiffs() {
        output("играет другие крутые риффы");
    }

    void playIncrediblyCoolSolo() {
        output("выдает невероятно крутое соло");
    }

    void playFinalAccord() {
        output("заканчивает песню мощным аккордом");
    }
};

class Bassist: public Musician {

public:
    Bassist(const char* name): Musician(name) {}

    void followTheDrums() {
        output("следует за барабанами");
    }
```

```
    void changeRhythm(const char* type) {
        char* text = strdup("перешел на ритм ");
        strncat(text, type, 15);
        strncat(text, "а", 15);
        output(text);
    }

    void stopPlaying() {
        output("заканчивает играть");
    }
};

class Drummer: public Musician {

public:
    Drummer(const char* name): Musician(name) {}

    void startPlaying() {
        output("начинает играть");
    }

    void stopPlaying() {
        output("заканчивает играть");
    }
};

/** Фасад, в данном случае - знаменитая рок-группа */
class BlackSabbath {

    Vocalist* vocalist;
    Guitarist* guitarist;
    Bassist* bassist;
    Drummer* drummer;

public:

    BlackSabbath() {
        vocalist = new Vocalist("Оззи Осборн");
        guitarist = new Guitarist("Тони Айомми");
        bassist = new Bassist("Гизер Батлер");
        drummer = new Drummer("Билл Уорд");
    }

    void playCoolSong() {
        guitarist->playCoolOpening();
        drummer->startPlaying();
        bassist->followTheDrums();
        guitarist->playCoolRiffs();
        vocalist->singCouplet(1);
        bassist->changeRhythm("припев");
        guitarist->playAnotherCoolRiffs();
        vocalist->singChorus();
        bassist->changeRhythm("куплет");
        guitarist->playCoolRiffs();
```

```
        vocalist->singCouplet(2);
        bassist->changeRhythm("припев");
        guitarist->playAnotherCoolRiffs();
        vocalist->singChorus();
        bassist->changeRhythm("куплет");
        guitarist->playIncrediblyCoolSolo();
        guitarist->playCoolRiffs();
        vocalist->singCouplet(3);
        bassist->changeRhythm("припев");
        guitarist->playAnotherCoolRiffs();
        vocalist->singChorus();
        bassist->changeRhythm("куплет");
        guitarist->playCoolRiffs();
        bassist->stopPlaying();
        drummer->stopPlaying();
        guitarist->playFinalAccord();
    }
};

int main() {
    std::cout << "OUTPUT:" << std::endl;
    BlackSabbath* band = new BlackSabbath();
    band->playCoolSong();
    return 0;
}

/**
 * OUTPUT:
 * Тони Айомми начинает с крутого вступления.
 * Билл Уорд начинает играть.
 * Гизер Батлер следует за барабанами.
 * Тони Айомми играет крутые риффы.
 * Оззи Осборн спел куплет №1.
 * Гизер Батлер перешел на ритм припева.
 * Тони Айомми играет другие крутые риффы.
 * Оззи Осборн спел припев.
 * Гизер Батлер перешел на ритм куплета.
 * Тони Айомми играет крутые риффы.
 * Оззи Осборн спел куплет №2.
 * Гизер Батлер перешел на ритм припева.
 * Тони Айомми играет другие крутые риффы.
 * Оззи Осборн спел припев.
 * Гизер Батлер перешел на ритм куплета.
 * Тони Айомми выдает невероятно крутое соло.
 * Тони Айомми играет крутые риффы.
 * Оззи Осборн спел куплет №3.
 * Гизер Батлер перешел на ритм припева.
 * Тони Айомми играет другие крутые риффы.
 * Оззи Осборн спел припев.
 * Гизер Батлер перешел на ритм куплета.
 * Тони Айомми играет крутые риффы.
 * Гизер Батлер заканчивает играть.
 * Билл Уорд заканчивает играть.
 * Тони Айомми заканчивает песню мощным аккордом.
```

```
 */
```

Реализация Proxy

```
/**
 * "Subject"
 */
class IMath
{
public:
    virtual double add(double, double) = 0;
    virtual double sub(double, double) = 0;
    virtual double mul(double, double) = 0;
    virtual double div(double, double) = 0;
};

/**
 * "Real Subject"
 */
class Math : public IMath
{
public:
    virtual double add(double x, double y)
    {
        return x + y;
    }

    virtual double sub(double x, double y)
    {
        return x - y;
    }

    virtual double mul(double x, double y)
    {
        return x * y;
    }

    virtual double div(double x, double y)
    {
        return x / y;
    }
};

/**
 * "Proxy Object"
 */
class MathProxy : public IMath
{
public:
    MathProxy()
    {
        math = new Math();
    }
    virtual ~MathProxy()
```

```
    {
        delete math;
    }
    virtual double add(double x, double y)
    {
        return math->add(x, y);
    }

    virtual double sub(double x, double y)
    {
        return math->sub(x, y);
    }

    virtual double mul(double x, double y)
    {
        return math->mul(x, y);
    }

    virtual double div(double x, double y)
    {
        return math->div(x, y);
    }

private:
    IMath *math;
};

#include <iostream>

using std::cout;
using std::endl;

int main()
{
    // Create math proxy
    IMath *proxy = new MathProxy();

    // Do the math
    cout << "4 + 2 = " << proxy->add(4, 2) << endl;
    cout << "4 - 2 = " << proxy->sub(4, 2) << endl;
    cout << "4 * 2 = " << proxy->mul(4, 2) << endl;
    cout << "4 / 2 = " << proxy->div(4, 2) << endl;

    delete proxy;
    return 0;
}
```

Реализация Pool

```
#include <vector>

class Object
{
        // ...
```

```
};


class ObjectPool
{
        private:
                struct PoolRecord
                {
                        Object* instance;
                        bool    in_use;
                };

                std::vector<PoolRecord> m_pool;

        public:
                Object* createNewObject()
                {
                        for (size_t i = 0; i < m_pool.size(); ++i)
            {
                                if (! m_pool[i].in_use)
                                {
                                        m_pool[i].in_use = true; // переводим объект в
список используемых
                                        return m_pool[i].instance;
                                }
            }
                        // если не нашли свободный объект, то расширяем пул
                        PoolRecord record;
                        record.instance = new Object;
                        record.in_use   = true;

                        m_pool.push_back(record);

                        return record.instance;
                }

                void deleteObject(Object* object)
                {
                        // в реальности не удаляем, а лишь помечаем, что объкт свободен
                        for (size_t i = 0; i < m_pool.size(); ++i)
            {
                                if (m_pool[i].instance == object)
                                {
                                        m_pool[i].in_use = false;
                                        break;
                                }
            }
                }

                virtual ~ObjectPool()
                {
                        // теперь уже "по-настоящему" удаляем объекты
                        for (size_t i = 0; i < m_pool.size(); ++i)
                                delete m_pool[i].instance;
```

```
                }
};


int main()
{
        ObjectPool pool;
        for (size_t i = 0; i < 1000; ++i)
        {
                Object* object = pool.createNewObject();
                // ...
                pool.deleteObject(object);
        }
        return 0;
}
```

Реализация Chain of Responsibility

```
#include <iostream>

/**
 * Вспомогательный класс, описывающий некоторое преступление
 */
class CriminalAction {

    friend class Policeman;      // Полицейские имеют доступ к материалам следствия

    int complexity;              // Сложность дела

    const char* description;     // Краткое описание преступления

public:
    CriminalAction(int complexity, const char* description): complexity(complexity),
description(description) {}

};

/**
 * Абстрактный полицейский, который может заниматься расследованием преступлений
 */
class Policeman {

protected:

    int deduction;       // дедукция (умение распутывать сложные дела) у данного
полицейского

    Policeman* next;     // более умелый полицейский, который получит дело, если для
текущего оно слишком сложное

    virtual void investigateConcrete(const char* description) {}     // собственно
расследование

public:
```

```
    Policeman(int deduction): deduction(deduction) {}

    virtual ~Policeman() {
        if (next) {
            delete next;
        }
    }

    /**
     * Добавляет в цепочку ответственности более опытного полицейского, который сможет принять на себя
     * расследование, если текущий не справится
     */
    Policeman* setNext(Policeman* policeman) {
        next = policeman;
        return next;
    }

    /**
     * Полицейский начинает расследование или, если дело слишком сложное, передает ее более опытному коллеге
     */
    void investigate(CriminalAction* criminalAction) {
        if (deduction < criminalAction->complexity) {
            if (next) {
                next->investigate(criminalAction);
            } else {
                std::cout << "Это дело не раскрыть никому." << std::endl;
            }
        } else {
            investigateConcrete(criminalAction->description);
        }
    }
};

class MartinRiggs: public Policeman {

protected:

    void investigateConcrete(const char* description) {
        std::cout << "Расследование по делу \"" << description << "\" ведет сержант Мартин Риггс" << std::endl;
    }

public:

    MartinRiggs(int deduction): Policeman(deduction) {}
};

class JohnMcClane: public Policeman {

protected:
```

```
    void investigateConcrete(const char* description) {
        std::cout << "Расследование по делу \"" << description << "\" ведет детектив Джон Макклейн" << std::endl;
    }

public:
    JohnMcClane(int deduction): Policeman(deduction) {}
};

class VincentHanna: public Policeman {

protected:

    void investigateConcrete(const char* description) {
        std::cout << "Расследование по делу \"" << description << "\" ведет лейтенант Винсент Ханна" << std::endl;
    }

public:
    VincentHanna(int deduction): Policeman(deduction) {}
};

int main() {
    std::cout << "OUTPUT:" << std::endl;
    Policeman* policeman = new MartinRiggs(3);  // полицейский с наименьшим навыком ведения расследований
    policeman
            ->setNext(new JohnMcClane(5))
            ->setNext(new VincentHanna(8));     // добавляем ему двух опытных коллег
    policeman->investigate(new CriminalAction(2, "Торговля наркотиками из Вьетнама"));
    policeman->investigate(new CriminalAction(7, "Дерзкое ограбление банка в центре Лос-Анджелеса"));
    policeman->investigate(new CriminalAction(5, "Серия взрывов в центре Нью-Йорка"));
    return 0;
}

/**
 * OUTPUT:
 * Расследование по делу "Торговля наркотиками из Вьетнама" ведет сержант Мартин Риггс
 * Расследование по делу "Дерзкое ограбление банка в центре Лос-Анджелеса" ведет лейтенант Винсент Ханна
 * Расследование по делу "Серия взрывов в центре Нью-Йорка" ведет детектив Джон Макклейн
 */
```

Реализация Command

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

class Document
{
```

```
    vector<string> data;
public:
    void Insert( int line, const string & str )
    {
        if ( line <= data.size() )
            data.insert( data.begin() + line, str );
        else
            cout << "Error!" << endl;
    }

    void Remove( int line )
    {
        if( !( line>data.size() ) )
            data.erase( data.begin() + line );
        else
            cout << "Error!" << endl;
    }

    string & operator [] ( int x )
    {
        return data[x];
    }

    void Show()
    {
        for( int i = 0; i<data.size(); ++i )
        {
            cout << i + 1 << ". " << data[i] << endl;
        }
    }
};

class Command
{
protected:
    Document * doc;
public:
    virtual void Execute() = 0;
    virtual void unExecute() = 0;

    void setDocument( Document * _doc )
    {
        doc = _doc;
    }
};

class InsertCommand : public Command
{
    int line;
    string str;
public:
    InsertCommand( int _line, const string & _str ): line( _line ), str( _str ) {}

    void Execute()
```

```
    {
        doc->Insert( line, str );
    }

    void unExecute()
    {
        doc->Remove( line );
    }
};

class Receiver
{
    vector<Command*> DoneCommands;
    Document doc;
    Command* command;
public:
    void Insert( int line, string str )
    {
        command = new InsertCommand( line, str );
        command->setDocument( &doc );
        command->Execute();
        DoneCommands.push_back( command );
    }

    void Undo()
    {
        if( DoneCommands.size() == 0 )
        {
            cout << "There is nothing to undo!" << endl;
        }
        else
        {
            command = DoneCommands.back();
            DoneCommands.pop_back();
            command->unExecute();
            // Don't forget to delete command!!!
            delete command;
        }
    }

    void Show()
    {
        doc.Show();
    }
};

int main()
{
    char s = '1';
    int line, line_b;
    string str;
    Receiver res;
    while( s!= 'e' )
    {
```

```
        cout << "What to do: \n1.Add a line\n2.Undo last command" << endl;
        cin >> s;
        switch( s )
        {
        case '1':
           cout << "What line to insert: ";
           cin >> line;
           --line;
           cout << "What to insert: ";
           cin >> str;
           res.Insert( line, str );
           break;
        case '2':
           res.Undo();
           break;
        }
        cout << "$$$DOCUMENT$$$" << endl;
        res.Show();
        cout << "$$$DOCUMENT$$$" << endl;
    }
}
```

Реализация Mediator(посредник)

```
#include <iostream>
#include <string>

class Colleague;
class Mediator;
class ConcreteMediator;
class ConcreteColleague1;
class ConcreteColleague2;

class Mediator
{
public:
        virtual void Send(std::string const& message, Colleague *colleague) const = 0;
};

class Colleague
{
protected:
        Mediator* mediator_;

public:
        explicit Colleague(Mediator *mediator):mediator_(mediator)
        {
        }
};

class ConcreteColleague1:public Colleague
{
public:
        explicit ConcreteColleague1(Mediator* mediator):Colleague(mediator)
```

```
        {
        }

        void Send(std::string const& message)
        {
                mediator_->Send(message, this);
        }

        void Notify(std::string const& message)
        {
                std::cout << "Colleague1 gets message '" << message << "'" <<
std::endl;
        }
};

class ConcreteColleague2:public Colleague
{
public:
        explicit ConcreteColleague2(Mediator *mediator):Colleague(mediator)
        {
        }

        void Send(std::string const& message)
        {
                mediator_->Send(message, this);
        }

        void Notify(std::string const& message)
        {
                std::cout << "Colleague2 gets message '" << message << "'" <<
std::endl;
        }
};

class ConcreteMediator:public Mediator
{
protected:
        ConcreteColleague1      *m_Colleague1;
        ConcreteColleague2      *m_Colleague2;
public:
        void SetColleague1(ConcreteColleague1 *c)
        {
                m_Colleague1=c;
        }

        void SetColleague2(ConcreteColleague2 *c)
        {
                m_Colleague2=c;
        }

        virtual void Send(std::string const& message, Colleague *colleague) const
        {
                if (colleague==m_Colleague1)
                {
```

```
                        m_Colleague2->Notify(message);
                }
                else if (colleague==m_Colleague2)
                {
                        m_Colleague1->Notify(message);
                }
        }
};

int main()
{
        ConcreteMediator m;

        ConcreteColleague1 c1(&m);
        ConcreteColleague2 c2(&m);

        m.SetColleague1(&c1);
        m.SetColleague2(&c2);

        c1.Send("How are you?");
        c2.Send("Fine, thanks");

        std::cin.get();
        return 0;
}

Output
Colleague2 gets message 'How are you?'
Colleague1 gets message 'Fine, thanks'
```

Реализация Memento(хранитель)

```
#include <iostream.h>
class Number;

class Memento
{
  public:
    Memento(int val)
    {
        _state = val;
    }
  private:
    friend class Number;
    int _state;
};

class Number
{
  public:
    Number(int value)
    {
        _value = value;
    }
```

```
    void dubble()
    {
        _value = 2 * _value;
    }
    void half()
    {
        _value = _value / 2;
    }
    int getValue()
    {
        return _value;
    }
    Memento *createMemento()
    {
        return new Memento(_value);
    }
    void reinstateMemento(Memento *mem)
    {
        _value = mem->_state;
    }
  private:
    int _value;
};

class Command
{
  public:
    typedef void(Number:: *Action)();
    Command(Number *receiver, Action action)
    {
        _receiver = receiver;
        _action = action;
    }
    virtual void execute()
    {
        _mementoList[_numCommands] = _receiver->createMemento();
        _commandList[_numCommands] = this;
        if (_numCommands > _highWater)
          _highWater = _numCommands;
        _numCommands++;
        (_receiver-> *_action)();
    }
    static void undo()
    {
        if (_numCommands == 0)
        {
            cout << "*** Attempt to run off the end!! ***" << endl;
            return ;
        }
           _commandList[_numCommands - 1]->_receiver->reinstateMemento
          (_mementoList[_numCommands - 1]);
        _numCommands--;
    }
    void static redo()
```

```
    {
        if (_numCommands > _highWater)
        {
            cout << "*** Attempt to run off the end!! ***" << endl;
            return ;
        }
  (_commandList[_numCommands]->_receiver->*(_commandList[_numCommands]
          ->_action))();
        _numCommands++;
    }
  protected:
    Number *_receiver;
    Action _action;
    static Command *_commandList[20];
    static Memento *_mementoList[20];
    static int _numCommands;
    static int _highWater;
};

Command *Command::_commandList[];
Memento *Command::_mementoList[];
int Command::_numCommands = 0;
int Command::_highWater = 0;

int main()
{
  int i;
  cout << "Integer: ";
  cin >> i;
  Number *object = new Number(i);

  Command *commands[3];
  commands[1] = new Command(object, &Number::dubble);
  commands[2] = new Command(object, &Number::half);

  cout << "Exit[0], Double[1], Half[2], Undo[3], Redo[4]: ";
  cin >> i;

  while (i)
  {
    if (i == 3)
      Command::undo();
    else if (i == 4)
      Command::redo();
    else
      commands[i]->execute();
    cout << "    " << object->getValue() << endl;
    cout << "Exit[0], Double[1], Half[2], Undo[3], Redo[4]: ";
    cin >> i;
  }
}
```

Реализация Observer(Наблюдатель)

```
#include <iostream>
```

```cpp
#include <string>
#include <list>

using namespace std;

class SupervisedString;
class IObserver
{
public:
    virtual void handleEvent(const SupervisedString&) = 0;
};

class SupervisedString // Observable class
{
    string _str;
    list<IObserver*> _observers;

    void _Notify()
    {
        for(auto iter : _observers)
        {
            iter->handleEvent(*this);
        }
    }

public:
    void add(IObserver& ref)
    {
        _observers.push_back(&ref);
    }

    void remove(IObserver& ref)
    {
        _observers.remove(&ref);
    }

    const string& get() const
    {
        return _str;
    }

    void reset(string str)
    {
        _str = str;
        _Notify();
    }
};

class Reflector: public IObserver // Prints the observed string into cout
{
public:
    virtual void handleEvent(const SupervisedString& ref)
    {
        cout << ref.get() << endl;
```

```
    }
};

class Counter: public IObserver // Prints the length of observed string into cout
{
public:
  virtual void handleEvent(const SupervisedString& ref)
  {
      cout << "length = " << ref.get().length() << endl;
  }
};

int main()
{
    SupervisedString str;
    Reflector refl;
    Counter cnt;

    str.add(refl);
    str.reset("Hello, World!");
    cout << endl;

    str.remove(refl);
    str.add(cnt);
    str.reset("World, Hello!");
    cout << endl;

    return 0;
}
```

Реализация Strategy

```
#include <iostream>

class Strategy
{
public:
        virtual ~Strategy() {}
        virtual void use(void) = 0;
};

class Strategy_1: public Strategy
{
public:
        void use(void){ std::cout << "Strategy_1" << std::endl; };
};

class Strategy_2: public Strategy
{
public:
        void use(void){ std::cout << "Strategy_2" << std::endl; };
};

class Strategy_3: public Strategy
```

```
{
public:
        void use(void){ std::cout << "Strategy_3" << std::endl; };
};

class Context
{
protected:
        Strategy* operation;

public:
        virtual ~Context() {}
        virtual void useStrategy(void) = 0;
        virtual void setStrategy(Strategy* v) = 0;
};

class Client: public Context
{
public:
        void useStrategy(void)
        {
                operation->use();
        }

        void setStrategy(Strategy* o)
        {
                operation = o;
        }
};

int main(int /*argc*/, char* /*argv*/[])
{
        Client customClient;
        Strategy_1 str1;
        Strategy_2 str2;
        Strategy_3 str3;

        customClient.setStrategy(&str1);
        customClient.useStrategy();
        customClient.setStrategy(&str2);
        customClient.useStrategy();
        customClient.setStrategy(&str3);
        customClient.useStrategy();

        return 0;
}
```

Реализация Visitor(посетитель)

```
#include <iostream>
#include <string>

template<typename TYPE,size_t COUNT>
inline size_t lenof(TYPE(&)[COUNT]){return COUNT;}

class Foo;
```

```
class Bar;
class Baz;

class Visitor{
public:
  virtual void visit(Foo&ref)=0;
  virtual void visit(Bar&ref)=0;
  virtual void visit(Baz&ref)=0;
};

class Element{
public:
  virtual void accept(Visitor&v)=0;
};

class Foo:public Element{
public:
  void accept(Visitor&v){v.visit(*this);}
};

class Bar:public Element{
public:
  void accept(Visitor&v){v.visit(*this);}
};

class Baz:public Element{
public:
  void accept(Visitor&v){v.visit(*this);}
};

class GetType:public Visitor{
public:
  std::string value;
public:
  void visit(Foo&ref){value="Foo";}
  void visit(Bar&ref){value="Bar";}
  void visit(Baz&ref){value="Baz";}
};

int main()
{
  Foo foo; Bar bar; Baz baz;
  Element*elements[]={&foo,&bar,&baz};
  for(size_t i=0;i<lenof(elements);i++)
  {
    GetType visitor;
    elements[i]->accept(visitor);
    std::cout<<visitor.value<<std::endl;
  }
  return 0;
}
```

Реализация State

```
#include <iostream>
using namespace std;
```

```
class Machine
{
  class State *current;
  public:
    Machine();
    void setCurrent(State *s)
    {
        current = s;
    }
    void on();
    void off();
};

class State
{
  public:
    virtual void on(Machine *m)
    {
        cout << "    already ON\n";
    }
    virtual void off(Machine *m)
    {
        cout << "    already OFF\n";
    }
};

void Machine::on()
{
  current->on(this);
}

void Machine::off()
{
  current->off(this);
}

class ON: public State
{
  public:
    ON()
    {
        cout << "    ON-ctor ";
    };
    ~ON()
    {
        cout << "    dtor-ON\n";
    };
    void off(Machine *m);
};

class OFF: public State
{
  public:
    OFF()
```

```
    {
        cout << "   OFF-ctor ";
    };
    ~OFF()
    {
        cout << "   dtor-OFF\n";
    };
    void on(Machine *m)
    {
        cout << "   going from OFF to ON";
        m->setCurrent(new ON());
        delete this;
    }
};

void ON::off(Machine *m)
{
  cout << "   going from ON to OFF";
  m->setCurrent(new OFF());
  delete this;
}

Machine::Machine()
{
  current = new OFF();
  cout << '\n';
}

int main()
{
  void(Machine:: *ptrs[])() =
  {
    Machine::off, Machine::on
  };
  Machine fsm;
  int num;
  while (1)
  {
    cout << "Enter 0/1: ";
    cin >> num;
    (fsm. *ptrs[num])();
  }
}
```

Реализация Publish-Subscribe

```
using namespace System;
delegate void Eventhandler(Object^ Source,double);
public ref class Manager
{
public:
    event Eventhandler^ OnHandler;
    void Method1()
    {
        double value = 10;
        onHandler (this,value);
```

```
    }
};

public ref class Watcher
{
public:
    Watcher(Manader^ m)
    {
        m->OnHandler += gcnew Eventhandler(this,&Watcher::f);
    }
    void f(Object^ source, double a){}
};

void main()
{
    Manader^ m = gcnew Manager();
    Watcher^ w = gcnew Watcher(m);
    m->Method1();
}
```