Александр Загоруйко © 2017

Design Patterns

C	Абстрактная фабрика
S	Адаптер
S	Мост
C	Строитель
B	Цепочка обязанностей
B	Команда
S	Компоновщик
S	Декоратор
S	Фасад
C	Фабричный метод
S	Приспособленец
B	Интерпретатор
B	Итератор
B	Посредник
B	Хранитель
C	Прототип
S	Прокси
B	Наблюдатель
C	Одиночка
B	Состояние
B	Стратегия
B	Шаблонный метод
B	Посетитель

Изобретаем автомобиль

Представьте, что вы хотите сделать новый автомобиль, но вы никогда этим не занимались. Сколько колес вы спроектируете для него? Сейчас вы скорее всего скажете что 4, однако почему не 3, 5, 10, 28? Потому что практикой использования уже было выяснено, что обычные автомобили лучше всего делать на 4-х колесах — это шаблон проектирования, сформированный временем. Именно такому же подходу служат паттерны в ООП, и вы не столкнётесь с ними в разработке до тех пор, пока вам не потребуется «сделать автомобиль». Однако иногда случается так, что вы создаёте «трицикл», и только потом, набив несколько шишек с его устойчивостью и неудачным вписыванием в колею на дороге, узнаёте, что существует паттерн «автомобиль», который значительно упростил бы вам жизнь, знай вы про него ранее.

Проверенные решения

Паттерны проектирования — это проверенные и готовые к использованию решения часто возникающих в повседневном программировании задач. Это не класс и не библиотека, которую можно подключить к проекту, это нечто большее. Кроме того, паттерны не зависят от языка программирования, и легко реализуются в большинстве языков. Следует, однако, помнить, что паттерн, будучи применённым неправильно или к неподходящей задаче, может принести немало проблем. Тем не менее, правильно избранный паттерн поможет решить задачу легко и просто.

Немного истории

Идея паттернов проектирования первоначально возникла в архитектуре. Архитектор Кристофер Александр в 1977-1979 гг. написал две революционные книги, содержащие описание шаблонов в строительной архитектуре и городском планировании. В книгах были представлены общие идеи, которые могли использоваться во многих областях, в том числе и в программировании.

Немного истории

В 1987 г. Кент Бэк на основе идеи Кристофера Александра создал несколько шаблонов разработки ПО для графических оболочек на языке Smalltalk.

В 1988 г. Эрих Гамма начал писать диссертацию на тему общей переносимости методики паттернов проектирования на разработку ПО.

Банда четырёх

В 1991 г. Гамма дописал диссертацию, и в сотрудничестве с Ричардом Хэлмом, Ральфом Джонсоном и Джоном Влиссидсом публикует книгу Design patterns – Elements of reusable Object-Oriented Software. В книге были описаны 23 паттерна, и вскоре команда авторов стала известна под названием «банда четырёх» (gang of four – GoF).

Немного истории

Следующим шагом стало описание Мартином Фаулером Enterprise Patterns, где были раскрыты типичные решения при разработке корпоративных приложений. Затем Джошуа Криевски показал, как можно постоянным рефакторингом, руководствуясь базовыми принципами ООП, обеспечить эволюцию кода, перемещая его от одного паттерна к другому, в зависимости от требований. Спустя какое-то время появилось понятие тестируемости программного кода, и все паттерны были переосмыслены с позиций тестируемости.

Причины возникновения

В конце 1980-х в сфере разработки ПО, в частности в ООП, накопилось множество сходных по своей сути решений. Эти решения требовали систематизации, обобщения и доступного описания. Такое упорядочивание знаний позволило бы повторно использовать готовые проверенные решения, а не заново изобретать велосипед.

Понятие паттерна

В общем смысле, паттерн проектирования представляет собой образец решения некоторой задачи так, что это решение можно использовать в различных ситуациях. Под ПП понимают описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте.

Процедурные алгоритмы

Алгоритмы процедурного программирования, вроде линейного поиска, суммирования элементов массива или быстрой сортировки, также являются паттернами, но не проектирования, а вычислений, поскольку они решают не архитектурные, а вычислительные задачи.

Антипаттерны

Паттерн – это общее описание хорошего способа решения задачи. Также существует понятие «антипаттерн» - это неудачное решение, которое не рекомендуется использовать. Цель паттерна – распознать возможность правильного решения проблемы. Цель антипаттерна – обнаружить плохую ситуацию и предложить подход к её устранению.

Знакомство с антипаттернами

- **Жёсткий код**, или «прибито гвоздями» - код настолько привязан к конкретной аппаратной конфигурации или системному окружению, что оторвать его для переноса в другое место можно только гвоздодёром ☺
- **Мягкий код** - код, конфигурируемый слишком гибко и запутано. Возникает вследствие выноса в файлы всех настроек программной логики.
- **Магические числа –** числовые литералы в программе без пояснения их смысла. Как правило, при изменении магического числа или его удалении код магически перестаёт работать. Типичные примеры — 17 и 54.
- **Магическая кнопка** - весь код написан в обработчике нажатия кнопки.

Знакомство с антипаттернами

- **Ненужная сложность** — делать сложно то, что можно сделать просто. Ведь как правило, программисту хочется побыстрее применить всё, что он изучил, а ещё заполучить по результатам проекта много новых строчек в резюме.
- **Спагетти** — это тот самый код, который вам однажды дадут сопровождать. После пяти минут причёсывания вставших дыбом волос от просмотра этого кода вы молча закрываете редактор и открываете браузер в поисках вакансии в другой фирме, где не практикуют такого жестокого обращения с программистами.
- **Божественный Объект** - небольшая часть кода, где сконцентрировано ВСЁ. Весь прочий код программы — исключительно декорация вокруг Божественного Объекта.

Знакомство с антипаттернами

- **Детонатор** — очень распространён, но редко обнаруживаем. Пример – «проблема 2000», когда для хранения даты не хватало одного байта.
- **Не виноватая я** - встречается в коде, написанном бывшими сотрудниками компании. В таком коде заключено столько старых проблем, что теперешние сотрудники могут защитить свои наработки от обвинений, утверждая, что именно чужой код — причина всех возникающих ошибок.
- **Паблик Морозов** — класс, который по запросу выдаёт доступ к приватным полям класса-родителя.
- **Спасти рядового Райана** - доступ к полю получить в принципе можно, но для этого надо снаряжать экспедицию, сопряжённую со множеством опасностей.

https://habrahabr.ru/post/59005

Индикаторы плохого дизайна

- Дубляж кода
- Большие методы
- Большие классы
- Зависть – класс чрезмерно использует методы другого класса
- Нарушение приватности – класс чрезмерно зависит от деталей реализации другого класса
- Нарушение завещания – класс переопределяет метод, и при этом нарушает его контракт (первоначальное предназначение)
- Ленивый класс – класс, который делает слишком мало
- Длинные идентификаторы

Классификация паттернов

В объектно-ориентированном анализе и проектировании (ООАП) на сегодняшний день разработано немало различных паттернов.

- **Архитектурные паттерны** – описывают фундаментальные способы структурирования программных систем и подсистем (не классов)
- **Паттерны проектирования** – 23 паттерна GoF
- **Паттерны анализа** – предоставляют общие схемы организации процесса объектно-ориентированного моделирования
- **Паттерны тестирования**

Паттерны проектирования

- **Основные** (fundamental) – наиболее важные паттерны, которые используются другими паттернами
- **Порождающие** (creational) – определяют способы создания объектов в системе
- **Структурные** (structural) – описывают способы построения сложных структур из классов и объектов
- **Поведенческие** (behavioral) – описывают способы взаимодействия между объектами
- **Паттерны параллельного программирования** (concurrency) – координируют работу потоков
- **MVC** (MVP, MVVM) – разделяют графический интерфейс, бизнес логику и данные
- **Паттерны корпоративных систем**

Порождающие паттерны

Абстрагируют процесс инстанциирования, позволяют сделать систему независимой от способа создания, композиции и представления объектов. Отвечают на вопросы «кто, когда и как создаёт объекты в системе».

- Abstract factory
- **Builder**
- Factory method
- Prototype
- **Singleton**

Структурные паттерны

Предлагают модели организации данных в структуры, которые наилучшим образом решают вопросы управления этими данными.

- **Adapter**
- Bridge
- Composite
- **Decorator**
- **Facade**
- Flyweight
- Proxy

Поведенческие паттерны

Служат для управления различными вариантами поведения объектов системы.

- Chain of responsibility
- Command
- Interpreter
- **Iterator**
- **Mediator**
- Memento
- **Observer**
- State
- **Strategy**
- Template method
- Visitor

Что почитать о паттернах

- Фримен, Паттерны проектирования (Head First O'Reilly)
- Будай, Дизайн патернів – просто як двері
- Шевчук – Design Patterns via C#
- Гамма – Приёмы объектно-ориентированного проектирования

https://yadi.sk/d/709uFvfbnfRTX

Сайты и видео-уроки

- https://sourcemaking.com/design_patterns
- https://geektimes.ru/post/84706/
- http://design-pattern.ru/
- https://en.wikipedia.org/wiki/Software_design_pattern
- https://www.lynda.com/Design-Patterns-training-tutorials/1304-0.html
- https://glamcoder.ru/video/design-patterns/
- http://itvdn.com/ru/patterns

Частота использования

- Iterator 
- Singleton 

Builder (строитель)

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

Плюсы: позволяет изменять внутреннее представление продукта; изолирует код, реализующий конструирование и представление; даёт более тонкий контроль над процессом конструирования.

Применение: 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.

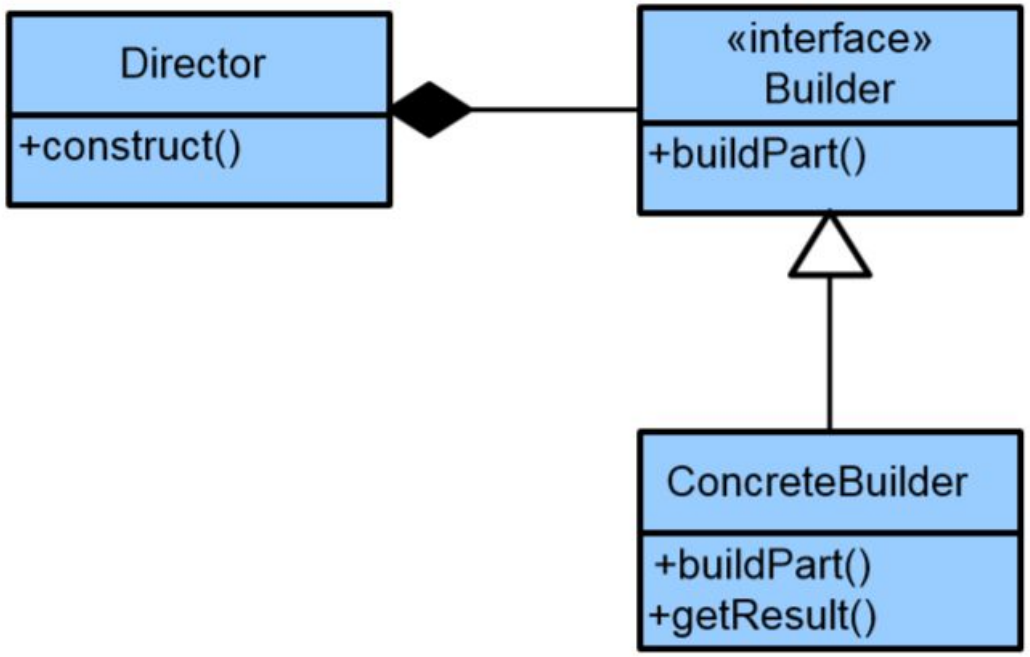
Builder (UML)

Строитель
Builder

Тип: Порождающий

Что это:

Разделяет создание сложного объекта и инициализацию его состояния так, что одинаковый процесс построения может создать объекты с разным состоянием.



Builder (участники)

- `Product` - **Продукт**:
  Представляет собой класс сложно-конструируемого объекта-продукта и содержит в себе набор методов для сборки конечного результата-продукта из частей. Класс продукта может быть связан связями отношений агрегации, с классами которые описывают составные части создаваемого продукта.

- `Builder` - **Абстрактный строитель**:
  Предоставляет набор абстрактных методов (интерфейс) для создания объекта-продукта из частей и получения готового результата.

- `ConcreteBuilder` - **Конкретный строитель**:
  Конструирует объект-продукт собирая его из частей, реализуя интерфейс, заданный абстрактным строителем (`Builder`). Предоставляет доступ к готовому продукту (возвращает продукт клиенту или в частном случае директору (`Director`)).

- `Director` – **Директор (Распорядитель)**:
  Пользуясь интерфейсом строителя (`Builder`), директор дает строителю указание построить продукт.

Builder (метафора)

«Строитель» внутри себя, как правило, содержит все сложные операции по созданию объекта (например, пакета сока). Вы говорите «хочу сока», а строитель запускает уже целую цепочку различных операций (создание пакета, печать на нём изображений, заправка в него сока, учёт того сколько пакетов было создано и т.п.). Если вам потребуется другой сок, например ананасовый, вы точно также говорите только то, что вам нужно, а «строитель» уже позаботится обо всем остальном (какие-то процессы повторит, какие-то сделает заново и т.п.). В свою очередь процессы в «строителе» можно легко менять (например изменить рисунок на упаковке), однако потребителю сока это знать не требуется, он также будет легко получать требуемый ему пакет сока по тому же запросу.

Builder (пример кода)

https://git.io/vrxd9

Facade (фасад)

Это структурный шаблон, позволяющий скрыть сложность системы путём сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы.

Проблема: как обеспечить унифицированный интерфейс с набором разрозненных реализаций или интерфейсов, например, с подсистемой, если нежелательно высокое связывание с этой подсистемой или реализация подсистемы может измениться?

Facade (фасад)

Решение: определить одну точку взаимодействия с подсистемой — фасадный объект, обеспечивающий общий интерфейс с подсистемой, и возложить на него обязанность по взаимодействию с её компонентами. Фасад — это внешний объект, обеспечивающий единственную точку входа для служб подсистемы. Реализация других компонентов подсистемы закрыта и не видна внешним компонентам.

Facade (фасад)

При проектировании сложных систем, зачастую применяется т.н. принцип декомпозиции, при котором сложная система разбивается на более мелкие и простые подсистемы. Причём, уровень декомпозиции (её глубину) определяет исключительно проектировщик. Благодаря такому подходу, отдельные компоненты системы могу быть разработаны изолированно, затем интегрированы вместе. Однако возникает проблема — высокая связность модулей системы. Это проявляется в большом объеме информации, которой модули обмениваются друг с другом. К тому же, для подобной коммуникации одни модули должны обладать достаточной информацией о природе других модулей.

Facade (фасад)

Фасад — есть ни что иное, как некоторый объект, аккумулирующий в себе высокоуровневый набор операций для работы с некоторой сложной подсистемой. Фасад агрегирует классы, реализующие функциональность этой подсистемы, но не скрывает их. Важно понимать, что клиент, при этом, не лишается более низкоуровневого доступа к классам подсистемы, если ему это, конечно, необходимо. Фасад упрощает выполнение некоторых операций с подсистемой, но не навязывает клиенту свое использование.

Facade (фасад)

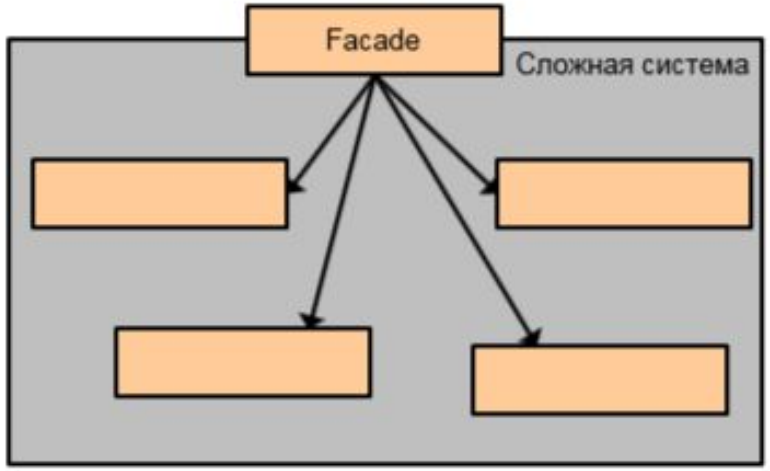
Используйте этот паттерн, если вы хотите:

- Предоставить простой интерфейс к сложной подсистеме
- Отделить систему от клиентов и от других систем
- Разделить подсистему на независимые слои (точка входа каждого слоя - фасад)
- Повысить переносимость

Facade (метафора)

Паттерн «фасад» используется для того, чтобы делать сложные вещи простыми. Возьмем для примера автомобиль. Представьте, если бы управление автомобилем происходило немного по-другому: нажать одну кнопку, чтобы подать питание с аккумулятора, другую, чтобы подать питание на инжектор, третью, чтобы включить генератор, четвертую, чтобы зажечь лампочку на панели и так далее. Всё это было бы очень сложно. Для этого такие сложные наборы действий заменяются более простыми и комплексными как «повернуть ключ зажигания». В данном случае, поворот ключа зажигания и будет тем самым «фасадом» для всего обилия внутренних действий автомобиля.

Facade (простая схема)



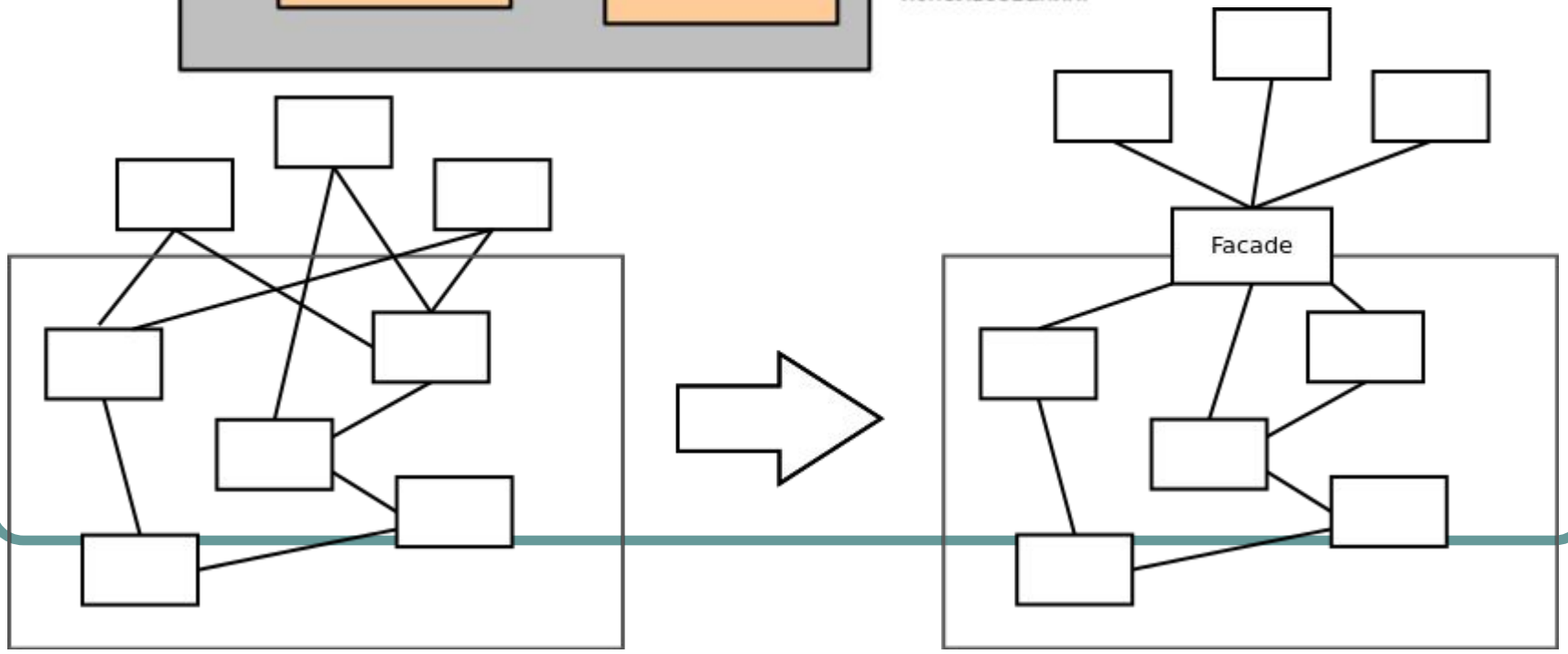
Фасад

Facade

Тип: Структурный

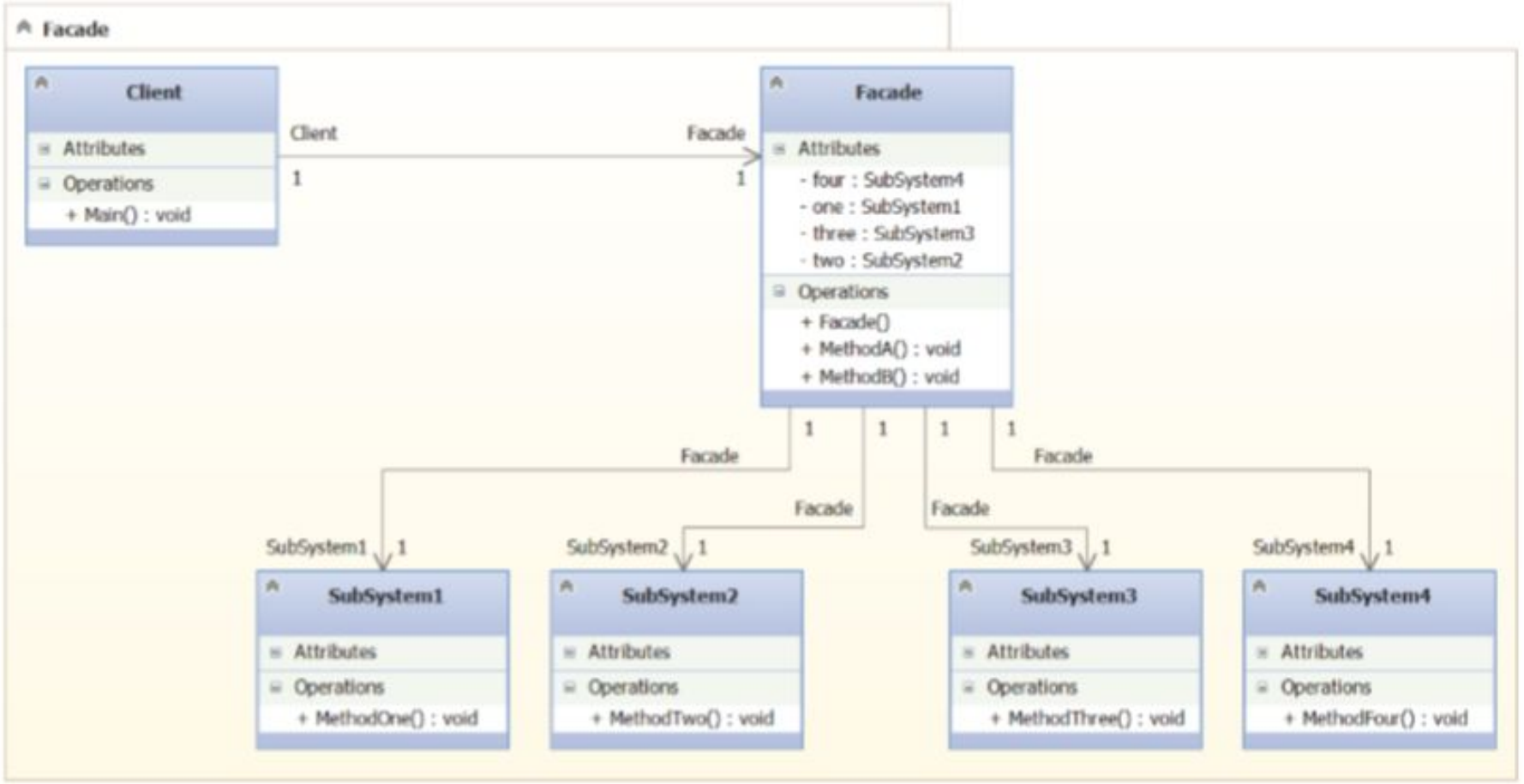
Что это:

Предоставляет единый интерфейс к группе интерфейсов подсистемы. Определяет высокоуровневый интерфейс, делая подсистему проще для использования.



Facade (UML)

Структура паттерна на языке UML



Facade (пример кода)

https://git.io/vrxbX

Decorator (декоратор) 

Структурный шаблон, предназначенный для динамического подключения дополнительного поведения к объекту. Предоставляет гибкую альтернативу практике создания подклассов с целью расширения функциональности.

Задача: объект, который предполагается использовать, выполняет основные функции. Однако может потребоваться добавить к нему некоторую дополнительную функциональность, которая будет выполняться до, после или даже вместо основной функциональности объекта.

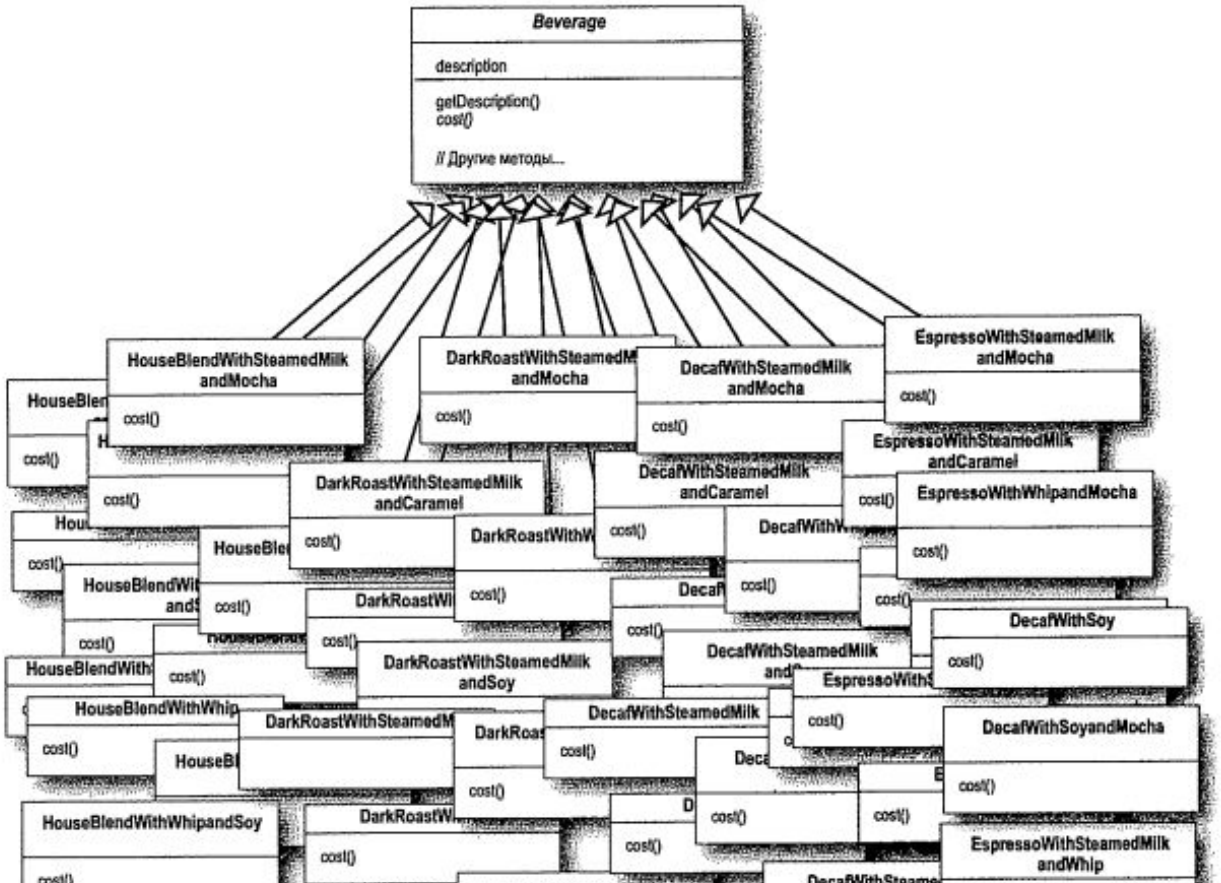
Decorator (метафора)

Как понятно из названия, данный паттерн чаще всего используется для расширения исходного объекта до требуемого вида. Например, мы условно можем считать «декоратором» человека с кистью и красной краской. Таким образом, какой бы объект (или определённый тип объектов) мы не передали в руки «декоратору», на выходе мы будем получать красные объекты.

Делаем кофе!

К кофе можно заказать различные дополнения (пенка, шоколад и т. д.), да еще украсить все сверху взбитыми сливками. Дополнения не бесплатны, поэтому они должны быть встроены в систему оформления заказов.

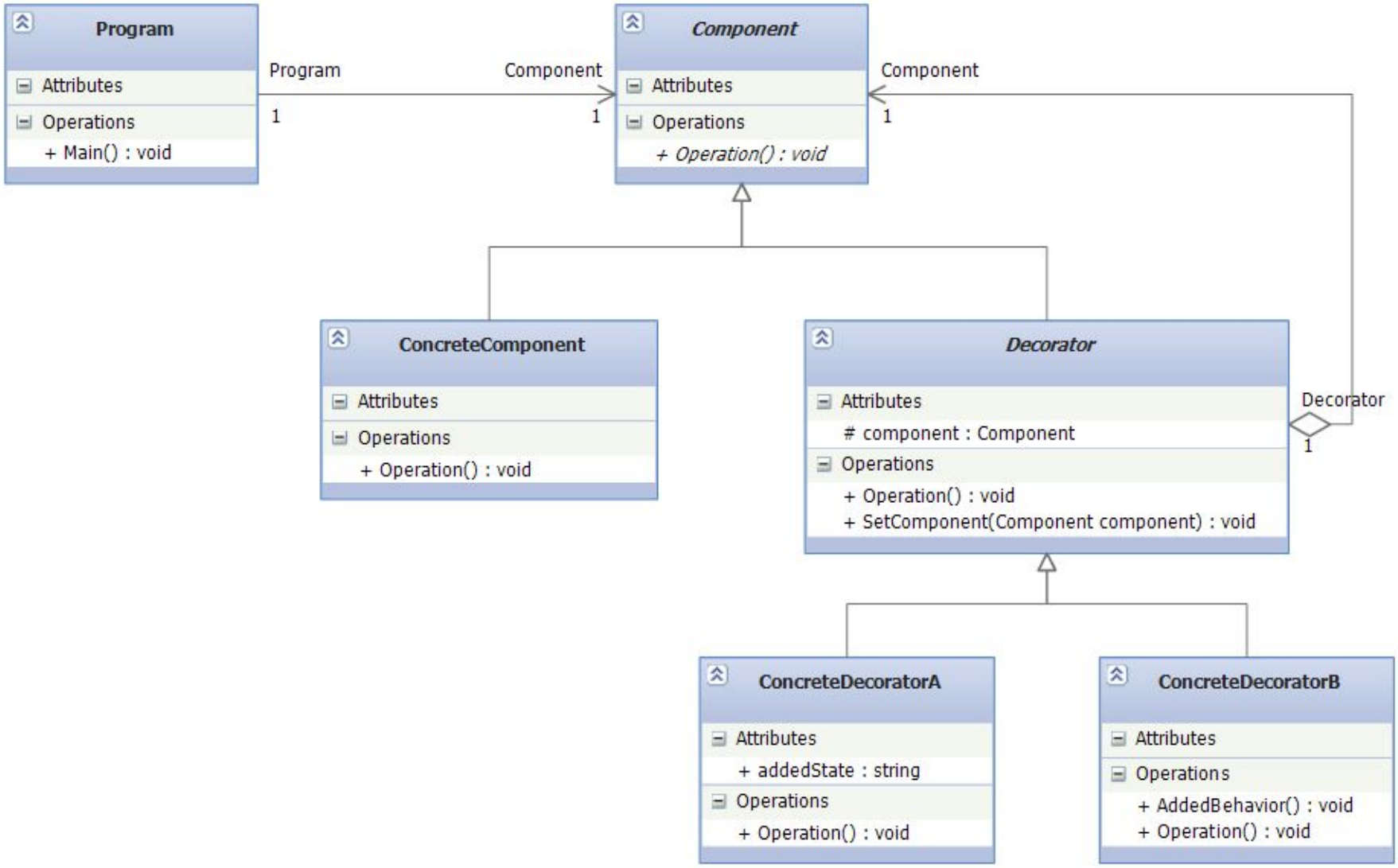
Первая попытка...



Decorator (пример кода)

https://git.io/vrxx9

Decorator (UML)



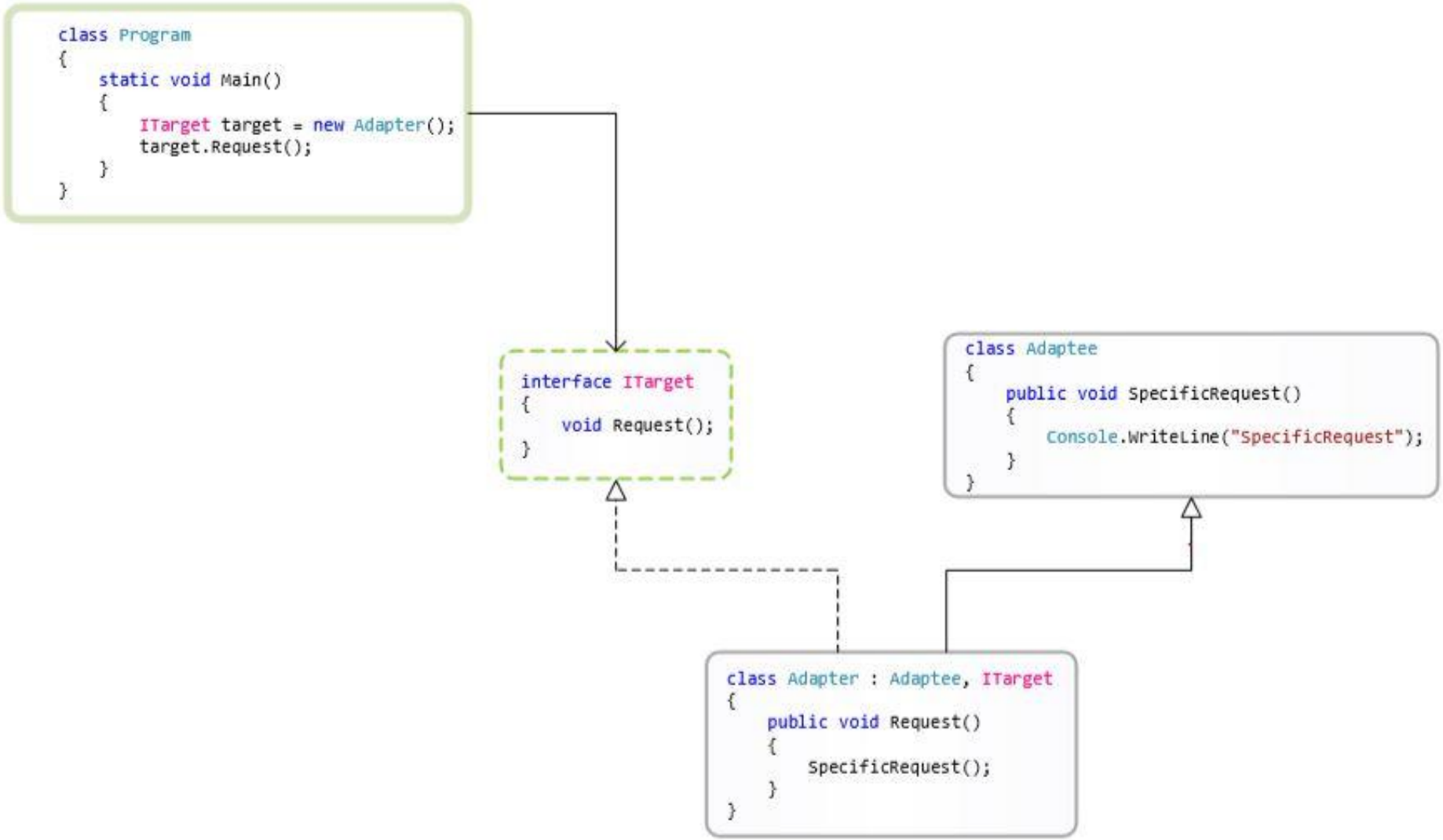
Adapter (адаптер)

Паттерн Adapter - преобразует интерфейс (набор имён методов) одного класса в интерфейс (набор имён методов) другого класса, который ожидают клиенты. Адаптер обеспечивает совместную работу классов с несовместимыми интерфейсами, такая работа без Адаптера была бы невозможна.

Adapter (метафора)

Самый широко распространённый вид адаптеров в жизни – адаптеры электрической сети. Исторически сложилось так, что в разных странах мира имеются свои технологические стандарты на производство и использование электробытовых устройств. И часто бывает так, что стандарты устройств одной страны не совместимы со стандартами устройств других стран.

Adapter (диаграмма классов)



Adapter (участники)

`Target` - **Цель**:
Формирует требуемый клиенту интерфейс (набор имен методов).

`Client` - **Клиент**:
Пользуется объектами с интерфейсом `Target`.

`Adaptee` - **Адаптируемый**:
Содержит интерфейс (набор методов) требующий адаптации.

`Adapter` - **Адаптер**
Адаптирует интерфейс `Adaptee` к интерфейсу `Target`.

Adapter (пример кода)

https://git.io/vok9B

Mediator (посредник)

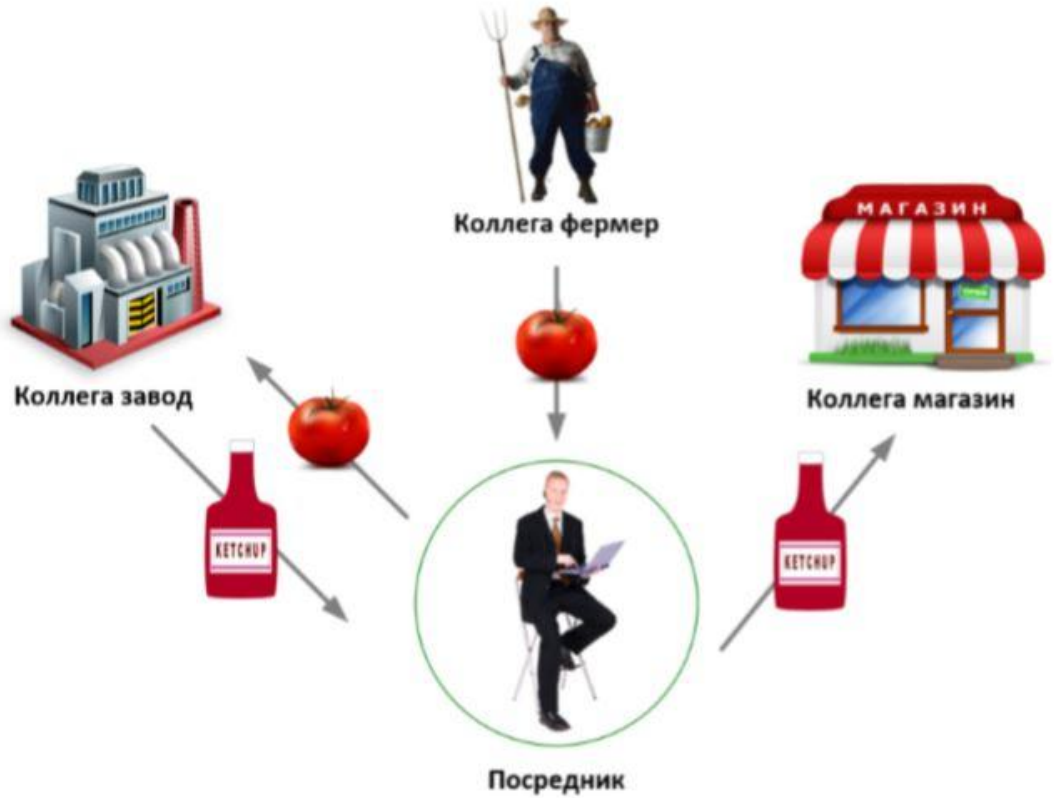
Поведенческий шаблон, обеспечивающий взаимодействие множества объектов, формируя при этом слабую связанность и избавляя объекты от необходимости явно ссылаться друг на друга.

Проблема: обеспечить взаимодействие множества объектов, сформировав при этом слабую связанность и избавив объекты от необходимости явно ссылаться друг на друга.

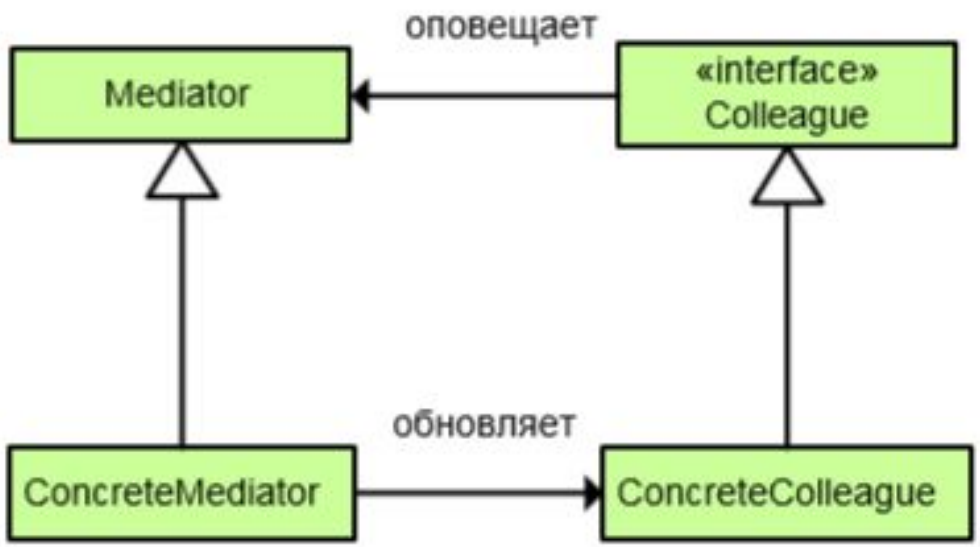
Решение: создать объект, инкапсулирующий способ взаимодействия множества объектов.

Mediator (метафора)

Предлагается рассмотреть пример работы человека-посредника в объективной реальности с точки зрения его полезности, а не с точки зрения извлечения им прибыли. На рисунке ниже показана роль посредника в процессе производства кетчупа. Посредник покупает свежие помидоры у фермера, отправляет эти помидоры на консервный завод для переработки в кетчуп, а полученный кетчуп оптом продает в магазин розничной торговли.



Mediator (UML)

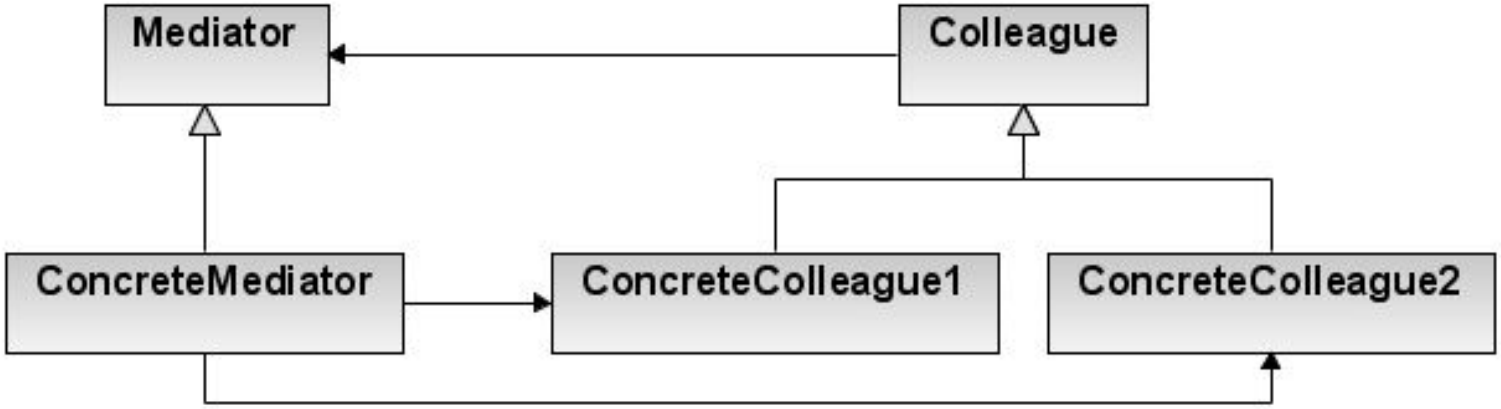


Посредник
Mediator

Тип: Поведенческий

Что это:

Определяет объект, инкапсулирующий способ взаимодействия объектов. Обеспечивает слабую связь, избавляя объекты от необходимости прямо ссылаться друг на друга и даёт возможность независимо изменять их взаимодействие.



Mediator (пример кода)

https://git.io/vrpfe

Strategy (стратегия)

Стратегия — поведенческий шаблон, предназначенный для определения семейства алгоритмов, инкапсуляции каждого из них и обеспечения их взаимозаменяемости. Это позволяет выбирать алгоритм путём определения соответствующего класса. Позволяет менять выбранный алгоритм независимо от объектов-клиентов, которые его используют.

Strategy (задача)

По типу клиента (или по типу обрабатываемых данных) выбрать подходящий алгоритм, который следует применить. Если используется правило, которое не подвержено изменениям, нет необходимости обращаться к шаблону «стратегия».

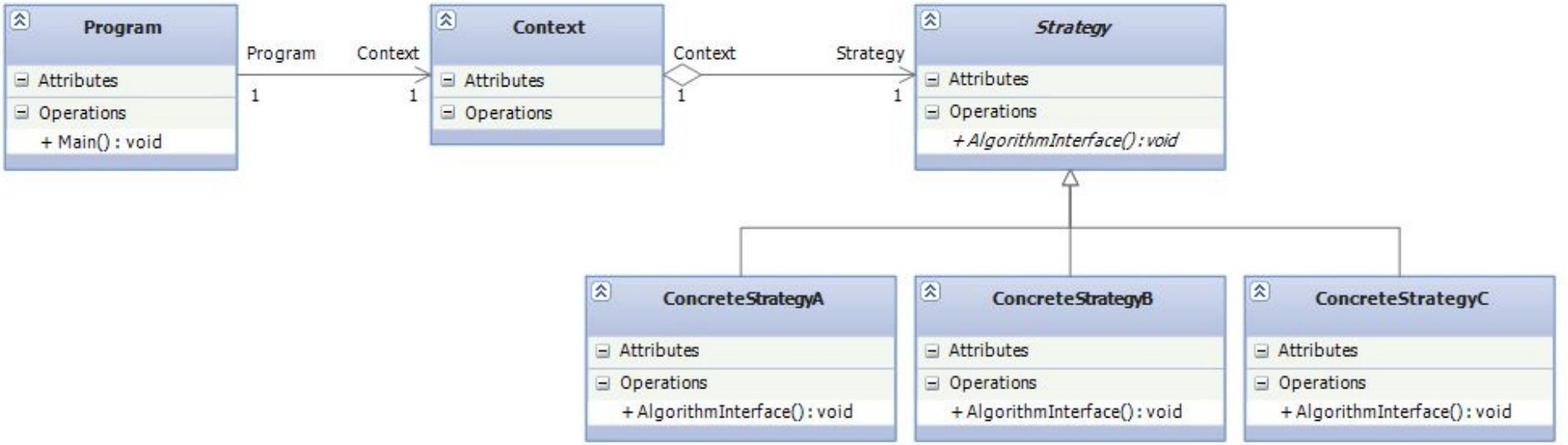
Strategy (мотивы)

- Программа должна обеспечивать различные варианты алгоритма или поведения
- Нужно изменять поведение каждого экземпляра класса
- Необходимо изменять поведение объектов на стадии выполнения
- Введение интерфейса позволяет классам-клиентам ничего не знать о классах, реализующих этот интерфейс и инкапсулирующих в себе конкретные алгоритмы.

Strategy (метафора)

Если на улице дождь – то мы берём с собой куртку и зонтик, а если жарит солнце – то майку и солнцезащитные очки. Что одевать – это наша стратегия, которая может изменяться в зависимости от обстоятельств. Другой пример – способ получения водительских прав. Можно прийти и сказать «Хочу права, денег мало», и в ответ мы получим права через длительное время и с большой тратой ресурсов. Если же сказать «Хочу права, денег много», то права появятся очень быстро ☺ Как себя вести – мы решаем сами.

Strategy (UML)



Strategy (пример кода)

- Проблемы наследования:

  https://git.io/voUtO
- Проблемы интерфейсов:

  https://git.io/voUqb
- Паттерн Стратегия:

  https://git.io/voUYJ

Observer (наблюдатель)

Наблюдатель — поведенческий шаблон. Также известен как «издатель-подписчик» (Publisher-Subscriber), Listener (слушатель). Создает механизм у класса, который позволяет получать экземпляру объекта этого класса оповещения от других объектов об изменении их состояния, тем самым наблюдая за ними.

Observer (метафора)

Очень распространенный паттерн в реальной жизни. Например если вы подписались на какую-либо email (или смс) рассылку, то ваш email (или номер сотового телефона) начинает реализовывать паттерн «наблюдатель». Как только вы подписываетесь на событие (например новая статья или сообщение), всем кто подписан на это событие (наблюдателям) будет выслано уведомление, а они уже в свою очередь могут выбрать, как на это сообщение реагировать.

Observer (участники)

При реализации шаблона «наблюдатель» обычно используются следующие классы:

Observable — интерфейс, определяющий методы для добавления, удаления и оповещения наблюдателей;

Observer — интерфейс, с помощью которого наблюдатель получает оповещение;

ConcreteObservable — конкретный класс, который реализует интерфейс Observable;

ConcreteObserver — конкретный класс, который реализует интерфейс Observer.

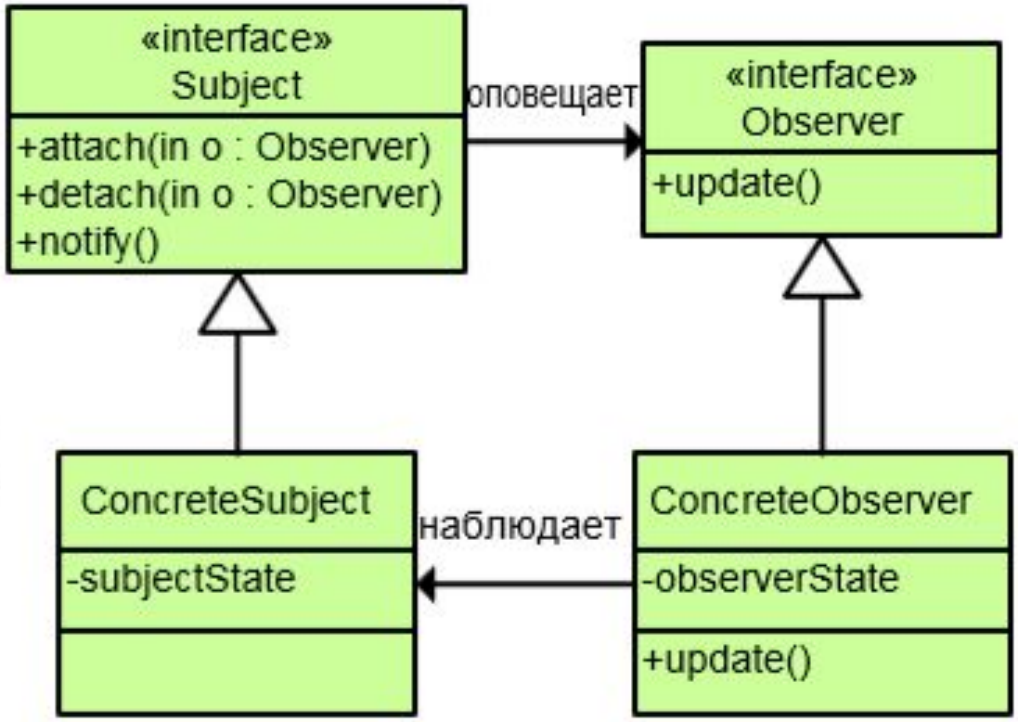
Observer (UML)

Наблюдатель
Observer

Тип: Поведенческий

Что это:

Определяет зависимость "один ко многим" между объектами так, что когда один объект меняет своё состояние, все зависимые объекты оповещаются и обновляются автоматически.



Observer (пример кода)

https://git.io/voUGI

Паттерн MVC

Шаблон **Model-View-Controller** — это методология разделения структуры приложения на специализированные компоненты. Это не готовая библиотека классов, не фреймворк, а просто, грубо говоря, сборник советов о том, как лучше организовать классы и взаимосвязи между ними. Схема MVC предполагает разделение всей системы на 3 взаимосвязанных компонента (подсистемы): модель, представление и контроллер. У каждого компонента своя цель, а главная особенность в том, что любой из них можно с лёгкостью заменить на другой или модифицировать, практически не затронув другие подсистемы. Преимущества такого подхода: модульность, расширяемость, простота поддержки и тестирования.

https://habrahabr.ru/post/215605/

Паттерн MVC

- Представление (вид) отвечает за отображение информации, поступающей из системы или в систему.
- Модель является сутью системы и отвечает за непосредственные алгоритмы, расчёты и тому подобное внутреннее устройство системы.
- Контроллер является связующим звеном между представлением и моделью системы, посредством которого и существует возможность произвести разделение между ними. Контроллер получает данные от пользователя и передает их в модель. Кроме того, он получает сообщения от модели, а также может изменять текущий режим представления.

https://ru.wikipedia.org/wiki/Model-View-Controller

Паттерн MVC (пример кода)

https://git.io/voU0d

Практика

Добавить в последний пример возможность узнать прогноз погоды для других 4-5 городов кроме Одессы.