Паттерны проектирования

Паттерн — это в переводе с английского — это «узор». В программировании, его можно переводить как «образец» или «шаблон».

Паттерн описывает задачи, которые возникают в работе программиста раз за разом, чтобы не решать их каждый раз заново.

Паттерн проектирования — это описание взаимодействия объектов и классов адаптированных для решения общей задачи проектирования в конкретном контексте.

Идиомы — это паттерны, описывающие типичные решения на конкретном языке программирования, в то время, как паттерны проектирования от языка не зависят.

В общем случае, паттерн содержит:

1. **Имя** — уникальный идентификатор паттерна. Может использоваться как ссылка на типичное решение.
2. **Задача** — это ситуация, в которой можно применять паттерны. При формулировки задачи, важно описать контекст, в котором паттерн возникает.
3. **Решение** задачи проектирования определяет общие функции каждого элемента дизайна. Рассматривается именно общая ситуация.
4. **Результаты —** это следствия применения паттерна. В них описываются преимущества и недостатки паттерна, компромиссы и вариации.

Антипаттерн — это часто повторяемое плохое решение, которое не рекомендуется использовать.

Признаки плохого кода и дизайна

- Дублирование кода
- Большие методы
- Большие классы
- Чрезмерная зависимость
- Нарушения приватности
- Нарушение завещания
- Ленивый класс
- Чрезмерная сложность
- Длинные имена

Классификации паттернов

1. **Архитектурные**
2. **Проектирования**

3. **Анализа**
4. **Тестирования**
5. **Реализации**

Паттерны проектирования

1. Основные (фундаментальные) паттерны — это наиболее важные паттерны, используемые другими.
2. Порождающие паттерны — определяют способы создания объектов
3. Структурные паттерны — описывают сложные структуры
4. Поведенческие паттерны — способы взаимодействия между объектами
5. Паттерны параллельного программирования — координируют параллельные операции, решая проблемы борьбы за ресурсы и взаимоблокировок
6. Паттерны MVC – описывают Модель-Вид-Контроллер
7. Паттерны корпоративных систем — решения для построения больших корпоративных систем

Среди 23 паттернов Банды Четырёх описаны Порождающие, Структурные и Поведенческие.

Порождающие паттерны

Абстрагируют процесс инстанцирования. Они помогают сделать систему независимой от способа создания, композиции и представления объектов. Отвечают на вопрос «Кто и как создаёт объекты в системе?»

Абстрактная фабрика

Abstract Factory, также известен, как Toolkit/Инструментарий

Цель: предоставляет интерфейс для создания семейства взаимосвязанных и взаимозависимых объектов, не идентифицируя конкретные классы.

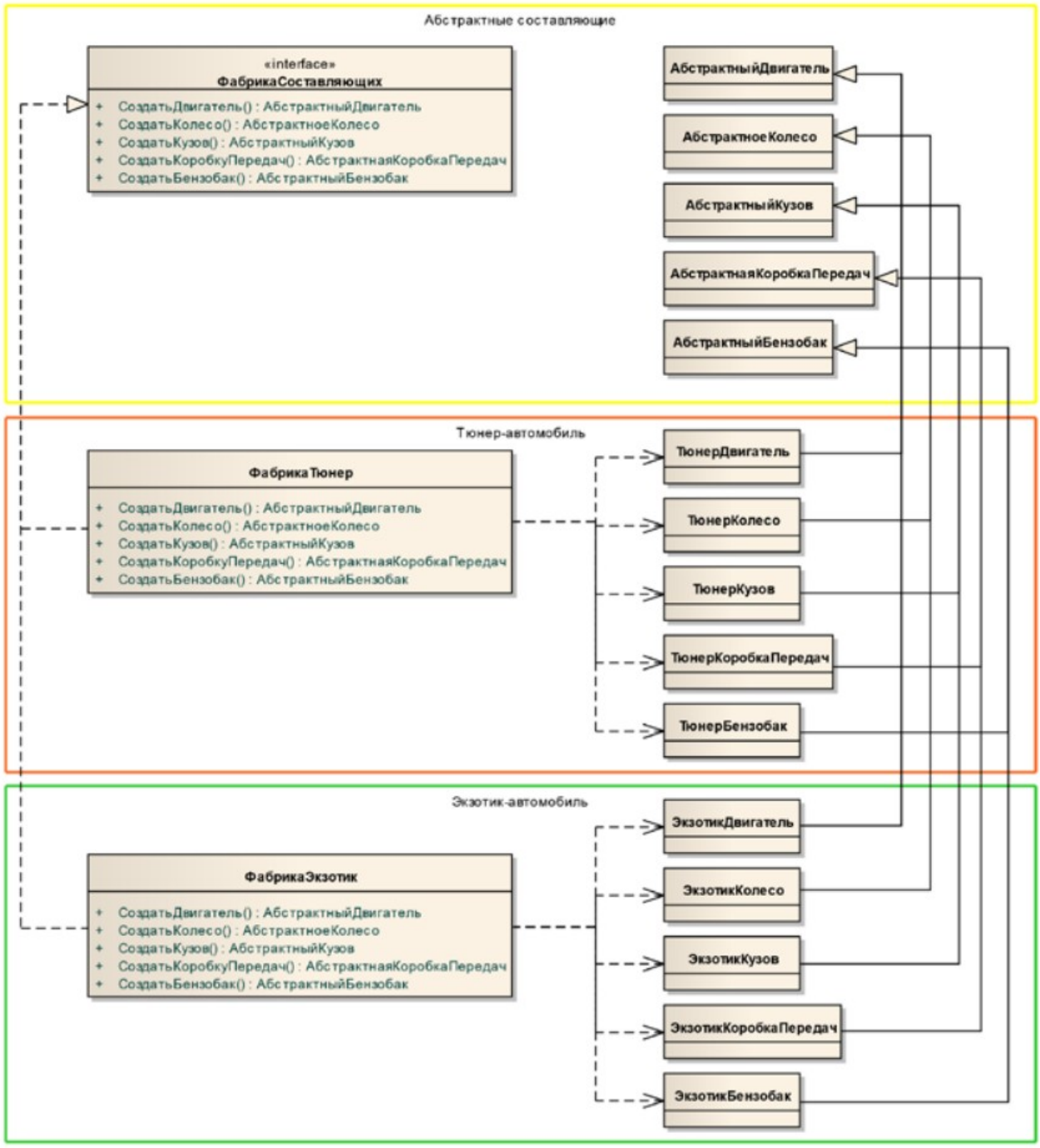
Используется, когда:

- Система не должна зависеть от того, как в ней создаются объекты

- Объекты в одном семействе используются вместе

- Система конфигурируется одним из семейств объектов

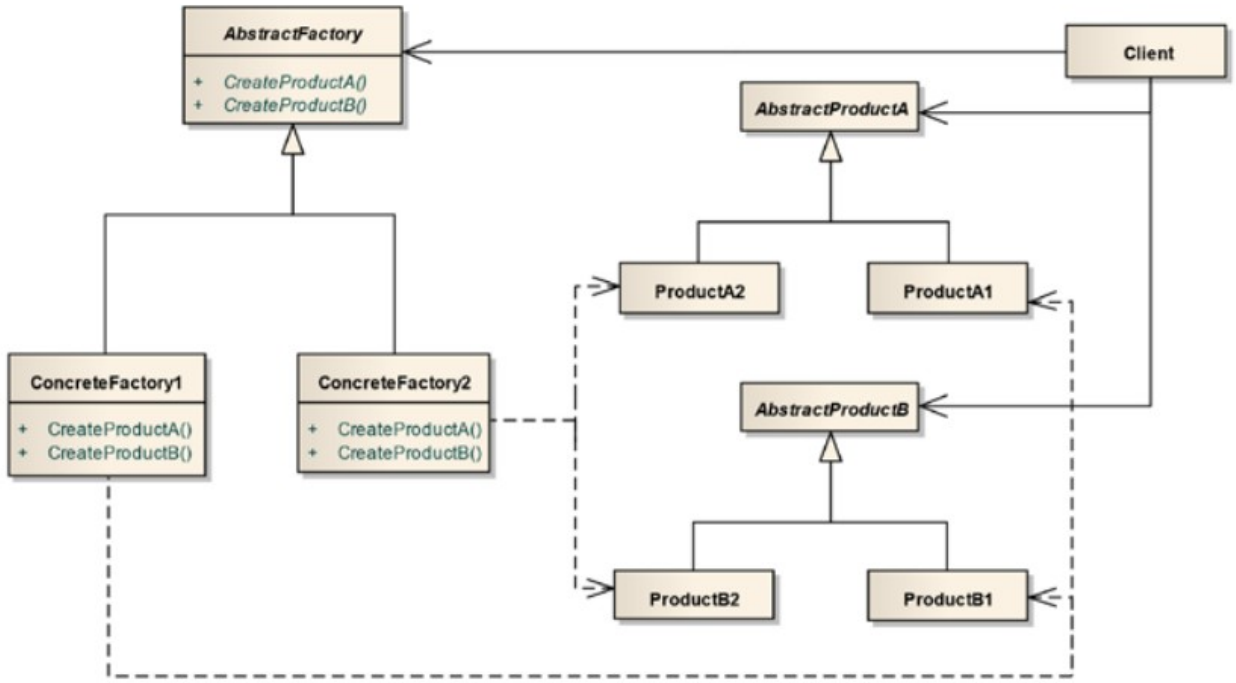
- Надо предоставить интерфейс библиотеки, не раскрывая её реализации



Существенным недостатком является то, что ФабрикаСоставляющих на этапе компиляции предусматривает создание фиксированного набора продуктов, следовательно — количество и тип продуктов жёстко определены на этапе компиляции, а при необходимости добавить новый продукт, необходимо изменить абстрактную фабрику.

Во избежание таких проблем можно рассмотреть использование паттерна прототип.

Саму фабрику можно создавать, как singleton.



Общая структура паттерна

***Абстрактная фабрика*, которая предоставляет общий интерфейс для создания семейства продуктов.**

***Конкретная фабрика*, которая реализует интерфейс Абстрактной фабрики и создаёт семейство конкретных продуктов.**

***Абстрактный продукт* — предоставляет интерфейс продукта, ссылку на который возвращают методы фабрик.**

***Конкретный продукт* — реализует тип продукта.**

Отношения:

Клиент знает только о существовании абстрактной фабрики и её абстрактных продуктах.

Для создания семейства конкретных продуктов клиент конфигурируется соответствующим экземпляром конкретной фабрики.

Методы конкретных фабрик создают экземпляры конкретных продуктов, возвращая ссылки на абстрактные продукты.

Результаты:

Изоляция конкретных классов продуктов.

Упрощает замену семейств продуктов.

Гарантия сочетаемости продуктов.

Недостаток паттерна: трудность поддержки нового вида продуктов — необходимо изменить всю иерархию фабрик и их код.

Builder

Строитель

Цель: отделяет процесс конструирования сложного объекта от его представления, так что в результате одного и того же процесса конструирования получаются разные представления.

Клиент может создавать сложные объекты, определяя не только его тип, но и содержимое, не зная при этом о деталях конструирования.

Паттерн используют, когда:

- Общий алгоритм построения сложного объекта не должен зависеть от специфики каждого шага

- В результате одного и того же процесса надо получить разные продукты

Причины возникновения

Допустим есть конвейер для производства автомобилей.

Шаги построения автомобиля:

1. Сборка кузова

2. Установка двигателя

3. Установка колёс

4. Покраска

5. Подготовка салона

Допустим на заводе производят автомобили «мини», спортивные и внедорожники.

В контексте ООП, это описывается так:

Класс Конвейер, который является прототипов реального конвейера.

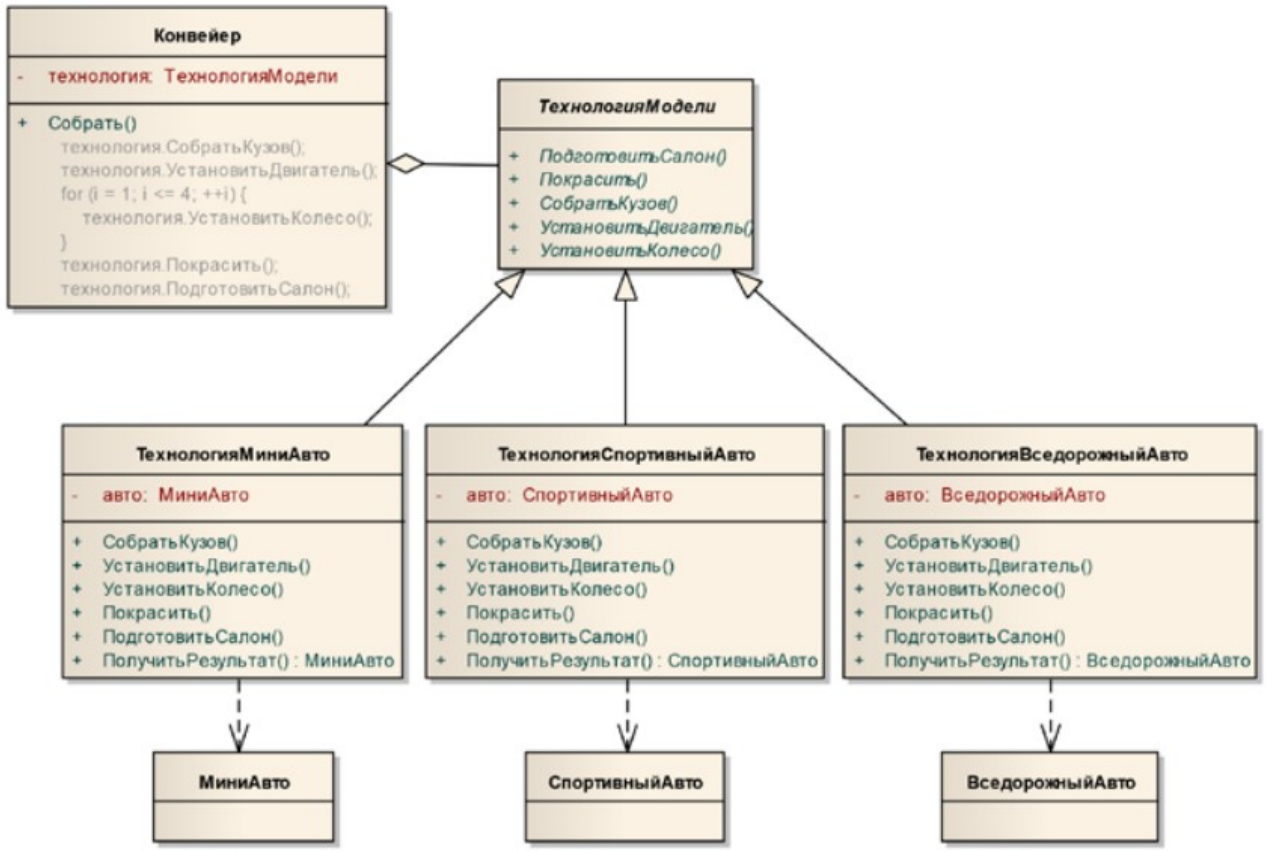
Метод Собрать исполняет процесс посредством выполнения этих шагов вне зависимости от технических деталей.

Ответственность за реализацию шагов несёт класс ТехнологияМодели.

Применяя различные подклассы мы получаем разные модели автомобилей. То есть, от него наследованы классы для каждого типа.

Для производства автомобиля, надо задать конкретную технологию и вызвать метод Собрать.

После завершения процесса сборка, автомобить можно получить с помощью метода «ПолучитьРезультат()».



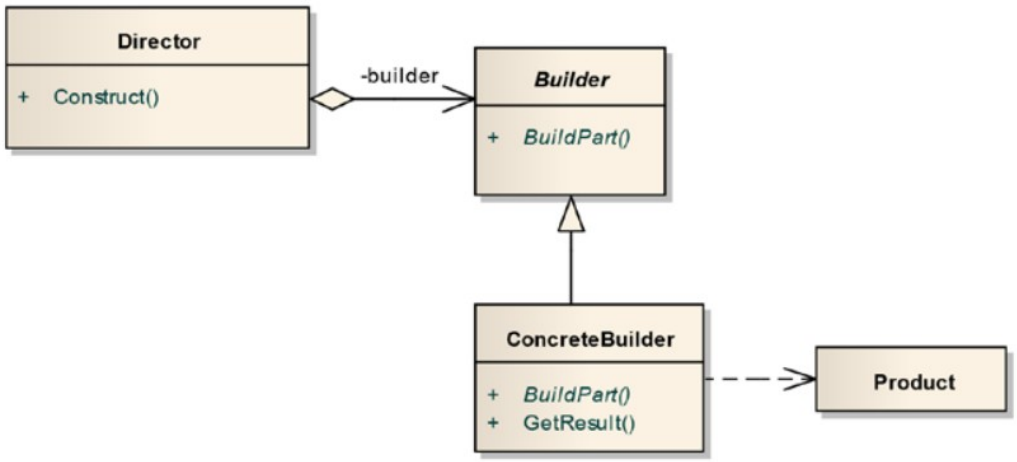
Преимущества модели:

1) конкретная технология конструирования строится по общему шаблону

2) общий алгоритм процесса не зависит от деталей конкретной технологии

3) есть возможность реализовать под общий алгоритм большое количество технологий

Структура паттерна



Builder/Строитель

Обеспечивает интерфейс для пошагового конструирования сложного объекта

ConcreteBuilder/Конкретный строитель

Реализует шаги построения сложного объекта

Создаёт результат построения

Определяет доступ к этому результату

Director/Распорядитель

Определяет общий алгоритм конструирования, используя для реализации шаги из Builder

Product/Продукт

Сложный объект, полученный в результате конструирования.

Отношения между участниками:

- Клиент конфигурирует Распорядителя.

- Распорядитель вызывает методы Строителя.

- Конкретный строитель создаёт продукт и следит за его конструированием.

- Конкретный строитель предоставляет интерфейс для доступа к продукту.

Результаты

Есть возможность изменять внутреннюю структуру объектов или создавать новые.

Разделение Распорядителя и Строителя повышает модульность ПО.

Пошаговое построение продукта позволяет обеспечить более точный контроль над процессом создания.

Factory

Фабрика/Виртуальный конструктор

Цель паттерна

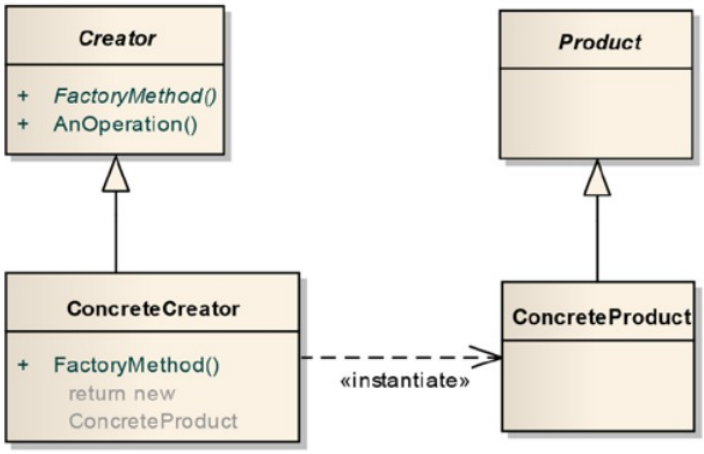
Предоставляет интерфейс для создания объектов, но оставляет за своими подклассами решение, какой объект надо создавать.

Паттерн используют, когда:

Класс не должен зависеть от конкретного типа создаваемого продукта.

Классу не известен конкретный тип продукта, который ему нужен.

Конкретные типы продуктов описаны в подклассах продукта.



Паттерн включает:

Creator/Абстрактный создатель

Предоставляет абстрактный метод для создания экземпляра продукта FactoryMethod().

ConcreteCreator/Конкретный создатель

Реализует метод создания экземпляра продукта.

Product/Абстрактный продукт

Предоставляет абстрактный интерфейс продукта, через который с ним работает Creator.

ConcreteProduct/Конкретный продукт

Реализует интерфейс продукта.

Отношение между участниками:

Creator предоставляет абстрактный интерфейс FactoryMethod() для создания экземпляра продукта.

Creator возлагает ответственность за создание конкретных экземпляров на свои подклассы.

Подклассы, такие как ConcreteCreator() реализуют этот метод, создавая экземпляры класса ConcreteProduct().

Результаты использования паттерна

Класс Создателя не зависит от конкретного типа создаваемых продуктов.

За создание продукта отвечают подклассы.

Позволяет объединить две параллельные иерархии создателей и продуктов.

Возможно определение реализации фабричного метода по умолчанию.

Конкретный класс создаваемого продукта может передаваться, как тип-параметр класса Создателя или его подклассов.

Prototype

Прототип

Цель паттерна

Определяет виды создаваемых объектов с помощью экземпляра-прототипа и создаёт новые объекты путём копирования прототипа.

Используется когда:

Клиентский код должен создавать объекты, ничего не зная об их классе.

Классы создаваемых объектов определяются динамически во время выполнения.

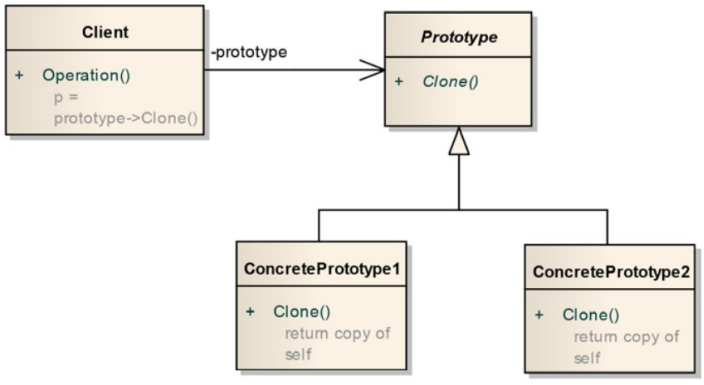
Экземпляры класса не могут пребывать в большом количестве состояний, поэтому может оказаться удобнее создать несколько прототипов и клонировать их вместо прямого создания экземпляра класса.

Prototype – прототип

Определяет интерфейс для клонирования самого себя (в C# есть встроенный интерфейс ICloneable)

ConcretePrototype – конкретный прототип

Реализует операцию клонирования самого себя.



Client – клиент

Создаёт новые объекты, запрашивая у прототипов операцию клонирования самого себя.

Отношения между участниками

Клиентский код обращается к прототипу с просьбой создать копию себя.

Результаты

Добавление/удаления новых типов продуктов во время выполнения.

Определение новых типов продуктов без необходимости наследования.

Использование диспетчера прототипов.

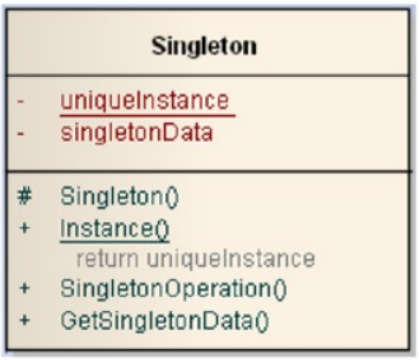
Отсутствие параметров в методе Clone() обеспечивает максимально общий интерфейс.

Singleton

Одиночка

Цель паттерна

Гарантирует, что у класса есть только один экземпляр и единая точка доступа к этому экземпляру.



Используется, когда

Должен быть только один экземпляр заданного класса, доступный всему коду.

1. Создать экземпляр класса может только сам класс.
2. Доступ к экземпляру может предоставить только этот класс.
3. Предусмотреть возможность модификации поведения экземпляра через наследование, чтобы при подмене его, использующие его классы не требовали модификаций.

Результаты

Контроль над доступом клиента к экземпляру.

Экземпляр класса имеет больше возможностей, чем обычная глобальная переменная.

Поведение можно уточнять через наследование.

Модификацией кода Instance() можно создавать более одного экземпляра.

Использование глобальных интерфейсов и одиночек может привести к появление «лгущих» интерфейсов — то есть, не показывающих явно зависимость от каких-то других интерфейсов.

При использовании в многопоточных системах надо помнить о том, что два потока могут попытаться получить доступ одновременно, поэтому нужно создавать критические секции вокруг одиночек.

Структурные паттерны

Идентификация данных — Организовывать атомарные величины в некие структурированные объекты.

Adapter

Базовые понятия

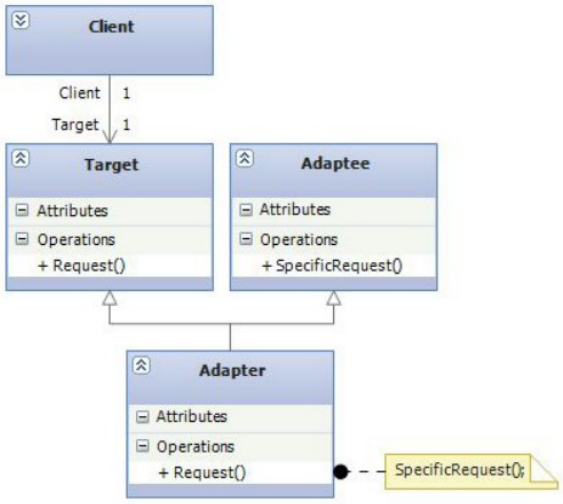
Клиент — это некоторый класс, которые использует другой класс, который называется адаптируемый/adaptee.

Адаптер — это класс, которые приводит интерфейс адаптируемого класса к интерфейсу, требуемому клиентом.

Причины

Бывает ситуация использования класса в неспицифичной среде. Например, для вывода графической информации о устройствах.

Если нет возможности или желания перегружать существующий класс, можно использовать некий класс-посредник для приведения к необходимому типу.

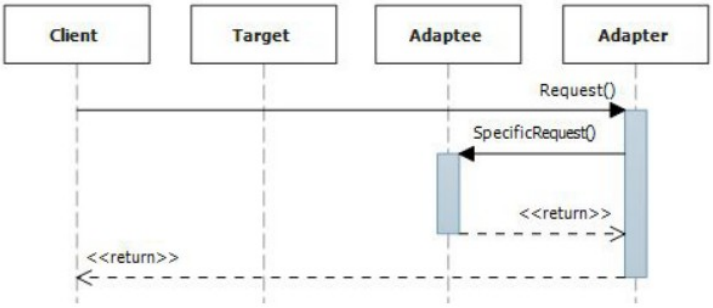


Client – это класс, который использует какие-то типы данных и ожидает стандартный интерфейс, описанный классом Target.

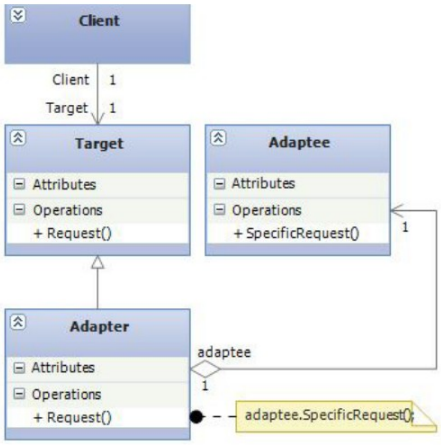
Target – это класс, у которого есть интерфейс, ожидаемый клиентом.

Adaptee – это класс, который нужен клиенту, но его интерфейс отличается от Target.

Adapter – класс, который приводит интерфейс класса Adaptee к интерфейсу класса Target.



Вместо наследования, можно поместить Adaptee внутрь Adapter, и вызывать метод из экземпляра, который хранится в атрибуте.



Результат

Основной положительный результат состоит в том, что мы можем гибко привести интерфейс к ожидаемому без изменения структуры самого класса.

Это повышает повторную используемость кода.

Bridge

Мост

Цель

Состоит в отделении абстракции от реализации.

Причины

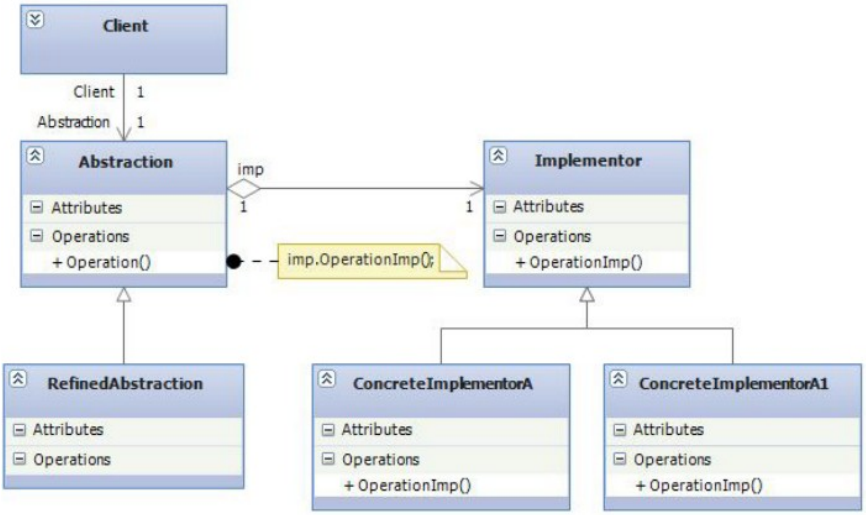
У реализации абстрактных классов есть слабые стороны:

- этот подход не всегда достаточно гибкий

- прямое наследование от абстракции связывает её с реализацией

Мост предполагает помещение интерфейса и его реализации в разные иерархии, что позволяет использовать их независимо.

Структура



Abstraction/Абстракция — определяет интерфейс абстракции и содержит объект исполнителя, которые определяет интерфейс реализации.

Implementator/Исполнитель — определяет интерфейс классов реализации, не обязан соответствовать интерфейсу абстракции.

RefinedAbstraction/Уточнённая абстракция — расширяет интерфейс, определённый абстракцией.

ConcreteImplementator/Конкретный исполнитель — реализует интерфейс исполнителя.

Результаты

Отделение абстракции от реализации делает код более гибким. Плюс, код становится более расширяемым, так как абстракции и реализации находятся в разных иерархиях.

Composite

Компоновщик

Цель

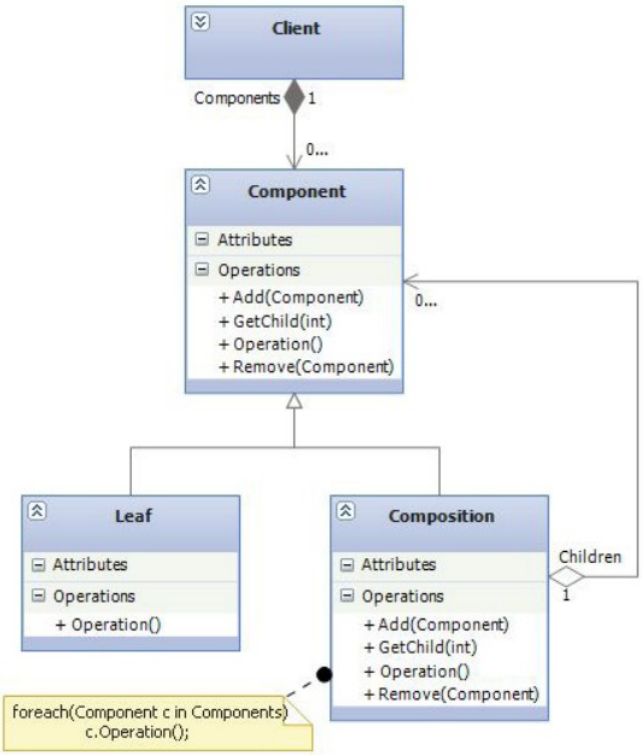
Позволяет представить объекты в виде дерева в связи часть-целое.

Причины

Структуры дерево являются очень распространёнными и часто используются для организации данных имеющих регулярную структуру.

Часть-целое — это иерархическая структура, которая предполагает, что элементов композиции данных может быть не только элементарный элемент, но и сама композиция.

Паттерн компоновщик предлагает рекурсивную структуру, которая не принимает решения о том, как обрабатывать отдельные элементы общей совокупности данных.



Структура

Component – описывает интерфейс объектов и их композиций. Реализует базовое поведение, общее и для объектов, и для их композиций. Определяет доступ к элементам композиции и интерфейс доступа к родительскому элементу рекурсивной структуры.

Leaf (лист) — это отдельный элемент композиции и описывает поведение атомарных элементов структуры.

Composition (композиция) определяет поведение для компонент, у которых есть дочерние структуры. Инкапсулирует дочерние структуры и реализует доступ к ним.

Client – это класс, который управляет элементами композиции.

Результаты

Упрощает организацию элементов в виде вложенной структуры данных и устраняет избыточность кода. Также упрощает клиентский объект, так как он обрабатывает композиции и элементарные объекты одинаково.

Также упрощает процесс добаления новых компонент.

Decorator

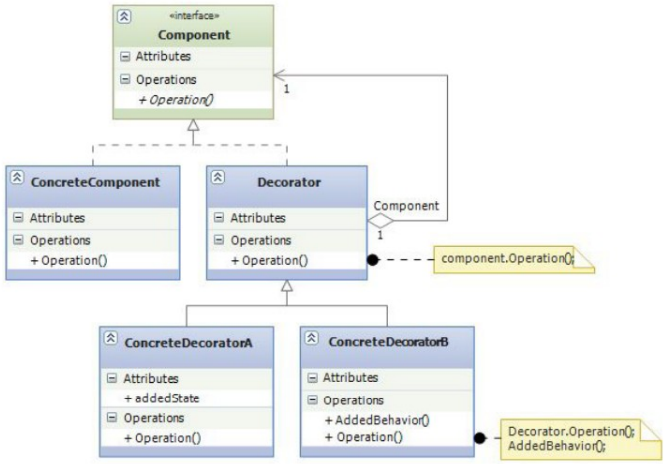
Декоратор

Цель

Реализовать возможность динамического добавления функционала объекту. А также распределить ответственность за выполнение отдельных функций между отдельными классами.

Альтернатива наследованию.

Причины



Принцип единичной ответственности требует, чтобы класс был проще к разработке и не имел доступа к тем операциям, за которые не отвечает.

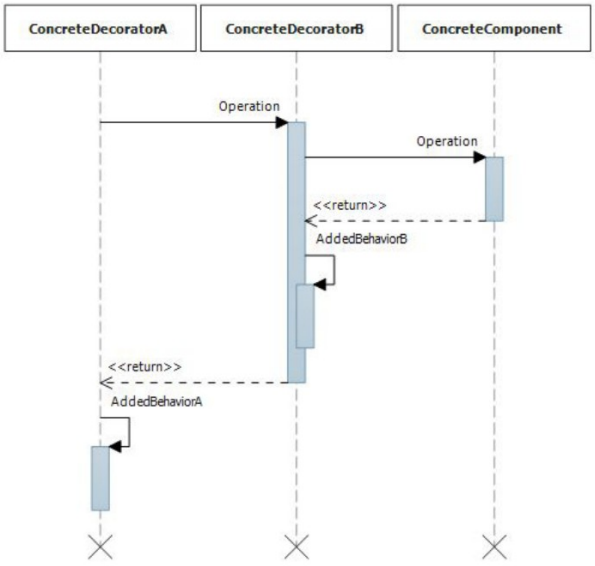
Один из методов это обеспечить — это наследование.

Структура

Component – это абстракция некоего элемента, для которого будут определены декорации.

ConcreteComponent – это реализация компонента.

Decorator – наследует и агрегирует компонент. Переопределяет реализацию операции так, чтобы выполнить функцию, инкапсулированную в компоненте и добавить новый функционал.



Результаты

Позволяет более гибко распределять обязанности по выполнению некоей сложной задачи между классами.

Предотвращает перенасыщение иерархии классов, поскольку устраняет необходимость создавать классы, которые имели бы функционал во всех необходимых комбинациях.

Facade

Фасад

Цель

Предназначен для скрытия содержимого и разделения логические частей на независимые под системы.

Система, которая состоит из минимально зависимых подсистем.