2. Проектирование ПО

2.4. Об архитектуре и паттернах проектирования

Признаки удачно спроектированной архитектуры

Эффективность системы

Надёжность
Безопасность
Производительность
Масштабируемость
Отзывчивость
…

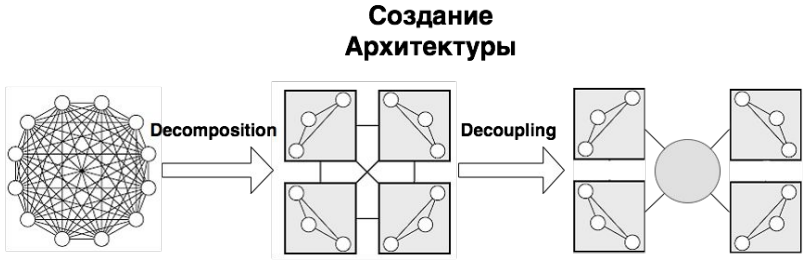
Гибкость системы

Легкость изменения текущей ф-ти
Исправление ошибок
Настройка системы
- Под пользователя
- Под различные сценарии

…

Поддерживаемость системы

Масштабируемость процесса разработки
Тестируемость
Обновляемость
Переиспользуемость
Обратная совместимость
…



Признаки неудачно спроектированной архитектуры

Жесткость

Тяжело модифицировать
Модификация одного модуля влечёт за собой (избыточные) модификации в других

…

Хрупкость

Сложность изменения текущей ф-ти
Изменение в одном модуле нарушают другие модули

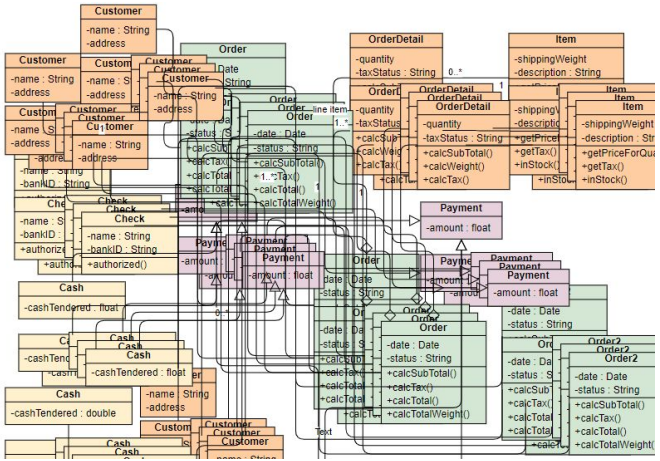
…

Неподвижность

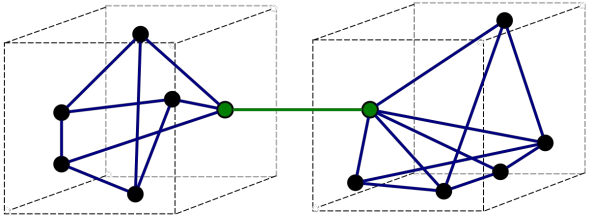
Тяжело “извлечь” модуль наружу

- Как правило, говорит о большом сопряжении и низкой связности некоторых модулей

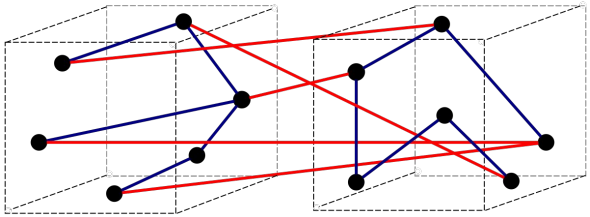
…



Модульность



a) Good (loose coupling, high cohesion)



b) Bad (high coupling, low cohesion)

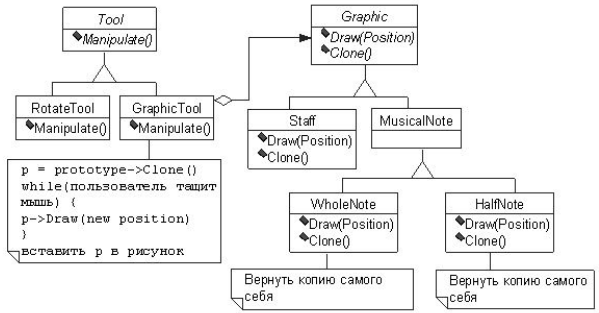
- Метрики взаимозависимости модулей
  - **Coupling (сопряжение)** -- мера того, насколько взаимозависимы разные модули в программе (по вызовам, как правило)
  - **Cohesion (связность)** -- степень, в которой задачи, выполняемые одним модулем, связаны друг с другом (в плане смысла)

Цель -- небольшое сопряжение и сильная связность (low coupling, high cohesion)

Паттерны

- Паттерн (от англ. Pattern) — образец, шаблон.
- В проектировании программ (и не только) -- это разумный, устоявшийся способ решения какой-либо задачи, который точно приведёт к намеченному результату*

- Паттерны ООП -- совсем тактические приёмы
- Архитектурные паттерны -- приёмы проектирования одной или нескольких подсистем
- Архитектурные стили -- что-то совсем глобальное …



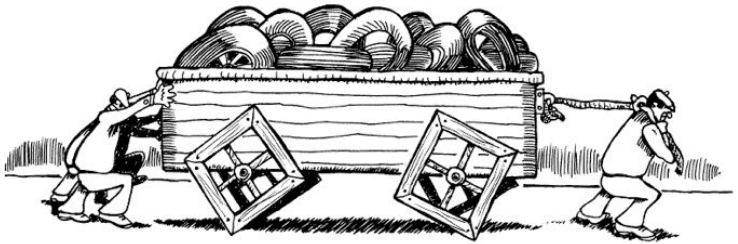
- “Тропа умнее человека” (народное творчество)

Анти-паттерны

- В проектировании программ (и не только) -- это пример того, как не нужно решать те или иные задачи

- В ООП
- В кодировании
- Архитектурные
- Методологические

...



Паттерны проектирования (в ООП)

- В проектировании программ (и не только) -- это разумный, устоявшийся способ решения какой-либо задачи, который точно приведёт к намеченному результату*
- Это некоторая формализация (именованная), которая описывает одну тактическую проблему, способ решения и результат

Книга GOF Patterns:
Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. Паттерны объектно-ориентированного проектирования / Пер. с англ.: А. Слинкин. — СПб.: Питер, 2021. — 448 с. — ISBN 978-5-4461-1595-2.

Имя	**Задача**
Решение	**Результат**

- “Тропа умнее человека” (народное творчество)

Паттерны проектирования

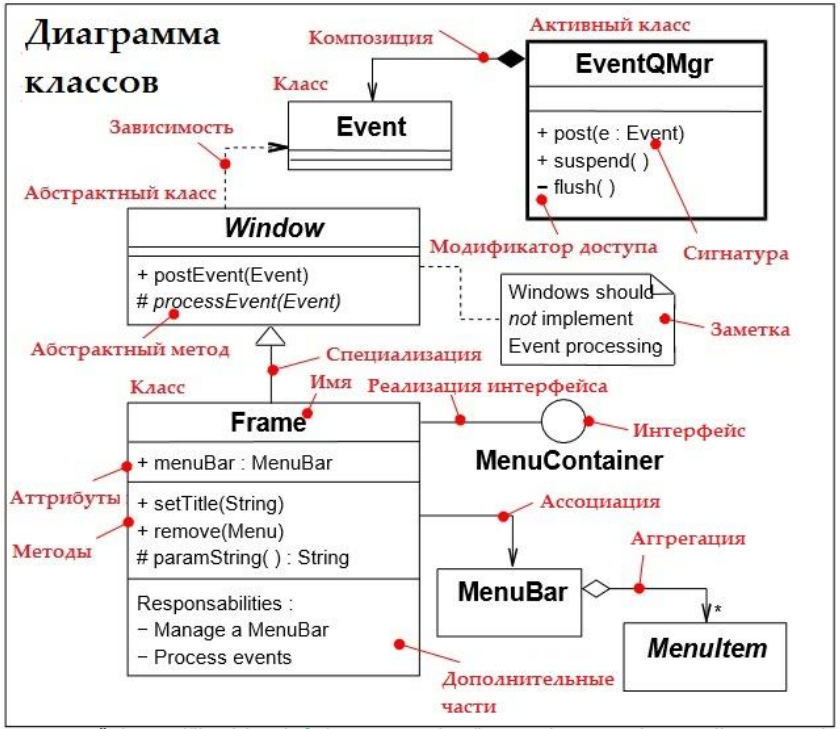
Порождающие - конструируют объекты (служат для контроля и оптимизации их создания)

Структурные - организуют структуру объектов и их структурное взаимодействие

Поведенческие - определяют или отслеживают поведение объектов

Ремарка о нотации

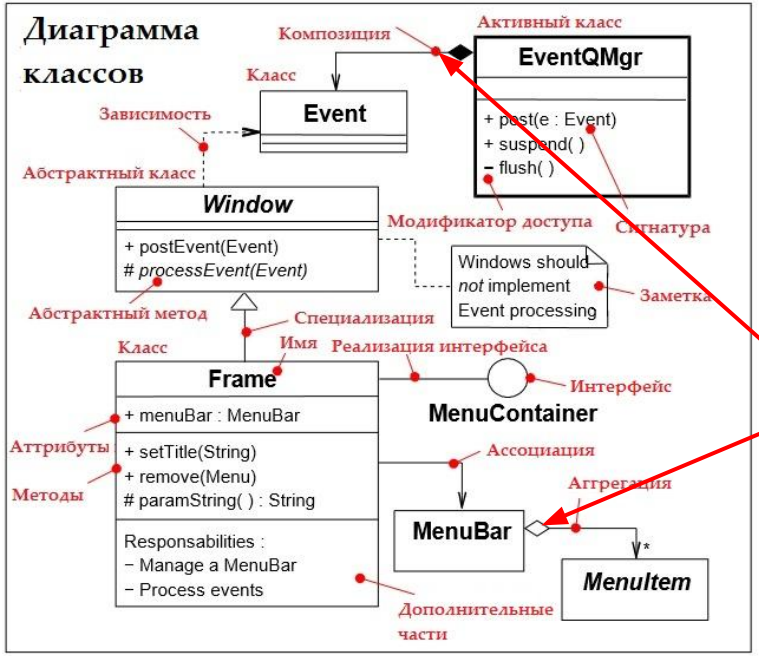
Нам понадобятся диаграммы классов -- будем пользоваться диаграммами классов UML.



“Шпаргалка по UML. Электронный источник”: https://lhs-blog.info/programming/kompaktnaya-shpargalka-po-uml-diagrammam-klassov/ (Дата обр. 13.03.2023)

Ремарка о нотации

Нам понадобятся диаграммы классов -- будем пользоваться диаграммами классов UML.



Существует два подвида **отношения включения**: если один объект создает другой объект и время жизни "части" зависит от времени жизни целого, то это называется "**композиция**", если же один объект получает ссылку (указатель) на другой объект в процессе конструирования, то это уже **агрегация.**

“Шпаргалка по UML. Электронный источник”: https://lhs-blog.info/programming/kompaktnaya-shpargalka-po-uml-diagrammam-klassov/ (Дата обр. 13.03.2013)

Порождающие паттерны

1/ Factory method

2/ Abstract factory

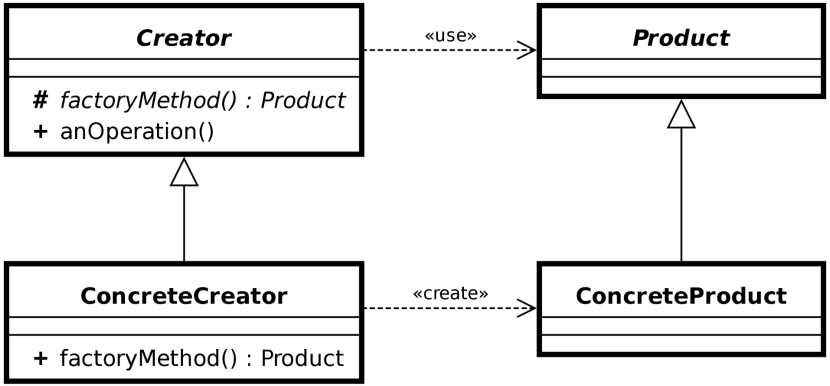
3/ Builder

4/ Singleton

5/ Prototype

Фабричный метод (Factory Method)

Фабричный метод
- Задача: единообразное создание объектов из общей иерархии
- Классы разделены (одним!) признаком



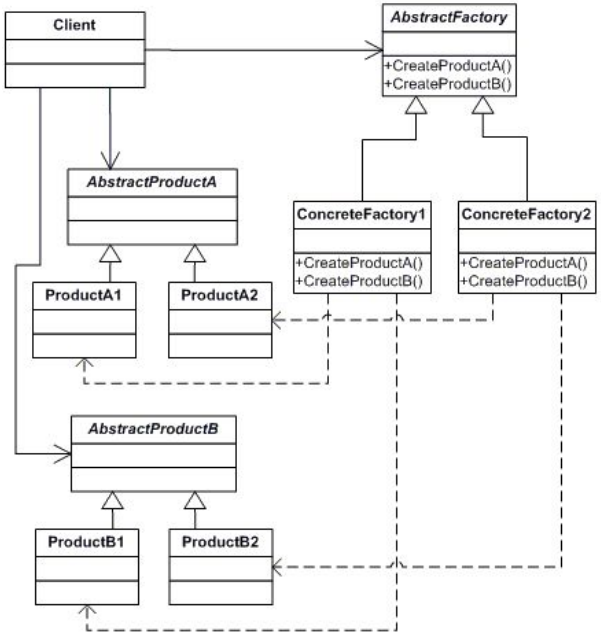
\+ В принципе, решает задачу
\- Выбор ограничен одним признаком в рамках одной иерархии

Варианты:
- Параметризованный:
  - Через enum
  - Через шаблон

Абстрактная фабрика (Abstract Factory)

Абстрактная Фабрика

- Задача: единообразное создание объектов из иерархий
- Классы разделены (не обязательно одним!) признаками. Один из признаков должен главным (быть более “жестким”).



- Осуществляем выбор по “главному” признаку
- Создаются конкретные фабрики с фиксированным “главным” признаком
- Созданные конкретные фабрики управляют созданием объектов с различными вторичными признаками

\+ Изолирует конкретные классы

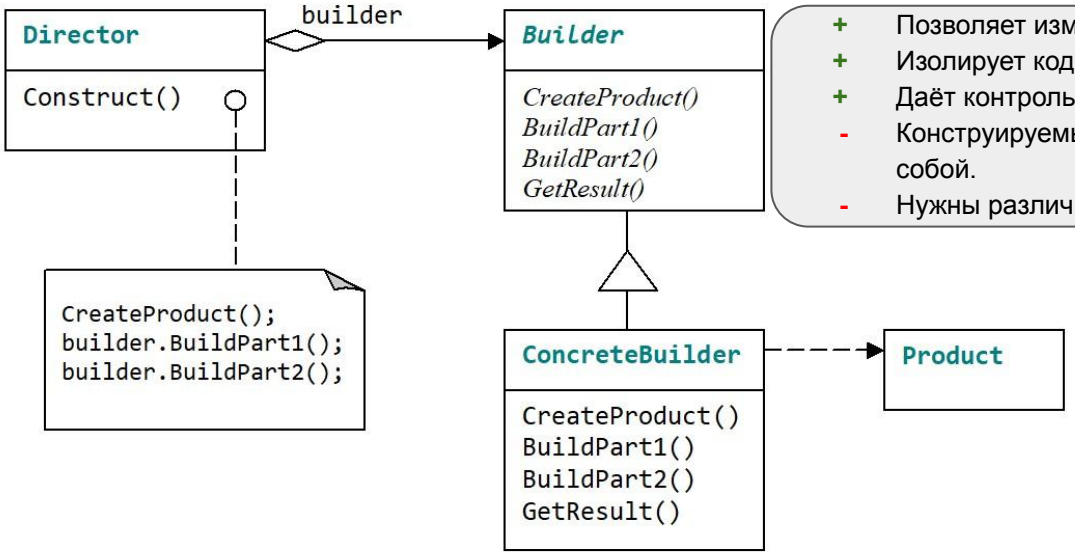
\+ Упрощает замену семейств продуктов (объектов)

\+ Гарантирует сочетаемость продуктов

\- Сложно добавить поддержку нового признака

Строитель (Builder)

Строитель
- Задача: создание сложного объекта по частям
- Последовательно вызываются методы, строящие объект, состояние создания доступно по GetResult()

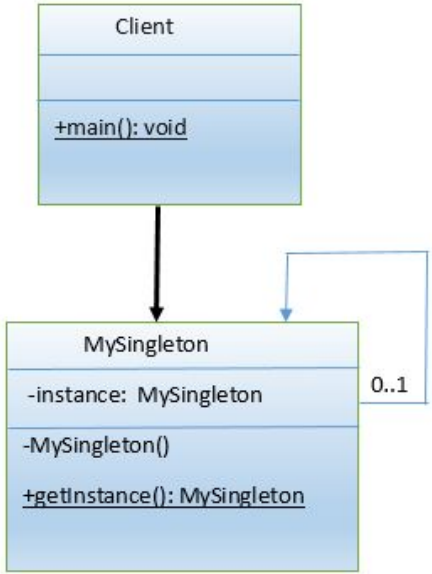


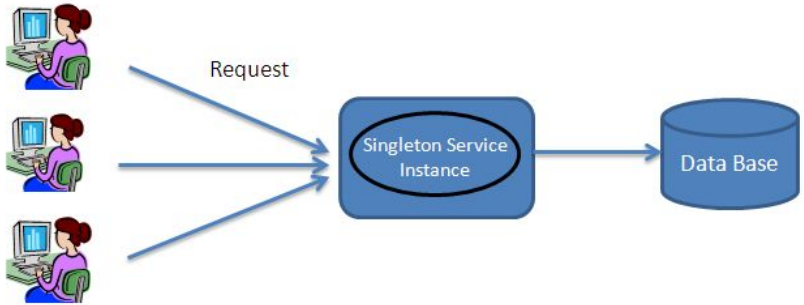
\+ Позволяет изменять внутреннее представление продукта.
\+ Изолирует код, реализующий конструирование и представление.
\+ Даёт контроль над процессом конструирования
\- Конструируемые части должны быть условно независимы между собой.
\- Нужны различные представления конструируемого объекта.

- Объект, руководящий билдером для создания объекта, часто называют Директором
- Порядок вызова BuildPartX и GetResult() зависит от ситуации

Одиночка (Singleton)

Одиночка
- Задача: обеспечить создание и существование объекта в единственном экземпляре.
-





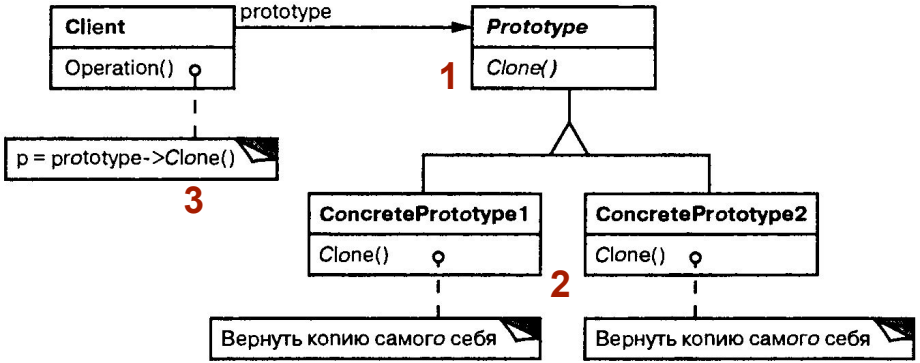
\+ Класс сам контролирует процесс создания единственного экземпляра.
\+ Паттерн легко адаптировать для создания нужного числа экземпляров.
\+ Возможность создания объектов классов, производных от Singleton.
\- В случае использования нескольких взаимозависимых одиночек их реализация может резко усложниться.

- Одиночка X нужен, чтобы на запрос некоторого объекта “создай мне X” ему либо создавали X, либо отвечали: “вот же он!”.
- Часто используется с паттерном “Реестр (Registry)”.

Прототип (Prototype)

Прототип@

- Задача: получить копию объекта, не разбираясь в его внутреннем устройстве, и иметь доступ к ней по указателю, вне зависимости от типа



1. Интерфейс прототипов описывает операции клонирования. В большинстве случаев – это единственный метод **clone**.
2. Конкретный прототип реализует операцию клонирования самого себя. Помимо банального копирования значений всех полей, здесь могут быть спрятаны различные сложности, о которых не нужно знать клиенту. Например, клонирование связанных объектов, распутывание рекурсивных зависимостей и прочее.
3. Клиент создаёт копию объекта, обращаясь к нему через общий интерфейс прототипов.

Паттерны поведения

1/ Итератор (Iterator)

2/ Посетитель (Visitor)

3/ Стратегия (Strategy)

4/ Медиатор

5/ Снимок

6/ Команда

7/ Цепочка ответственностей

8/ Шаблонный метод

9/ Состояние

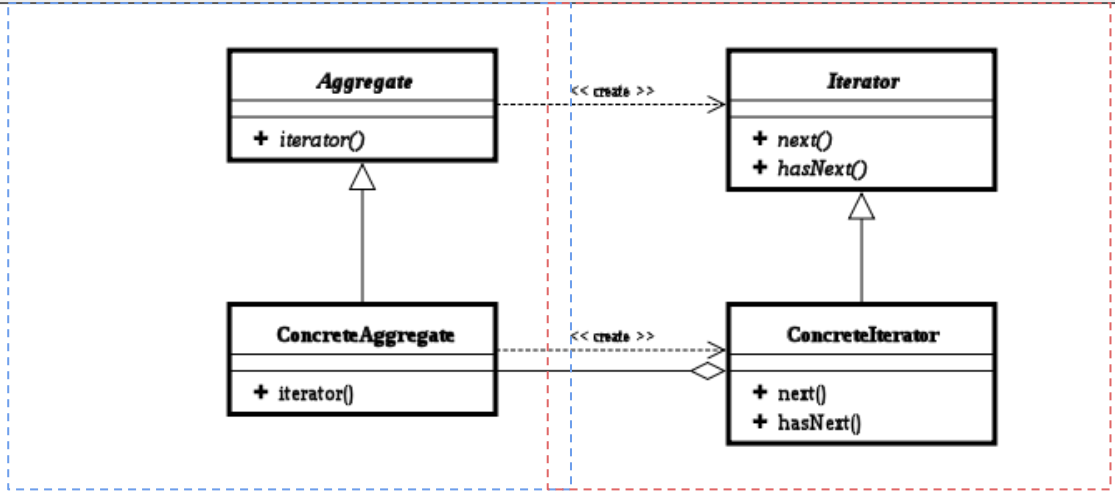
10/ Наблюдатель

11/ …

Итератор (Iterator)

Итератор @

- Задача: последовательно обходить элементы составных объектов, не раскрывая их внутреннего устройства, и (возможно) предоставлять доступ к содержимому.
- Решение:

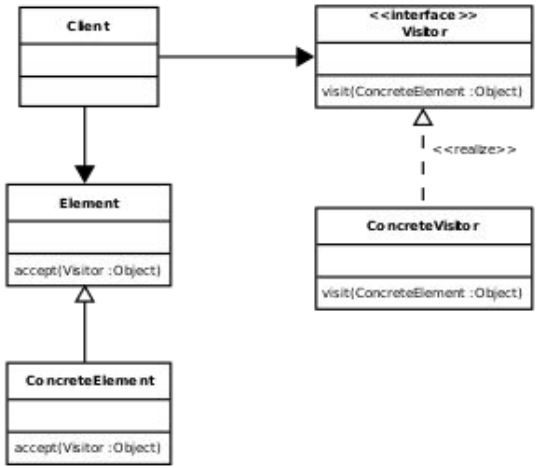


- Требования:
  - Не нарушать инварианты

- Составной объект, включающий итератор, часто называют “агрегатом”.

Посетитель (visitor)

Посетитель @
- Задача: не изменяя основного класса, добавить в него новые операции.
- Решение:



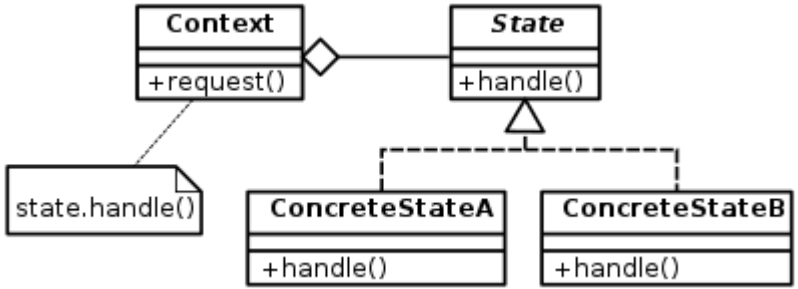
- Требования:
  - Не изменять основной класс (изменения ограничиваются добавлением accept())
- Обход элементов в случае основного класса-хранилища из элементов, как правило, не регламентируется*

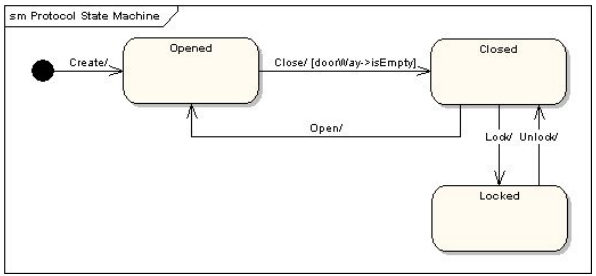
*Давайте перечислим 3 основных способа

Состояние (State)

Состояние @

- Задача: реализовать стейт-машину (конечный автомат) на базе объекта
- Когда:
  - Необходимо изменять поведение объекта в зависимости от его состояния
  - Объект должен моделировать сущность, поведение которой выразимо через конечный автомат
    - Конечное число состояний, переход между состояниями -- моментальный
- Когда недостаточно: логика поведения объекта не укладывается в модель конечного автомата



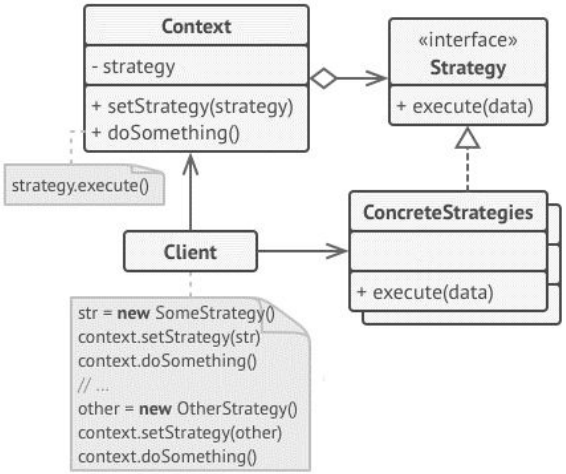


- Преимущества:
  -
  -
  -
- Недостатки:
  - Конечное число состояний (по построению)

Стратегия (Strategy)

Стратегия @

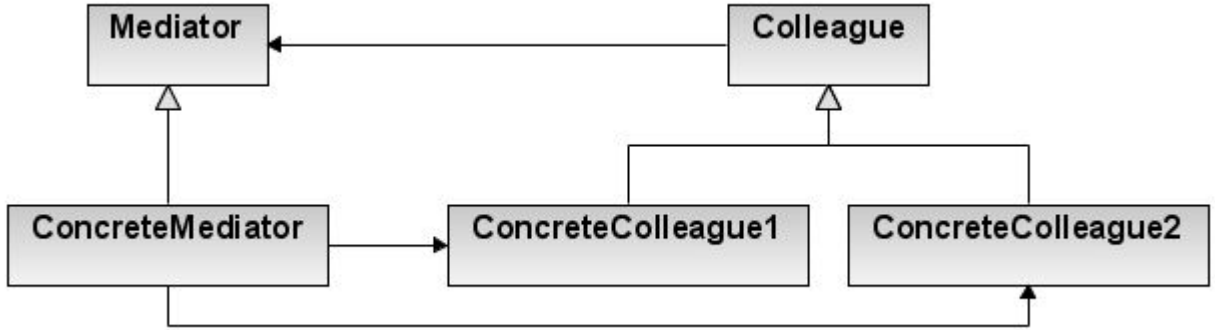
- Задача: реализовать стратегию.
- Когда:
  - Когда нужно использовать разные вариации какого-то алгоритма внутри одного объекта.
  - Когда есть множество похожих классов, отличающихся только некоторым поведением.
  - Когда не нужно обнажать детали реализации алгоритмов для других классов.
  - Когда различные вариации алгоритмов реализованы в виде развесистого условного оператора.

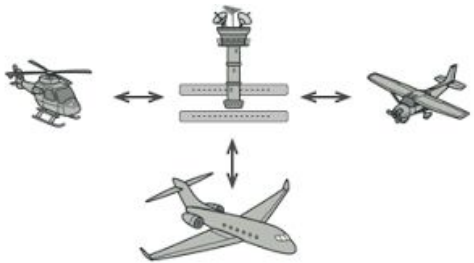


- Преимущества:
  - Горячая замена алгоритмов “на лету”.
  - Изолирует код и данные алгоритмов от остальных классов.
  - Уход от наследования к делегированию.
  - Реализует принцип открытости/закрытости.
- Недостатки:
  - Усложняет программу за счёт дополнительных классов.
  - Клиент должен знать, в чём состоит разница между стратегиями, чтобы выбрать подходящую.

Медиатор (Mediator)

- Задача: реализовать шину передачи между объектами, устраняя таким образом лишние связи между взаимодействующими объектами, улучшает показатели связности.
- Когда:
  - Необходимо передавать сообщения (в т.ч. вызывать методы) и следить/управлять этим процессом
  - Выполнять диспетчеризацию передачи
  - Реализовывать механизм подписок одних объектов на события, происходящие с другими
- Когда избыточно: объект А напрямую может вызвать метод объекта В (промежуточный объект не нужен)



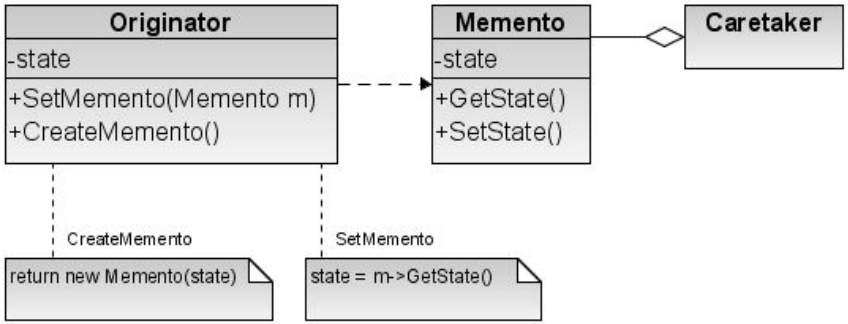


- Преимущества:
  - Устраняется связанность между взаимодействующими объектами, централизуется управление.
- Недостатки:
  - Избыточность, если взаимодействие очень простое

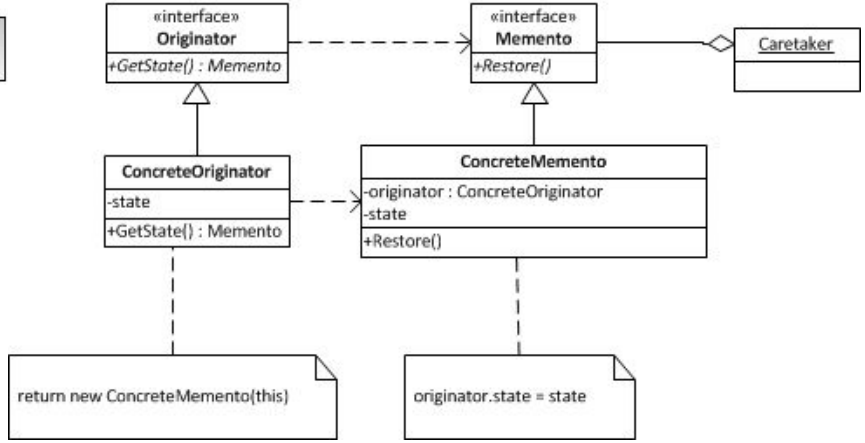
Снимок (Memento)

Снимок @

- Задача: реализовать сохранение / восстановление состояния объекта, не нарушая инкапсуляцию.
- Когда:
  - Поддержка восстановления по контрольной точке
  - Поддержка UNDO



1. Стандартный вид

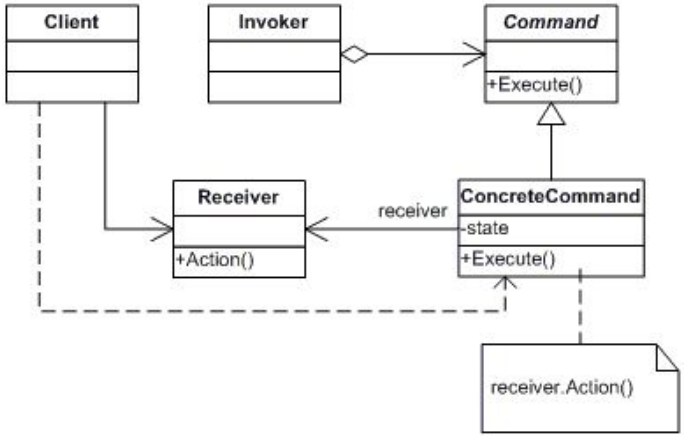


2.Нестандартный вид (класс-опекун ограничен)

Команда (Command)

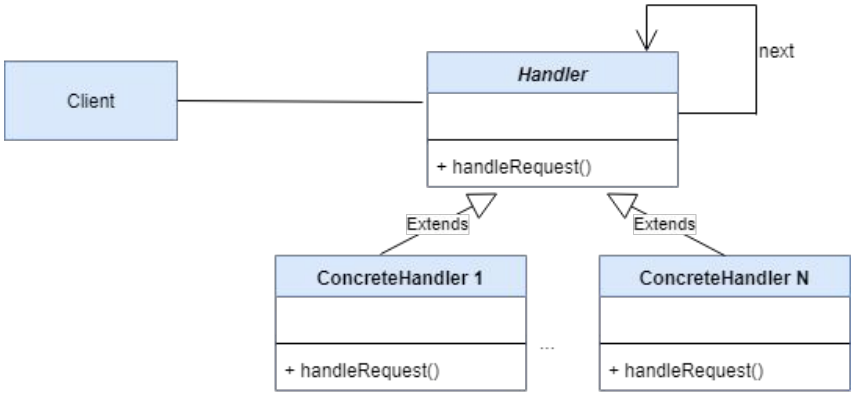
Команда @

- Задача: обернуть команду и параметры в объект, передаваемый между классами. Объект используется для инкапсуляции информации, необходимой для выполнения действия или вызова события. Информация: имя метода, объект, который является владельцем метода, значения параметров метода
- Когда:
  - Нужно, чтобы класс-отправитель и класс-получатель не зависели друг от друга напрямую
  - Нужно организовать callback к классу, который включает в себя класс-отправитель



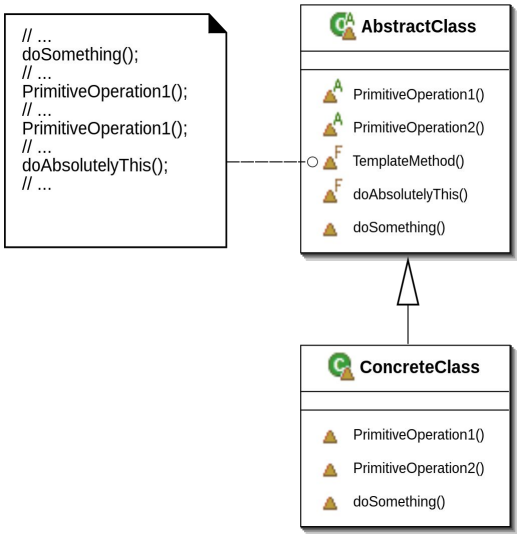
Цепочка ответственности (Chain of responsibility)

- Задача: организовать обработку сообщений по порядку либо по уровням ответственности
- Когда:
  - имеется группа объектов, которые могут обрабатывать сообщения определенного типа;
  - Сценарий 1:
    - все сообщения должны быть обработаны хотя бы одним объектом системы;
    - сообщения обрабатываются по схеме «обработай сам либо перешли другому»: одни сообщения обрабатываются на том уровне, где получены, а другие пересылаются другим
  - Сценарий 2: сообщение обрабатывается всеми обработчиками в некотором порядке, причём если n-1 обработчик не смог обработать, n-й и далее уже не обрабатывают сообщение.



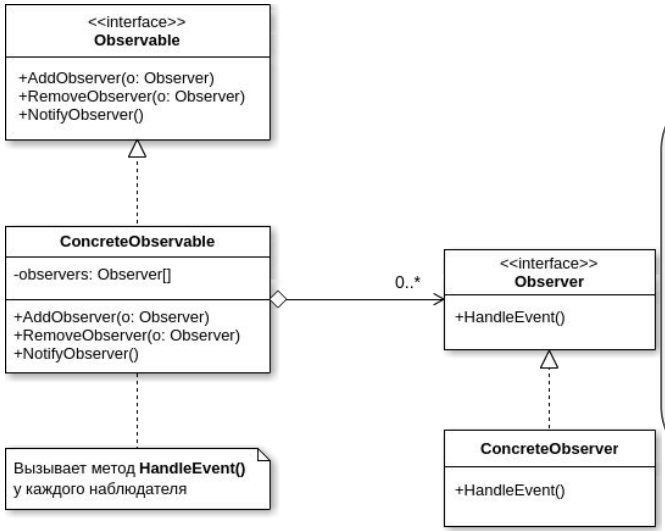
Шаблонный метод (Template method)

- Задача: Реализовать “шаблон” выполнения алгоритма в виде метода
- Когда:
  - Класс является интерфейсом к группе объектов, выполняющих примерно одно и то же (и в том же порядке), но немного по-разному;



Наблюдатель (Observer)

- Задача: реализовать зависимость "один-ко-многим" между объектами так, что при изменении состояния одного объекта все зависящие от него объекты уведомляются и обновляются автоматически;
- Когда:
  - Когда нужно реализовать “подписку” у группы объектов (1 или более) на события, происходящие в некотором объекте
    - При этом необходимо избежать сильного сопряжения



- **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.

Структурные паттерны

1/ Адаптер

2/ Декоратор

3/ Фасад

4/ Прокси

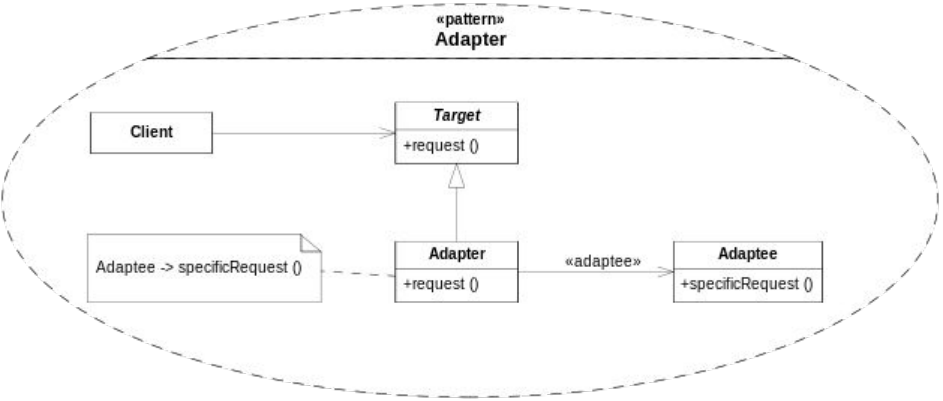
5/ Компоновщик

6/ Приспособленец

7/ Мост

Адаптер (Adapter)

- Задача: позволить объектам с несовместимыми интерфейсами работать вместе
- Когда:
  - система поддерживает требуемые данные и поведение, но имеет неподходящий интерфейс.
  - для организации использования функций объекта, недоступного для модификации, через специально созданный интерфейс

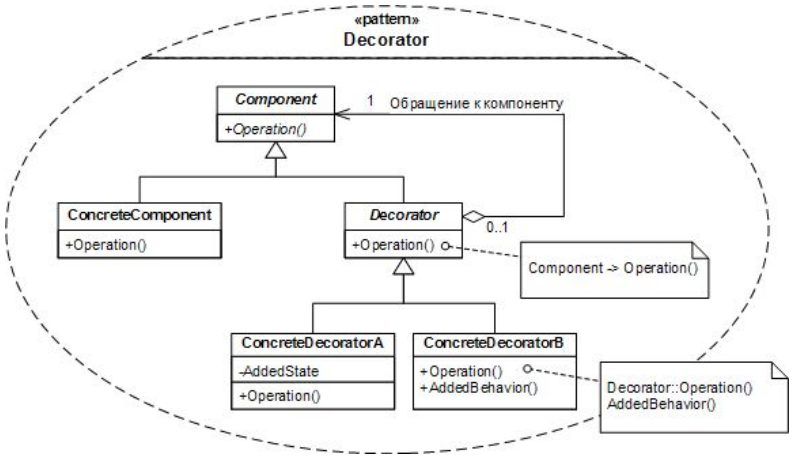


\+ инкапсуляция реализации внешних классов (компонентов, библиотек), система становится независимой от интерфейса внешних классов;

\+ переход на использование других внешних классов не требует переделки самой системы, достаточно реализовать один класс `Adapter`.

Декоратор (Decorator)

- Задача: Подключить дополнительное поведение к объекту
- Когда:
  - Требуется добавить объекту некоторую дополнительную функциональность, которая будет выполняться до, после или даже вместо основной функциональности объекта
  - Но при этом не хочется наследоваться



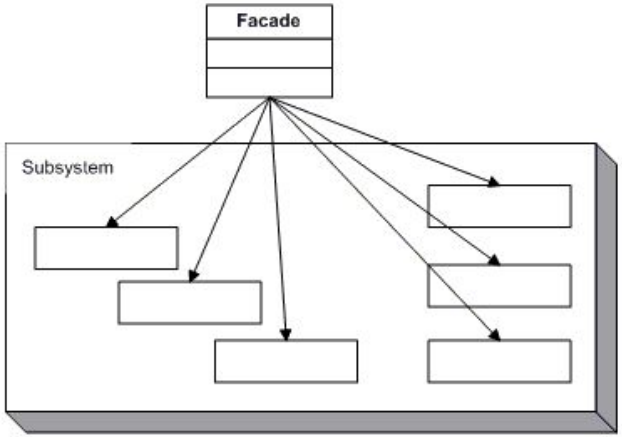
\+ Обеспечивает динамическое добавление функциональности до или после основной функциональности конкретного объекта

\+ Позволяет избежать перегрузки неинтерфейсными классами на верхних уровнях иерархии

\- оборачивает ровно тот же интерфейс, что предназначен для внешнего мира, что вызывает смешение публичного интерфейса и интерфейса кастомизации, которое не всегда желательно.

Фасад (Fasade)

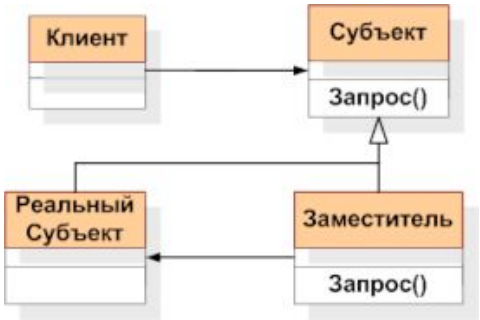
- Задача: обеспечить унифицированный интерфейс с набором разрозненных реализаций или интерфейсов для связи с некоторой подсистемой
- Когда:
  - требуется скрыть сложность системы путём сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам подсистемы
  - нежелательно сильное связывание с этой подсистемой или реализация подсистемы может измениться



\+ Фасад — это внешний объект, обеспечивающий единственную точку входа для служб подсистемы.
\+ Реализация других компонентов подсистемы закрыта и не видна внешним компонентам.
\- Иногда не подходит в виду врожд ённой непрозрачности (см. паттерн “Прокси”).

Прокси, заместитель (Proxy)

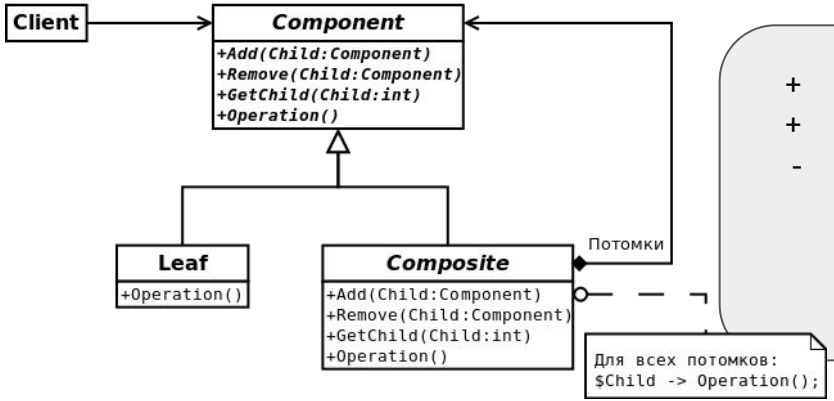
- Задача: предоставить объект, который контролирует доступ к другому объекту, перехватывая все вызовы
- Когда:
  - необходимо контролировать доступ к объекту, не изменяя при этом поведение клиента.
  - необходимо иметь доступ к объекту так, чтобы не создавать реальные объекты непосредственно, а через другой объект, который может иметь дополнительную функциональность.



\+ может выполнять оптимизацию, различные проверки;
\+ Может обеспечить создание реального «Субъекта» только тогда, когда он действительно понадобится
\+ защищает «Субъект» от опасных клиентов (или наоборот);
\- Может ухудшать производительность (доп. вызов и т.д.)

Компоновщик, композит (Composite)

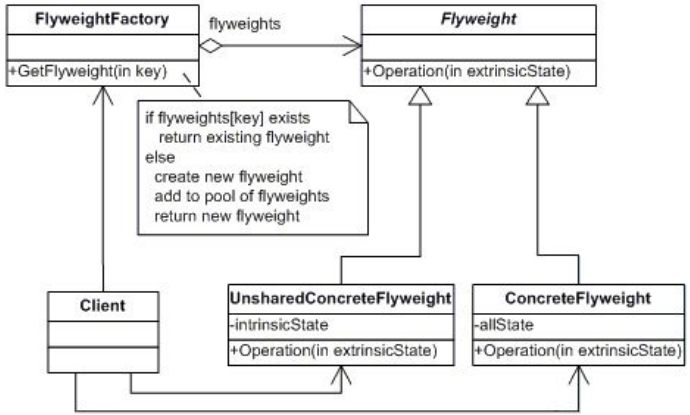
- Задача: организовать объекты в древовидную структуру для представления иерархии от частного к целому
- Когда:
  - необходимо предоставить иерархию по включению.
  - собрать иерархию классов, которые одновременно могут состоять из примитивных и сложных объектов.



\+ упрощает архитектуру клиента;
\+ упростить процесс добавления новых видов объекта
\- избыточная реализация методов, которые отдельные вершины не поддерживают (как минимум, нужно вставить стабики в композит, которые будут либо печатать, что операция не поддерживается, либо кидать исключение, либо запретить их вызов из клиента)

Приспособленец (Flyweight)

- Задача: представить объект как уникальный экземпляр в разных местах программы
- Когда:
  - Оптимизация работы с памятью путём предотвращения создания экземпляров элементов, имеющих общую сущность.



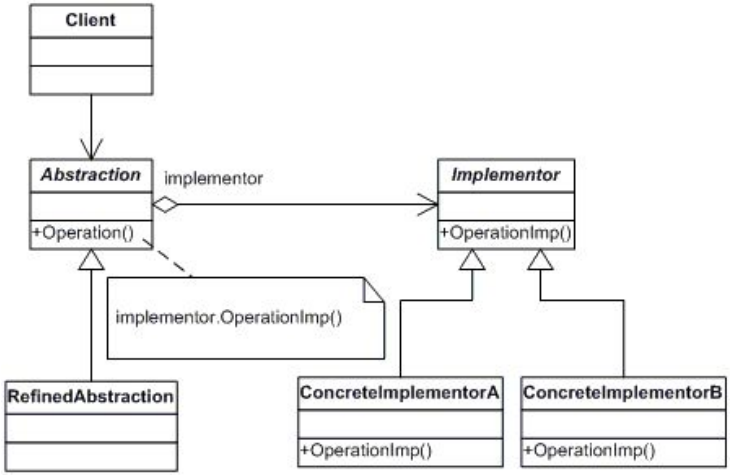
\+ Оптимизирует приложение по памяти

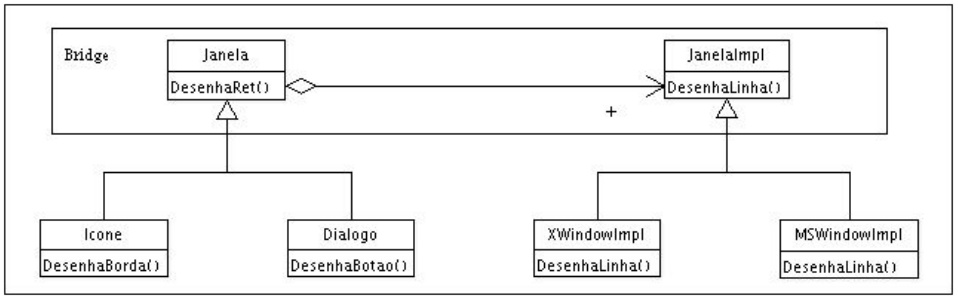
\+ дополняет шаблон “Фабричный метод” таким образом, что при обращении клиента к Factory Method для создания нового объекта ищет уже созданный объект с такими же параметрами, что и у требуемого, и возвращает его клиенту

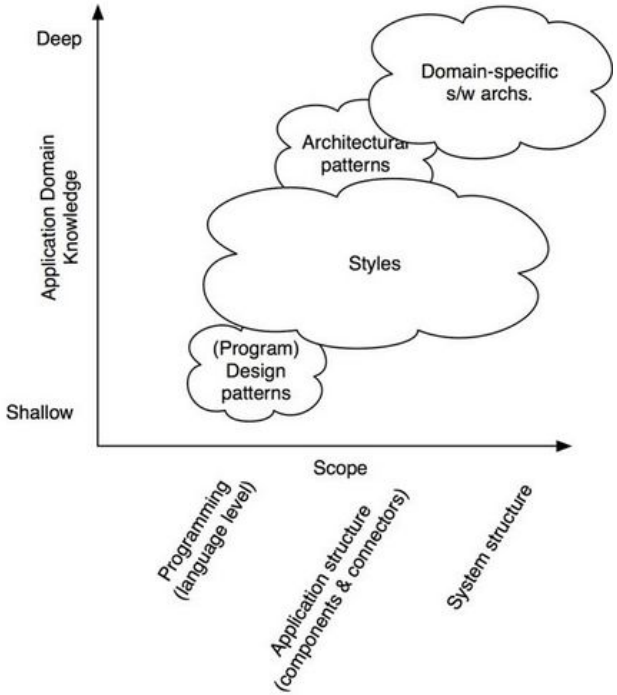
\-

Мост (Bridge)

- Задача: Разделить абстракцию и реализацию так, чтобы они могли бы модифицироваться независимо
- Когда:
  -
  - часто меняется не только сам класс, но и то, что он делает.







N. Medvidovic