Юлия Николаевна

Технология разработки ПО

ТР ПО – совокупность процессов и методов программного продукта.

ТР ПО – система инженерных принципов для создания экономического ПО, которое эффективно и надёжно работает на реальных ПК.

Программный продукт – прога с документацией (полный комплект).

Технология программирования (как термин) был введён «Ершов Андрей Петрович.»

Разработка состоит из:

1. Анализ (требования, рынок и т.д.);
2. Проектирование;
3. Разработка (программирование);
4. Тестирование;
5. Отладка.

**Цели и задачи ТРПО**

Особенности современных информационных систем:

* Сложность описания;
* Наличие совокупности тесно связанных, взаимодействующих компонентов; (типо, есть банк, а есть банкомат, и они связаны)
* Отсутствие прямых аналогов; (нужно придумывать чё-нить новое, а не копипастить уже созданное)
* Необходимость интеграции существующего, и разработанного приложения; (если сделали моб. прогу, не нужно переделывать основу)
* Функционирование в неоднородной среде, на нескольких аппаратных платформах;
* Разобщённость и разнородность групп разработчиков по уровню квалификации и использованию инструментальных средств;
* Временная протяжённость проекта.

ПО – полный набор, или часть программ (процедур, правил) и связанный с ними документации.

ПО – интеллектуальный продукт, не зависящий от среды, на которой он написан.

Программные средства = ПО

Программных продукт – набор компьютерных программ, процедур и док-ции, предназначенный для передачи пользователю.

Смертельный марш – (ввёл Э. Йордан) под ним понимается проект, параметры которого откланяются от нормы на 50%+ (Хотя бы одного):

1. план проекта сжат более чем на половину;
2. кол-во разрабов уменьшено на 50%;
3. Бюджет урезается более чем на половину;
4. Более чем на половину увеличивается функционал.

Кризис программирования – появился в начале 70-х в USA, выражался в том, что большие проекты стали выполнятся с отставанием, с превышением сметы расходов, разработанный продукт не обладал требуемыми функциональными возможностями и херовым качеством. (в 95 году примерно 16% проектов были норм.)

**Сложная система**

Сложная система – система, состоящая из взаимодействующих составляющих (подсистем), в следствии чего, она приобретает новые св-ва.

Признаки сложной системы:

1. Наличие общей задачи;
2. Большое кол-во компонентов;
3. Возможность декомпозиции системы;
4. Иерархическая архитектура системы;
5. Сложность поведения системы;
6. Устойчивость системы к внешним воздействиям;
7. Высокая надёжность системы.

**Понятие и основные этапы жизненного цикла ПО**

Жизненный цикл ПО – момент жизни проги, начиная с момента, в котором она задумывается, и оканчивающийся в тот момент, когда заканчивается её поддержка.

Стандарт ISO/IEC 12207 – (от 1995 года) регламентирует жизненный цикл ЖЦ ПО

Процесс – совокупность взаимосвязанных действий, которые преобразуют input to output. (согласно ISO/IEC 12207)

Каждый процесс хар-ся задачами и методами.

Процесс делится на действие, а действие на задачи.

В соответствии со стандартом, процессы делятся на:

1. Основные процессы (процессы без которых ничего работать не будет);
   1. Процесс приобретения (действие заказчика);
   2. Процесс поставки (действие разработчика, а точнее компании по сдаче продукта);
   3. Процесс разработки (создание ПО и компонентов);
   4. Процесс эксплуатации;
   5. Поддержка;
2. Вспомогательные;
   1. Документирование (формализованное описание информации в течении ЖЦ ПО);
   2. Управление конфигурацией (проджект-мененджеминг);
   3. Процесс обеспечения качества;
   4. Процесс верификации (проверка на соответствие требованиям);
   5. Процесс совместной оценки;
   6. Процесс аудита (Проверка ПО сторонней компанией);
   7. Разрешение проблем;
   8. Процесс аттестации (оценка достоверности проведённого тестирования).
3. Организационные
   1. Процесс управления
   2. Процесс создания инфраструктуры (выбор ПО, раб места и т.п.)
   3. Процесс усовершенствования (оценка измерение усовершенствование ЖЦ ПО
4. Процесс обучения (имеется в виду обучение заказчика и его персонала пользованию ПО)

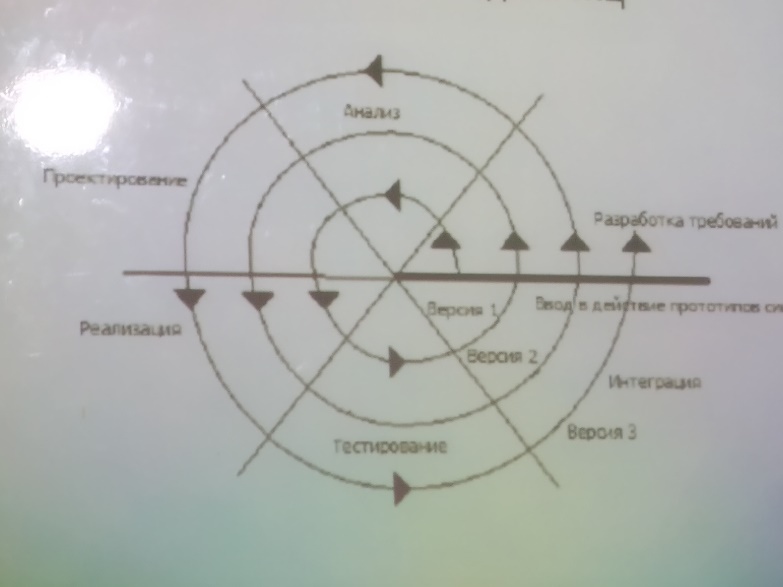
Модели ЖЦ ПО со стандартом ISO/IEC 12207

Модель ЖЦ ПО – структура определяющая последовательность выполнения и взаимосвязи процессов, действий задач на протяжении ЖЦ. Она зависит от специфики, масштаба и сложности проекта  
Стадия создания ПО – часть процесса создания ПО, ограниченная временными рамками и заканчивающаяся выпуском конкретного продукта

1. Стадии ЖЦ ПО:  
   Формирование требований к ПО  
   Проектирование  
   Реализация  
   Тестирование  
   Ввод в действие  
   Эксплуатация и сопровождение  
   Снятие с эксплуатации

Две модели ЖЦ ПО: Каскадная (1970-80):

**Илья забыл написать, так что представим, что тут модель, в которой всё идёт последовательно. Если будет не лень – когда-нить вставлю пикчу.**

Спиральная (1986-90):  


**Как я понял, суть темы: последовательность «разработка требований > анализ > проектирование > реализация > тестирование > интеграция» прогоняется по кругу до тех пор, пока текущий Билд не будет устраивать заказчика или его «карман».**

Прототип – действующий программный компонент, реализующий часть функций и часть интерфейсов программы.

**Управление требованиями системы**

Требования – условие которому должна соответствовать система;

Анализ требований *–* предварительно определение того, что не удастся реализовать, перед тем, как реализация провалится;

Оценка требований – доступное и умелое разъяснение заказчикам требований, которые не будут реализованы;

Управление требованиями – убеждение заказчика в том, что хочет именно он (но то, что вы можете реализовать);

Управление требованиями – системный подход, систематический подход к обнаружению, организации, документированию и сопровождению изменяющихся требований к системе.

Необходимость требования к системе:

1. Добиться понимания с заказчиком, как должна выглядеть система;
2. Дать разработчикам понимание требований к системе;
3. Определить границы системы;
4. Обеспечить базу для оценки стоимости и времени для разработки
5. Обеспечить базу для планирования итераций;
6. Определить пользовательский интерфейс;

Виды требований:

1. Функциональные – описание поведения и сервисов системы, её функционала. Определяют то, что должна система делать;
2. Нефункциональные – определяет надёжность, особенности поставки уровень качества и т.д.

Уровни требований:

1. Бизнес – определяет цели клиента или заказчика ПО (**ПОЧЕМУ**?);
2. Пользовательские – определяет цели/задачи которые должны выполнятся пользовательской системой при помощи создаваемого ПО (**КТО**? И **ЧТО**?);
3. Функциональные – функциональность ПО, которая должна быть создана разрабами для предоставления возможностей выполнения пользователями своих обязанностей в рамках бизнес-требований (**ЧТО**?);

**P.S. капсом+<b> писал не я, так было на доске**

Свойства требований:

1. Ясность, недвусмысленность;
2. Полнота и непротиворечивость;
3. Необходимый уровень детализации;
4. Прослеживаемость;
5. Тестируемость;
6. Модифицируемость;

Методы выявления требований:

1. Интервьюирование – получение получения инфы от пользователя во время интервью;
2. Сценарии – приведение примеров из реала вместо абстрактных значений. Продумываются сценарии работы;
3. Этнография – аналитик внедряется в будущее окружение системы и наблюдает повседневную работу. Тем самым выявляя не выявленные требования;
4. Прототипы – создаются предварительные модели (вплоть до бумажных);
5. Наблюдение – непосредственное наблюдение аналитиков за пользователем;
6. Выявление требований на основе опорных точек зрения;

**Отличие наблюдения от этнографии в том, что наблюдение со стороны пользователя;**

**Методы выявления требований могут комбинироваться и дополнятся.**

Документирование требований (документы):

1. Обзор продукта (внешнее описание системы);
2. Системные модели (диаграммы прецедентов, деятельности, user story);
3. ТЗ (техническое задание);
4. Частичные ТЗ;

Управление изменениями требований:

* Анализ проблем – анализ проблем и предложений их изменений;
* Анализ примечаний – чекается результат изменений;
* Реализация изменений – реализация.

Работа по управлению требований;

* Обнаружение требований и их документирование;
* Установление между заказчиком и исполнителем соглашений;
* Отслеживание изменений и оценка их влияния на процесс.

**RAD**

**(Rapid Application Development)**

**(Методология быстрой разработки)**

RAD Включает:

1. Короткий график разработки;
2. Маленькая команда разрабов;
3. Повторяющиеся циклы (Требования дополняются общением с заказчиком);
4. Команда разработчиков должна быть лютыми многофункциональными профессионалами.

Главная идея методологии RAD – как можно быстрее донести до заказчика рез-ты работы.

ЖЦ ПО в соответствии RAD:

1. Анализ и планирование требований – узнаём, что требует заказчик, и узнаём приоритетность требований, ограничивается масштаб проекта и временные рамки для каждой фазы. Выясняется, если смысл в реализации данного проекта в текущих финансовых условиях.  
   **В результате** фазы мы получаем список и приоритетность функций, предварительная функциональная и информационная модель;
2. Проектирование – Разработчики и конечные пользователи участвуют в техническом проектировании. Используются CASE-средства, для быстрого получения прототипа. Более подробно рассматриваются процессы в системе. Анализируется и корректируется функциональная модель. Каждый процесс рассматривается детально. Определяются требования к разграничению данных. Оценивается кол-во функциональных моментов, разрабатываемых системой, для разбиение её на подсистемы (Прога разбивается на более мелкие подсистемы), для разработки за приемлемое время (60-90 дней)  
   **В результате** – общая информационная модель, нормальная модель системы и подсистемы, определены интерфейсы, между автономно работающими подсистемами, построенные прототипы экранов и отчётов и диалогов с системой;
3. Построение – Пишется код проги. Пишется быстро из-за того, что часть кода генерится с помощью CASE-средств. Во время разработки осуществляется тестирование каждой подсистемы. Завершается физическое проектирование системы. Производится анализ использования данных и делаются БД. Определяются требования к аппаратным рес-ам. Завершается раз-ка документации.  
   **В результате ­**– готовая система, удовлетворяющая всем требованиям;
4. Внедрение – производится обучение пользователей. Фаза короткая, за сим, планируется заранее  
   **В результате**  - внедрённая информационная система;

Методология RAD не универсальная, подходит для небольших проектов, которые разрабатываются для конкретного заказчика.  
Методология RAD не подходит для сложных расчётных программ, ОС.

Оценка размеров приложений в RAD делается на основании функциональных элементов (меньше 1000 элементов – разрабатывает 1 чел, От 1000 до 4000 – тима)  
Основные принципы RAD:

1. Итерационная разработка;
2. Необязательно завершать работу на каждом этапе ЖЦ ПО;
3. Применение CASE-средств;
4. Участие конечных пользователей в разработке;
5. Разработка прототипов;
6. Тестирование параллельно разработке;
7. Разработка подсистем командами;

**Восходящее и нисходящее проектирование**

Технология нисходящего проектирования с пошаговой детализацией является неотъемлемой частью создания хорошо структурированных программ. *При написании программы с использованием этой технологии вся задача рассматривается как единственное предложение (вершина), выражающее общее назначение программы*. Так как вершина редко отображает достаточное количество деталей, на основании которых можно написать программу, то поэтому надо начинать процесс детализации – функциональной декомпозиции*. Вершина разделяется на ряд более мелких задач (функций)* в том порядке, в котором эти задачи должны выполнятся. В результате получим первую детализацию. *Далее каждая из подзадач разбивается на подзадачи*, принадлежащие второму уровню детализации. Программист завершает процесс нисходящей разработки с пошаговой детализацией, когда алгоритм настолько детализирован, чтобы его можно было бы преобразовать в программу.

**(Суть технологии нисходящего проектирования в том, что мы берём как бы цель программы, которую мы хотим написать, и разбиваем её на более мелкие цели, а после этого, делим полученное ещё на части. И так до тех пор, пока не получится такой алгоритм, что его можно будет преобразовать в прогу)**

**(При разработке проги с помощью нисходящего проектирования, сначала пишется «ядро» проги, а все подмодули затыкаются заглушками. После, эти заглушки заменяются на полноценные части)**

Важно, что при декомпозиции используются только указанные выше три управляющих конструкции, что позволяет говорить о структурной декомпозиции или структурном проектировании программ.

Недостатки нисходящего проектирования **–** необходимость заглушек, и сложно определить размер ПО и его эксплуатационные хар-ки (до окончания написания ПО).

Преимущество нисходящего проектирования – Хорошо написанное ядро.

**Вывод**: пошаговая реализация это - тактика разработки программы, а нисходящее проектирование — это стратегия программирования.

При восходящем проектировании разработка идет снизу-вверх. На первом этапе разрабатываются модули самого низкого уровня. На следующем этапе к ним подключаются модули более высокого уровня, и проверяется их работоспособность. На завершающем этапе проектирования разрабатывается головной модуль, отвечающий за логику работы всего программного комплекса.

Преимущество восходящего программирования – не нужно писать заглушки.

Недостаток восходящего программирования – головной модуль разрабатывается на завершающем этапе проектирования, что порой приводит к необходимости дорабатывать модули более низких уровней.

**Метод структурного проектирования**

Метод структурного проектирования — это регламентированная последовательность действий, которая позволяет разработать структуру аппаратных и программных средств выстраиваемой системы, удовлетворяющих техническим требованиям к проектируемому устройству (программному продукту).

**Шаги структурного проектирования**

1. Более полное описание технических требований;
2. Разбиение таска на подтаски;
3. Создание графического образа работы системы;
4. Детальное определение функций каждого блока (речь идёт о схеме, разработанной во время предыдущего шага);
5. Написание исходного кода на «С» или «ассемблер». Под этим подразумевается не только написание кода, но и его отладка. Вот 3 способа проектирования:
   1. При проектировании «сверху-вниз» пишется и отлаживается сначала main (всё что не написано – заменяется на временные конструкции), а потом всё остальное;
   2. При проектировании «снизу-вверх». При этом пишутся и отлаживаются не соединённые функции, а потом пишется и отлаживается ядро, которое соединяет их;
   3. Попеременно пишутся и отлаживается и верх и них, а на среднем этапе соединяется;
6. Определение критериев того, что прога готова к сдаче.

Полная документация на ПО содержит комментарии, письменное описание принципов построения программы, технические условия на написание программы, исходный текст программы, руководство пользователя, историю разработки программы и ее модификаций. Весь этот набор документов может быть разделен на внешнюю и внутреннюю документацию.

Внешняя документация состоит из специально написанного текста, который поясняет, как работает программа.

Внутренняя документация включает полный набор читаемых и понимаемых документов, который необходим для легкого исправления кода. Внутренняя документация состоит из комментариев, исходного текста программы, структуры и специальных записей.

**Структурированная программа**

Каждая программа может быть создана на основе элементарных базовых конструкций 3-типов:

1. Простой вычислительной последовательности (Последовательное преобразовании вычислительных данных);
2. Выбора или альтернативы (Проверка условия, и выбор одного из двух и более);
3. Повторения или итерации (Оператор или группа итераторов повторяется 1+ раз).

**Так же, есть ещё конструкции, но они поломанные и стоит ограничить их использование (Типо “GO TO”).**

Структурированной считается прога которая:

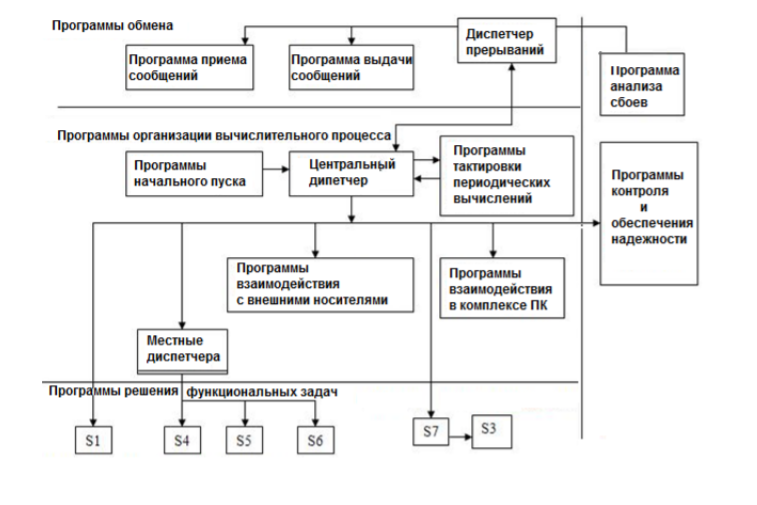
1. Не имеет переходов внутрь циклов или условных операторов;
2. Не имеет выход из внутренней части циклов и условных операторов;
3. Число итераций должно быть задано до входа в цикл;
4. Ограничено использование GO TO.

**Типовая структура программного комплекса**

Каждый программный комплекс включает:

1. Программы решения функциональных задач;
2. Программы обмена информации с внешними абонентами;
3. Программы организации вычислительного процесса;
4. Программы контроля и обеспечения надёжности.

Графически это можно представить, как:

**Проблема сложности больших систем**

Если кратко: текущие системы оч большие, и человеческий мозг тупо не может уследить за всем этим безобразием, так что «глупые юдишки» придумали способ как сделать такой проект исполнимым – разделить его на более мелки части. Такой принцип позволяет при разработке подсистемы помнить только о ней, а не о всём проекте.

Правильная (рабочая, а не как обычно) декомпозиция означает:

1. Кол-во связей между отдельными подсистемами должно быть минимальным;
2. Связность отдельных частей внутри каждой подсистемы должна быть максимальной;
3. Каждая подсистема должна инкапсулировать своё содержимое;
4. Каждая система должна иметь свой интерфейс с другими подсистемами.

**Структурный подход к разработке ПО**

У структурного подхода (его именуют «функционально-модульным или структурным») в основе лежит принцип функциональной декомпозиции, при которой структура программы описывается терминами иерархии её функций и передачи информации между отдельными функциональными эл-тами.

Второй, объектно-ориентированный подход, использует объектную декомпозицию. При этом структура системы описывается в терминах объектов и связей между ними, а поведение системы описывается в терминах обмена сообщениями между объектами.

Итак, сущность структурного подхода к разработке ПО ЭИС, заключается в разбиении его на автоматизируемые функции.

Базовые принципы структурного подхода:

1. Абстракция;
2. Формальность (каждая программа должна реализовывать определённый алгоритм);
3. Divide at impera (Разделяй и властвуй. Согласно нему, нужно всё крупное, делить на более мелкое. И если надо, можно повторить);
4. Модульность (Ускоряет создание больших программ, за счёт использования ранее созданных описаний);
5. Открытости (должны легко модифицироваться).

В структурном подходе используется 2 группы средств, в каждой группе есть модели (диаграммы), наиболее распространённые из них:

1. DFD (Data Flow Diagrams) - диаграммы потоков данных;
2. SADT (Structured Analysis and Design Technique — метод структурного анализа и проектирования) — модели и соответствующие функциональные диаграммы;
3. ERD (Entity-Relationship Diagrams) — диаграммы "сущность-связь".

Конкретный вид перечисленных диаграмм зависит от стадии ЖЦ ПО:

**На стадии формирования требований** к ПО SADT-модели и DFD используются для построения модели "AS-IS" и модели "ТО-ВЕ", отражая, таким образом, существующую и предлагаемую структуру бизнес-процессов организации и взаимодействие между ними (использование SADT-моделей, как правило, ограничивается только данной стадией, поскольку они изначально не предназначались для проектирования ПО). С помощью ERD выполняется описание используемых в организации данных на концептуальном уровне, не зависимом от средств реализации базы данных (СУБД).

**На стадии проектирования** DFD используются для описания структуры проектируемой системы ПО, при этом они могут уточняться, расширяться и дополняться новыми конструкциями. Аналогично ERD уточняются и дополняются новыми конструкциями, описывающими представление данных на логическом уровне, пригодном для последующей генерации схемы базы данных. Данные модели могут дополняться диаграммами, отражающими системную архитектуру ПО, структурные схемы программ, иерархию экранных форм и меню и др.

**Функциональное моделирование**

При функциональном моделировании, взаимодействие системы с окружающим миром описывается как вход (нечто, что перерабатывается системой), выход (результат деятельности системы), управление (стратегии и процедуры, под управлением которых производится работа) и механизм (ресурсы, необходимые для проведения работы). Находясь под управлением, система преобразует входы в выходы, используя механизмы.

**Функциональное моделирование** - это процесс моделирования функций, выполняемых рассматриваемой информационной системой/объектом, путем создания описательного структурированного графического изображения, показывающего что, как и кем делается в рамках функционирования объекта и объектов, связывающих эти функции, с учетом имеющейся информации.

Цель создания функциональной модели – точная спецификация всех функций, осуществляемых в рамках процесса более высокого уровня иерархии, а также характера взаимосвязей между ними. Так можно получить информацию обо всех потоках и материалах внутри проги.

Функциональная модель сети процессов обработки информации должна создаваться в соответствии со следующим примерным перечнем требований:

1. Строиться с точки зрения руководства предприятия.;
2. Содержать процессы, определенные как обязательные в рамках требований соответствующей нормативно-технической документации;
3. Охватывать все стадии жизненного цикла продукции, относящиеся к сфере деятельности предприятия.

Бизнес-процесс – совокупность процессов, результатом которых является продукция.

Процесс в функциональной модели – есьм процесс преобразования входных материалов в продукт.

**Моделирование потоков данных**

В основе лежит построение модели ИС.

Таким образом, основными компонентами диаграмм потоков данных являются:

1. Внешние сущности;
2. Системы/подсистемы;
3. Процессы;
4. Накопители данных; П
5. Потоки данных.

**Моделирование данных**

Цель моделирования данных состоит в обеспечении разработчика ИС концептуальной схемой базы данных в форме одной модели или нескольких локальных моделей, которые относительно легко могут быть отображены в любую систему баз данных. Наиболее распространенным средством моделирования данных являются диаграммы "сущность-связь" (ERD). С их помощью определяются важные для предметной области объекты (сущности), их свойства (атрибуты) и отношения друг с другом (связи). ERD непосредственно используются для проектирования реляционных баз данных.

**Методологические основы Case-технологии**

*Computer Aided Software Engineering*

Использование Case-средств подразумевает разработку сложных информационных систем:

1. Создание и сопровождение ИС (информационные системы)
2. Анализ
3. Формулировка требований, проектирование БД.
4. Генерация кода, тестирование и документирование.
5. Управление проектом.

Кейс средства используют структурный анализ и ООП подход.

**Польза от использования кейс-средств:**

1. Высокий уровень технической поддержки процессов разработки и сопровождения.
2. Положительное воздействие на производительность, качество, документирование и соблюдение стандартов.
3. Отдача от внедрения кейс-средств.

**Составляющие кейс-средств:**

1. Единый графический язык и правила.
2. Графические редакторы.
3. Генераторы.
4. Репозиторий.

**Недостатки использования кейс-средств:**

1. Эффект не всегда моментальный.
2. Затраты на внедрение могут превысить затраты на приобретение.

**Концептуальная модель UML и её элементы**

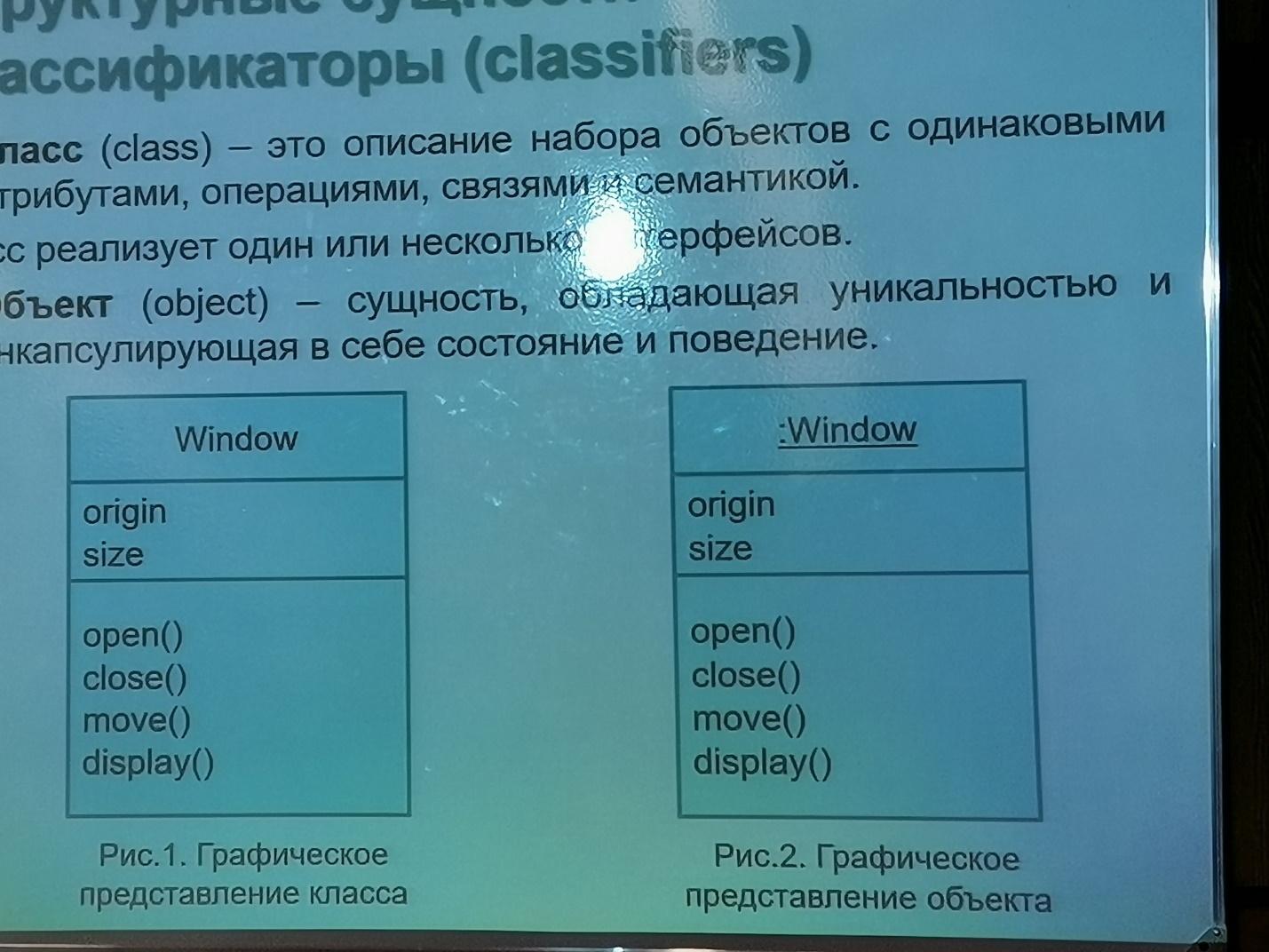
Модель UML– совокупность конечного множества конструкций языка, главные из которых – **сущности** и **отношения между ними**.

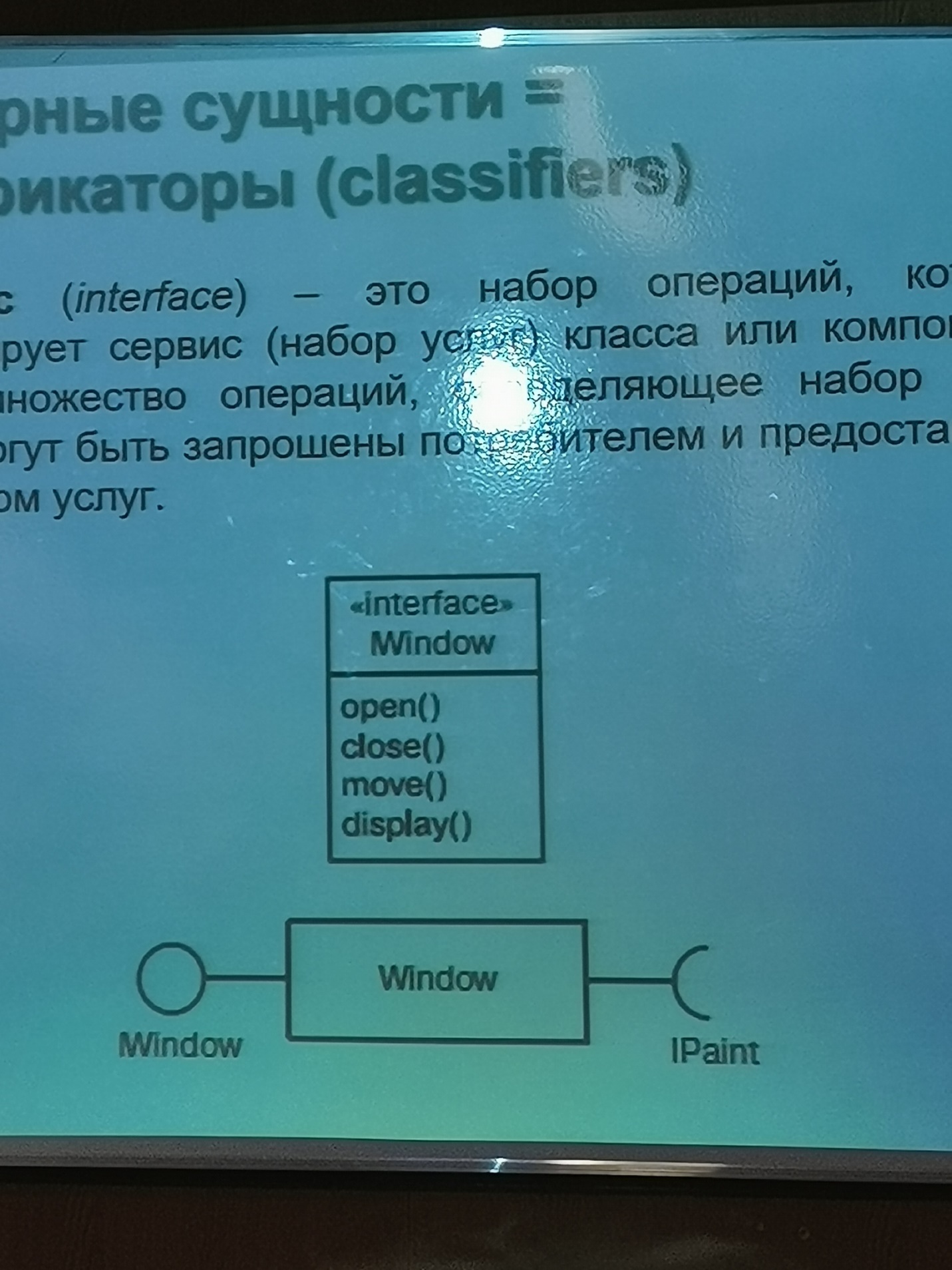
Сущности – абстракции, которые являются основными эл-тами модели, связи соединяют их между собой, диаграммы группируют представляющие интерес наборы сущностей.

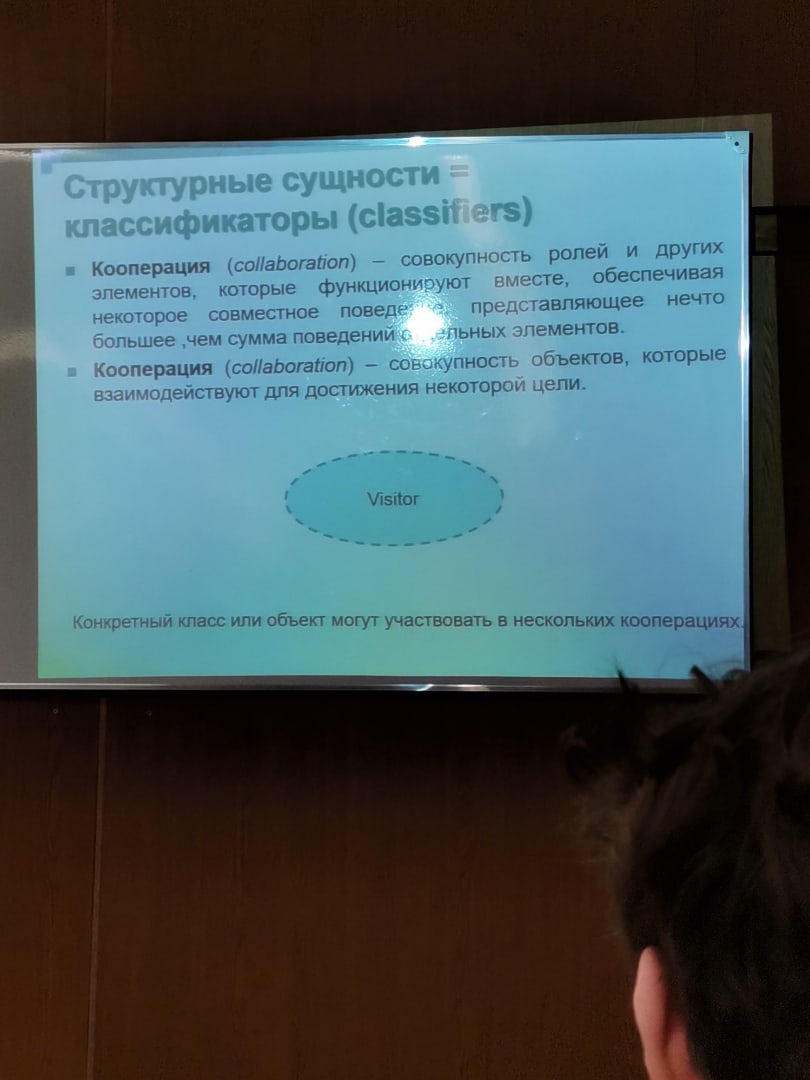
**Сущности бывают**:

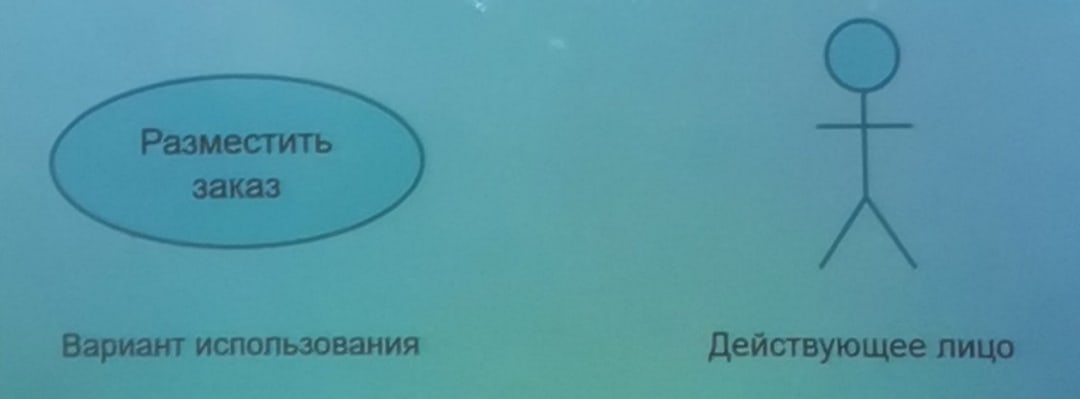
* Структурные:
  + Класс – описание набора объектов с одинаковыми атрибутами, операциями, связями и семантикой;
  + Объект – сущность, обладающая уникальностью и инкапсулирующая в себе состояние и поведение;
  + Интерфейс – набор операций, который специфицирует сервис (набор услуг) класса или компонента т.е. множество операций, определяющий набор услуг которые могут быть запрошены потребителем и предоставлены поставщиком услуг;
  + Кооперация – совокупность ролей и других элементов, которые функционируют вместе, обеспечивая некоторое совместное поведение представляющее нечто большее, чем сумма поведений отдельных элементов;
  + Вариант использования (case) – описание последовательности действий, выполняемых системой и приносящих значимый результат конкретному действующему лицу;
  + Действующее лицо (actor) – сущность, находящаяся вне моделируемой системы и непосредственно взаимодействующая с ней;
  + Активный класс – класс, объекты которого являются владельцами одного или нескольких процессов или потоков, и таким образом, могут инициировать управляющие воздействия;
  + Компонент – модульная часть системы, которая скрывает свою реализацию за набором внешних интерфейсов;
  + Артефакт – элемент информации, который используется или порождается в процессе разработки ПО.(файлы исходного кода, скрипты, исполняемые программы);
  + Узел – физический элемент, который существует во время исполнения и представляет собой вычислительный ресурс, имеющий по меньшей мере некоторую память и частно – вычислительные возможности;
* Поведенческие:
  + Взаимодействие – поведение, которое заключается в обмене сообщениями между наборами объектов или ролей в определённом контексте для достижения некоторой цели;
  + Автомат – поведение, характеризуемое последовательностью состояний объекта, в которых он оказывается на протяжении своего жизненного цикла в ответ на события, вместе с его реакцией на эти события. Делится на:
    - Состояния;
    - Переходы;
    - События
    - Действия.
  + Деятельность – определяет последовательность шагов процесса вычислений. Отдельный шаг – действие.
  + Пакет – механизм общего назначения для организации проектных решений, который упорядочивает конструкции реализации. Вариации пакетов: подсистемы, каркасы;
* Аннотирующие:
  + Примечание – простой символ, служащий для описания ограничений или комментариев, относящихся к элементу модели.
* Группирующие.

Пикчи:

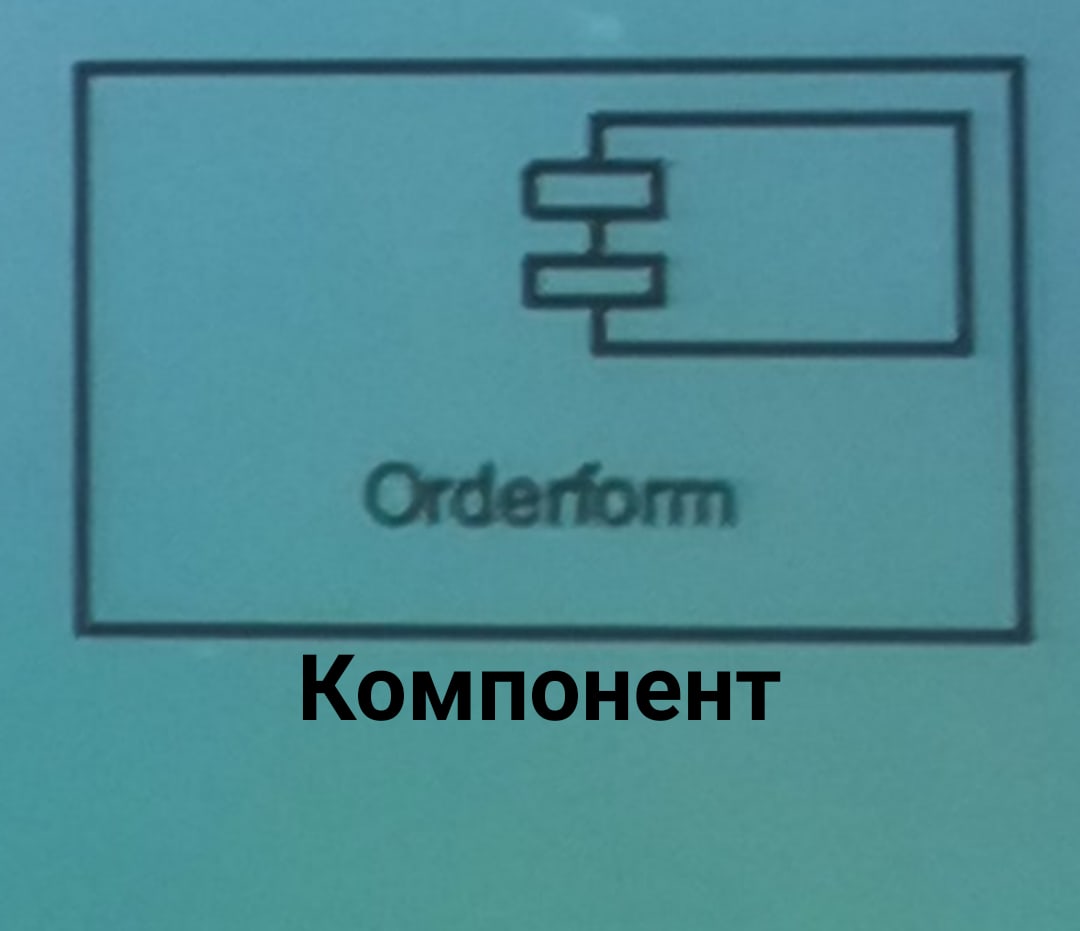




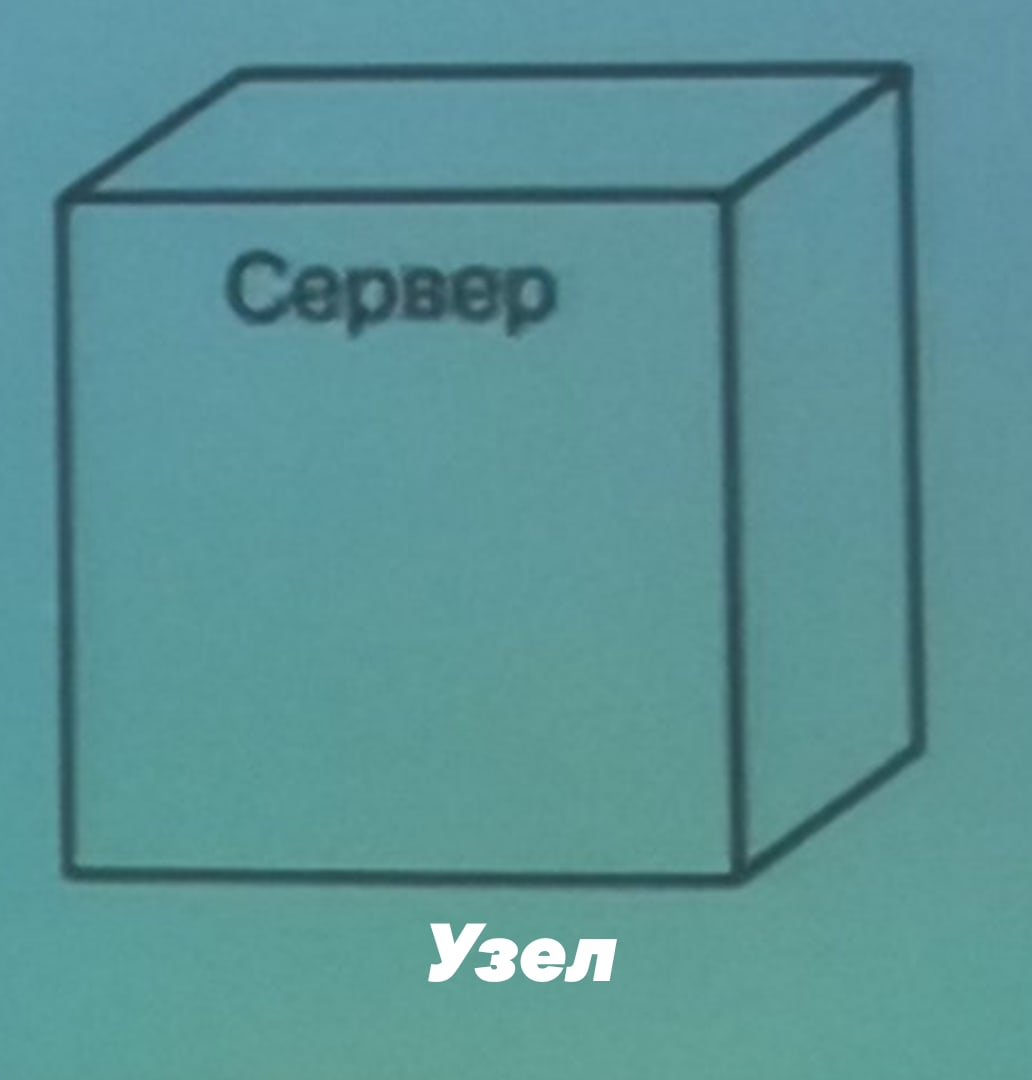


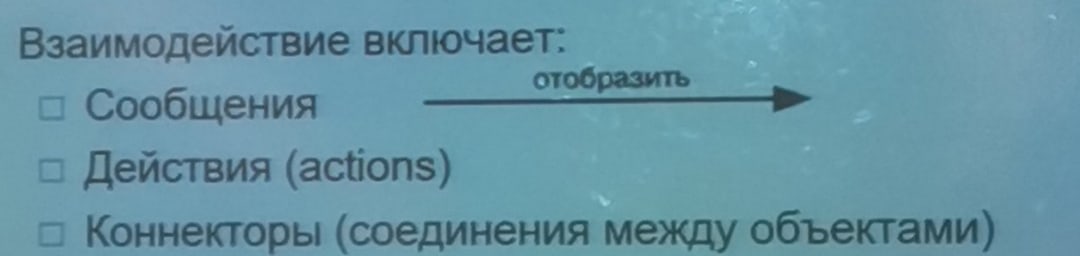


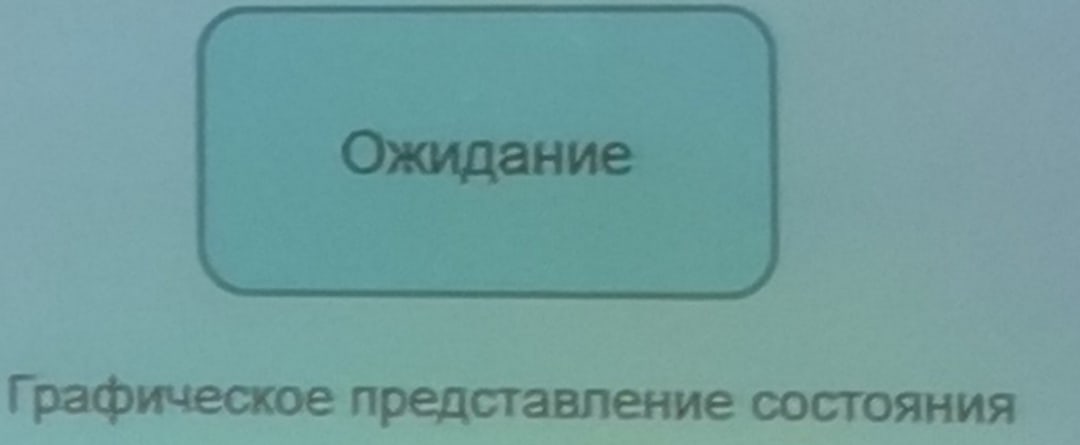


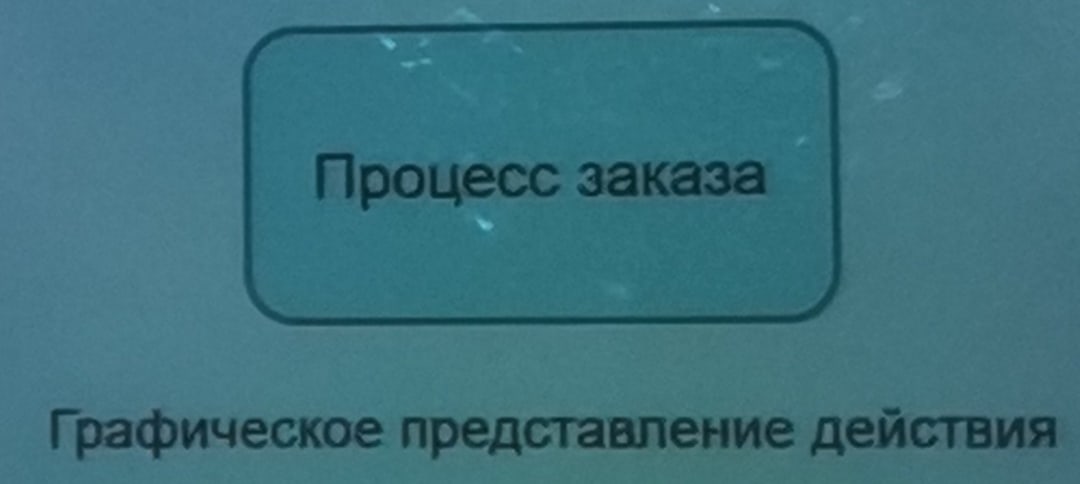




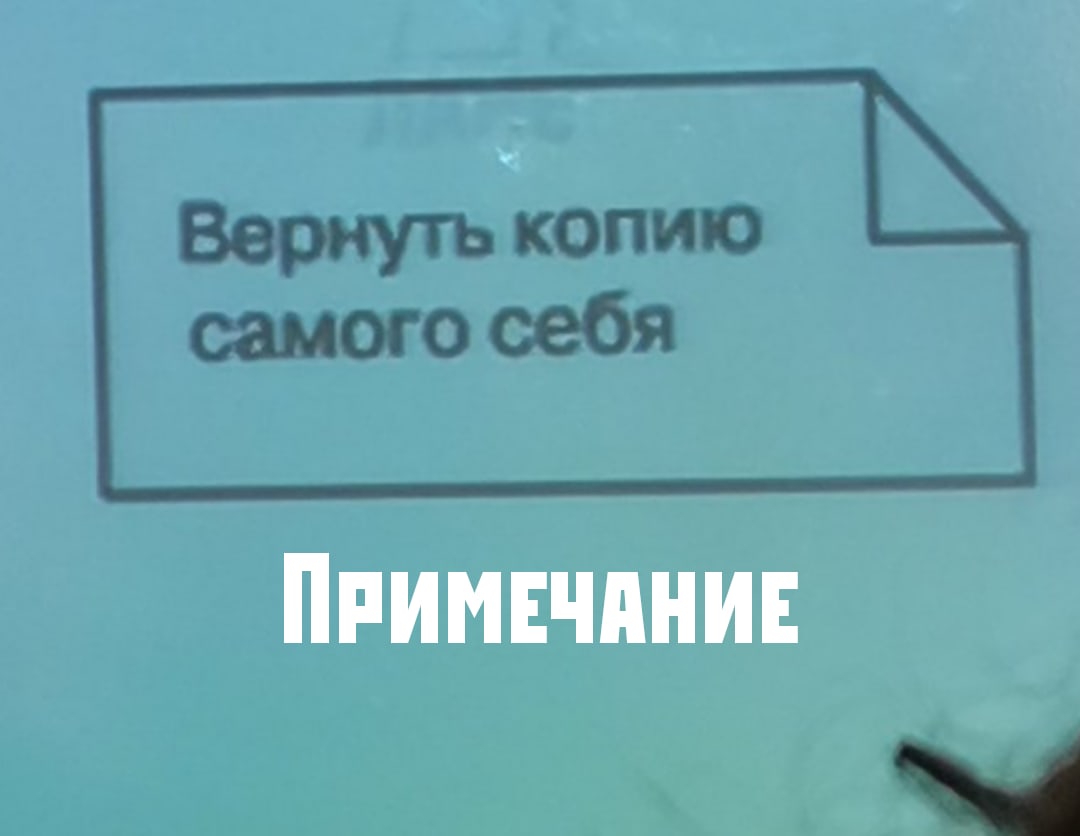


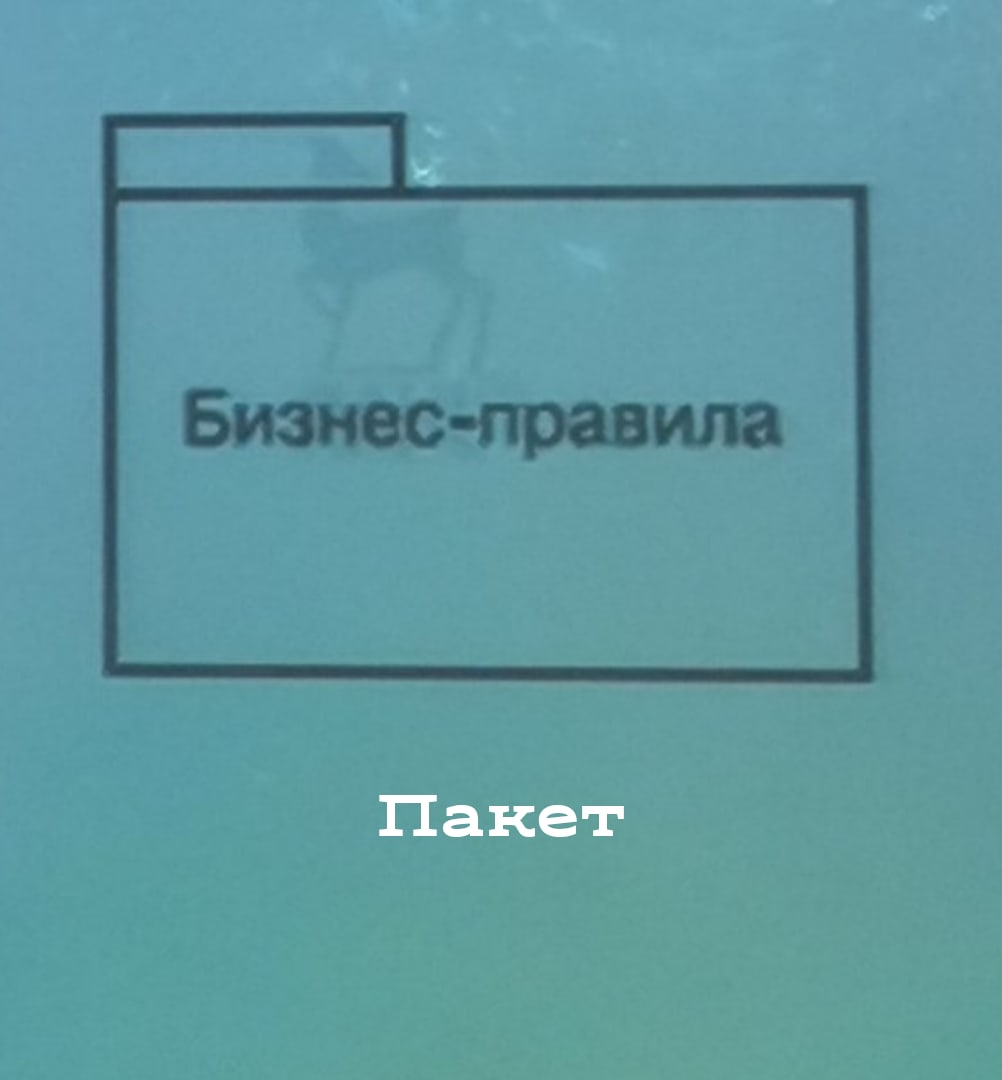






(Деятельность)





**Существует 4 типа связей в UML**:

1. Зависимость – связь между элементами модели, в которой изменение одного элемента, может примести к изменению семантики другого элемента.  
   Обозначается штрихами;
2. Ассоциация – Структурная связь между классами, которая описывает набор связей существующими между объектами (экземплярами классов).  
   Агрегация – особая разновидность ассоциации, определяющая структурную связь целого с его частями.  
   Обозначается прямой линией;
3. Обобщение – обобщение выражает специализацию или обобщение, в котором специализированный эл-т стоится по спецификации обобщённого элемента. Потомок разделяет структуру и поведение родителя.  
   Обозначается линией со стрелкой;
4. Реализация – связь между классификаторами, когда один из них специфицирует соглашение, которого второй обязан придерживаться. Отношение реализации указывает, что одна из сущностей является реализацией другой.  
   Обозначается штриховой линией со стрелкой.

**Диаграмма классов**

При проектировании системы информационная составляющая предметной области, представляется в виде логической модели уровня сущностей.

Логическая модель описывает сущности автоматизируемой предметной области и связи между ними.

В языке UML, для представления логической модели используется диаграмма классов.

Диаграмма классов – диаграмма, на которой показано множество классов, интерфейсов, коопераций и отношений между ними. Её изображают в виде множества вершин и дуг.

**Назначение** – диаграмма служит для представления статической структуры модели.

Диаграмма классов, может отражать взаимодействие между отдельными сущностями предметной области такими как объекты и подсистемы, а так же описывает их внутреннюю структуру и типы отношений.

**Особенность диаграммы**: на диаграмме классов не указывается инфа о временных аспектах функционирования системы. Зато она содержит классы, интерфейсы и кооперации.

Класс - для обозначения множества объектов, структурой, поведением и отношениями с объектами из других классов. Графически изображается прямоугольником (3 слота: имя – уникальное в пределах пакета (который описывается совокупностью диаграмм), атрибуты , операции).

Абстрактный класс – класс, который не может иметь экземпляров (объектов)

Атрибут– именованное свойство класса, общее для всех его объектов. Класс может иметь n+1 количество атрибутов, или не иметь их вообще. В каждый момент времени, любой атрибут объекта, принадлежащего данному классу, обладает вполне определёнными значениями.

Операции класса – реализация услуги, которую можно запросить у любого объекта класса для воздействия на поведение (абстракция того, что может делать объект). У всех объектов класса, общий набор операций. Операций может быть много, или не быть вообще. Как правило обращение к операции объекта, изменяет его состояние или его данные.

Советы по составлению UML:

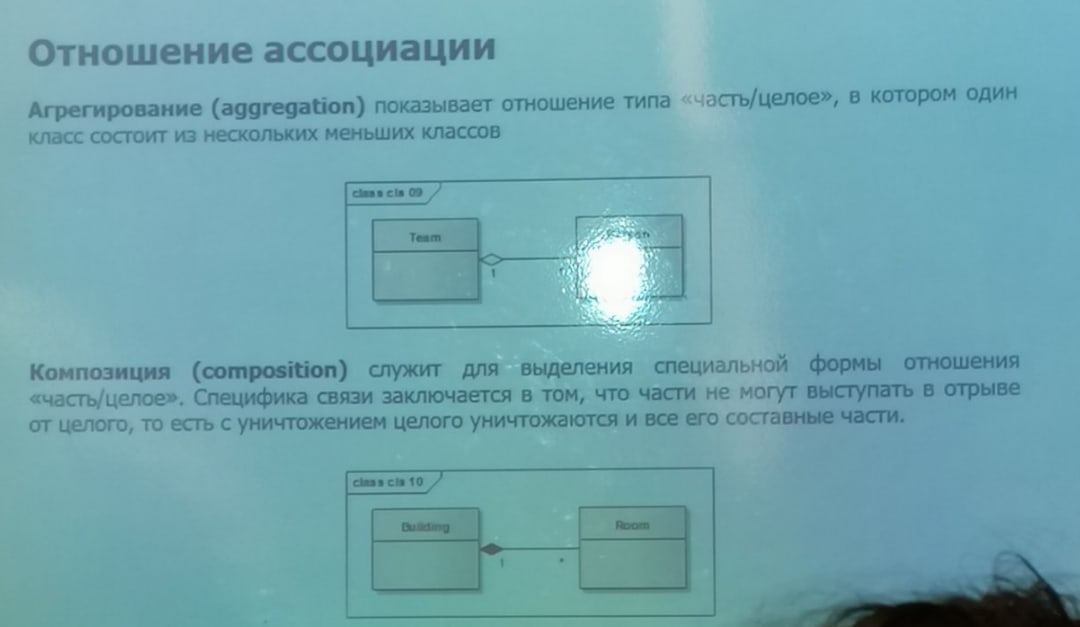
1. Показывать только те свойства класса, которые важны для понимания абстракции в данном контексте;
2. Разделяйте длинные списки атрибутов и операций на группы;
3. Взаимосвязанные классы изображать на одной диаграммы.

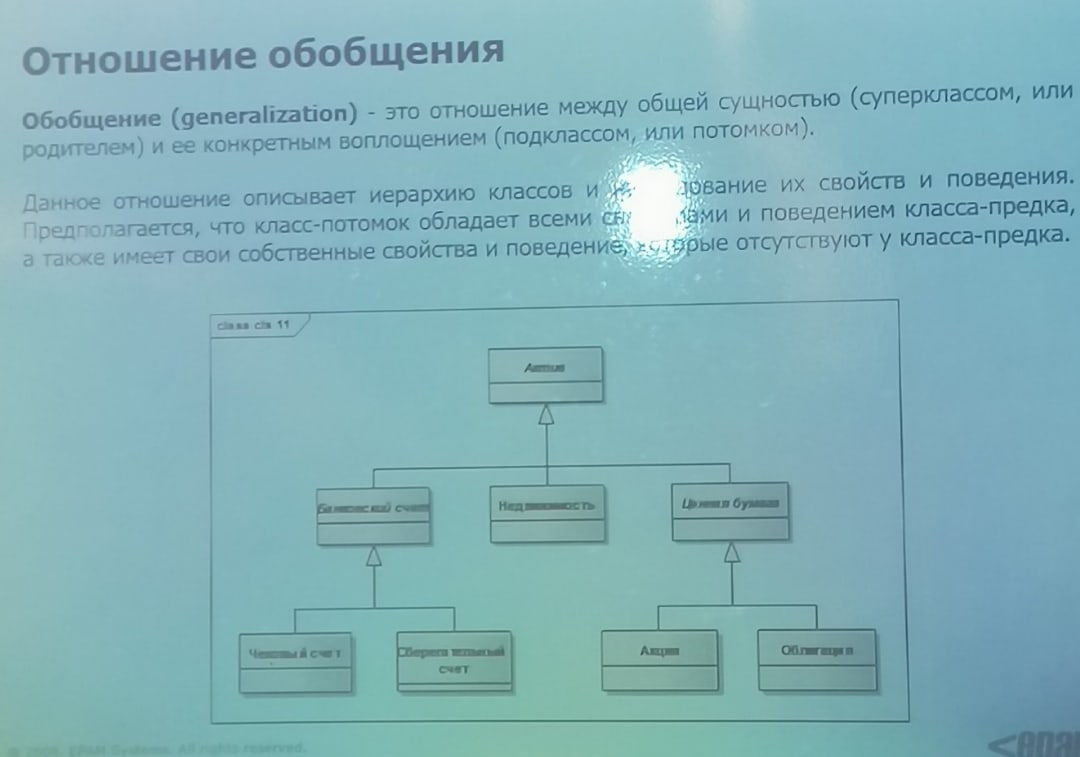
**Отношения на диаграмме классов:**

1. Зависимость – связь между элементами модели, в которой изменение одного элемента, может примести к изменению семантики другого элемента. Причём обратное не обязательно.  
   Обозначается штрихами со стрелкой от зависимого эл-та, до того, от которого он зависит;
2. Ассоциация – Структурная связь между классами, которая описывает набор связей существующими между объектами (экземплярами классов).

Бывает агрегацией и композицией (композиция более жёсткая чем агрегация)  
Ассоциация, связывающая 2 класса, называется бинарной.

Поля: имя (описывает природу отношения), роль, кратность.  
Обозначается прямой линией;

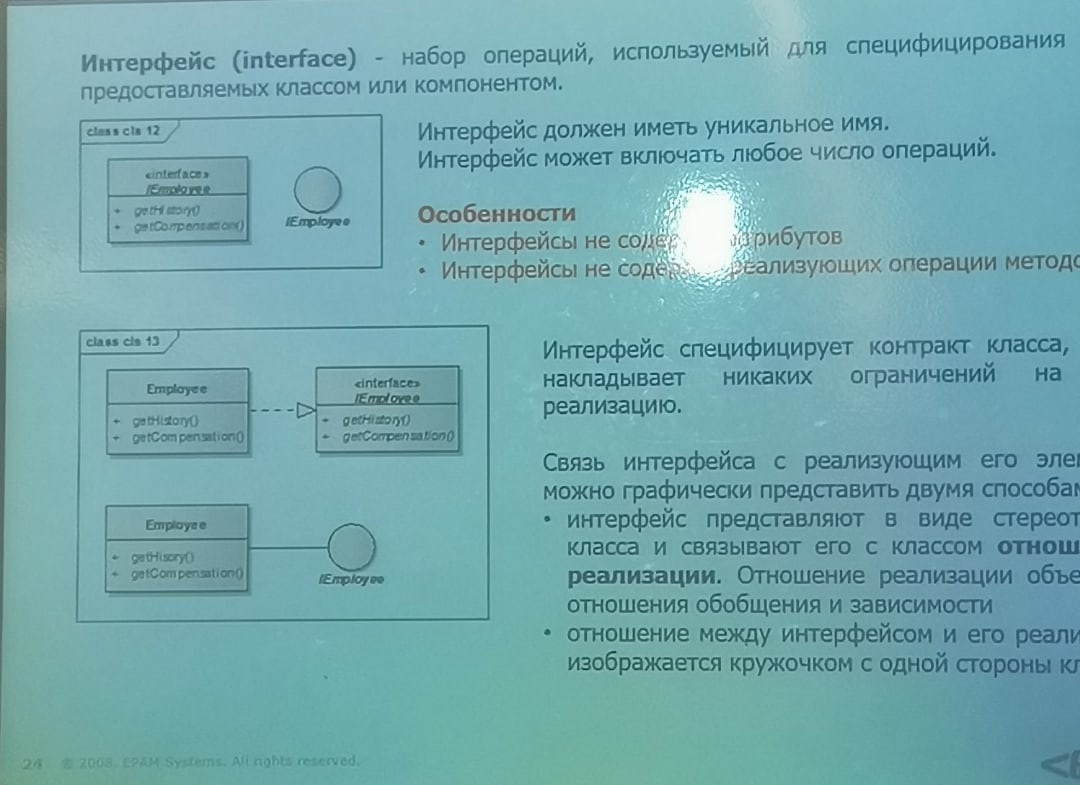


1. Обобщение – обобщение выражает специализацию или обобщение, в котором специализированный эл-т стоится по спецификации обобщённого элемента. Потомок разделяет структуру и поведение родителя.  
   Обозначается линией со стрелкой;  
   

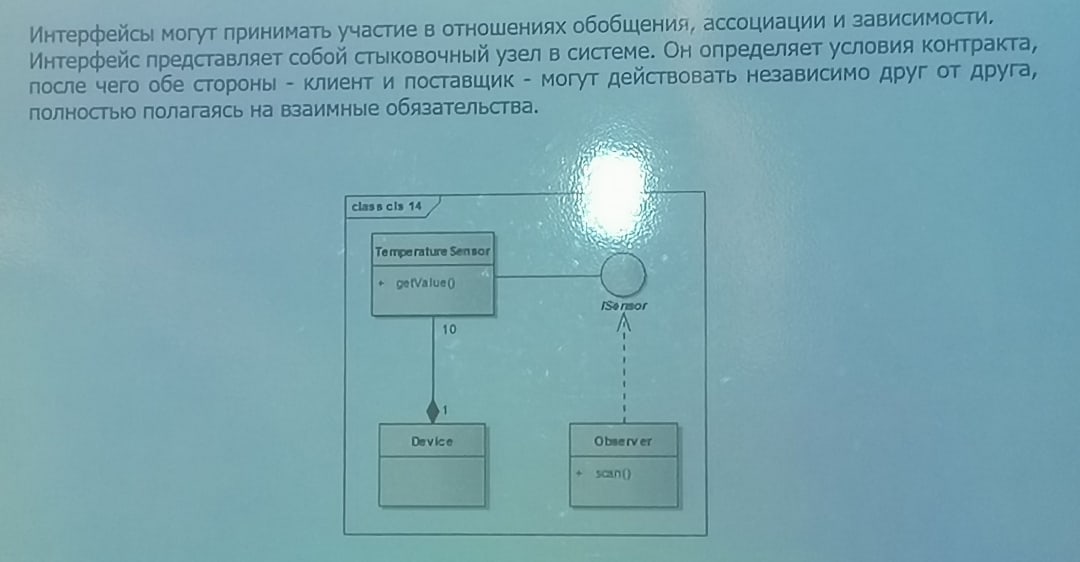
**Советы по моделированию UML:**

1. Используйте зависимость, только если моделируемое отношение не является структурным;
2. Используйте обобщение, только если имеет место отношение типа «является»;
3. Избегайте множественного наследования;
4. Иерархия наследования не должна быть ни слишком глубокой, ни слишком широкой;
5. Применяйте ассоциации прежде всего там, где между объектами существуют структурные отношения.

**Интерфейс:**



(на пикче выше показано 2 способа его реализации и способы связи)



Интерфейс в данном случае представляет собой стыковочный узел в системе.

**Диаграмма деятельности**

Диаграмма деятельности – моделирует динамические аспекты системы.

Диаграммы взаимодействий моделируют поведение сообщества совестно работающих объектов. Автоматы моделируют поведение отдельного объекта.

**Может показывать:**

1. Передачу потока управления от одного состояния объекта к другому;
2. Передачу потока управления от одной деятельности к другой.

Диаграмма деятельности предназначена для моделирования динамических аспектов поведения системы. Диаграмма деятельности описывает последовательность выполнения операций различными элементами, входящими в состав системы, во времени и показывает как поток управления переходит от одной деятельности к другой.

**Диаграмма деятельности может содержать**:

1. Состояния деятельности;
2. Переходы;
3. Объекты.

**Состояния деятельности:**

1. Действие – элементарная единица спецификации поведения, которая не может быть далее декомпозирована в форме деятельности;
2. Деятельность – продолжающийся во времени неатомарный шаг вычислений в автомате;
3. Деятельность может представлять собой совокупность более мелких деятельностей и атомарных действий.

Параметр деятельности – служит для представления входных и выходных параметров (обозначается прямоугольником с левого и правого края).

Объекты деятельности - служат для представления данных поступающих на вход деятельности или получаемых в результате её выполнения.

Дуга деятельности – абстрактный класс для направленных соединений между двумя узлами деятельности.

Существует 2 вида дуг деятельности:

1. Поток управления – дуга деятельности, по которой передаются только маркеры управления;
2. Поток объектов – дуга деятельности, по которой передаются только маркеры объектов или данных.

Узлы управления -Абстрактный узел деятельности, который предназначен для координации потоков в деятельности.

Различают следующие узлы:

1. **Начальный узел** – является узлом направления, к котором начинается поток при вызове действия;
2. **Узел финала** – является узлом, который прекращает или останавливает все потоки деятельности;
3. **Узал финала потока** – является финальным узлом, который завершает отдельный поток, не завершая содержащей его деятельности;
4. **Узел решения** – узел, который выбирает между двумя потоками (if. Де факто, разделяет поток на 2);  
   **Узел слияния** – узел, который выбирает один из потоков, и пихает его (обозначается так же как и узел решения )
5. **Узел разделения** – узел, который расщепляет поток на несколько параллельных  
   **Узел соединения** – синхронизирует несколько потоков (обозначается так же как и узел разделения).

