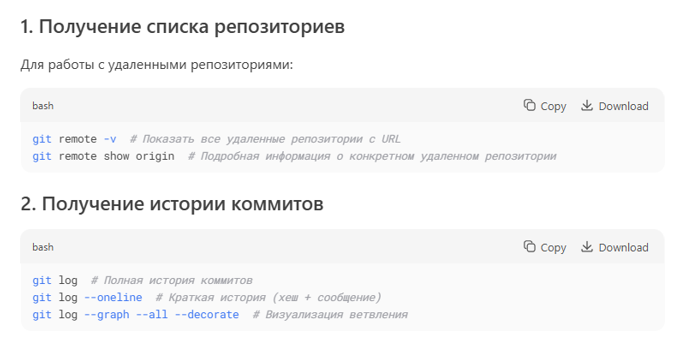
# **Тема 1. Использование системы контроля версий Git. Основные команды и возможности.**



**3. Получение списка всех веток**

Все ветки:

git branch –a

Локальные ветки

git branch

Удаленные ветки

Git branch –r

**4. Получение списка меток**

git tag

**5. Индексация файла**

Индексация (или добавление файла в staging area) — это подготовка файлов к последующему коммиту.

Основные команды:

git add <file> — добавляет конкретный файл в индекс.

git add . — добавляет все измененные и новые файлы в текущей директории.

git add -A или git add --all — добавляет все измененные и новые файлы во всем проекте

git add -u — добавляет только измененные и удаленные файлы (но не новые).

**6. Удаление файла**

Удаление только из индекса (оставление в рабочей директории).

1. Полное удаление файла (из Git и файловой системы)

git rm <file> — удаляет файл из индекса и рабочей директории.

2. Удаление только из индекса (оставить в файловой системе)

git rm --cached <file> — файл останется на диске, но Git перестанет его отслеживать.

3. Удаление папки

git rm -r <directory> — рекурсивное удаление папки.

**7. Добавление ветви**

Основная команда:

git branch <имя-новой-ветки>

Создание ветки и немедленный переход на нее:

git checkout -b <имя-новой-ветки>

**8. Переход на другую ветвь**

Классическая команда:

git checkout <имя-существующей-ветки>

Более новая

git switch <имя-существующей-ветки>

**9. Удаление ветви**

Локально:

Чтобы удалить локальную ветку, используется команда:

git branch -d <название-ветки>

-d (delete) удаляет ветку только если все изменения в ней слиты с текущей веткой.

Если хочешь удалить ветку принудительно, даже без слияния:

git branch -D <название-ветки>

Удаление удалённой ветки (на сервере, например, GitHub):

git push origin --delete <название-ветки>

**10. Разрешение конфликтов**

Конфликты возникают, когда Git не может автоматически объединить изменения из разных веток. Пример: два человека отредактировали одну и ту же строку кода.

Процесс разрешения:

Попытка слияния:

git merge <другая-ветка>

или при git pull, если есть изменения на сервере.

Git сообщает о конфликте и помечает конфликтующие файлы как "unmerged".

Чтобы разрешить конфликт нужно вручную открыть файл и выбрать необходимые куски кода:  
Git вставит специальные метки в файл:

markdown

<<<<<<< HEAD

твои изменения

=======

изменения из другой ветки

>>>>>>> ветка

— Выбери или объедини нужные куски, убери метки <<<<<<<, =======, >>>>>>>.

После разрешения:

git add <имя-файла>

git commit

или если конфликт возник при merge, можно использовать:

git merge --continue

**11. Создание репозитория**

Локально:

Если у тебя есть папка с проектом, и ты хочешь инициализировать Git:

cd <папка-с-проектом>

git init

Это создаст скрытую папку. git, где хранятся все данные Git.

Далее можно

git add .

git commit -m "Первый коммит"

С удалённым репозиторием (например, GitHub):

Создай репозиторий на GitHub вручную.

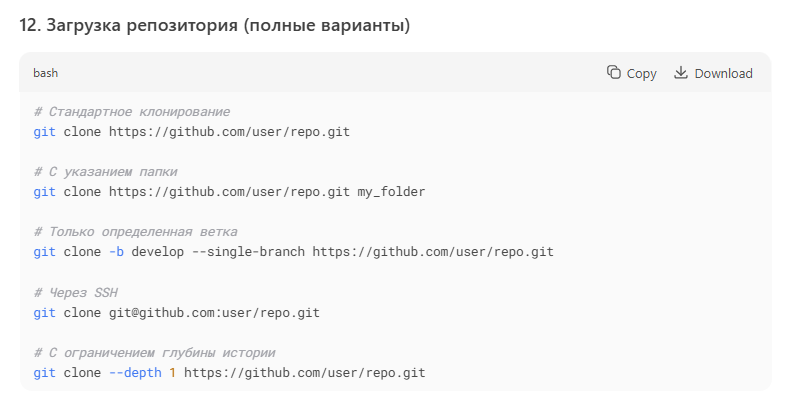
Привяжи его:

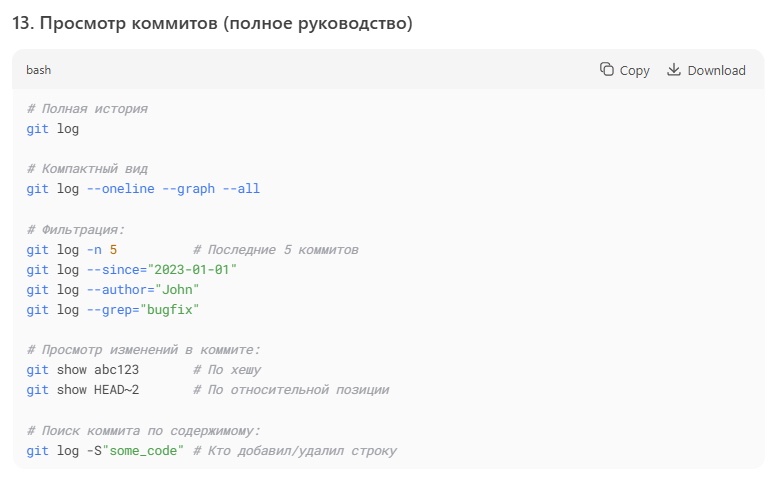
git remote add origin https://github.com/твой-логин/название-репозитория.git

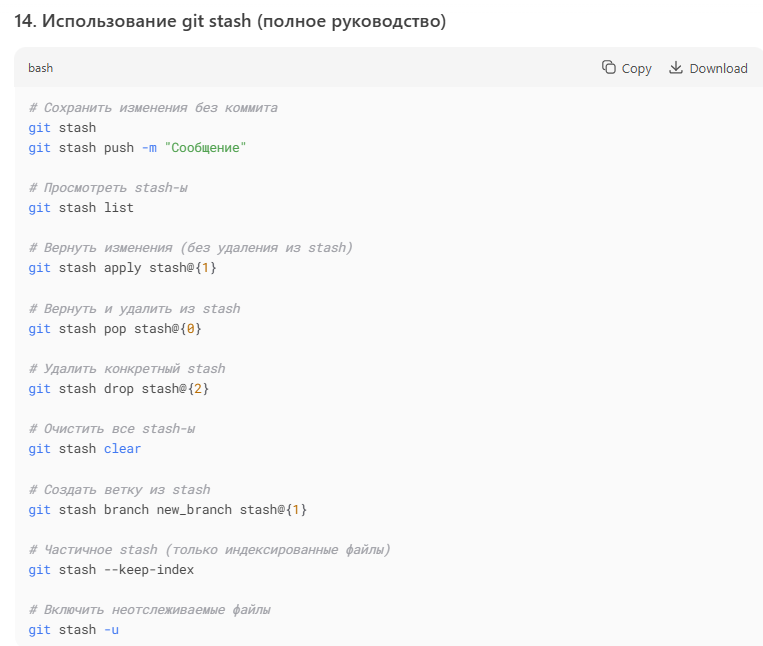
Отправь изменения:

git push -u origin master

(или main, если ветка называется так)







# **Тема 2. Шаблоны проектирования**

**15. Объектно-ориентированное проектирование (ООП)**

**Определение:** Методология проектирования программных систем на основе объектов, классов и их взаимодействия.

**Основные принципы:**

1. **Абстракция** — выделение ключевых характеристик объекта (например, класс Car с полями model, speed).
2. **Инкапсуляция** — скрытие внутренней реализации (private-поля + public-методы).
3. **Наследование** — повторное использование кода (класс ElectricCar наследует Car).
4. **Полиморфизм** — возможность объектов вести себя по-разному (переопределение методов).

**16. Назначение паттернов проектирования**

**Паттерны** — это типовые решения часто встречающихся проблем в ООП.

**Зачем нужны:**

* ✅ **Стандартизация** — единый язык для разработчиков.
* ✅ **Ускорение разработки** — не изобретать велосипед.
* ✅ **Повышение гибкости** — легче модифицировать код.
* ✅ **Снижение ошибок** — проверенные подходы.

**Категории паттернов (по GoF):**

1. **Порождающие** (Creational) — создание объектов (Singleton, Factory).
2. **Структурные** (Structural) — компоновка классов (Adapter, Composite).
3. **Поведенческие** (Behavioral) — взаимодействие объектов (Observer, Strategy).

**17. Отличие паттернов проектирования**

Паттерны различаются по:

1. **Решаемой проблеме**:
   * Singleton — гарантирует единственный экземпляр класса.
   * Observer — уведомляет объекты об изменениях.
2. **Уровню применения**:
   * **Классовые** (Template Method) — работают на уровне наследования.
   * **Объектные** (Command) — используют композицию.
3. **Гибкости**:
   * Factory Method — позволяет подклассам выбирать создаваемый объект.
   * Abstract Factory — создаёт семейства связанных объектов.

**18. Абстрактная фабрика: назначение, использование**

**Абстрактная фабрика (Abstract Factory)** — это порождающий паттерн проектирования, который предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не указывая их конкретных классов.

**Назначение**

1. **Создание групп связанных объектов** – позволяет создавать объекты, которые должны работать вместе.
2. **Изоляция клиентского кода от конкретных классов** – клиент работает только с абстрактными интерфейсами.
3. **Гибкость и масштабируемость** – упрощает добавление новых семейств объектов.

**Использование**

Применяется, когда:

* Система должна быть независимой от процесса создания объектов.
* Необходимо создавать семейства связанных объектов (например, элементы UI для разных ОС).
* Требуется обеспечить консистентность используемых объектов.



**Плюсы**

✔ Избегает жесткой привязки к конкретным классам.  
✔ Упрощает замену семейств объектов.  
✔ Гарантирует согласованность создаваемых объектов.

**Минусы**

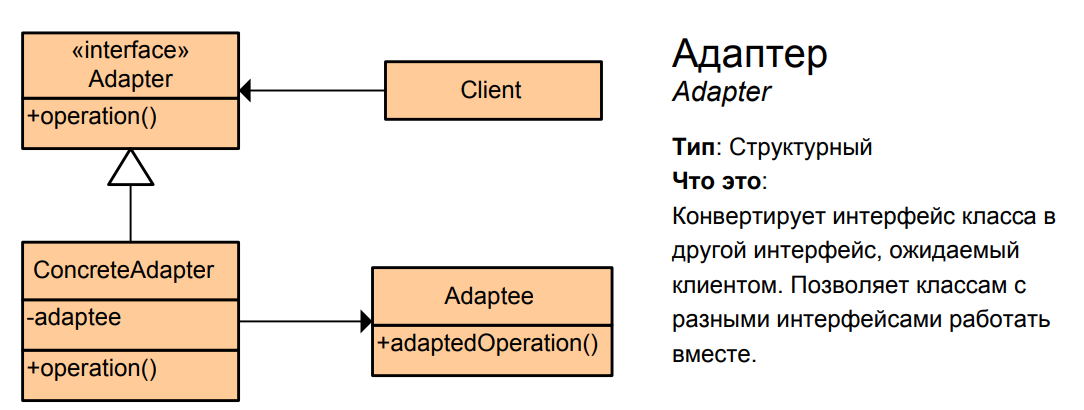
✖ Усложняет код из-за введения множества классов.  
✖ Трудно добавить поддержку нового типа объекта.

**Вывод:** Абстрактная фабрика полезна, когда система должна конфигурироваться целыми семействами объектов, а их конкретные реализации могут варьироваться.

**19. Адаптер: назначение, использование**

Конвертирует интерфейс класса в другой интерфейс, ожидаемый клиентом. Позволяет классам с разными интерфейсами работать вместе, иными словами обеспечивает совместимость интерфейсев.

Вместо переписывания объекта или клиента, можно обернуть объект адаптером — специальным классом, который преобразует вызовы в нужный формат.



class OldAPI:

def old\_method(self): print('old')

class Adapter:

def new\_method(self):

OldAPI().old\_method()

**20. Диспетчер: назначение, использование**

Распределяет задачи, сообщения или события между различными обработчиками в системе, централизует обработку.

Пример: Event Dispatcher вызывает нужные обработчики при наступлении событий в приложении.

class Dispatcher:

def handle(self, event):

if event == "click":

self.on\_click()

**21. Заместитель: назначение, использование**

Предоставляет замену другого объекта для контроля доступа к нему

class Image:

def display(self): pass

class RealImage(Image):

def \_\_init\_\_(self, filename):

self.filename = filename

print(f"Загружено изображение: {filename}")

def display(self):

print(f"Показать изображение: {self.filename}")

class ImageProxy(Image):

def \_\_init\_\_(self, filename):

self.filename = filename

self.real\_image = None

def display(self):

if self.real\_image is None:

# Ленивая загрузка, только при первом вызове!

self.real\_image = RealImage(self.filename)

self.real\_image.display()

# Использование:

img = ImageProxy("photo.png")

img.display() # Загружено изображение: photo.png \n Показать изображение: photo.png

img.display() # Показать изображение: photo.png (без повторной загрузки)

**22. Итератор: назначение, использование**

Назначение:

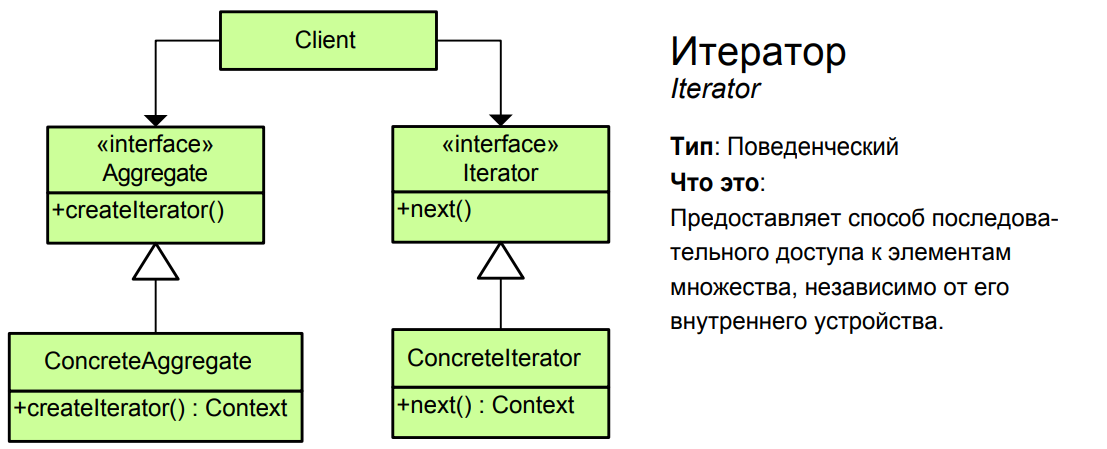
Паттерн Итератор (Iterator) предназначен для:

* последовательного обхода элементов какой-либо коллекции (например: список, множество, дерево), без раскрытия внутренней структуры этой коллекции.
* Итератор скрывает детали устройства коллекции и предоставляет стандартный способ её обхода.

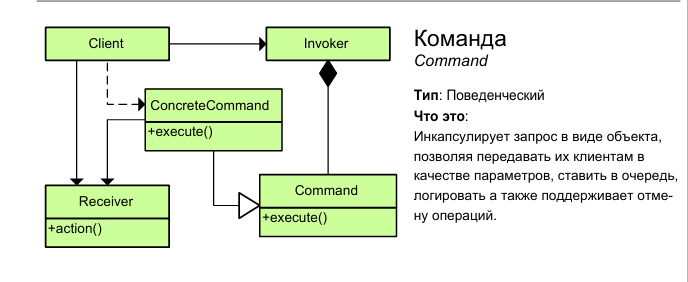
Пример:

for element in collection:

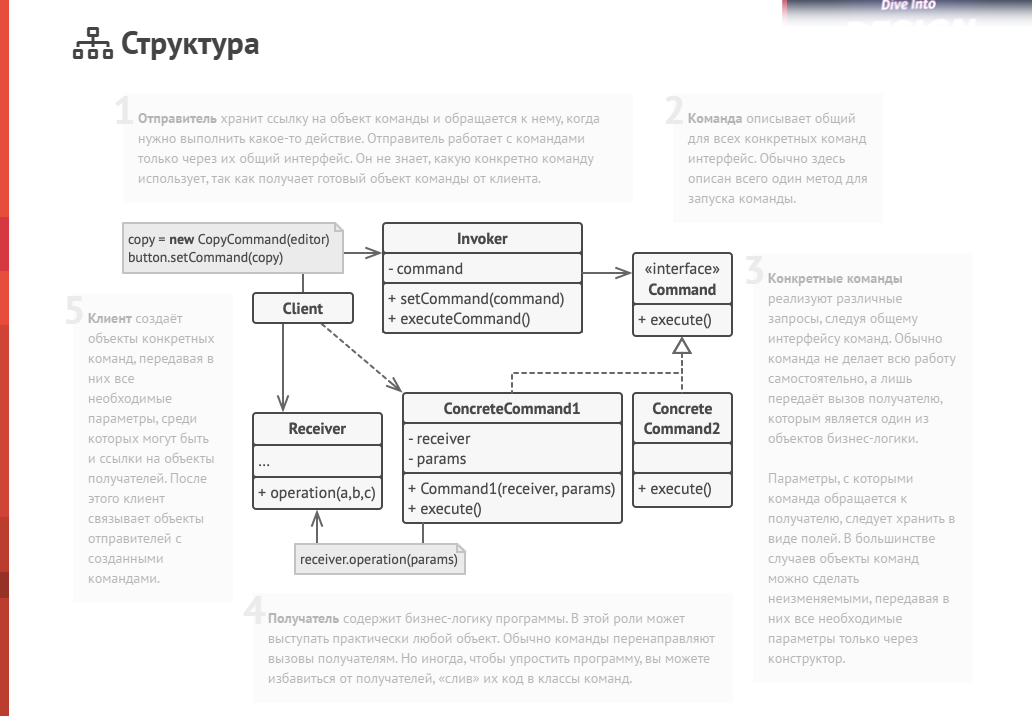
Под капотом у коллекции есть методы \_\_iter\_\_ и \_\_next\_\_, которые реализуют паттерн Итератор.



**23. Команда: назначение, использование**

****

**Команда** — это 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

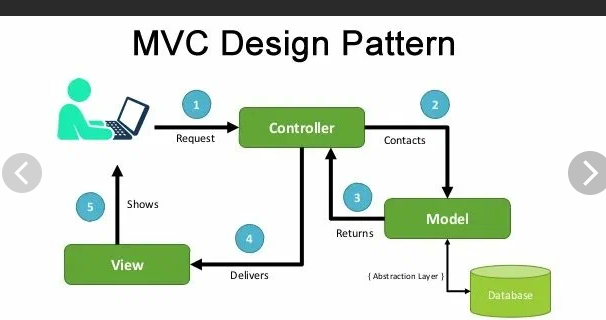
****

**24. Контроллер: назначение, использование**

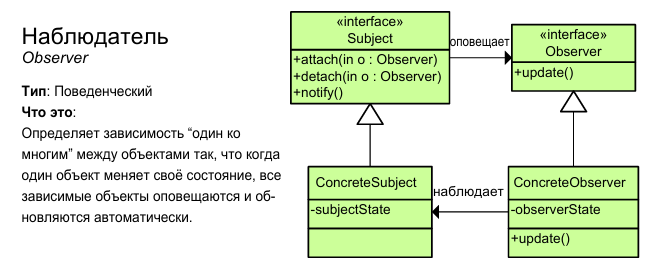
**Контроллер** — это структурный шаблон проектирования, часто используемый в архитектуре Model–View–Controller (MVC).

Контроллер выполняет роль посредника между пользователем (интерфейсом) и бизнес-логикой (моделью). Он:

* Принимает входные данные от пользователя (нажатия, запросы, действия);
* Обрабатывает их и инициирует изменения в модели;
* Обновляет или управляет отображением данных через представление (View).

****

**25. Наблюдатель: назначение, использование**

****

**Наблюдатель** — это поведенческий паттерн проектирования, который создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.

Проблема

Представьте, что вы имеете два объекта: Покупатель и Магазин. В магазин вот-вот должны завезти новый товар, который интересен покупателю.

Покупатель может каждый день ходить в магазин, чтобы проверить наличие товара. Но при этом он будет злиться, без толку тратя своё драгоценное время.

Постоянное посещение магазина или спам?

С другой стороны, магазин может разослать спам каждому своему покупателю. Многих это расстроит, так как товар специфический, и не всем он нужен.

Получается конфликт: либо покупатель тратит время на периодические проверки, либо магазин тратит ресурсы на бесполезные оповещения.

Решение

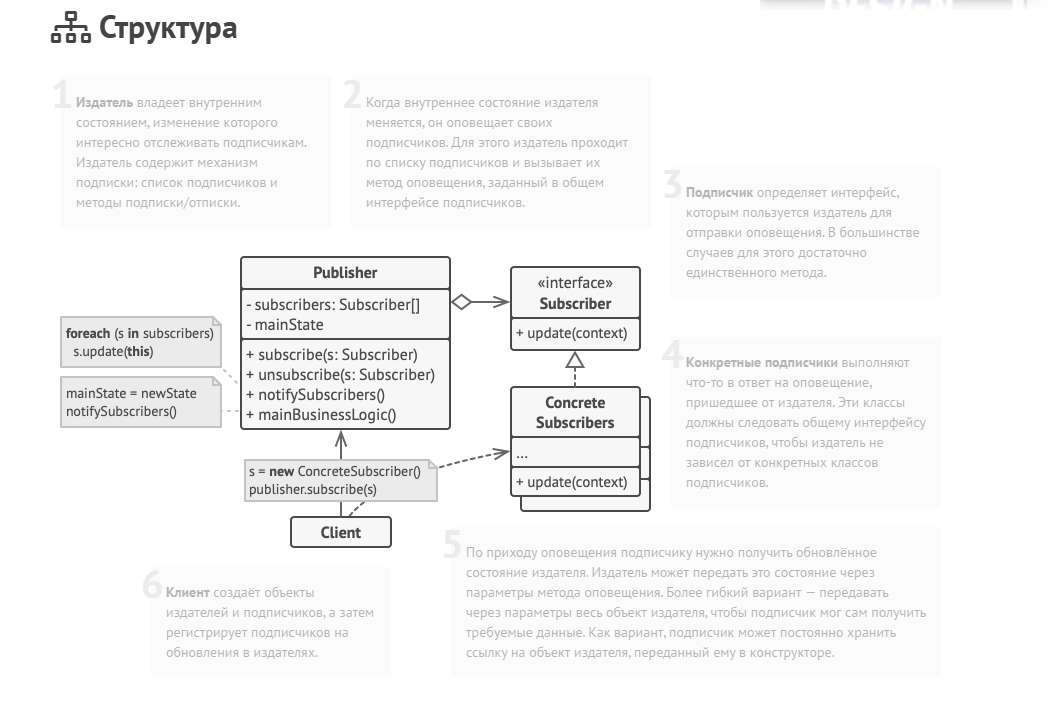
Давайте называть Издателями те объекты, которые содержат важное или интересное для других состояние. Остальные объекты, которые хотят отслеживать изменения этого состояния, назовём Подписчиками.

Паттерн Наблюдатель предлагает хранить внутри объекта издателя список ссылок на объекты подписчиков, причём издатель не должен вести список подписки самостоятельно. Он предоставит методы, с помощью которых подписчики могли бы добавлять или убирать себя из списка.

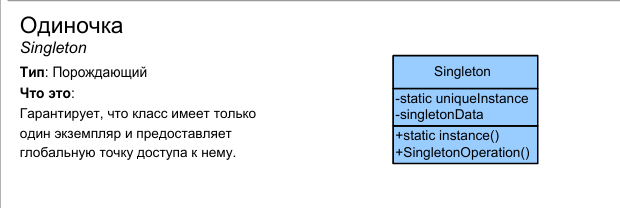
Теперь самое интересное. Когда в издателе будет происходить важное событие, он будет проходиться по списку подписчиков и оповещать их об этом, вызывая определённый метод объектов-подписчиков.

Издателю безразлично, какой класс будет иметь тот или иной подписчик, так как все они должны следовать общему интерфейсу и иметь единый метод оповещения.

Увидев, как складно всё работает, вы можете выделить общий интерфейс, описывающий методы подписки и отписки, и для всех издателей. После этого подписчики смогут работать с разными типами издателей, а также получать оповещения от них через один и тот же метод.

****

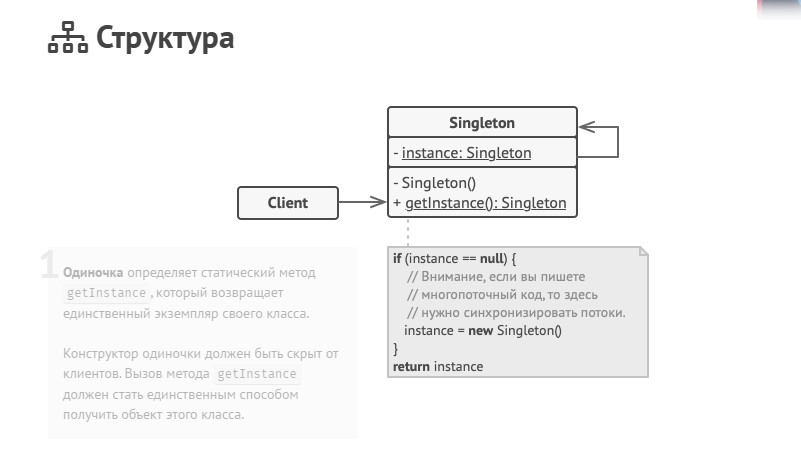
**26. Одиночка: назначение, использование**

****

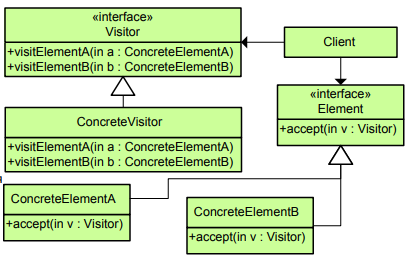
**Одиночка** — это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

Все реализации одиночки сводятся к тому, чтобы скрыть конструктор по умолчанию и создать публичный статический метод, который и будет контролировать жизненный цикл объекта-одиночки.

Если у вас есть доступ к классу одиночки, значит, будет доступ и к этому статическому методу. Из какой точки кода вы бы его ни вызвали, он всегда будет отдавать один и тот же объект.

****

**27. Посетитель: назначение, использование**



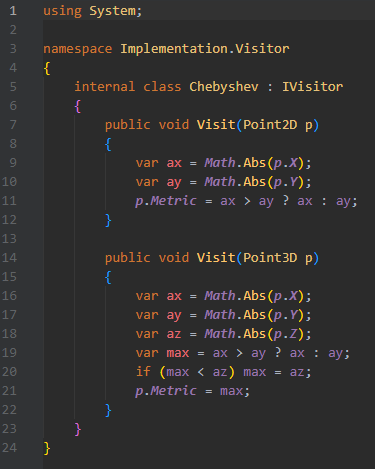
*Назначение:*

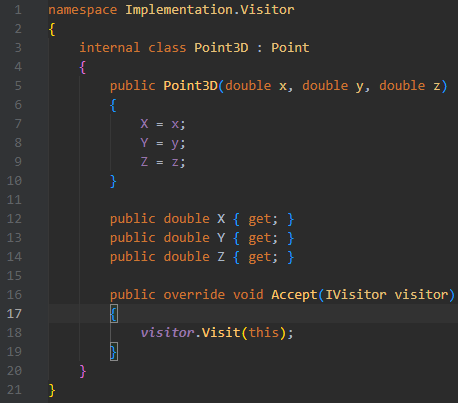
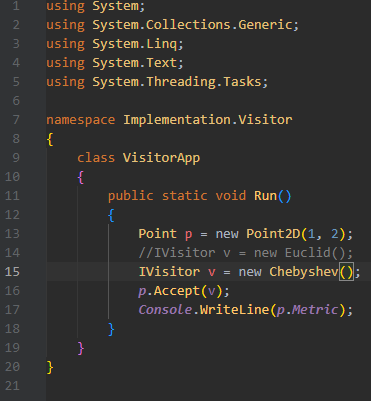
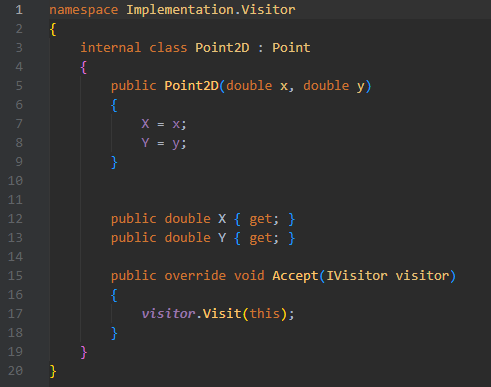
Позволяет определить новую операцию без изменения классов элементов, над которыми она выполняется

*Использование:*

Если есть вероятность изменения иерархии обслуживаемого класса, либо она будет нестабильной или открытый интерфейс достаточно эффективен для доступа шаблона, то его использование будет вредоносным.

Создается базовый класс Visitor с методами visit() для каждого подкласса родительского Element. Добавьте метод accept(visitor) в иерархию Element. Для каждой операции, которая должна выполняться для объектов Element, создайте производный от Visitor класс. Реализации метода visit() должны использовать открытый интерфейс класса Element. В результате: клиенты создают объекты Visitor и передают их каждому объекту Element, вызывая accept().

*Псевдокод*

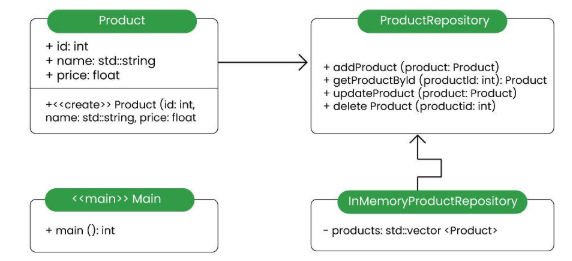


*Недостатки:*

Если есть вероятность изменения иерархии обслуживаемого класса, либо она будет нестабильной или открытый интерфейс достаточно эффективен для доступа шаблона, то его использование будет вредоносным.

затруднено добавление новых классов, поскольку нужно обновлять иерархию посетителя и его сыновей.

**28. Репозиторий: назначение, использование**

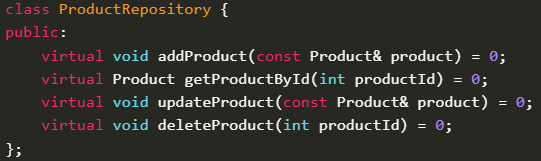
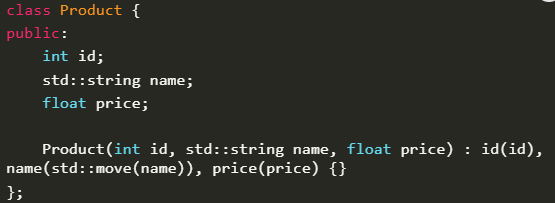
**

*Назначение:*

Разделение бизнес-логики от деталей реализации слоя доступа к данным.

*Использование:*

Создается интерфейс репозитория. Далее создаются реализации этого интерфейса. Потом используется для манипуляции с объектами



*Недостатки:*

*Cложность архитектуры:*

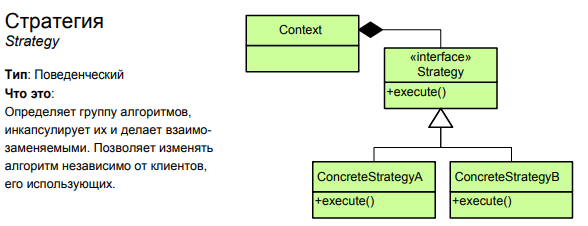
*Снижение производительности:*

*Ограниченная функциональность:*

*Трудности в реализации сложных сценариев:*

*Переусложнение для простых приложений:*

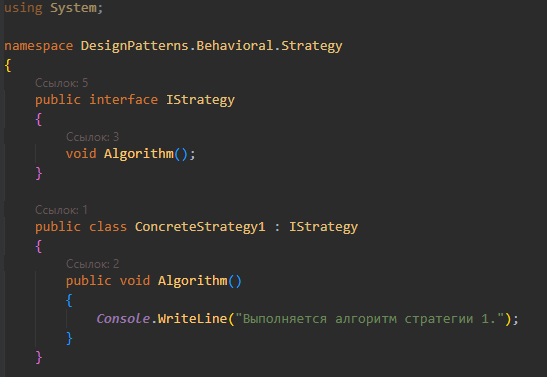
**29. Стратегия: назначение, использование**

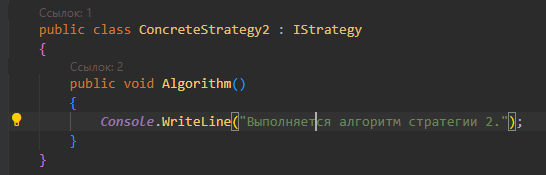
****

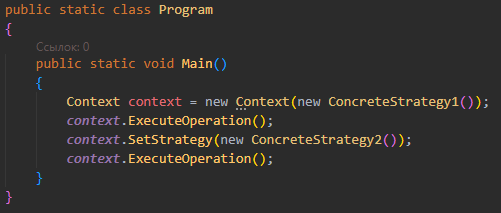
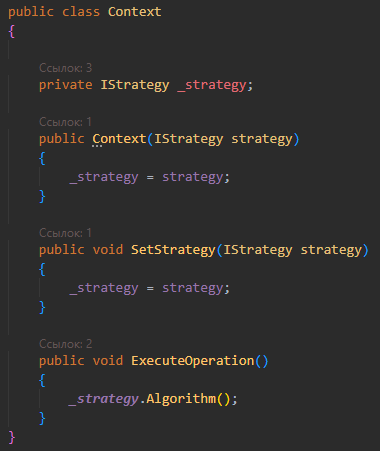
*Назначение*

Нужен для определения семейства алгоритмов, инкапсуляции каждого из них и обеспечения их взаимозаменяемости. Это позволяет выбирать алгоритм путём определения соответствующего класса. Шаблон Strategy позволяет менять выбранный алгоритм независимо от объектов-клиентов, которые его используют.

*Использование*

Класс, который использует алгоритм (Context), включает абстрактный класс (Strategy), обладающий абстрактным методом, определяющим способ вызова алгоритма. Каждый производный класс реализует один требуемый вариант алгоритма.





*Недостатки:*

Усложняет программу за счёт дополнительных классов.

Клиент должен знать, в чём состоит разница между стратегиями, чтобы выбрать подходящую.

**30. Строитель: назначение, использование**

Разделяет конструирование *сложного* объекта от его представления, позволяя создавать разные представления этого объекта с одним и тем же процессом конструирования. Позволяет создавать объект пошагово.

Использовать: Когда алгоритм создания сложного объекта должен быть независим от частей, из которых состоит объект, и того, как они конструируются ИЛИ когда процесс конструирования должен позволять создавать разные представления объекта.

**31. Фабричный метод: назначение, использование**

Фабричный метод — это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.

Использовать: Когда классу заранее неизвестно, объекты каких подклассов ему нужно создавать ИЛИ когда класс хочет, чтобы его подклассы специфицировали создаваемые объекты

# **Тема 3. UML, SOLID**

**1. Принципы GRASP (General Responsibility Assignment Software Patterns)**

**GRASP** — это набор принципов для распределения обязанностей между классами и объектами в ООП. Основные принципы:

* **Информационный эксперт (Information Expert)** – Назначать ответственность классу, который обладает максимумом информации для её выполнения.
* **Создатель (Creator)** – Класс A должен создавать объекты класса B, если:
  + A содержит B
  + A агрегирует B
  + A использует B
* **Контроллер (Controller)** – Вводить промежуточный класс между UI и бизнес-логикой для обработки системных событий.
* **Низкая связанность (Low Coupling)** – Минимизировать зависимости между классами.
* **Высокая связность (High Cohesion)** – Класс должен выполнять одну логическую задачу.
* **Полиморфизм (Polymorphism)** – Использовать наследование и интерфейсы для обработки вариаций поведения.
* **Чистая выдумка (Pure Fabrication)** – Создавать искусственные классы, если нет естественного кандидата на ответственность.
* **Посредник (Indirection)** – Вводить промежуточный объект для уменьшения связанности.
* **Устойчивость к изменениям (Protected Variations)** – Изолировать изменяющиеся части системы.

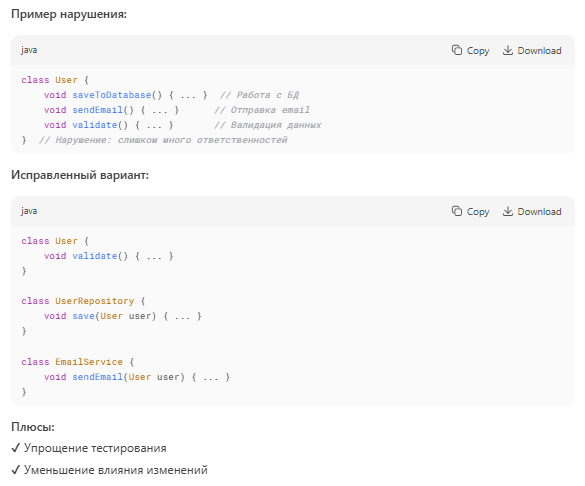
**2. Принцип единственной ответственности (Single Responsibility Principle, SRP)**

**Формулировка:**

*"Класс должен иметь только одну причину для изменения, то есть только одну ответственность."*

**Суть:**

* Каждый класс должен решать только одну задачу.
* Изменение в одной функциональности не должно затрагивать другие.



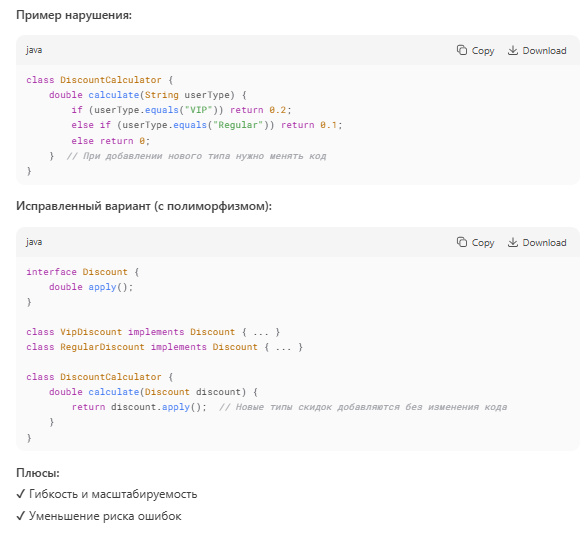
**3. Принцип открытости/закрытости (Open/Closed Principle, OCP)**

**Формулировка:**

*"Программные сущности должны быть открыты для расширения, но закрыты для модификации."*

**Суть:**

* Поведение класса можно расширять (через наследование, интерфейсы), но не изменять его исходный код.



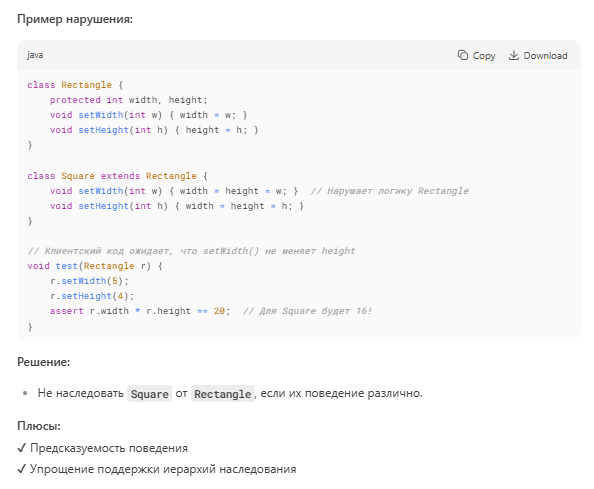
**4. Принцип подстановки Барбары Лисков (Liskov Substitution Principle, LSP)**

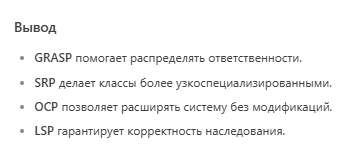
**Формулировка:**

*"Подклассы должны быть заменяемыми на свои базовые классы без нарушения работы программы."*

**Суть:**

* Наследник не должен ужесточать предусловия или ослаблять постусловия.
* Поведение подкласса не должно противоречить поведению родителя.





**5. Принцип разделения интерфейса (The Interface Segregation Principle)**

*Клиенты не должны зависеть от интерфейсов, которые они не используют.*

Суть принципа:

* Избегаем "жирных" интерфейсов – не следует заставлять классы реализовывать методы, которые им не нужны.
* Разделяем интерфейсы – если интерфейс содержит слишком много методов, его стоит разбить на несколько более мелких и специализированных.
* Уменьшаем зависимость клиентов – классы должны зависеть только от тех интерфейсов, которые они реально используют.



**6. Принцип инверсии зависимостей**

Официальная формулировка:

* Модули верхних уровней не должны зависеть от модулей нижних уровней. И те, и другие должны зависеть от абстракций.
* Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Иными словами:

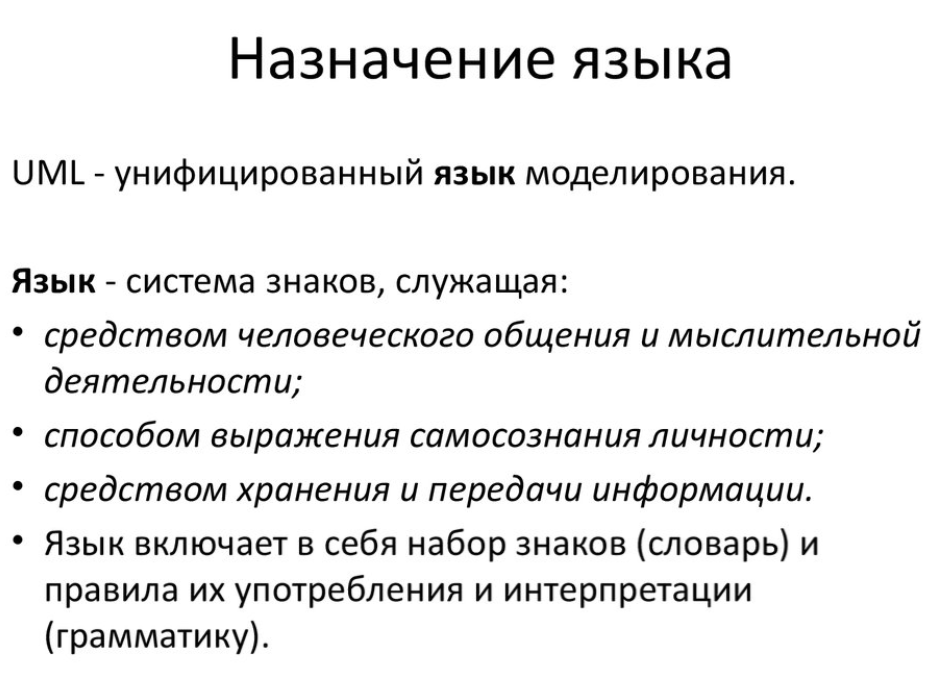
* Код должен зависеть не от конкретных реализаций (деталей), а от абстракций (например, от интерфейсов).
* Более "высокий" бизнес-уровень (логика) не знает, как устроена "низкая" реализация, ведь он работает через абстракцию.

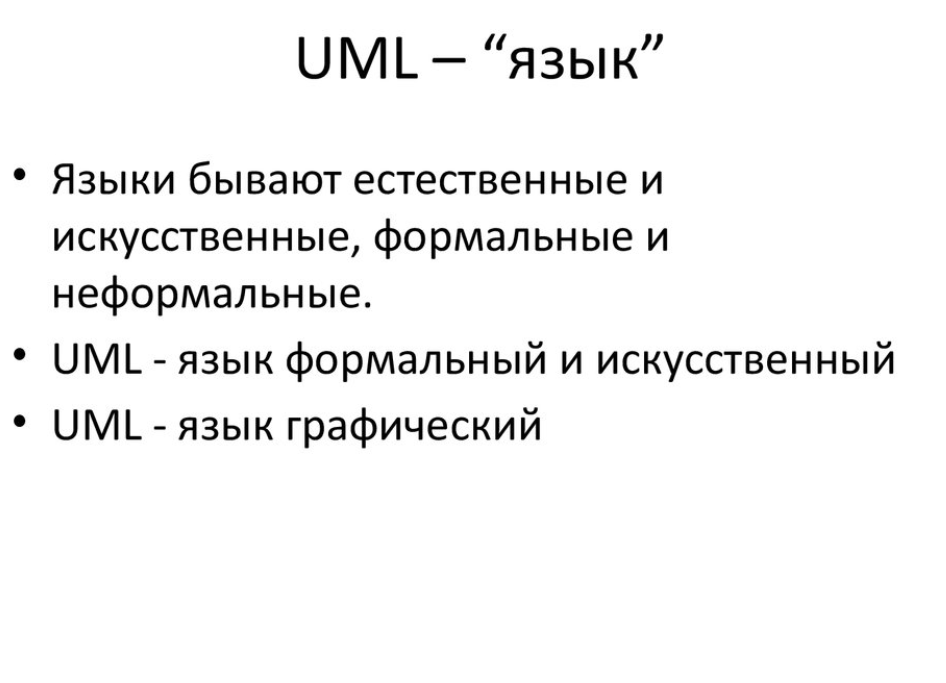
Т.е. если класс напрямую использует конкретную реализацию — жёсткая зависимость, сложно заменить часть системы.

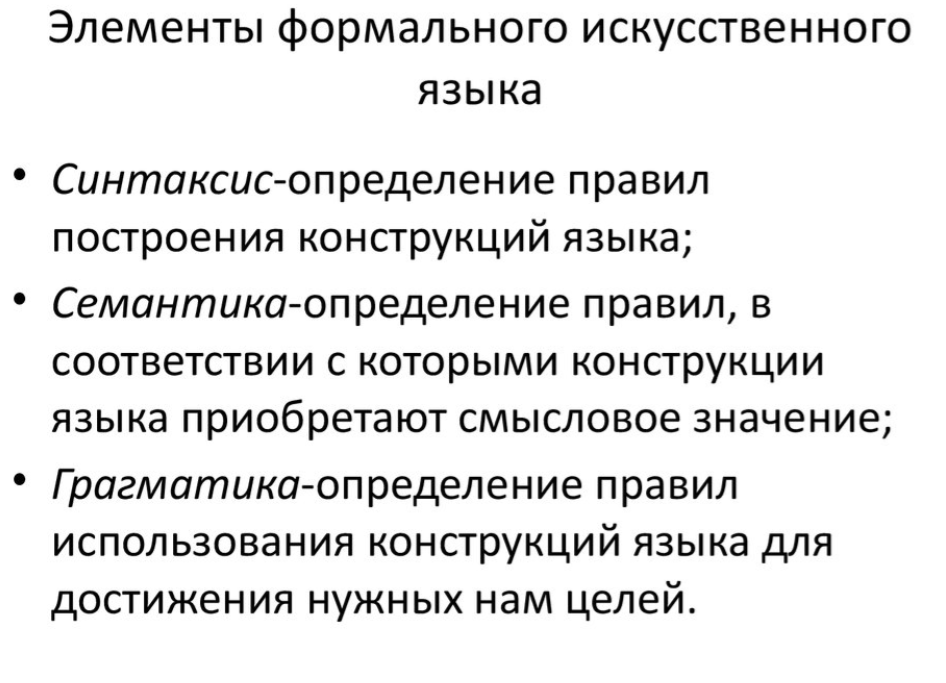
Вместо этого нужно, чтобы класс зависел от абстракции — легко заменить детали "под капотом", не переписывая логику.

**7. Назначение языка UML. UML – как язык. Элементы формального искусственного языка**

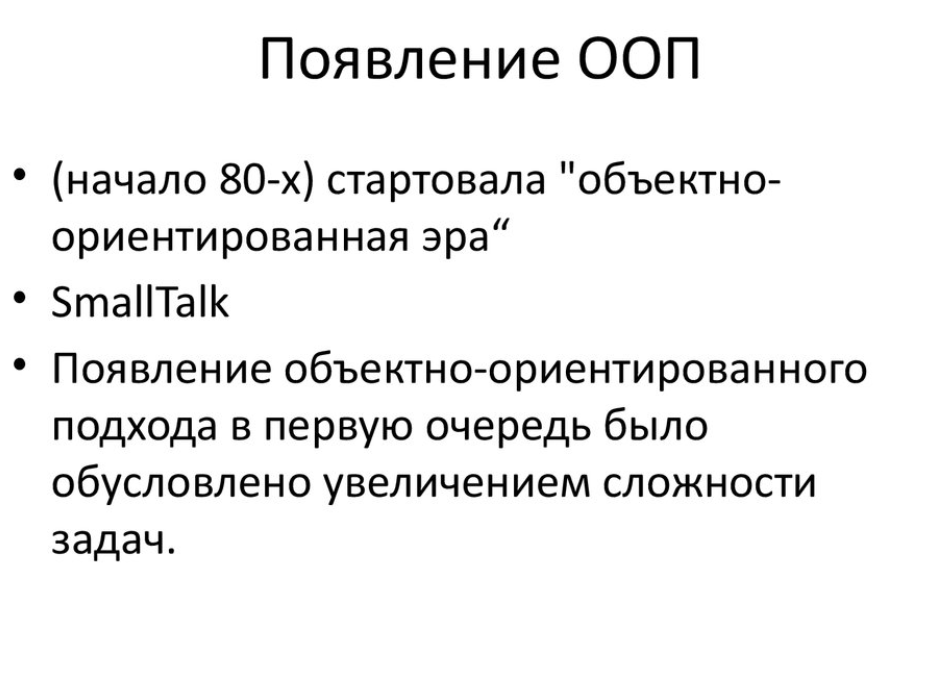
Unified Modeling Language — унифицированный язык моделирования

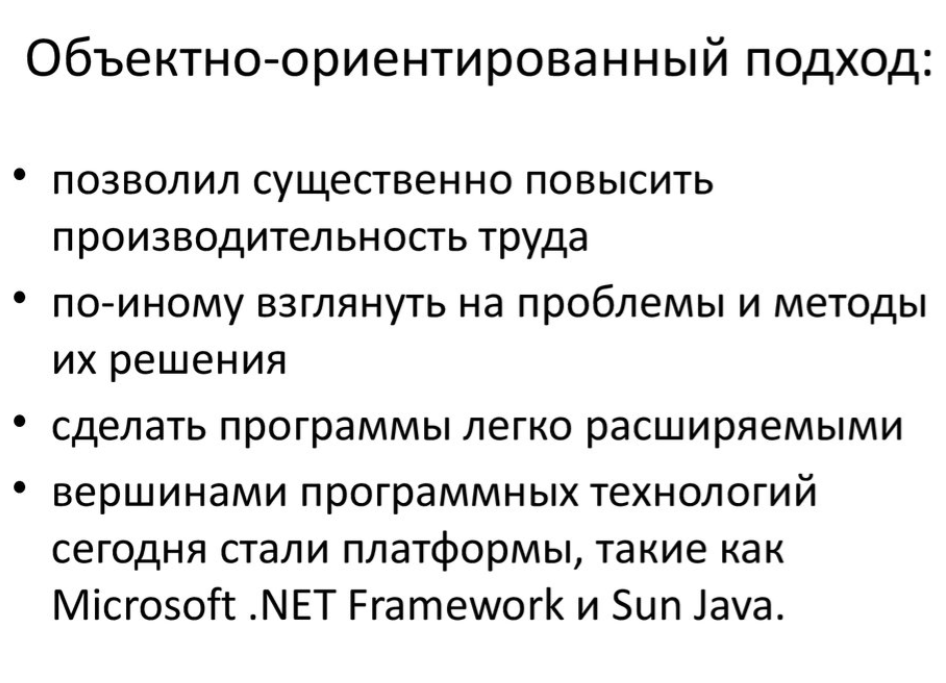


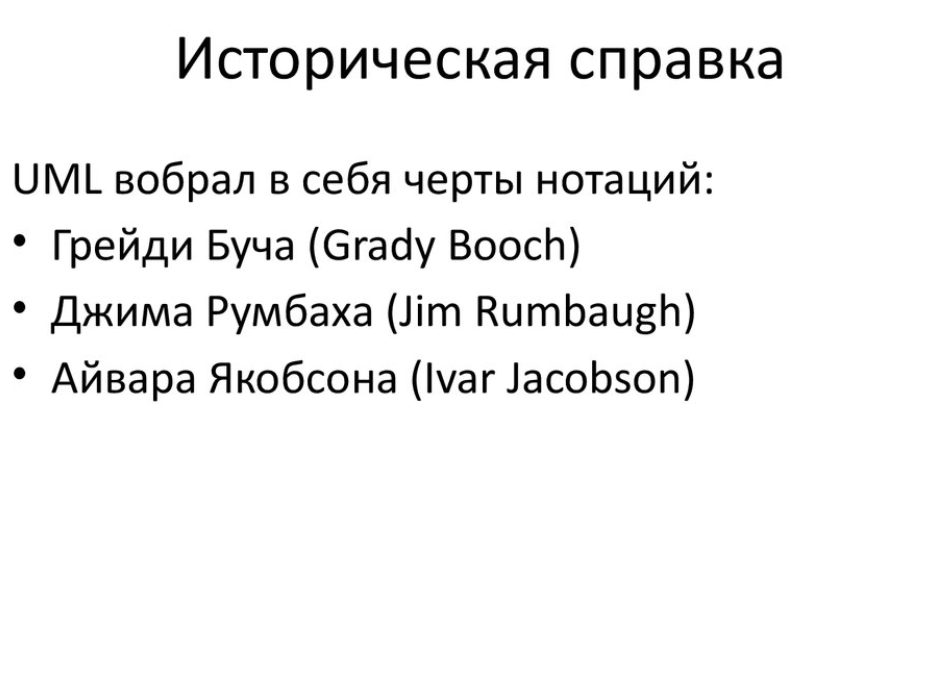


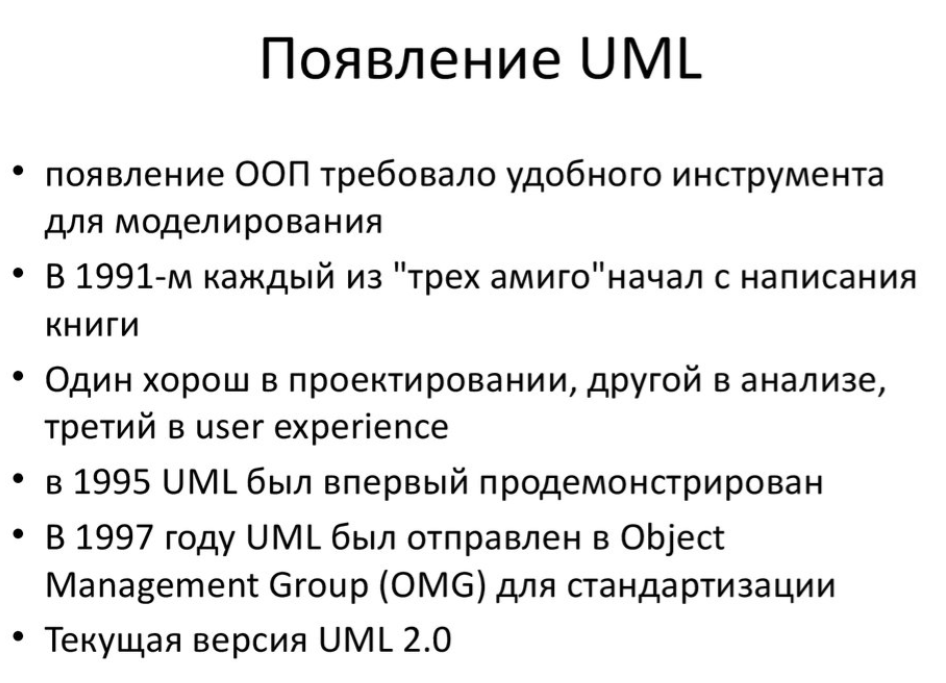


**8. Объектно-ориентированный подход к программированию. Появление UML, предпосылки.**









**9. Возможности UML по спецификации, визуализации, проектированию и документированию.**

UML (унифицированный язык моделирования) предоставляет следующие возможности:

Спецификация: UML позволяет точно и формально описывать модели приложений и систем. Это помогает снизить риск расхождений в толковании спецификаций.

Визуализация: графические средства UML позволяют визуализировать модели, что облегчает понимание и обмен информацией между участниками проекта.

Проектирование: UML не только описывает абстрактные модели, но и позволяет манипулировать артефактами, входящими в состав приложений, включая программный код. Это означает, что с помощью UML можно создавать модели, для которых возможна автоматическая генерация программного кода.

Документирование: модели UML могут храниться и использоваться в форме электронных документов или в виде твёрдой копии. Это обеспечивает возможность документирования моделей и их использования в различных форматах.

**10. Терминология и нотация, области применения UML.**

UML (Unified Modeling Language) — это графический язык моделирования, применяемый для визуализации, описания, проектирования и документирования программных систем.

Терминология и нотация

Термины в UML созданы так, чтобы избежать привязки к конкретным языкам программирования и существующим традициям. Это сделано для точной передачи смысла понятий, даже если термины могут показаться непривычными.

UML использует графическую нотацию, а не текстовый синтаксис. Основные элементы нотации:

Фигуры — двумерные замкнутые формы (например, прямоугольники, эллипсы), могут содержать другие элементы.

Линии — соединяют фигуры или значки; бывают сплошными и пунктирными; могут содержать стрелки и подписи.

Значки — простые изображения без внутреннего содержимого; не меняют форму и служат для обозначения.

Тексты — поясняющие надписи; различаются по начертанию (прямой, курсив, подчёркнутый), а не по шрифтам и цветам.

Нотация UML гибкая: важно не оформление, а понятность. Диаграммы должны быть читаемыми даже при чёрно-белой печати.

Области применения UML

UML применяется в разных сферах разработки ПО:

Анализ требований — с помощью диаграмм прецедентов и взаимодействия.

Проектирование архитектуры системы — диаграммы классов, компонентов, развертывания.

Документирование систем — создание моделей, понятных разработчикам и заказчикам.

Моделирование поведения — диаграммы состояний, активности, последовательностей.

Объектно-ориентированное проектирование — описание классов, связей, наследования.

Бизнес-моделирование — отображение процессов и взаимодействий на уровне бизнеса.

UML подходит как для ручного моделирования, так и для работы с CASE-средствами. Он универсален и поддерживает как ранние стадии разработки, так и детальную проработку архитектуры.

**11. Понятия: системы, подсистемы, модели, диаграммы. Диаграммы в проектирование ПО.**

Понятия:

Система — это описание некоторой части реального мира, например, при моделировании приложения рассматривается его программная реализация, аппаратура, на которой исполняются программы, и пользователи, взаимодействующие с приложением.

Подсистема — часть системы, которая может быть описана отдельно.

Модель — описание физической системы с определённой точки зрения, которое может отличаться степенью детальности, используемыми средствами и расстановкой акцентов.

В UML 1.x всего определено 9 канонических типов диаграмм. Ниже перечислены их названия, принятые в этой книги

• Диаграмма использования

• Диаграмма классов

• Диаграмма объектов

• Диаграмма состояний

• Диаграмма деятельности

• Диаграмма последовательности

• Диаграмма кооперации

• Диаграмма компонентов

• Диаграмма размещения

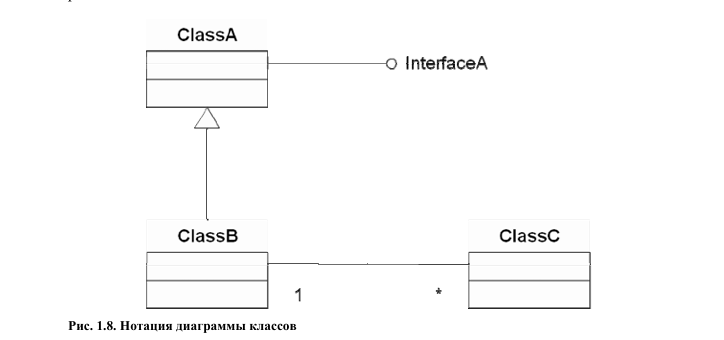


Диаграммы в проектировании ПО:

Диаграммы в UML используются для визуализации и моделирования различных аспектов системы. Они помогают понять структуру, поведение и взаимодействие компонентов системы. Примеры диаграмм:

Диаграмма использования — показывает функциональное назначение системы и взаимодействие между действующими лицами и вариантами использования.

Диаграмма классов — моделирует структуру объектов и связей между ними.



Диаграммы компонентов и размещения — моделируют структуру компонентов и вычислительных ресурсов.

Диаграммы взаимодействия — описывают взаимодействие объектов для выполнения задачи или достижения цели

**12. Диаграмма прецедентов (use case diagram). Основные понятия, назначение, нотация.**

Диаграмма прецедентов (диаграмма использования) — это наиболее общее представление функционального назначения системы, которое призвано ответить на вопрос: что делает система во внешнем мире?

Основные понятия:

Варианты использования — описывают различные сценарии взаимодействия с системой.

Действующие лица — внешние сущности, которые взаимодействуют с системой.

Назначение: Диаграмма использования показывает функциональное назначение системы и взаимодействие между действующими лицами и вариантами использования. Она является основным и, по мнению автора документа, единственным средством моделирования использования в UML.

Нотация: На диаграмме использования применяются следующие типы сущностей:

действующие лица;

варианты использования;

примечания;

пакеты.

Между этими сущностями устанавливаются следующие типы отношений:

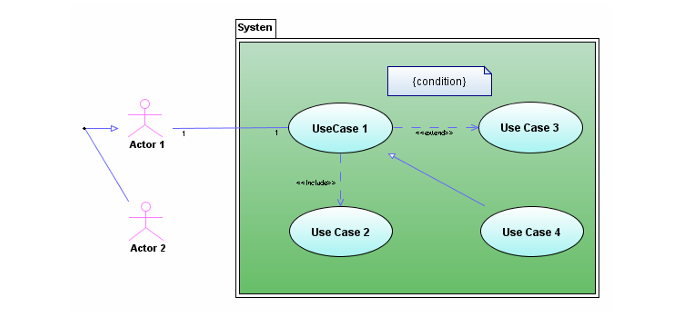
ассоциация между действующим лицом и вариантом использования;

обобщение между действующими лицами;

обобщение между вариантами использования;

зависимости между вариантами использования;

зависимости между пакетами.



**13. Диаграмма прецедентов (use case diagram). Сценарии. Спецификация функциональных требований.**

После формирования диаграммы прецедентов, следующим этапом проработки задачи является составление сценариев использования. Как правило, каждый из прецедентов ложится в основу определенного сценария.

Спецификация функциональных требований

На основе сценариев формируется документ, который формализует функциональные требования (что система должна делать) и нефункциональные (как она это делает).

Ключевые компоненты:

1. Модель прецедентов (Use Case Model)
   1. Содержит диаграммы и сценарии. Описывает поведение системы через взаимодействие с акторами.
   2. Пример: Покупатель → Оформить заказ → Оплатить товар.
2. Дополнительная спецификация (Supplementary Specification)
   1. Фиксирует нефункциональные требования:
      1. Производительность (например, Формирование отчета ≤ 3 сек).
      2. Безопасность, лицензирование, интеграции.
3. Словарь терминов (Glossary)
   1. Определяет ключевые понятия предметной области (например, Заказ = набор товаров + адрес доставки).
4. Приоритизация требований
   1. Метод MoSCoW:
      1. Must: Без чего система неработоспособна (например, Авторизация пользователя).
      2. Should: Важно, но не критично (например, Уведомления о скидках).
      3. Could: Желательные функции (например, Интеграция с соцсетями).
      4. Would: Отложенные задачи (например, Мультиязычный интерфейс).

**14. Диаграмма классов (class diagram). Основные понятия, назначение, нотация.**

**Понятия**

Класс (class) - категория вещей, которые имеют общие атрибуты и операции.

классы - это строительные блоки любой объектно-ориентированной системы

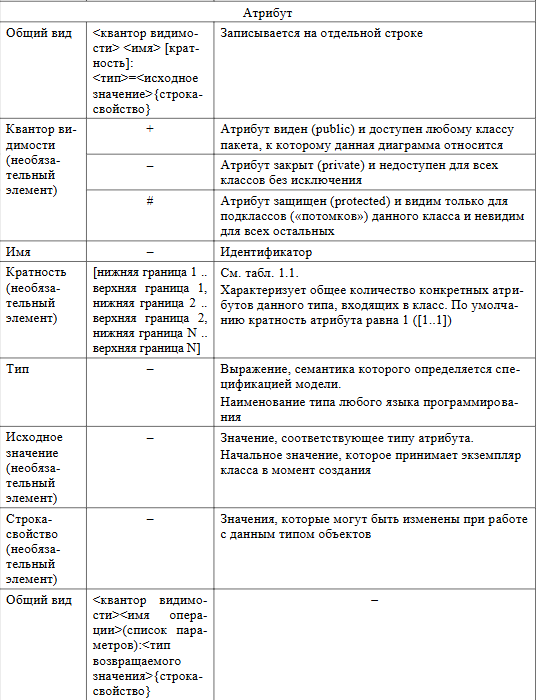
Диаграмма классов - это набор статических, декларативных элементов модели.

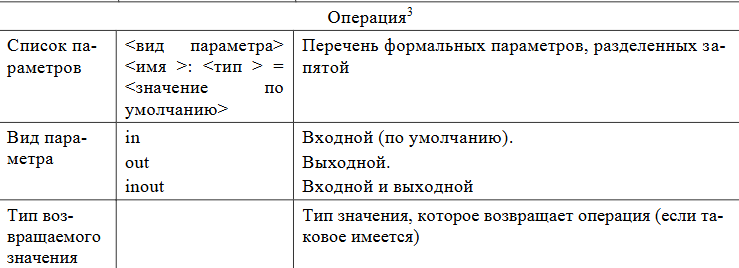
Класс на диаграмме изображается в виде прямоугольника, разделенного горизонтальными линиями на три части

* название класса. Имя класса состоит из одного, максимум двух слов.
* перечень атрибутов класса, которые характеризуют тот или иной объект этого класса в модели предметной области.
* перечень операций, отражающих его поведение в модели предметной области

**Назначение диаграммы классов**

1. Визуализация структуры системы:
   1. Показывает классы, их атрибуты, методы и отношения между ними.
2. Моделирование статической структуры:
   1. Описывает, из каких компонентов состоит система на уровне кода, без учета времени и последовательности действий.
3. Документирование:
   1. Служит "чертежом" для разработчиков, архитекторов и аналитиков.
4. База для других диаграмм:
   1. Используется как основа для диаграмм последовательности, состояний и др.







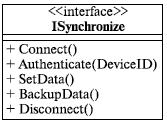
**15. Диаграмма классов (class diagram). Модификаторы видимости, Интерфейс, Обобщение. Инкапсуляция, наследование и полиморфизм.**

**Модификаторы видимости**

* + public - открытый доступ
* - private - только из операций того же класса
* # protected - только из операций этого же класса и классов, создаваемых на его основе
* ~ internal

**Интерфейс**

* интерфейс - это логическая группа открытых (public) операций объекта
* Интерфейс всегда реализуется некоторым классом, который в таком случае называют классом, поддерживающим интерфейс
* один и тот же объект может иметь несколько интерфейсов

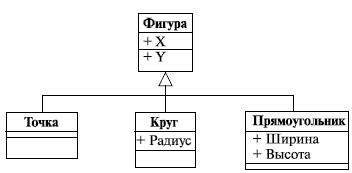


**Обобщение**

Обобщение - это отношение между более общей сущностью, называемой суперклассом, и ее конкретным воплощением, называемым подклассом.

*Эффективное моделирование наследование*

* Найдите атрибуты, операции и обязанности, общие для двух или более классов из данной совокупности. Это позволит избежать ненужного дублирования структуры и функциональности объектов.
* Вынесите эти элементы в некоторый общий суперкласс, а если такого не существует, то создайте новый класс.
* Отметьте в модели, что подклассы наследуются от суперкласса, установив между ними отношение обобщения



**Инкапсуляция**

Инкапсуляция- Сокрытие от пользователя внутреннего устройства объектов называется.

Инкапсуляция – это защита отдельных элементов объекта, не затрагивающих существенных характеристик его как целого.

**Наследование**

Наследование в объектно-ориентированном программировании — это концепция, согласно которой одни классы, называемые родительскими, могут лежать в основе других — дочерних. При этом, дочерние классы перенимают свойства и поведение своего родителя.

Наследование (inheritance) – это отношение типа «общее-частное». Позволяет определить такое отношение между классами, когда один класс обладает поведением и структурой ряда других классов. При создании производного класса на основе базового (одного или нескольких) возникает иерархия наследования. Реализация принципов наследования является ключевой предпосылкой возможности повторного использования кода, поскольку это основной инструмент достижения полиморфизма.

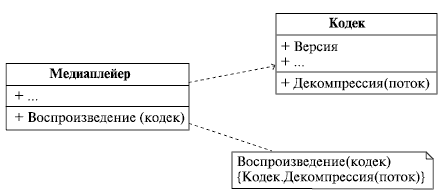
**Полиморфизм**

Полиморфизм — это концепция теории типов, согласно которому одно и то же имя может обозначать экземпляры разных классов, связанных с общим суперклассом. Таким образом, любой объект с этим именем может по-разному выполнять общий набор операций. Используя полиморфизм, одну и ту же операцию можно по-разному реализовывать в классах, образующих иерархию. В результате подкласс может расширять возможности суперкласса или замещать базовые операции

**16. Диаграмма классов (class diagram). Отношения между классами, Агрегация и композиция**

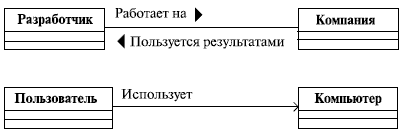
Отношение зависимости означает, что реализация одного класса зависит от спецификации операций другого класса.

* **Зависимость**



* **Ассоциация- просто связь между объектами, по которой можно между ними перемещаться.**

Ассоциация выражает отношение между несколькими равноправными объектами и может иметь направление, роли и кратность, а также изображаться в виде класса ассоциации.



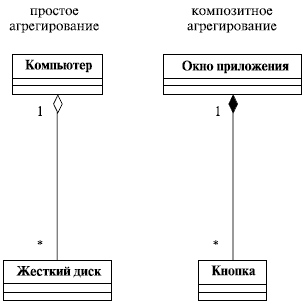
Ассоциация может быть представлена в виде класса.



**Агрегации и композиции**

Композиция и агрегация используются, если между объектами существуют отношения типа "часть-целое", причем композиция предполагает, что части не могут существовать отдельно от целого.

* Под агрегацией понимается более сложное отношение между классами, например, связь типа «часть-целое»
* Под композитным же агрегированием понимается ситуация, когда целое владеет своими частями и их время жизни соответствует времени жизни целого



**17. Диаграмма объектов (Object Diagram)**

**Основные понятия:**

Диаграмма объектов показывает **конкретные экземпляры** классов (объекты) и **связи между ними** в определённый момент времени.

Является **статическим снимком** системы на определённой стадии выполнения.

**Назначение:**

Отображение **конкретной конфигурации объектов**.

Используется для **отладки**, **анализов сценариев**, примеров и тестирования структуры.

Демонстрирует, как классы на диаграмме классов проявляются в реальных объектах.

**Нотация:**

Прямоугольники, представляющие объекты, с указанием имени в формате:

КопироватьРедактировать

имяОбъекта : ИмяКласса

Связи между объектами отображаются линиями (ассоциациями).

Можно показывать значения атрибутов.

**18. Диаграмма последовательностей (Sequence Diagram)**

**Основные понятия:**

Диаграмма показывает **взаимодействие объектов во времени**, акцент на **последовательность сообщений**.

Горизонтальная ось — участники (объекты), вертикальная — время.

**Назначение:**

Моделирование **сценариев использования**.

Анализ **логики взаимодействия** между объектами.

Планирование и документирование **поведения системы**.

**Нотация:**

**Объекты** сверху, по горизонтали (подписываются объект : Класс).

**Жизненная линия** (lifeline) — пунктирная вертикальная линия от объекта.

**Сообщения** — горизонтальные стрелки (с синхронным или асинхронным вызовом).

**Активность** — прямоугольник на линии жизни (период выполнения метода).

Петли, альтернативы и условия можно оформлять через фреймы (alt, loop, opt и др.).

**19. Диаграмма взаимодействия (кооперации, Collaboration Diagram)**

(также может называться Communication Diagram)

**Основные понятия:**

Показывает **структуру** объектов и **поток сообщений** между ними.

Альтернатива диаграмме последовательностей, с акцентом на **пространственные отношения** и взаимодействие.

**Назначение:**

Анализ **взаимных связей объектов** и порядка передачи сообщений.

Используется для **моделирования поведения** и **сценариев**.

**Нотация:**

Объекты представлены прямоугольниками.

Связи между ними — линии.

Сообщения помечаются на линиях с номерами порядка выполнения (1:, 2:, 2.1: и т.п.).

**Композитная структурная диаграмма (Composite Structure Diagram):**

Показывает **внутреннее устройство** класса/компонента.

Включает **части** (parts), **порты**, **связи**, **интерфейсы**.

Используется для описания **композиции** и **взаимодействий компонентов** в более абстрактной форме.

**20. Диаграмма состояний (Statechart Diagram)**

**Основные понятия:**

Показывает **изменения состояний** объекта в ответ на **события**.

Используется для описания **поведения объекта или системы со сложной логикой**.

**Назначение:**

Моделирование **жизненного цикла** объекта.

Уточнение, как объект **реагирует на события** и **переходит между состояниями**.

**Нотация:**

**Состояние** — прямоугольник с закруглёнными углами, внутри — имя и (по желанию) действия (entry, do, exit).

**Начальное состояние** — закрашенный кружок.

**Конечное состояние** — кружок с обводкой и точкой внутри.

**Переходы** — стрелки с указанием **события** и/или **условия/действия** (например, event [guard] / action).

