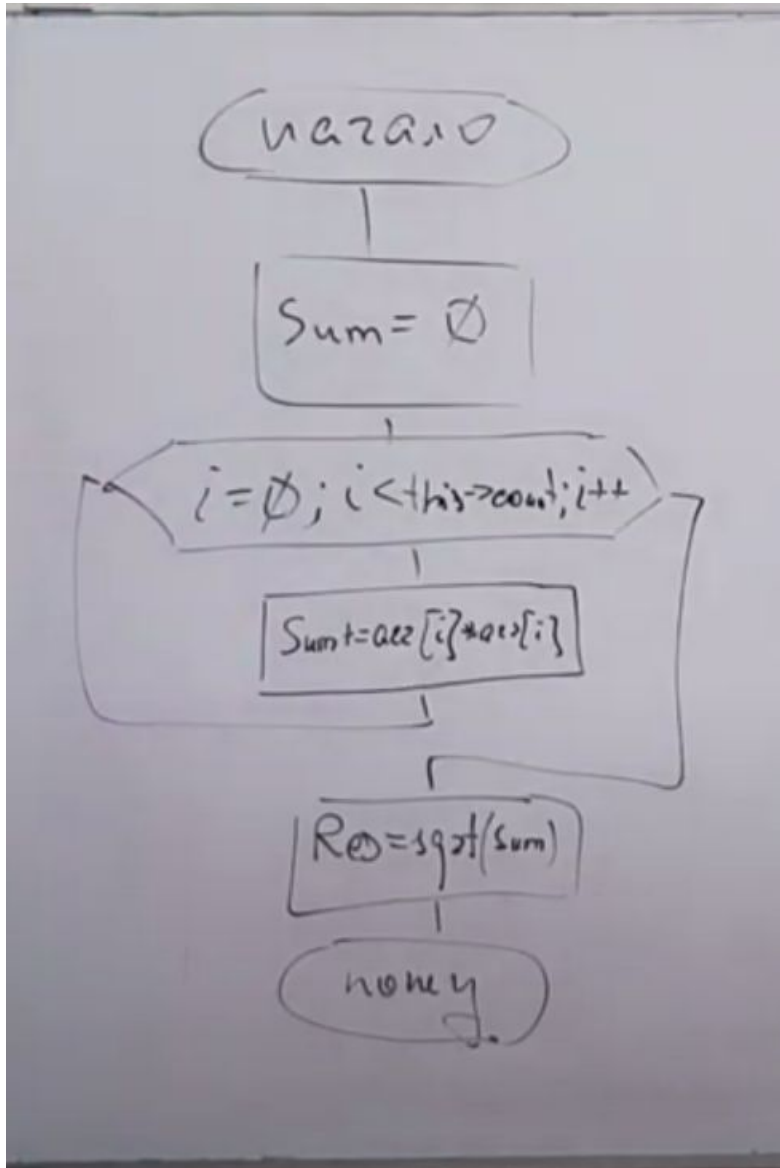


Лекция 11

Рассмотрим схему алгоритма нахождения длины вектора.



Чем он плох? Мы видим порядок выполнения действий, но мы не анализируем откуда берутся эти данные и как эти действия связаны между собой. Далее рассмотрим другое представления алгоритма.

Задача оценки возможности распараллеливания вычисления.

Оцениваем не только порядок вычислений, но и доступность данных для выполнения того или иного действия.

Диаграмма потоков данных действий. (ДПДД)

Мы наш алгоритм разбиваем на процессы, которые происходят.
Диаграмма строится для каждого состояния каждой модели состояний.

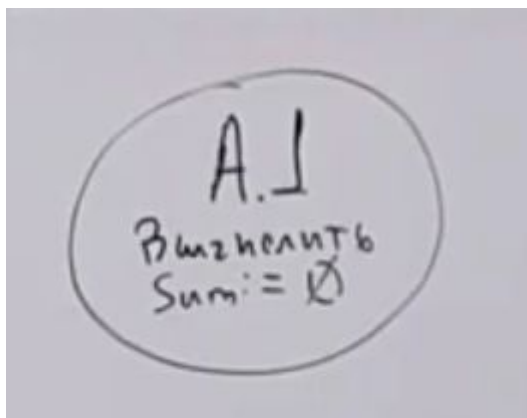
Каждое действие можем выделить как процесс, который происходит.
Каждый процесс отвечает на вопрос “Что он делает?”

Выделяем процесс вычисления суммы:

A - ключевой литерал (массив array)

1 - номер процесса

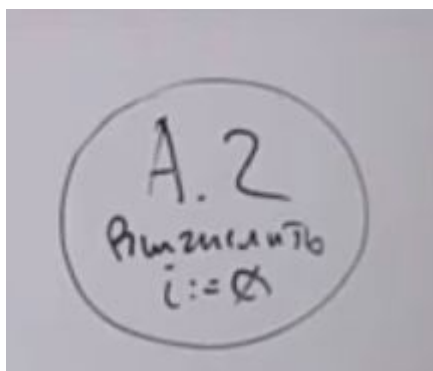
вычислить $sum:=0$ - определяем, что он делает.



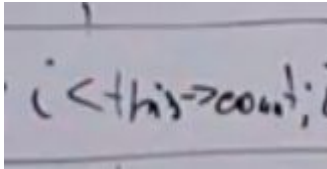
Далее идет цикл.

Процесс второй:

Вычислить $i:=0$

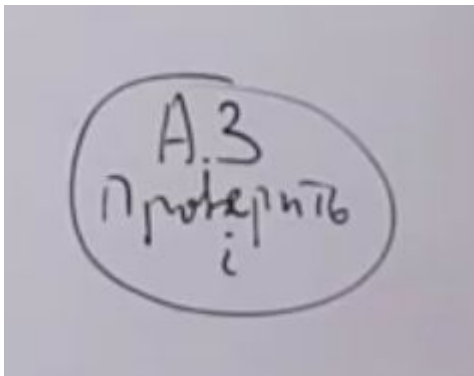


Теперь займемся данным выражением. Выделим его как отдельный процесс

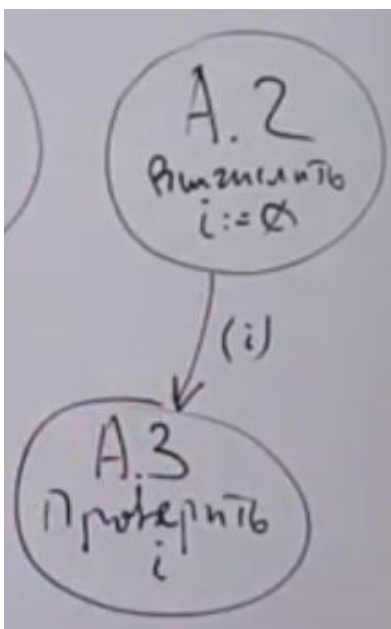


`i <+this->count; i`

Процесс 3
Проверить.



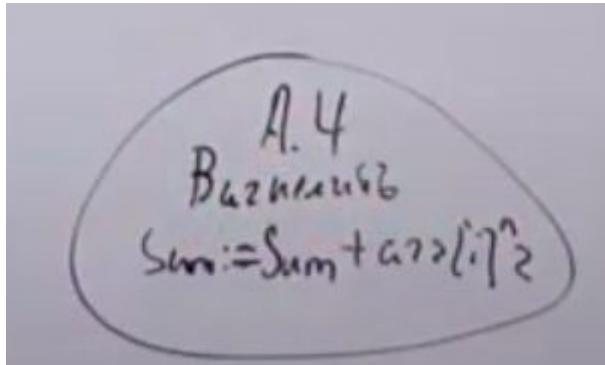
Но, чтобы проверить i нам нужно получить i , поэтому процесс 3 должен выполняться после процесса 2. Рисуем стрелочку и на ней показываем данное которое передается. В круглых скобках, потому что рассматривается неустойчивое данное, т.е. данное, которое используется только во время выполнения нашего действия.



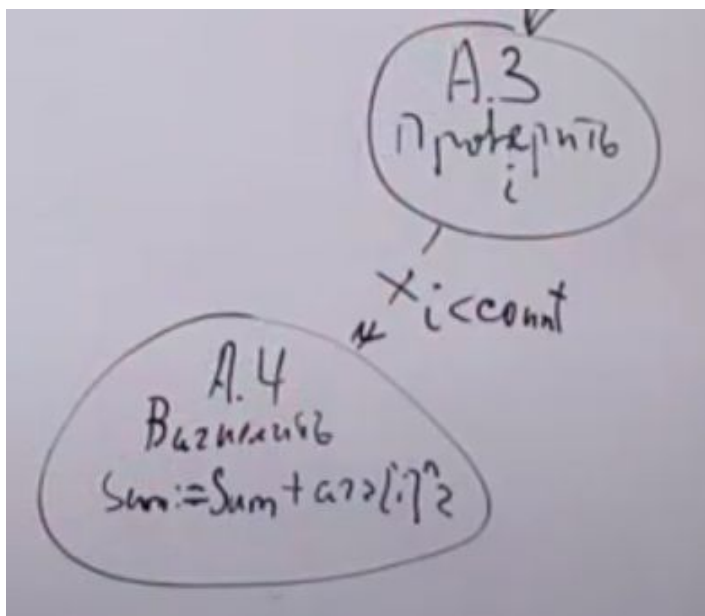
После проверки у нас возможны разные варианты. В зависимости от этой проверки. В данном случае процесс 3 не выполнится, пока что не выполнится процесс 2.

Процесс A4

вычислить $sum := sum + arr[i]^2$



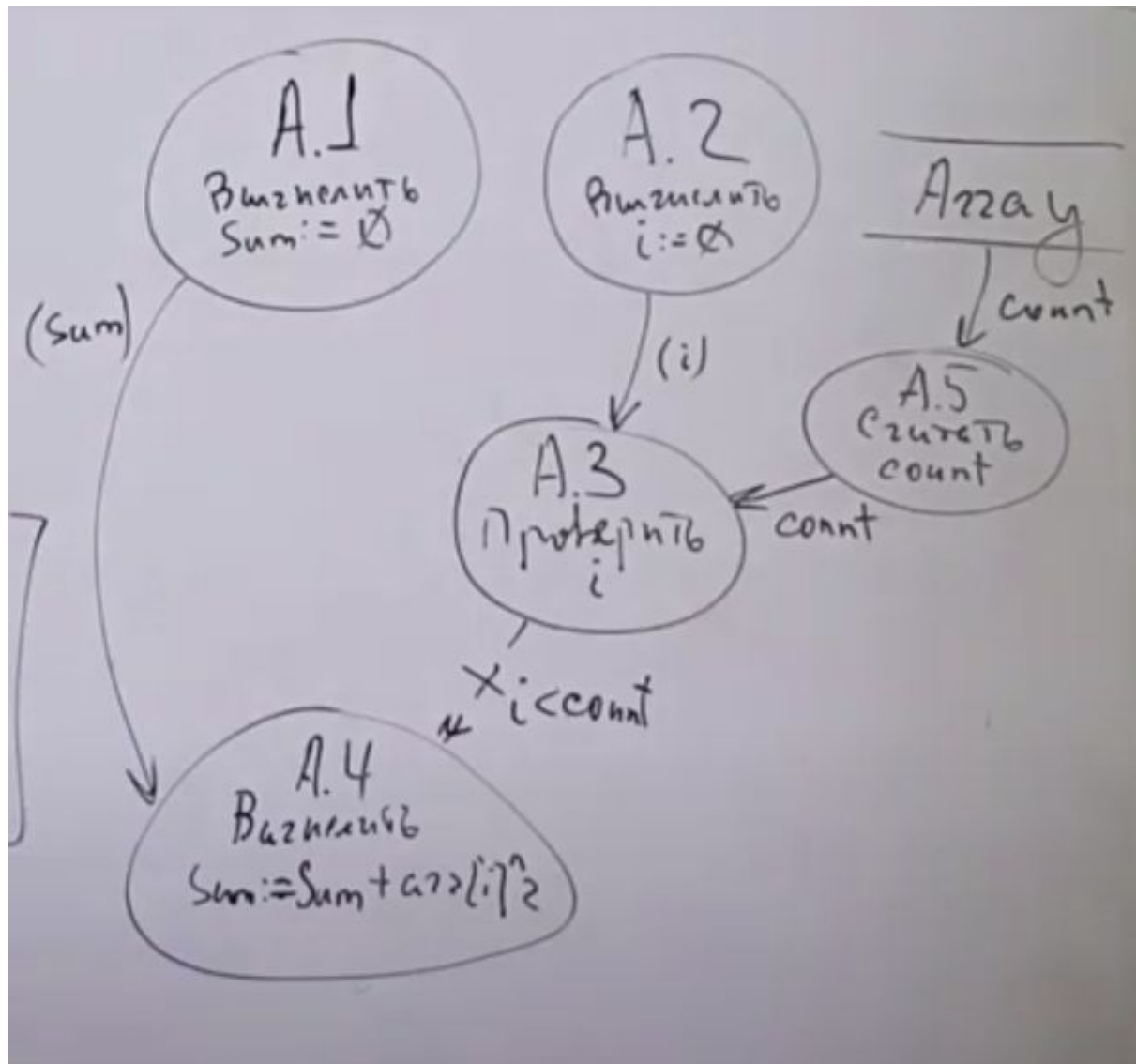
Данный процесс (A4) использует условие A3. Здесь идет **условное управление** (пунктирная линия). Т.е. A4 не может выполнить перед процессом A3, но он также выполняется по условию поэтому записываем условие на ребре ($i < count$)



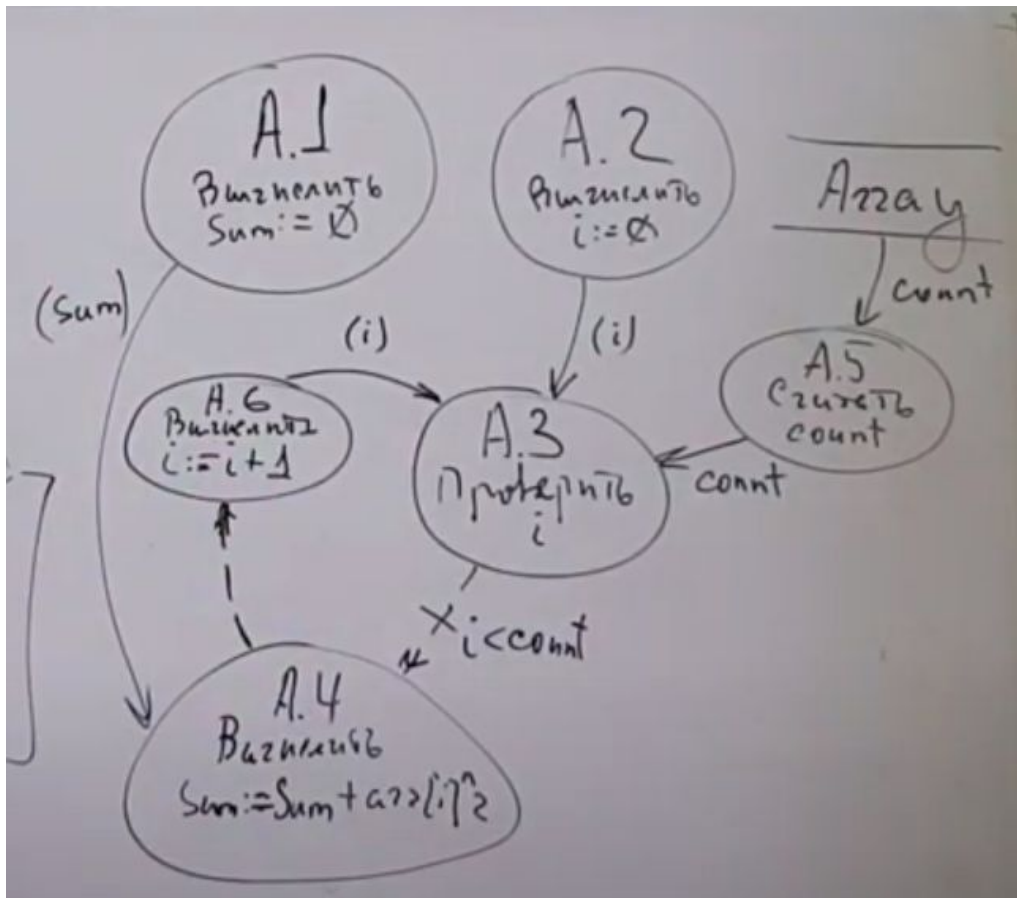
Но этот процесс, чтобы выполнить эту проверку, должен взять откуда-то count. Данные самого объекта рассматриваются как архив данных. И доступ должен происходить к данным через процесс.

Т.е. создаем процесс, который берет из архива данных количество (считать count) и этот процесс используется проверкой.

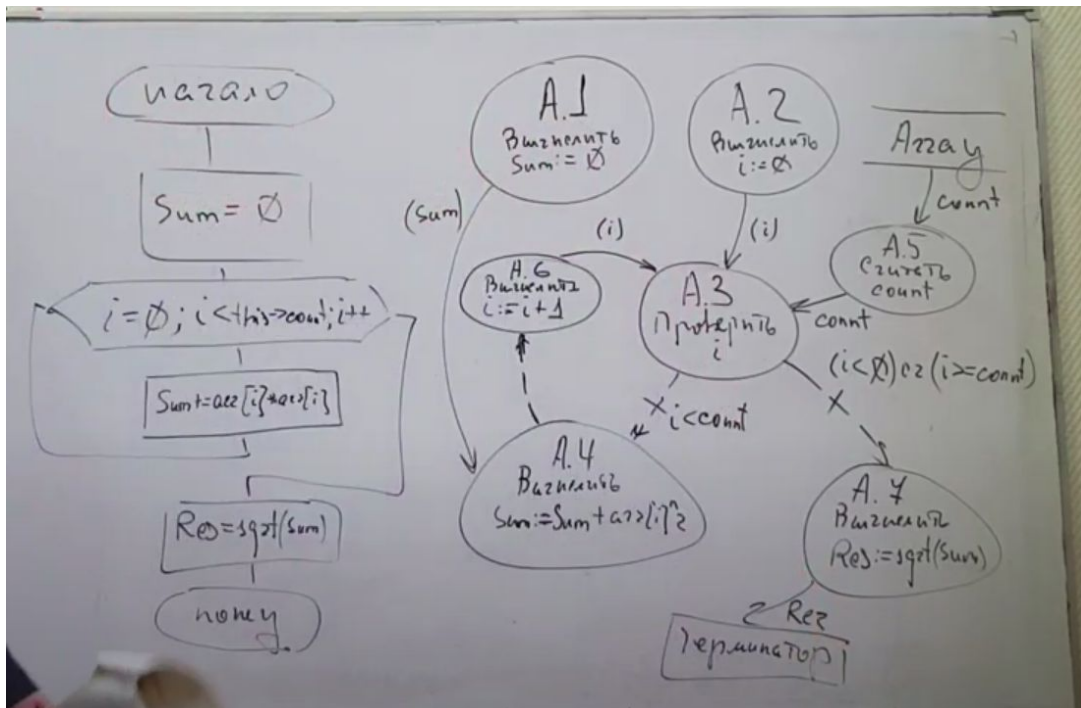
Также мы должны использовать сумму (Т.е. должна быть передана сумма в процесс A4)



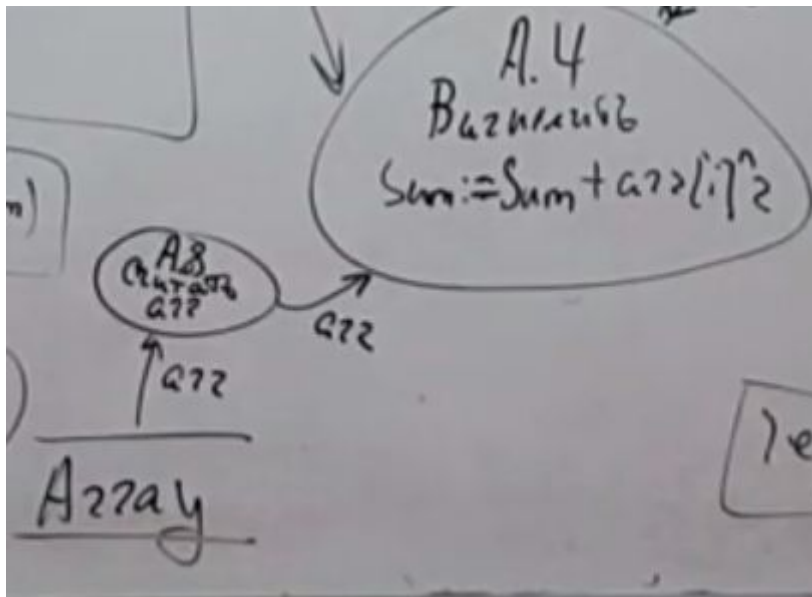
После того, как мы вычислили сумму мы можем вычислить $i++$ (инкремент). Данный процесс не берет ни от кого данных. Но он выполняется после A4. И после этого полученное i используется в A3.



Теперь выход.



Мы не показали в 4 процессе, что считываем атг из архива. Архив можно продублировать на диаграмме для удобства.



Теперь все.

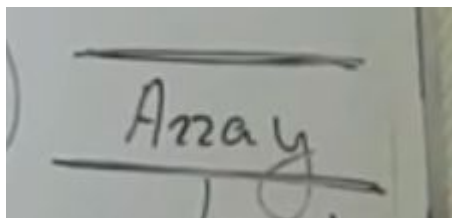
Правило: Все входы должны быть доступны.

Резюме:

Стрелками мы показываем потоки данных (передача данных) если стрелка направлена к процессу, то это входные данные, если от процесса - то выходные. Каждую стрелочку помечаем теми данными, которые передаются (одно данное).

Если читаем атрибуты самого объекта, то просто записываем имя атрибута. Если это объект того же класса, но это другой объект, или это другой объект, то имя нужно аннотировать впереди идентификатора. (array.count - другой объект, не тот же самый)

Атрибуты самого объекта, других объектов того же класса или других классов мы рассматриваем как архив данных и обозначается вот так:



Процесс может принимать событие или порождать событие. Процесс, который принимает событие записывается в овале и рисуется

стрелочка из неоткуда. Помечается данным событием и описанием события (установить таймер)



Также процесс может порождать события. Стрелочка в никуда.

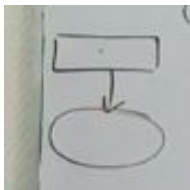


У нас могут быть безусловные потоки управления и условные.

Безусловные потоки указывают только на порядок в котором выполняются процессы, передача данных не происходит. (Если i считали один раз, то дальше мы можем его использовать)

Если это данное, которое порождается во время выполнения действия помечаем его как неустойчивое (в скобках “()”). Данные из архива - устойчивые.

Процесс берет из какой-то внешней сущности, рисуем как терминатор:



Правила выполнения процесса:

1. Процесс может выполниться, когда все входы (входы) доступны.
2. Выводы процесса доступны когда процесс выполнится.

Что касается данных:

3. Данные событий всегда доступны (которые приводят к выполнению).
4. Данные из архива данных и данные терминаторов всегда доступны.

Какие мы выделяем виды процессы:

1. Аксессоры.
2. Генераторы событий.
3. Преобразования
4. Проверки.

Аксессор - это такой процесс, единственная цель которого состоит в том, чтобы получить доступ к архиву данных.

Виды Аксессоров:

1. Создания
2. Уничтожения
3. Чтения (Аксессорный процесс A5 и A8)
4. Записи.

Генераторы событий - процесс, который создает на выходе событие.



Замечание: Этот процесс, на который должен реагировать кто-то, определяется для того объекта, который принимает событие



Преобразование - это процесс, который выполняет какие-либо преобразования данных (A1, A2 A4, A6, A7)

Проверки - результат: условный переход (один или несколько) (у нас было две)

Когда мы выделяем процессы мы должны четко описать что они из себя представляют - наименование процесса и ответ на вопрос “что делает?”. Имя формируется из имени сущности и номера процесса.

Если процесс преобразования, то должны описать, что вычисляем.

Выделение общих процессов: (в дальнейшем должны уйти от дублирования когда)

- Процессы выполняют одни и те же вычисления.
- Процессы читают или записывают одни и те же данные
- Создают или уничтожают одни и те же объекты.
- Принимают одни и те же атрибуты от терминаторов
- Создают одни и те же данные
- Которые порождают одни и те же события
- Порождают одни и те же выходы условного управления.

Все процессы, которые происходят в нашей подсистеме мы объединяем в таблицу процессов состояний:

(Сначала мы разбиваем процессы по ДДПД (на разных дпдд разные процессы). Далее мы анализируем и пытаемся объединить все эти процессы, которые проходят в одной модели состояний (мы начинаем выявлять подобные процессы, происходящие на одной модели состояний). Мы четко выделяем общие процессы.

(Название процесса) (где используется, модель состояний / действие)

Id процесса	Тип	Название проц.	где используется	
			мод. сост.	Действие

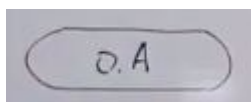
(Один и тот же процесс может использоваться в разных моделях состояний и в разных действиях, выделив такие процессы (которые явл. общими для разных моделей состояний) мы можем выделять действия присущие этой модели состояний, но не явл. обработчиками состояний. Если в разных моделях состояний происходят одни и те же процессы, и если модели родственные (объединены общей базой) ,то эти процессы можно вынести на уровень базовый, если это процессы, которые происходят в моделях состояний, которые не объединены (не относятся к

подклассам), то в этом случае можно выделить их общей базой (выделить суперкласс))

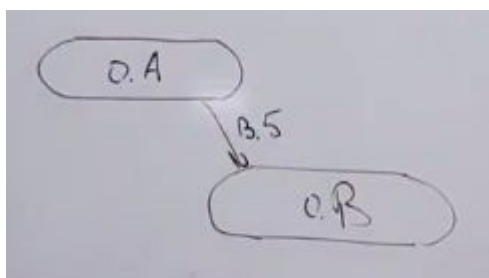
Модель (диаграмма) доступа к объектам (МДО)

(У нас наши объекты могут взаимодействовать за счет аксессоров, т.е. одна модель состояния может использовать аксессор, относящийся к другому объекту (или другому модели состояния) В этом случае строится МДО) (Коротко: На основе выделенных аксессорных процессов строится модель доступа к объектам.)

На этой диаграмме каждый объект рисуется вот так:



И мы рисуем стрелочку от того объекта, который использует аксессор к тому объекту, аксессор которого используется. (Какой процесс (Любой: создание, уничтожение, чтение, запись) ? - "В" и номер, например, 5) (Объект А использует аксессор объекта В)



В результате получаем модель доступа к объектам (МДО), так же есть выделенные нами процессы общие (таблица процессов), описание этих процессов и мы представили алгоритм в форме взаимодействия процессов (ДПДД)

Изначально ДПДД (Де Марк) расписывал, чтобы понять какие процессы могут выполняться одновременно. А мы используем, чтобы выделить эти процессы.

Домены

Домен - это отдельный реальный гипотетический мир населенный отчетливым набором объектов, которые ведут себя в соответствии с характерными для домена правилами и линиями поведения.

Мы нашу систему разбиваем на домены.

Выделяем домены:

1. Прикладной домен (та задача, которую решают).
2. Сервисный домены (Домены, которые обеспечивают сервисные функции (к примеру интерфейс или чтение из базы данных)) - то, то необходимо нашей задаче.
3. Архитектурный домен - очень большая часть (много доменов) (мы их не рассматриваем) Он нам предоставляет общие механизмы и структуры для управления данными и управление всей нашей системой как единым целым. Задачи, которые решает: придать однородность структуре программного обеспечения, обеспечить систему как единое целое. Линия поведения: должен реализовывать доступность к данным, каналы управления, задает структуру прикладного и сервисных доменов, взаимодействия различных доменов, подсистем.
4. Домена реализации - домены, которые связаны с общими библиотеками классов, функционал операционных систем.

Домен разделяют на подсистемы (если домен разросся) подсистемы с минимальными связями. В каждой группе много связей, между группами мало.

Диаграмма сущность связь -> (реализовываем диаграмму) модель связей подсистем.

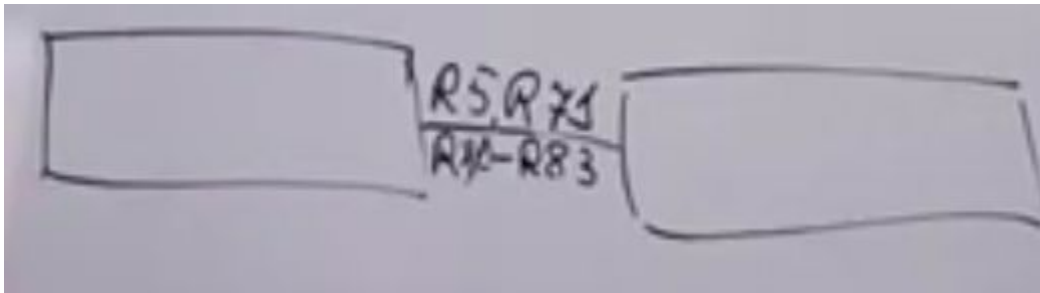
Модель взаимодействия объектов (Событийная) -> модель взаимодействия подсистем.

Модель доступа к объектам (Синхронная) -> модель доступа к подсистемам.

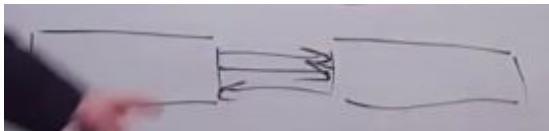
Диаграммы отличаются только связями

Подсистема рисуется в прямоугольнике.

Связи подсистем мы рисуем связь, соединяя линией и перечисляем связи (может быть диапазон).

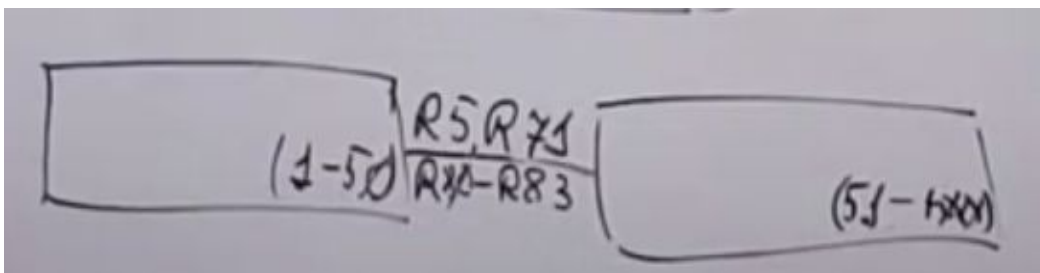


На модели взаимодействия подсистем указывают все события, которые происходят. Каждое событие помечается идентификатором.



Модель доступа к подсистемам так же, только эти стрелки уже не события, а акцессорные события.

В прямоугольнике пишем, что за подсистема и указываем диапазон номеров объектов, которые у нас в каждой подсистеме.



Это все только в том случае, если домен разбиваем на подсистемы.

Домены могут взаимодействовать между собой.

Клиент - тот домен, который использует возможности другого домена.

Сервер - тот домен, который предоставляет что-то другому.

О связи между доменами мы говорим как о “мосте” (существует мост между сервером и клиентом...)

Все полученные документы нужны на этапе анализ, чтобы построить модель нашей системы (задачи).

Объектно ориентированное проектирование.

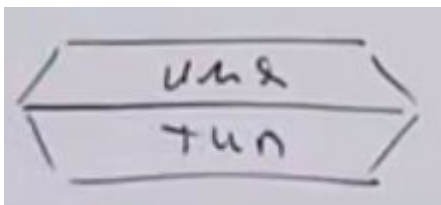
(Object Oriented Design Language)

Выделяется 4 диаграммы:

1. Диаграмма классов - диаграмма внешнего представления одного класса.
2. Схема структуры класса. Используется, чтобы показать внутреннюю структуру класс и доступа к данным.
3. Диаграмма зависимостей - показывает дружественные связи. И клиент серверный доступ за счет аксессуаров
4. Диаграмма наследования.

Диаграмма классов.

Мы проектируем класс вокруг данных этого класса (атрибутов класса). У нас есть информационная модель, где мы выделили каждую сущность и каждая сущность должна быть классом. У нас есть имя этой сущности -> имя нашего класса. Далее у нас есть **данные (атрибуты класса)**, которые становится членами класса при проектировании. Мы перечисляли какие значения может принимать атрибут. Теперь эти перечисления помогут нам понять какого типа у нас будет компонент. Имя берем то, которое у нас было на информационной модели. И на основе тех значений, которые может принимать значения определяется тип.



Выделяем операции:

1. Те, которые выполняет на объектом.

Операция создать - операция над классом. Когда мы используем эту операцию объекта еще нет. Объект появляется после выполнения этого метода (пример конструктор или какой-то статический метод, который порождает объект данного класса).

2. Те, которые выполняем над классом.

Для метода мы рисуем линию и должны показать, какие данные получает процесс, какие изменяет, какие возвращает. При создании результатом

явл. объект, мы это не указываем. Компоненты, которые получаем рисуем в “гробиках” (тоже имя и тип).

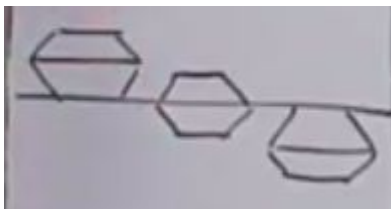


С правой стороны это операции класса.

С левой стороны это методы объекта.

Расположение гробиков:

1. Над линией - это данные, который метод получает.
2. На линии - изменяемые данные
3. Под линией - возвращаемые данные.



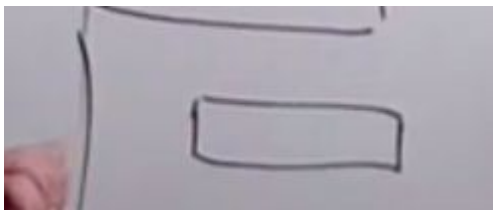
Если метод обрабатывает исключительную ситуацию мы у такого метода рисуем ромбик:



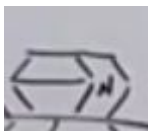
Если метод - обработчик состояния, то помечаем чертой



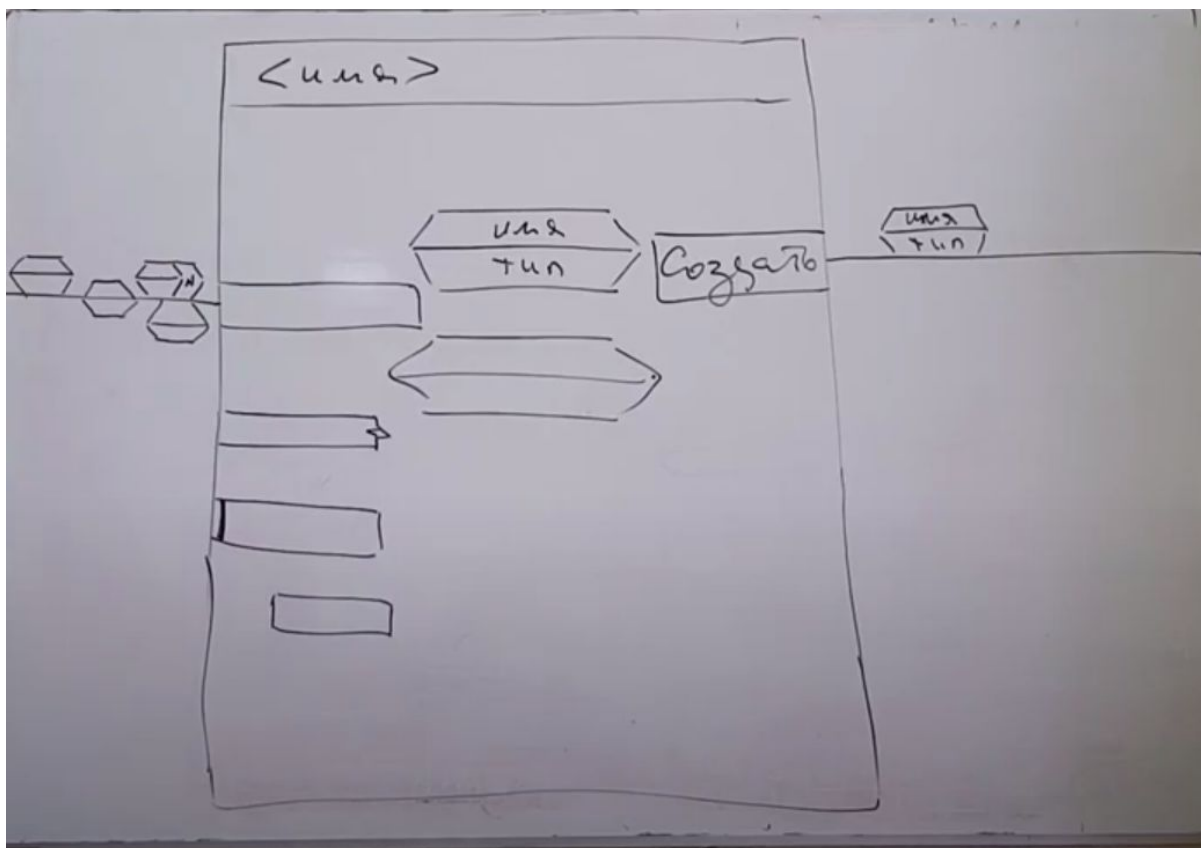
Если метод скрытый, то можно его пометить (но в основном нас интересует интерфейс). Он имеет только название и данные, какие он получает, мы не рассматриваем, потому что извне он ничего не получает (Рисуем внутри)



Если метод принимает или возвращает несколько данных одного типа, то мы рисуем “гробик в пространстве” и указываем “много” - N. Если получает несколько данных (параметров) этого типа.



Результат:

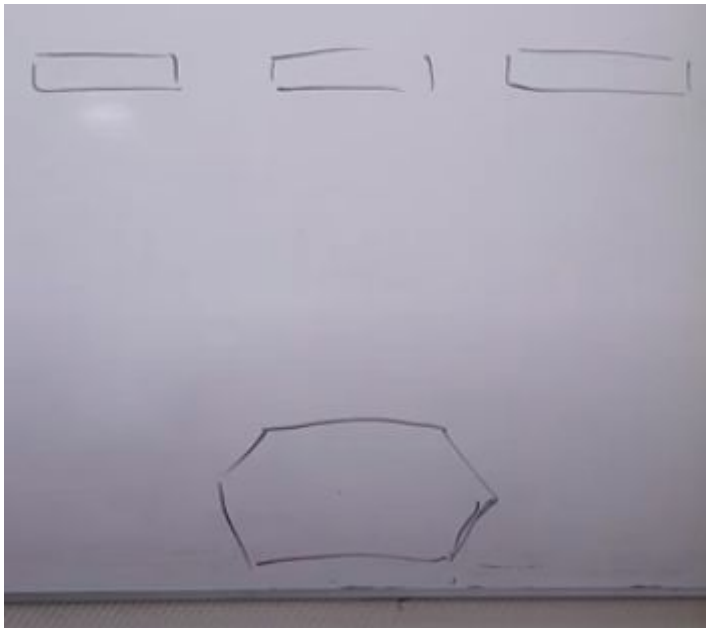


Структура класса:

Наша задача проследить потоки данных и доступ к данным самого класса. На этой диаграмме все строится вокруг данных объекта (то что на пдд называли архивом данных). Это данные экземпляра:



У нас есть три каких-то метода:



Линии - это линии по доступу к передаче данных.

На каждой линии мы показываем какие данные получает метод, а какие возвращает.

Мы рисуем гробики. И показываем куда идет передача этого данного.

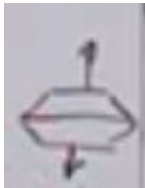
Стрелка вниз - это данное, которое принимает метод



Стрелка вверх - это данное, которое возвращает метод.



И вниз и вверх - то это изменяемые данные.



Мы четко выделяем слои. Слой методов, которые вызываются извне.



Также четко выделяются методы, которые не вызываются извне, это скрытые методы



Важно, чтобы методы интерфейса не вызывали друг друга. Если есть что-то общее, то мы выделяем метод скрытого слоя, который используется методами общедоступного слоя.

Если метод обрабатывает исключение:



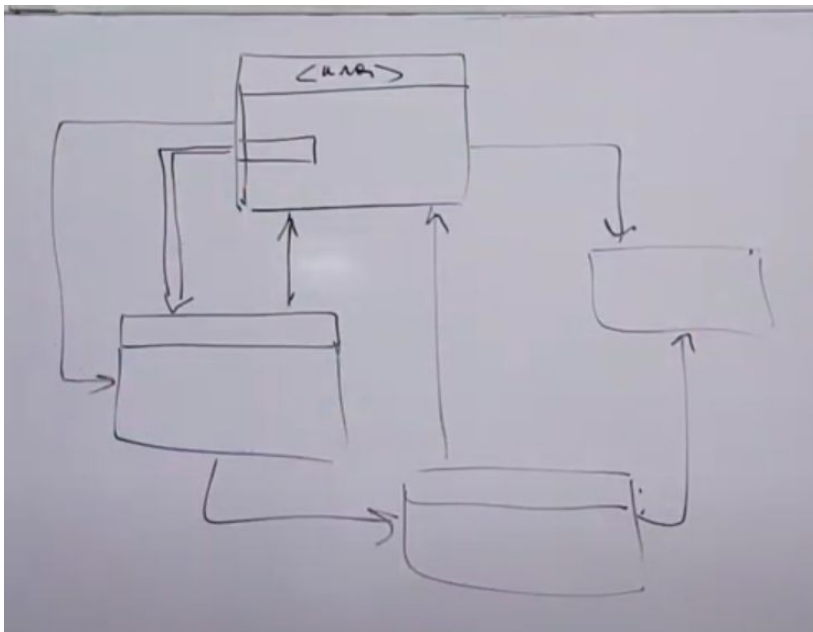
Если метод получает или передает (как результат) какие-то данные во внешний модуль, то мы показываем это вот таким образом:



На схеме структуре классов мы четко прослеживаем потоки данных и какие компоненты методы используют наши сущности нашего объекта, какие изменяют.

Диаграмма зависимостей:

Нам нужно имя класса и те методы, которые участвуют в взаимодействии.



Если у нас двойная стрелочка, то это дружественные связи.

Если одинарная, то просто схема использования.

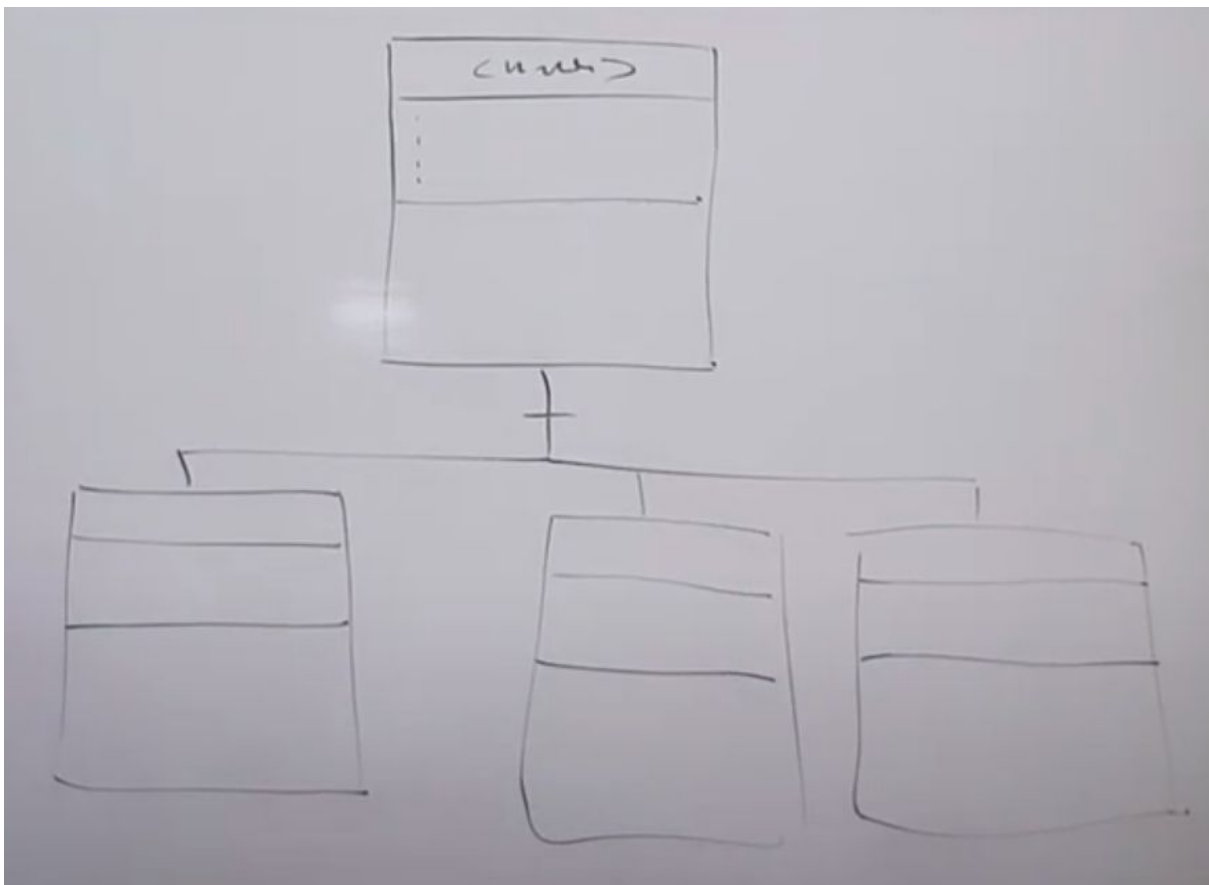
Мы можем показать, какой метод имеет доступ к методам другого класса. И какой метод вызывается (один метод вызывает другой метод).

Диаграмма зависимости - это могут быть аксессоры и те процессоры, которые мы выделили и которые могут использоваться другими объектами. Эти процессы мы выделяем как методы.

Диаграмма наследования:

Имя класса. Нас интересует методы и компоненты объектов. Можем разбивать компоненты и методы. Нужно понимать какие есть компоненты у супер класса и подклассов (производные классы) и какие методы.

Супер класс всегда абстрактный.



Архитектурный паттерн Конечная модель состояния (КМС)

В 4-м лр. в каждом обработчике состояний мы вынуждены устанавливать из каких состояний может быть переход в это состояние и это проверять.

Идея: задать возможные переходы и осуществлять эти переходы. КМС реализует эту идею - задания переходов.

Состоит из 3 классов:

1. Переход.
2. Конечная модель состояния (КМС).
3. Активный экземпляр.

Диаграмма классов:

Нам нужно задать переход:

Старое состояние (целое), новое состояние (целое) и событие (можно определять по номеру (целое)).

Операции и методы класса:

Создать переход, конструктор
передаем: старое состояние (целое), событие (целое), новое состояние (целое).

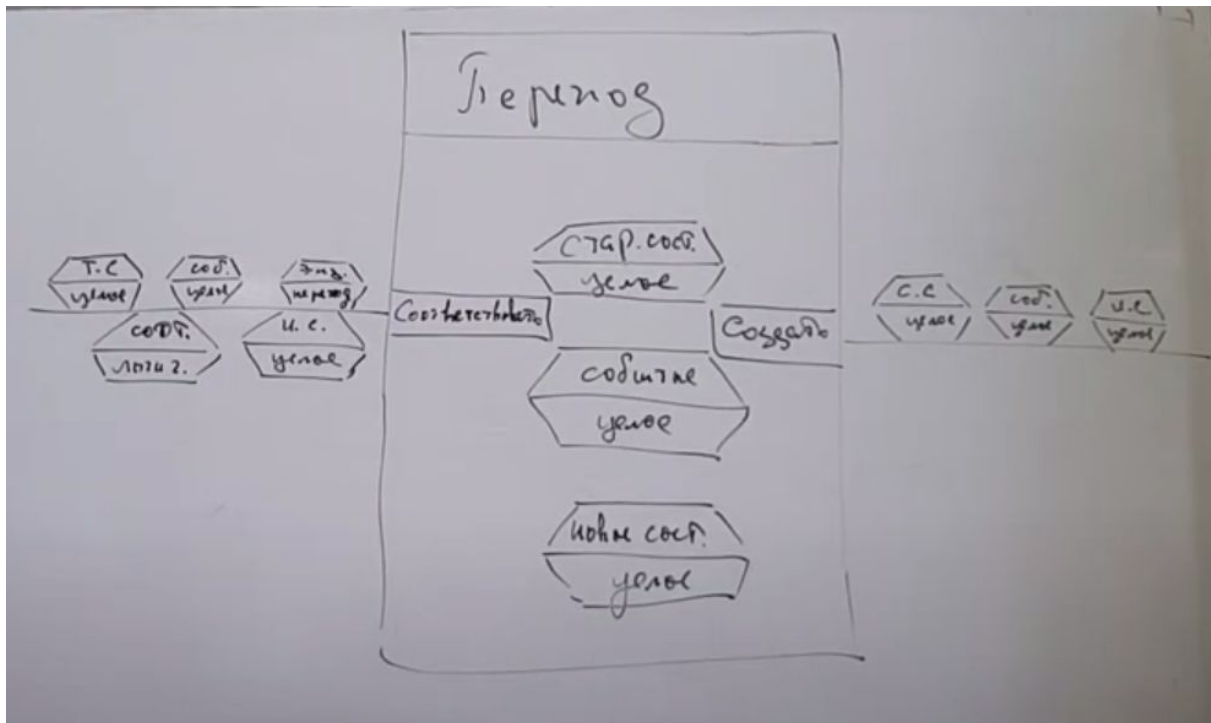
Операции и методы объекта:

- Перейти (соответствовать)

Узнать текущее состояние объекта (целое)

Получаем: т.с. текущее состояние (объект). Событие, которое происходит, и т.к. это метод объекта, мы получаем экземпляр объекта (типа переход)

Возвращаем: переход может не выполнен, значит мы возвращаем соответствие (логическое (bool) был переход или нет), если переход нашелся, то вернуть новое целое состояние.



Теперь КМС.

КМС должна содержать все возможные переходы.

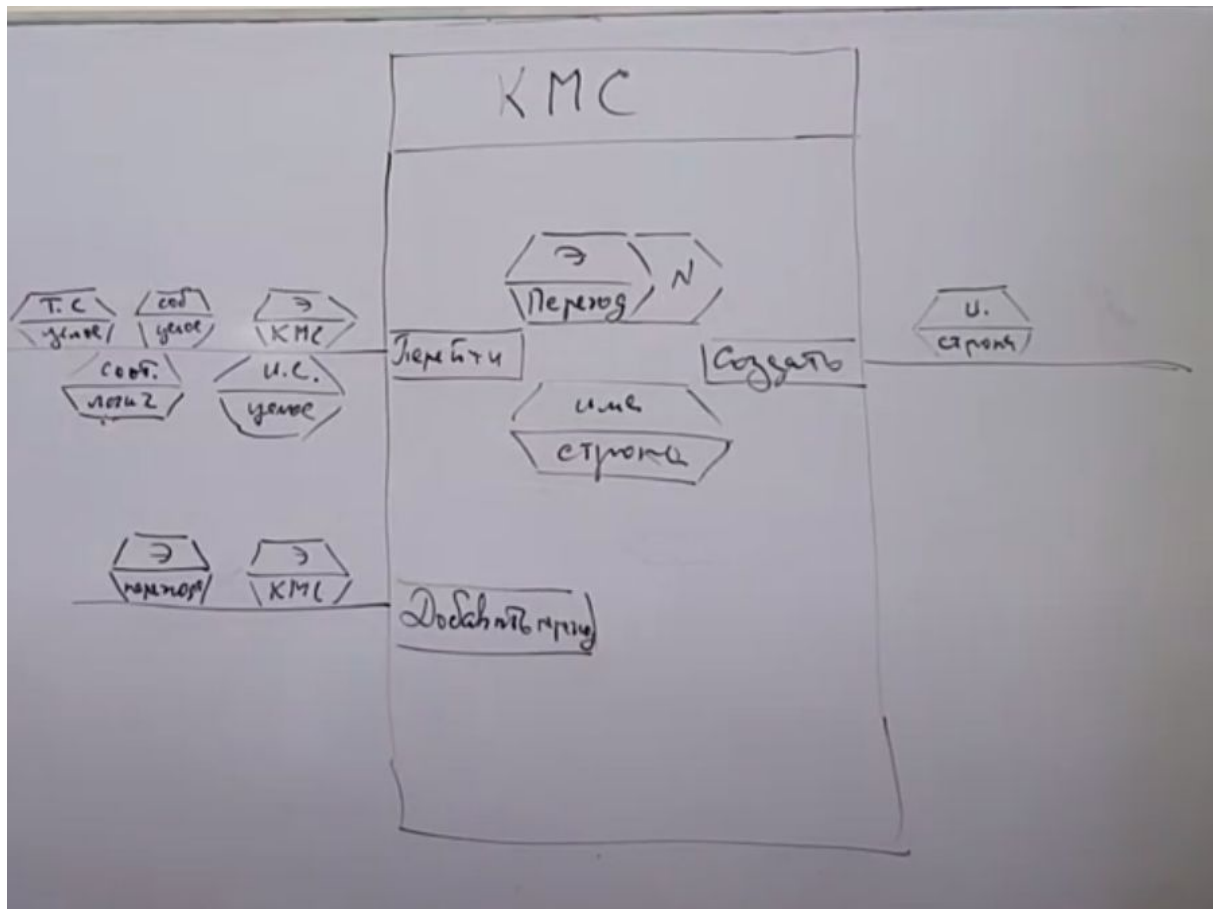
Метод: добавить переход.

Принимает: Экземпляр перехода и КМС.

Метод: перейти:

Принимает: текущее состояние (целое) , событие (целое), Экземпляр КМС.

Возвращаем: (Либо переходим, либо игнорируется (прочерк на табл. состояний)) Соответствие (есть ли такой переход (bool)) и новое состояние(целое).



Активный экземпляр:

Хранит только текущее состояние (целое).

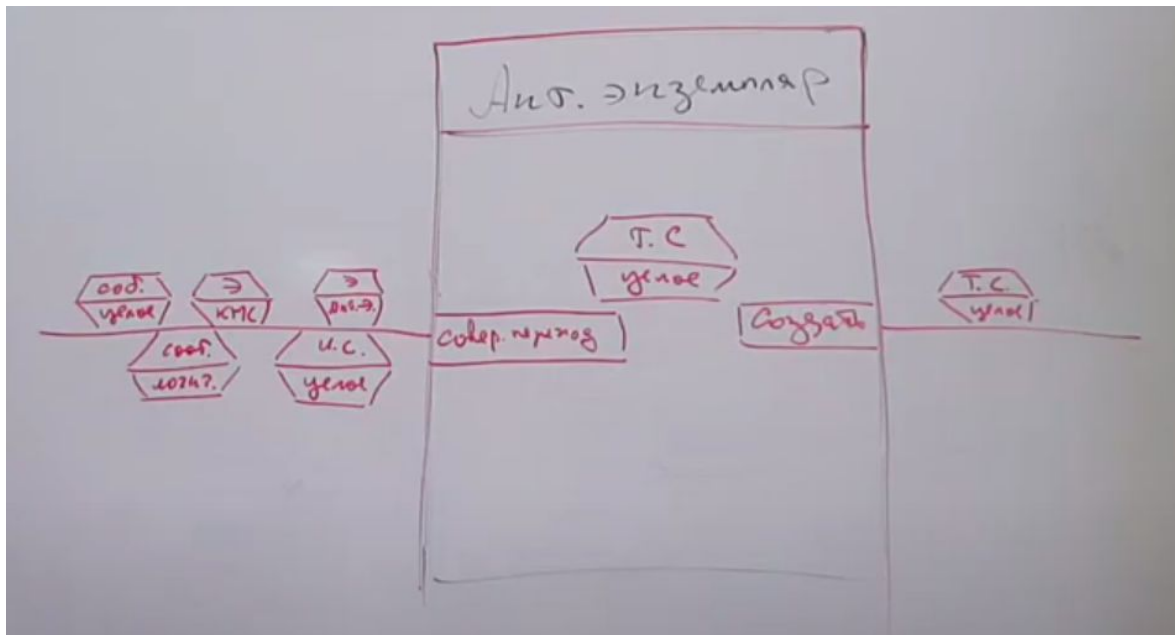
Метод создать Принимает текущее состояние активного экземпляра.

Метод перейти: (совершить событие?)

Происходит событие мы находимся в текущем состоянии. И нам нужно перейти. Чтобы совершить переход нам нужен экземпляр КМС.

Принимает: событие (целое), Элемент КМС, Элемент активный экземпляр.

Возвращает: соответствие(логическое, переход совершен или нет?).
и новое состояние (целое).



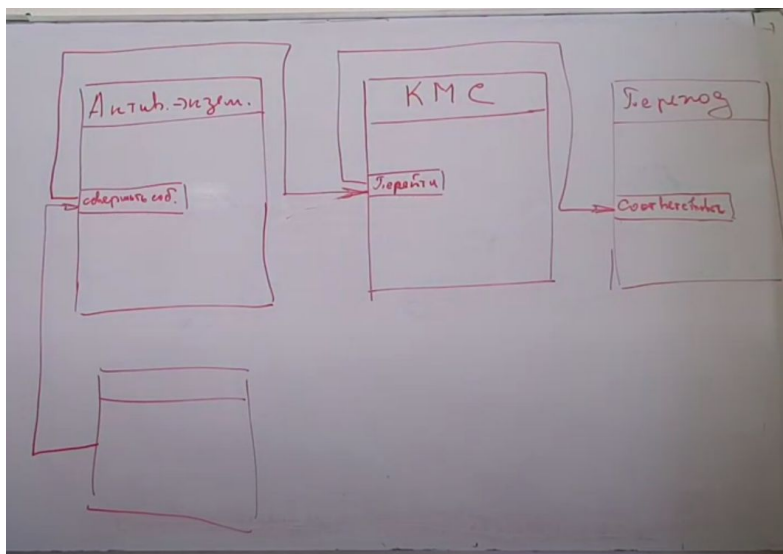
Теперь посмотрим как взаимодействуют эти объекты между собой (эти классы)

Имеем активный экземпляр. У него есть “совершить событие”

Есть КМС. У него есть перейти.

Есть переход. У него соответствовать.

У нас есть какой-то класс для которого мы хотим модель состояний. Будет вызываться соответствовать. В соответствовать передаем КМС и метод совершить событие вызывает переход. Переход просматривает возможные соответствия и выбирает подходящие. Просмотрев возможные переходы, находится подходящий переход и активный экземпляр меняет свое состояние на новое.



Идея паттерна: у нас есть текущее состояние. Произошло событие с текущим состоянием вызвался этот обработчик, мы проверяем, возможен ли переход и осуществляем его.