1. Структурное программирование. Нисходящая разработка, использование базовых логических структур, сквозной структурный контроль.

Структурное программирование — разбивка по действиям от сложного к простому. В основе структурного программирования лежит алгоритмическая декомпозиция — разбиение задачи на подзадачи по действию, отвечая на вопросы, что нужно делать.

Три основные части технологии:

- Нисходящая разработка.
- Сквозной структурный контроль.
- Использование базовых логических структур.

Этапы создания программного продукта при структурном программировании:

1) Анализ
2) Проектирование (разработка алгоритмов)
3) Кодирование
4) Тестирование
5) Сопровождение
6) Модификация

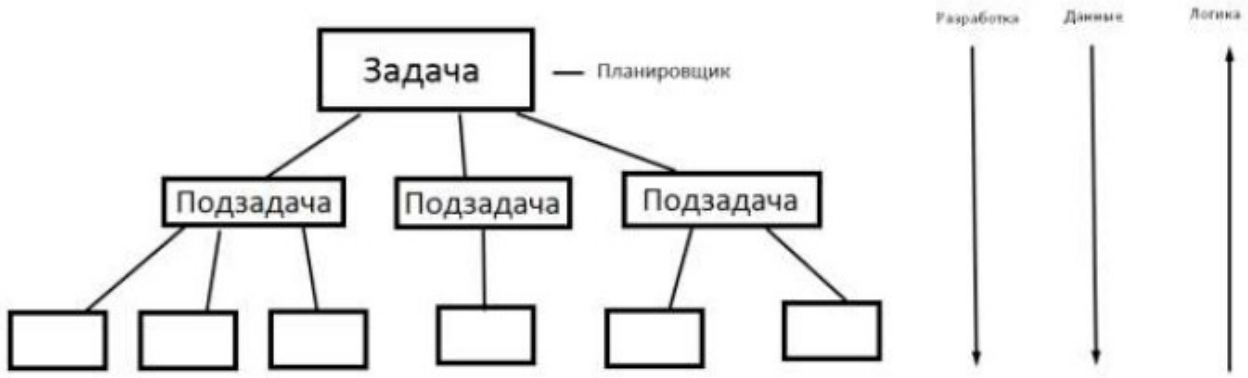
Нисходящая разработка — используется на этапах от проектирования до тестирования. Основные положения нисходящей разработки — программа строится в виде дерева, движение начинается от головного модуля, модуль начинает разрабатываться, если уже готов модуль, обращающийся к нему, тестирование в том же порядке. В процессе нисходящей разработки используются алгоритмы декомпозиции — разбиение задачи на множество подзадач. Выделенные подзадачи разбиваются дальше, и таким образом формируется иерархическая структура программы (данные нисходящие, логика восходящая, разработка нисходящая). Логика поднимается на более высокий уровень. Данные наоборот опускаются на более низкие уровни программы. Для каждой полученной подзадачи создаётся отдельный отладочный модуль. Готовятся отдельные тестирующие пакеты (до этапа написания кода). Подзадача не принимает решения за модуль уровнем выше (функция отрабатывает свой код, и возвращает результат, который затем анализируется верхними уровнями). Все данные должны передаваться явно. Блок, функция, файл — уровни абстракции, которые нельзя нарушать. Ограничения вложенности — 3 (глубина вложенности), если больше, то необходимо выделять подфункции.

Используется на этапах проектирования, кодирования и тестирования.

Основные принципы нисходящей разработки:

- тесты составляются до написания кода.
- подзадачи разбиваются на модули.
- не должно быть участков кода, выполняющих одинаковые действия (отсутствие дублирования).
- результат работы функций движется вверх по ветке программы.
- обработка всех ошибок с возвращением их в вышестоящий уровень.
- без функций с сюрпризами.
- глубина вложенности не более трёх уровней.
- при разбиении программы на подзадачи выделяется не более семи подзадач.
- комментарии находятся перед функциями.
- логика находится на верхних уровнях.
- данные передаются и возвращаются явно.
- уровни абстракции: файл → функция → блок.

Три подхода программирования модулей:
1) Иерархический (по уровню абстракции)
2) Операционный (в порядке вызова модулей)
3) Смешанный



Использование базовых логических структур — согласно Майеру любой алгоритм можно реализовать с помощью трёх логических структур — следование, развилка (условие) и повторение (цикл). Введено IBM. Безусловный переход (go to) не используется ни в каких случаях.
Развилка:
if, switch.
Виды циклов:
С постусловием: do..while();
С предусловием: while();
Со счётчиком: for();
Безусловный: loop (цикл с выходом из тела цикла);

Повторение:
until – цикл «до»
while- цикл «пока»
for – цикл «пока» с переменной-счётчиком
loop – безусловный цикл

Выход из цикла должен быть один.

Сквозной структурный контроль — серия принципов, позволяющая облегчить контроль за разработкой ПО при структурной разработке.
- размер рабочей группы не должен превышать 7 человек.
- уровень руководителя группы должен быть выше, чем у подчинённых, однако руководитель не должен быть вовлечён в написание кода.
- контроль над написанием кода осуществляют сами программисты.
- производится формализация информации и устраиваются «контрольные сессии» с коллегами.
- задачи контрольный сессий: выявление недостатков (особенно на ранних стадиях), мозговой штурм.

2. Преимущества и недостатки структурного и объектно-ориентированного программирования.

Преимущества и недостатки структурного подхода:
+ легко распределять работу между программистами.
+ естественные контрольные точки.
+ легко выявлять ошибки.
+ легко поддаётся тестированию (комплексное тестирование).
+ раннее начало процесса кодирования.
+ снижается вероятность допустить логическую ошибку.
+ возможен контакт с заказчиком на ранних стадиях, управление сроками.
+ упрощённое чтение кода.
- отсутствие гибкости системы, после внесения некоторого количества модификаций, происходит смещение уровней абстракции, нарушается структура программы, что приводит к потере надёжности, вследствие чего сопровождение затруднительно, и стоит много денег.
- сложно изменять формы данных и структур.
- сложно сопровождать программный продукт.

Преимущества и недостатки ООП:
+ возможность лёгкой модификации.
+ возможность отката при наличии версий.

+ более лёгкая расширяемость.
- требуется другая квалификация.
- резко увеличивается время на анализ и проектирование систем.
- увеличивается время выполнения программ.
- размер кода увеличивается.
- неэффективно с точки зрения памяти из-за образования мёртвого кода (мёртвый код это код, не использующийся в программе)
- сложность распределения работ на начальном этапе.
- повышается себестоимость.

3. Основные понятия ООП. Инкапсуляция, наследование, полиморфизм. Понятие отношения между классами. Понятие домена.

Технология ООП была предложена Хоаром в 1966 году «Совместное использование кода».
Основные принципы ООП:
1) Инкапсуляция — объединение данных и действий над этими данными, по определению лекции Маслова — для каждого типа данных свои функции-действия.
2) Наследование — модификация развития программы за счёт надстроек, вместо изменения написанного кода создаются новые надстройки над ним.
3)Полиморфизм — использование объектов с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Организация взаимодействия между объектами — перенесение взаимодействия объектов из физического мира в программирование.

Объект — конкретная реализация абстрактного типа, обладающая характеристиками состояния, поведения и индивидуальности.
Состояние — один из возможных вариантов условий существования объекта.
Поведение — описание объекта в терминах изменения его состояния и передача сообщений (данных) в процессе воздействия.
Индивидуальность — сущность объекта, отличающаяся от других объектов.

Модель Мура состоит из:
- множества состояний, каждое состояние представляет стадию в жизненном цикле типичного объекта.
- множества событий, каждое события представляет собой инцидент или указание на то, что происходит эволюционирование.
- множества правил перехода — определяет какое новое состояние получает в следствие какого-нибудь события (событие может не изменять объект)

- множества действий — действие это деятельность или операция, которая должна быть выполнена над объектом, чтобы он мог достичь состояния. (каждому действию соответствует состояние)

Категории объектов:
Реальные объекты — абстракция реально существующего физического объекта.
Роли — абстракции цели или назначения человека, части оборудования или организации.
Инциденты — абстракция чего-то произошедшего или случившегося.
Взаимодействия — объекты, получаемые из отношений между другим объектами.
Спецификации — используется для представления правил, критериев качества, стандартов.

Отношения между объектами:
Отношения использования (старшинства) — каждый объект включается в отношения, может играть три роли:
1. Активный объект — объект может воздействовать на другие объекты, но сам не поддаётся воздействию (воздействующий).
2. Пассивный объект — объект может только подвергаться управлению, но не выступает в роли воздействующего (исполнитель).
3. Посредники — такой объект может выступать как в роли воздействующего, так и в роли исполнителя (создаются для помощи воздействующим объектам). Чем больше объектов-посредников, тем легче модифицировать программу.

Отношения включения — один объект включает другие объекты.

Класс — такая абстракция множества предметов реального мира, что все предметы этого множества (объекты) имеют одни и те же характеристики, все экземпляры подчинены и согласованы с одним и тем же поведением.

Отношения между классами:
1. Наследование — на основе одного класса мы строим новый класс, путём добавления новых характеристик и методов.
2. Использование — один класс вызывает методы другого класса.
3. Представление (наполнение) — один класс содержит другие классы.
4. Метакласс — класс, существующий для создания других классов.

Домен — отдельный, реальный, гипотетический или абстрактный мир, населённый отчётливым набором объектов, которые ведут себя в соответствии

с предусмотренным доменом правилами. Каждый домен образует отдельное и связное единое целое.

4. Цикл разработки ПО с использованием ООП: анализ, проектирование, эволюция, модификация. Рабочие продукты объектно-ориентированного анализа.

Этапы разработки:
1) Анализ - построение модели программы.
2) Проектирование — перенос документов анализа в документы написания кода.
3) Эволюция — этап объединяет кодирование и тестирование. Позволяет при этом вернуться к этапу анализа и проектирования. Изменение должно сводиться только к добавлению класса или изменению его реализации.
4) Модификация — добавление нового функционала после получения завершённого продукта.

Эволюция и модификация — разные вещи, модификация — продукт готов, и получает изменения после выпуска в продакшн, эволюция — добавление нового функционала на этапе кодирования и тестирования.

Преимущества эволюции:
- обратная связь с пользователем.
- различные версии структур системы (плавный переход от старой системы к новой).
- меньше вероятности отмены проекта.

Изменения в процессе эволюции с возрастанием сложности:
- добавление класса.
- изменение реализации класса.
- изменение представления класса.
- реорганизация структуры класса.
- изменение интерфейса.

Анализ — построение модели системы.

Результаты проектирования

- ПО
  - схема домена
  - проектная матрица
- класс
  - модель состояний
  - диаграмма потоков данных действий
- домен
  - модель связи подсистемы
  - модель взаимодействия подсистемы
  - модель доступа к подсистемам
- процесс
  - описание
  - псевдокод процесса
- подсистема
  - информационная модель (получаем описание объектов, атрибутов, связей).
  - модель взаимодействия объектов (получаем список событий).
  - модель доступа
  - таблица процессов состояний для всей подсистемы.

Действия при анализе и результаты (рабочие продукты):
1. Разбиваем задачу на домены
- Схема доменов
- Проектная матрица
2. Разбиваем домены на подсистемы
- Модель взаимодействия подсистемы
- Модель связей подсистемы
- Модель доступа к подсистемам
3. Для каждой подсистемы получаем
- Информационная модель
- Модель взаимодействия объектов — получаем описание классов и их атрибутов, а также описание их связей
- Модель доступа к объектам — получаем таблицу процессов состояний
4. Для каждого объекта получаем модель переходов состояний
5. Для каждого состояния каждой модели состояния строим диаграмму потоков данных действий
6. Для каждого процесса получаем описание процесса

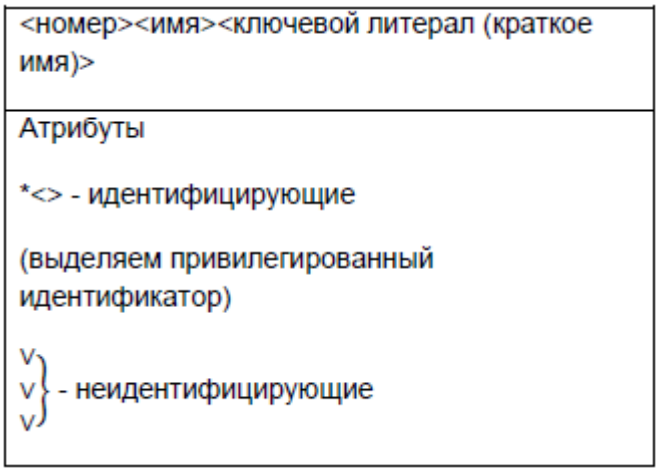
Не путать домен (мир) с сервисом (функционалом)!

На основе полученных в ООА документов мы приходим к проектированию. Четыре основных рабочих продукта:
- Диаграмма класса — проектируется вокруг объекта класса и класса
- Схема структуры класса — для внутренней структуры класса
- Диаграмма зависимостей — схема использования
- Диаграмма наследований — схема наследования классов

5. Концепции информационного моделирования. Понятие атрибута. Типы атрибутов. Правила атрибутов. Понятие связи. Типы связей. Формализация

связей. Композиция связей. Подтипы и супертипы. Диаграмма сущность-связь.

Концепции информационного моделирования:
- Выделение физических объектов.
- Короткое описание классов, чтобы установить, является ли объект экземпляром класса.
- Выделение характеристик объектов и идентификаторов — множество из одного или нескольких атрибутов класса, однозначно определяющих экземпляр класса.
- Графическое обозначение класса на информационной модели:



- Строим таблицу атрибутов: для каждого объекта должен быть определён однозначно атрибут (<имя>(<идентификатор>, <список атрибутов>) - текстовое описание)

Основные концепции:
- Выделение сущностей, которыми мы работаем.
- Описание или понятие этой сущности:
    - Выделение атрибутов, каждая характеристика, которая является общей для всех возможных экземпляров классов выделяется как отдельный атрибут.
    - Идентификатор — это множество из одного или множества атрибутов, которое определяет класс.
    - Выделение привилегированных атрибутов.
- Графическое представление:
- Все сущности нумеруются. Для классов, связей и проч. мы создаём ключевой литера, состоящий из 1-3 букв.

Атрибут — какое-либо свойство класса.
Описательные атрибуты — какая-то характеристика, внутренне присущая каждому объекту (к примеру для человека это пол, возраст и тд). Если значение

описательного атрибута меняется, то какой-то аспект изменяется, но сам объект остаётся.
Указывающие атрибуты — используются как идентификатор, или как часть идентификатора (имя студента).
Вспомогательные атрибуты — используются для формализации связи одного объекта с другими объектами. Для активных объектов будем выделять время жизни. При изменении соответственно меняются связи или состояние объекта.

Если описательный, от описание — информационная строка, которая показывает реальную характеристику, как определяется характеристика и почему она уместна для данного объекта.

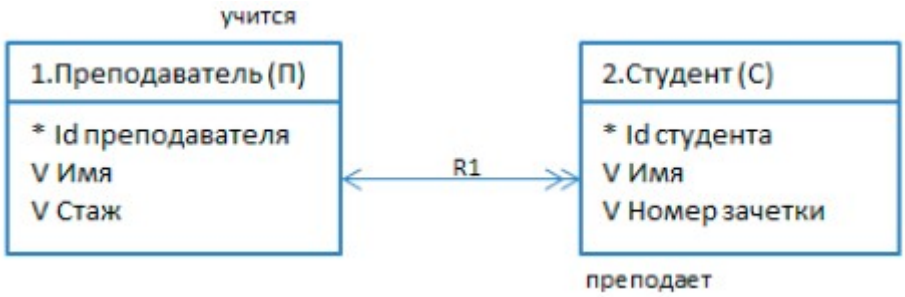
Если указывающий, то описание — форма указания, кто назначает атрибуты, использующиеся в идентификаторе.

Если вспомогательный, то описание — какое отношение или состояние сохраняются.

Правила атрибутов:
1. Один объект класса имеет одно единственное значение для каждого атрибута в любой момент времени.
2. Атрибут не должен содержать никакой внутренней структуры.
3. Когда объект имеет составной идентификатор, каждый атрибут, являющийся частью идентификатора представляет характеристику всего объекта, а не его части.
4. Каждый атрибут, который не является частью идентификатора, представляет характеристику объекта, указанного идентификатором, а не характеристику другого атрибута.

Связи:
Связь — абстракция набора отношений, которые систематически возникают между различными видами реальных объектов.

Каждая связь задаётся из перспективы участвующих объектов.
Каждой связи присваивается уникальный идентификатор, состоящий из буквы и номера:

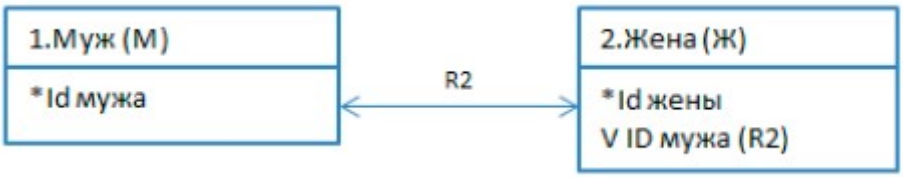


Типы связей:
По количеству связываемых объектов:
- Один к одному
- Один ко многим
- Многие ко многим
По виду участия объектов:
- безусловная — связь всегда держится
- условная — один из объектов может не участвовать в связи
- биусловная — возможно неучастие с обеих сторон

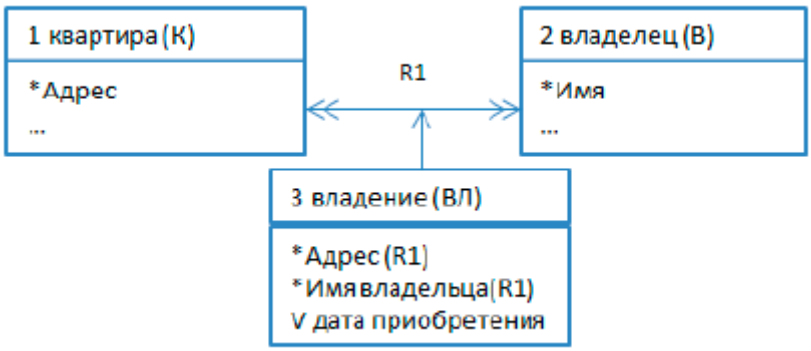
Иногда с какой-то стороны объект может не участвовать в связи, тогда связь — условная (преподаватель не руководит студентами — условная связь со стороны преподавателя, т.к. у студента должен быть руководитель, а преподаватель не обязан руководить). Если с обеих сторон объекты могут не участвовать в связи, то связь — биусловная (Учебный курс — студент: курс не читается в этом семестре, или студент не выбрал этот курс). В итоге получается 3 безусловных (Один к одному, один ко многим, многие ко многим), 4 условных (Один условно ко многим, один ко многим условно, многие условно ко многим, многие ко многим условно) и 3 биусловных (один условно к одному условно, один условно ко многим условно, многие условно ко многим условно).

Формализация связей:

1) Безусловная формализация связей один к одному: добавляем атрибут связи в любой из объектов (тот, который более осведомлён о системе).



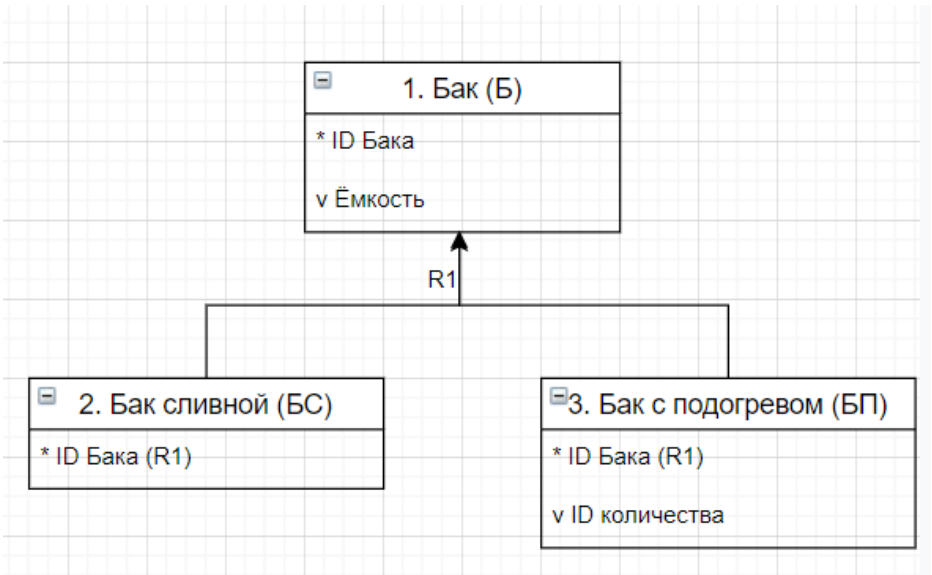
2) Безусловная формализация связей один ко многим: формализуется со стороны многих (добавляем им атрибут связи).
3) Формализация связей многие ко многим: формализуется через ассоциативный объект:



4) Если связь условна, биусловна или имеет динамическое поведение, то она формализуется ассоциативным объектом.

Композиция связей — в некоторых случаях можно создать композитную связь, являющуюся комбинацией нескольких последовательных связей.

Подтипы и супертипы:
В супертипе — общие атрибуты для разных объектов. Супертип — всегда абстрактное понятие.



Подтип - это класс объектов, представляющий разбитую группу в рамках супертипа. (К примеру Бак сливной на диаграмме выше).

При информационном моделировании получаем:
1) Информационную модель.
2) Описание объектов и их атрибутов.
3) Описание связей и их формализацию.

Диаграмма сущность-связь — диаграмма предназначена для разработки моделей данных и обеспечивают стандартный способ определения данных и отношений между ними.
Сущность — абстракция, являющаяся элементом модели.
Связь — соединяет между собой несколько сущностей.
Диаграмма — группирует представляющие интерес наборы сущностей.

Сущности:
- структурные
- поведенческие
- аннотирующие

Структурные сущности — части модели, представляющие либо концептуальные, либо физические элементы. Основным видом структурной сущности в диаграмме классов является класс.

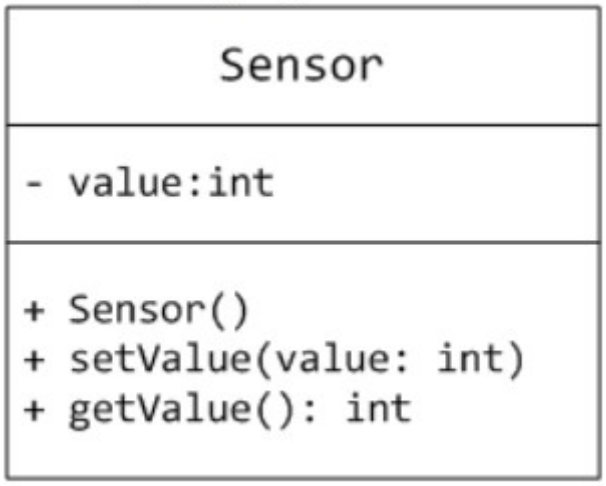
Поведенческие сущности — часть моделей, предоставляющая поведение модели во времени и пространстве. Одной из таких сущностей является взаимодействие — поведение, заключающееся в обмене сообщениями между наборами объектов или ролей в определённом контексте для достижения некоторой цели. Изображается в виде стрелки с именем операции.

Аннотирующие сущности — поясняющая часть диаграмы.

Структурные сущности — классы:
класс — описание набора объектов с одинаковыми атрибутами, операциями связями и семантикой.
Класс отображается в виде прямоугольника, разделённого на 3 блока горизонтальными линиями:
- Имя класса
- Атрибуты (свойства) класса
- Операции (методы) класса

Для атрибутов может быть указан один из трёх видов видимости:
- private
protected
+ public

Имя — текстовая строка.



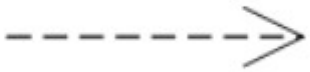
Для абстрактного класса имя задаётся курсивом.
Атрибут (свойство) — именованное свойство класса, описывающее диапазон значений, которые может принимать экземпляр атрибута. Класс может иметь любое количество атрибутов или не иметь одного. Можно уточнить спецификацию атрибута, указав его тип, кратность (если атрибут — массив значений) и начальное значение по умолчанию.

Операция (метод) — реализация метода класса, класс может иметь любое число методов, либо не иметь их совсем. Методы описываются аналогично атрибутам в нижнем блоке класса. Можно специфицировать операцию, устанавливая её сигнатуру, включающую имя, тип, значение по умолчанию всех параметров, а применительно к функциям — тип возвращаемого значения.

Абстрактные методы класса обозначаются курсивом.
Статические методы обозначаются подчёркиванием.

Виды связей в диаграмме сущность-связь:
- зависимость
- ассоциация
- обобщения
- реализация

Зависимость — связь между двумя элементами модели, в которой изменение независимого элемента может привести к изменению зависимого.



Ассоциация — структурная связь между элементами модели, которая описывает набор связей, существующих между объектами. Ассоциация показывает, что объекты одной сущности (класса) связаны с объектами другой сущности таким образом, что можно перемещаться от объектов одного класса к другому.

Агрегация — особая разновидность ассоциации, представляющая структурную связь целого с его частями. Как тип ассоциации, агрегация может быть именованной. Одно отношение ассоциации не может содержать в себе более двух классов (контейнер и содержимое). Данный вид связи встречается, когда один класс является коллекцией или контейнером других. По умолчанию агрегацией называют агрегацию по ссылке.

Композиция — более строгий вариант агрегации, известна как агрегация по значению. Композиция — форма агрегации с чётко выраженными отношениями владения и совпадением времени жизни частей и целого. Композиция имеет жёсткую зависимость времени существования экземпляров класса контейнера и экземпляров содержащихся классов. Если контейнер будет уничтожен, то всё его содержимое будет также уничтожено.

Обобщение — выражает специализацию или наследование.

Реализация — связь между классами, когда один из них (поставщик) определяет соглашение, которого второй (клиент) обязан придерживаться. Эти связи между интерфейсами и классами, которые реализуют эти интерфейсы. Поставщик обычно является абстрактным классом.

6. Модель поведения объектов. Жизненный цикл и диаграмма перехода в

состояния (ДПС). Виды состояний. События, данные событий. Действия состояний. Таблица перехода в состояния (ТПС). Правила переходов.

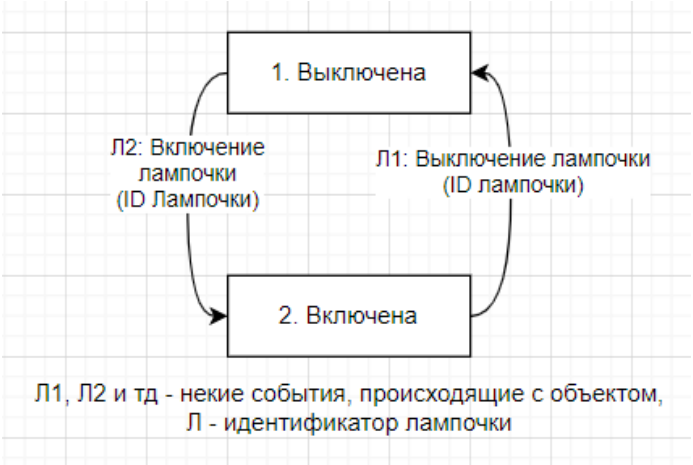
Модель поведения объектов — модель Мура поведения объектов состоит из:
- множества состояний
- множества событий (инценденты)
- правил перехода
- действий

Жизненный цикл объектов:
- в каждый момент времени объект находится на какой-то одной стадии.
- переход из одной стадии в другую происходит скачкообразно и является реакцией на какой-то инцидент.
- переходы возможны не из всех состояний.

Формы жизненных циклов:
1. Циркулярный
2. Рождение-смерть

Случаи, в которых формируются жизненные циклы:
1. Создание или уничтожение во время выполнения
2. Миграция между подклассами
3. Объект производится или возникает поэтапно.
4. Объект — задача или запрос
5. Динамическая связь

Диаграмма перехода в состояние:



Состояние — положение объекта, в котором применяется особый набор правил и линий поведения, предписаний физических законов (для состояния ставятся в соответствие уникальные в рамках данной модели состояний имя и номер).

Виды состояний:
1) Состояние создания (переход в состояния создания — не из состояния) — показывается внешней стрелкой, которая идёт в состояние создания.
2) Заключительные состояния:
- состояния, из которых объект больше не переходит.
- состояния, в которых объект уничтожается.
Данное состояние обозначается пунктирным прямоугольником.
3) Текущее состояние (состояние, в котором объект может находиться, кроме состояний создания и заключения).

Если для объекта выделяется модель состояний, то на информационной модели для класса этого объекта нужно добавить атрибут «статус», хранящий состояние.

Событие — это абстракция инцидента или сигнала в реальном мире, сообщающего о перемещении чего-либо в новое состояние.

Характеристики событий:
- Значение — короткая фраза, которая сообщает, что происходит с объектом в реальном мире.
- Предназначение — модель состояний, принимающее событие (единственная)
- Метка события — ключевой литерал и уникальный номер, внешние события помечаются буквой E.
- Данные — необходимы, чтобы выполнилось действие, которое соответствует состоянию. Могут быть идентифицирующие и вспомогательные (дополнительные).
- События должны переносить данные.

События, переводящие объект из одного состояния в другое (не создающие объект) должны нести идентификатор объекта.

Правила связи состояний и событий:
1) Все события, которые вызывают переход в одно и то же состояние должны нести одни и те же данные.
2) Если это не состояние создания, то событие должно переносить идентификатор объекта.
3) Событие, переводящее в состояние создания, не несёт идентификатор объектов.

Можно добавлять состояния, соответствующие переходным процессам.

Действие — деятельность или операция, которая выполняется при достижении объектом состояния, Каждому состоянию ставится в соответствие одно действие. Действие должно выполняться любым объектом одинаково.

Действие может:
1) выполнять любые вычисления
2) порождать любые события для текущего объекта
3) порождать события для чего-либо вне области анализа
4) порождать события для классов это домена
5) читать, записывать атрибуты собственного класса и других классов

Требования к действиям:
1) Действие не должно оставлять данные объекта противоречивыми
2) Действие должно менять статус или атрибут состояния (если не переводит в то же состояние)

Если псевдокод действия маленький, то его можно написать под состоянием.

Действие-событие-время:
1) Только одно действие конечного автомата может выполняться в конкретный момент.
2) Действия различных КА могут выполняться параллельно.
3) События никогда не теряются.
4) Если событие порождено для объекта, который в данный момент выполняет действие, то данное событие не будет принято, пока действие не закончится.
5) Не все события обрабатываются, событие может быть проигнорировано.

Таблица переходов состояний (ТПС):

Матрица: строки — состояние, столбцы — события.

В таблице должны быть указаны все состояния и события, все клетки таблицы должны быть заполнены.

Варианты заполнения:
- номер нового состояния
- игнорирование (-)
- событие не может произойти (x) — такие клетки необходимо обрабатывать и добавлять новые события для того, чтобы избежать ошибок

Формы жизненных циклов:
- циркуляционные
- «рождение-смерть»

Для объектов, которые осведомлены о системе — циклический
Для объектов с ЖЦ в одном классе посмотреть, отличаются ли ЖЦ, если да, то разделить на разные классы.
При сравнивании ЖЦ выделить общую и разную части.
Миграция объекта между подклассами.

При выделении ЖЦ:
1. Создание и/или уничтожение во время выполнения
2. Миграция между подклассами (например наполнение атрибутов)
3. Объект производится или возникает поэтапно
4. Объект — задача или запрос
5. Формализация динамических связей (ассоциативные объекты)
6. Объект — пассивный или спецификация

Выявление и идентификация отказов в ООМ (модели)
- гарантируется ли исполнение события?
- для отказа добавляем событие, контролирующее отказ.

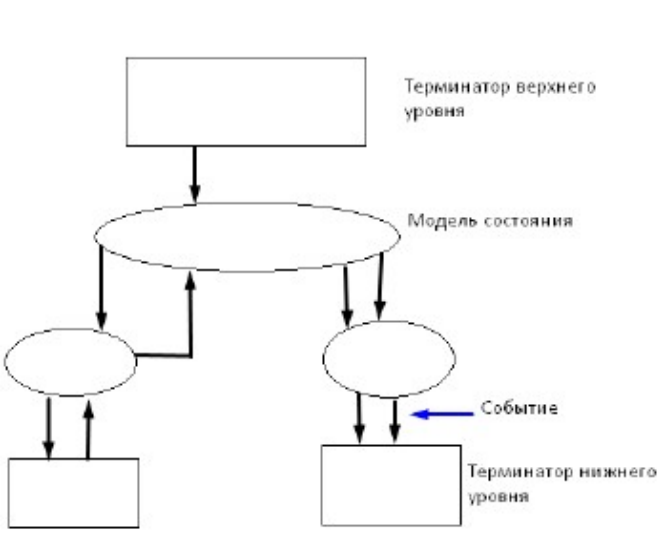
Продукты анализа событий:
1) ДПС

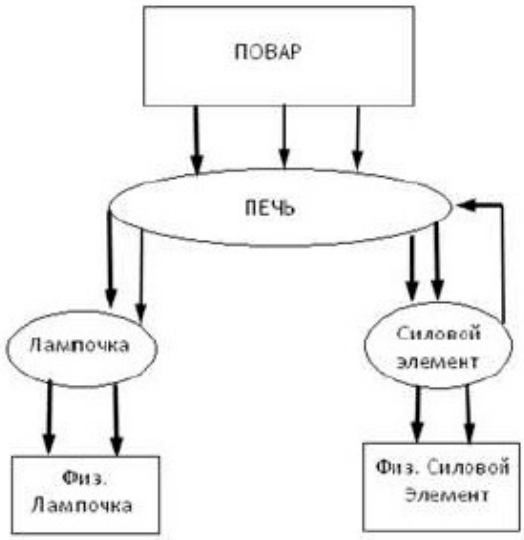
2) ТПС
3) Алгоритмы действий
4) Список действий

Правила переходов — правила, определяющие, как достигается новое состояние, если конкретное событие происходит в определённом состоянии.

7. Модель взаимодействия объектов (МВО). Диаграмма взаимодействия объектов в подсистеме. Типы событий. Схемы управления. Имитирование. Каналы управления.

Модель взаимодействия объектов — графическое представление взаимодействия. Каждая модель состояний — овал, стрелочки — события.



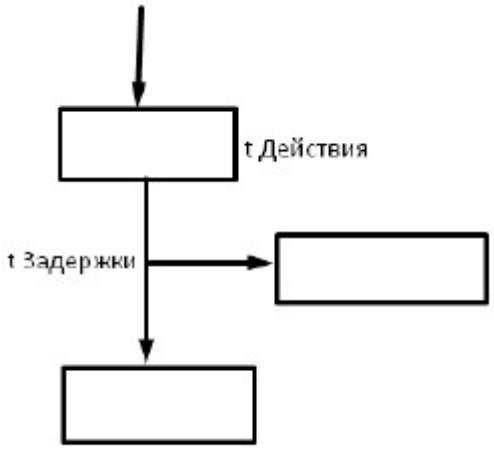


События, которые приходят в систему — приходят извне, эта внешняя сущность — терминатор.

Типы событий.
1. Внешние события (приходят от терминатора)
- Не запрашиваемые события (не являются результатом действия предыдущей действенности подсистемы)
- Запрашиваемые события
2. Внутренние события (порождаются какой-либо моделью состояний подсистемы)
Процесс имитирования: задаётся некоторое начальное состояние. Генерируем некоторое внешнее события и смотрим за изменением состояния всех объектов в системе.

Время имитирования:
1. Время выполнения действия.
2. Время задержи — время, в течении которого объект должен находиться в определённом состоянии (резкий переход из состояния в состояние невозможен).

Этапы имитирования:
1. Установка начального состояния
2. Приём незапрашиваемого события и выполнения канала управления
3. Оценка конечного результата



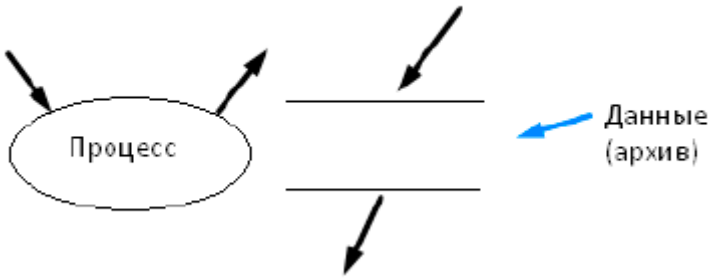
Канал управления — последовательность событий и действий, происходящих в ответ на поступление некоторого незапрашиваемого события. Если возникло событие к терминатору, влекущее за собой новые события от терминатора, то они тоже включаются в канал управления.

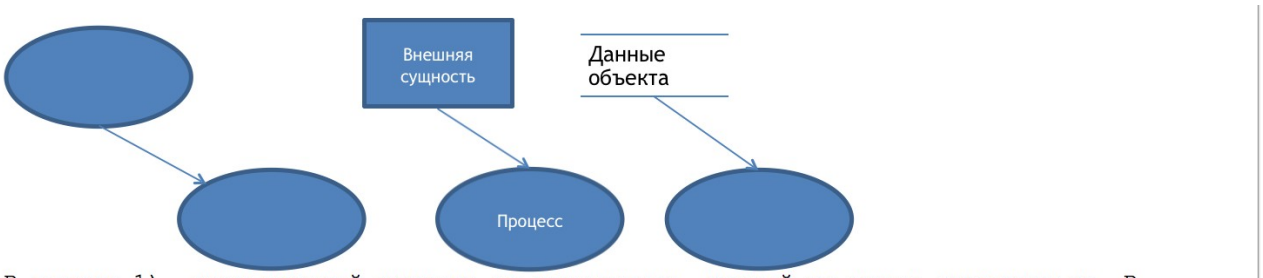
8. Диаграмма потоков данных действий (ДПДД). Типы процессов: акцессоры, генераторы событий, преобразования, проверки. Таблица процессов (ТП). Модель доступа к объектам (МДО).

Диаграмма потоков данных действий (ДПДД) — графическое представление модулей процесса в пределах действия и взаимодействие между ними. Для каждого действия каждого состояния каждой модели состояния.

Строится для каждого состояния каждого объекта класса.
Процесс построения:
Разбиваем действия на процессы, которые могут происходить:

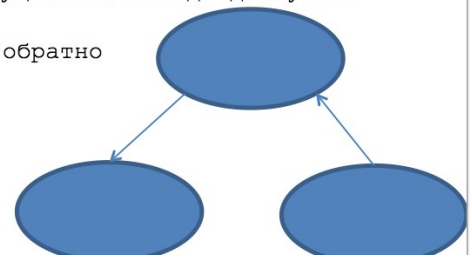
- процесс проверки — «проверить, что...»
- процесс преобразования — что делает процесс
- аксессоры (процесс для получения данных из одного архива данных)
1) Создание
2) Чтение
3) Запись
4) Уничтожение
- генераторы событий (создаёт одно событие как вывод)



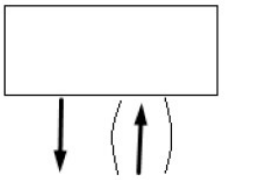


В случае 1) ,если верхний процесс не выполнился, второй не может выполниться. В 2), процесс может выполниться, поскольку данные внешних сущностей всегда доступно. То же самое касается атрибутов самого себя в 3)

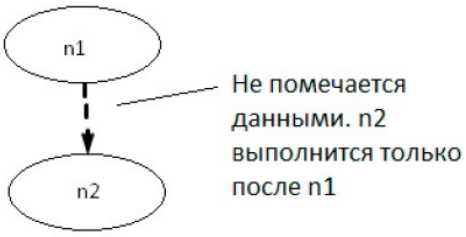
Результатом процесса могут быть данные, возвращающиеся обратно



Процесс не может выполняться, пока все вводы не доступны.
Выводы процесса доступны тогда, когда процесс завершит выполнение.
Данные объектов, событий, архива и терминаторов всегда доступны.



Чтение (запись) данных
из внешних сущностей



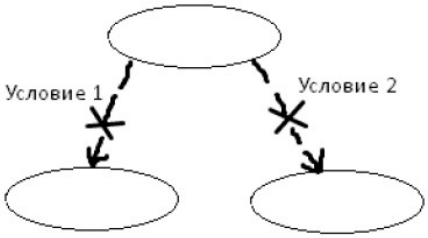
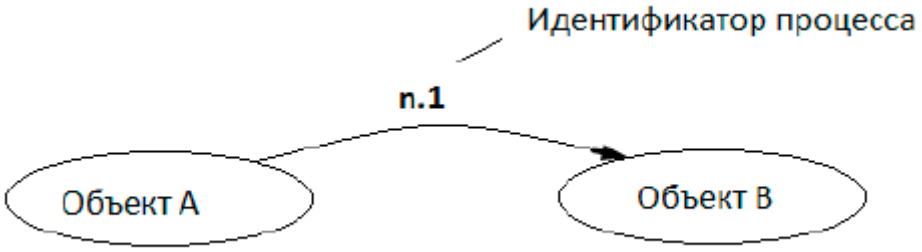


Таблица процессов действий состояний:
Все процессы в подсистеме объединяются в единую таблицу. В разных действиях могут происходить одни и те же процессы — они будут общими. Общие процессы могут выполнять одну и ту же функцию, читать и записывать и создавать и уничтожать одни и те же объекты, и т.д.

Id процесса	Тип	Название	Где используются	
			Модель состояний	Действия

Модель доступа к объектам:
Данная модель строится на основе выделенных аксессорных процессов. На модели доступа модели состояний (объектов) рисуются вытянутыми овалами. Если А использует аксессор модели состояний В, то рисуется стрелка, А будет аксессором.



9. Домены. Модели доменного уровня. Типы доменов. Мосты, клиенты, сервера.

Домен — отдельный, реальный, гипотетический или абстрактный мир, населённый отчётливым количеством объектов, которые ведут себя в соответствии с предусмотренным доменом правилами. Каждый домен образует отдельное и связное единое целое.

Типы доменов:
- Прикладные — предметная область системы с точки зрения пользователя
- Сервисные — функционал для поддержки прикладного домена
- Архитектурные — обеспечивают единые механизмы управления данными и всех программой как едиными целым
- Реализационные — библиотечный класс — взаимодействие по сети, протоколы взаимодействия по сети

Сервис и домен — разные понятия!

Мост — связь между доменами, когда один домен использует механизмы и возможности другого.

Клиент — использующий возможности домен.
Сервер — домен, предоставляющий свои возможности для использования.

Клиент рассматривает мост как набор предложений, которые будут представлены другим доменом. Сервер подходит к мосту как к набору требований для выполнения.

Схема доменов или модель доменного уровня:
Графическое представление связей между доменами. Содержит в себе домены и мосты между ними. В прямоугольниках — домены. Мосты изображаются стрелками. Домен к задаче находится внизу. Сервисные домены находятся вверху.

Схемы для подсистем:
- Модель связи подсистем — по информационной модели
- Модель взаимодействия подсистем (по МВО)
- Модель доступа подсистем (по МДО)(стрелки — аксессоры) — модель доступа к объектам.

Стрелочкой помечается идентификатор процесса. Модель взаимодействия асинхронная, событийная модель. Модель доступа — синхронное взаимодействие (один объект может получить данные другого объекта). В итоге получаем модель доступа к объектам, таблицу процессов состояний и диаграмму потоков данных действий.

10. Объектно-ориентированное проектирование. Диаграмма класса. Структура класса. Диаграмма зависимостей. Диаграмма наследования.

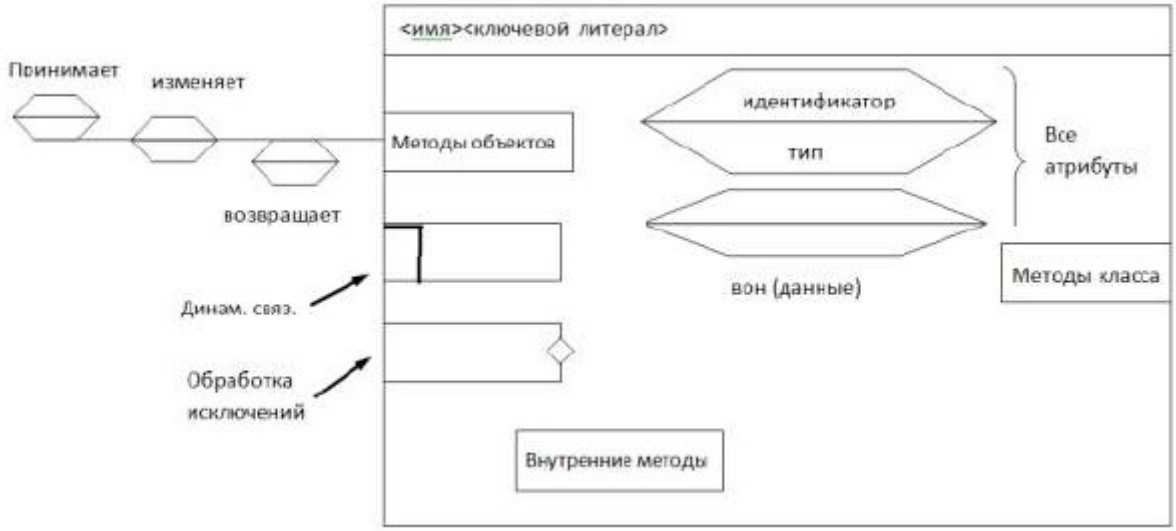
Объектно-ориентированное проектирование — часть объектно-ориентированной методологии, которая представляет программистам возможность оперировать понятием «объект», помимо понятия «процедура», при разработке кода.

Объекты инкапсулируют данные и процедуры, сгруппированные вместе, отражая сущность объекта.

Нотация OODL (object oriented design language):
- Диаграмма класса (Буга-Бахра)
- Схема структуры класса (внутренняя структура кода операций класса)

- Диаграмма зависимостей (взаимодействие клиент-сервер, механизм использования/выполнения и дружественных связей)
- Диаграмма наследования (показать наследование)

Диаграмма класса — описывает внешнее представление данного класса:



Описывает данные объекта и класса, внешнее представление данного класса.

Схема структуры класса:

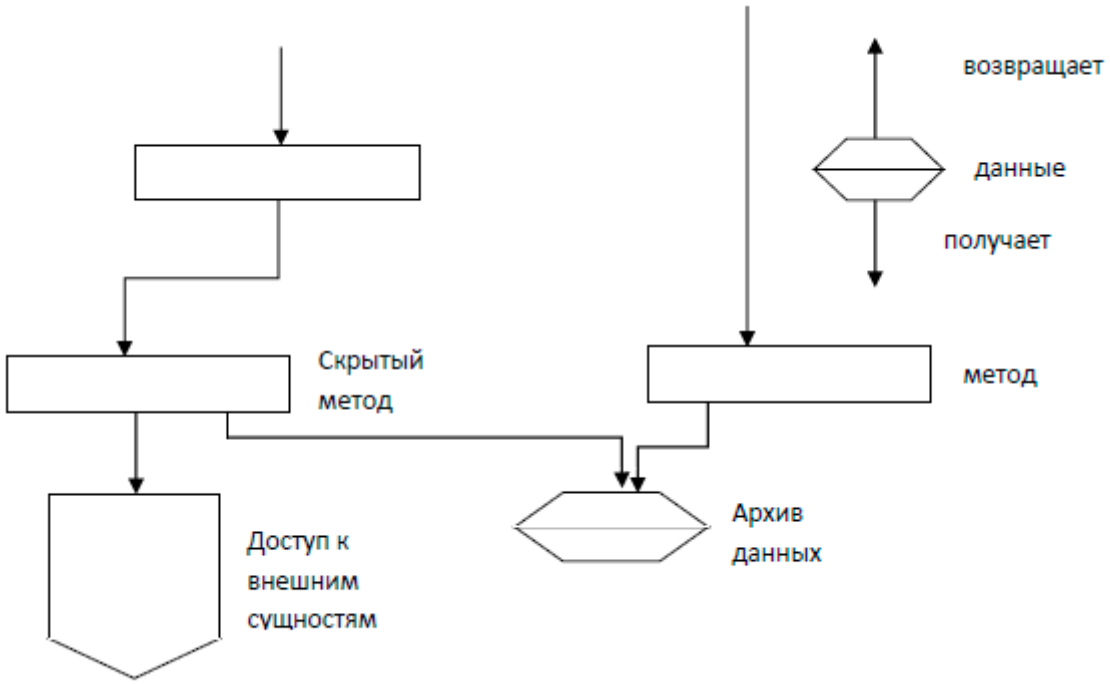


Схема структуры класса используется для того, чтобы конкретизировать потоки данных.

Диаграмма зависимости — строится диаграмма класса, оставляется только заголовок, а атрибуты перечисляются списком.

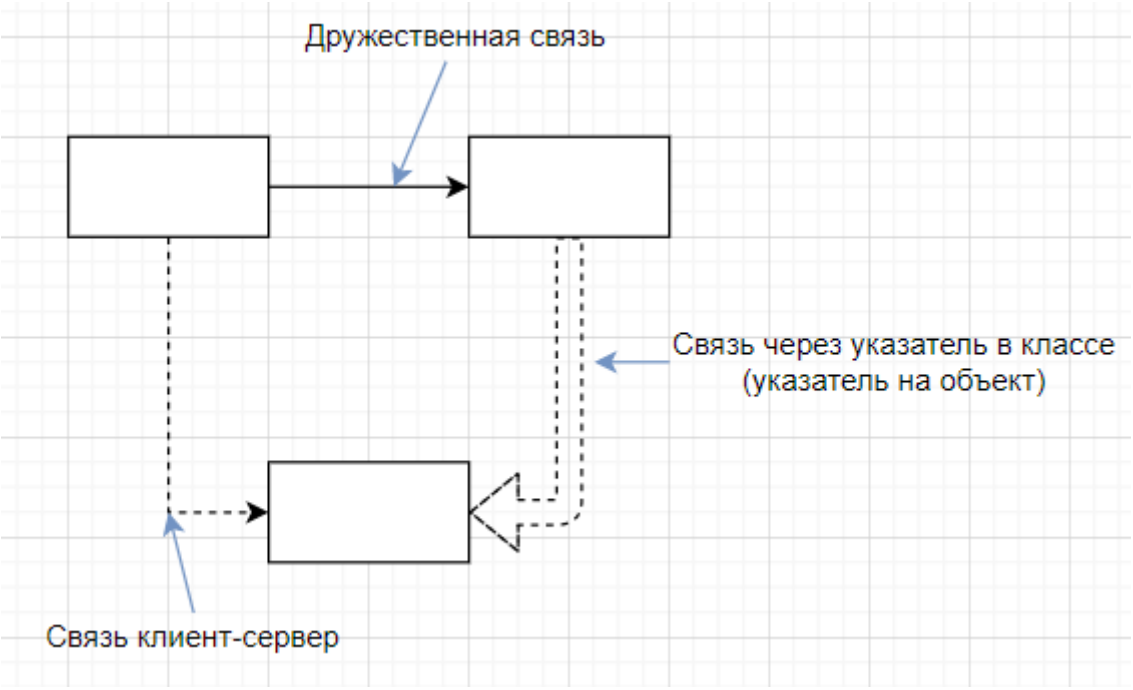
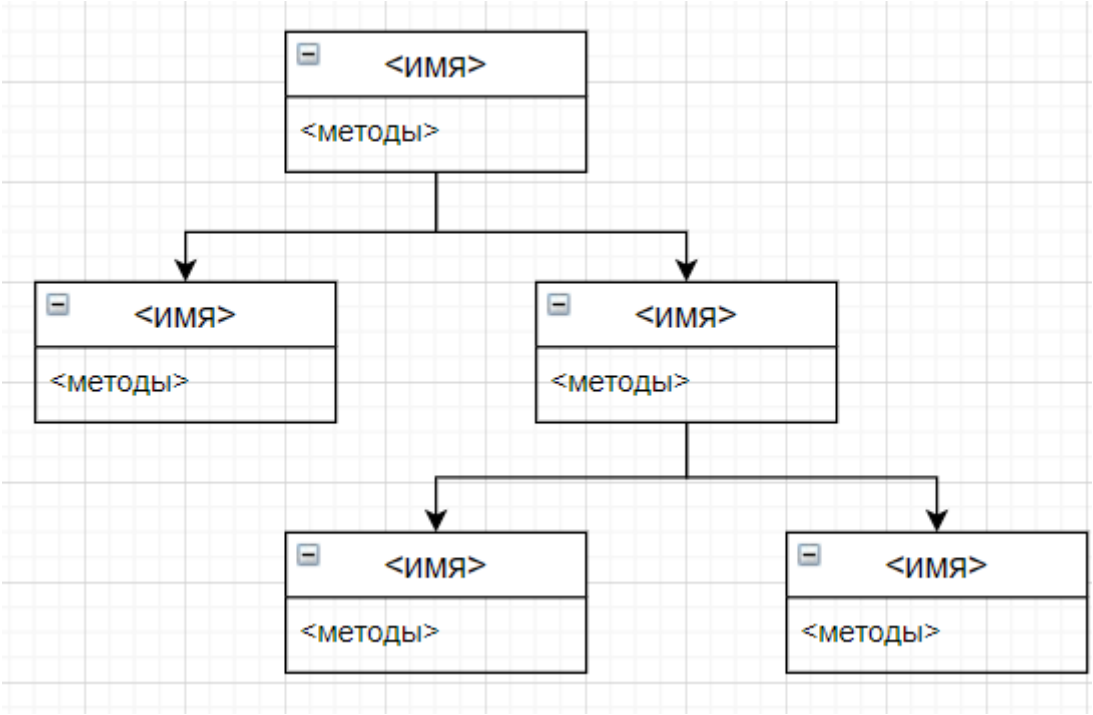


Диаграмма наследования — от базового класса наследуются подклассы. Внутри атрибуты и методы.



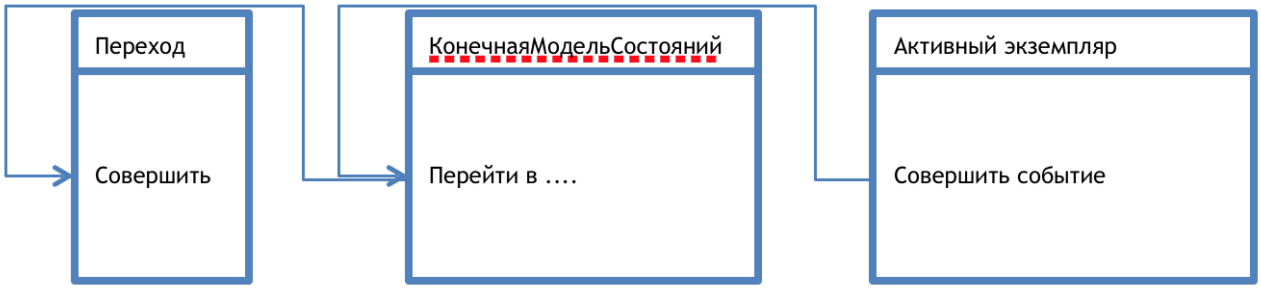
В диаграмме наследования у каждого класса указывается имя, атрибуты и методы.

11. Архитектурный домен. Паттерн КМС. Шаблоны для создания прикладных классов.

Архитектурный домен обеспечивает единые механизмы управления данными и всей программой, как единым целым. Архитектурный домен можно реализовать как паттерн КМС.

Создаются несколько классов для архитектурного домена, задача которых — задать правила перехода из состояния в состояние при возникновении событий. Эти классы представляют активный экземпляр, все остальные будут производными от активного.

Классы, выделяемые для архитектурного домена:

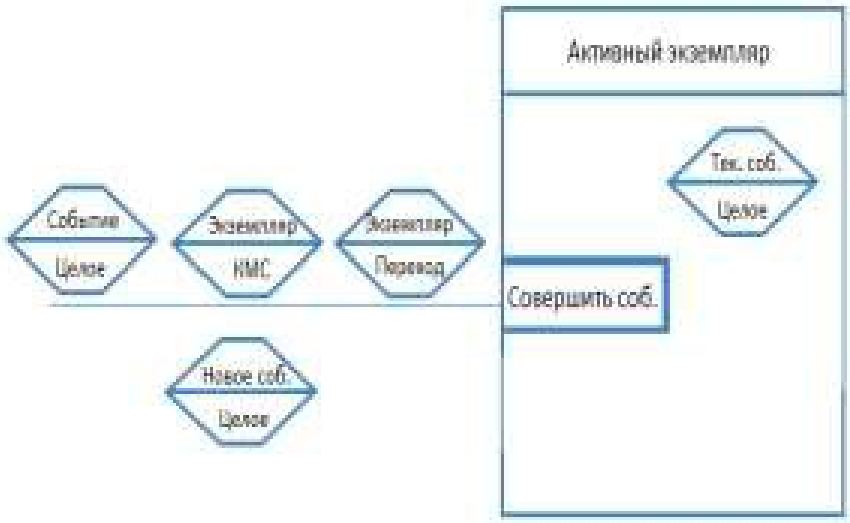


Паттерн КМС берёт на себя функцию контроля. То есть уже нет необходимости проверять возможность перехода в данное состояние.

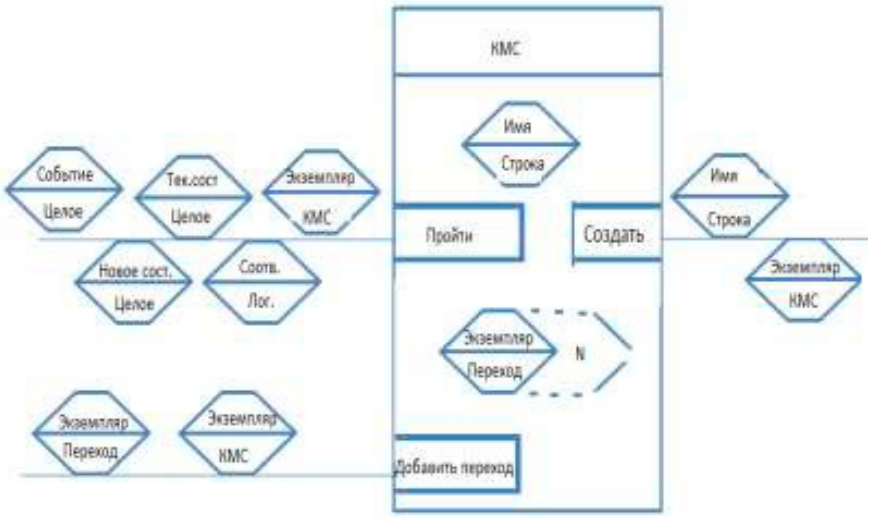
КМС делается шаблонным классом, это позволяет сделать систему гибкой.
Все активные классы будут производными от активного экземпляра.

Выделяют два типа объектов:
1. Пассивные — конструкторы, аксессоры (объектов, класса)
2. Активные — обработчики событий, конструкторы, аксессоры, инициализатор КМС. (Для активных выделяют жизненные циклы).

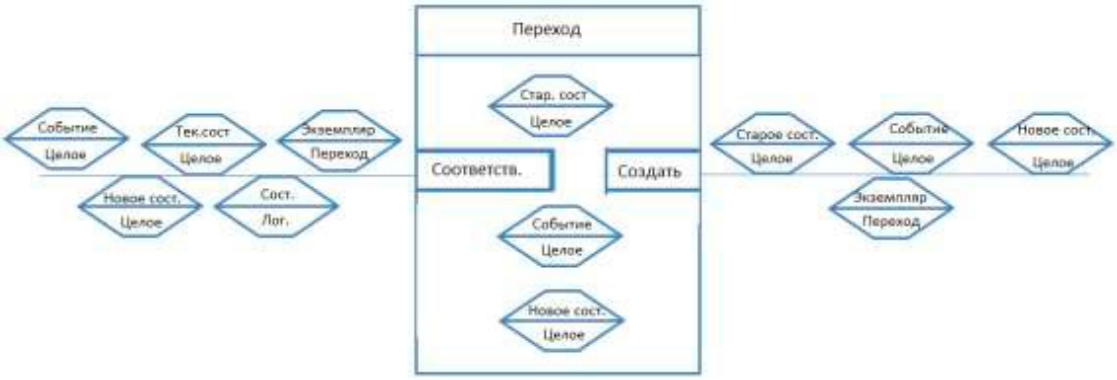
Выделяются также объекты, осуществляющие только связь между другими — определители: конструкторы, обработчики, инициализатор КМС. (Определяют модель состояний связей объектов)
Активный экземпляр:



КМС:



Переход:



12. Структурные паттерны: адаптер (Adapter), компоновщик (Composite), декоратор (Decorator), заместитель (Proxy), мост (Bridge), фасад (Facade).

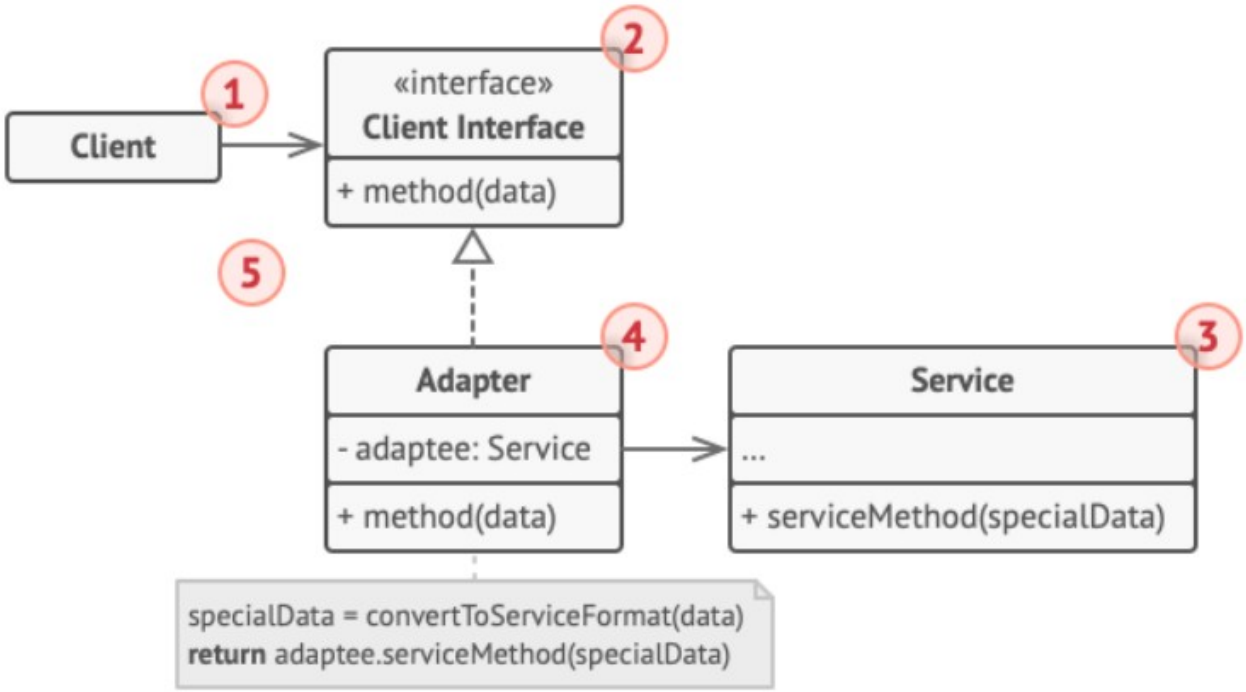
Структурные паттерны — позволяют строить удобные в поддержке иерархии классов.

Адаптер (Adapter) — структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.

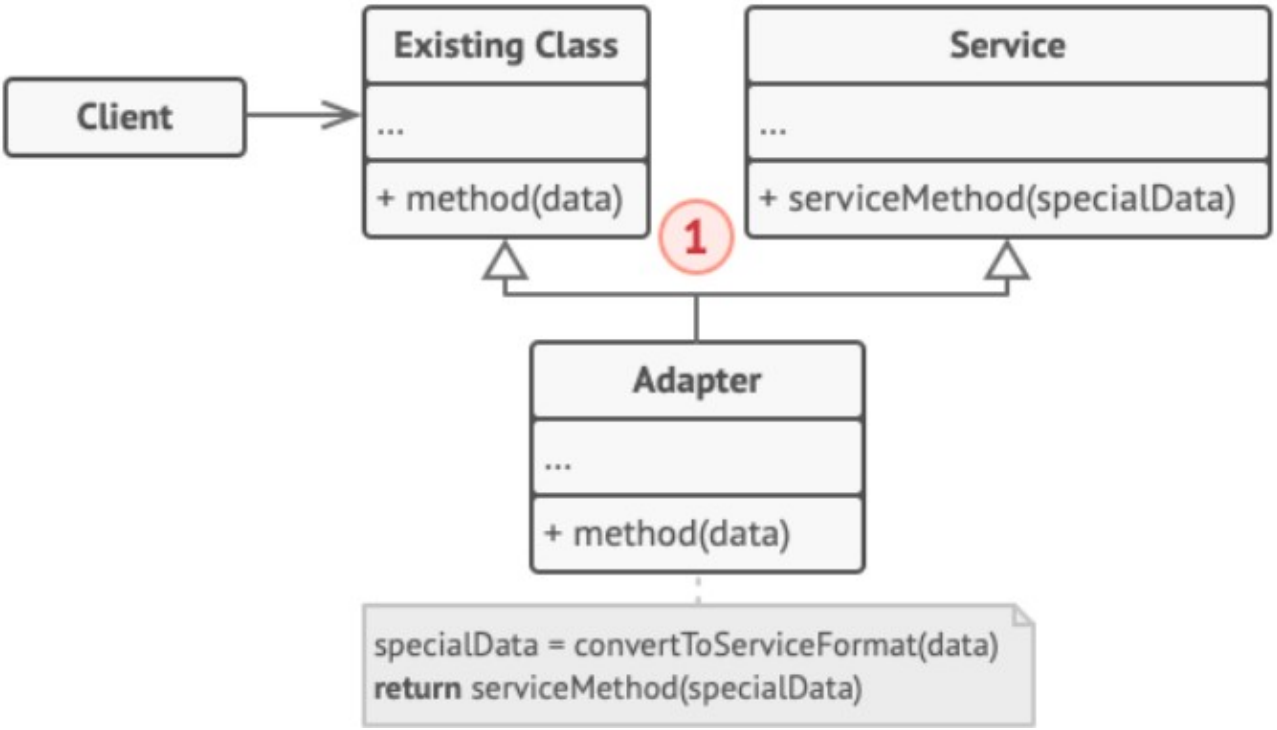
Принцип работы адаптера:
1. Адаптер имеет интерфейс, который совмести с одним из объектов.
2. Данный объект может свободно вызывать методы адаптера.
3. Адаптер получает эти вызовы, и перенаправляет их второму объекту, но в формате и последовательности, соответствующие интерфейсу второго объекта.

Возможен вариант создания двухстороннего адаптера, который может работать в обе стороны.

Структура паттерна адаптера:
Использует агрегацию — объект адаптера оборачивает (содержит ссылку) на служебный объект.
1. Клиент — класс, который содержит существующую бизнес-логику программы.
2. Клиентский интерфейс — описывает протокол, через который клиент может работать с другими классами.
3. Сервис — полезный сторонний класс, который не может работать с клиентским интерфейсом напрямую, для чего и создаётся адаптер.
4. Адаптер — класс, который может одновременно работать и с клиентом и с сервисом. Он реализует клиентский интерфейс и содержит ссылку на объект сервиса. Адаптер получает вызовы от клиента через методы клиентского интерфейса, а затем переводит из в вызовы методов обёрнутого объекта в правильном формате.
5. Работая с адаптером через интерфейс, клиент может не привязываться к конкретному классу адаптера. Благодаря этому возможно добавление в программу новых видов адаптеров, независимо от клиентского кода.



Адаптер классов — данная реализация базируется на наследовании. Адаптер наследует оба интерфейса одновременно, такой подход возможен только в языках, поддерживающих множественное наследование.



Применение:
- создание объекта-прокладки, который будет превращать вызовы приложения в формат, понятный стороннему классу.

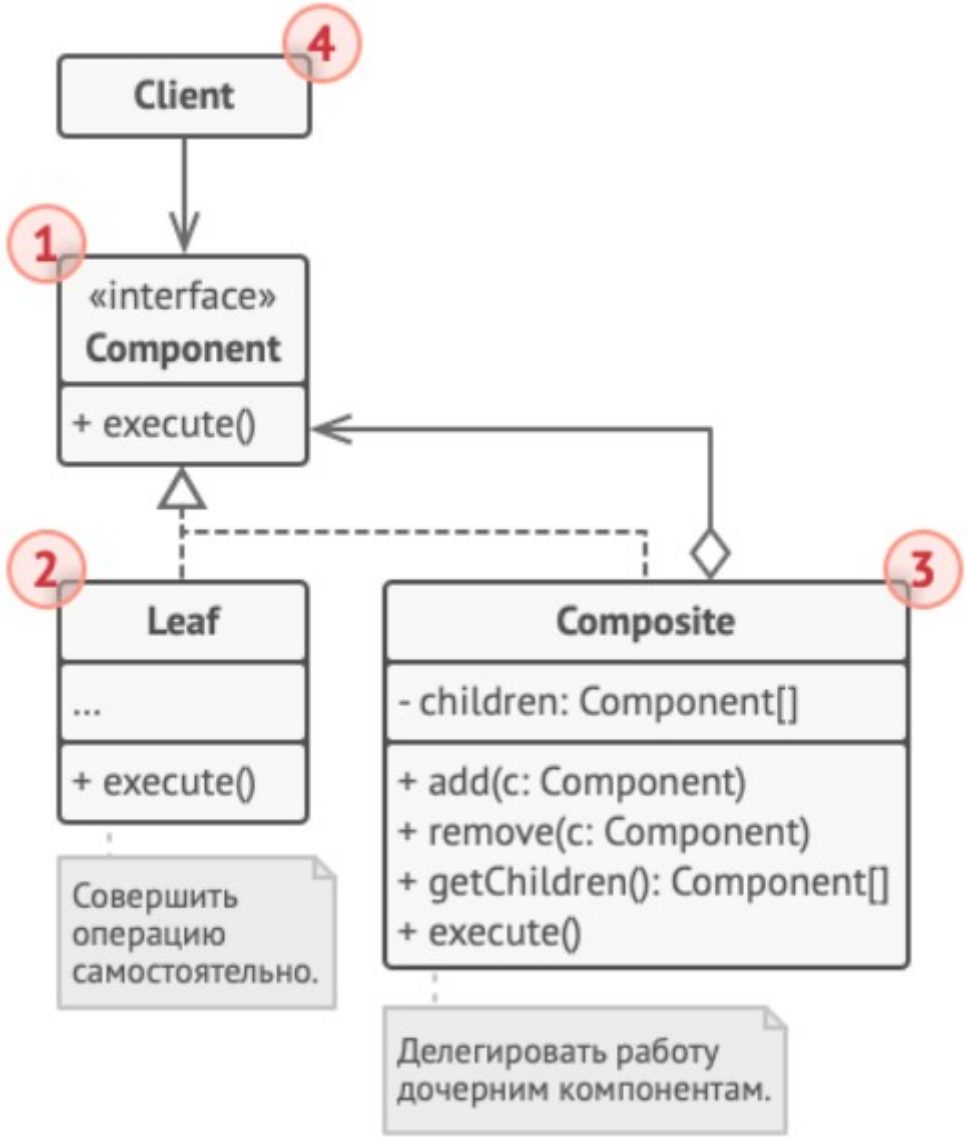
- если нужно добавить дополнительную функциональность в иерархию классов без изменения суперкласса. Создаётся адаптер для работы с суперклассом, который также сможет работать со всеми подклассами иерархии.

Преимущества и недостатки:
+ Отделяет и скрывает от клиента подробности преобразования различных интерфейсов.
- Усложняет код программы из-за введения дополнительных классов.

Компоновщик (Composite) — структурный паттерн проектирования, который позволяет сгруппировать множество объектов в древовидную структуру, а затем работать с ней так, как будто это единичный объект.

Паттерн имеет смысл только тогда, когда основная модель вашей программы может быть сгруппирована в виде дерева.

Структура паттерна Компоновщик:
1. Компонент — определяет общий интерфейс для простых и составных компонентов дерева.
2. Лист — простой компонент дерева, не имеющий ответвлений.
3. Контейнер (композит) — составной компонент дерева. Содержит набор дочерних компонентов, но ничего не знает об их типах. Это могут быть как простые компоненты-листья, так и другие контейнеры. Методы контейнера переадресуют работу своим дочерним компонентам, хотя и могут добавлять что-то своё к результату.
4. Клиент — работает с деревом через общий интерфейс компонентов.

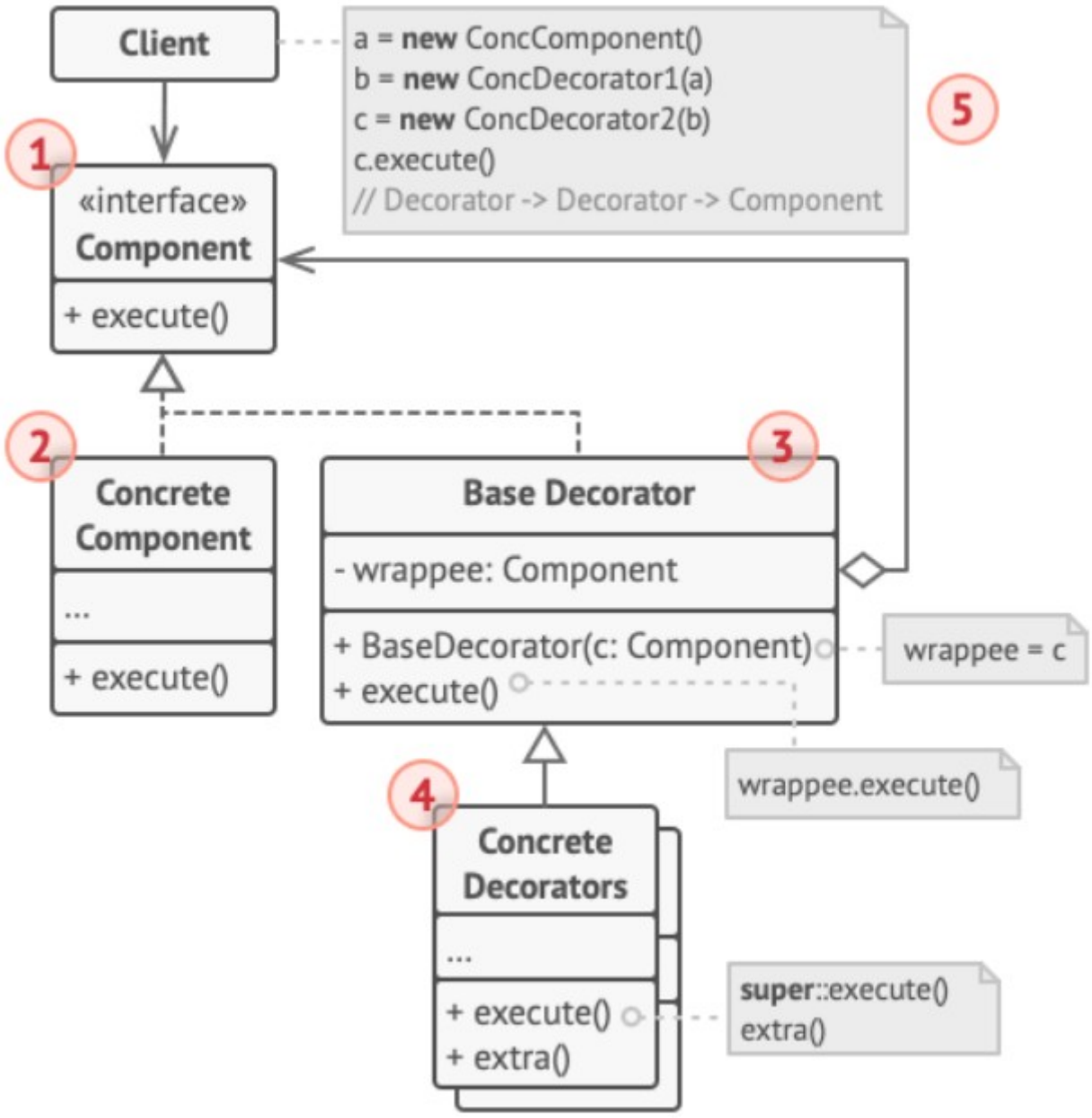


Применение:
- если необходимо представить древовидную структуру объектов — за счёт составных ссылок.
- если клиенты единообразно должны трактовать простые и составные объекты — так как у простых и составных объектов общий интерфейс, клиенту безразлично, с каким именно объектом ему нужно работать.

Преимущества и недостатки:
+ упрощает архитектуру клиента про работе со сложным деревом компонентов.
+ облегчает добавление новых типов компонентов.
- создаёт слишком общий дизайн классов.

Декоратор — структурный паттерн проектирования, который позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».

Структура паттерна декоратор:



1. Компонент — задаёт общий интерфейс обёрток и оборачиваемых объектов.
2. Конкретный компонент — определяет класс оборачиваемых объектов, содержит какое-то базовое поведение, которое потом изменяют декораторы.
3. Базовый декоратор — хранит ссылку на вложенный объект-компонент. Им может быть как конкретный компонент, так и один из конкретных декораторов.

Базовый декоратор делегирует все свои операции вложенному объекту. Дополнительное поведение будет жить в конкретных декораторах.
4. Конкретные декораторы — это различные вариации декораторов, которые содержат добавочное поведение. Оно выполняется до или после вызова аналогичного поведения обёрнутого объекта.
5. Клиент — может оборачивать простые компоненты и декораторы в другие декораторы, работая со всеми объектами через общий интерфейс компонентов.

Применение:
- для добавления обязанностей объектам во время выполнения программы, незаметно для кода, который их использует.
- если нельзя расширить обязанности объекта с помощью наследования (если заблокировано наследование класса).

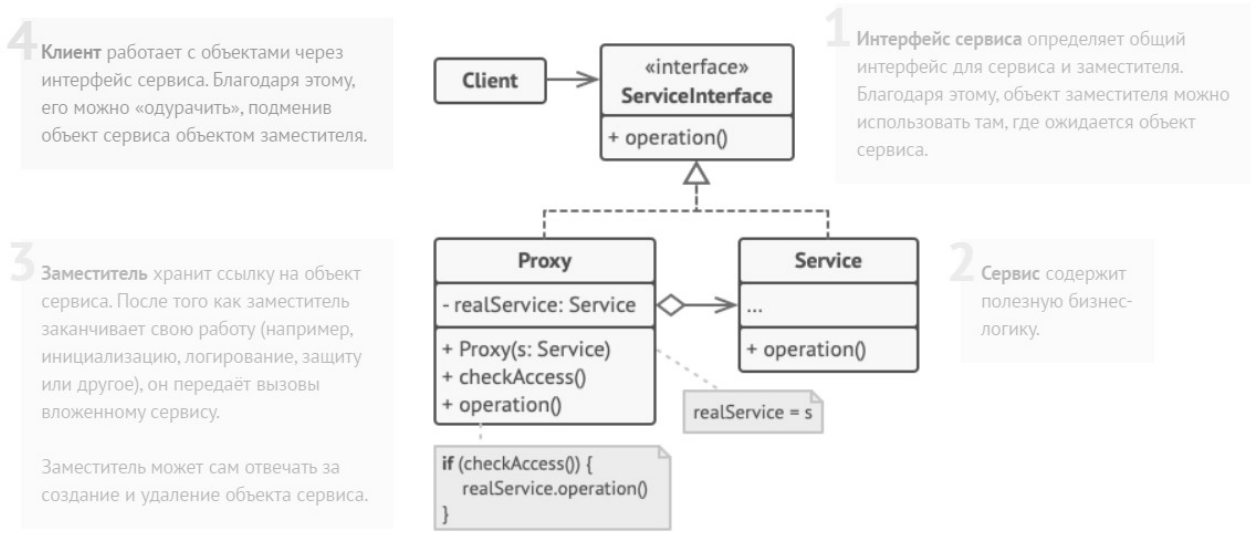
Преимущества и недостатки:
+ большая гибкость, чем у наследования.
+ позволяет добавлять обязанности на лету.
+ можно добавлять несколько новых обязанностей сразу.
+ позволяет иметь несколько мелких объектов вместо одного объекта на все случаи жизни.
- трудно конфигурировать многократно обёрнутые объекты.
- много крошечных классов.

Заместитель (Proxy) – структурный паттерн проектирования, который позволяет подставлять вместо реальных объектов специальные объекты-заменители. Эти объекты перехватывают вызовы к оригинальному объекту, позволяя сделать что-то до или после вызова оригинала.

Структура паттерна:
1. Интерфейс сервиса — определяет общий интерфейс для сервиса и заместителя. Благодаря этому объект заместителя можно использовать там, где ожидается объект сервиса.
2. Сервис — содержит полезную бизнес-логику.
3. Заместитель — хранит ссылку на объект сервиса. После того, как заместитель заканчивает свою работу, он передаёт вызовы вложенному сервису.

Заместитель может сам отвечать за создание и удаление объекта сервиса.

4. Клиент — работает с объектами через интерфейс сервиса. Благодаря этому, объект сервиса может быть заменён объектом заместителя.

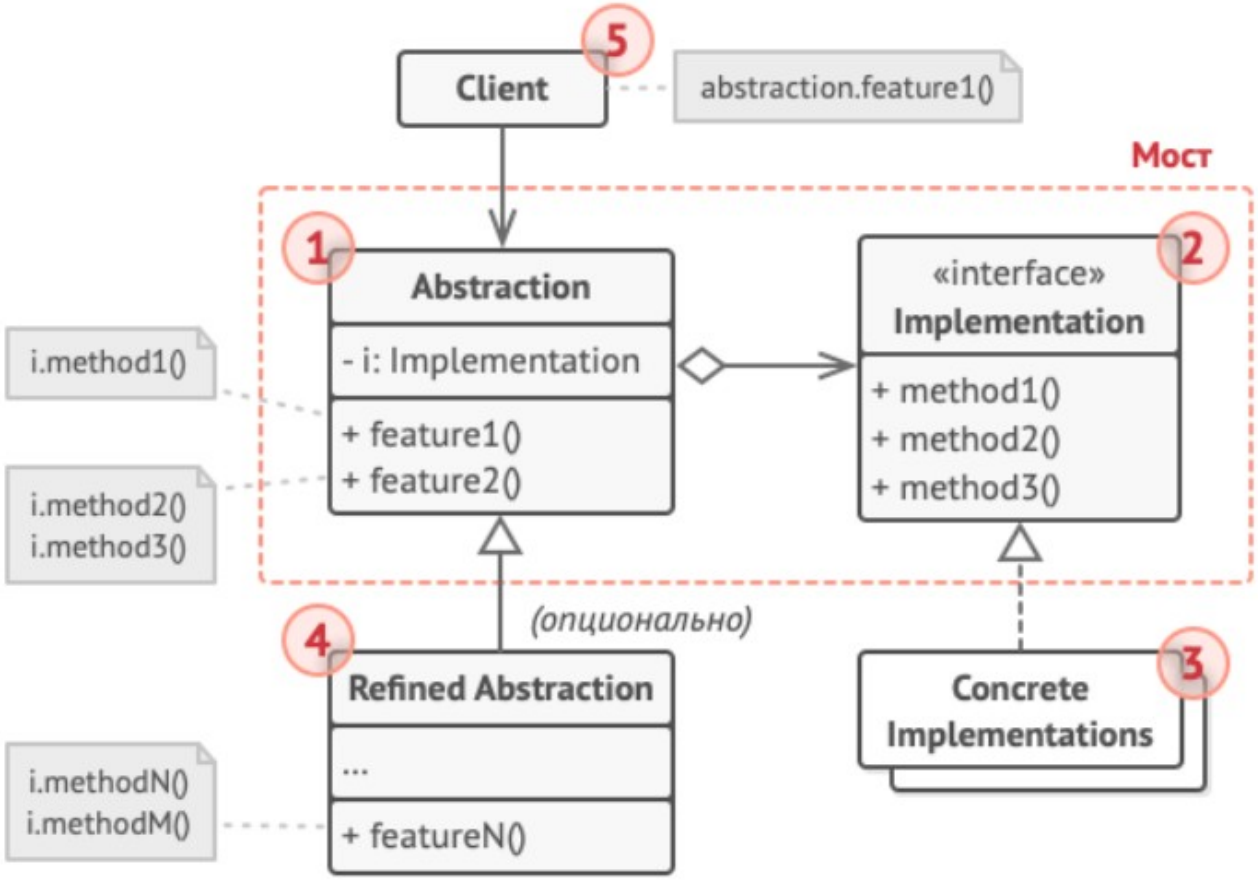


Применение:
- ленивая инициализация (виртуальный прокси), если у нас есть тяжёлый объект, грузящий данные из файловой системы или базы данных. Объект создаётся тогда, когда он действительно нужен.
- защита доступа (защищающий прокси), если в программе есть разные типы пользователей, и нам нужно защитить объект от неавторизированного доступа. В этом случае прокси проверяет доступ при каждом вызове перед передачей управления служебному объекту, если доступ разрешён.
- локальный запуск сервиса (удалённый прокси), в случае если настоящий сервисный объект находится на удалённом сервере, заместитель транслирует запросы клиента в вызовы по сети в протоколе, понятном удалённому сервису.
- логирование запросов (логирующий прокси), в случае, если требуется хранить историю обращений к сервисному объекту.
- кеширование объектов (умная ссылка), в случае если нужно кешировать результаты запросов клиентов и управлять их жизненным циклом.

Преимущества и недостатки:
+ позволяет контролировать сервисный объект незаметно от клиента.
+ может работать, даже если сервисный объект ещё не создан.
+ может контролировать жизненный цикл служебного объекта.
- усложняет код программы из-за введения дополнительных классов.
- увеличивает время отклика от сервиса.

Мост (Bridge) – структурный паттерн проектирования, который разделяет один или несколько классов на две отдельные иерархии — абстракцию и реализацию, позволяя изменять их независимо друг от друга.

Структура паттерна мост:
1. Абстракция — содержит управляющую логику. Код абстракции делегирует реальную работу связанному объекту реализации.
2. Реализация — задаёт общий интерфейс для всех реализаций. Все методы, которые здесь описаны, будут доступны из класса абстракции и его подклассов.
3. Конкретные реализации — содержат платформо-зависимый код.
4. Расширенные абстракции — содержат различные вариации управляющей логики. Как и родитель, работает с реализациями только через общий интерфейс реализации.
5. Клиент — работает только с объектами абстракции. Не считая начального связывания абстракции с одной из реализаций, клиентский код не имеет прямого доступа к объектам реализации.



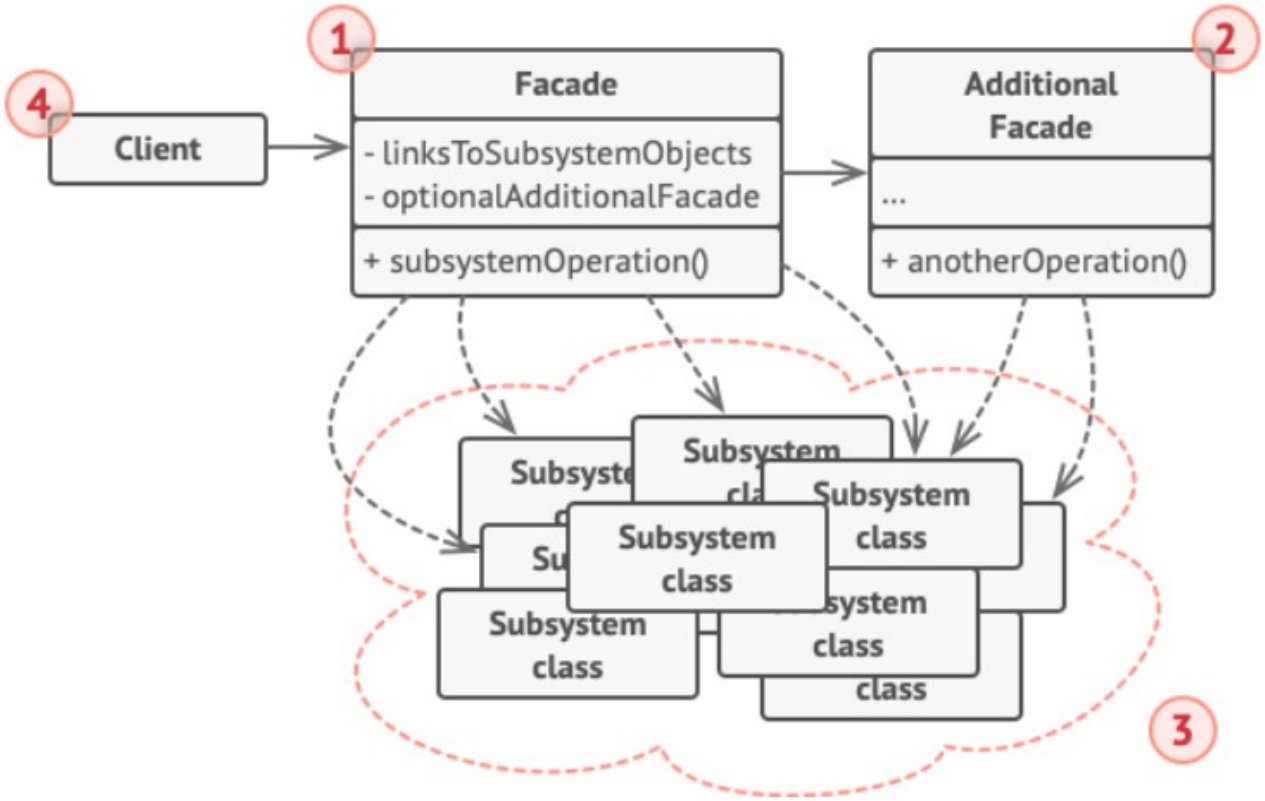
Применение:
- когда нужно разделить монолитный класс, который содержит несколько реализаций какой-то функциональности.

- когда класс нужно расширять в двух независимых плоскостях.
- когда нужно, чтобы реализацию можно было изменять во время выполнения программы.

Преимущества и недостатки:
+ позволяет строить платформо-независимые программы.
+ скрывает лишние или опасные детали реализации от клиентского кода.
+ реализует принцип открытости/закрытости (программные сущности открыты для расширения, закрыты для изменения).
- усложняет код программы из-за введения дополнительных классов.

Фасад (Facade) – структурный паттерн проектирования, который предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку.

Структура класса:



1. Фасад — предоставляет быстрый доступ к определённой функциональности подсистемы.
2. Дополнительный фасад — может быть введён, чтобы не захламлять единственный фасад разнородной функциональностью. Может быть использован как клиентом, так и другими фасадами.

3. Сложная подсистема — состоит из множества разнообразных классов. Классы подсистемы не знают о существовании фасада и работают друг с другом напрямую.
4. Клиент — использует фасад вместо прямой работы с объектами сложной подсистемы.

Применение:
- если нужно представить простой или урезанный интерфейс к сложной подсистеме.
- если нужно разложить подсистему на отдельные слои.

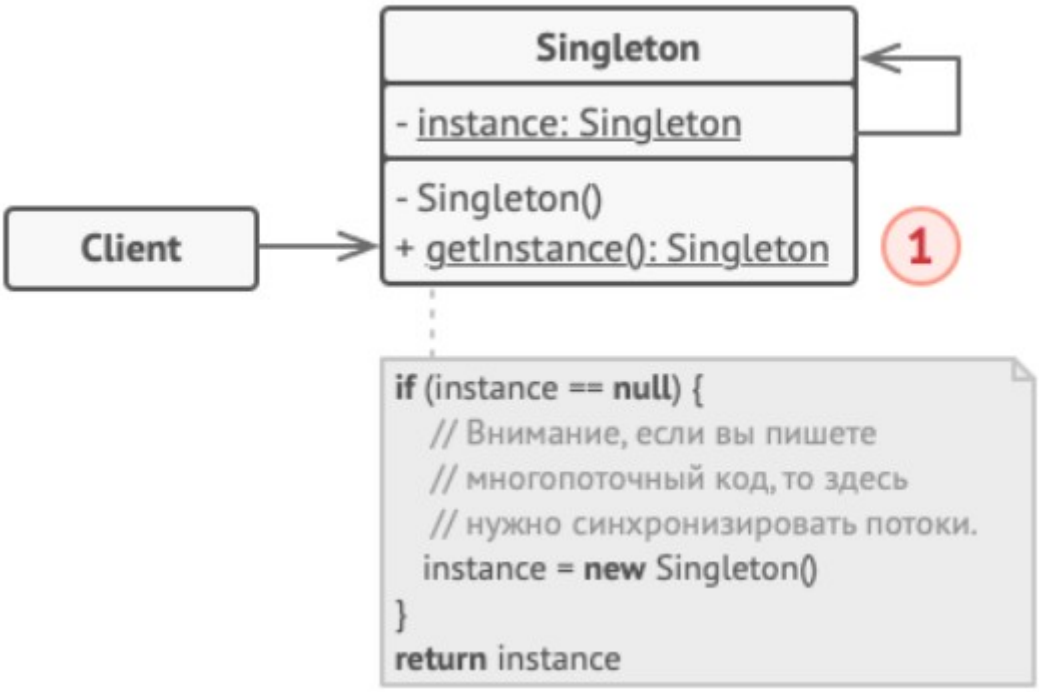
Преимущества и недостатки:
+ изолирует клиентов от компонентов сложной подсистемы.
- фасад рискует стать божественным объектом, привязанным ко всем классам программы.

13. Порождающие паттерны: одиночка (Singletone), фабричный метод (Factory Method), абстрактная фабрика (Abstract Factory), строитель (Builder), прототип (Prototype), пул объектов (Object Pool).

Одиночка (Singletone) – порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

Одиночка решает две проблемы, нарушая принцип единственной ответственности класса:
1. Гарантируется наличие единственного экземпляра класса.
2. Предоставляет глобальную точку доступа.

Структура паттерна одиночки:



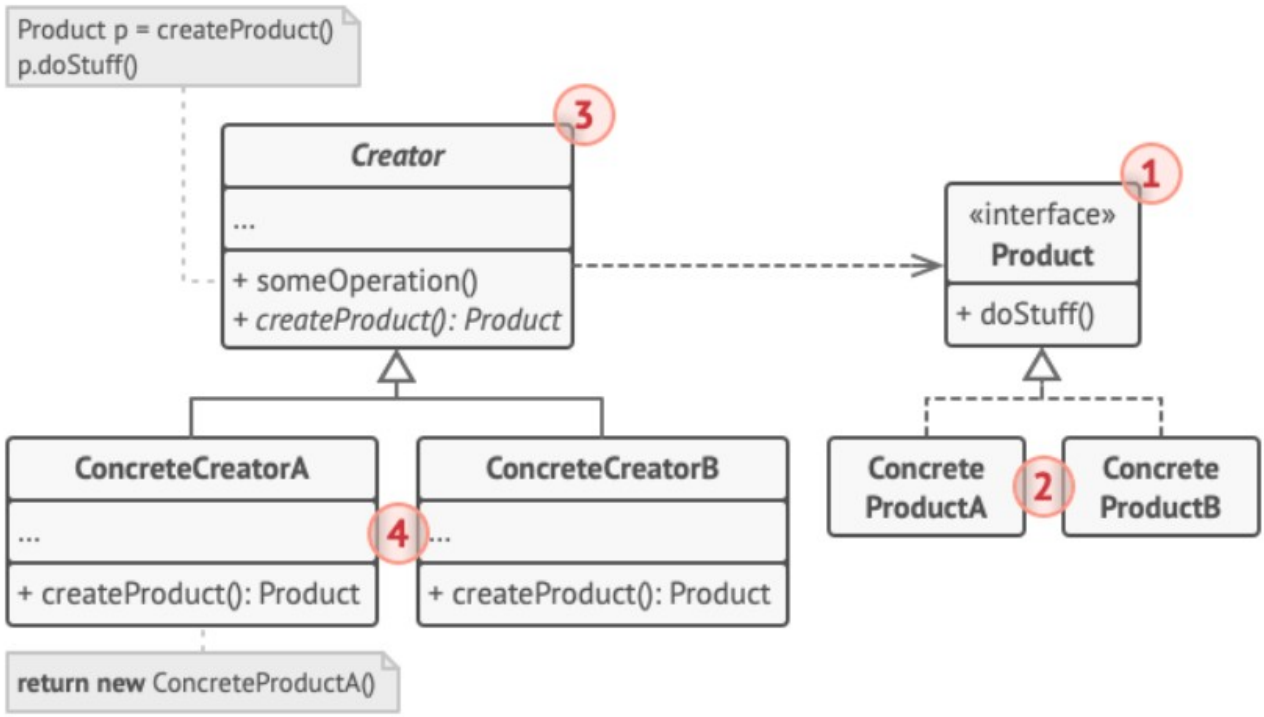
1. Одиночка — определяет статический метод getInstance, который возвращает единственный экземпляр своего класса. Конструктор одиночки должен быть скрыт от клиентов. Вызов метода getInstance должен стать единственным способом получить объект этого класса.

Применимость:
- если в программе должен быть единственный экземпляр какого-то класса, доступный всем клиентам.
- если хочется иметь больше контроля над глобальными переменными.

Преимущества и недостатки:
+ гарантирует наличие единственного экземпляра класса.
+ предоставляет к классу глобальную точку доступа.
+ реализует отложенную инициализацию объекта-одиночки.
- нарушает принцип единственной ответственности класса.
- маскирует плохой дизайн.
- проблемы мультипоточности.

Фабричный метод (Factory Method) – порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых классов.

Структура паттерна Фабричный метод:



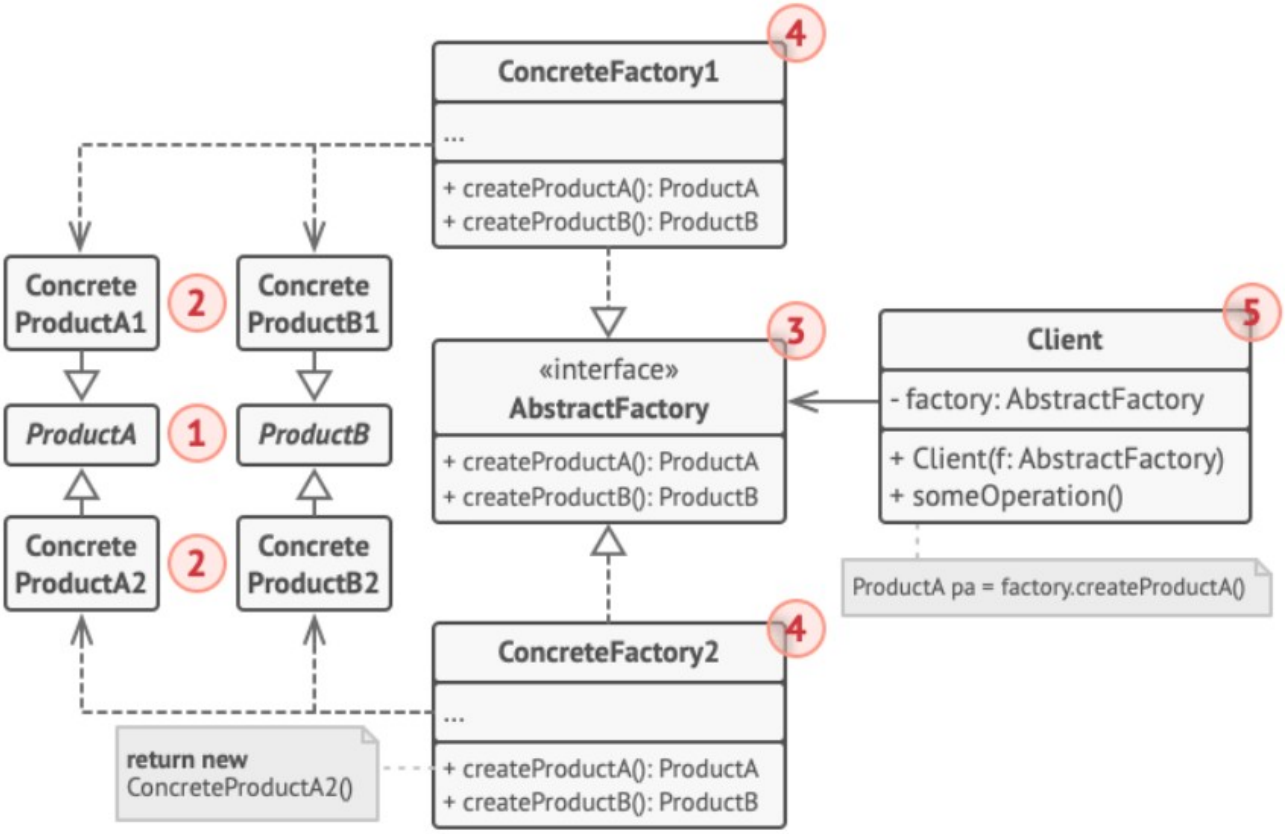
1. Продукт — определяет общий интерфейс объектов, которые может произвести создатель и его подклассы.
2. Конкретные продукты — содержат код различный продуктов. Продукты будут отличаться реализацией, но интерфейс у них будет общий.
3. Создатель — объявляет фабричный метод, который должен возвращать новые объекты продуктов. Важно, чтобы тип результата совпадал с общим интерфейсом продуктов.
4. Конкретные создатели — по-своему реализуют фабричный метод, производя те или иные конкретные продукты.

Применимость:
- заранее известны типы и зависимости объектов, с которыми должен работать код.
- если нужно дать возможность пользователям расширять части фреймворка или библиотеки.
- если нужно сэкономить системные ресурсы, повторно используя уже созданные объекты вместо порождения новых.

Преимущества и недостатки:
+ избавляет класс от привязки к конкретным классам продуктов.
+ выделяет код производства продуктов в одно место, упрощая поддержку кода.
+ упрощает добавление новых продуктов в программу.
+ реализует принцип открытости/закрытости.
- может привести к созданию больших параллельных иерархий классов, так как для каждого класса продукта надо создать свой подкласс создателя.

Абстрактная фабрика (Abstract Factory) — порождающий паттерн проектирования, который позволяет создавать семейства объектов, не привязываясь к конкретным классам создаваемых объектов.

Структура паттерна:
1. Абстрактные продукты — объявляют интерфейсы продуктов, которые связаны друг с другом по смыслу, но выполняют разные функции.
2. Конкретные продукты — большой набор классов, которые относятся к различным абстрактным продуктам, но имеют одни и те же вариации.
3. Абстрактная фабрика — объявляет методы создания различных абстрактных продуктов.
4. Конкретные фабрики — относятся каждая к своей вариации продуктов и реализуют методы абстрактной фабрики, позволяя создавать все продукты определённой вариации.
5. Не смотря на то, что конкретные фабрики порождают конкретные продукты, сигнатуры их методов должны возвращать соответствующие абстрактные продукты. Это позволит клиентскому коду, использующему фабрику, не привязываться к конкретным классам продуктов. Клиент сможет работать с любыми вариациями продуктов через абстрактные интерфейсы.



Применение:
- в случае, если бизнес-логика программы должна работать с разными видами связанных друг с другом продуктов, не завися от конкретных классов продуктов.
- в случае, если в программе уже используется фабричный метод, но очередные изменения предполагают введение новых типов продуктов.

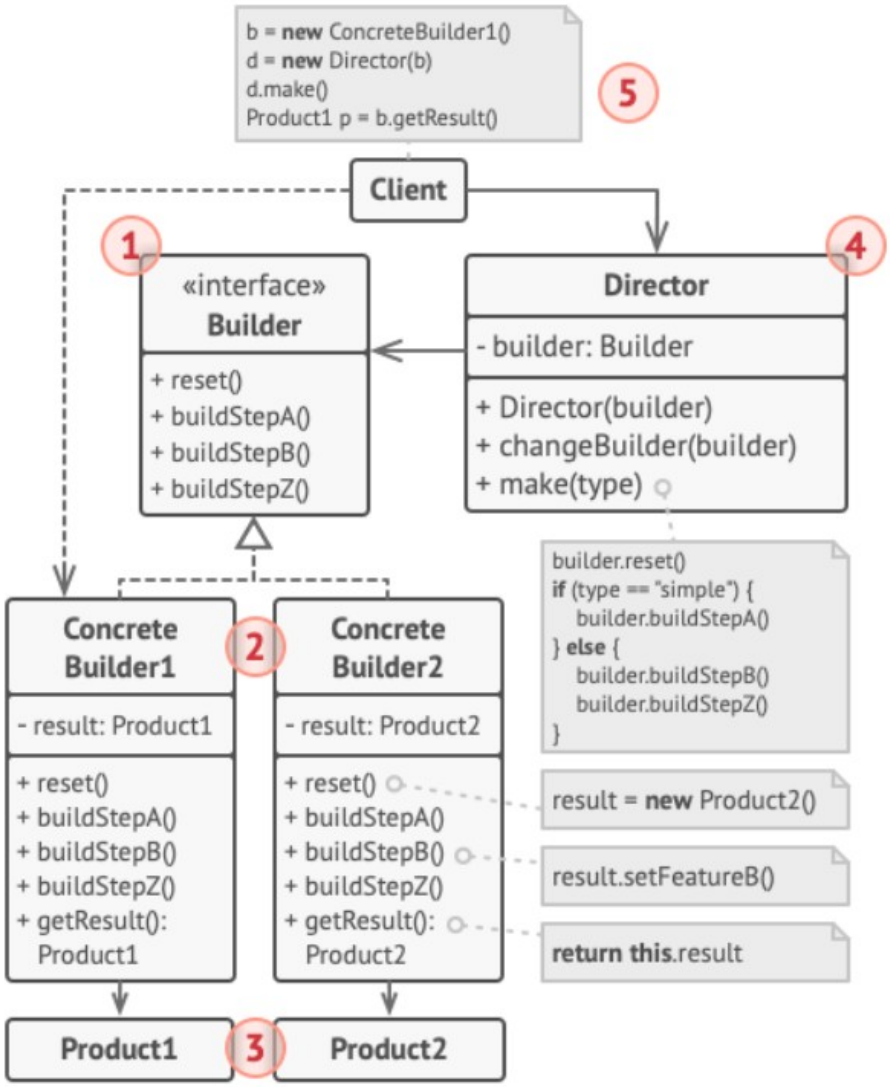
Преимущества и недостатки:
+ гарантирует сочетаемость создаваемых продуктов.
+ избавляет клиентский код от привязки к конкретным классам продуктов.
+ выделяет код производства продуктов в одно место, упрощая поддержку кода.
+ упрощает добавление новых продуктов в программу.
+ реализует принципе открытости/закрытости.
- усложняет код программы из-за введения множества дополнительных классов.
- требует наличия всех типов продуктов в каждой вариации.

Строитель (Builder) — порождающий паттерн проектирования, который позволяет создавать сложные объекты пошагово. Строитель даёт возможность

использовать один и тот же код строительства для получения разных представлений объектов.

Директор — класс, в который выделяются методы строителя. Данный класс задаёт порядок шагов строительства, а строитель то, как их выполнять.

Структура паттерна Строитель:



1. Интерфейс строителя — объявляет шаги конструирования продуктов, общие для всех видов строителей.
2. Конкретные строители — объявляет шаги конструирования продуктов, каждый по-своему. Конкретные строители могут производить разнородные объекты, не имеющие общего интерфейса.
3. Продукт — создаваемый объект. Продукты, сделанные разными строителями, не обязаны иметь общий интерфейс.
4. Директор — определяет порядок вызова строительных шагов для производства той или иной конфигурации продуктов.

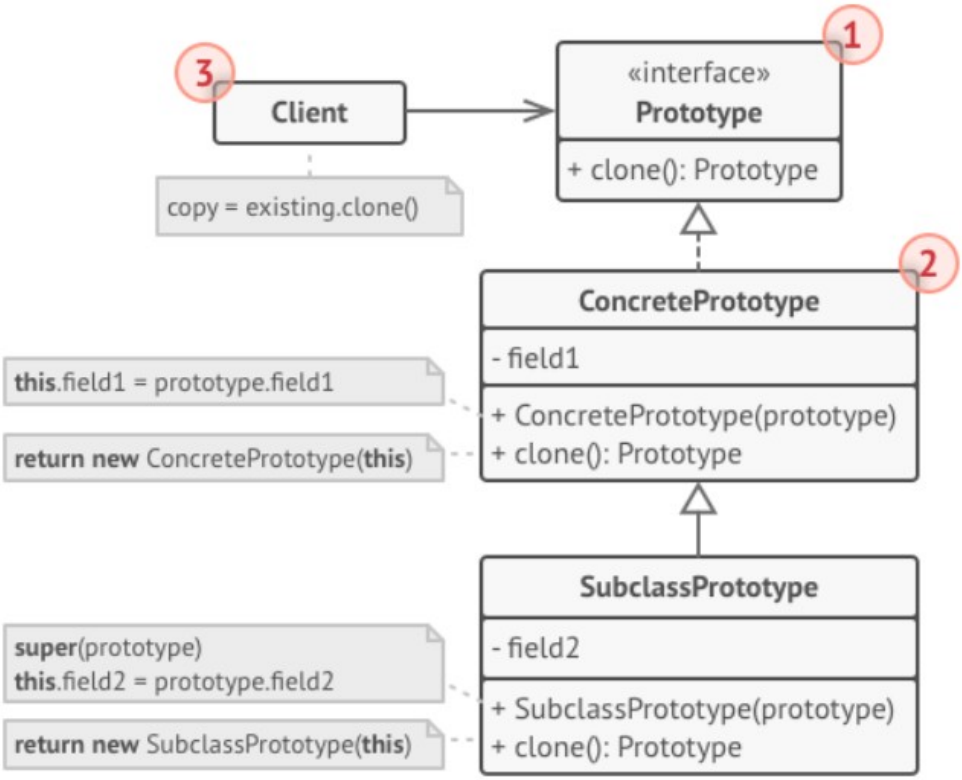
5. Обычно клиент подаёт в конструктор директора уже готовый объект строитель, и в дальнейшем директор использует только его. В другом варианте клиент передаёт строителя через параметр строительного метода директора.

Применение:
- когда нужно избавиться от «телескопического конструктора» (много раз перегруженного конструктора для всех типов создаваемого объекта).
- если нужно, чтобы код создавал разные представления какого-то объекта.
- если нужно собирать сложные составные объекты, например деревья Компоновщика.

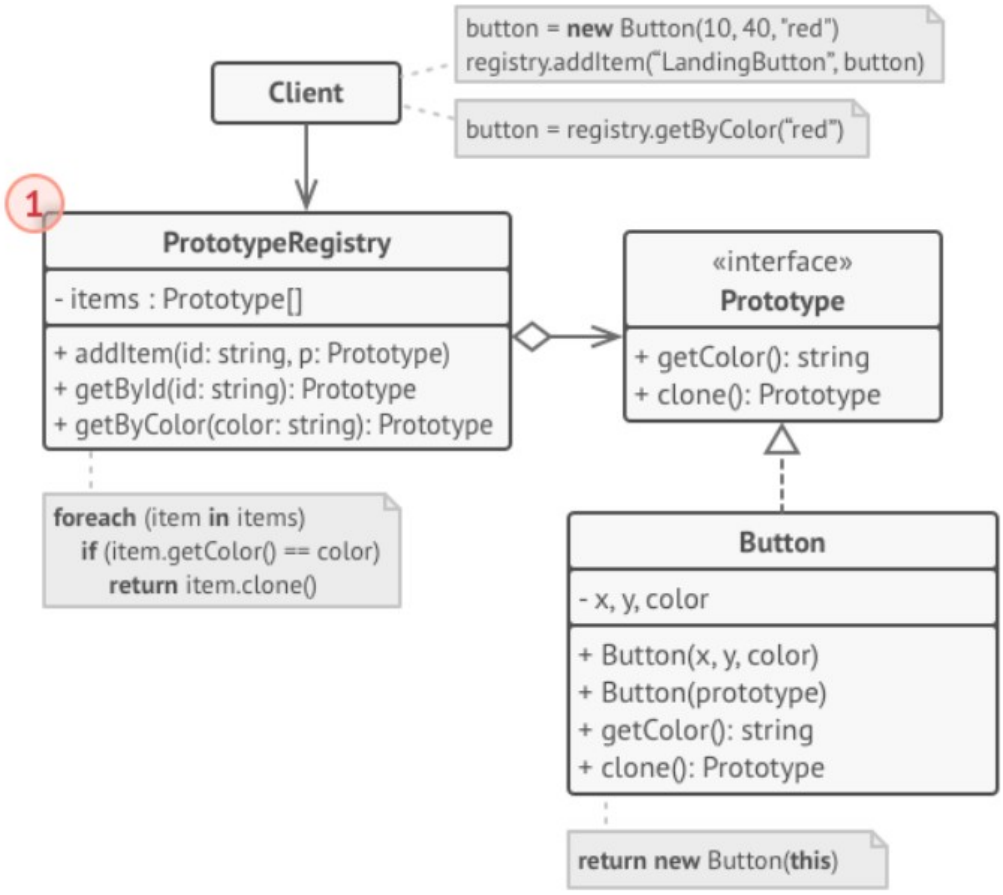
Преимущества и недостатки:
+ позволяет создавать продукты пошагово.
+ позволяет использовать один и тот же код для создания различных продуктов.
+ изолирует сложный код сборки продукта от его основной бизнес-логики.
- усложняет код программы из-за введения дополнительных классов.
- клиент будет привязан к конкретным классам строителей, так как в интерфейсе директора может не быть метода получения результата.

Прототип (Prototype) – порождающий паттерн проектирования, который позволяет копировать объекты, не вдаваясь в подробности их реализации.

Структура паттерна Прототип:
1. Интерфейс прототипов — описывает операции клонирования. В большинстве случаев это единственный метод clone().
2. Конкретный прототип — реализует операцию копирования самого себя. Помимо копирования всех полей здесь могут быть распутывания сложных методов копирования.
3. Клиент- создаёт копию объекта, обращаясь к нему через общий интерфейс прототипов.



Реализация с хранилищем прототипов:



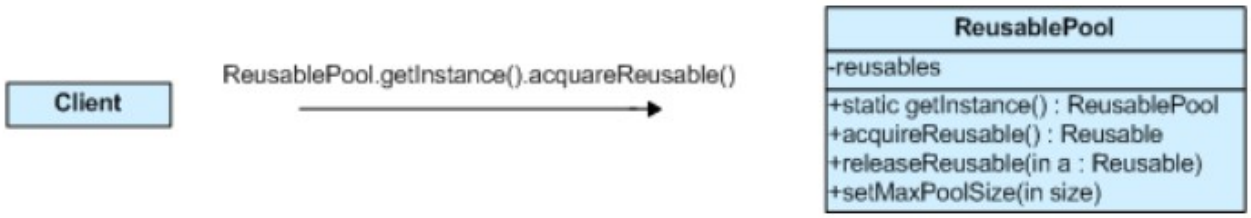
1. Хранилище прототипов — облегчает доступ к часто используемым прототипам, храня набор предварительно созданных эталонных, готовых к использованию объектов.

Применение:
- если код не должен зависеть от классов копируемых объектов.
- если есть множество подклассов, отличающихся начальными значениями полей — можно использовать набор прототипов вместо создания подклассов для описания популярных конфигураций объектов.

Преимущества и недостатки:
+ позволяет клонировать объекты, не привязываясь к их классам.
+ меньше повторяющегося кода инициализации объектов.
+ ускоряет создание объектов.
+ альтернатива созданию подклассов для конструирования сложных объектов.
- сложно клонироваться составные объекты, имеющие ссылки на другие объекты.

Пул объектов (Object Pool) – порождающий паттерн проектирования, позволяющий избежать создания новых экземпляров класса в случае возможности их повторного использования.

Структура паттерна Object Pool:



Reusable – экземпляры классов в этой роли взаимодействуют с другими объектами в течении 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.

Если в пуле нет ни одного свободного объекта, возможна одна из трёх стратегий:
- расширение пула
- отказ в создании пула, аварийная остановка

- в случае многозадачной системы можно организовать очередь, в которой запрос ожидает, пока один из объектов не освободится.

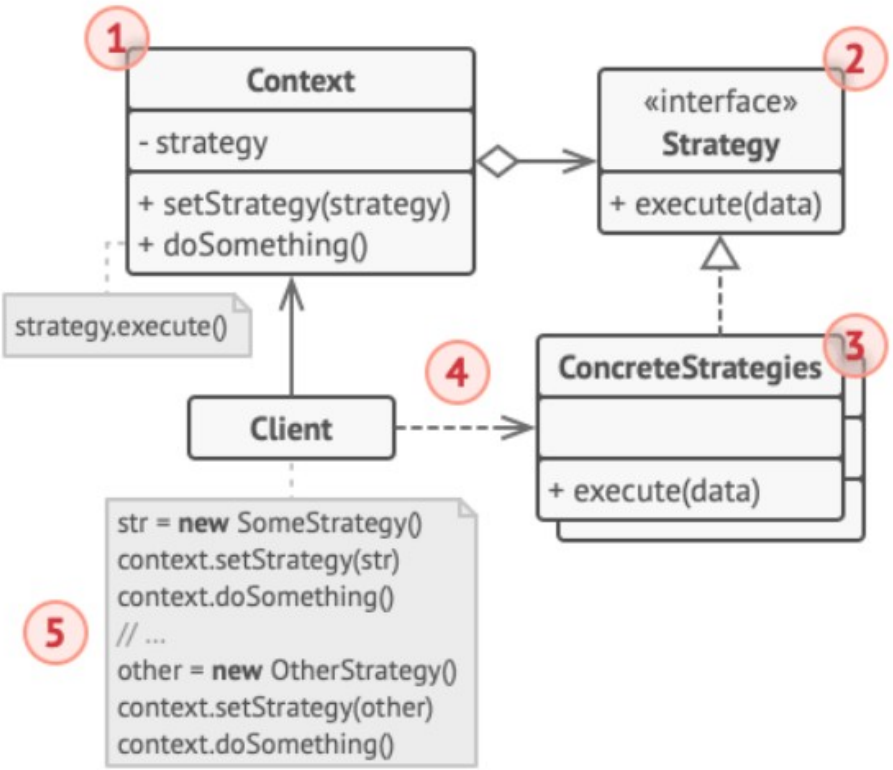
Применение:
- если инициализация объекта дороже, чем обслуживание объекта

Преимущества и недостаки:
+ положительно влияет на производительность из-за предварительной инициализации объектов
- при переполнении пула приходится особым образом реагировать, чтобы не возникла критическая ошибка
- повторное использование объектов может привести к утечке информации

14. Паттерны поведения: стратегия (Strategy), команда (Command), хранитель (Holder), посредник (Mediator), шаблонный метод (Template Method), итератор (Iterator), подписчик-издатель (Publish-Subscribe).

Стратегия (Strategy) – поведенческий паттерн проектирования, который определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы.

Структура паттерна:



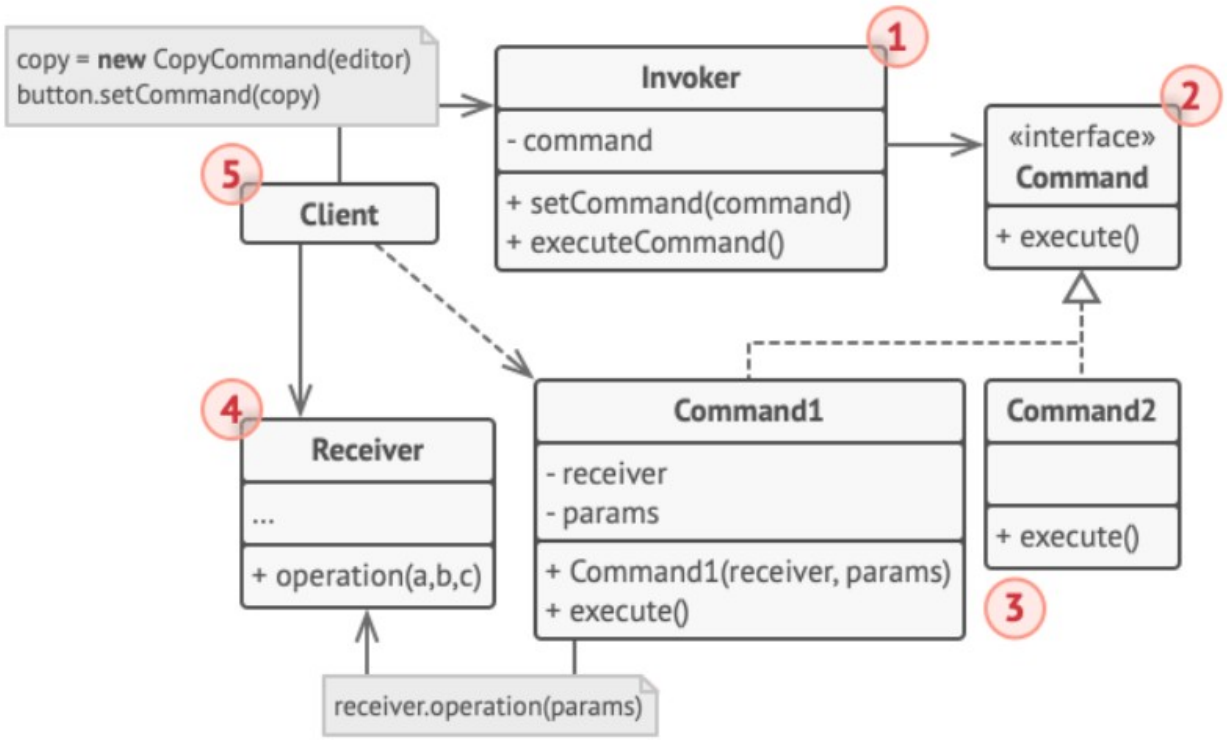
1. Контекст — хранит ссылку на объект конкретной стратегии, работая с ним через общий интерфейс стратегии.
2. Стратегия — определяет интерфейс, общий для всех вариаций алгоритма. Контекст использует этот интерфейс для вызова алгоритма.
3. Конкретные стратегии — реализуют различные вариации алгритма.
4. Во время выполнения программы контекст получает вызовы от клиента и делегирует их объекту конкретной стратегии.
5. Клиент должен создать объект конкретной стратегии и передать его в конструктор контекста. Кроме этого клиент должен иметь возможность заменить стратегию на лету, использую сеттер. Благодаря этому контекст не будет знать о том, какая именно стратегия сейчас выбрана.

Применение:
- когда нужно использовать разные вариации какого-то алгоритма внутри одного объекта.
- когда у нас есть множество похожих классов, отличающихся только некоторым поведением.
- когда мы не хотим обнажать детали реализации алгоритмов для других классов.
- когда различные вариации алгоритмов реализованы в виде развесистого условного оператора. Каждая ветка такого оператора представляет собой вариацию алгоритма.

Преимущества и недостатки:
+ замена алгоритмов на лету
+ изолирует код и данные алгоритмов от остальных классов
+ уход от наследования к делегированию
+ реализация принципа открытости/закрытости
- усложняет программу за счёт дополнительных классов
- клиент должен знать, в чём состоит разница между стратегиями, чтобы выбрать подходящую.

Команда (Command) – поведенческий паттерн проектирования, который превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.

Структура паттерна Команда:



1. Отправитель — хранит ссылку на объект команды и обращается к нему, когда нужно выполнить какое-то действие. Отправитель работает с командами через общий интерфейс. Он не знает, какую конкретно команду использует, так как получает готовый объект команды от клиента.
2. Интерфейс команды — общий интерфейс для всех конкретных команд.
3. Конкретные команды — реализуют различные запросы, следуя общему интерфейсу команд.
4. Получатель — содержит бизнес-логику программы.
5. Клиент — создаёт объекты конкретных команд, передавая в них все необходимые параметры, среди которых могут быть и ссылки на объекты получателей. После этого клиент связывает объекты отправителей с созданными командами.

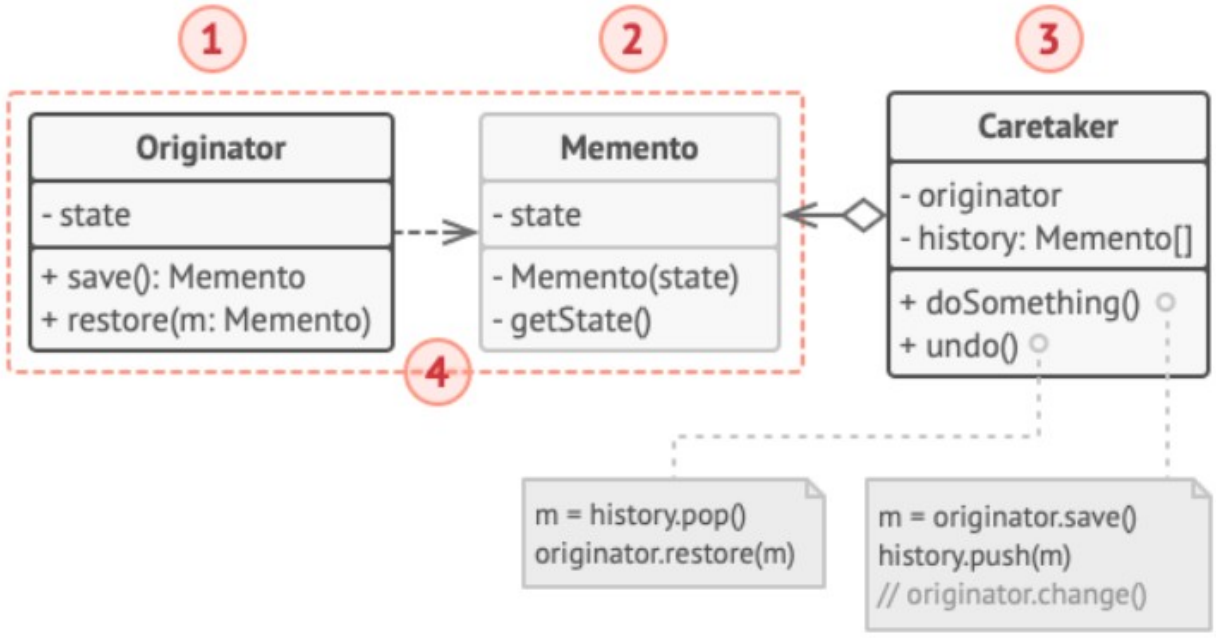
Применение:
- если нам нужно параметризовать объекты выполняемым действием.
- если нам нужно ставить операции в очередь, выполнять их по расписанию или передавать по сети.
- если нам нужна операция отмены — реализация с помощью стека команд, каждая из которых сохраняет состояния объекта.

Преимущества и недостатки:
+ убирает прямую зависимость между объектами, вызывающими операции, и объектами, которые их непосредственно выполняют.
+ позволяет реализовать простую отмену и повтор операций.

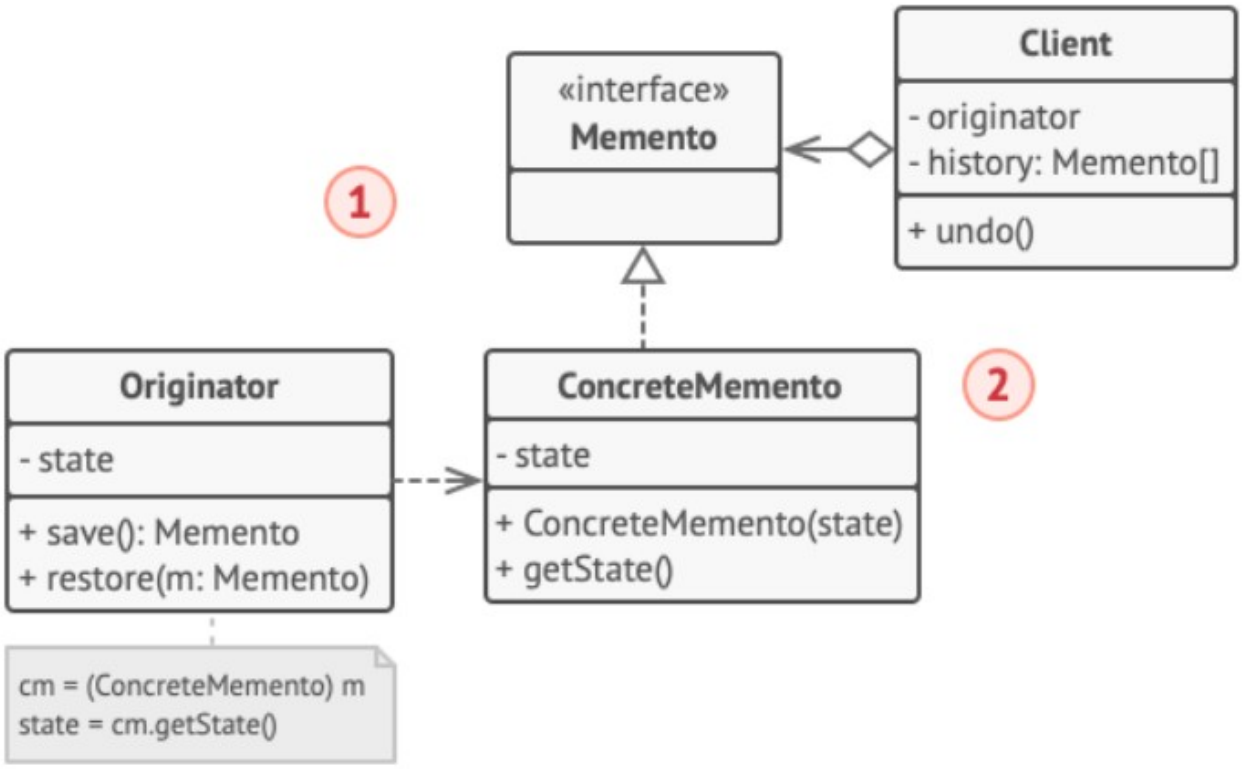
+ позволяет реализовать отложенный запуск операций.
+ позволяет собирать сложные команды из простых.
+ реализует принцип открытости/закрытости.
- усложняет код программы из-за введения множества дополнительных классов.

Хранитель (Holder, Memento) – поведенческий паттерн проектирования, который позволяет сохранять и восстанавливать прошлые состояния объектов, не раскрывая подробностей их реализации.

Структура паттерна Хранитель:
1. Создатель — может производить снимки своего состояния, а также воспроизводить прошлое состояние, если подать в него готовый снимок.
2. Снимок — это простой объект данных, содержащий состояние создателя. Надёжнее всего сделать объекты снимков неизменяемыми, передавая в них состояние только через конструктор.
3. Опекун — этот класс отвечает за то, когда нужно делать снимок, и когда его нужно восстанавливать.
4. В данной реализации снимок — внутренний класс по отношению к классу создателя, поэтому он имеет доступ ко всем полям создателя.



Реализация с промежуточным интерфейсом:



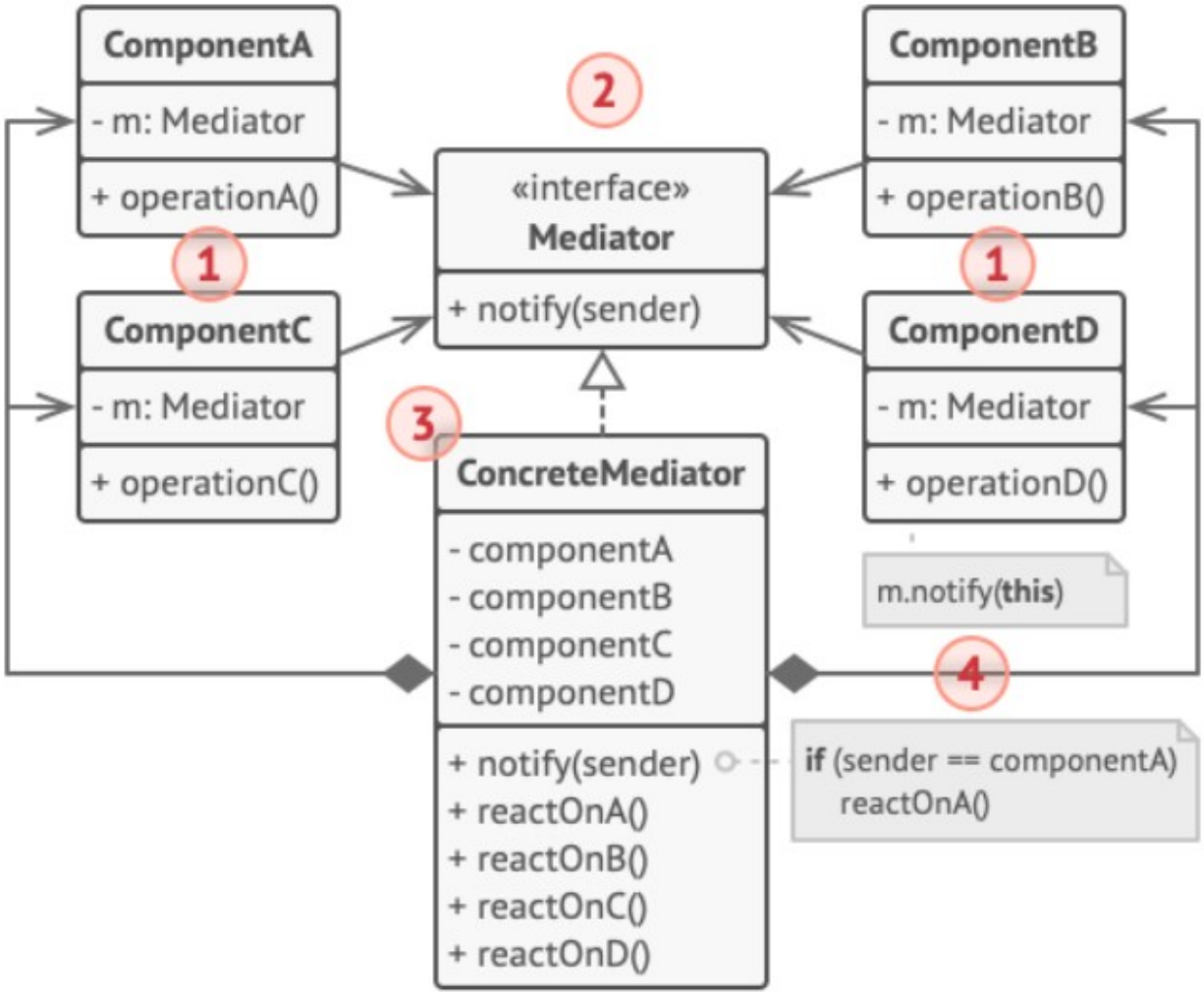
1. В этой реализации создатель напрямую работает с классом снимка, а опекун только с его ограниченным интерфейсом.
2. Благодаря этому достигается тот же эффект — создатель имеет полный доступ к снимку, а опекун нет.

Применение:
- если нужно сохранять мгновенные снимки состояния объекта, чтобы впоследствии объект можно было восстановить в том же состоянии.
- если прямое получение состояние объекта раскрывает приватные детали его реализации, нарушая инкапсуляцию.

Преимущества и недостатки:
+ не нарушает инкапсуляцию исходного объекта.
+ упрощает структуру исходного объекта.
- требует много памяти, если снимки создаются слишком часто.
- может повлечь дополнительные издержки памяти, если объекты, хранящие историю, не освобождают ресурсы, занятые устаревшими снимками.
- в некоторых языках сложно гарантировать, чтобы только исходный объект имел доступ к состоянию снимка

Посредник (Mediator) – поведенческий паттерн проектирования, который позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник.

Структура паттерна посредник:



1. Компоненты — разнородные объекты, содержащие бизнес-логику программы. Каждый компонент хранит ссылку на объект посредника, но работает с ним только через абстрактный интерфейс посредников. Благодаря этому компоненты можно повторно использовать в другой программе, связав их с посредником другого типа.
2. Посредник — определяет интерфейс для обмена информацией с компонентами. Обычно хватает одного метода, чтобы оповещать посредника о событиях, произошедших в компонентах. В параметрах этого метода можно передавать детали события: ссылку на компонент, в котором оно произошло, и любые другие данные.
3. Конкретный посредник — содержит код взаимодействия нескольких компонентов между собой. Зачастую этот объект не только хранит ссылки на

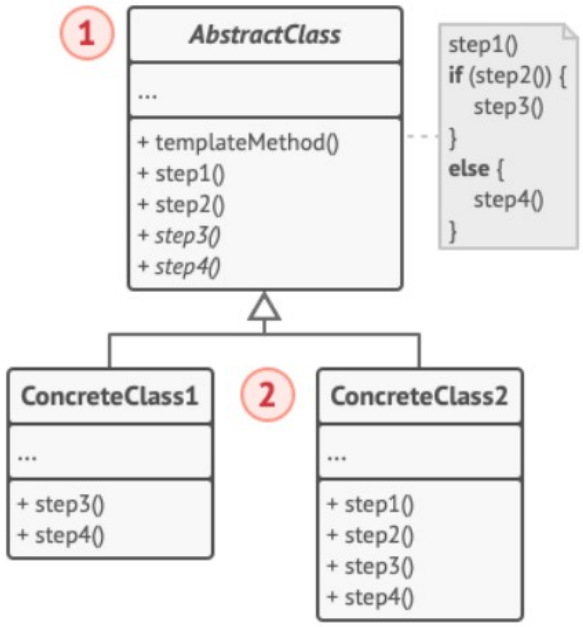
все свои компоненты, но и сам их создаёт, управляя их дальнейшим жизненным циклом.
4. Компоненты не должны общаться друг с другом напрямую. Если в компоненте происходит важное событие, то он должен оповестить посредника, а тот сам решит — касается ли событие других компонентов, и стоит ли их оповещать.

Применение:
- если сложно менять некоторые классы из-за того, что они имеют множество хаотичных связей с другими классами.
- когда мы не можем повторно использовать класс, поскольку он зависит от множества других классов.
- если нам приходится создавать множество подклассов компонентов, чтобы использоваться одни и те же компоненты в разных контекстах.

Преимущества и недостатки:
+ устраняет зависимости между компонентами, позволяя повторно их использовать.
+ упрощает взаимодействие между компонентами.
+ централизует управление в одном месте.
- посредник может сильно раздуться.

Шаблонный метод (Template Method) – поведенческий паттерн проектирования, который определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подкласса переопределять шаги алгоритма, не меняя его общей структуры.

Структура Шаблонного Метода:



1. Абстрактный класс — определяет шаги алгоритма и содержит шаблонный метод, состоящий из вызовов этих шагов. Шаги могут быть как абстрактными, так и содержать реализацию по умолчанию.
2. Конкретный класс — переопределяет некоторые (или все) шаги алгоритма. Конкретные класс не переопределяют сам шаблонный метод.

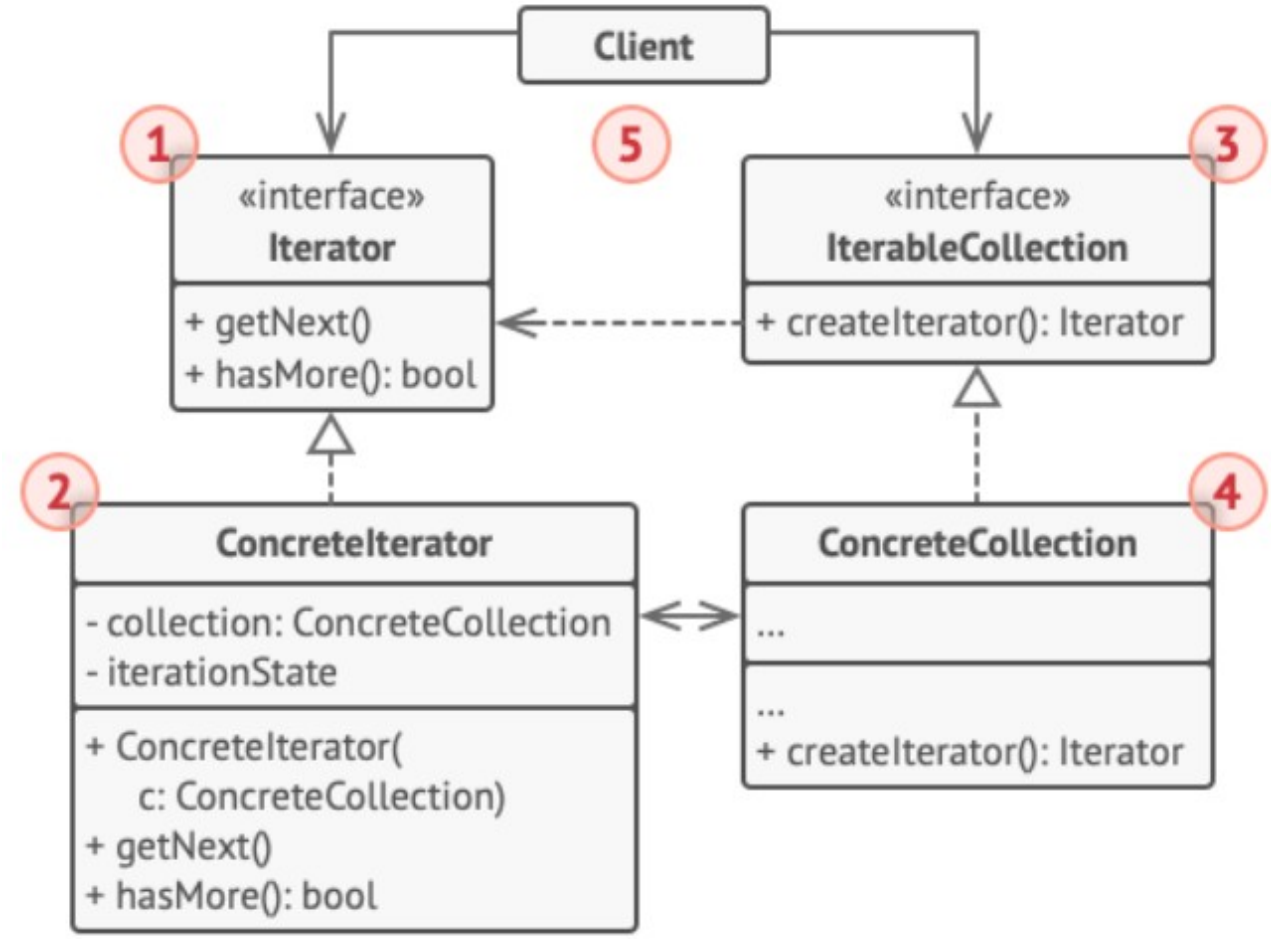
Применение:
- если подклассы должны расширять базовый алгоритм, не меняя его структуры.
- если есть несколько классов, делающих одно и то же с незначительными отличиями. Если вы редактируете один класс, то приходится вносить такие же правки и в остальные классы.

Преимущества и недостатки:
+ облегчает повторное использование кода.
- жёсткое ограничение скелетом существующего алгоритма.
- можно нарушить принцип подстановки Барбары Лисков (функции которые используют базовый тип должны иметь возможность использовать подтипы базового типа не зная об этом), изменяя базовое поведение одного из шагов алгоритма через подкласс.
- с ростом количества шагов шаблонный метод становится слишком сложно поддерживать.

Итератор (Iterator) – поведенческий паттерн проектирования, который даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.

Структура:
1. Итератор — описывает интерфейс для доступа и обхода элементов коллекции.
2. Конкретный итератор — реализует алгоритм обхода какой-то конкретной коллекции. Объект итератора должен сам отслеживать текущую позицию про обходе коллекции, чтобы отдельные итераторы могли обходить одну и ту же коллекцию независимо.
3. Коллекция — описывает интерфейс получения итератора из коллекции.
4. Конкретная коллекция — возвращает новый экземпляр определённого конкретного итератора, связав его с текущим объектом коллекции.
5. Клиент — работает со всеми объектами коллекции через интерфейсы коллекции и итератора. Таким образом клиентский код не зависит от

конкретных классов, что позволяет применять различные итераторы, не изменяя существующий код программы.

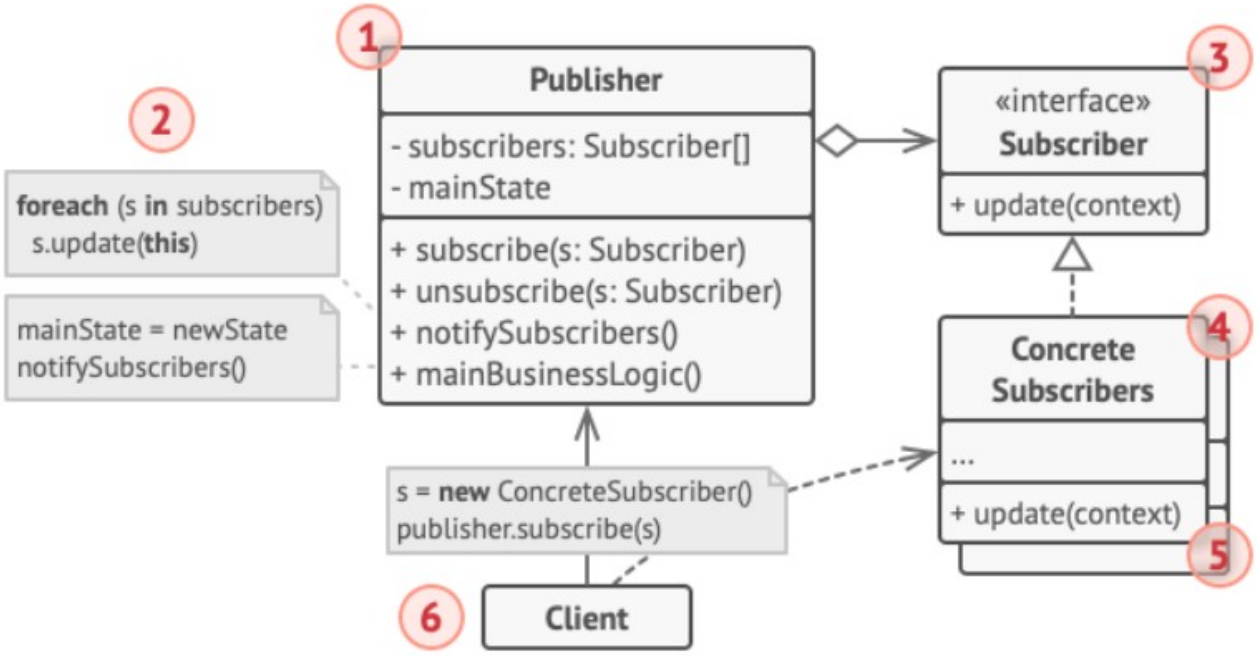


Применение:
- если у нас есть сложная структура данных и мы хотим скрыть от клиента детали её реализации.
- если нам нужно иметь несколько вариантов обхода одной и той же структуры данных.
- если нам хочется иметь единый интерфейс обхода различных структур данных.

Преимущества и недостатки:
+ упрощает классы хранения данных.
+ позволяет реализовать различные способы обхода структуры данных.
+ позволяет одновременно перемещаться по структуре данных в разные стороны.
- не оправдан, если можно обойтись простым циклом.

Подписчик-издатель (Publish-Subscribe, Observer) – поведенческий паттерн проектирования, который создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.

Структкура:



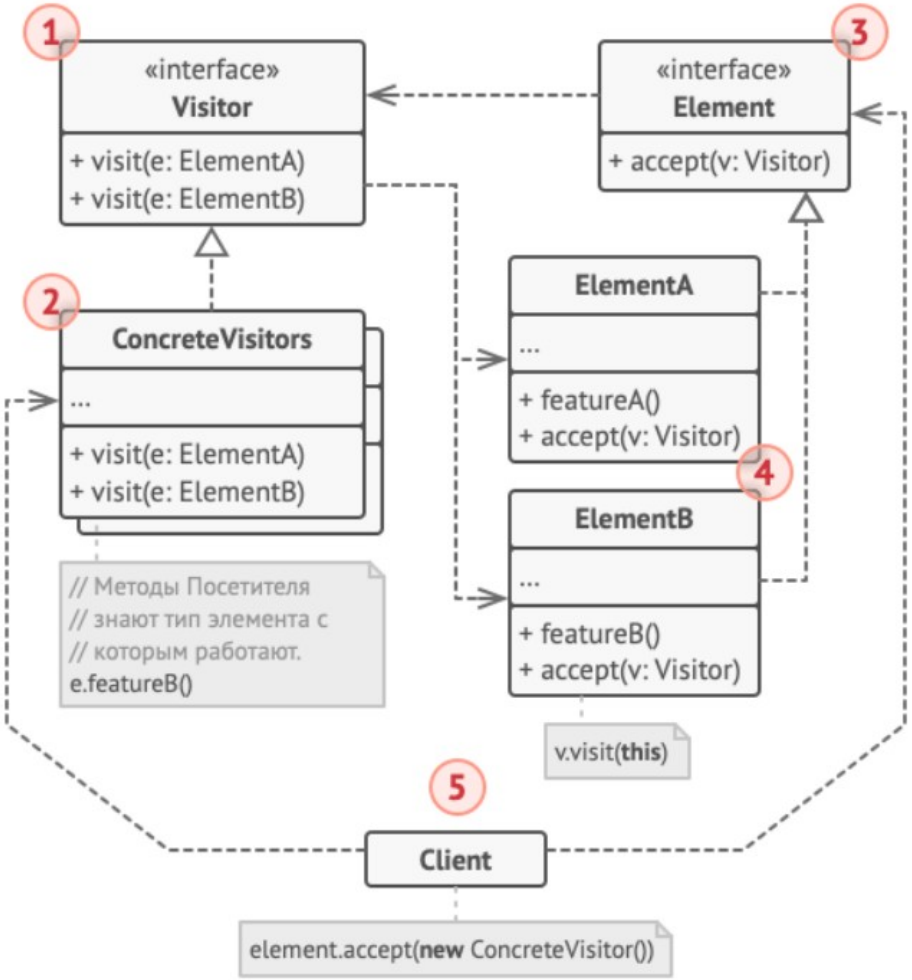
1. Издатель — владеет внутренним состоянием, изменение которого интересно отслеживать подписчикам. Издатель содержит механизм подписки: список подписчиков и методы подписки/отписки.
2. При изменении внутреннего состояния издатель оповещает всех подписчиков.
3. Подписчик — определяет интерфейс, которым пользуется издатель для отправки оповещения. В большинстве случаев для этого достаточно единственного метода.
4. Конкретные подписчики — выполняют что-то в ответ на оповещение, пришедшее от издателя.
5. По приходу оповещение подписчик должен получить обновлённое состояние издателя.
6. Клиент — создаёт объекты издателей и подписчиков, а затем регистрирует подписчиков на обновления в издателях.

Применение:
- если после изменения состояния одного объекта требуется что-то сделать в других, но мы не знаем точно, какие объекты должны отреагировать.
- если одни объекты должны наблюдать за другими, но только в определённых случаях.

Преимущества и недостатки:
+ издатели не зависят от конкретных классов подписчиков и наоборот.
+ мы можем подписывать и отписывать получателей на лету.
+ реализует принцип открытости/закрытости
- подписчики оповещаются в случайном порядке

Посетитель (Visitor) - поведенческий паттерн проектирования, который позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться.

Структура:



1.Посетитель — описывает общий интерфейс для всех типов посетителей. Он объявляет набор методов, отличающихся типом входящего параметра, которые нужны для запуска операции для всех типов конкретных элементов. В языках, поддерживающих перегрузку методов, эти методы могут иметь одинаковые имена, но типы их параметров должны отличаться.

2. Конкретные посетители — реализуют какое-то особенное поведение для всех типов элементов, которые можно подать через методы интерфейса посетителя.
3. Элемент — описывает метод принятия посетителя. Этот метод должен иметь единственный параметр, объявленный с типом интерфейса посетителя.
4. Конкретные элементы — реализуют методы принятия посетителя. Цель этого метода — вызвать тот метод посещения, который соответствует типу этого элемента. Так посетитель узнает, с каким именно элементом он работает.
5. Клиент — коллекция или другой сложный объект.

Применение:
- если нам нужно выполнить какую-то операцию над всеми элементами сложной структуры объектов, например, деревом.
- если над объектами сложной структуры объектов надо выполнять некоторые не связанные между собой операции, но мы не ходим засорять классы такими операции.
- если новое поведение имеет смысл только для некоторых классов из существующий иерархии.

Преимущества и недостатки:
+ упрощение добавления операций, работающих со сложными структурами объектов.
+ объединение родственных операций в одном классе.
+ посетитель может накапливать состояние при обходе структуры элементов.
- паттерн не оправдан, если иерархия элементов часто меняется
- может привести к нарушению инкапсуляции