1. **Структурное программирование: нисходящая разработка, использование базовых логических структур, сквозной структурный контроль.**

ОТВЕТ: Структурное программирование: нисходящая разработка, использование базовых логических структур, сквозной структурный контроль.

Дейкстра, Милдс выделили три идеи структурного программирования:

1. нисходящая разработка
2. использование базовых логических структур
3. сквозной структурный контроль

Нисходящая разработка

Этапы создание программного продукта:

1. Анализ. (Оцениваем задачу, переработка ТЗ)
2. проектирование
3. кодирование
4. тестирование

2-4 используют нисходящий подход.

Используются алгоритмы декомпозиции – разбиение задачи на подзадачи, выделенные подзадачи разбиваются дальше на подзадачи, формируется иерархическая структура (данные нисходящие, логика восходящая, разработка нисходящая).

Правила структурного программирования:

- Данные на низком уровне, на высшем логика.
- Для каждой полученной подзадачи создаем отладочный модуль. Готовятся тестирующие пакеты (до этапа кодирования). Принцип полного недоверия к данным.
- Возврат результата наверх и анализ последующего результата там.
- Явная передача данных через список параметров (не более 3х).
- Функция может возвращать не более одного параметра.
- Не более 7 подзадач у задачи.
- Глубина вложенности конструкций - не больше трёх.
- Иерархия уровня абстракции должна соответствовать иерархии данных [Нельзя работать с полями полей структур].
- Чем больше уровней абстракции, тем лучше.

Принципы работы с кодом:

1. Сегментирование (функция разбивается на логические куски).
2. Пошаговая реализация.
3. Вложенные конструкции (глубина не более 3х).

Заглушка - то, что должна выдавать функция при данных входных данных.

Блок, функция, файл – уровни абстракции. Ограничения вложенности – 3 (глубина вложенности), если больше то выделить подфункции.

Базовые логические структуры

Майер: Любой алгоритм можно реализовать с помощью трех логических структур:

- следование,
- развилка (ветвление)
- повторение

Развилка (ветвление): выбор между двумя альтернативами, множественный выбор `switch` – ветви имеют const выражения.

Повторение: `while`, `until`, `for`, безусловный цикл `loop`.

Принципы структурного программирования:

- Выход из цикла должен быть один.
- Не использовать оператора безусловного перехода `goto`.
- Любая программа строится из трёх базовых управляющих конструкций: последовательность, ветвление, цикл.
- В программе базовые управляющие конструкции могут быть вложены друг в друга произвольным образом.
- Повторяющиеся фрагменты программы можно оформить в виде подпрограмм (процедур и функций).
- Все перечисленные конструкции должны иметь один вход и один выход.

Сквозной структурный контроль

Организация контрольных сессий. На контрольной сессии никогда не присутствует начальство (руководство), иногда приглашаются умные люди со стороны. Количество замечаний не влияет на программиста. Задачи контрольной сессии – выявить недостатки на ранних стадиях. Готовят плакаты с алгоритмами и архитектурными решениями.

К оглавлению

2. Преимущества и недостатки структурного программирования. Идеи Энтони Хоара.

ОТВЕТ: Преимущества и недостатки структурного Технология структурного программирования

Преимущества:

1. Логические ошибки исправляются на ранних стадиях разработки ПО.
2. При таком подходе почти нет "кода в корзину".
3. Начиная с самых ранних стадий идет взаимодействие с заказчиком.
4. Можно объединять этапы кодирования, проектирования и тестирования (вести параллельно).
5. Комплексная отладка - тесты можно писать до этапа проектирования на основе ТЗ. Также отладка облегчается за счёт того, что можно ставить заглушки (нисходящая разработка это позволяет).
6. Удобное распределение работы между программистами.
7. Из-за иерархической структуры (многоуровневой абстракции) возникают естественные контрольные точки за наблюдением за проектом - легко контролировать процесс.
8. Локализация ошибок. (много уровней абстракции, легко выявить место)
9. Вероятность невыполнения проекта (отказ) сводится к нулю.
10. Повторное использование кода (выделяются библиотеки).
11. Плавное распределение ресурсов (нагрузки) при разработке программного продукта. Нет аврала в конце проекта.
12. Стандартизация - использование базовых логических структур

На начальном этапе используется иерархический подход (на этапе распределения ролей), а потом операционный(разработка).

Иерархический - порядок программирования и тестирования модулей определяется их расположением в схеме иерархии

Операционный - модули разрабатываются в порядке их выполнения при запуске готовой программы.

Недостатки:

1. Сложность модификации: изменение уже написанного кода понижает его надёжность (плюс трата времени на разбор чужого кода).
2. Добавление нового функционала доставляет много проблем
3. Исключительные ситуации обрабатываются вперемешку с логикой задачи - это приводит к большому количеству проверок и необходимости "протаскивать" ошибку через весь код до того места, где её можно будет обработать
4. Может возникать дублирование кода, от которого сложно избавиться
5. Сложно распределять коды ошибок между программистами, если их несколько.

Возникают моменты, когда легче написать свою программу с нуля.

Итого: писать код - великолепно, модифицировать - проблемно.

> Энтони Хоар внес значительный вклад в разработку концепций структурного программирования, особенно с его работой над созданием и популяризацией структурированного программирования, которое предлагает использование иерархически организованных блоков кода и контрольных структур, таких как последовательности, ветвления и циклы. Эти принципы помогают упростить понимание, тестирование и обслуживание программного обеспечения, что приводит к более надежным и эффективным программам.

Одной из основных его идей в контексте структурного программирования была мысль о том, что программы должны быть написаны с использованием блоков, которые можно тестировать независимо друг от друга, что облегчает отладку и обеспечивает более высокую надежность программного продукта. Это подход к разработке, который исключает использование операторов goto, что способствует созданию более структурированного и легко читаемого кода.

Также Хоар активно продвигал идею модульности, подчеркивая важность разделения программы на независимые модули, каждый из которых выполняет свою уникальную функцию. Это не только упрощает разработку и тестирование каждого модуля в отдельности, но и способствует повторному использованию кода.

Важно отметить, что подходы Хоара к структурному программированию положили начало практикам, которые сейчас являются стандартными в разработке программного обеспечения, включая аспекты безопасности и надежности кода.

3. Преимущества и недостатки объектно-ориентированного программирования.

ОТВЕТ:Технология объектно-ориентированного программирования:

Преимущества:

1. Возможность легкой модификации (при грамотном анализе и проектировании)
2. Возможность отката при наличии версий
3. Более легкая расширяемость (масштабируемость)
4. «Более естественная» декомпозиция программного обеспечения, которая существенно облегчает его разработку.
5. Сокращение количества межмодульных вызовов и уменьшение объемов информации, передаваемой между модулями.
6. Увеличивается показатель повторного использования кода.

Недостатки:

1. Резко увеличивается время на анализ и проектирование систем.
2. Этапы разработки не идут одновременно: надо сначала построить объектную модель, а потом её кодировать.
3. Высокий порог входа. В структурном программировании кто угодно может просто сесть и начать писать код, а тут обязательно надо заниматься анализом.
4. Увеличение размера кода ⇒ времени написания программы.
5. Увеличение времени выполнения программы.
6. Неэффективно с точки зрения памяти ⇒ нужно больше памяти.
7. Проблема с ресурсами: сложнее выявлять утечки памяти, т.к. программа разбита на куски.

В современных ЯП и в C++11 и выше проблема решена на уровне языка.

1. Сложность распределения работ на начальном этапе.
2. Из-за рекурсивного дизайна возникает мертвый код - тот, который написан, но не используется.

В современных ЯП проблема решена за счёт совмещения выполнения и компиляции программы, т.е. компилируется только то, что используется.

Не все недостатки технологии ООП перекрываются возможностью лёгкого изменения программы.

К оглавлению

4. Основные понятия ООП: инкапсуляция, наследование, полиморфизм. Понятие объекта. Категории объектов. Отношения между объектами. Понятие класса. Отношения между классами. Понятие домена.

ОТВЕТ: **Основные понятия ООП: инкапсуляция, наследование, полиморфизм. Понятие объекта. Категории объектов. Отношения между объектами. Понятие класса. Отношения между классами. Понятие домена.**

Есть данное. А давайте выделим, что мы можем сделать с данным. Выделили. Теперь изменение того, что мы можем сделать с данным не влияет на код - абстрагирование от данных и работа с набором действий.

Инкапсуляция

Инкапсуляция - объединение данных и действий над данными

«Инкапсуляция — это способность объектов скрывать часть своего состояния и поведения от других объектов, предоставляя внешнему миру только определённый интерфейс взаимодействия с собой.» [из книги: Александр Швец. «Погружение в Паттерны Проектирования»]

Наследование

Наследование - подмена одного на другое

«Наследование — это возможность создание новых классов на основе существующих. Главная польза от наследования — повторное использование существующего кода. » [из книги: Александр Швец. «Погружение в Паттерны Проектирования»]

Полиморфизм

Полиморфизм - безразличие к тому, что представляет функционал

«Полиморфизм — это способность программы выбирать различные реализации, при вызове операций с одним и тем же названием. С другой стороны, полиморфизм — это способность объектов притворяться чем-то другим.» [из книги: Александр Швец. «Погружение в Паттерны Проектирования»]

Понятие объекта

Хоар говорил, что весь мир можно представить как набор объектов: их обозначение, описание и манипуляции. Перенесем эту идею в программу.

Объект - конкретная реализация какого-то абстрактного понятия, обладающий характеристикой состояния, поведения и индивидуальности.

Состояние - одна из возможных стадий жизненного цикла существования объекта.

Поведение - описание объекта в терминах его состояния (то, как он изменяет состояние).

Индивидуальность - сущность, присущая каждому объекту, отличающая его от других объектов.

Модель Мура (модель состояний) включает в себя множество**:**

- состояний
- событий (которые приводят к изменению состояния)
- правил перехода
- действий (перевести объект из одного состояния в другое)

Категории объектов:

- реальные (абстракции фактического существования предметов в реальном мире, проектирование всегда начинается с реальных объектов)
- роли (абстракции цели, назначения: человека, организации и т.д.)
- инцидент (абстракция чего-либо случившегося: выборы, скачок напряжения)
- спецификации (для представления правил, стандартов - ПДД, расписание занятий и т.д.)
- Взаимодействия (получаемые в результате взаимоотношения между объектами: взятка, перекресток и т.д.)

Один и тот же объект физически может выполнять несколько ролей. В программе такого быть не должно.

Отношения между объектами:

Использования (старшинства)

Любой объект принимает одно из трех состояний:

1. Воздействия - активные объекты, они воздействуют на другие объекты, но сами не подвержены воздействию со стороны других
2. Исполнители - пассивные объекты, они подвержены воздействию со стороны других, но сами ни на кого не воздействуют
3. Посредники - и то, и то (они преобладают)

Извне мы должны работать с большИм количеством объектов - это плохо, с одной стороны, но это дает нам более гибкую схему: извне управляем бОльшим количеством связей объектов.

Включения

Более жесткая схема, но она упрощает взаимодействие: появляется оболочка, нам не нужно управлять большим количеством мелких объектов, мы абстрагируемся от их внутреннего взаимодействия (пример: лектор общается с потоком)

Класс

Класс - абстракция множества предметов, имеющих одни и те же характеристики и одинаковое поведение.

Отношения между классами:

1. Наследование.
2. Использование.
3. Включение (один класс содержит другой).
4. Метаклассы - устаревшее (используются для создания других классов).

Домен

Домен - отдельный реальный, гипотетический или абстрактный мир, населённый отчётливым набором объектов, которые ведут себя в соответствии с определенными правилами и линиями поведения.

Главная идея разбиения системы на домены состоит в том, чтобы можно было максимально безболезненно подменять эти домены, а также делегировать их разработку разным людям/отделам.

Какой-либо класс мы определяем только в одном домене. Наличие этого класса требует наличия других классов в этом же домене, но не требует наличия классов в другом домене. Например, есть домен *задача*, а *интерфейсный* домен мы можем полностью менять.

К оглавлению

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

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

ООП направлено на то, чтобы в дальнейшем легко модифицировать программу.

Этапы объектно-ориентированного проектирования:

1. Этап анализа - построение модели задачи и начальной объектно-ориентированной модели (выделение ключевых абстракций). Этап анализа проходит вместе с заказчиком. Анализ строится на основе физического мира.

   Требования к модели:

   - Полная
   - Понятная всем заинтересованным лицам
2. Этап проектирования. На современном уровне как правильно автоматизирован. Создаем проектные документы на основе документов, которые мы получили на этапе анализа.
3. Этап эволюции. Эволюция = кодирование + тестирование + рекурсивный дизайн (от простого состояния системы развиваем до сложного).

   Эволюция - развитие проекта в рамках разработки продукта. До сдачи продукта.

4. Этап модификации - изменения, вносимые уже после сдачи продукта заказчику. Этапы эволюции и модификации выполняют разные люди.

Преимущества выделения этапа эволюции:

- Предоставляется обширная обратная связь

- Получаются разные версии системы - дает механизм отката и можно выпускать альтернативные версии продукта.
- Можно рассматривать каждую версию для демонстрации и обсуждения с заказчиком. Уже на начальных этапах есть результаты.
- Интерфейс разрабатывается отдельно от модели.

Какие изменения могут быть реализованы при эволюции:

- Добавление новых классов (безболезненная операция).
- Изменение реализации класса (безболезненная подмена).
- Изменение представления класса (более сложная операция).
- Реорганизация структуры класса (ещё более тяжёлая процедура).

Некоторые изменения можно реализовать за счёт применения паттернов проектирования.

Мы ни в коем случае не должны изменять интерфейс базового класса.

Рабочие продукты объектно-ориентированного анализа.

Продукт анализа - рабочие документы. Они лягут в основу проектных документов (на этапе проектирования).

Какие документы создаются при анализе:

1. Для всей программы:

   1.1. Схема доменов

   1.2. Проектная матрица (контролируем, какие этапы анализа нами пройдены)

2. Для каждого домена:

   Если в домене больше 30-50 классов, разбиваем домен на подсистемы. Разбиваем граф связности по принципу минимума связей - внутри подсистемы связей много, между подсистемами связей мало.

   2.1. Модель связей подсистем

   2.2. Модель доступа к подсистемам

   2.3. Модель взаимодействия подсистем

3. Для каждой подсистемы:

   3.1. Информационная модель (диаграмма сущность-связь)

   3.2. Описание классов и их атрибутов (членов данных)

   3.3. Описание связей

   3.4. Модель взаимодействия объектов (МВО)

   3.5. Список событий, происходящих в подсистеме

   3.6. Модель доступа к объектам (МДО)

4. Для каждого класса:

   4.1. Модель состояний (диаграмма переходов состояний - ДПС)

   4.2. Таблица процессов состояний (ТПС)

   4.3. Алгоритм действий состояний

5. Для каждого действия:

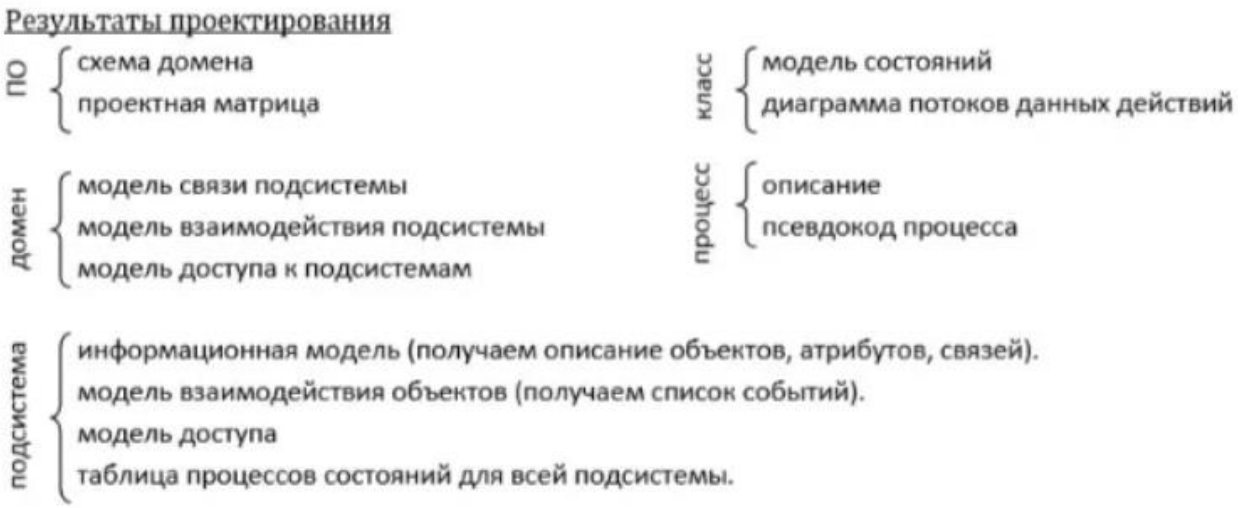
   5.1 Диаграмма потоков данных действий (ДПДД)

   5.2. Описание процессов

С чего начинать разработку проекта: разбили задачу на домены, рассматриваем прикладной домен (наш). Разработка прикладного домена начинается с информационного моделирования.

Информационное моделирование всегда начинаем с физических (реальных) объектов. Смотрим, какие объекты существуют. Пытаемся эти объекты сгруппировать по принципу одних и тех же характеристик (выделяем общее). Выделяем, чем характеризуется объект.

Выделяем атрибуты объектов.



6. Концепции информационного моделирования. Понятие атрибута. Типы атрибутов. Правила атрибутов. Понятие связи. Типы связей. Формализация связей. Композиция связей. Подтипы и супертипы. Диаграмма сущность-связь.

Ответ:Концепции информационного моделирования. Понятие атрибута. Типы атрибутов. Правила атрибутов. Понятие связи. Типы связей. Формализация связей. Композиция связей. Подтипы и супертипы. Диаграмма сущность-связь.

С чего начинать разработку проекта:

- Разбили задачу на домены. Рассматриваем прикладной домен (наш).
- Разработка прикладного домена начинается с информационного моделирования. Мы начинаем всегда с физических объектов.
- Смотрим, какие объекты существуют.
- Пытаемся эти объекты сгруппировать по принципу одних и тех же характеристик.
- Выделяем, чем характеризуется объект - выделяем атрибуты объектов.

Концепции информационного моделирования

Разработка прикладного домена начинается с информационного моделирования.

Информационное моделирование включает в себя:

- Выделение сущностей, с которыми мы работаем
- Описание и анализ сущностей. Выделение их характеристик.
- Графическое представление сущностей.

Понятие атрибута. Типы атрибутов. Правила атрибутов.

Атрибуты - характеристики сущностей, переменные-члены объектов (данные). Каждая характеристика, которая является общей для всех экземпляров класса, выделяется как отдельный атрибут.

Идентификатор – то, что чётко определяет конкретный объект. Может состоять из одного или нескольких атрибутов.

Изменение атрибута не приводит к изменению самого объекта – объект просто меняет «имя», но остаётся тем же самым.

Типы атрибутов:

- Описательные атрибуты. Какая-то характеристика, внутренне присущая каждому объекту.
- Указывающие атрибуты. Которые используются как идентификатор, или как часть идентификатора.
- Вспомогательные атрибуты. Для формализации связи одного объекта с другими объектами. Для активных объектов будем выделять время жизни. Атрибут состояния.

Правила атрибутов:

1. Один объект класса имеет одно единственное значение для каждого атрибута в любой момент времени. Значения всех атрибутов должны быть определены.
2. Атрибут должен быть простым (не должен содержать никакой внутренней структуры). Сложный атрибут ⇒ новая сущность.
3. Когда объект имеет составной идентификатор, каждый атрибут являющийся частью идентификатора, представляет характеристику всего объекта, а не его частей.
4. Каждый атрибут, не являющийся указательным (частью идентификатора), должен характеризовать именно тот объект, который указан идентификатором, а не что-то другое.

Для каждого атрибута выделяем, какие значения может принимать атрибут. (Чтобы в дальнейшем определить тип для атрибута).

Каждый атрибут необходимо описать.

Из описания атрибута должно стать понятно, зачем мы выделили этот атрибут:

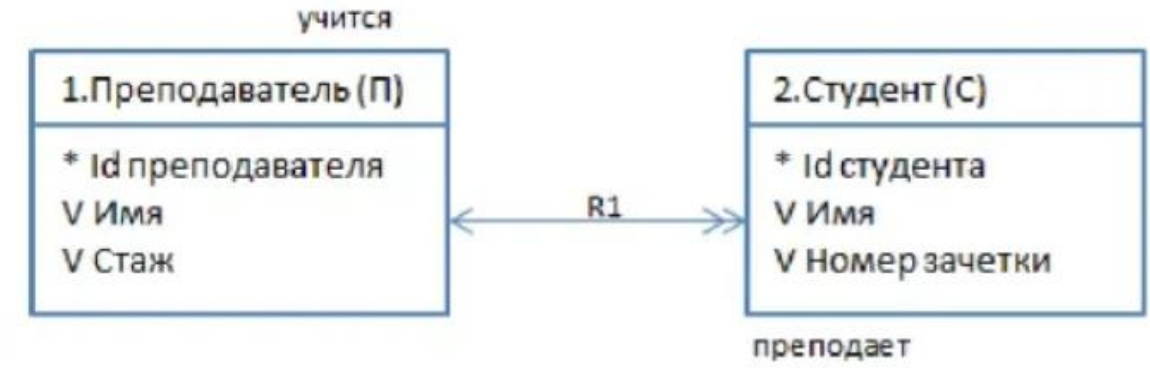
- Для описательного: показываем, какую характеристику хранит атрибут, как определяется и кто задает этот атрибут.
- Для идентифицирующего: показываем форму указания, кто назначает указание и степень, в которой идентифицирующий атрибут идентифицирует объект. Из группы идентифицирующих атрибутов выделяется привилегированный. (Если имя и фамилия - идентифицирующие атрибуты, из них фамилия - привилегированный).
- Для вспомогательного: показываем, какую связь формализует атрибут и почему мы так ее формализуем.

Понятие связи. Типы связей. Формализация связей. Композиция связей.

Связь – это абстракция отношений, которые возникают между объектами.

Задаём связь из перспективы каждого участвующего объекта.

Каждой связи присваивается уникальный идентификатор, который состоит из буквы и номера

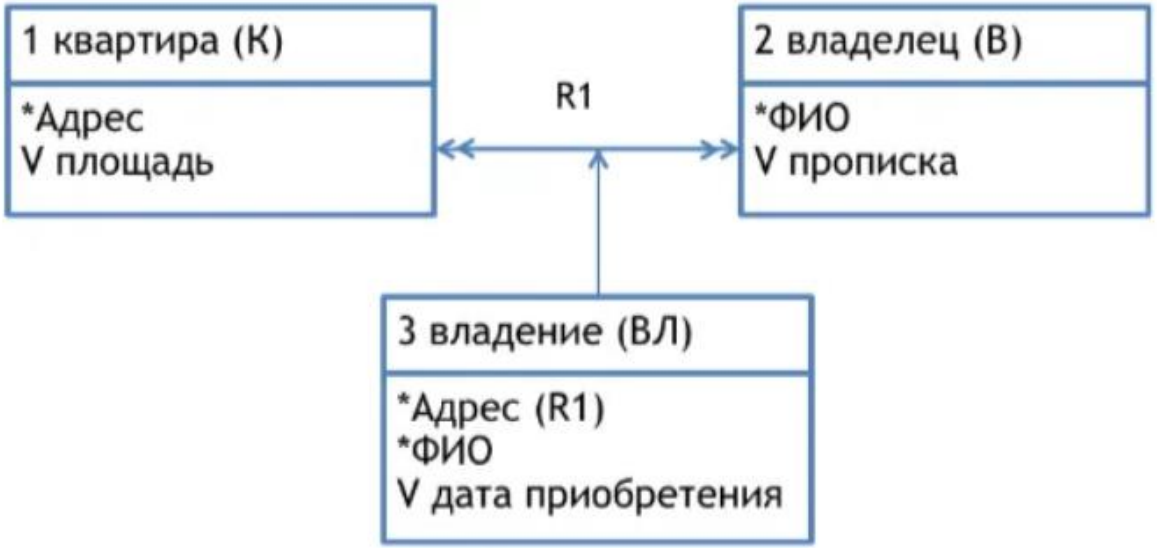


Типы связей:

- По множественности:
  - Один к одному <---------> (Муж - жена)
  - Один ко многим <---------->> (Преподаватель - студенты)
  - Многие ко многим <<---------->> (Студенты - учебные курсы)
- По условности:
  - Условные (один может не участвовать)
  - Безусловные (оба участвуют)
  - Биусловные (оба могут не участвовать) Если со стороны объекта связь условная, то у стрелки с его стороны ставится У.(не у всех преподавателей могут быть студенты, ставим У у студента)

Формализация связей:

- Один к одному: атрибут связи добавляется в главный из объектов, но если у связи динамическое поведение - с помощью ассоциативного объекта. Главный - наиболее осведомленный о всей системе.
- Один ко многим: атрибут связи добавляется со стороны многих (плохо, что связь держит зависимый объект, но если нам не важно, кто главнее, то и так сойдёт), но если у связи динамическое поведение или один главнее многих - с помощью ассоциативного объекта.
- Многие ко многим - с помощью ассоциативного объекта



Когда мы должны использовать ассоциативный объект:

1. Если связь имеет динамическое поведение
2. Если жизненные циклы объектов не совпадают

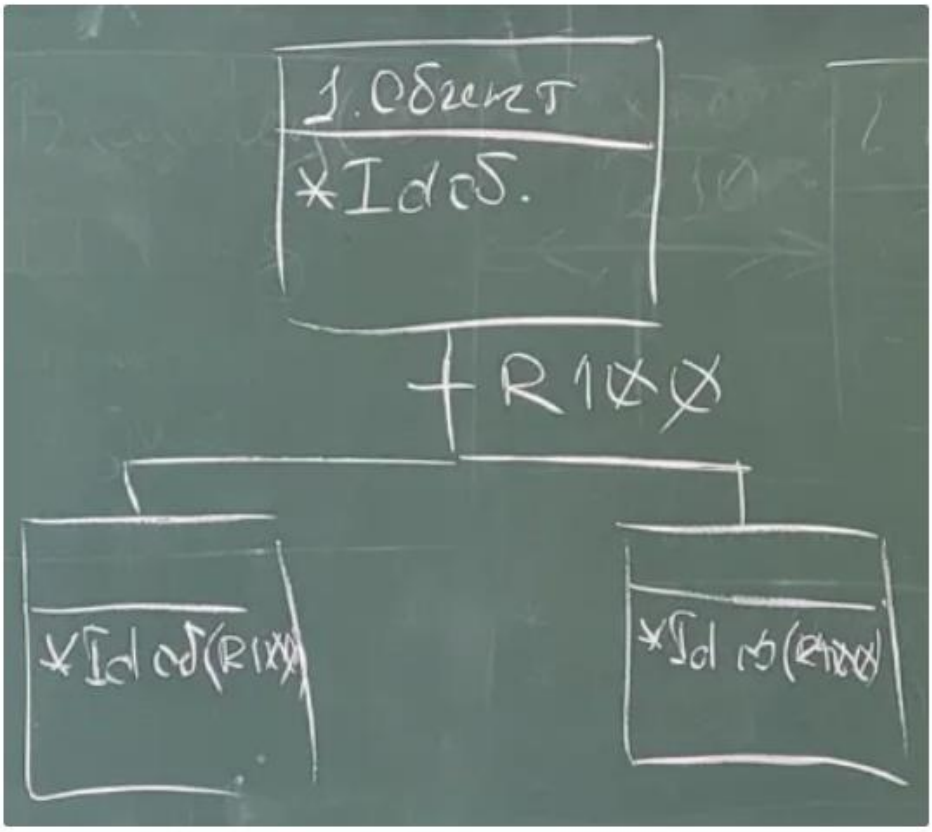
На информационной модели некоторые связи могут быть следствием другой связи. Такие связи мы обозначаем как композицию связей и не формализуем на диаграмме (потому что будет трудно разрулить ситуацию с несколькими ассоциативными объектами).

Если для объектов разных классов существует некий общий атрибут - объединяем их суперклассом.

Подтипы и супертипы.

Общие атрибуты для разных классов выносим в их суперкласс.

В объектно-ориентированном анализе суперкласс - всегда абстрактное понятие. Мы не рассматриваем возможность создания объектов суперкласса.



Связь суперкласса с подклассами обозначается номером, начинающемся со 100 (101, 102, 103...).

Диаграмма сущность-связь(ДСС).

На диаграмме сущность-связь сущности располагаются в прямоугольнике.

Каждой сущности присваиваем и уникальный для домена номер.

Выделяем имя сущности (желательно существительное).

Для имени указываем ключевой литерал.

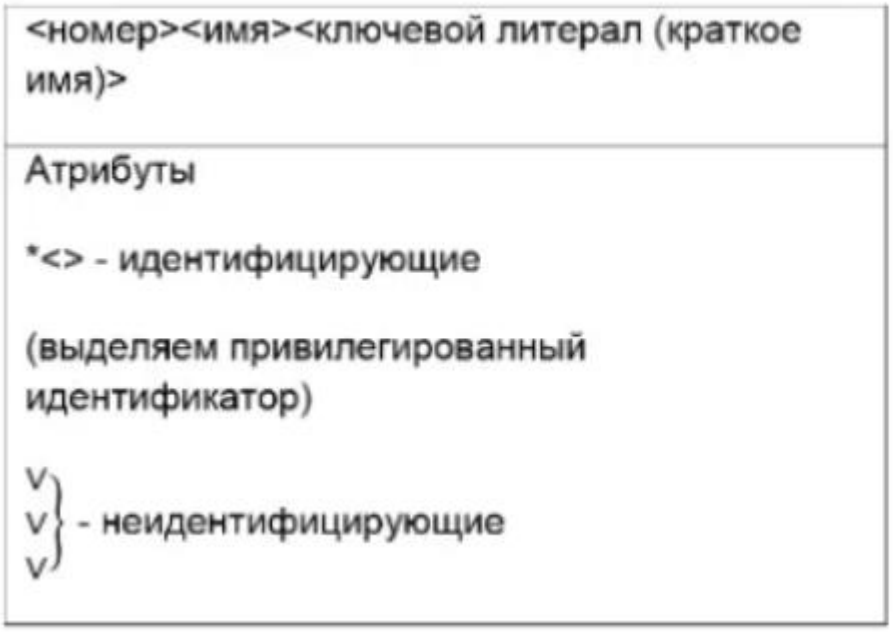
Указываем атрибуты:

- Привилегированный указывающий атрибут обозначаем “*”
- Все остальные атрибуты - просто перечисляем с “-”.

Между сущностями графически указываем связи.

Остальные правила и термины приведены выше.

- Графическое обозначение класса на информационной модели:



7. 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.

ОТВЕТ: 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.

Выделяем поведение объектов, отталкиваясь от реального физического мира. Объекты в течении времени жизни проходят некоторые стадии.

То, как эволюционирует объект между стадиям, характеризует черту поведения объекта.

Любой объект в данный момент времени находится в какой-то стадии.

Объект переходит из одной стадии в другую скачкообразно.

Не все переходы из одной стадии в другую возможны, т.е. есть некие правила переходов.

В физическом мире происходят инциденты, которые заставляют объекты переходить из одной стадии в другую. Или инциденты являются следствием перехода объекта из одной стадии в другую.

Для описания поведения объектов используем модель Мура:

- Множество состояний (стадий) объекта
- Множество событий (инцидентов), приводящих к переходу состояний
- Множество действий состояний
- Правила переходов состояний

Модель состояний можно формализовать таблицей и диаграммой.

Жизненный цикл и диаграмма перехода в состояния (ДПС).

Формы жизненных циклов:

1. Циркуляционный
2. Рождение-смерть

Когда формируются жизненные циклы:

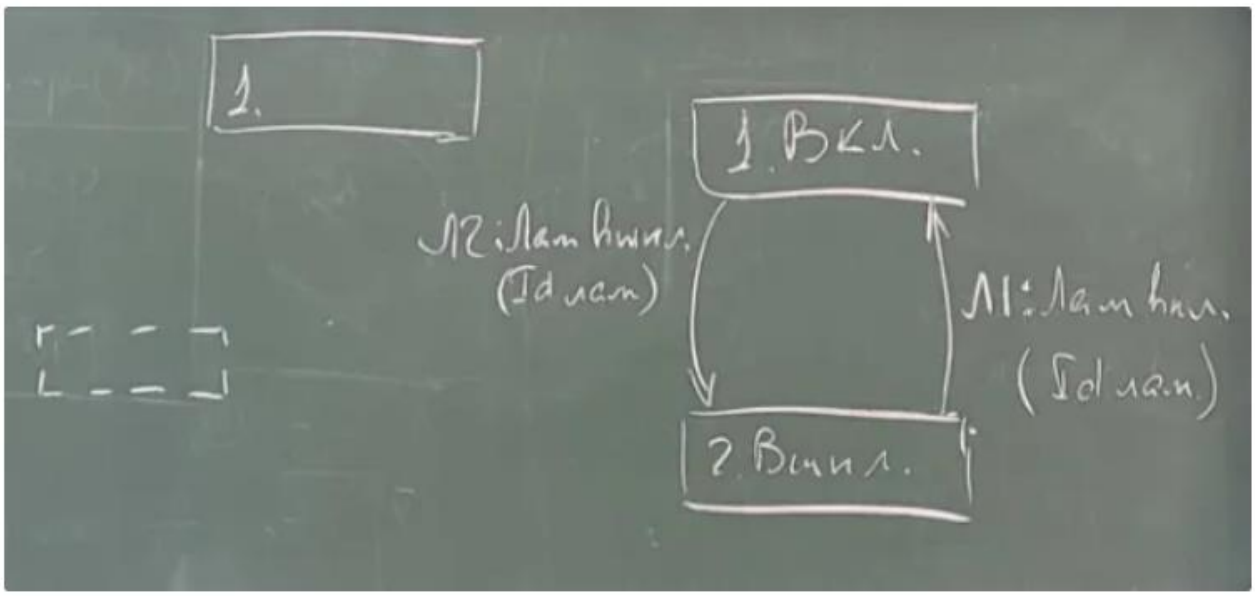
1. Создание или уничтожение во время выполнения
2. Миграция между подклассами
3. Объект производится или возникает поэтапно.
4. Объект – задача или запрос
5. Динамическая связь. Для пассивных объектов мы не выделяем жизненные циклы. Но иногда мы это делаем в интересах активных объектов.

ДПС - диаграмма переходов состояний.

Строим для каждой сущности.

Каждое состояние рисуем прямоугольником. В пунктирном прямоугольнике находится состояние уничтожения.

Каждому состоянию присваиваем номер и имя состояния.



Состояние. Виды состояний.

Состояние – это положение объектов, в котором определяются определённый набор правил, линий поведения, предписаний, определённых законов.

Виды состояний:

1. Состояние создания - в них объект появляется первый раз. В эти состояния происходит переход не из состояния.
2. Текущее состояние (не создания и не заключительное). Их может и не быть в физическом мире.
3. Заключительные состояния:

   3.1. Объект уничтожается - состояние рисуем пунктирной линией

   3.2. Состояние, из которого объект больше не переходит в другие состояния, но не уничтожается - рисуется как обычное состояние, из которого нет переходов

Чтобы определять текущее состояние, в класс добавляется вспомогательный атрибут - статус, который хранит текущее состояние.

Состояния контекста - промежуточные состояния, которые определяются предыдущим и следующим состояниями. Их цель - организовать этот переход.

События, данные событий. Действия состояний.

Событие – это абстракция инцидента или сигнала в реальном мире. События могут переносить данные.

Переход из одного состояние в другое происходит в результате события.

С каждым состоянием надо связывать действие (обработчик состояния), задача которого - перевести объект в это состояние.

Описание события:

- Значение события - короткая фраза, которая сообщает нам, что происходит с объектом в реальном мире.
- Предназначение - это модель состояний, которое принимает событие, может быть один единственный приёмник, для данного события.
- Метка – уникальная метка должна обеспечиваться для каждого события. Внешние события помечаются буквой «E».
- Данные события – события переносят данные. Все события, которые переносят объект из одного состояния в другое должны нести его идентификатор.

Действия состояний

Действие – это деятельность или операция, которая выполняется при достижении объектом состояния. Каждому состоянию ставится в соответствие одно действие.

Задачи действия:

1. Переводить в состояние
2. Менять идентификатор статуса
3. Выполнять любые вычисления
4. Порождать события других объектов своего или другого класса (в том числе для чего-либо вне области анализа нашей подсистемы)
5. Работать с таймером - создавать, удалять, считывать, запускать, очищать
6. Получать доступ к любым атрибутам любых классов (только на этом этапе формализации!)

Что обязано выполнять действие:

1. Гарантировать непротиворечивость объекта - после выполнения действия атрибуты объекта не должны противоречить друг другу.
2. При создании и удалении объектов (собственного класса) действие должно позаботиться о связях - чтобы они тоже не были противоречивыми (убиваешь мужа - переведи жену во вдовы).
3. Действие должно менять атрибут статуса на то состояние, которому оно соответствует.

Алгоритм действия можно разместить на ДПС непосредственно под действием (если оно в 1-2 строки).

Описание действия - псевдокод.

Если алгоритм большой - выносим его с ДПС.

(!) - Для данного объекта в момент времени может выполняться только одно действие.

(!) - События никогда не теряются.

(!) - Если событие порождено для объекта, который в данный момент выполняет действие, то оно будет принято объектом только после того, как действие будет выполнено. Но терять события нельзя!

(!) - Действия разных объектов могут выполняться одновременно.

(!) - Событие исчезает после того, как оно было принято.

Таблица перехода в состояния (ТПС). Правила переходов.

Правила, связывающие состояния и события:

1. Все события, которые переводят объект в одно и тоже состояние, должны нести одни и те же данные, т.к. состоянию ставится в соответствие действие, принимающее одни и те же данные. Идентификатор сущности, к которой применяется событие, должен переноситься как данные.
2. События, приводящие к созданию объекта, не несут его идентификатор.
3. События, переводящие объект из одного состояния в другое, должны нести его идентификатор.

Таблица переходов состояний - таблица, в которой мы устанавливаем правила переходов состояний.

В таблице строка - это состояние, состояния нумеруются. Столбец - это событие.

События могут игнорироваться (но пропускать их нельзя!) - событие не переводит объект в новое состояние. В этом случае ставится '-'.

Если данное событие не может произойти, когда объект находится в данном состоянии, то в ячейке ставится 'x' - желательно, чтобы таких ячеек было минимум или не было, потому что это означает исключительную ситуацию (плохо). *Можно выделить ошибочное состояние.*

Все атипичные ситуации должны обрабатываться так же, как и типичные.

В таблице все ячейки должны быть заполнены.

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

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

Модель взаимодействия объектов

Идея: свести всё к главной модели состояний.

МВО(модель взаимодействия объектов) – графическое представление взаимодействия между моделями состояний и внешними сущностями. Каждая модель состояний – овал, внешняя сущность – прямоугольник(называется терминатором).

Терминатор - внешняя сущность, из которой можно брать данные.

Виды терминаторов:

- Верхнего уровня: управляют подсистемой и рисуются на МВО сверху. Может быть только один.
- Нижнего уровня: управляются подсистемой и рисуются на МВО снизу. Их может быть много.

События, которые порождаются одной моделью состояний для другой или терминатором, рисуются стрелкой. События могут быть направлены как к терминаторам, так и от них.

- **МВО. Упрощённый пример со светильником**

Диаграмма взаимодействия объектов в подсистеме

МВО формируется иерархически – объекты, наиболее осведомленные о всей системе (активные) располагаются вверху диаграммы. Может быть схема верхнего и нижнего управления – система ограничена терминаторами сверху или снизу.

Типы событий

- Внешние события (приходят от терминатора или уходят к нему)
    - Незапрашиваемые события (не являются результатом предыдущих действий подсистемы, ~~то есть это события управляющие, при этом не переводят объект в новое состояние~~)
    - Запрашиваемые события(являются результатом предыдущих действий подсистемы, переводят объект в новое состояние)
- Внутренние (соединяют одну модель состояний с другой)

Схемы управления

- Схема верхнего управления (Эти события приходят от верхних терминаторов)
- Схема нижнего управления (Эти события приходят от нижних терминаторов)

Имитирование

Итак, мы рассмотрели взаимодействие нашей подсистемы с "внешним миром” (другими подсистемами). Теперь надо проверить систему на коллизии, т.е. сымитировать работу системы.

Имитирование - задание некоторых начальных состояний и генерация события. Наблюдение за работой системы. Оценка конечного результата.

Этап имитирования

Мы генерируем некоторое начальное состояние. Принимает незапрашиваемое событие и смотрим, какое состояние приняли все объекты в нашей системе. Процесс имитирования может быть очень сложным.

Для каждого действия есть 2 времени:

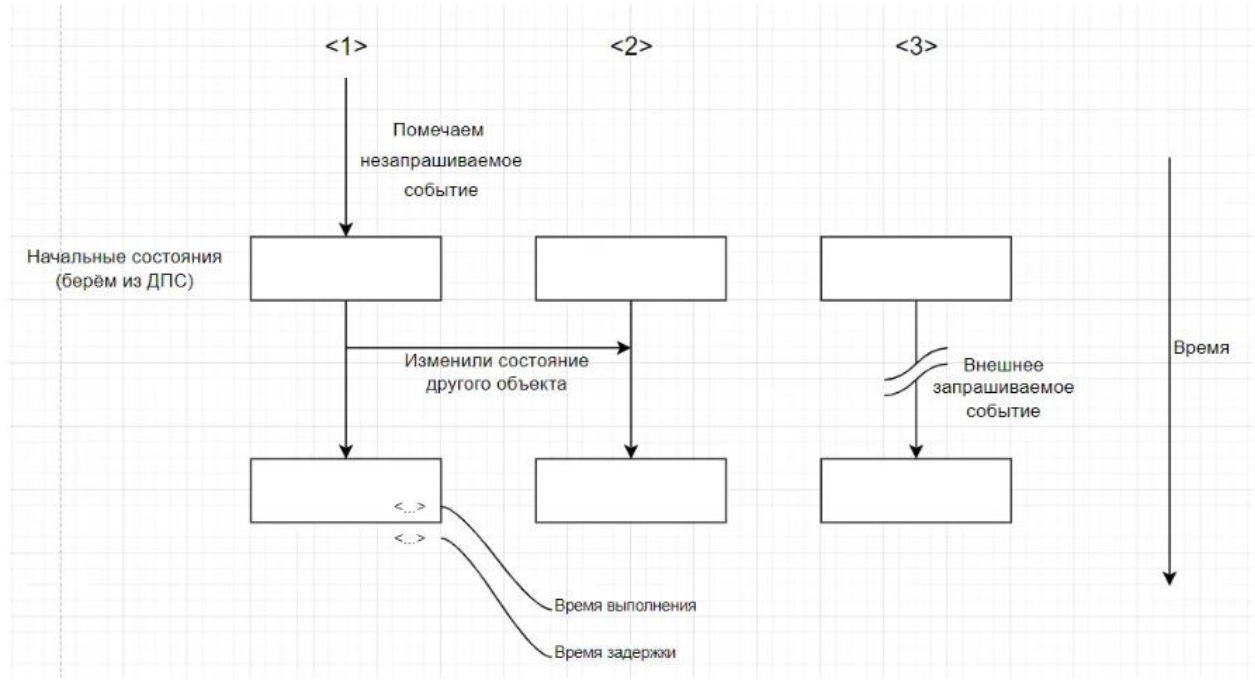
- Время выполнения
- Время задержки – время, на протяжении которого объект должен находиться в состоянии (невозможен резкий переход из одного состояния в другое, мы должны учитывать время задержки)

Этапы имитирования (тесты)

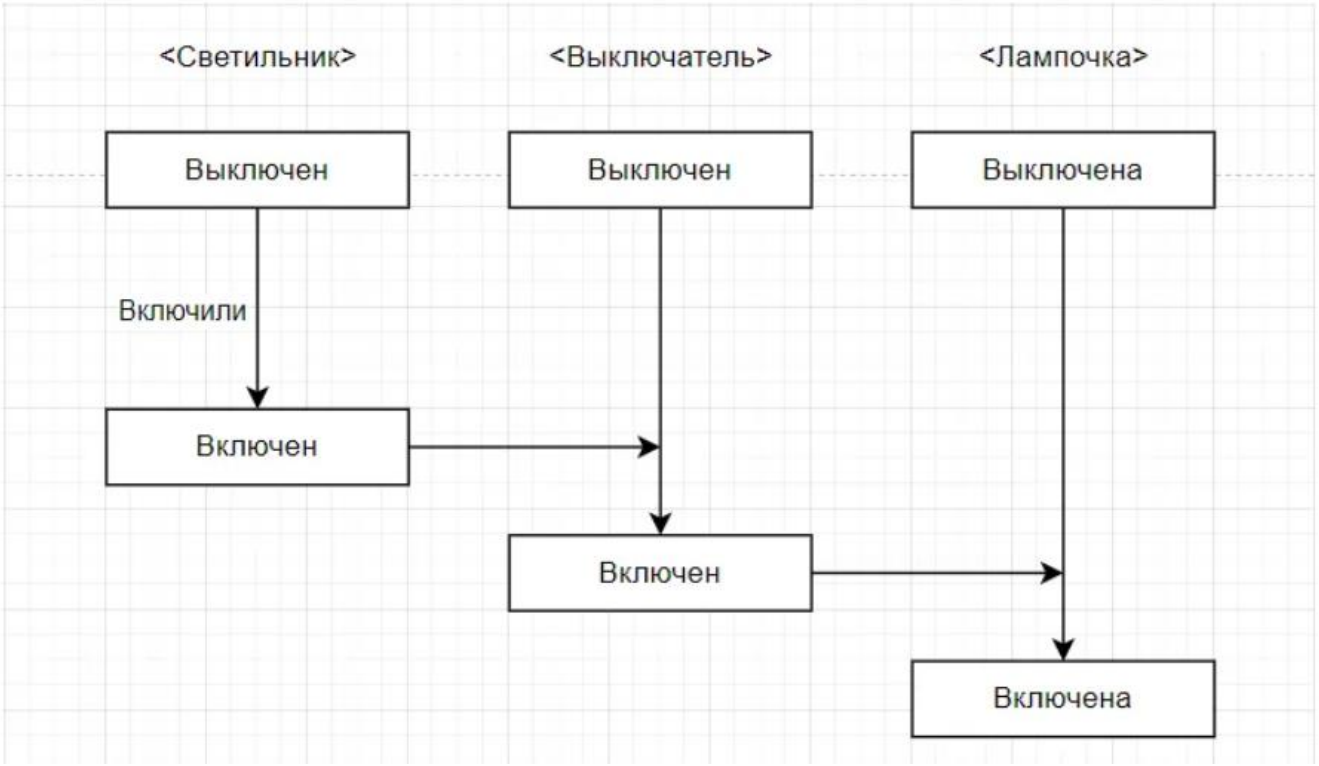
- Установить начальное состояние системы
- В каждом состоянии проверить, как подсистема будет реагировать на все незапрашиваемые события, и построить соответствующие каналы управления.
- Оценить конечный результат

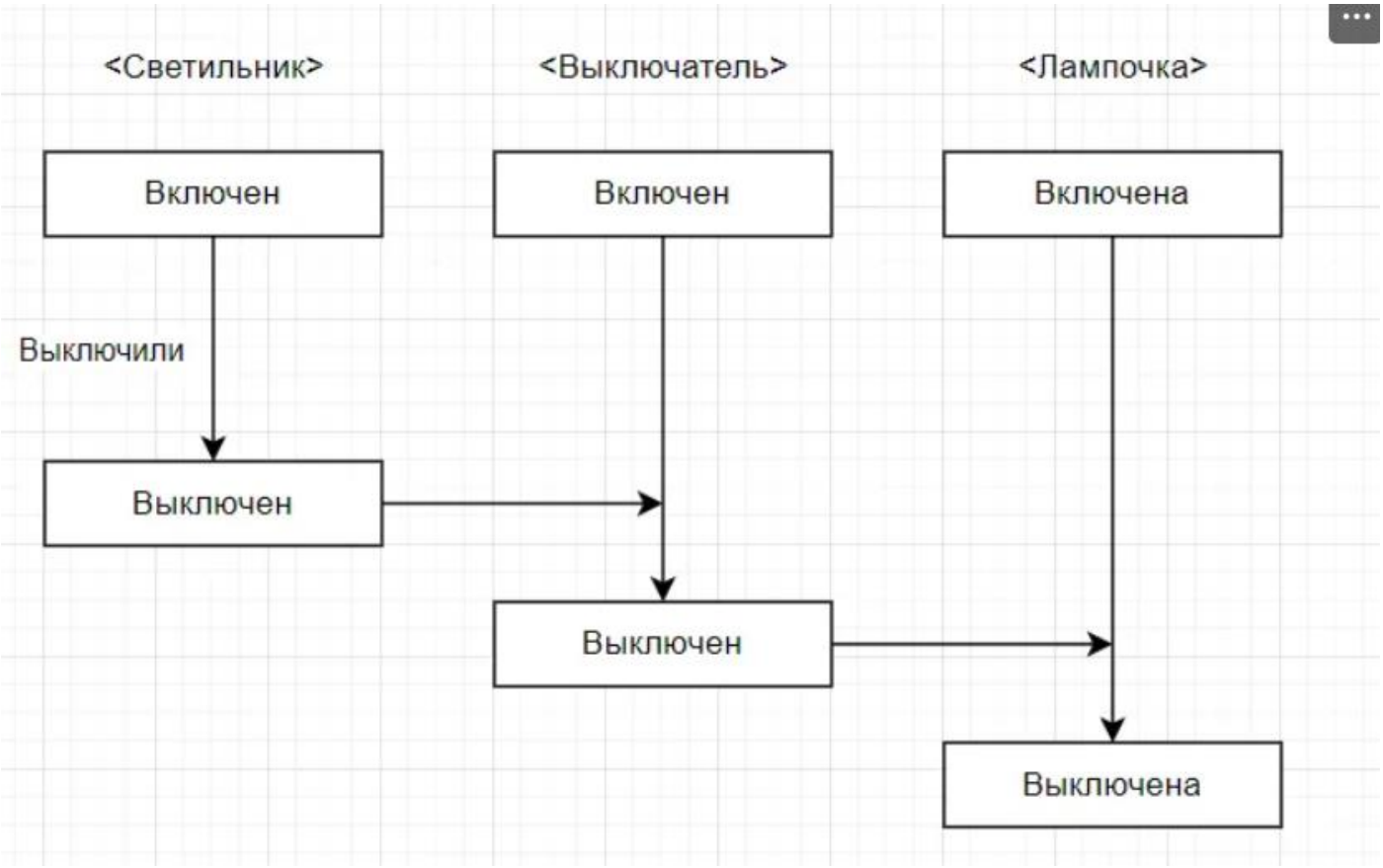
Каналы управления

Канал управления – последовательность действий и событий, которые происходят в ответ на поступление некоторого незапрашиваемого состояния, когда система находится в определённом состоянии. Если возникло событие к терминатору, и эти события приводят к дальнейшим событиям от терминатора, то мы их тоже включаем в канал управления.



Каналы управления. Пример





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

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

Диаграмма потоков данных действий

Пришла на смену схеме алгоритма, которая не показывала зависимость процессов по данным и выстраивала чёткую последовательность операций, которой могло бы и не быть…

ДПДД (Диаграмма потоков данных действий) – диаграмма, показывающая зависимость процессов по данным и позволяющая рассматривать алгоритм с точки зрения распараллеливания.

Обеспечивает графическое представление модулей процесса в пределах действия и взаимодействия между ними.

Дёт понимание, что в каком порядке нужно выполнять, что можно распараллелить и где можно изменить порядок. Но нам важнее выделить процессы.

Строится для каждого действия каждого состояния каждой модели состояний.

На диаграмме каждый процесс рисуется овалом, внешняя сущность – прямоугольником. Процессы могут получать данные от других процессов и от каких-либо внешних сущностей.

Типы процессов:

- Аксессоры (А) - процесс, чья единственная цель состоит в том, чтобы получить доступ к данным одного архива данных (процессы, которые читают какой-либо атрибут, записывают, создают или уничтожают объекты):
    - Чтение
    - Запись
    - Создание
    - Уничтожение
- Генераторы событий (Г) - создаёт/принимает лишь одно событие (стрелочка наружу процесса)
    - Принимающие события
    - Порождающие события
- Вычисления/преобразования (В)
- Проверки - условные переходы (У)

Контроль за считывание/запись атрибутов в других объектах (за их целостность) лежит на аксессорах, т.к. сами объекты не могут это проконтролировать.

Каждый процесс нужно именовать и описывать.

Аксессоры – какие атрибуты считывают или записывают, какие объекты создают или уничтожают. Генераторы событий – результат - событие, метка события. Преобразования – что делают. Проверки –«проверить, что...»

Правила построения ДПДД:

1. Процесс может выполняться, когда все вводы доступны.
2. Выводы процесса доступны, когда он завершает своё выполнение.
3. Данные событий (на диаграмме это стрелка сверху), данные из архивов данных и терминаторов всегда доступны

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

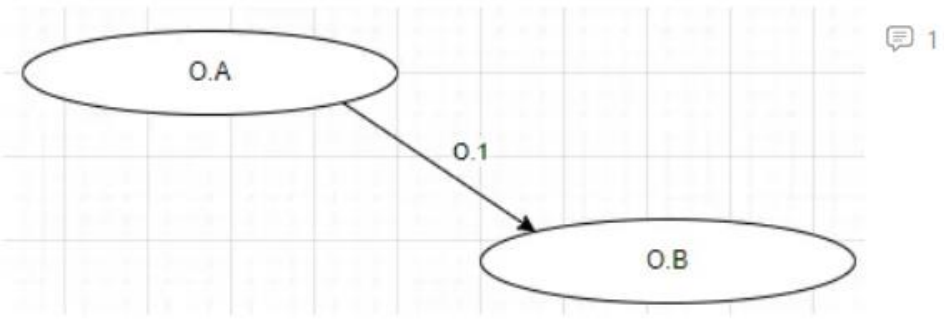
Пример:

Таблица процессов состояний (ТПС)

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.. Их надо выносить в общее действие.

Модель доступа к объектам (МДО).

На основе выделенных аксессорных процессов строится модель доступа к объектам. На модели доступа, модели состояний (объектов) рисуются вытянутыми овалами. Если А использует аксессор модели состояний В, то рисуется стрелка, А будет аксессором. На стрелке указывается ID процесса и его название.



Аксессоры реализуются добавлением в объект действий по записи и чтению атрибутов. Данная диаграмма представляет синхронное взаимодействие. Может использоваться совместно с асинхронной – не обязательно данные переносят события. Автобусная остановка: событие «пришел автобус», говорит лишь о том что «пришел транспорт», подходящий по функции; а может и нести информацию «пришел автобус №». Если данное не переносится, то с объектом надо вступить в аксессорное взаимодействия.

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

ОТВЕТ: Домены. Модели доменного уровня. Типы доменов. Мосты, клиенты, сервера.

Домены

Когда имеется большая система, мы разбиваем задачу на домены.

Домен - отдельный реальный, абстрактный или гипотетический мир, населенный отчетливым набором объектов, существующих по характерному набору правил или линий поведения.

Каждый домен образует отдельное и связное единое целое.

Главная идея разбиения системы на домены состоит в том, чтобы можно было максимально безболезненно подменять эти домены, а также делегировать их разработку разным людям/отделам.

Один домен не должен требовать наличия класса в другом домене (чтобы можно было подменять домены).

Классификация доменов:

- Прикладные - основные домены, которые решают непосредственно нашу задачу.
- Архитектурный – домен, отвечающий за архитектуру построения системы (один домен на систему); обеспечивает общие механизмы для управления системой и данными (механизмами их передачи). Формализацию начинают с него. Есть уже готовые решения - архитектурные паттерны, которые могут перерастать в целые технологии.
- Сервисные домены – обеспечивают функции, необходимые для поддержания прикладного домена, всевозможные сервисные функции. Пример: интерфейс.
- Домены реализации – стандартные, библиотечные функции и так далее. Дают возможность лёгкой замены одной реализации на другую. По сути это набор функционала. Примеры: функционал ОС, библиотеки QT, Boost, STL и т.д.

Мосты, клиенты, сервера

Выделяется 2 вида доменов:

- Которые **предоставляют возможности - сервера**
- Которые **используют возможности других доменов – клиенты**

Между клиентом и сервером есть **мост**. Каждую сторону интересует только то, что будет на мосту, и не более того.

Клиент рассматривает мост как набор каких-то предложений, которые кто-то ему представляет.

Сервер – набор требований для выполнения.

Схема доменов

Для доменов реализуется диаграмма связи доменов (ДСД).

Диаграмма доменного уровня как правило содержит в верхней части – домены, наиболее осведомлённые о системе (прикладные), внизу – сервисные и реализации.

При разбиении задачи для проектирования каждого домена можно использовать разные технологии. Например в задаче отрисовки, непосредственно рисование – структурный подход, а различные взаимодействия на сцене – объектный. Доменный подход позволяет в дальнейшем легко заменить один домен на другой; сервер рассматривается как набор предложений.

Прикладной домен разбивается на подсистемы, в то время как сервисный – просто набор функций. Для домена, так же как и для подсистемы, рисуется три диаграммы.

- Модель связей подсистем (по информационной модели);
- Модель взаимодействия подсистем (по МВО);
- Модель доступа подсистем (по МДО);
    - 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, таблица процессов состояний, диаграмма потоков данных действий.
- **Доп**

  Все как бы на более высоком уровне

    - Диаграмма сущность связь —> Диаграмма связей подсистем
    - Модель доступа к объектам —> Диаграмма доступа подсистем
    - Модель взаимодействия объектов —> Диаграмма взаимодействия подсистем

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

ОТВЕТ: Объектно-ориентированное проектирование. Диаграмма класса. Структура класса. Диаграмма зависимостей. Диаграмма наследования.

Объектно-ориентированное проектирование

Объектно-ориентированное проектирование заключается в формировании диаграмм и структур классов, диаграммы зависимостей и наследования.

Недостаток документов, которые мы получили на этапе анализа:

- Непригодны для написания кода (да, у нас уже много чего есть, даже алгоритмы, но мы, например, ничего не говорили о типах данных)

Ни одна из существующих нотаций не отражает всех возможностей технологии ООП. UML, например, слишком избыточна. В данном случае лучшим (но не идеальным) вариантом будет нотация Гради Буча. Несмотря на то, что она тоже избыточна, в ней надо построить всего 4 диаграммы. Данные для них мы собираем по всем диаграммам, которые мы получили на этапе анализа.

Диаграмма класса

Описывает внешнее представление данного класса.

Слева: принимаемые, возвращаемые, изменяемые параметры

Ромб – кидает исключительную ситуацию.

Пунктир – отсроченный (асинхронный) вызов метода

Палочка слева - метод с динамическим связыванием.

- Картинки

Кидает исключительную ситуацию

метод с динамическим связыванием

Также надо определить начальные значения и варианты инициализации атрибутов.

ID здесь можно использовать по надобности, а можно и опускать.

Схема структуры класса

Задача: проследить потоки данных и доступ к данным конкретного класса. На этой диаграмме все строится вокруг данных объекта. Нас интересуют методы, которые используют данные извне.

Схема структуры класса конкретизирует внутреннее представление и структуру класса. Класс реализуется по слоям. По сравнению с прошлой диаграммой, у нас теперь есть не только публичные методы, но и приватные; доступ к нижним слоям напрямую снаружи недопустим, как и вызов публичного метода публичным методом(во второй лабе тассов закрывал на это глаза).

На каждой линии показываем, какие данные получают методы, и какие - возвращают. Данные показываем гробиками. От гробика стрелочка вверх - данные возвращаемые, вниз - принимаемые методом. И вверх, и вниз - изменяемые параметры.

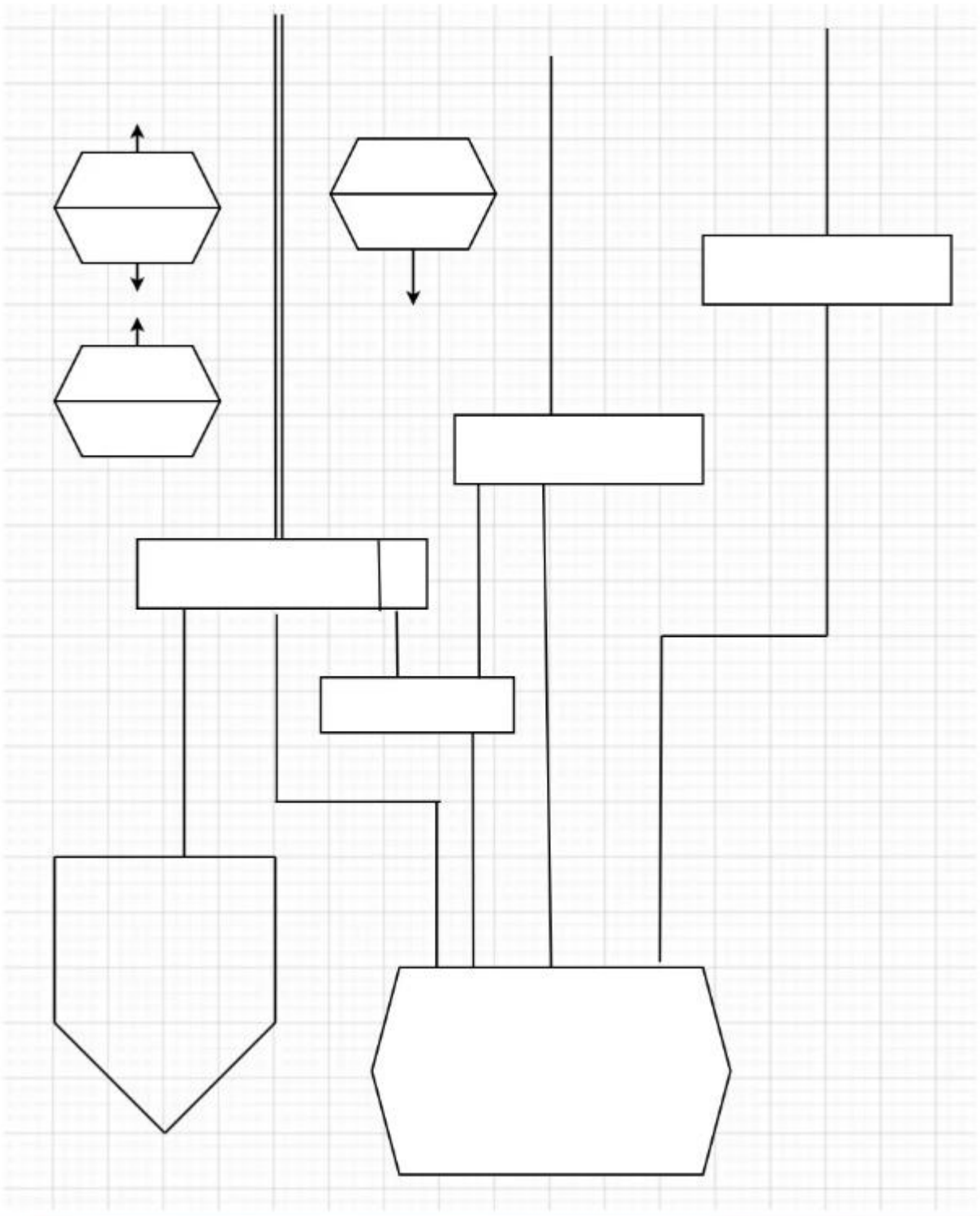
Четко выделяются слои методов: слой методов, вызываемых извне класса, - общедоступные, находятся выше. И слой скрытых методов, не вызываемых извне. Желательно, чтобы методы интерфейса не вызывали друг друга - это очень важно. Если в интерфейсных методах есть общая часть, мы выделяем ее в метод скрытого слоя.

Если метод обрабатывает исключение, помечаем его вертикальной линией в правой части.

Если метод получает или передает данные во внешний модуль - показываем поток данных из этого метода в перевернутый домик.

Пунктир - граница между public (сверху) и private (снизу) методами. Нотация не идеальна в том плане, что здесь, например, не понятно, что private, а что protected. Хотя вроде бы тут можно делать больше уровней…

Шестиугольник - архив данных.



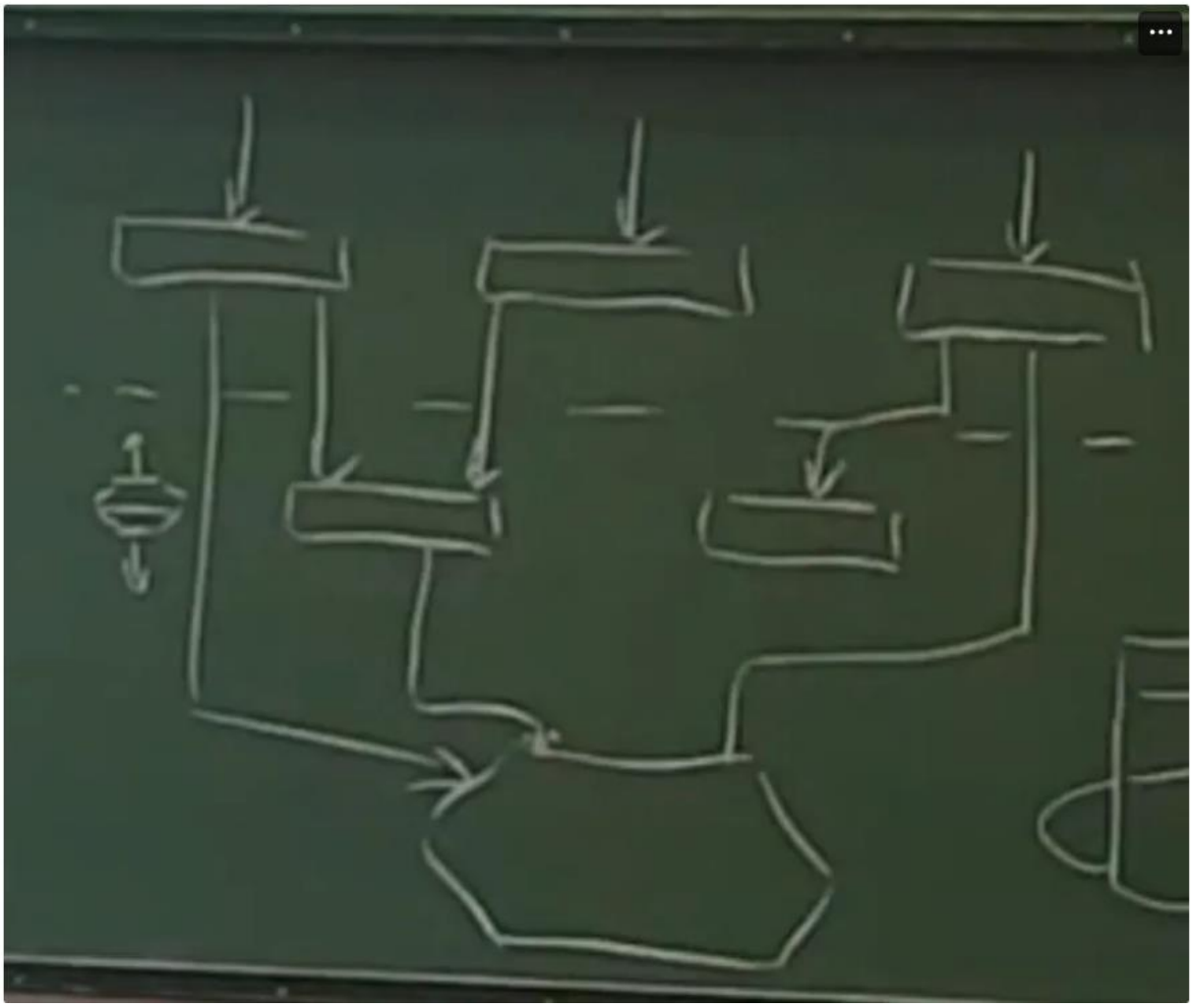
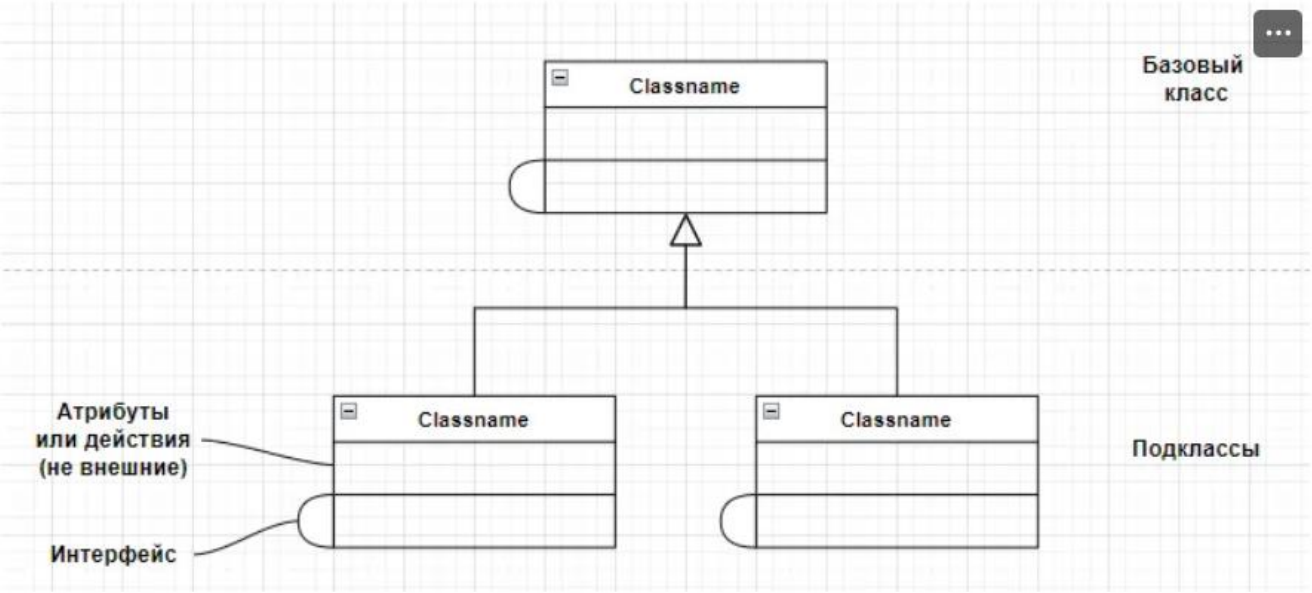


Диаграмма наследования

Класс разбивается на методы и атрибуты.

Диаграмма наследования – от базового класса подклассы. Внутри атрибуты и методы.



Базовое понятие должно быть абстрактным.

Если у базы нет атрибутов, то рассматриваем её как интерфейс (в C++ их нет).

Диаграмма зависимостей

Диаграмма зависимостей – строится диаграмма класса, только оставляем заголовок, а атрибуты перечисляем списком.

Одинарная стрелка - либо ассоциация классов (если стрелка от класса к классу), либо вызов одного метода другим (если стрелка от метода к методу).

Двойная стрелка - агрегация (жизненные циклы объектов связаны).

12. Порождающие паттерны: фабричный метод (Factory Method), абстрактная фабрика (Abstract Factory), строитель (Builder). Их преимущества и недостатки.

ОТВЕТ: Порождающие паттерны: фабричный метод (Factory Method), абстрактная фабрика (Abstract Factory), строитель (Builder),

Отличие шаблонов от паттернов: шаблон - конкретная реализация чего-либо, а паттерн - шаблон для решения какой-то задачи (как может решаться данная задача). Паттерн мы всегда адаптируем к своей задаче.

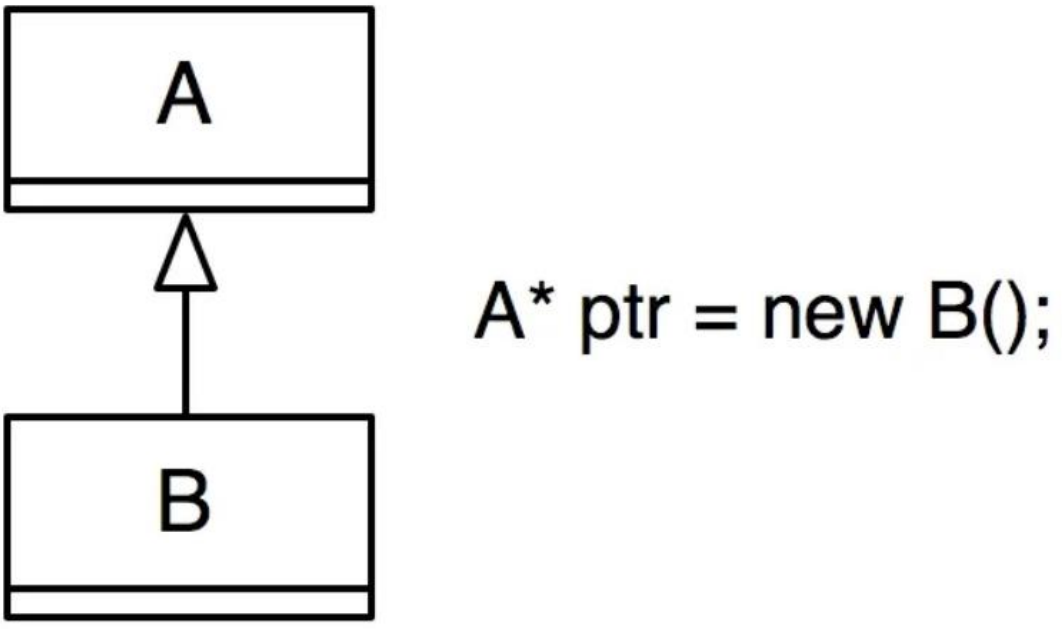
Преимущества использования паттернов:

- Мы имеем готовое решение
- За счет готового решения - нюансы все выявлены => надежный код
- Повышается скорость разработки
- Повышается читаемость кода
- Улучшается взаимодействие с коллегами (достаточно сказать название паттерна, который вы используете, и всё сразу станет понятно)

Задача

Задача порождающего паттерна: создание объектов

Проблема: не может быть полиморфных конструкторов. Если внутри метода мы работаем со ссылкой/указателем на базовое понятие, будет вызван конструктор базового понятия.



Фабричный метод

Идея

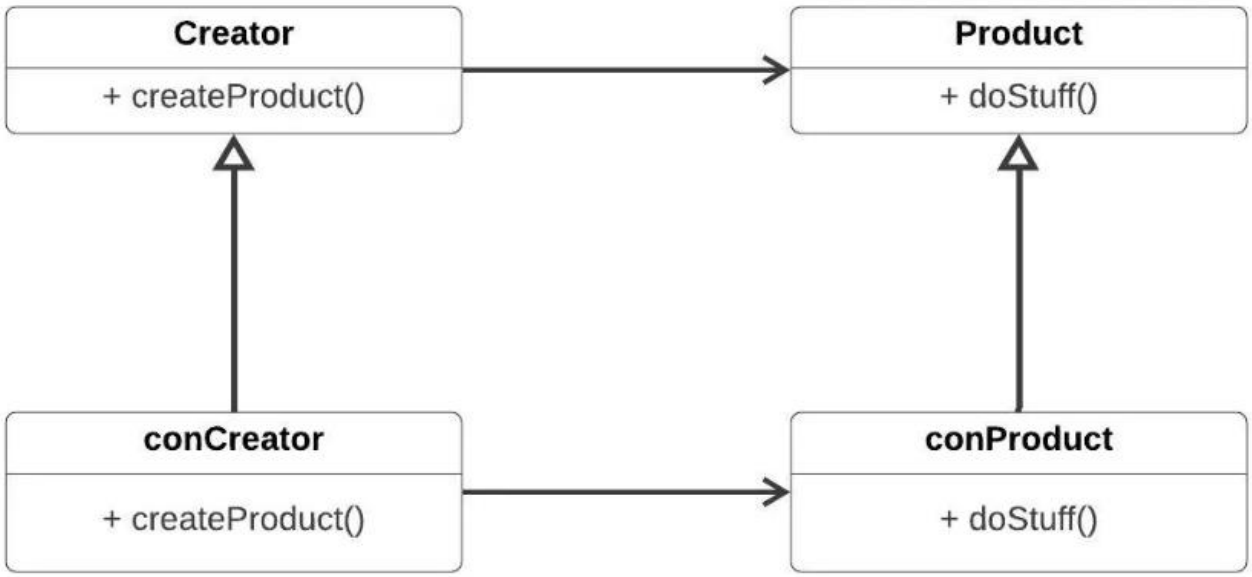
Разнести на две задачи:

1. Принятие решения, какой объект создавать.

2. Создание объекта, причем при создании объекта нужно "отвязаться" от конкретного типа.

Мы будем создавать специальные объекты, которые отвечают за порождение других объектов.

Диаграмма



Использование

Когда нам нужно использовать фабричный метод:

1. Основная задача: подмена одного объекта на другой.
2. Когда принятие решение в одном месте кода, создание - в другом.

Преимущества

- Облегчается добавление новых классов, избавляем методы от привязки к конкретным классам.
- Код очищается от `new`, используя полиморфизм по полной (создаём новые классы, не изменяя уже написанный код)
- Паттерн работает во всех языках
- Позволяет разнести в коде принятие решения о создании объекта (solution) и само создание (creator)
- Решение о том, какой объект создавать, принимается во время выполнения, тогда же можно менять это решение
- Пример. Фабричный метод (Factory Method). Новый объект.

```
#include <iostream>
#include <memory>

using namespace std;

class Product;

class Creator {
 public:
    virtual unique_ptr<Product> createProduct() = 0;
};
```

```
template <typename Tprod>
class ConCreator : public Creator {
 public:
    virtual unique_ptr<Product> createProduct() override {
        return unique_ptr<Product>(new Tprod());
    }
};

class Product {
 public:
    virtual ~Product() = 0;
    virtual void run() = 0;
};

Product::~Product() {}

class ConProd1 : public Product {
 public:
    virtual ~ConProd1() override { cout << "Destructor;" << endl; }
    virtual void run() override { cout << "Method run;" << endl; }
};

#pragma endregion

int main() {
    shared_ptr<Creator> cr(new ConCreator<ConProd1>());
    shared_ptr<Product> ptr = cr->createProduct();

    ptr->run();
}
```

Пример. Фабричный метод (Factory Method). Без повторного создания.

```
# include <iostream>
# include <memory>

using namespace std;

class Product;

class Creator
{
public:
    shared_ptr<Product> getProduct();

protected:
    virtual shared_ptr<Product> createProduct() = 0;

private:
    shared_ptr<Product> product;
};

template <typename Tprod>
class ConCreator : public Creator
{
protected:
    virtual shared_ptr<Product> createProduct() override
    {
        return shared_ptr<Product>(new Tprod());
```

```
    }
};

shared_ptr<Product> Creator::getProduct()
{
    if (!product)
    {
        product = createProduct();
    }

    return product;
}

class Product
{
public:
    virtual ~Product() = 0;
    virtual void run() = 0;
};

Product::~Product() {}

class ConProd1 : public Product
{
public:
    virtual ~ConProd1() override { cout << "Destructor;" << endl; }
    virtual void run() override    { cout << "Method run;" << endl; }
};

int main()
{
    shared_ptr<Creator> cr(new  ConCreator<ConProd1>());
    shared_ptr<Product> ptr1 = cr->getProduct();
    shared_ptr<Product> ptr2 = cr->getProduct();

    cout << ptr1.use_count() << endl;
    ptr1->run();
}
```

- **Solution**

  Принятие решения о том, кто будет создавать объект, мы выносим. Класс, принимающий решение, объект не создает. Он создает объект для его создания.

  Solution предоставляет метод для регистрации creator-ов.

  На основе чего Solution может принять решение, какой класс создавать? Solution должен быть независим от реализации, от конкретного набора классов - следовательно, **мы не можем использовать switch-case.**

  Идея решения:

  создаем карту продуктов, которые у нас существуют. При добавлении нового класса, регистрируем его в этой карте. Используя эту карту, осуществляем выбор. **Solution предоставляет метод для регистрации креаторов классов (в этой карте).**

- Пример. Фабричный метод (Factory Method). Разделение обязанностей.

  Solution предоставляет метод для регистрации (в данном случае) Creator'ов. В данном случае для карты - map, состоящий из пар (pair): ключ + значение. Таким образом мы избавились от конструкции switch.

  - Код

```
# include <iostream>
# include <memory>
# include <map>

using namespace std;

class Product;

class Creator
{
public:
    virtual unique_ptr<Product> createProduct() = 0;
};

template <typename Tprod>
class ConCreator : public Creator
{
public:
    virtual unique_ptr<Product> createProduct() override
    {
        return unique_ptr<Product>(new Tprod());
    }
};

#pragma region Product
class Product
{
public:
    virtual ~Product() = 0;
    virtual void run() = 0;
};

Product::~Product() {}

class ConProd1 : public Product
{
public:
    virtual ~ConProd1() override { cout << "Destructor;" << endl; }
    virtual void run() override { cout << "Method run;" << endl; }
};

unique_ptr<Creator> createConCreator()
{
    return unique_ptr<Creator>(new ConCreator<ConProd1>());
}

class Solution
{
public:
    typedef unique_ptr<Creator> (*CreateCreator)();

    bool registration(size_t id, CreateCreator createfun)
    {
        return callbacks.insert(CallBackMap::value_type(id,
createfun)).second;
    }
```

```
    bool check(size_t id) { return callbacks.erase(id) == 1; }

    unique_ptr<Creator> create(size_t id)
    {
        CallBackMap::const_iterator it = callbacks.find(id);

        if (it == callbacks.end())
        {
            throw IdError();
        }

        return unique_ptr<Creator>((it->second)());
    }

private:
    using CallBackMap = map<size_t, CreateCreator>;

    CallBackMap callbacks;
};

int main()
{
    Solution solution;

    solution.registration(1, createConCreator);

    shared_ptr<Creator> cr(solution.create(1));
    shared_ptr<Product> ptr = cr->createProduct();

    ptr->run();
}
```

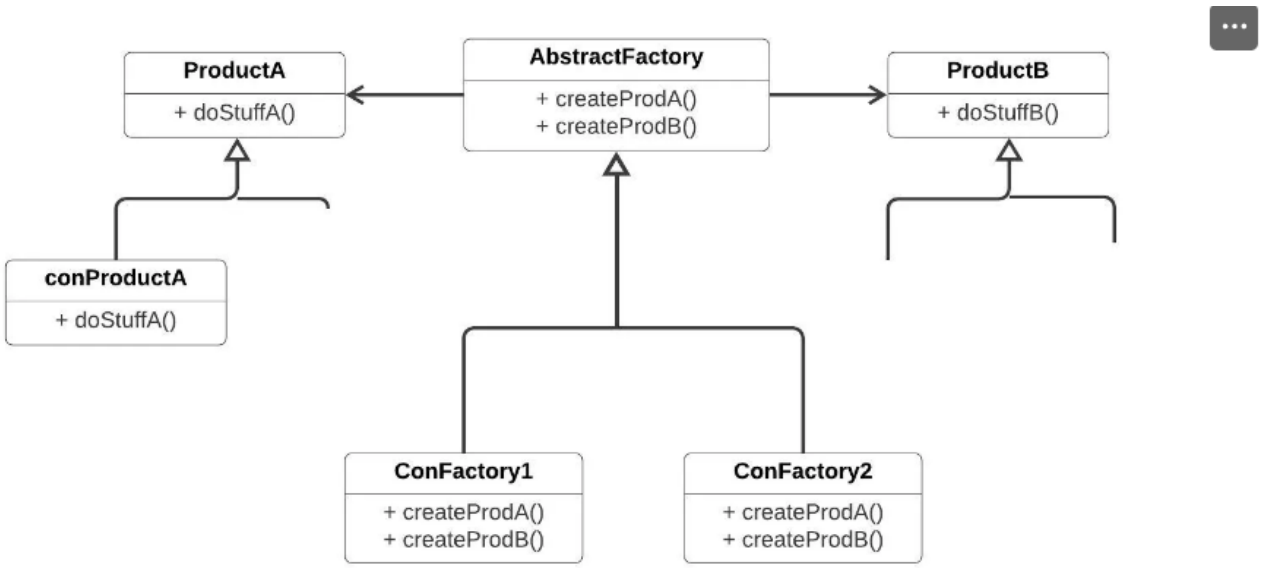
Абстрактная фабрика

Абстрактная фабрика - развитие фабричного метода с добавлением функционала

- Задача: создание "семейства" разных объектов, но связанных между собой

  Можно "плодить" разные ветви Creator'ов под каждый тип. продуктов, но мы теряем связь между этими продуктами.

Пример: в графических библиотеках: кисточка, ручка, сцена и т. п.



Каждая конкретная фабрика будет отвечать за создание определенного семейства объектов

Как и для фабричного метода, должен быть solution, который принимает решение, какую фабрику создавать

- Преимущества: не надо контролировать создание каждого объекта - только всего семейства целиком. Целостность системы.
- Недостаток: абстрактная фабрика накладывает требования на продукты (в разных семействах должны быть представлены все продукты, которые определяет базовая абстрактная фабрика). Очень сложно привести всё к единому интерфейсу.
- **Пример кода. Абстрактная фабрика (Abstract Factory).**

```
# include <iostream>
# include <memory>

using namespace std;

class Image {};
class Color {};

class BaseGraphics
{
public:  virtual ~BaseGraphics() = 0;
};
BaseGraphics::~BaseGraphics() {}

class BasePen {};
class BaseBrush {};

class QtGraphics : public BaseGraphics
{
public:
  QtGraphics(shared_ptr<Image> im) { cout << "Constructor QtGraphics;" << endl; }
  virtual ~QtGraphics() override { cout << "Destructor QtGraphics;" << endl; }
};

class QtPen : public BasePen {};
class QtBrush : public BaseBrush {};

class AbstractGraphFactory
{
public:
  virtual unique_ptr<BaseGraphics> createGraphics(shared_ptr<Image> im) = 0;
  virtual unique_ptr<BasePen> createPen(shared_ptr<Color> cl) = 0;
  virtual unique_ptr<BaseBrush> createBrush(shared_ptr<Color> cl) = 0;
};

class QtGraphFactory : public AbstractGraphFactory
{
  virtual unique_ptr<BaseGraphics> createGraphics(shared_ptr<Image> im)
  { return unique_ptr<BaseGraphics>(new QtGraphics(im)); }

  virtual unique_ptr<BasePen> createPen(shared_ptr<Color> cl)
  { return unique_ptr<BasePen>(new QtPen()); }

  virtual unique_ptr<BaseBrush> createBrush(shared_ptr<Color> cl)
  { return unique_ptr<BaseBrush>(new QtBrush()); }
};
```

```
int main()
{
  shared_ptr<AbstractGraphFactory> grfactory(new QtGraphFactory());

  shared_ptr<BaseGraphics> graphics1 = grfactory->createGraphics(shared_ptr<Image>(new Image()));
  shared_ptr<BaseGraphics> graphics2 = grfactory->createGraphics(shared_ptr<Image>(new Image()));
}
```

13. Порождающие паттерны: одиночка (Singleton), прототип (Prototype), пул объектов (Object Pool). Их преимущества и недостатки.

ОТВЕТ:

Одиночка (Singleton)

Возникают задачи, в которых должно быть гарантировано, что создан только один объект класса.

Решение: убрать конструкторы из public части. В паблик части сделать статический метод, который будет при необходимости порождать объект. В статическом методе содержится статический член класса своего объекта. Поскольку он статический – будет создан только один раз. Конструктор находится в private части. Нельзя копировать – запрещаем конструктор копирования и оператор присваивания.

Недостатки метода:

- Глобальный объект: доступ через глобальный интерфейс, вызовом статического метода
- Проблема шаблона: лишаемся подмены. Решение о том, какой объект создавать, принимается на этапе компиляции. Шаблоны здесь лучше не использовать.

Альтернатива:

- фабричный метод.
- **Пример кода. Singleton обычный (Запрещаем конструктор копирования и оператор присваивания)**

```
# include <iostream>
# include <memory>

using namespace std;

class Product
{
public:
    static shared_ptr<Product> instance()
    {
        static shared_ptr<Product> myInstance(new Product());

        return myInstance;
    }
    ~Product() { cout << "Destructor;" << endl; }

    void f() { cout << "Method f;" << endl; }

    Product(const Product&) = delete; // запрещаем
    Product& operator=(const Product&) = delete; // запрещаем

private:
    Product() { cout << "Default constructor;" << endl; }
};

int main()
{
    shared_ptr<Product> ptr(Product::instance());

    ptr->f();
```

- }
- **Пример кода. Singleton шаблонный (Запрещаем конструктор копирования и оператор присваивания)**

```
# include <iostream>
# include <memory>

using namespace std;

template < Type>
class Singleton
{
public:
    static Type& instance()
    {
        static unique_ptr<Type> myInstance(new Type());

        return *myInstance;
    }

    Singleton() = delete;
    Singleton(const Singleton<Type>&) = delete;
    Singleton<Type>& operator=(const Singleton<Type>&) = delete;
};

class Product
{
public:
    Product() { cout << "Default constructor;" << endl; }
    ~Product() { cout << "Destructor;" << endl; }

    void f() { cout << "Method f;" << endl; }
};

int main()
{
    Product& d = Singleton<Product>::instance();

    d.f();
}
```

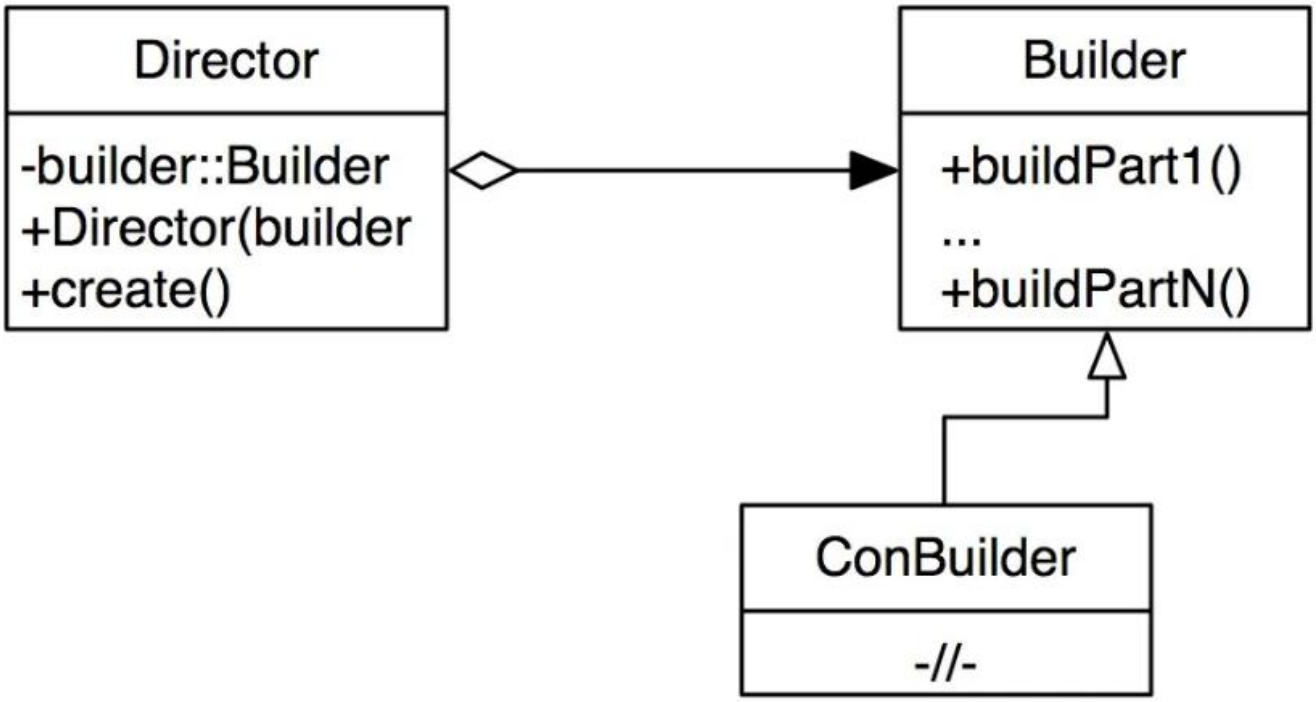
Строитель

Проблема: сложные объекты создаются поэтапно, иногда - этапы создания разнесены в разных частях программы.

Идея решения: вынести в отдельный код этапы создания сложных объектов.

- Строитель - класс, который включает в себя этапы создания сложного объекта.
- Кроме того, выделяем еще один класс, который контролирует создание - Директор.
- Строитель - создает объект.
- Директор - подготавливает данные для создания, контролирует этапы создания, отдает объект клиенту.

Диаграмма (должен быть ещё производный директор ConDirector):



- **Пример кода. Строитель**

```
# include <iostream>
# include <memory>

using namespace std;

class Product
{
public:
    Product() { cout << "Default constructor;" << endl; }
    ~Product() { cout << "Destructor;" << endl; }

    void run() { cout << "Method run;" << endl; }
};

class Builder
{
public:
    virtual bool buildPart1() = 0;
    virtual bool buildPart2() = 0;

    shared_ptr<Product> getProduct();

protected:
    virtual shared_ptr<Product> createProduct() = 0;

    shared_ptr<Product> product;
};

class ConBuilder : public Builder
{
public:
    virtual bool buildPart1() override { cout << "Completed part: " << ++part << ";" << endl; return true; }
    virtual bool buildPart2() override { cout << "Completed part: " << ++part << ";" << endl; return true; }
```

```
protected:
    virtual shared_ptr<Product> createProduct() override;

private:
    size_t part{0};
};

class Director
{
public:
    shared_ptr<Product> create(shared_ptr<Builder> builder)
    {
        if (builder->buildPart1() && builder->buildPart2()) return builder->getProduct();

        return shared_ptr<Product>();
    }
};

shared_ptr<Product> Builder::getProduct()
{
    if (!product) { product = createProduct(); }

    return product;
}

shared_ptr<Product> ConBuilder::createProduct()
{
    if (part == 2) { product = shared_ptr<Product>(new Product()); }

    return product;
}

int main()
{
    shared_ptr<Builder> builder(new ConBuilder());
    shared_ptr<Director> director(new Director());

    shared_ptr<Product> prod = director->create(builder);

    if (prod)
        prod->run();
}
```

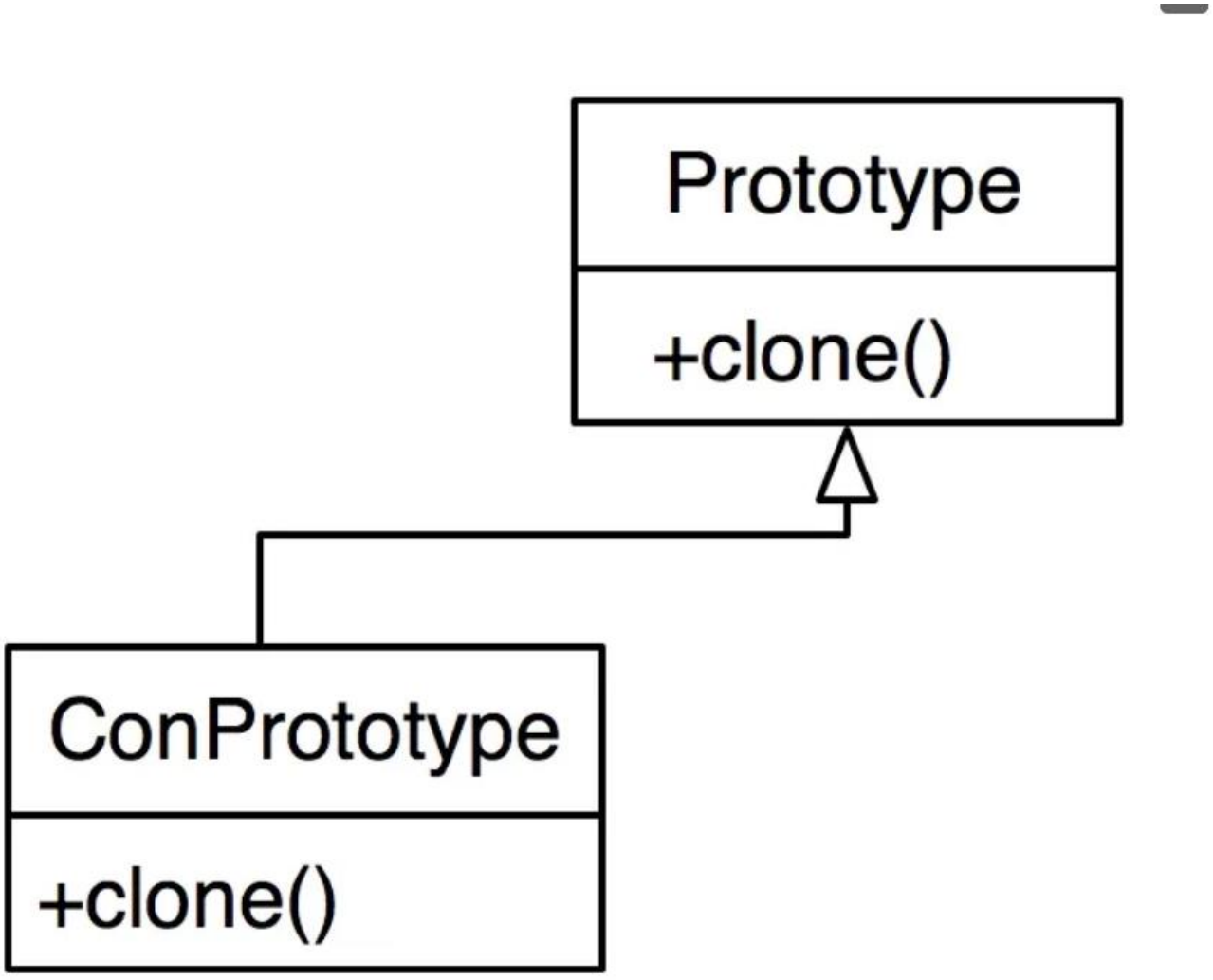
Как и для фабричного метода, должен быть solution, который принимает решение, какого директора создавать

- Когда надо использовать?
    1. Для поэтапного создания сложных объектов
    2. Когда создание объекта разнесено в коде, объект создается не сразу (например, данные подготавливаются поэтапно)
- Преимущество: вынесение создания и контроля в отдельные классы
- Проблема: с данными - конкретные строители базируются на одних и тех же входных данных и количество этапов строительства не меняется ⇒ возникнут проблемы при подмене одного строителя на другой).
- Решение: сделать агрегацию не на уровне базовых классов, а на уровне производных. Тогда можно будет менять интерфейс базового билдера (в том числе расширять или сужать), и

конкретный директор будет работать с конкретным билдером не как с базовым классом, а как с производным.

Прототип

- **Проблема:** Хотим создать копию объекта, не зная его класса. (Например, в подменяемом методе). Также мы не хотим тащить за ним его Creator'ы.
- **Решение:** Добавляем в базовый класс метод clone(), который создаст новый объект на основе существующего. Производные классы реализуют clone() под себя.



- **Пример кода. Prototype**

```
# include <iostream>
# include <memory>

using namespace std;

class BaseObject
{
public:
  virtual ~BaseObject() = default;

  virtual unique_ptr<BaseObject> clone() = 0;
};

class Object1 : public BaseObject
{
```

```
public:
  Object1() { cout << "Default constructor;" << endl; }
  Object1(const Object1& obj) { cout << "Copy constructor;" << endl; }
  ~Object1() { cout << "Destructor;" << endl; }

  virtual unique_ptr<BaseObject> clone() override
  {
          return unique_ptr<BaseObject>(new Object1(*this));
  }
};

int main()
{
  unique_ptr<BaseObject> ptr1(new Object1());

  auto ptr2 = ptr1->clone();
}
```

Преимущества: очень удобно использовать, когда объект сложно создаётся.

Пул объектов

Пул объектов предоставляет набор готовых объектов, которые мы можем использовать

Пример использования

Многопроцессорная система. Количество заданий, которое мы должны создать, должно быть не больше количества процессоров. Чтобы каждый процессор выполнял свою задачу.

Когда надо использовать?

Когда создание или уничтожение какого-либо объекта - трудоемкий процесс и надо "держать" определенное количество объектов в системе.

Задачи

1. Он держит эти объекты.
2. Может их создавать (то есть может расширяться).
3. По запросу отдает объект.
4. Если клиенту этот объект не нужен, он может его вернуть в пул.

Исходя из пунктов 3 и 4, для каждого включенного в пул объекта **мы должны установить, используется он или не используется.**

Возможна утечка информации. Мы взяли объект из пула, поработали, вернули в пул в том состоянии, в каком оставили – **его теперь надо или вернуть в исходное состояние, или очистить**, чтобы следующему клиенту не попала информация прошлого.

Пул объектов можно с помощью одиночки – чтобы пул объектов был только один. Пул объектов – контейнерный класс, для него используем итератор. **Необходимо знать, занят объект или нет, используем пару: ключ (bool) и объект.**

- **Диаграмма**

- **Пример кода. ObjectPool**

```
#include <iostream>
#include <iterator>
#include <memory>
#include <vector>

using namespace std;

class Product {
 private:
    static size_t count;

 public:
    Product() { cout << "Constructor(" << ++count << ");" << endl; }
    ~Product() { cout << "Destructor(" << count-- << ");" << endl; }

    void clear() { cout << "Method clear: 0x" << this << endl; }
};

size_t Product::count = 0;

template <typename Type>
class ObjectPool {
 public:
    static shared_ptr<ObjectPool<Type>>
    instance();  // статический метод из одиночки

    shared_ptr<Type> getObject();
    bool releaseObject(shared_ptr<Type>& obj);
    size_t count() const { return pool.size(); }

    iterator<output_iterator_tag, const pair<bool, shared_ptr<Type>>> begin()
        const;
    iterator<output_iterator_tag, const pair<bool, shared_ptr<Type>>> end()
        const;

    ObjectPool(const ObjectPool<Type>&) = delete;
    ObjectPool<Type>& operator=(const ObjectPool<Type>&) = delete;

 private:
    vector<pair<bool, shared_ptr<Type>>> pool;

    ObjectPool() {}

    pair<bool, shared_ptr<Type>> create();

    template <typename Type>
    friend ostream& operator<<(ostream& os, const ObjectPool<Type>& pl);
};

template <typename Type>
shared_ptr<ObjectPool<Type>> ObjectPool<Type>::instance() {
    static shared_ptr<ObjectPool<Type>> myInstance(new ObjectPool<Type>());

    return myInstance;
}

template <typename Type>
```

```
shared_ptr<Type> ObjectPool<Type>::getObject() {
    size_t i;
    for (i = 0; i < pool.size() && pool[i].first; ++i)
        ;

    if (i < pool.size()) {
        pool[i].first = true;
    } else {
        pool.push_back(create());
    }

    return pool[i].second;
}

template <typename Type>
bool ObjectPool<Type>::releaseObject(shared_ptr<Type>& obj) {
    size_t i;
    for (i = 0; i < pool.size() && pool[i].second != obj; ++i)
        ;

    if (i == pool.size()) return false;

    obj.reset();
    pool[i].first = false;
    pool[i].second->clear();

    return true;
}

template <typename Type>
pair<bool, shared_ptr<Type>> ObjectPool<Type>::create() {
    return pair<bool, shared_ptr<Type>>(true, shared_ptr<Type>(new Type()));
}

template <typename Type>
ostream& operator<<(ostream& os, const ObjectPool<Type>& pl) {
    for (auto elem : pl.pool)
        os << "{" << elem.first << ", 0x" << elem.second << "} ";

    return os;
}

int main() {
    shared_ptr<ObjectPool<Product>> pool = ObjectPool<Product>::instance();

    vector<shared_ptr<Product>> vec(4);

    for (auto& elem : vec) elem = pool->getObject();

    pool->releaseObject(vec[1]);

    cout << *pool << endl;

    shared_ptr<Product> ptr = pool->getObject();
    vec[1] = pool->getObject();

    cout << *pool << endl;
}
```

Минусы После использования объекта мы возвращаем его в пул, и здесь возможна так называемая **утечка информации**. Мы работали с этим объектом. Вернув его в пул, он *находится* в том состоянии, с которым мы с ним перед этим работали. Его надо либо вернуть в исходное состояние, либо очистить, чтобы **при отдаче этого объекта другому клиенту не произошла утечка информации. Пару комментариев**

- Пул объектов - контейнерный класс, удобно использовать **итераторы!**
- Необходимо знать, занят объект или нет, используем пару: ключ (bool) и объект.

14. Структурные паттерны: адаптер (Adapter), декоратор (Decorator), компоновщик (Composite), мост (Bridge), заместитель (Proxy), фасад (Facade). Их преимущества и недостатки.

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

Отличие шаблонов от паттернов: шаблон - конкретная реализация чего-либо, а паттерн - шаблон для решения какой-то задачи (как может решаться данная задача). Паттерн мы всегда адаптируем к своей задаче.

Преимущества использования паттернов:

- Мы имеем готовое решение
- За счет готового решения - нюансы все выявлены => надежный код
- Повышается скорость разработки
- Повышается читаемость кода
- Улучшается взаимодействие с коллегами (достаточно сказать название паттерна, который вы используете, и всё сразу станет понятно)

Структурные паттерны

Структурные паттерны предлагают структурные решения: определенную декомпозицию классов использованием схем наследования, включения и прочее.

Адаптер

Проблема: объект в разных местах программы играет разные роли. Это плохо, потому что:

- Для каждой роли разрабатывается свой интерфейс. Несколько ролей для одного объекта - значит, избыточный интерфейс.
- Роль объекта - это возложение на него определенной ответственности. Несколько ролей - это несколько ответственностей. Недопустимо с точки зрения принципов ООП.

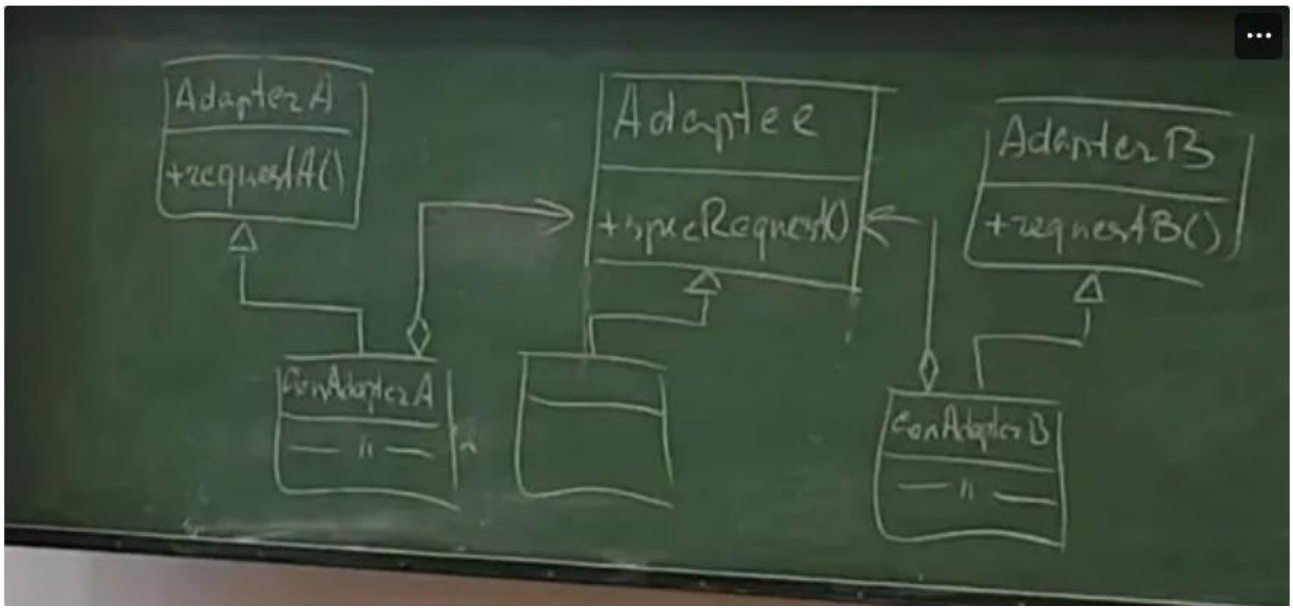
Идея решения: У объекта был один интерфейс. Подменяем этот интерфейс другим - соответственно той роли, в которой мы хотим использовать объект. Таким образом, в зависимости от ситуации мы можем использовать этот объект в разных ролях. С этим объектом работаем через объект другого класса. Объект, через класс которого мы работаем, имеет интерфейс необходимой нам роли.

Использование паттерна:

1. Один объект может выступать в нескольких ролях.
2. Нам нужно встроить в систему сторонние классы, имеющие другой интерфейс. Класс с любым интерфейсом можем встроить в нашу программу.
3. Мы, используя полиморфизм, сформировали интерфейс для базового. Определенные сущности, наследуемые от базового класса, должны поддерживать еще какой-то функционал. Мы не можем расширить этот функционал и изменять написанный код. Решаем эту проблему за счет адаптера, который предоставляет расширенный интерфейс.

Диаграмма:

Диаграмма:



- Пример кода:

```
# include <iostream>
# include <memory>

using namespace std;

class Adapter
{
public:
    virtual ~Adapter() = 0;

    virtual void request() = 0;
};

Adapter::~Adapter() = default;

class BaseAdaptee
{
public:
    virtual ~BaseAdaptee() = 0;

    virtual void specificRequest() = 0;
};

BaseAdaptee::~BaseAdaptee() = default;

class ConAdapter : public Adapter // подменяет интерфейс
{
private:
    shared_ptr<BaseAdaptee> adaptee;

public:
    ConAdapter(shared_ptr<BaseAdaptee> ad) : adaptee(ad) {}

    virtual void request() override;
```

```
};

class ConAdaptee : public BaseAdaptee
{
public:
    virtual void specificRequest() override { cout << "Method ConAdaptee;" << endl; }
};

#pragma region Methods
void ConAdapter::request()
{
    cout << "Adapter: ";

    if (adaptee)
    {
        adaptee->specificRequest();
    }
    else
    {
        cout << "Empty!" << endl;
    }
}

#pragma endregion

int main()
{
    shared_ptr<BaseAdaptee> adaptee(new ConAdaptee());
    shared_ptr<Adapter> adapter(new ConAdapter(adaptee));

    adapter->request();
}
```

Преимущества

1. Он позволяет нам класс с любым интерфейсом использовать в нашей программе.
2. Позволяет не создавать нам классы с несколькими ответственностями. Разносим это по другим классам.
3. Позволяет расширять интерфейс класса

Есть мини проблемка:

- Интерфейсы могут пересекаться

Декоратор

Проблема: нам надо добавить/подменить классам функционал. Причем одинаковый для нескольких классов. Если мы подменим функционал в производных для каждого из классов, которые мы хотим изменить - разрастается иерархия, приходится дублировать код.

- **Картинка с разрастанием иерархии(*должна быть стрелочка от базового к B*)**

Что сразу бросается в глаза, когда мы добавляем *одно и то же*

1. У нас будет повторяющийся код. Мы должны с этим бороться .
2. Это приводит к резкому разрастанию всей иерархии наследования. Могут быть другие добавления, а к ним еще.

Идея решения: вынести это добавление в отдельный класс - декоратор.

Использование паттерна: добавление/подмена небольшой части функционала, одинаковой для разных классов.

Преимущества

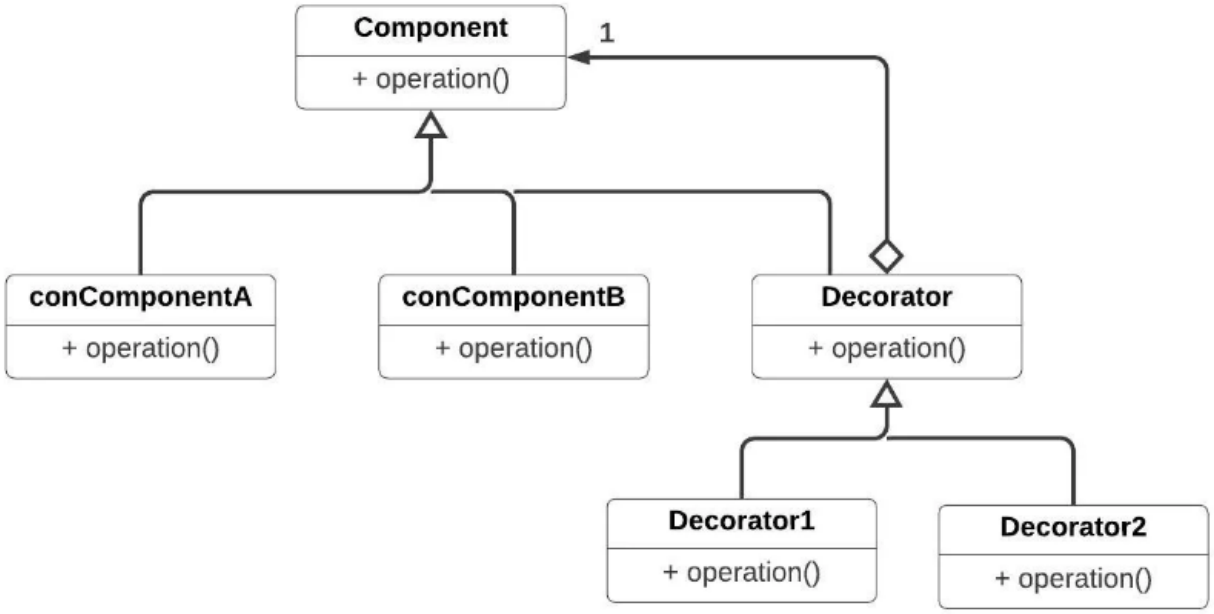
1. Мы получаем крайне гибкую систему, избавляясь от колоссального количества классов. Резко уменьшается иерархия.
2. Декорировать можем во время выполнения работы программы.
3. Избавляемся от дублирования кода. Этот код уходит в конкретный декоратор, не дублируясь.

Недостатки

- Когда мы сделали сложную обертку, нам, к сожалению, убрать какую-то обёртку будет проблематично. Нам придется **заново** создавать компонент с обёртками (должен быть ответственный за эту сборку). Мы не можем её вычеркнуть.
- Вызов кучи виртуальных методов замедляет программу.

Диаграмма:

Диаграмма:



- Пример кода
- `# include <iostream>`
- `# include <memory>`
-
- `using namespace std;`
-

```
class Component
{
public:
    virtual ~Component() = 0;

    virtual void operation() = 0;
};

Component::~Component() = default;

class ConComponent : public Component
{
public:
    virtual void operation() override { cout << "ConComponent"; }
};

class Decorator : public Component
{
protected:
    shared_ptr<Component> component;

public:
    Decorator(shared_ptr<Component> comp)
        : component(comp) {}
};

class ConDecorator : public Decorator
{
public:
    using Decorator::Decorator;

    virtual void operation() override;
};

#pragma region Method
void ConDecorator::operation()
{
    if (component)
    {
        component->operation();
        cout << "ConDecorator" << endl;
    }
}

#pragma endregion

int main()
{
    shared_ptr<Component> component(new ConComponent());
    shared_ptr<Component> decorator1(new ConDecorator(component));

    decorator1->operation();

    shared_ptr<Component> decorator2(new ConDecorator(decorator1));

    decorator2->operation();
}
```

Компоновщик

Введение

Очень часто мы работаем со многими объектами однотипно, то есть выполняем над ними одни и те же операции. Более того, возникают ситуации, когда нужно выполнять над группой объектов эти действия совместно.

Пример

Есть 3D модель, которую мы можем поворачивать, переносить. А если моделей несколько? У нас возникает необходимость объединять эти 3D модели в одну - сделать сборку и выполнять уже над этой сборкой совместные действия. У нас на сцене объектов может быть много, и естественно **хотелось бы рассматривать и каждый в отдельности объект, и совместно**.

Идея

Вынести интерфейс, который предлагает контейнер (объект, включающий в себя другие объекты), на уровень базового класса

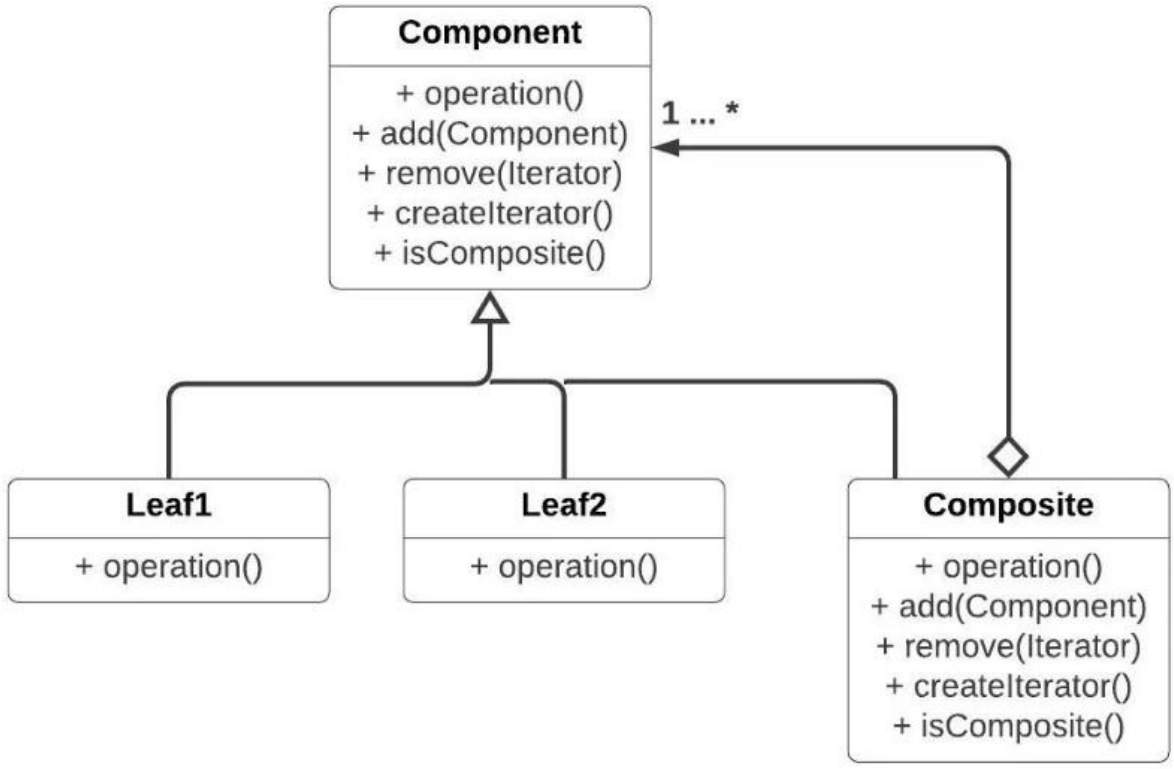
Компоновщик компонует объекты в древовидную структуру, в которой над всей иерархией объектов можно выполнять такие же действия, как над простым элементом или над его частью.

Почему древовидная структура?

Каждый композит может включать в себя как простые компоненты, так и другие композиты. Получается древовидная структура. Системе неважно, композит это или не композит. Она работает с любым элементом.

Задача методов интерфейса для компонентов - пройтись по списку компонент и выполнить соответствующий метод.

Диаграмма:



1. Базовый класс - Component. Нам должно быть безразлично, с каким объектом мы работаем: то ли это один компонент, то ли это объект, включающий в себя другие объекты (контейнер). Если это контейнер, то нам надо работать с содержимым контейнера, удалять, добавлять в него объекты. **Идея - вынести этот интерфейс на уровень базового класса** (добавление компонента - add(Component), удаление компонента - remove(Iterator), createIterator()). Нам надо четко понимать, когда мы работаем с каким-то компонентом, чем он является: один объект или контейнер. Для этого нам нужен метод isComposite(). То, что мы можем делать с Component - operation() - чисто виртуальные методы. Остальные (add, remove, и т. д.) мы будем реализовывать.
2. Leaf - простой компонент, его задачей будет только реализовать остальные методы - operation.
3. Composite - контейнерный класс. Реализует все те методы, что есть в компоненте. Он содержит в себе список компонент.

- **Пример кода**

```
# include <iostream>
# include <memory>
# include <vector>
# include <iterator>

using namespace std;

class Component;

using VectorComponent = vector<shared_ptr<Component>>;
using IteratorComponent = VectorComponent::const_iterator;

class Component
{
```

```
public:
  virtual ~Component() = 0;

  virtual void operation() = 0;

  virtual bool isComposite() const { return false; }
  virtual bool add(shared_ptr<Component> comp) { return false; }
  virtual bool remove(const IteratorComponent& it) { return false; }
  virtual IteratorComponent begin() const { return IteratorComponent(); }
  virtual IteratorComponent end() const { return IteratorComponent(); }
};

Component::~Component() {}

class Figure : public Component
{
public:
  virtual void operation() override { cout << "Figure method;" << endl; }
};

class Camera : public Component
{
public:
  virtual void operation() override { cout << "Camera method;" << endl; }
};

class Composite : public Component
{
private:
  VectorComponent vec;

public:
  Composite() = default;
  Composite(shared_ptr<Component> first, ...);

  virtual void operation() override;

  virtual bool isComposite() const override { return true; }
  virtual bool add(shared_ptr<Component> comp) { vec.push_back(comp); return
false; }
  virtual bool remove(const IteratorComponent& it) { vec.erase(it); return false;
}
  virtual IteratorComponent begin() const override { return vec.begin(); }
  virtual IteratorComponent end() const override { return vec.end(); }
};

Composite::Composite(shared_ptr<Component> first, ...)
{
  for (shared_ptr<Component>* ptr = &first; *ptr; ++ptr)
          vec.push_back(*ptr);
}

void Composite::operation()
{
  cout << "Composite method:" << endl;
  for (auto elem : vec)
          elem->operation();
}

```

```
int main()
{
  using Default = shared_ptr<Component>;
  shared_ptr<Component> fig1(new Figure()), fig2(new Figure), cam1(new Camera()), cam2(new Camera());
  shared_ptr<Component> composite1(new Composite(fig1, cam1, fig2, cam2, Default()));

  composite1->operation();
  cout << endl;

  IteratorComponent it = composite1->begin();

  composite1->remove(++it);
  composite1->operation();
  cout << endl;

  shared_ptr<Component> composite2(new Composite(shared_ptr<Component>(new Figure()), composite1, Default()));

  composite2->operation();
}
```

Заместитель(proxy)

Идея

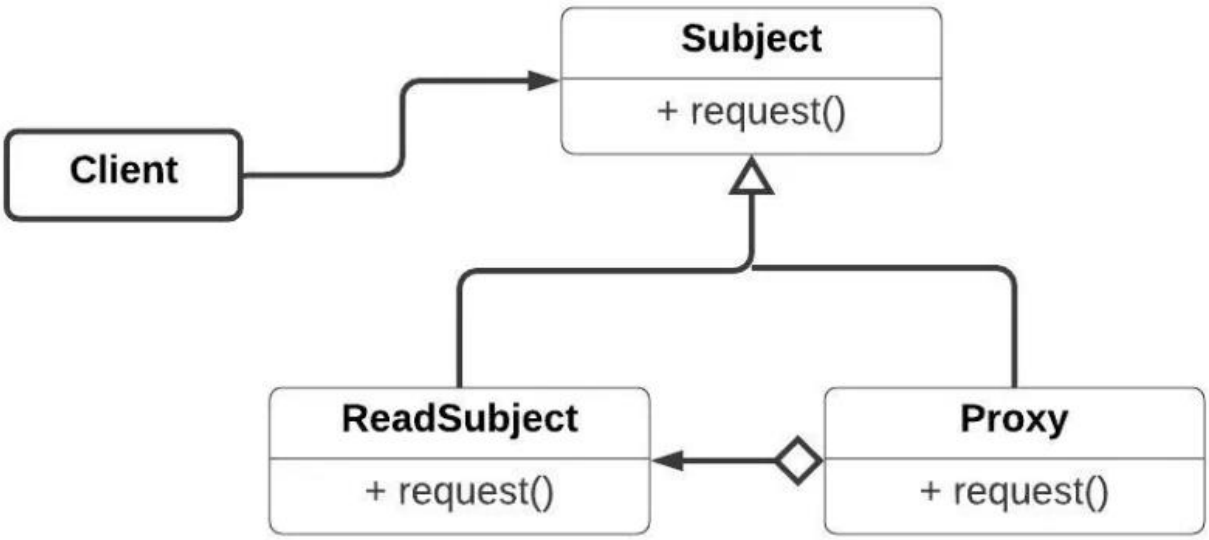
Заместитель (или Proxy) позволяет нам работать не с реальным объектом, а с другим объектом, который подменяет реальный. В каких целях это можно делать?

1. **Подменяющий объект может контролировать другой объект, задавать правила доступа к этому объекту**. Например, у нас есть разные категории пользователей. В зависимости от того, какая у пользователя категория, определять, давать доступ к самому объекту или не давать. *Это как защита*.
2. **Так как запросы проходят через заместителя, он может контролировать запросы, заниматься статистической обработкой**: считать количество запросов, какие запросы были и так далее.
3. **Разгрузка объекта с точки зрения запросов.** Дело в том, что реальные объекты какие-то операции могут выполнять крайне долго, например, обращение "поиск в базе чего-либо" или "обращение по сети куда-то". Это выполняется долго. **Proxy может сохранять предыдущий ответ и при следующем обращении смотреть, был ответ на этот запрос или не был**. Если ответ на этот вопрос был, он не обращается к самому хозяину, он заменяет его тем ответом, который был ранее. Естественно, если состояние объекта изменилось, Proxy должен сбросить ту историю, которую он накопил, чтобы выдавать только актуальную информацию.

Это очень удобно, когда у нас тяжелые объекты, операции которых выполняются долго. Зачем еще раз спрашивать, если мы уже спрашивали об этом? Proxy может выдать нам этот ответ.

Диаграмма. Опечатка, должен быть *RealSubject*

Диаграмма. Опечатка, должен быть *RealSubject*



Базовый класс Subject, реальный объект RealObject и объект Proxy, который содержит ссылку на объект, который он замещает. Когда мы работаем через указатель на базовый объект Subject, мы даже не можем понять, с кем мы реально работаем: с непосредственно объектом RealSubject или с его заместителем Proxy. А заместитель может выполнять те задачи, которые мы на него возложили.

Если состояние RealObject изменилось, Прокси должен сбросить историю, которую он накопил

- **Пример кода(Очень много строк, не стоит наверное писать на экзамене)**

```
# include <iostream>
# include <memory>
# include <map>
# include <random>

using namespace std;

class Subject
{
public:
    virtual ~Subject() = 0;

    virtual pair<bool, double> request(size_t index) = 0;
    virtual bool changed() { return true; }
};

Subject::~Subject() = default;

class RealSubject : public Subject
{
private:
    bool flag{ false };
    size_t counter{ 0 };

public:
    virtual pair<bool, double> request(size_t index) override;
    virtual bool changed() override;
```

```
};

class Proxy : public Subject
{
protected:
    shared_ptr<RealSubject> realsubject;

public:
    Proxy(shared_ptr<RealSubject> real) : realsubject(real) {}
};

class ConProxy : public Proxy
{
private:
    map<size_t, double> cache;

public:
    using Proxy::Proxy;

    virtual pair<bool, double> request(size_t index) override;
};

#pragma region Methods
bool RealSubject::changed()
{
    if (counter == 0)
    {
        flag = true;
    }
    if (++counter == 7)
    {
        counter = 0;
        flag = false;
    }
    return flag;
}

pair<bool, double> RealSubject::request(size_t index)
{
    random_device rd;
    mt19937 gen(rd());

    return pair<bool, double>(true, generate_canonical<double, 10>(gen));
}

pair<bool, double> ConProxy::request(size_t index)
{
    pair<bool, double> result;

    if (!realsubject)
    {
        cache.clear();

        result = pair<bool, double>(false, 0.);
    }
    if (!realsubject->changed())
    {
        cache.clear();
```

```
        result = realsubject->request(index);

        cache.insert(map<size_t, double>::value_type(index, result.second));
    }
    else
    {
        map<size_t, double>::const_iterator it = cache.find(index);

        if (it != cache.end())
        {
            result = pair<bool, double>(true, it->second);
        }
        else
        {
            result = realsubject->request(index);

            cache.insert(map<size_t, double>::value_type(index, result.second));
        }
    }

    return result;
}

#pragma endregion

int main()
{
    shared_ptr<RealSubject> subject(new RealSubject());
    shared_ptr<Subject> proxy(new ConProxy(subject));

    for (size_t i = 0; i < 21; ++i)
    {
        cout << "( " << i + 1 << ", " << proxy->request(i % 3).second << " )" << endl;

        if ((i + 1) % 3 == 0)
            cout << endl;
    }

}
```

Пример кода(Покороче, стоит писать)

```
#include <memory>
#include <iostream>
using namespace std;

class BaseSubject
{
public:
    BaseSubject() = default;
    virtual void get_size() = 0;
    virtual string render() = 0;
};

class RealSubject : public BaseSubject
{
public:
    RealSubject() { cout << "real subject constructed" << endl; };
    virtual void get_size() {};
```

```
    virtual string render() { cout << "real: render" << endl; return "render"; };
};

class Proxy : public BaseSubject
{
    shared_ptr<RealSubject> rs;
    shared_ptr<string> r;
public:
    Proxy() { cout << "proxy constructed" << endl; };
    Proxy(shared_ptr<RealSubject> rs) : rs(rs) {};
    void get_size()
    {
        cout << "Proxy: size is 3" << endl;
    }
    string render()
    {
        if (rs)
        {
            if (r)
                cout << "Proxy: " << *r << endl;
            else
                r = make_shared<string>(rs->render());
        }
        else
        {
            rs = make_shared<RealSubject>();
            r = make_shared<string>(rs->render());
        }
        return *r;
    }
};

int main()
{
    shared_ptr<Proxy> pr = make_shared<Proxy>();
    pr->get_size();
    pr->render();
    pr->render();
}
```

Плюсы:

- Дает возможность контролировать объект незаметно для клиента
- Может работать тогда, когда объекта нет (пример: ответ от прокси, что объект устаревший)
- Прокси может отвечать за жизненный цикл объекта (может создавать и удалять его)

Минусы:

- Время доступа увеличивается (обработка идет через прокси)
- Прокси должен хранить историю (влияет на память)

Мост

Идея

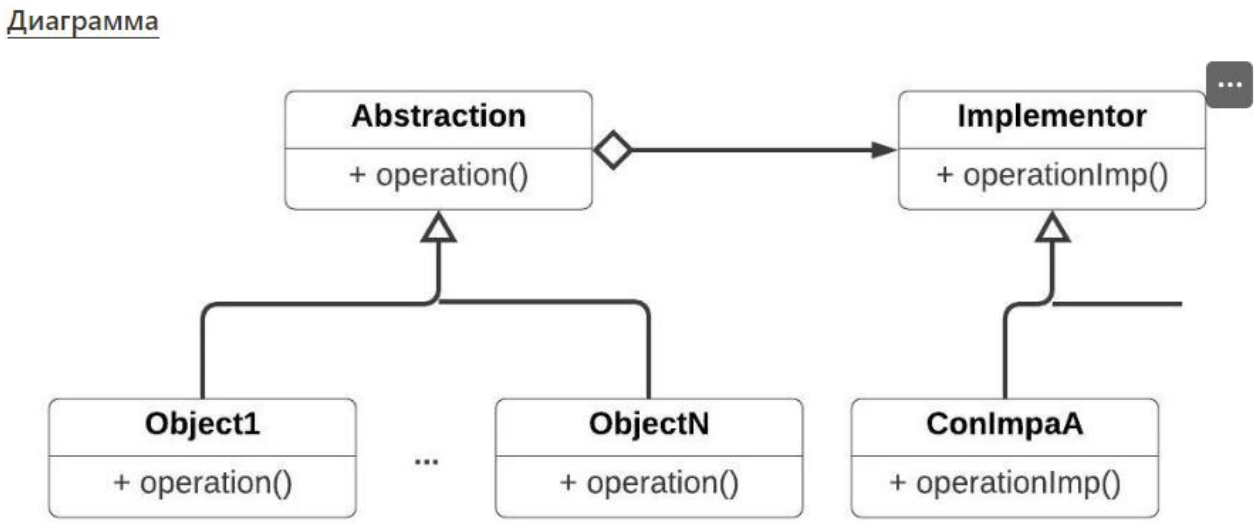
Имеются следующие проблемы:

1. У нас большая иерархия. В иерархии возможно несколько внутренних реализаций для объекта, это разные классы, разные ветви. У нас один объект, и во время работы надо поменять реализацию. Каким-то образом нам надо мигрировать от одного класса к другому.
2. Постоянно происходит дублирование кода, когда у нас разрастается иерархия классов.

Было предложено **разделить понятие самого объекта, его сущности и реализации, в отдельные иерархии**. Таким образом, мы сократим количество классов и сделаем систему более гибкой. Мы во время работы сможем менять реализацию. Мы можем уйти (не полностью, частично) от повторного кода.

Паттерн Мост (или Bridge) отделяет саму абстракцию, сущность, от реализаций. Мы можем независимо менять логику (сами абстракции) и наращивать реализацию (добавлять новые классы реализации).

Диаграмма



- **Пример кода. Bridge**

```
# include <iostream>
# include <memory>

using namespace std;

class Implementor
{
public:
    virtual ~Implementor() = 0;

    virtual void operationImp() = 0;
};

Implementor::~Implementor() = default;

class Abstraction
{
protected:
    shared_ptr<Implementor> implementor;

public:
```

```
    Abstraction(shared_ptr<Implementor> imp) : implementor(imp) {}
    virtual ~Abstraction() = 0;

    virtual void operation() = 0;
};

Abstraction::~Abstraction() = default;

class ConImplementor : public Implementor
{
public:
    virtual void operationImp() override { cout << "Implementor;" << endl; }
};

class Entity : public Abstraction
{
public:
    using Abstraction::Abstraction;

    virtual void operation() override { cout << "Entity: "; implementor->operationImp(); }
};

int main()
{
    shared_ptr<Implementor> implementor(new ConImplementor());
    shared_ptr<Abstraction> abstraction(new Entity(implementor));

    abstraction->operation();
}
```

Таким образом, для каждого объекта мы можем менять реализацию динамически в любой момент во время выполнения. Как логика самого объекта может меняться, так и реализация может меняться. Мы независимо от сущности добавляем реализации и наоборот.

Схема гибкая.

Возникает проблема выноса связи на уровень базового класса. Есть проблема, не всегда это удается сделать.

Но если удается, то это великолепно.

Использование

Используем, когда:

1. Нам нужно во время выполнения менять реализацию
2. Когда у нас большая иерархия, и по разным ветвям этой иерархии идут одинаковые реализации. **Дублирование кода мы выносим в дерево реализаций**. Такой подход дает возможность независимо изменять управляющую логику и реализацию.

Фасад

Идея

У нас есть группа объектов, связанных между собой. Причем эти связи *довольно жёсткие*. Чтобы извне не работать с каждым объектом в отдельности, мы можем эти все объекты объединить в один класс - фасад, который будет представлять интерфейс для работы со всем этим объединением.

Нам не нужно извне работать с мелкими объектами. Кроме того, фасад может выполнять такую роль, как следить за целостностью нашей системы. Извне, мы, работая с фасадом, работаем, как с простым объектом. Он представляет интерфейс одной сущностью, а внутри у нас целый мир из объектов.

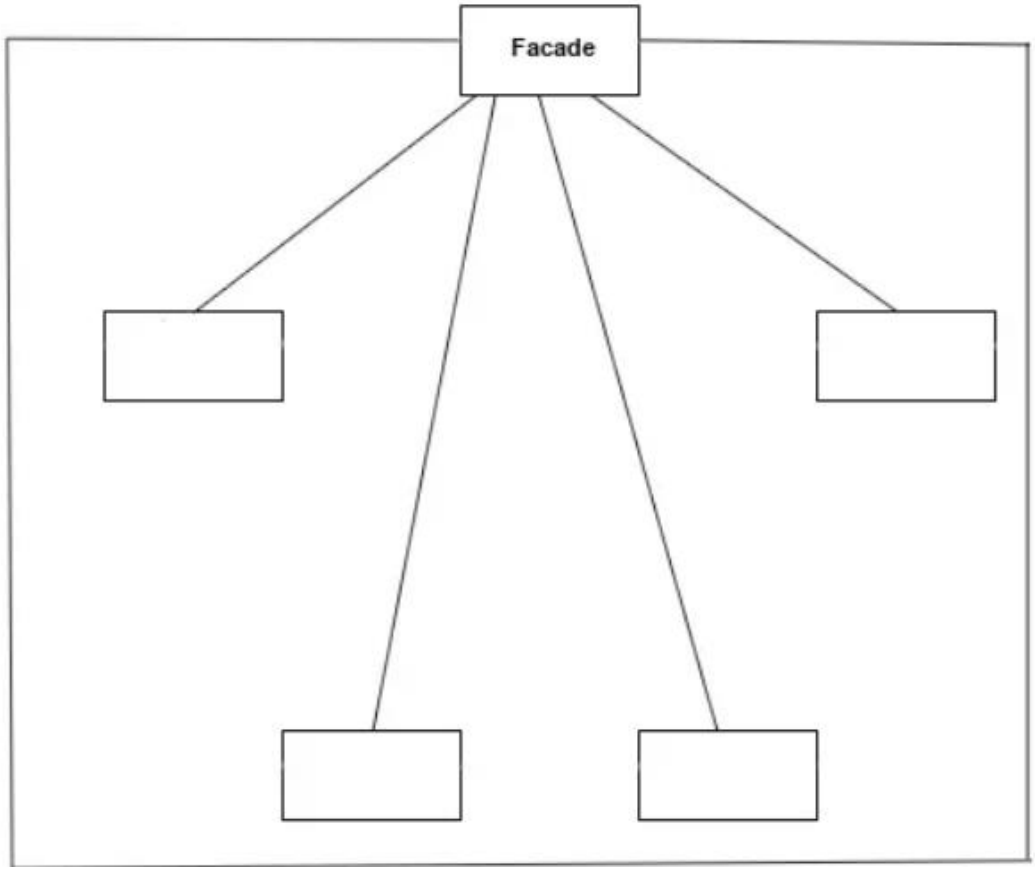
Таким образом:

- упрощается взаимодействие
- уменьшается количество связей за счет объединений фасада.

Это для нас очень важно. При такой организации клиенту не нужно знать и уметь работать с этими объектами. Ему достаточно уметь работать с фасадом.

Диаграмма

Диаграмма



15. **Паттерны поведения: стратегия (Strategy), команда (Command), цепочка обязанностей (Chain of Responsibility), подписчик-издатель (Publish-Subscribe), посредник (Mediator). Их преимущества и недостатки.**

    ОТВЕТ: Паттерны поведения: стратегия (Strategy), команда (Command), цепочка обязанностей (Chain of Responsibility), подписчик-издатель (Publish-Subscribe), посредник (Mediator),

Отличие шаблонов от паттернов: шаблон - конкретная реализация чего-либо, а паттерн - шаблон для решения какой-то задачи (как может решаться данная задача). Паттерн мы всегда адаптируем к своей задаче.

Преимущества использования паттернов:

- Мы имеем готовое решение
- За счет готового решения - нюансы все выявлены => надежный код
- Повышается скорость разработки
- Повышается читаемость кода
- Улучшается взаимодействие с коллегами (достаточно сказать название паттерна, который вы используете, и всё сразу станет понятно)

Стратегия

Идея

Нам во время выполнения надо менять реализацию какого-либо метода. Мы можем делать производные классы с разными реализациями и осуществлять "миграцию" между классами во врем выполнения - это неудобно, ибо мы начинаем работать с конкретными типами (классами)

То, что может меняться (это действие) вынести в отдельный класс, который будет выполнять **только это действие**. Мы можем во время выполнения подменять одно на другое.

Паттерн Стратегия определяет семейство всевозможных алгоритмов.

Плюсы**:**

- Возможность подмены алгоритма
- Отделение сущности от реализации
- Ограничение разрастания иерархии
- Разъеб(уменьшение) дублирования кода
- Позволяет добавлять функционал классу

Минусы:

- Не всегда можем свести к базовой, реализация может быть разная, а мы хотим подменять, алгоритм базируется на обработке данных, а данные могут быть разные
- **Диаграмма(Лек 14_1 14мин):**

Клиент может установить для нашего класса конкретную стратегию (алгоритм) и, работая с объектом, он будет вызывать этот конкретный алгоритм. Во время работы мы **можем этот алгоритм поменять**.

Стратегия это вырожденный паттерн МОСТ, только выносим не всю реализацию, а лишь часть - метод

ПИШЕМ ЛЮБОЙ

- **Пример кода 1 Держим указатель. Стратегия (Strategy).**

  Объект держит указатель на стратегию. Мы один раз установили стратегию, можем, конечно, её поменять. Вызывая для объекта, вызывается та стратегия, которая нас интересует.

  - Код

```
# include <iostream>
# include <memory>
# include <vector>

using namespace std;

class Strategy
{
public:
    virtual ~Strategy() = 0;

    virtual void algorithm() = 0;
};

Strategy::~Strategy() = default;

class ConStrategy1 : public Strategy
{
public:
    virtual void algorithm() override { cout << "Algorithm 1;" << endl; }
};

class ConStrategy2 : public Strategy
{
public:
    virtual void algorithm() override { cout << "Algorithm 2;" << endl; }
};

class Context
{
private:
    unique_ptr<Strategy> strategy;

public:
    explicit Context(Strategy* ptr) : strategy(ptr) {}

    void algorithmStrategy() { strategy->algorithm(); }
};

int main()
{
    Context obj(new ConStrategy1());

    obj.algorithmStrategy();
}
```

- **Пример кода 2 Передаем стратегию. Стратегия (Strategy).**

  При выполнении мы передаем, какую стратегию хотим использовать. Мы не держим указатель, а устанавливаем при работе.

  - Код

```
# include <iostream>
# include <memory>
```

```
# include <vector>

using namespace std;

class Strategy
{
public:
    virtual ~Strategy() = 0;

    virtual void algorithm() = 0;
};

Strategy::~Strategy() = default;

class ConStrategy1 : public Strategy
{
public:
    virtual void algorithm() override { cout << "Algorithm 1;" << endl; }
};

class ConStrategy2 : public Strategy
{
public:
    virtual void algorithm() override { cout << "Algorithm 2;" << endl; }
};

class Context
{
public:

    void algorithmStrategy(shared_ptr<Strategy> strategy) { strategy-
>algorithm(); }
};

int main()
{
    Context obj;

    obj.algorithmStrategy(shared_ptr<Strategy>(new ConStrategy1()));

    obj.algorithmStrategy(shared_ptr<Strategy>(new ConStrategy2()));
}
```

- **Пример кода 3 Статический полиморфизм. Стратегия (Strategy).**

  Вариант со статическим полиморфизмом. Статический полиморфизм - на этапе компиляции происходит связывание, не можем выбрать на этапе выполнения.

  Единственный плюс этого варианта - быстрее.

  Код

```
# include <iostream>
# include <memory>
# include <vector>

using namespace std;

class Strategy
{
public:
    virtual ~Strategy() = 0;

    virtual void algorithm() = 0;
};

Strategy::~Strategy() = default;

```

```
class ConStrategy1 : public Strategy
{
public:
    virtual void algorithm() override { cout << "Algorithm 1;" << endl; }
};

class ConStrategy2 : public Strategy
{
public:
    virtual void algorithm() override { cout << "Algorithm 2;" << endl; }
};

template <typename CStrategy>
class Context
{
private:
    unique_ptr<CStrategy> strategy;

public:
    Context() : strategy(new CStrategy()) {}

    void algorithmStrategy() { strategy->algorithm(); }
};

int main()
{
    Context<ConStrategy1> obj;

    obj.algorithmStrategy();
}
```

Команда

Идея

Возможны разные запросы (загрузить, повернуть, перенести и подобное). Идея такая: обернуть каждый запрос в отдельный класс (класс может быть как простой, так и составной). Если мы чётко знаем, кто должен обработать запрос, то надо держать объект (обработчик) и указатель на метод, и их можно передавать тому, кто это обработает.

Диаграмма

Запрос, или команда, идет в виде объекта. Базовый класс - и у нас может быть не одна команда, а несколько, в зависимости от того, что нам нужно. Команда может нести данные, может нести *что надо сделать*, (указатель на метод какого-либо объекта, например.)

- **UML**

- **Пример кода. Команда (Command).**

```
# include <iostream>
# include <memory>

using namespace std;

class Command
{
public:
    virtual ~Command() = default;
```

```
    virtual void execute() = 0;
};

template <typename Reseiver>
class SimpleCommand : public Command
{
private:
    typedef void(Reseiver::*Action)();

    shared_ptr<Reseiver> reseiver;
    Action action;

public:
    SimpleCommand(shared_ptr<Reseiver> r, Action a) : reseiver(r), action(a) {}

    virtual void execute() override { ((*reseiver).*action)(); }
};

class Object
{
public:
    void run() { cout << "Run method;" << endl; }
};

int main()
{
    shared_ptr<Object> obj(new Object());
    shared_ptr<Command> command(new SimpleCommand<Object>(obj, &Object::run));

    command->execute();
}
```

Преимущества

1. Унификация обработки запросов к системе (событий)
2. Уменьшается зависимость между объектами, не нужно держать связь
3. Команду можем выполнить не сразу, а через время, можно сформировать очередь
4. Если добавить к команде композит, то можно формировать сложные команды из нескольких команд
    - UML

Цепочка обязанностей

Часто задача сводится к тому, что запрос должны обрабатывать несколько объектов, а не один.

- Идея: создать цепочку обработчиков.
- Преимущества: позволяет передавать запрос последовательно по цепочке обработчиков, каждый обработчик сам решает, передавать дальше или нет.
- **Диаграмма**

Handler определяет общий интерфейс и задает механизм передачи запроса, а каждый Handler1..n содержат код обработки запросов

- **Пример кода. Цепочка обязанностей (Chain of Responsibility).**

  Рекурсивный список

  > В примерах – идем по цепочке, пока не наткнемся на обработчик с состоянием true, дальше по цепочке не пойдем.

```
# include <iostream>
# include <memory>

using namespace std;

class AbstractHandler
{
protected:
    shared_ptr<AbstractHandler> next;

    virtual bool run() = 0;

public:
    using Default = shared_ptr<AbstractHandler>;

    virtual ~AbstractHandler() = default;

    virtual bool handle() = 0;

    void add(shared_ptr<AbstractHandler> node);
    void add(initializer_list<shared_ptr<AbstractHandler>> list);
};

class ConHandler : public AbstractHandler
{
private:
    bool condition{ false };

protected:
    virtual bool run() override { cout << "Method run;" << endl; return true; }

public:
    ConHandler() = default;
    ConHandler(bool c) : condition(c) { cout << "Constructor;" << endl; }
    virtual ~ConHandler() override { cout << "Destructor;" << endl; }

    virtual bool handle() override
    {
        if (!condition) return next ? next->handle() : false;

        return run();
    }

};

void AbstractHandler::add(shared_ptr<AbstractHandler> node)
{
    if (next)
        next->add(node);
    else
        next = node;
}

void AbstractHandler::add(initializer_list<shared_ptr<AbstractHandler>> list)
{
```

```
            for (auto elem : list)
                    add(elem);
    }

    int main()
    {
        shared_ptr<AbstractHandler> chain(new ConHandler());

        chain->add({
            shared_ptr<AbstractHandler>(new ConHandler(false)),
            shared_ptr<AbstractHandler>(new ConHandler(true)),
            shared_ptr<AbstractHandler>(new ConHandler(true)),
            AbstractHandler::Default()}
        );

        cout << "Result = " << chain->handle() << ";" << endl;;
    }
```

Когда надо использовать?

- один и тот же запрос может выполняться разными способами
- есть четкая последовательность в обработчиках (можем передавать до тех пор, пока не обработается, но важна последовательность)
- на этапе выполнения мы решаем, какие объекты будут в этой цепочке

Подписчик-издатель

Часто нам надо какой-либо запрос передавать не одному, а многим объектам, и это должно выполняться на этапе выполнения

Этот паттерн задает механизм: тот, кто хочет принять сообщение, тот подписывается у издателя (подписался - получил, не подписался - не получил)

Группа объектов реагирует на один объект, на его изменения, издатель оповещает всех подписчиков, когда происходят изменения, вызывая их методы

- **Диаграмма (у Publisher еще один метод - `unsubscribe()`)**

- **Пример кода. Синхронно**
- `#include <iostream>`
- `#include <memory>`
- `#include <vector>`
-
- `using namespace std;`
-
- `class Subscriber;`
-
- `using Reseiver = Subscriber;`
-
- `class Publisher {`
- `    using Action = void (Reseiver::*)();`
- `    using Pair = pair<shared_ptr<Reseiver>, Action>;`
-
- `private:`
- `    vector<Pair> callback; // храним пару подписчик-метод`
-

```
    int indexOf(shared_ptr<Reseiver> r);

 public:
    bool subscribe(shared_ptr<Reseiver> r, Action a);
    bool unsubscribe(shared_ptr<Reseiver> r);
    void run();
};

class Subscriber {
 public:
    virtual ~Subscriber() = default;

    virtual void method() = 0;
};

class ConSubscriber : public Subscriber {
 public:
    virtual void method() override { cout << "method;" << endl; }
};

#pragma region Methods Publisher
bool Publisher::subscribe(shared_ptr<Reseiver> r, Action a) {
    if (indexOf(r) != -1) return false;

    Pair pr(r, a);

    callback.push_back(pr);

    return true;
}

bool Publisher::unsubscribe(shared_ptr<Reseiver> r) {
    int pos = indexOf(r);

    if (pos != -1) callback.erase(callback.begin() + pos);

    return pos != -1;
}

void Publisher::run() {
    cout << "Run:" << endl;
    for (auto elem : callback) ((*elem.first).*(elem.second))();
}

int Publisher::indexOf(shared_ptr<Reseiver> r) {
    int i = 0;
    for (auto it = callback.begin(); it != callback.end() && r != (*it).first;
         i++, ++it)
        ;

    return i < callback.size() ? i : -1;
}
#pragma endregion

int main() {
    shared_ptr<Subscriber> subscriber1 = make_shared<ConSubscriber>();
    shared_ptr<Subscriber> subscriber2 = make_shared<ConSubscriber>();
    shared_ptr<Publisher> publisher = make_shared<Publisher>();
```

```
    publisher->subscribe(subscriber1, &Subscriber::method);
    if (publisher->subscribe(subscriber2, &Subscriber::method))
        publisher->unsubscribe(subscriber1);

    publisher->run();
}
```

Пример кода. Асинхронно.

```
# include <iostream>
# include <string>
# include <QObject>

using namespace std;

class Employer : public QObject // Publisher
{
    Q_OBJECT

private:
    const int day_salary = 15;
    const int day_advance = 25;

public:
    Employer() = default;
    void notifyEmployeis(int day);

signals:
    void salaryPayment();
    void taskIssuance();
};

# pragma region Method Publisher
void Employer::notifyEmployeis(int day)
{
    if (day == day_salary || day == day_advance)
    {
        emit salaryPayment();
    }
    else
    {
        emit taskIssuance();
    }
}

# pragma endregion

enum class Mood
{
    happy,
    sad
};

class Employee : public QObject // Subscribe
{
    Q_OBJECT

private:
    string name;
```

```
    Mood mood;

public:
    Employee(string name) : name(name), mood(Mood::sad) {}

    string getName() { return name; }
  Mood getMood() { return mood; }

    void subscribeOnEmployer(const Employer* ptrEmployer);
    void unsubscribeFromEmployer(const Employer* ptrEmployer);

public slots:
    void onSalaryPayment() { mood = Mood::happy; }
    void onTaskIssuance() { mood = Mood::sad; }
};

# pragma region Methods Subscribe
void Employee::subscribeOnEmployer(const Employer* ptrEmployer)
{
    QObject::connect(ptrEmployer, SIGNAL(salaryPayment()), this,
SLOT(onSalaryPayment()));
    QObject::connect(ptrEmployer, SIGNAL(taskIssuance()), this,
SLOT(onTaskIssuance()));
}

void Employee::unsubscribeFromEmployer(const Employer* ptrEmployer)
{
    QObject::disconnect(ptrEmployer, SIGNAL(salaryPayment()), this,
SLOT(onSalaryPayment()));
    QObject::disconnect(ptrEmployer, SIGNAL(taskIssuance()), this,
SLOT(onTaskIssuance()));
}

# pragma endregion

ostream& operator<< (ostream &out, const Employee &employee)
{
    string mood = employee.getMood() == Mood::happy ?
                "Happy with a lot of money!!!!))))" :
                "Sad with a lot of work....((((";

    return out << "Name: " << employee.getName() << ", Mood: " << mood << endl;
}

int main()
{
    //Создадим работодателя, который выступает в роли объекта publisher
    Employer google;

    //Создадим несколько работников, которые выступают в роли объекта subscriber
    Employee vanya("Vanya");
    Employee artem("Artem");
    Employee maxim("Maxim");
    Employee dmitrii("Dmitrii");

    //Проверим текущее состояние работников
    cout << "Employees states after creation:" << endl;
    cout << vanya;
    cout << artem;
```

```
    cout << maxim;
    cout << dmitrii << endl;

    //Подпишем нескольких работников на объект работодателя
    vanya.subscribeOnEmployer(&google);
    artem.subscribeOnEmployer(&google);
    dmitrii.subscribeOnEmployer(&google);

    //Проверим состояние после подписки на объект работодателя
    cout << "Employees states after publishing:" << endl;
    cout << vanya;
    cout << artem;
    cout << maxim;
    cout << dmitrii << endl;

    //Вызовем метод получения оповещения для всех подписчиков, в котором вызовется сигнал salaryPayment
    int day_salary = 25;
    google.notifyEmployeis(day_salary);

    //Проверим состояние работников, после получения зарплаты
    cout << "Employees states after salary event:" << endl;
    cout << vanya;
    cout << artem;
    cout << maxim;
    cout << dmitrii << endl;

    //Вызовем метод получения оповещения для всех подписчиков, в котором вызовется сигнал taskIssuance
    int someDay = 10;
    google.notifyEmployeis(someDay);

    //Проверим состояние работников, после получения задания
    cout << "Employees states after task event:" << endl;
    cout << vanya;
    cout << artem;
    cout << maxim;
    cout << dmitrii << endl;

    return 0;
}
```

Издатель держит список объектов, которые на него подписались. Вектор пар: подписчик и метод подписчика, который необходимо вызвать. У издателя должны быть методы подписаться() и отписаться().

Отправленный сигнал может обрабатываться подписчиками синхронно и асинхронно.

- Преимущества:
  - Издатели не зависят от подписчиков
  - Схема гибкая: можно как подписаться, так и отписаться (надо делать iterator)
- Недостатки:
  - издатель должен держать список объектов, которые на него подписаны (паттерн тяжёлый)
  - нет порядка в оповещении подписчиков
  - если издателей много, (и каждый подписчик может быть издателем), связей будет очень много

Посредник

- Идея: (последний недостаток подписчика-издателя*(издателей много - много связей)*) связи вынести в отдельный объект, тогда каждый объект будет обращаться к этому отдельному объекту, а он в свою очередь будет искать нужные связи (с кем ему связаться).
- Позволяет уменьшить количество связей между объектами благодаря перемещению связей в него, в посредник. Так как посредник должен работать со всеми объектами, то он содержит указатели на все эти объекты.
- **Диаграмма**

- **Пример кода. Посредник (Mediator)**

  В данном примере кода будут от левого, отправляться всем правым, а от правого, всем левым

```
# include <iostream>
# include <memory>
# include <list>
# include <vector>

using namespace std;

class Message {};          // Request

class Mediator;

class Colleague
{
private:
  weak_ptr<Mediator> mediator;

public:
  virtual ~Colleague() = default;

  void setMediator(shared_ptr<Mediator> mdr) { mediator = mdr; }

  virtual bool send(shared_ptr<Message> msg);
  virtual void receive(shared_ptr<Message> msg) = 0;
};

class ColleagueLeft : public Colleague
{
public:
  virtual void receive(shared_ptr<Message> msg) override { cout << "Right - >
Left;" << endl; }
};

class ColleagueRight : public Colleague
{
public:
  virtual void receive(shared_ptr<Message> msg) override { cout << "Left - >
Right;" << endl; }
};

class Mediator
{
protected:
  list<shared_ptr<Colleague>> colleagues;

public:
  virtual ~Mediator() = default;
```

```
  virtual bool send(const Colleague* coleague, shared_ptr<Message> msg) = 0;

  static bool add(shared_ptr<Mediator> mediator,
initializer_list<shared_ptr<Colleague>> list);
};

class ConMediator : public Mediator
{
public:
  virtual bool send(const Colleague* coleague, shared_ptr<Message> msg) override;
};

#pragma region Methods Colleague
bool Colleague::send(shared_ptr<Message> msg)
{
  shared_ptr<Mediator> mdr = mediator.lock();

  return mdr ? mdr->send(this, msg) : false;
}
#pragma endregion

#pragma region Methods Mediator
bool Mediator::add(shared_ptr<Mediator> mediator,
initializer_list<shared_ptr<Colleague>> list)
{
  if (!mediator || list.size() == 0) return false;

  for (auto elem : list)
  {
          mediator->colleagues.push_back(elem);
          elem->setMediator(mediator);
  }

  return true;
}

bool ConMediator::send(const Colleague* colleague, shared_ptr<Message> msg)
{
  bool flag = false;
  for (auto& elem : colleagues)
  {
          if (dynamic_cast<const ColleagueLeft*>(colleague) &&
dynamic_cast<ColleagueRight*>(elem.get()))
          {
                  elem->receive(msg);
                  flag = true;
          }
          else if (dynamic_cast<const ColleagueRight*>(colleague) &&
dynamic_cast<ColleagueLeft*>(elem.get()))
          {
                  elem->receive(msg);
                  flag = true;
          }
  }

  return flag;
}
#pragma endregion

int main()
{
  shared_ptr<Mediator> mediator =make_shared<ConMediator>();

  shared_ptr<Colleague> col1 = make_shared<ColleagueLeft>();
  shared_ptr<Colleague> col2 = make_shared<ColleagueRight>();
  shared_ptr<Colleague> col3 = make_shared<ColleagueLeft>();
```

```
  shared_ptr<Colleague> col4 = make_shared<ColleagueLeft>();

  Mediator::add(mediator, { col1, col2, col3, col4 });

  shared_ptr<Message> msg = make_shared<Message>();

  col1->send(msg);
  col2->send(msg);
}
/*
Left - > Right;
Right - > Left;
Right - > Left;
Right - > Left;
*/
```

Конкретный медиатор устанавливает, кому передается и сообщение.

- Преимущества:
  - Упрощаются объекты (выносим из них связи)
  - Нет прямой зависимости между ними
  - Происходит централизованное взаимодействие (появляется контроль)
- Недостатки:
  - Использование посредников замедляет выполнение программы

16. **Паттерны поведения: посетитель (Visitor), опекун (Memento), шаблонный метод (Template Method), хранитель (Holder), итератор (Iterator), свойство (Property). Их преимущества и недостатки.**

ОТВЕТ: посетитель (Visitor), опекун (Memento), шаблонный метод (Template Method), хранитель (Holder), итератор (Iterator), свойство (Property).

Посетитель

Идея

- Следующая проблема - связанная с изменением интерфейса объектов. Если мы используем полиморфизм, мы не можем в производном классе ни сузить, ни расширить интерфейс, так как он должен четко поддерживать интерфейс базового класса.
- Если нам необходимо расширить интерфейс, можно использовать паттерн Визитёр. Он позволяет **во время выполнения** (в отличие от паттерна Адаптера, который решает эту проблему до выполнения) подменить или расширить функционал.
- **Диаграмма**

  Чтобы можно было поменять/расширять функционал, а базовом классе добавляем метод accept(Visitor). Соответственно, все производные классы могут подменять этот метод.

- **Пример кода. Visitor**

```
#include <iostream>
#include <memory>
#include <vector>

using namespace std;

class Circle;
class Rectangle;

class Visitor {
 public:
    virtual ~Visitor() = default;

    virtual void visit(Circle& ref) = 0;
    virtual void visit(Rectangle& ref) = 0;
};

class Shape {
 public:
    virtual ~Shape() = default;

    virtual void accept(shared_ptr<Visitor> visitor) = 0;
};

class Circle : public Shape {
 public:
    virtual void accept(shared_ptr<Visitor> visitor) override {
        visitor->visit(*this);
    }
};
```

```
class Rectangle : public Shape {
 public:
    virtual void accept(shared_ptr<Visitor> visitor) override {
        visitor->visit(*this);
    }
};

class ConVisitor : public Visitor {
 public:
    virtual void visit(Circle& ref) override { cout << "Circle;" << endl; }
    virtual void visit(Rectangle& ref) override {
        cout << "Rectangle;" << endl;
    }
};

class Formation {
 public:
    static vector<shared_ptr<Shape>> initialization(
        initializer_list<shared_ptr<Shape>> list) {
        vector<shared_ptr<Shape>> vec;

        for (auto elem : list) vec.push_back(elem);

        return vec;
    }
};

int main() {
    vector<shared_ptr<Shape>> figure = Formation::initialization(
        {make_shared<Circle>(), make_shared<Rectangle>(),
         make_shared<Circle>()});
    shared_ptr<Visitor> visitor = make_shared<ConVisitor>();

    for (auto& elem : figure) elem->accept(visitor);
}
```

Визитёр один функционал собирает в одно место для разных классов. Для каждого такого класса/подкласса есть свой метод, который принимает элемент этого подкласса. Конкретный визитёр уже реализует этот функционал.

Преимущества

- В один класс сводим методы, относящиеся к одному функционалу

Недостатки

- Расширяется иерархия, добавляются новые классы. Проблема связи на уровне базовых классов
- Может меняться иерархия (крайне редкая проблема). Тогда визитёр не срабатывает.
- Есть проблема как у стратегии - проблема с данными, данные могут меняться

Тассов говорил что вопросы с иерархией можно решить с помощью темплейтного визитера и паттерна CRTP с переменным числом параметров, ну это так, пиздануть, чтобы ему понравилось *(14_3 23:33)*

Опекун

Этот паттерн имеет много названий. Одно из них - Опекун.

Идея

Когда мы выполняем много операций, объект изменяется, но, те изменения которые у нас произошли, могут нас не устраивать, и мы можем вернуться к предыдущему состоянию нашего объекта (откат).

Как вариант - хранить те, которые были у объекта. Если возложить эту задачу на объект - он получится тяжелым.

Идея - выделить эту обязанность другому объекту, задача которого - хранить предыдущие состояния нашего объекта, который, если нужно, позволит нам вернуться к какому-либо предыдущему состоянию.

- **Диаграмма(set Object)**

  Memento - *снимок* объекта в какой-то момент времени. Опекун отвечает за хранение этих снимков и возможность вернуться к предыдущему состоянию объекта

- **Пример кода. Опекун (Memento).**

```
# include <iostream>
# include <memory>
# include <list>

using namespace std;

class Memento;

class Caretaker
{
public:
  unique_ptr<Memento> getMemento();
  void setMemento(unique_ptr<Memento> memento);

private:
  list<unique_ptr<Memento>> mementos;
};

class Originator
{
public:
  Originator(int s) : state(s) {}

  const int getState() const { return state; }
  void setState(int s) { state = s; }

  std::unique_ptr<Memento> createMemento() { return make_unique<Memento>(*this); }
  void restoreMemento(std::unique_ptr<Memento> memento);

private:
  int state;
};
```

```
•
•  class Memento
•  {
•    friend class Originator;
•
•  public:
•    Memento(Originator o) : originator(o) {}
•
•  private:
•    void setOriginator(Originator o) { originator = o; }
•    Originator getOriginator() { return originator; }
•
•  private:
•    Originator originator;
•  };
•
•  #pragma region Methods Caretaker
•  void Caretaker::setMemento(unique_ptr<Memento> memento)
•  {
•    mementos.push_back(move(memento));
•  }
•
•  unique_ptr<Memento> Caretaker::getMemento() {
•
•    unique_ptr<Memento> last = move(mementos.back());
•
•    mementos.pop_back();
•
•    return last;
•  }
•
•  #pragma endregion
•
•  #pragma region Method Originator
•  void Originator::restoreMemento(std::unique_ptr<Memento> memento)
•  {
•    *this = memento->getOriginator();
•  }
•
•  #pragma endregion
•
•  int main()
•  {
•    auto originator = make_unique<Originator>(1);
•    auto caretaker = make_unique<Caretaker>();
•
•    cout << "State = " << originator->getState() << endl;
•    caretaker->setMemento(originator->createMemento());
•
•    originator->setState(2);
•    cout << "State = " << originator->getState() << endl;
•    caretaker->setMemento(originator->createMemento());
•    originator->setState(3);
•    cout << "State = " << originator->getState() << endl;
•    caretaker->setMemento(originator->createMemento());
•
•    originator->restoreMemento(caretaker->getMemento());
•    cout << "State = " << originator->getState() << endl;
```

```
  originator->restoreMemento(caretaker->getMemento());
  cout << "State = " << originator->getState() << std::endl;
  originator->restoreMemento(caretaker->getMemento());
  cout << "State = " << originator->getState() << std::endl;
}
```

Преимущества

Позволяет не грузить сам класс задачей сохранять предыдущие состояния.

Недостатки

Опекуном надо управлять. Он наделал снимков, а они нам не нужны. Кто-то должен их очищать. Какой механизм очистки? Много памяти тратится.

Над этим всем должен стоять кто-то, который решает, когда, например делать снимки

Шаблонный метод

Идея

Этот паттерн является скелетом какого-либо метода. Мы любую задачу разбиваем на этапы - формируем шаги, которые мы выполняем для того, чтобы то, что мы получили на входе, преобразовать в результат, который нам нужен.

Есть диаграмма, которая по существу задает нам методы.

- **Диаграмма**

  А вот реальная диаграмма:

- **Пример кода. Шаблонный метод (Template Method).**

```
# include <iostream>

using namespace std;

class AbstractClass
{
public:
    void templateMethod()
    {
        primitiveOperation();
        concreteOperation();
        hook();
    }

protected:
    virtual void primitiveOperation() = 0;
    void concreteOperation() { cout << "concreteOperation;" << endl; }
    virtual void hook() { cout << "hook Base;" << endl; }
```

```
};

class ConClassA : public AbstractClass
{
protected:
    virtual void primitiveOperation() override { cout << "primitiveOperation A;" << endl; }
};

class ConClassB : public AbstractClass
{
protected:
    virtual void primitiveOperation() override {    cout << "primitiveOperation B;" << endl; }
    void hook() { cout << "hook B;" << endl; }
};

int main()
{
    ConClassA ca;
    ConClassB cb;
    ca.templateMethod();
    cb.templateMethod();
}
```

Хранитель(Holder)

Существует проблема. Предположим, у нас есть класс A, в котором есть метод `f()`. Страшный код. Мы не знаем, что творится внутри `f()`, и, естественно, мы используем механизм обработки исключительных ситуаций. Внутри `f()` происходит исключительная ситуация, она приводит к тому, что мы перескакиваем на какой-то обработчик, неизвестно где находящийся. Это приводит к тому, что объект `p` не удаляется - происходит утечка памяти.

- Страшный код:

```
{
  A* p = new A;
  p->f();   // Внутри f() происходит исключительная ситуация
  delete p; // Объект p не удаляется
}
```

Как решить эту проблему?

Мы можем указатель `p` обернуть в какой-то объект - хранитель. Этот объект будет содержать указатель на объект `A`. Задача объекта: при выходе из области видимости объекта-хранителя

будет вызываться деструктор `obj`, в котором мы можем уничтожить объект `A`.

```
{
    Holder<A> obj(new A);
}
```

Для объекта хранителя достаточно определить три операции - `*` (получить значение по указателю), `->` (обратиться к методу объекта, на который указывает указатель) и `bool` (проверить, указатель указывает на объект, `nullptr` он или нет). Чтобы можно было записать `obj->f();`. То есть эта оболочка должна быть "прозрачной". Её задача должна быть только вовремя освободить память, выделенную под объект. Мы работаем с объектом класса A через эту оболочку.

- **Пример кода. Хранитель(Holder)**

```
template <typename T>
class Holder
{
    T* ptr{nullptr}; // Указатель на объект (сразу же его обнуляем)
public:
    explicit Holder(T *p) : ptr(p) {}; // Захватываем указатель и запрещаем неявный вызов конструктора
    ~Holder() {delete ptr;}            // Задача деструктора - удалить объект

  // Определяем джентельменский набор из трёх операторов
    T& operator *() const {return *ptr;}
    T* operator ->() const {return ptr;}
    operator bool() const { return ptr != nullptr; }

    // Запрещаем конструктор копирования и оператор присваивания
    Holder(const Holder &) = delete; // Если у нас параметр T по умолчанию, можно его явно не указвать
                                     // а использовать & (просто пояснение)
    Holder operator=(const Holder &) = delete;
};
```

- **Проблема висящего указателя**

  Этот хранитель решает ситуацию, связанную с обработкой исключительных ситуаций. Но предположим, что у нас есть один объект класса `A` и класс `B` держит указатель на объект класса `A`.

```
class A {...};

class B
{
  A* p;
}
```

  Например, мы получили указатель `p`. Этот объект может быть удалён, и в этом случае возникает проблема: **указатель, инициализированный каким-то адресом, будет указывать на удалённый объект**. Можно рассматривать каждый объект, который держит указатель, как хранитель. То есть мы отдаём указатель на объект, а объект-хранитель считает, что этот объект его собственный, происходит захват.

  В случае если хранитель отдаёт объект, нужно позаботиться о том, чтобы не образовался "висящий" указатель, то есть указатель на объект, которого нет.

  Проблема с утечкой памяти не такая острая как проблема с висящим указателем. Утечка памяти приводит всего лишь к нехватке памяти, в то время как с висящим указателем мы можем случайно вызвать метод несуществующего объекта, что приведёт к падению системы.

  Представим, что на один объект держат указатели несколько объектов. Как понять, какой из объектов должен удалять этот указатель? Если это отдавать на откуп программиста, то о надежности такого кода говорить нельзя, возможно ошибка. Допустим, мы выбрали один из объектов ответственным. Какая гарантия, что он не уничтожится раньше, чем другие два объекта?

Итератор (Iterator)

Предоставляет способ доступа к элементам контейнера, независимо от его внутреннего устройства. в с++ введён новый цикл for each:

- **Пример разложения**

```
// Это:
for (auto elem : obj)
  cout << elem;

// Разложится в это:
for (Iterator <Type> It = obj.begin(); It != obj.end(); ++It)
{
  auto elem = *It;
  cout << elem;
}
```

- **Пример кода. Iterator. (без проверок и обработки исключительных ситуаций)**

```
# include <iostream>
# include <memory>
# include <iterator>
# include <initializer_list>

using namespace std;

template <typename Type>
class Iterator;

class BaseArray
{
public:
  BaseArray(size_t sz = 0) { count = shared_ptr<size_t>( new size_t(sz) ); }
  virtual ~BaseArray() = default;

  size_t size() { return bool(count) ? *count : 0; }
  operator bool() { return size(); }

protected:
  shared_ptr<size_t> count;
};

template <typename Type>
class Array final : public BaseArray
{
public:
  Array(initializer_list<Type> lt);
  virtual ~Array() {}

  Iterator<Type> begin() const { return Iterator<Type>(arr, count); }
  Iterator<Type> end() const { return Iterator<Type>(arr, count, *count);     }

private:
  shared_ptr<Type[]> arr{ nullptr };
};

template <typename Type>
class Iterator : public std::iterator<std::input_iterator_tag, Type>
{
  friend class Array<Type>;

private:
```

```
  Iterator(const shared_ptr<Type[]>& a, const shared_ptr<size_t>& c, size_t ind = 0) : arr(a), count(c), index(ind) {}
public:
  Iterator(const Iterator &it) = default;

  bool operator!=(Iterator const& other) const;
  bool operator==(Iterator const& other) const;

  Type& operator*();
  const Type& operator*() const;
  Type* operator->();
  const Type* operator->() const;
  Iterator<Type>& operator++();
  Iterator<Type> operator++(int);

private:
  weak_ptr<Type[]> arr;
  weak_ptr<size_t> count;
  size_t index = 0;
};

#pragma region Method Array

template <typename Type>
Array<Type>::Array(initializer_list<Type> lt)
{
  if (!(*count = lt.size())) return;
  arr = shared_ptr<Type[]>(new Type[*count]);
  size_t i = 0;
  for (Type elem : lt)
          arr[i++] = elem;
}

#pragma endregion

#pragma region Methods Iterator

template <typename Type>
bool Iterator<Type>::operator!=(Iterator const& other) const { return index != other.index; }

template <typename Type>
Type& Iterator<Type>::operator*()
{
         // нужно проверить корректность итератора
  shared_ptr<Type[]> a(arr);
  return a[index];
}

template <typename Type>
Iterator<Type>& Iterator<Type>::operator++()
{
  shared_ptr<size_t> n(count);
  if (index < *n)
          index++;
  return *this;
}

template <typename Type>
```

```
Iterator<Type> Iterator<Type>::operator++(int)
{
  Iterator<Type> it(*this);
  ++(*this);
  return it;
}

#pragma endregion

template <typename Type>
ostream& operator<<(ostream& os, const Array<Type>& arr)
{
   for (auto elem : arr)
       cout << elem << " ";
   return os;
}

int main()
{
  Array<int> arr{ 1, 2, 3, 4, 5 };
  cout << " Array: " << arr << endl;
}
```

Паттерн Свойство(Property)

Не столько паттерн поведения, сколько шаблон, который нам даёт возможность формализовать свойства. В современных языках есть понятие свойство. Предположим, у нас есть какой то класс, и у его объекта есть свойство V.

```
A objl
obj V = 2;
int i = obj.V;
```

(В реальном мире такого доступа быть не должно). Доступ осуществляется через методы, один устанавливает - set(), другой возвращает - get(). Если мы будем рассматривать V как открытый член, но не простое данное... Если рассматривать V как объект, мы для него должны определить V как оператор присваивания.

Нам нужен класс, в котором есть перегруженный оператор = и оператор приведения типа. Этот перегруженный оператор равно должен вызывать метод установки сет, а этот гет.

Удобно создать шаблон свойства. Первый параметр - тип объекта для которого создается шаблон, а второй параметр - тип объекта к которому приводится и ли инициализируется значение. В данном случае это целый тип. Getter - метод класса который возвращает Type, а Setter - установка для вот этого объекта.

- **Пример кода. Свойство(Property)**

```
# include <iostream>
# include <memory>

using namespace std;

template <typename Owner, typename Type>
class Property
{
private:
```

```
  using Getter = Type (Owner::*)() const;
  using Setter = void (Owner::*)(const Type&);

  Owner* owner;
  Getter methodGet;
  Setter methodSet;

public:
  Property() = default;
  Property(Owner* owr, Getter getmethod, Setter setmethod) : owner(owr), methodGet(getmethod), methodSet(setmethod) {}

  void init(Owner* owr, Getter getmethod, Setter setmethod)
  {
          owner = owr;
          methodGet = getmethod;
          methodSet = setmethod;
  }

  operator Type() { return (owner->*methodGet)();  }                                    // Getter
  void operator=(const Type& data) {    (owner->*methodSet)(data); }   // Setter

//       Property(const Property&) = delete;
//       Property& operator=(const Property&) = delete;
};

class Object
{
private:
  double value;

public:
  Object(double v) : value(v) { Value.init(this, &Object::getValue, &Object::setValue); }

  double getValue() const { return value; }
  void setValue(const double& v) { value = v; }

  Property<Object, double> Value;
};

int main()
{
  Object obj(5.);

  cout << "value = " << obj.Value << endl;

  obj.Value = 10.;

  cout << "value = " << obj.Value << endl;

  unique_ptr<Object> ptr(new Object(15.));

  cout << "value =" << ptr->Value << endl;

  obj = *ptr;
  obj.Value = ptr->Value;
}
```