

ПРОЕКТИРОВАНИЕ ПРОГРАММНЫХ СИСТЕМ

ОСНОВНЫЕ ТЕМЫ КУРСА

- **Основы программной инженерии**
- **Язык UML**
- **Архитектура программных систем.**
Моделирование систем. Объектная модель
- **Определение требований к ПО**
- **Анализ и проектирование. Проектирование баз данных**
- **Шаблоны проектирования**
- **Управление проектами**
- **Методологии разработки ПО**

ИТОГОВАЯ АТТЕСТАЦИЯ

Лекции - 34 часа

Лабораторные занятия - 26 часов

Управляемая самостоятельная работа - 8 часов

Форма аттестации - зачет

Зачет при условии:

- 1. сданы и зачтены преподавателем все лабораторные работы,**
- 2. Сданы все тесты ≥ 6 баллов**

ГДЕ ПРИМЕНЯТЬ?

- Аналитик (Системный, бизнес-аналитик)
- Архитектор (системный, информационный)
- Проектировщик

(Analyst; Architects, Project Manager, Tech Leader)

ЛЕКЦИЯ 1

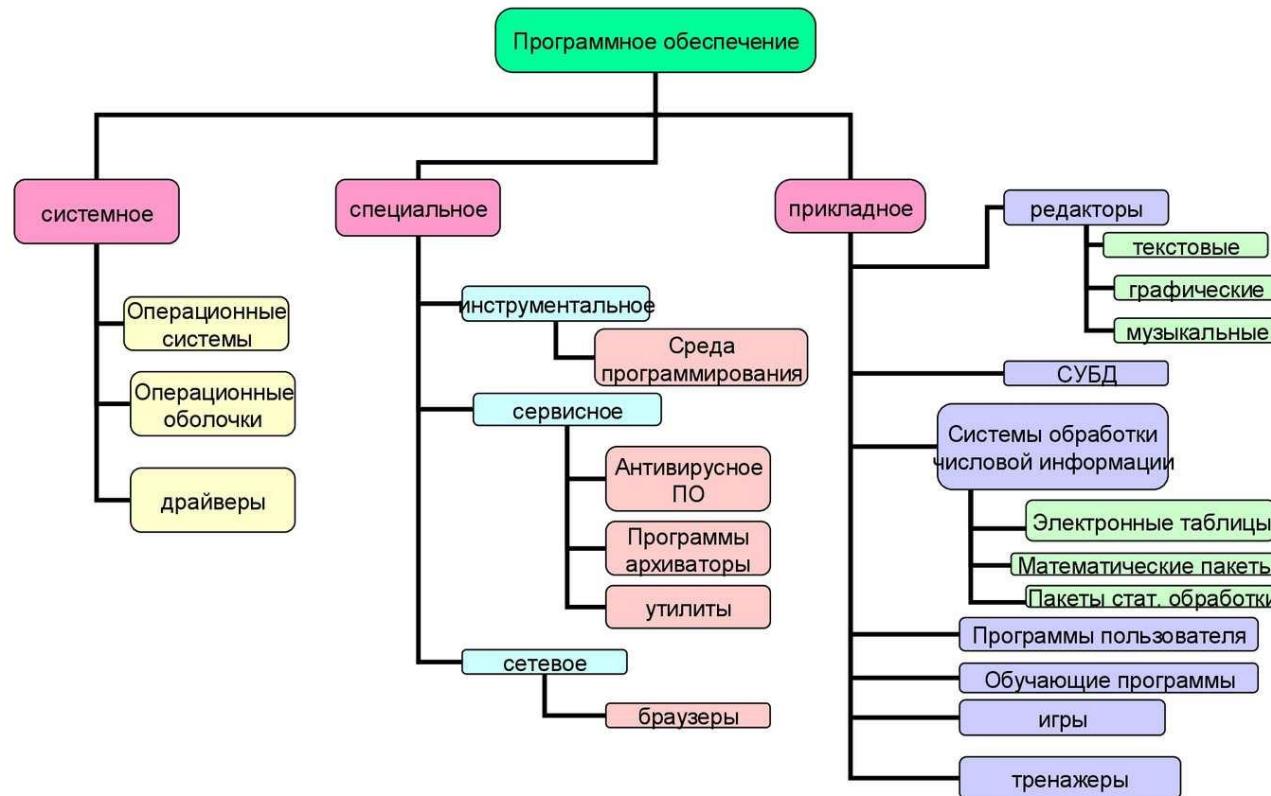
ОСНОВЫ
ПРОГРАММНОЙ
ИНЖЕНЕРИИ

ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Программное обеспечение (ПО) – это не только программы, но и вся сопутствующая документация, а также конфигурационные данные, необходимые для корректной работы программ.

КЛАССИФИКАЦИЯ ПО

КЛАССИФИКАЦИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ПК



СИСТЕМНЫЙ ПОДХОД

Основой проектирования программного обеспечения является системный подход.

Системный подход – это методология исследования объекта любой природы как системы.

СИСТЕМА

- это совокупность взаимосвязанных частей, работающих совместно для достижения некоторого результата. Определяющий признак системы – поведение системы в целом не сводимо к совокупности поведения частей системы.**
- это совокупность элементов и связей между ними. Ее можно представить в виде модели ниже. В рамках данной модели объекты моделируются сущностями, а связи моделируются отношениями. В рамках данного курса будем изучать методы моделирования и проектирования программных систем.**

ПРОГРАММНЫЕ СИСТЕМЫ –

это большие, сложные совокупности программных компонентов, которые предназначены для управления целым рядом, как принято говорить, ресурсных контуров – это людские ресурсы, финансовые ресурсы, различные другие виды производственных ресурсов, специфичные нефтегазовые ресурсы, документы и тому подобные ресурсы.

ПРОГРАММНЫЕ СИСТЕМЫ

Программные системы состоят из :

- совокупности программ,**
- файлов конфигурации, необходимых для установки этих программ,**
- и документации, которая описывает структуру системы, а также содержит инструкции для пользователей, объясняющие работу с системой,**
- и часто адрес web-узла, где пользователь может найти самую последнюю информацию о данном программном продукте.**

ТИПЫ ПРОГРАММНЫХ ПРОДУКТОВ

Специалисты по программному обеспечению разрабатывают программные продукты, т.е. такое ПО, которое можно продать потребителю. Программные продукты делятся на два типа.

- 1. Общие программные продукты.** Это автономные программные системы, которые созданы компанией по производству ПО и продаются на открытом рынке программных продуктов любому потребителю, способному их купить. Примерами этого типа программных продуктов могут служить системы управления базами данных, текстовые процессоры, графические пакеты и средства управления проектами.
- 2. Программные продукты, созданные на заказ.** Это программные системы, которые создаются по заказу определенного потребителя. Такое ПО разрабатывается специально для данного потребителя согласно заключенному контракту. Программные продукты этого типа включают системы управления для электронных устройств, системы поддержки определенных производственных или бизнес-процессов, системы управления воздушным транспортом и т.п.

ПРИКЛАДНАЯ СИСТЕМА

**При этом, если мы говорим о прикладной системе или
приложении, речь идет о том, чтобы решать задачи
прикладного плана.**

**Например, управление корпоративными ресурсами в аспекте
людских ресурсов:**

- прием на работу, перевод, увольнение,
- зачисление, обучение, повышение квалификации и т.д.

ПРОГРАММНАЯ ИНЖЕНЕРИЯ

Для решения такого рода задач по созданию программных систем, как систем прикладных, так и систем, имеющих другое назначение, как раз и возникла дисциплина, которая названа программной инженерией.

СТАНОВЛЕНИЕ И РАЗВИТИЕ ПРОГРАММНОЙ ИНЖЕНЕРИИ

Термин – software engineering (программная инженерия) – впервые был озвучен в октябре 1968 года на конференции подкомитета НАТО по науке и технике (г. Гармиш, Германия).

Присутствовало 50 профессиональных разработчиков ПО из 11 стран.

Рассматривались проблемы проектирования, разработки, распространения и поддержки программ.

СТАНОВЛЕНИЕ И РАЗВИТИЕ ПРОГРАММНОЙ ИНЖЕНЕРИИ

Становление и развитие этой области деятельности было вызвано рядом проблем, связанных с

- высокой стоимостью программного обеспечения, его создания**
- необходимостью управления и прогнозирования процессов разработки.**

СТАНОВЛЕНИЕ И РАЗВИТИЕ ПРОГРАММНОЙ ИНЖЕНЕРИИ

В конце 60-х – начале 70-х годов прошлого века произошло событие, которое вошло в историю как первый кризис программирования.

Событие состояло в том, что стоимость программного обеспечения стала приближаться к стоимости аппаратуры («железа»)

СТАНОВЛЕНИЕ И РАЗВИТИЕ ПРОГРАММНОЙ ИНЖЕНЕРИИ

ПРЕДПОСЫЛКИ

- Повторное использование кода (модульное программирование)
- Рост сложности программ (структурное программирование)
- Модификация программ (ООП)

НЕКОТОРЫЕ ИТОГИ

Программная инженерия (или технология промышленного программирования) как некоторое направление возникло и формировалось под давлением роста стоимости создаваемого программного обеспечения.

Главная цель этой области знаний – сокращение стоимости и сроков разработки программ. Программная инженерия прошла несколько этапов развития, в процессе которых были сформулированы фундаментальные принципы и методы разработки программных продуктов.

Основной принцип программной инженерии состоит в том, что программы создаются в результате выполнения нескольких взаимосвязанных этапов (анализ требований, проектирование, разработка, внедрение, сопровождение), составляющих жизненный цикл программного продукта.

Фундаментальными методами проектирования и разработки являются модульное, структурное и объектно-ориентированное проектирование и программирование.

ПРОДОЛЖЕНИЕ КРИЗИСА ПРОГРАММИРОВАНИЯ

Рубеж 80–90-х годов отмечается как начало информационно-технологической революции, вызванной взрывным ростом использования информационных средств: персональный компьютер, локальные и глобальные вычислительные сети, мобильная связь, электронная почту, Internet и т.д.

Цена успеха – кризис программирования принимает хронические формы.

	2004	2006	2008	2010	2012	2013
УСПЕШНЫЕ	29%	35%	32%	37%	39%	36%
ПРОВАЛЬНЫЕ	18%	19%	24%	21%	18%	16%
СПОРНЫЕ	53%	46%	44%	42%	43%	48%



По некоторым исследованиям каждый шестой ИТ-проект оказывается провальным.

Это подразумевает разрастание изначального бюджета (чуть ли не до 200% от запланированного) и затягивание сроков (почти на 70%).



ФРЕДЕРИК БРУКС

- Характерные вопросы и задачи инженерии программирования, изложенные Фредериком Бруксом (л-т премии Тьюринга) еще в 1999 г, актуальны и по сегодняшний день.
- Как проектировать и строить программы, образующие системы?
- Как проектировать и строить программы и системы, являющиеся надежным, отлаженным, документированным и сопровождаемым продуктом?
- Как осуществлять интеллектуальный контроль в условиях большой сложности?



Фредерик Филипс Брукс мл. (Frederick Phillips Brooks, Jr.), род. 19 апреля 1931 года — американский менеджер, инженер и ученый, наиболее известен как руководитель разработки операционной системы OS/360. В 1975 году, обобщая опыт этой работы, написал книгу "Мифический человекомесяц". Брукс насмешливо называл свою книгу "библией программной инженерии": "все ее читали, но никто ей не следует!".

ОПРЕДЕЛЕНИЕ ПРОГРАММНОЙ ИНЖЕНЕРИИ

(из книги Липаева «Программная инженерия»)

Программная инженерия – это комплекс задач, методов, средств и технологий создания (проектирования и реализации) сложных, расширяемых, тиражируемых, высококачественных программных систем, возможно включающих базы данных.

ФУНДАМЕНТАЛЬНАЯ ИДЕЯ ПРОГРАММНОЙ ИНЖЕНЕРИИ:

**Проектирование ПО является формальным
процессом, который можно изучать и
совершенствовать.**

**Освоение и правильное применение методов и
средств программной инженерии позволяет
повысить качество, обеспечить управляемость
процесса проектирования.**

ПРОБЛЕМЫ РАЗРАБОТКИ ПО

- **Достижение высокого качества при**
 - Нехватке ресурсов
 - Возрастающей сложности программных систем
 - Часто изменяющихся требованиях
 - Большом количестве участников
- **Устаревание ПО**
- **Высокие темпы разработки**
- **Информатизация все новых отраслей хозяйства**
- **Высокая конкуренция**

ОГРАНИЧЕНИЯ ДЛЯ РЕАЛИЗАЦИИ ПРОГРАММНЫХ СИСТЕМ:

- **размер реализуемых программных систем комплексов и компонент, из которых строятся такого рода системы;**
- **сложность, т.е. количество модулей и связей между ними;**
- **программную среду, в рамках которой происходит реализация;**
- **и определенные ограничения, связанные с человеческим фактором, с людьми, с людскими ресурсами.**

**По сути дела, при реализации
программных систем мы
сталкиваемся с задачей
многофакторной оптимизации.**

СИСТЕМНЫЙ ПОДХОД КАК ОСНОВА ПРОЕКТИРОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

СИСТЕМНЫЙ ПОДХОД –

**общенаучный обобщенный эвроритм,
предусматривающий всестороннее исследование
сложного объекта с использованием
компонентного, структурного, функционального,
параметрического и генетического видов анализа.**

(Эвроритм – порядок действия человека при выполнении какой-то деятельности. В отличие от алгоритма может изменяться в процессе исполнения благодаря разумности исполнителя.)

Компонентный анализ – рассмотрение объекта, включающего в себя составные элементы и входящего, в свою очередь, в систему более высокого ранга.

Структурный анализ – выявление элементов объекта и связей между ними.

Функциональный анализ – рассмотрение объекта как комплекса выполняемых им полезных и вредных функций.

Параметрический анализ – установление качественных пределов развития объекта – физических, экономических, экологических и др.

Применительно к программам параметрами могут быть: время выполнения какого-нибудь алгоритма, размер занимаемой памяти и т.д.

При этом выявляются ключевые технические противоречия, мешающие дальнейшему развитию объекта, и ставится задача их устранения за счет новых технических решений.

Генетический анализ – исследование объекта на его соответствие законам развития программных систем. В процессе анализа изучается история развития (генезис) исследуемого объекта: конструкции аналогов и возможных частей, технологии изготовления, объемы тиражирования, языки программирования и т.д.

ОСОБЕННОСТИ И КЛАССИФИКАЦИЯ СОВРЕМЕННЫХ ПРОГРАММНЫХ ПРОЕКТОВ

ОСОБЕННОСТИ СОВРЕМЕННЫХ ПРОГРАММНЫХ ПРОЕКТОВ

- сложность – неотъемлемая характеристика создаваемого ПО;
- отсутствие полных аналогов и высокая доля вновь разрабатываемого ПО;
- наличие унаследованного ПО и необходимость его интеграции с разрабатываемым ПО;
- территориально распределенная и неоднородная среда функционирования;
- большое количество участников проектирования, разобщенность и разнородность отдельных групп разработчиков по уровню квалификации и опыту.

СПЕЦИФИЧЕСКИЕ ОСОБЕННОСТИ РАЗРАБОТКИ ПО

- неформальный характер требований к ПО и формализованный основной объект разработки – программы;
- творческий характер разработки;
- дуализм ПО, которое, с одной стороны, является статическим объектом – совокупностью текстов, с другой стороны, – динамическим, поскольку при эксплуатации порождаются процессы обработки данных;
- при своем использовании (эксплуатации) ПО не расходуется и не изнашивается;
- «неощущимость», «воздушность» ПО, что подталкивает к безответственному переделыванию, поскольку легко стереть и переписать, чего не сделаешь при проектировании зданий и аппаратуры.

КЛАССИФИКАЦИЯ ПРОЕКТОВ

Институтом качества программного обеспечения SQI (Software Quality Institute, США) специально для выбора модели ЖЦ разработана схема классификации проектов по разработке ПС. Основу данной классификации составляют четыре категории критериев:

- **Характеристики требований к проекту.**
- **Характеристики команды разработчиков.**
- **Характеристики пользователей (заказчиков).**
- **Характеристики типов проектов и рисков.**

КЛАССИФИКАЦИЯ ПРОЕКТОВ

Набор критериев данной категории, предложенный институтом SQI, является недостаточно полным и может быть дополнен критериями из других общепринятых классификаций:

- масштаб продукта проекта (размер или сложность продукта проекта);
- стабильность продукта проекта (эволюция продукта проекта);
- критичность продукта проекта (степень влияния на общество повреждений продукта проекта).

КЛАССИФИКАЦИЯ ПРОЕКТОВ ПО РАЗМЕРУ

- **небольшие проекты – проектная команда (ПК) менее 10 чел., срок от 3 до 6 мес.;**
- **средние проекты – ПК от 20 до 30 чел., протяженность проекта 1-2 года;**
- **крупномасштабные проекты – ПК от 100 до 300 чел., длительность проекта 3-5 лет;**
- **гигантские проекты – ПК от 1000 до 2000 чел. и более (включая консультантов и соисполнителей), протяженность проекта от 7 до 10 лет.**

ЖИЗНЕННЫЙ ЦИКЛ ПО

В ЭТОМ РАЗДЕЛЕ ЛЕКЦИИ ИСПОЛЬЗОВАНЫ
МАТЕРИАЛЫ ЭЛЕКТРОННОГО КУРСА
«РАЗРАБОТКА КОРПОРАТИВНЫХ СИСТЕМ. ЧАСТЬ
1. МОДЕЛИ ЖИЗНЕННОГО ЦИКЛА» НИЯУ МИФИ.

ЖИЗНЕННЫЙ ЦИКЛ ПО –

это достаточно протяженный во времени процесс, который начинается с концепции, может быть с неформальной идеи, только с общего представления самых предварительных контуров программного продукта и заканчивается с полным выводом из эксплуатации этого программного продукта или программной системы

(книга "Быстрая разработка" Стива Мак Коннелла).

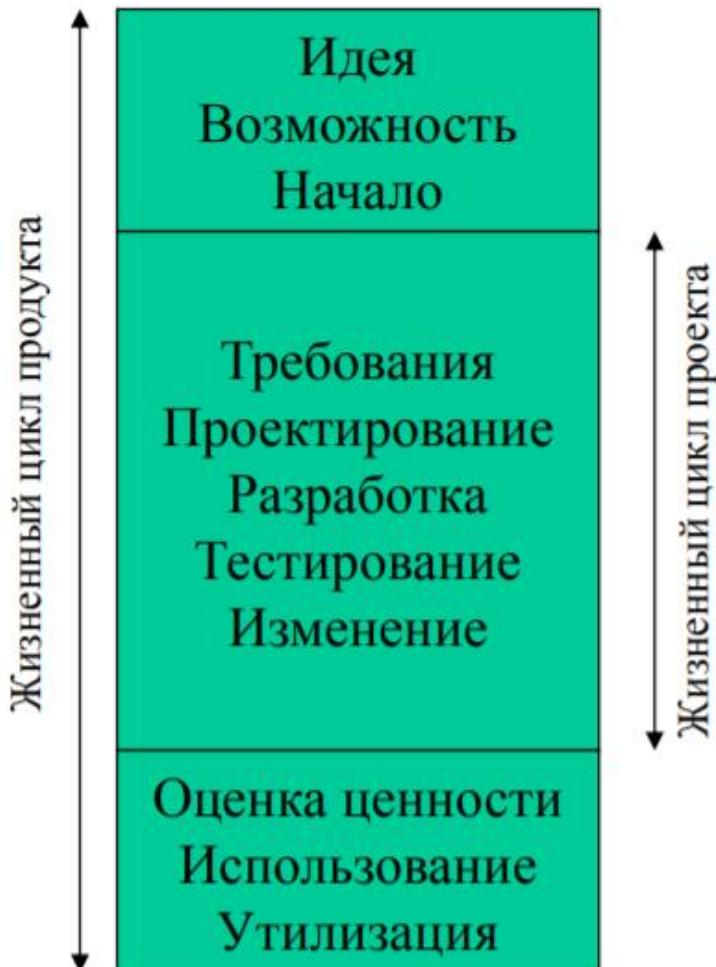
ОСОБЕННОСТИ ЖЦ РАЗРАБОТКИ ПО

- Программная система разрабатывается постепенно и развивается, начиная от зарождения идеи ПО до реализации и сдачи пользователю, и далее.
- Каждый этап завершается разработкой части системы или связанной с ней документации (план тестирования, руководство пользователя и т.д.).
- Теоретически, для каждого этапа четко определены начальные и конечные точки, а также известно, что он должен передать следующему этапу.
- На практике все сложнее.

ЦЕЛИ ИЗУЧЕНИЯ ЖЦ

- **Организация и управление разработкой ПО.**
- **Основа для анализа разработки ПО.**
- **Основа для планирования разработки ПО.**
- **Корректная постановка процессов разработки ПО.**
- **Анализ ЖЦ обязателен для сложных проектов.**

СВЯЗЬ ЖЦ ПРОДУКТА И ПРОЕКТА



Проект – это нечто измеримое, его жизненный цикл выглядит несколько иначе, более конкретно и включает разработку требований, проектных спецификаций, реализацию, проектирование, интеграцию, тестирование и управление изменениями с учетом тех возможных коррективов в требованиях, которые могут потребоваться заказчику.

МОДЕЛЬ ЖИЗНЕННОГО ЦИКЛА

- **Модель ЖЦ ПО** – это структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач на протяжении всего ЖЦ.
- ***В любой модели ЖЦ рассматривается как совокупность стадий.***
- **Стадия ЖЦ** – это часть ЖЦ ограниченная временными рамками, по завершении которой достигается определенный важный результат в соответствии с требованиями для данной стадии ЖЦ.
- **Между двумя стадиями, идущими друг за другом, находится контрольная точка (веха).** Так называют момент времени, разделяющий стадии жизненного цикла (или итерации, если они предусмотрены в модели ЖЦ), по наступлении которого должны достигаться результаты важные для всего проекта и должны приниматься решения о дальнейшем управлении проектом.

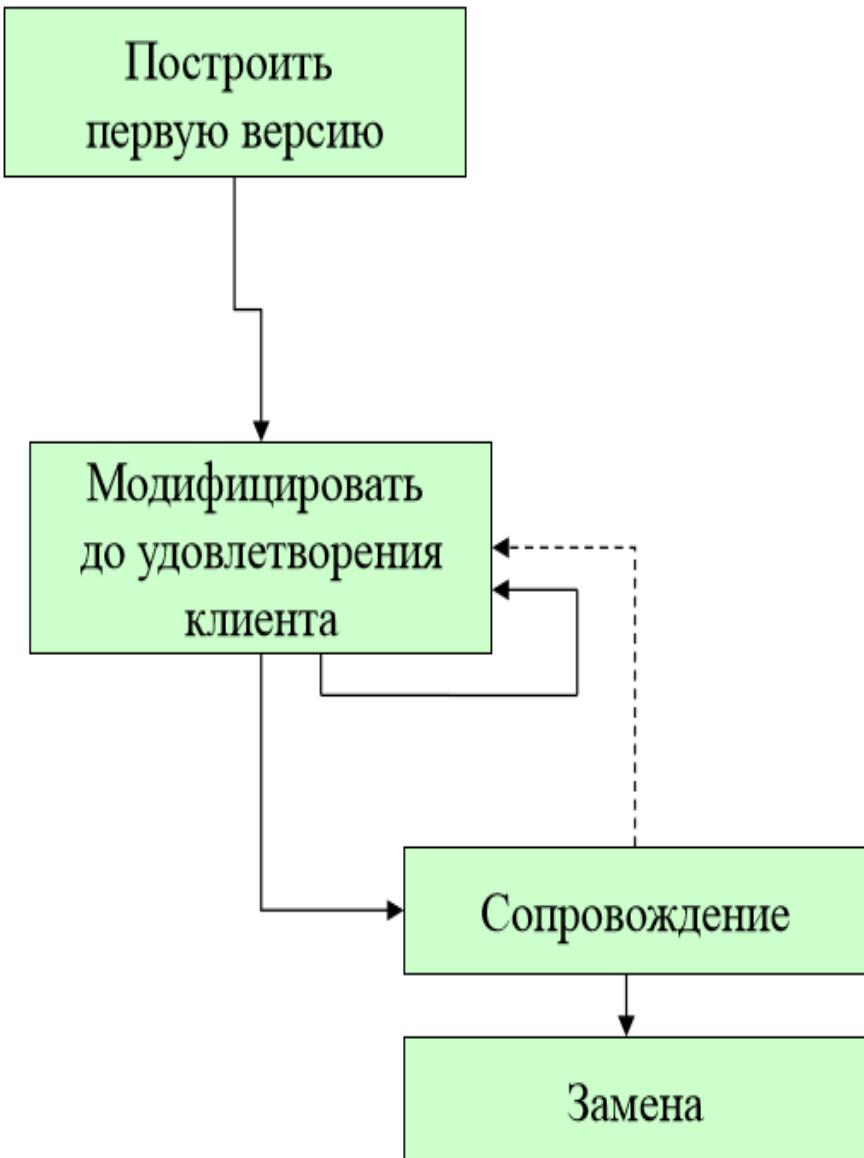
МОДЕЛИ ЖЦ ПО ОПРЕДЕЛЯЮТ:

- **Характер и масштаб проекта (объем, сроки, риски, ...)**
- **Экономику проекта**
- **Степень сопровождаемости**
- **Перспективы развития (прогноз запросов клиента)**
- **Архитектуру проекта (стабильная эволюция, революционные усовершенствования)**
- **Скорость поиска и устранения ошибок**
- **Управление рисками проекта**
- **Степень полноты реализации (прототип, промежуточное решение, готовый продукт)**

ОГРАНИЧЕННЫЕ МОДЕЛИ ЖИЗНЕННОГО ЦИКЛА

МОДЕЛИ, КОТОРЫЕ НЕСКОЛЬКО ОГРАНИЧЕНЫ В СИЛУ РАЗЛИЧНЫХ ПРИЧИН: ПРОСТОТА, НЕПРИГОДНОСТЬ ДЛЯ БОЛЬШИХ И СЛОЖНЫХ ПРОЕКТОВ, НЕСАМОСТОЯТЕЛЬНОСТЬ.

МОДЕЛЬ CODE-AND-FIX (BUILD-AND- FIX)



По сути дела, речь идет о модели проб и ошибок.

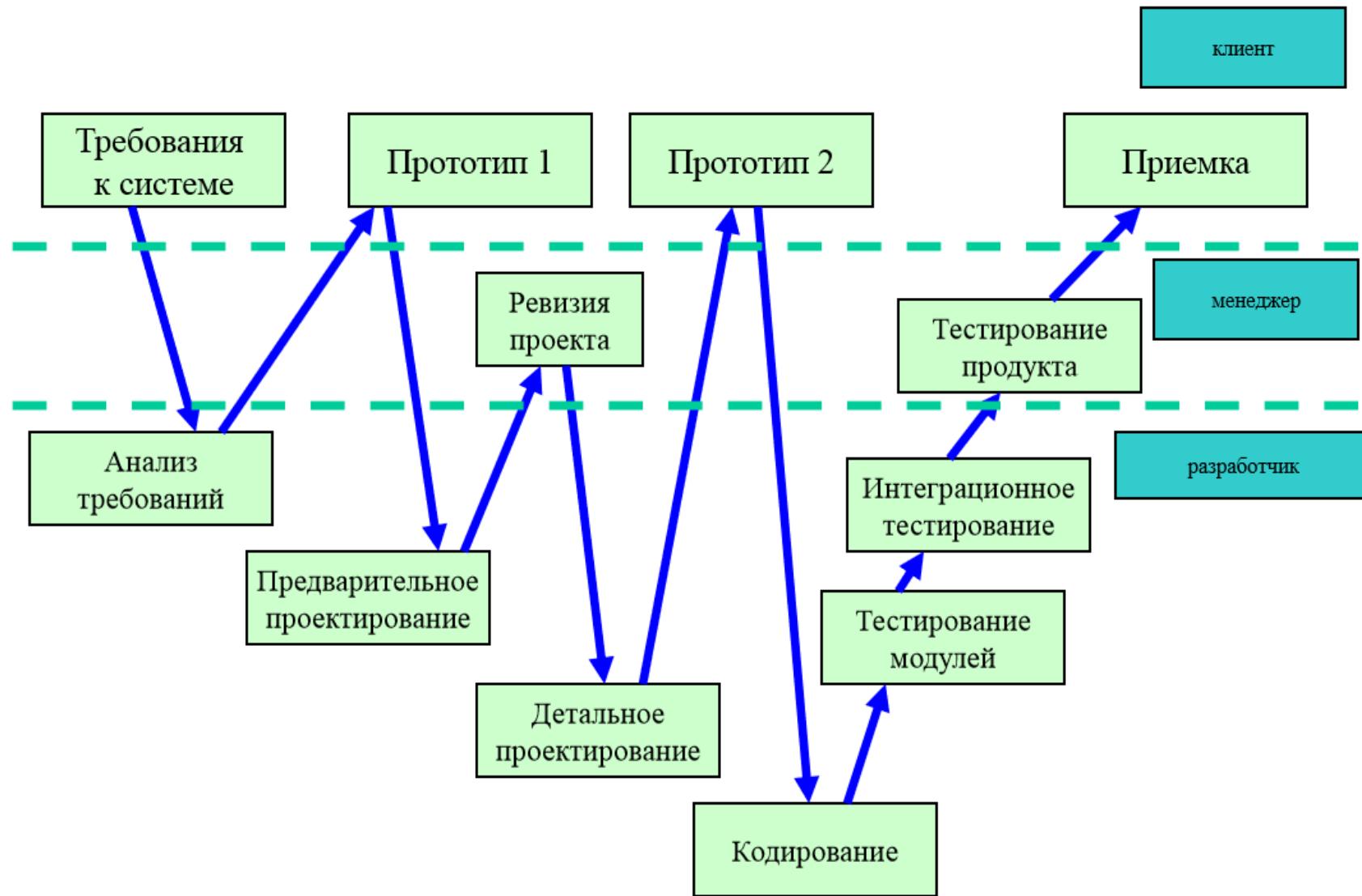
Здесь приходится строить продукт заново каждый раз до тех пор, пока клиент не будет доволен, не будет удовлетворен.

ВОДОПАДНАЯ/КАСКАДНАЯ МОДЕЛЬ



Существенной особенностью этой модели является однопроходная разработка без циклических повторений. Другим существенным аспектом на каждой стадии является непременное условие верификации.

МОДЕЛЬ БЫСТРОГО ПРОТОТИПИРОВАНИЯ/«ЗУБЬЯ АКУЛЫ»



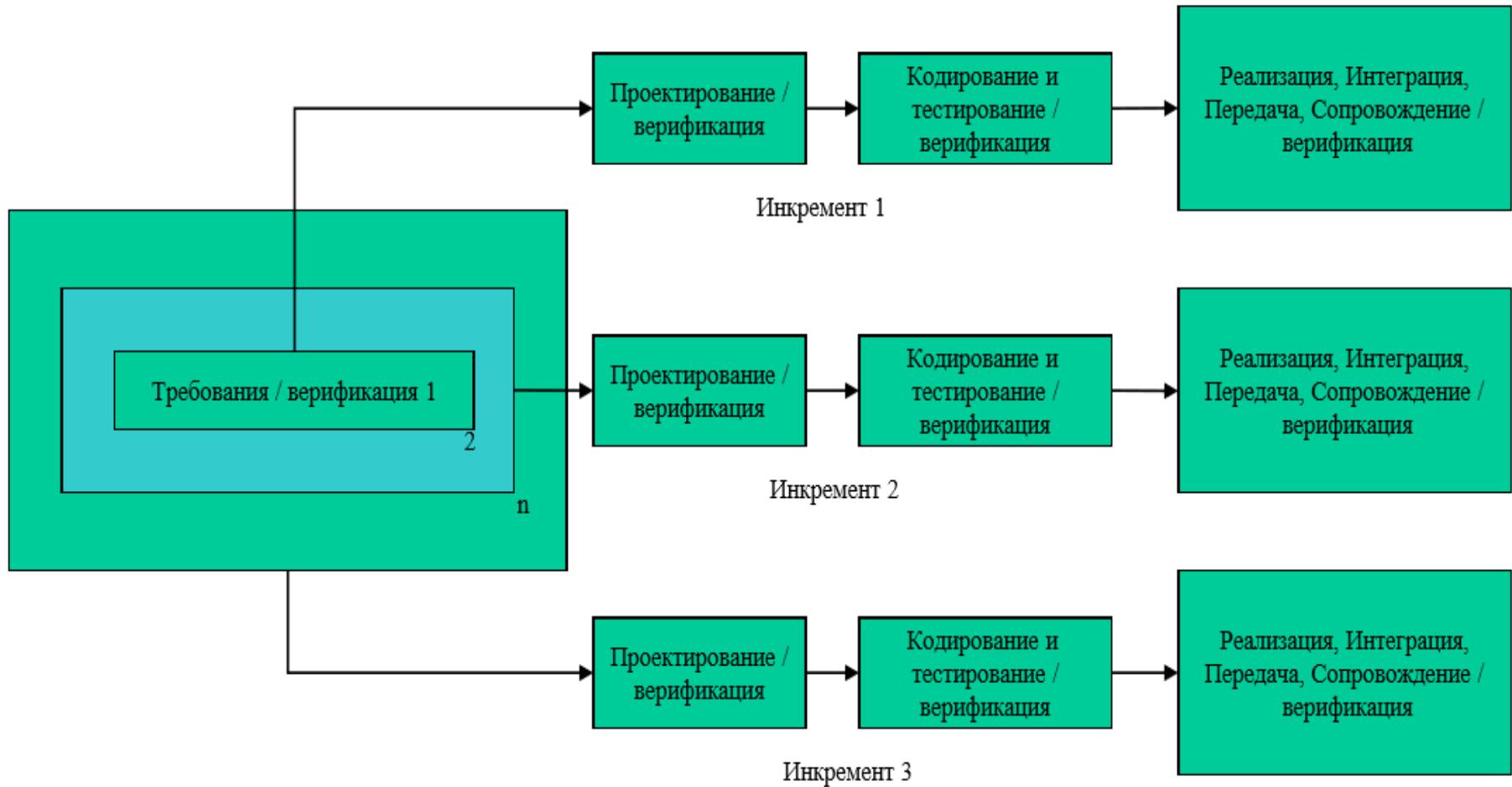
ОГРАНИЧЕННЫЕ МОДЕЛИ ЖЦ ПО: СРАВНИТЕЛЬНЫЙ АНАЛИЗ

Модель ЖЦ	Преимущества	Недостатки
Build-and-Fix (Code – and –Fix)	Хороша для небольших, не требующих сопровождения проектов	Абсолютно непригодна для нетривиальных проектов
Водопадная	Четкая дисциплина проекта, документно-управляемая	ПО может не соответствовать требованиям клиента
Быстрого прототипирования	Обеспечивает соответствие ПО требованиям клиента	Вызывает соблазн повторного использования кода, который следует заново реализовать

ЦИКЛИЧЕСКИЕ МОДЕЛИ ЖИЗНЕННОГО ЦИКЛА

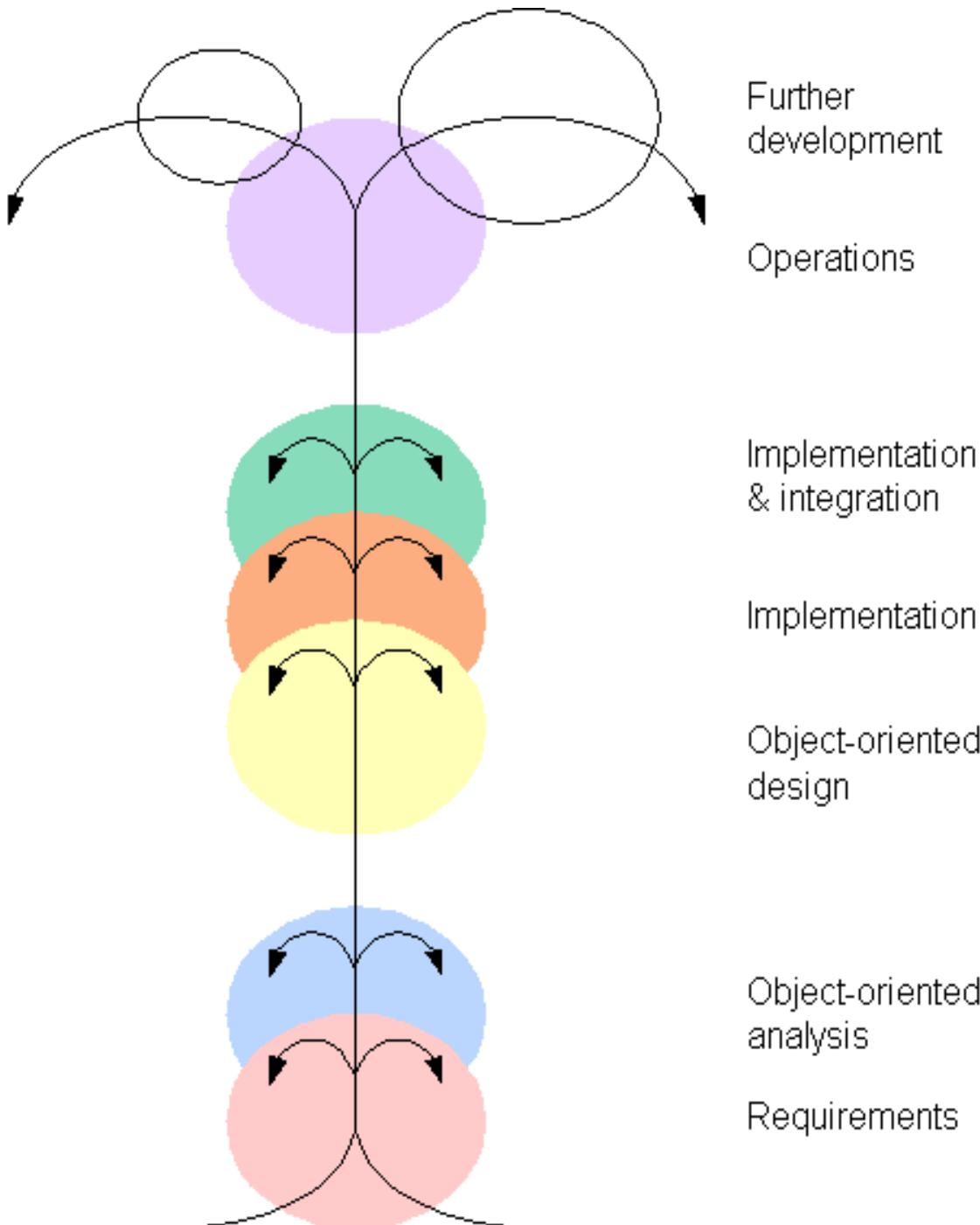
МОДЕЛИ, В КОТОРЫХ СТАДИИ ЖИЗНЕННОГО ЦИКЛА
СМЕНЯЮТ ДРУГ ДРУГА ИТЕРАТИВНО ИЛИ ЦИКЛИЧЕСКИ

ИНКРЕМЕНТНАЯ ИЛИ ИНКРЕМЕНТАЛЬНАЯ МОДЕЛЬ



*Каждый релиз включает детальное проектирование ,
реализацию, интеграцию, тестирование и передачу*

ОБ'ЄКТНО- ОРИЕНТИРОВАНА Я МОДЕЛЬ



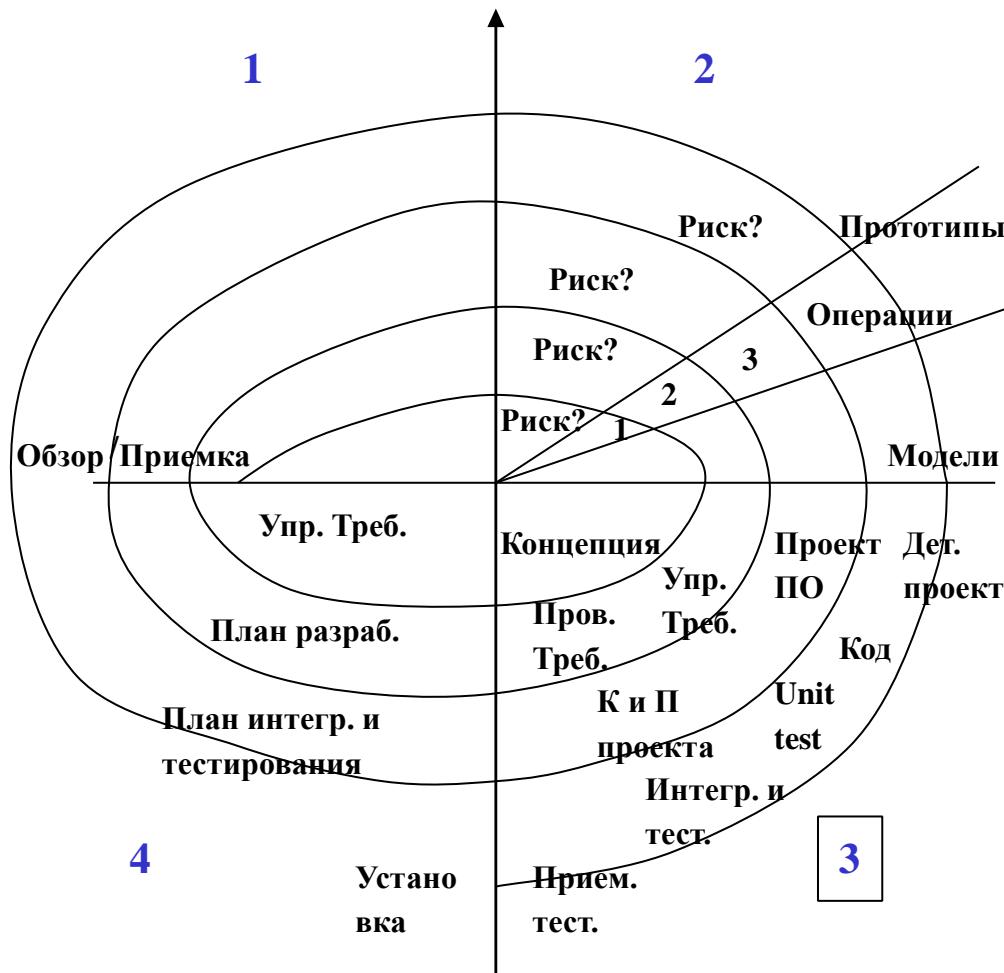
ЦИКЛИЧЕСКИЕ МОДЕЛИ ЖЦ ПО: СРАВНИТЕЛЬНЫЙ АНАЛИЗ

Модель ЖЦ	Преимущества	Недостатки
Инкрементная	Максимально ранний возврат инвестиций; способствует сопровождаемости	Требует открытой архитектуры; может выродиться в Build-and-fix
ОО-модель	Обеспечивает итерацию внутри фаз и параллелизм между фазами	Может выродиться в Build-and-fix

СПЕЦИАЛИЗИРОВАННЫЕ МОДЕЛИ ЖИЗНЕННОГО ЦИКЛА

СПИРАЛЬНАЯ МОДЕЛЬ ВКЛЮЧАЕТ В СЕБЯ ОЦЕНКУ РИСКОВ, ЧТО САМО ПО СЕБЕ ЯВЛЯЕТСЯ ДОСТАТОЧНО СЛОЖНЫМ И ЗАТРАТНЫМ МЕРОПРИЯТИЕМ, А МОДЕЛЬ СИНХРОНИЗАЦИИ И СТАБИЛИЗАЦИИ (ВКЛЮЧАЕТ ДВА ДОСТАТОЧНО СЛОЖНЫХ ПРОЦЕССА: СИНХРОНИЗАЦИЮ И СТАБИЛИЗАЦИЮ)

СПИРАЛЬНАЯ МОДЕЛЬ ЖЦ (ВОЕНМ, 1987)



Каждый виток состоит из 4 фаз:

1. Определить цели: определить продукт, определить деловые цели, понять ограничения, предложить альтернативы
2. Оценить альтернативы: анализ риска, прототипирование
3. Разработать продукт: детальный проект, код, unit test, интеграция
4. Спланировать следующий цикл: оценка клиентом, планирование проектирования, поставка клиенту, внедрение

МОДЕЛЬ СИНХРОНИЗАЦИИ И СТАБИЛИЗАЦИИ (MICROSOFT)

Особенности:

3-4 инкрементных версии ПО, включающих:

- Синхронизацию (проверка, сборка, тестирование)
- Стабилизацию(устранение ошибок, найденных тестами)
- «Заморозку» - работающий «срез» ПО

Преимущества:

- «частое и раннее» тестирование (и выявление ошибок)
- Постоянная интероперабельность (модули тестируются в сборе => всегда есть работающая версия ПО => связи между модулями легко тестировать)
- Ранняя коррекция проекта (полная «сборка» ПО первых версий позволяет выявить недочеты проекта до полно-масштабной реализации и снизить стоимость редизайна)

СПЕЦИАЛИЗИРОВАННЫЕ МОДЕЛИ ЖЦ ПО : СРАВНИТЕЛЬНЫЙ АНАЛИЗ

Модель ЖЦ	Преимущества	Недостатки
Синхронизации и стабилизации	Удовлетворяет будущим потребностям клиента; обеспечивает интеграцию компонент	Не получила широкого применения вне Microsoft
Сpirальная	Объединяет характеристики всех перечисленных выше моделей	Пригодна лишь для крупных внутренних проектов; разработчики должны владеть управлением рисками

МОДЕЛЬ ФОРМАЛЬНОЙ РАЗРАБОТКИ СИСТЕМЫ

**Основана на разработке формальной
математической спецификации программной
системы и преобразования этой спецификации
посредством специальных математических
методов в исполняемые программы.**

**Проверка соответствия спецификации и
системных компонентов также выполняется
математическими методами**

МЕТОДОЛОГИЯ РАЗРАБОТКИ ПО

Методология – это система принципов, а также совокупность идей, понятий, методов, способов и средств, определяющих стиль разработки программного обеспечения.

Методологии могут включать различные модели в своей основе.

В полном научном смысле понятие методологии в области разработки ПО не совсем правильно. Это больше набор практических приемов, нет математических моделей, а часто и экономического обоснования.

**ОБЩИЕ
СВЕДЕНИЯ О
СЕМЕЙСТВЕ
СТАНДАРТОВ
12207**

В основе практически всех современных промышленных технологий создания ПС лежит международный стандарт ISO/IEC 12207 «Системная и программная инженерия. Процессы жизненного цикла программных средств.»

В состав семейства входят:

ISO/IEC 12207:1995 «Information technology–Software life cycle processes» с дополнениями и изменениями ISO/IEC 12207:1995/AMD 1:2002 и ISO/IEC 12207:2002/AMD 2:2004 (принят в новой редакции в 2008 году) ISO/IEC 12207:2008 «Systems and software engineering–Software life cycle processes»

ISO/IEC TR 15271:1998 Information technology – Guide for ISO/IEC 12207 (Software Life Cycle Processes)

ISO/IEC TR 16326:1999 Software engineering – Guide for the application of ISO/IEC 12207 to project management

СТАНДАРТЫ И ГОСТЫ В ОБЛАСТИ ИНФОРМАТИЗАЦИИ РБ

<http://nmo.basnet.by/documents/normative/standarts.php>

**Структура стандарта имеет гибкий,
модульный скелет, что позволяет быть
адаптируемым к потребностям детализации
для любого пользователя**

ОРГАНИЗАЦИЯ РАЗРАБОТКИ ТРЕБОВАНИЙ К СЛОЖНЫМ ПРОГРАММНЫМ СРЕДСТВАМ

ГРУППЫ ПРОЦЕССОВ

- **основные**
(заказ, поставка, разработка, эксплуатация, сопровождение);
- **вспомогательные**
(документирование, управление конфигурацией, обеспечение качества, верификация, аттестация, совместный анализ, аудит, разрешение проблем);
- **организационные**
(управление, создание инфраструктуры, усовершенствование, обучение).

Проекты программных средств различаются по

- уровню сложности,
- масштабу
- и необходимому качеству.

Они имеют различное

- назначение,
- содержание
- и относятся к разным областям применения.

Поэтому существует потребность в четко организованном процессе, методах формализации и управления требованиями к конкретным программным продуктам.

Чаще всего проблемами, с которыми встретились не достигшие своих целей проекты программных продуктов, являются:

- недостаток информации от пользователя или заказчика о функциях проекта,*
- неполные, некорректные требования,*
- а также многочисленные изменения требований и*



Формализация и управление требованиями – это систематический метод выявления, организации и документирования требований к ПС и/или ПО,

а также процесс, в ходе которого вырабатывается и обеспечивается соглашение между заказчиком и выполняющими проект специалистами, в условиях меняющихся требований к системе.

- Команда разработчиков должна применить методы и процессы для того, чтобы *понять решаемую проблему заказчика до начала разработки ПС.*
- Для этого следует использовать *метод анализа, выявления и освоения проблемы и интересов заказчика:*
- **достигнуть соглашения между заказчиком и разработчиком по определению проблемы, целей и задач проекта;**
- **выделить основные причины – проблемы, являющиеся её источниками и стоящие за основной проблемой проекта системы и ПС;**
- **выявить заинтересованных лиц и пользователей, чье коллективное мнение и оценка в конечном итоге определяет успех или неудачу проекта;**
- **определить, где приблизительно находятся область и границы возможных решений проблем;**
- **понять ограничения, которые будут наложены на проект, команду и решения проблем.**

Для сложных систем требуются стратегии управления информацией о требованиях.

Для этого применяется информационная иерархия; она начинается с потребностей пользователей, описанных с помощью функций, которые затем превращаются в более подробные требования к ПС, выраженные посредством прецедентов или традиционных форм описания и стандартизованных характеристик.

Эта иерархия отражает уровень абстракции при рассмотрении взаимосвязи области проблемы и области решения.

Концепцию требований, модифицированную в соответствии с конкретным содержанием комплекса программ, необходимо иметь в каждом проекте. В требованиях к ПС следует указывать, какие функции должны осуществляться, а не то, как они могут реализоваться. Они используются для задания функциональных и конструктивных требований, а также ограничений ресурсов проектирования.

Концепция требований к проекту (или системный проект) должна быть “живым” документом, чтобы было легче его использовать и пересматривать.

Следует сделать концепцию *официальным каналом* изменения функций так, чтобы проект всегда имел достоверный, соответствующий современному состоянию документ.

Проекты, как правило, инициируются с объемом функциональных возможностей, значительно превышающим тот, который разработчик может реализовать, обеспечив приемлемое качество при заданных ресурсах. Тем не менее, необходимо ограничиваться, чтобы иметь возможность предоставить в срок достаточно целостный и качественный продукт.

Существуют различные методы задания очередности выполнения (приоритетов) требований и понятие базового уровня – совместно согласованного представления о том, в чем будут состоять ключевые функции системы как продукта проекта – понятие состава требований, задающих *ориентир* для принятия решений и их оценки.

Если масштаб проекта и сопутствующие требования заказчика превышают реальные ресурсы, в любом случае придется ограничиваться в функциях и качестве ПС.

Поэтому следует определять, что обязательно должно быть сделано в версии программного продукта при имеющихся ресурсах проекта.

Для этого придется вести переговоры.

ПРОЦЕССЫ РАЗРАБОТКИ ТРЕБОВАНИЙ К ХАРАКТЕРИСТИКАМ СЛОЖНЫХ ПРОГРАММНЫХ СИСТЕМ

ПРИ ПРОЕКТИРОВАНИИ РЕКОМЕНДУЕТСЯ НАБОР КРИТЕРИЕВ КАЧЕСТВА ТРЕБОВАНИЙ К КОМПЛЕКСАМ ПРОГРАММ, КОТОРЫЙ ВКЛЮЧАЕТ:

- **корректность** – отсутствуют дефекты и ошибки в формулировках требований к комплексу программ;
- **недвусмысленность** – каждое требование должно быть однозначно и не допускать различного понимания и толкования специалистами;
- **полнота** – состав и содержание требований должны быть достаточны для производства и применения корректного комплекса программ и компонентов;
- **непротиворечивость** – между разными требованиями к компонентам и комплексу программ отсутствуют конфликты и противоречия;
- **модифицируемость** – каждое требование допускает возможность его простого и согласованного изменения и развития при производстве комплекса программ;
- **трассируемость** – требование имеет однозначный идентификатор и возможность детализации и перехода к производству компонента или комплекса программ.

**ПРИ ПЛАНИРОВАНИИ, РАЗРАБОТКЕ И РЕАЛИЗАЦИИ
ТРЕБОВАНИЙ К ХАРАКТЕРИСТИКАМ КАЧЕСТВА ПС
НЕОБХОДИМО, В ПЕРВУЮ ОЧЕРЕДЬ, УЧИТЬ ВАТЬ
СЛЕДУЮЩИЕ ОСНОВНЫЕ ФАКТОРЫ:**

- **функциональную пригодность
(функциональность) конкретного проекта
ПС;**
- **возможные конструктивные
характеристики качества комплекса
программ, необходимые для улучшения
функциональной пригодности;**
- **доступные ресурсы для создания и
обеспечения всего жизненного цикла ПО с
требуемым качеством.**

Требования к характеристикам комплексов программ, определяющие их функциональную пригодность, разделяются на две группы:

- *требования к количественным, измеряемым, функциональным характеристикам, непосредственно влияющим на оперативное функционирование и возможность применения программного продукта, в которые входят требования к надежности, безопасности, производительности, допустимым рискам применения;*
- *требования к структурным характеристикам, определяющим архитектуру комплекса программ, влияющие на возможности его модификации и сопровождения версий, на мобильность и переносимость на различные платформы, на документированность, удобство практического освоения и применения программного продукта вне оперативного функционирования.*

*Разработку и утверждение требований к
характеристикам и атрибутам качества без учета рисков,
целесообразно проводить итерационно на этапах
системного и детального проектирования ПС.*

На этапах проектирования последовательно должны определяться требования:

- при проектировании концепции – предварительные требования к назначению, функциональной пригодности и к номенклатуре необходимых конструктивных характеристик качества ПС;
- при предварительном проектировании – требования к шкалам и мерам применяемых атрибутов характеристик качества с учетом общих ограничений ресурсов;
- при детальном проектировании – подробные требования к атрибутам качества с детальным учетом и распределением реальных ограниченных ресурсов, а также, возможно, их оптимизация по критерию качество/затраты.

ЗАКАЗЧИК



ДИЗАЙНЕР



ВЕРСТАЛЬЩИК



ПРОГРАММИСТ



Эти требования закрепляются в контракте и техническом задании, по которым разработчик впоследствии должен отчитываться перед заказчиком при завершении проекта или его версии.

Выбор требований к характеристикам и атрибутам качества при проектировании программных систем начинается с *определения исходных данных*. Для корректного выбора и установления требований к характеристикам качества, прежде всего, необходимо *определить особенности проекта*:

- **класс, назначение и основные функции создаваемой ПС;**
- **комплект стандартов и их содержание, которые целесообразно использовать при выборе характеристик качества ПС;**
- **состав потребителей характеристик качества ПС, для которых важны соответствующие атрибуты качества;**
- **реальные ограничения всех видов ресурсов проекта.**



Этап разработки концепции проекта целесообразно начинать с формализации и обоснования набора исходных данных, отражающих общие особенности класса, потребителей и этапов жизненного цикла проекта ПС, каждый из которых влияет на выбор определенных характеристик качества комплекса программ.

Из конструктивных характеристик и атрибутов качества, прежде всего, следует выделять те, на которые в наибольшей степени воздействует класс, назначение и функции ПС.

Требования стандартов к функциональной пригодности должны выполняться для любых классов и назначения ПС. Однако номенклатура учитываемых требований к конструктивным характеристикам качества существенно зависит от назначения и функций комплексов программ. Так, например, при проектировании:

систем управления объектами в реальном времени наибольшее значение имеет защищенность, корректность и надежность функционирования стабильного комплекса программ и менее важно может быть качество обеспечения сопровождения и конфигурационного управления, способность к взаимодействию и практичность;

административных систем кроме корректности, важно обеспечивать практичность применения, комфортное взаимодействие с пользователями и внешней средой и может не иметь особого значение эффективность использования вычислительных ресурсов и обеспечение мобильности комплекса программ;

операционных систем наиболее жесткие требования предъявляются к корректности, эффективности использования вычислительных ресурсов и защищенности, и не всегда могут учитываться мобильность и унификация способности к взаимодействию её компонентов;

пакетов прикладных программ для вычислений или моделирования процессов, возможно не учитывать их мобильность, защищенность и временную эффективность, но особенно важна корректность.

СТРУКТУРА ОСНОВНЫХ ДОКУМЕНТОВ, ОТРАЖАЮЩИХ ТРЕБОВАНИЯ К ПРОГРАММНЫМ СИСТЕМАМ

При разработке требований к проектам программных средств, кроме основных целей, назначения и функций важно учесть и сформулировать содержание достаточно полного множества характеристик, каждая из которых может влиять на успех проекта программного продукта. Для уменьшения вероятности случайного пропуска важного требования заказчикам и пользователям целесообразно иметь типовые проекты перечней (шаблоны) наборов требований, которые можно целеустремленно сокращать и адаптировать, обеспечивая целостность требований для конкретных проектов ПС.

СОСТАВ КОНЦЕПЦИИ ОСНОВНЫХ ТРЕБОВАНИЙ К ПРОГРАММНОМУ СРЕДСТВУ:

- **описание обобщенных результатов обследования и изучения существующей системы и внешней среды;**
- **описание целей, назначения программного продукта и потребностей заказчика и потенциальных пользователей к нему в заданной среде применения;**
- **перечень базовых стандартов предполагаемого проекта программного продукта;**
- **общие требования к характеристикам комплекса задач ПС:**
 - цели создания программного продукта и назначение комплекса функциональных задач;
 - перечень объектов среды применения ПС (технологических объектов управления, подразделений предприятия и т. п.), при управлении которыми должен решаться комплекс задач;
 - периодичность и продолжительность решения комплекса задач;
 - связи и взаимодействие комплекса задач с внешней средой и другими компонентами системы;
 - распределение функций между персоналом, программными и техническими средствами при различных ситуациях решения требуемого комплекса функциональных задач;



СОСТАВ КОНЦЕПЦИИ ОСНОВНЫХ ТРЕБОВАНИЙ К ПРОГРАММНОМУ СРЕДСТВУ:

- **сопоставительный анализ требований заказчика и пользователей к программному продукту и набора функций в концепции ПС для удовлетворения требований заказчика и пользователей;**
- **обоснование выбора оптимального варианта требований к содержанию и приоритетам комплекса функций ПС в концепции;**
- **общие требования к структуре, составу компонентов и интерфейсам с внешней средой;**
- **ожидаемые результаты и возможная эффективность реализации выбранного варианта требований в концепции ПС;**
- **ориентировочный план реализации выбранного варианта требований концепции ПС;**
- **общие требования к составу и содержанию документации проекта ПС;**
- **оценка необходимых затрат ресурсов на разработку, ввод в действие и обеспечение функционирования ПС;**
- **предварительный состав требований, гарантирующих качество применения ПС;**
- **предварительные требования к условиям испытаний и приемки системы и ПС.**

СОСТАВ КОНЦЕПЦИИ ОСНОВНЫХ ТРЕБОВАНИЙ К ПРОГРАММНОМУ СРЕДСТВУ:

требования к входной информации:

- источники информации и их идентификаторы;
- перечень и описание входных сообщений (идентификаторы, формы представления, регламент, сроки и частота поступления);
- перечень и описание структурных единиц информации входных сообщений или ссылка на документы, содержащие эти данные;

требования к выходной информации:

- потребители и назначение выходной информации;
- перечень и описание выходных сообщений;
- регламент и периодичность их выдачи;
- допустимое время задержки решения определенных задач;

описание и оценка преимуществ и недостатков разработанных альтернативных вариантов функций в концепции создания проекта ПС;

СПЕЦИФИКАЦИЯ ТРЕБОВАНИЙ К СИСТЕМЕ И К КОМПЛЕКСУ ПРОГРАММ НА ЭТАПЕ ДЕТАЛЬНОГО ПРОЕКТИРОВАНИЯ:

- **требования проекта системы к комплексу программ, как к целому в общей архитектуре системы;**
- **требования к унификации интерфейсов и базы данных комплекса программ;**
- **требования и обоснование выбора проектных решений уровня системы, состава компонентов системы, описание функций системы и ПС с точки зрения пользователя;**
- **спецификация требований верхнего уровня комплекса программ, производные требования к компонентам ПС и требования к интерфейсам между системными компонентами, элементами конфигурации ПС и аппаратуры;**
- **описание распределения системных требований по компонентам ПС с учетом требований, которые обеспечивают заданные характеристики качества;**
- **требования к архитектуре системы, содержащей идентификацию и функции компонентов системы, их назначение, статус разработки, аппаратные и программные ресурсы;**
- **требования совместного целостного функционирования компонентов ПС, описание и характеристики их динамических связей;**

СПЕЦИФИКАЦИЯ ТРЕБОВАНИЙ К СИСТЕМЕ И К КОМПЛЕКСУ ПРОГРАММ НА ЭТАПЕ ДЕТАЛЬНОГО ПРОЕКТИРОВАНИЯ

требования анализа трассируемости функций компонентов программного средства к требованиям проекта системы;

требования для системы или/и подсистем и методы, которые должны быть использованы для гарантии того, что каждое требование к комплексу программ будет выполнено и прослеживаемо к конкретным требованиям системы:

- к режимам работы;
- к производительности системы;
- к внешнему и пользовательскому интерфейсу системы;
- к внутреннему интерфейсу компонентов и к внутренним данным системы;
- по возможности адаптации ПС к внешней среде;
- по обеспечению безопасности системы, ПС и внешней среды;
- по обеспечению защиты, безопасности и секретности данных;
- по ограничениям доступных ресурсов проекта ПС;
- по обучению и уровню квалификации персонала;
- по возможностям средств аттестации результатов и компонентов, включающие в себя демонстрацию, тестирование, анализ, инспекцию и требуемые специальные методы для контроля функций, и качества конкретной системы или компонента ПС.

СПАСИБО ЗА ВНИМАНИЕ!

ПРОЕКТИРОВАНИЕ ПРОГРАММНЫХ СИСТЕМ

УНИФИЦИРОВАННЫЙ ЯЗЫК МОДЕЛИРОВАНИЯ UML

ЛЕКЦИЯ 2

- **Важный принцип создания проектов на многих ООП языках программирования – сначала проектирование, а потом программирование**
- **На этапе проектирования и происходит использование UML-диаграмм (самый популярный инструмент)**

ЧТО ТАКОЕ UML

- **UML – аббревиатура полного названия Unified Modeling Language.**
- **Правильный перевод этого названия на русский язык – унифицированный язык моделирования.** Таким образом, обсуждаемый предмет характеризуется тремя словами, каждое из которых является точным термином.

ЧТО ТАКОЕ UML

Язык — это знаковая система для хранения и передачи информации.

UML — язык графического описания для объектного моделирования в области разработки программного обеспечения. UML является языком широкого профиля, это — открытый стандарт, использующий графические обозначения для создания абстрактной модели системы, называемой UML-моделью.

ЧТО ТАКОЕ UML

Спецификация – это декларативное описание того, как нечто устроено или работает.

Основное назначение UML – предоставить, с одной стороны, достаточно формальное, с другой стороны, достаточно удобное, и, с третьей стороны, достаточно универсальное средство, позволяющее до некоторой степени снизить риск расхождений в толковании спецификаций

КТО МОЖЕТ ИСПОЛЬЗОВАТЬ UML

- Заказчик – общие цели и задачи проекта, что будет выполнять система
- Аналитик – проверяет подходы, правильность работы системы и ее частей, «слои» приложения
- Разработчик/архитектор – чаще всего дизайн кода, архитектура классов, объектов, взаимодействий
- Тестировщик – может проверять все уровни, также производительность по диаграммам времени
- Менеджер – общая картина проекта, самый верхний уровень

- UML задает описание или поведение процесса, но не показывает реализацию
- Цели UML – проектирование, документирование, визуальное описание основных моментов проекта
- Ключевое понятие – диаграмма (есть разные типы) – визуальное описание процесса, классов, взаимодействия и пр.
- На каждый процесс или тип задачи – своя диаграмма
- UML не является языком программирования (но можно генерировать готовый код из диаграмм)
- Хранение информации в электронном виде (легче программировать, так как не надо держать в голове проект)
- Позволяет посмотреть на проект с верхнего уровня – сразу видны ключевые моменты
- Дополняет стандартные документы типа ТЗ
- Нужно хорошо знать ООП

UML - ЭТО ЯЗЫК

Язык — это знаковая система для хранения и передачи информации.

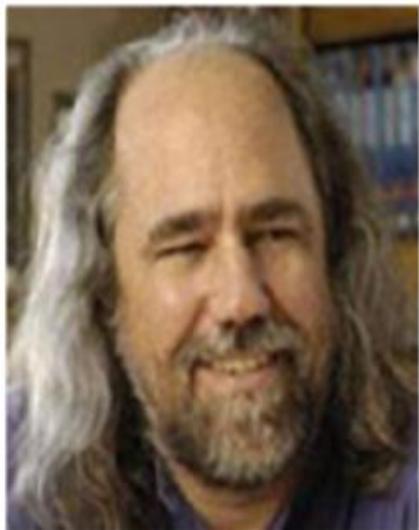
Формальный искусственный язык описан, если описание содержит:

- Синтаксис (*syntax*).
- Семантика (*semantics*).
- Прагматика (*pragmatics*).

УНИФИЦИРОВАННЫЙ ЯЗЫК МОДЕЛИРОВАНИЯ UML

Unified Modeling Language (UML) - это язык для определения, представления, проектирования и документирования программных систем, организационно-экономических систем, технических систем и других систем различной природы.

ИСТОРИЯ РАЗВИТИЯ UML



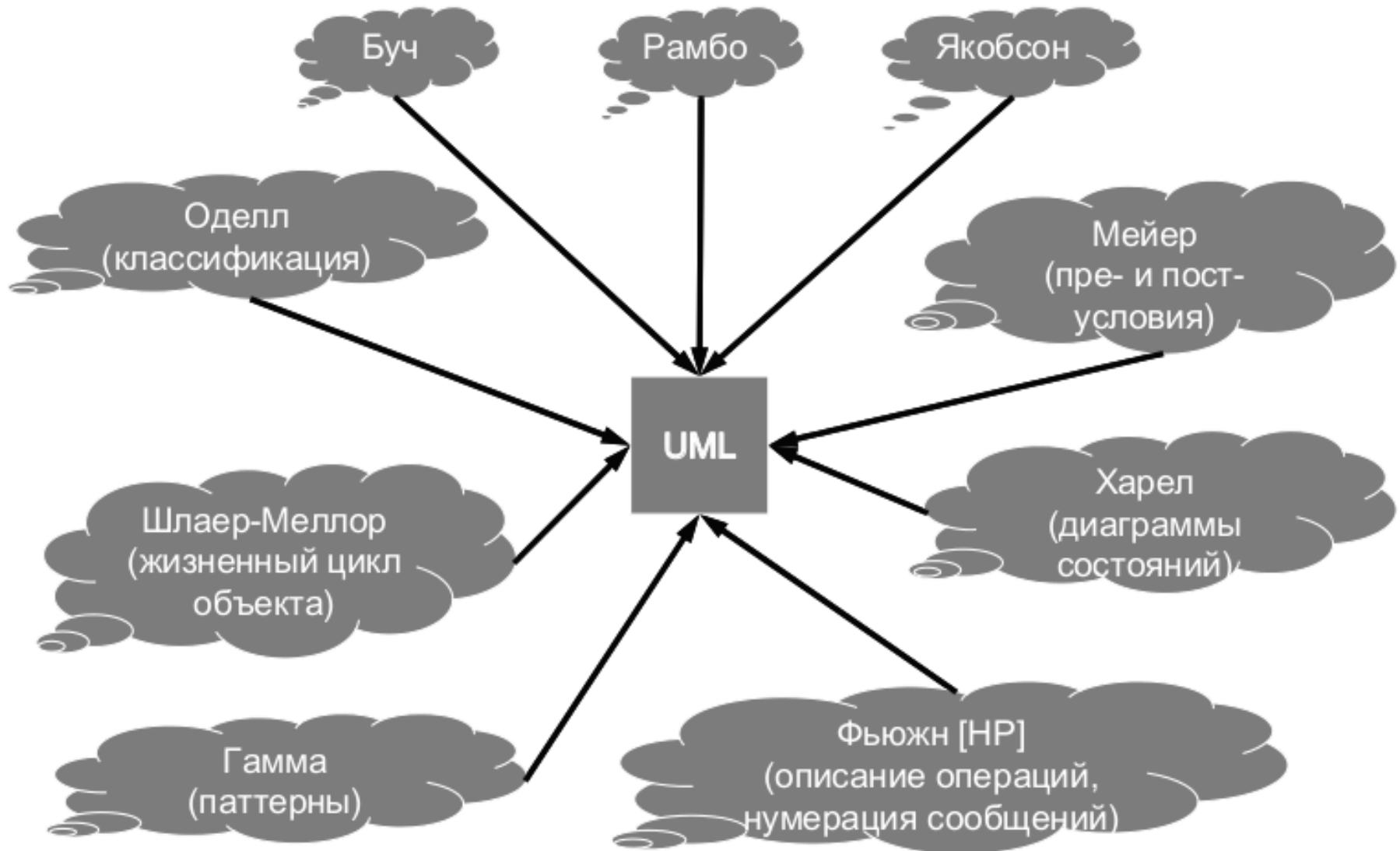
Grady Booch
Грэди Буч



James Rumbaugh
Джеймс Рамбо



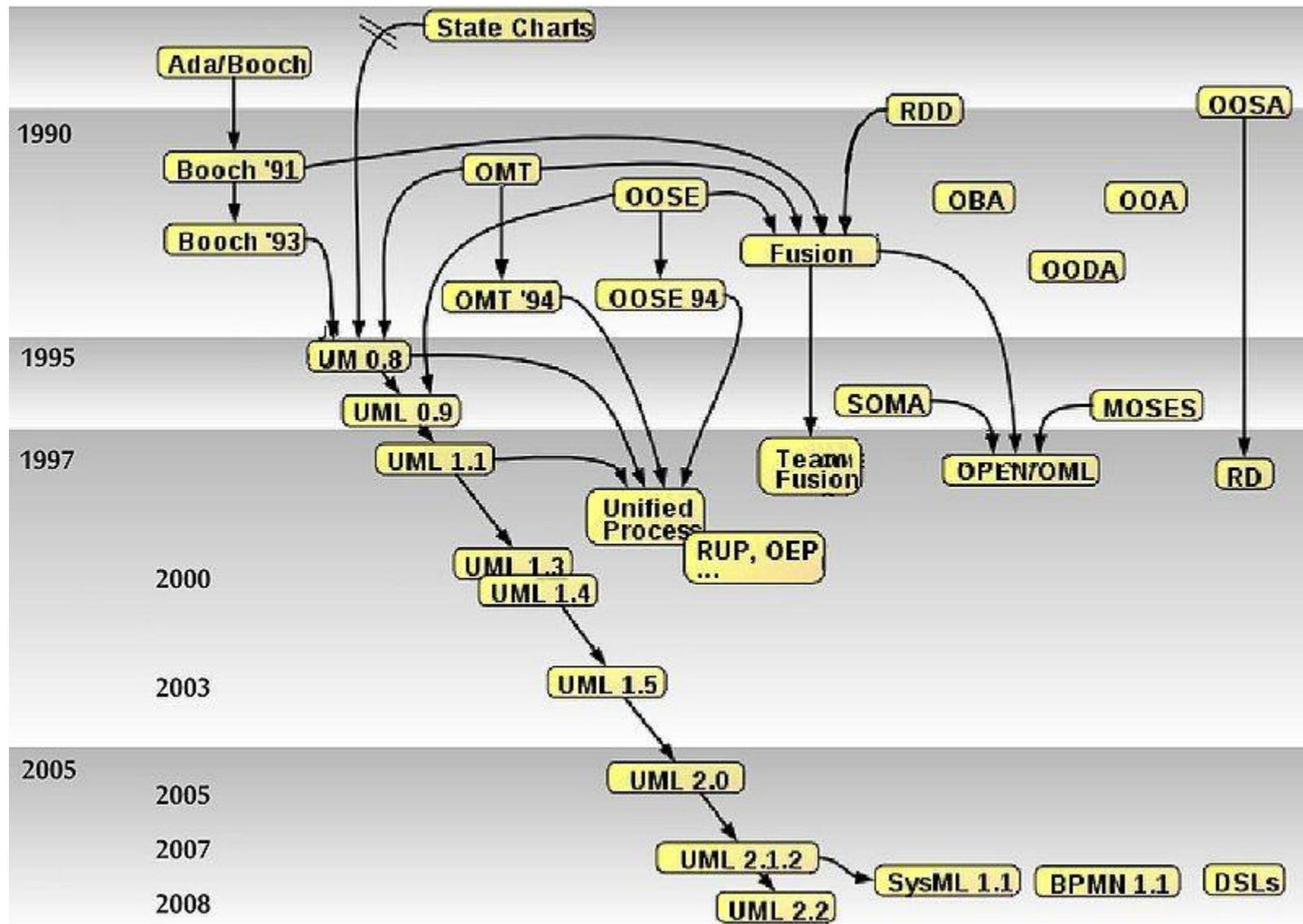
Ivar Jacobson
Айвар Якобсон



Вклад в развитие UML

ЦЕЛИ РАЗВИТИЯ UML

- предоставить разработчикам единый язык визуального моделирования;
- предусмотреть механизмы расширения и специализации языка;
- обеспечить независимость языка от языков программирования и процессов разработки;
- интегрировать накопленный практический опыт.



История развития UML

ИНСТРУМЕНТЫ ДЛЯ РАЗРАБОТКИ НА UML

- Онлайн инструмент
- Отдельные программы
- Плагины
- Специальные IDE

БЛОКИ UML

- **элементы модели (классы, интерфейсы, компоненты, варианты использования и др.);**
- **связи (ассоциации, обобщения, зависимости и др.);**
- **механизмы расширения (стереотипы, ограничения, метасвойства, примечания);**
- **диаграммы.**

СОСТАВ ДИАГРАММ UML 1.X

★ структурные:

- диаграммы классов
- диаграммы компонентов
- диаграммы размещения

★ поведенческие:

- диаграммы вариантов использования
- диаграммы взаимодействия:
 - диаграммы последовательности
 - коммуникационные диаграммы
- диаграммы состояний
- диаграммы деятельности

КЛАССИФИКАЦИЯ ДИАГРАММ В UML 2.X

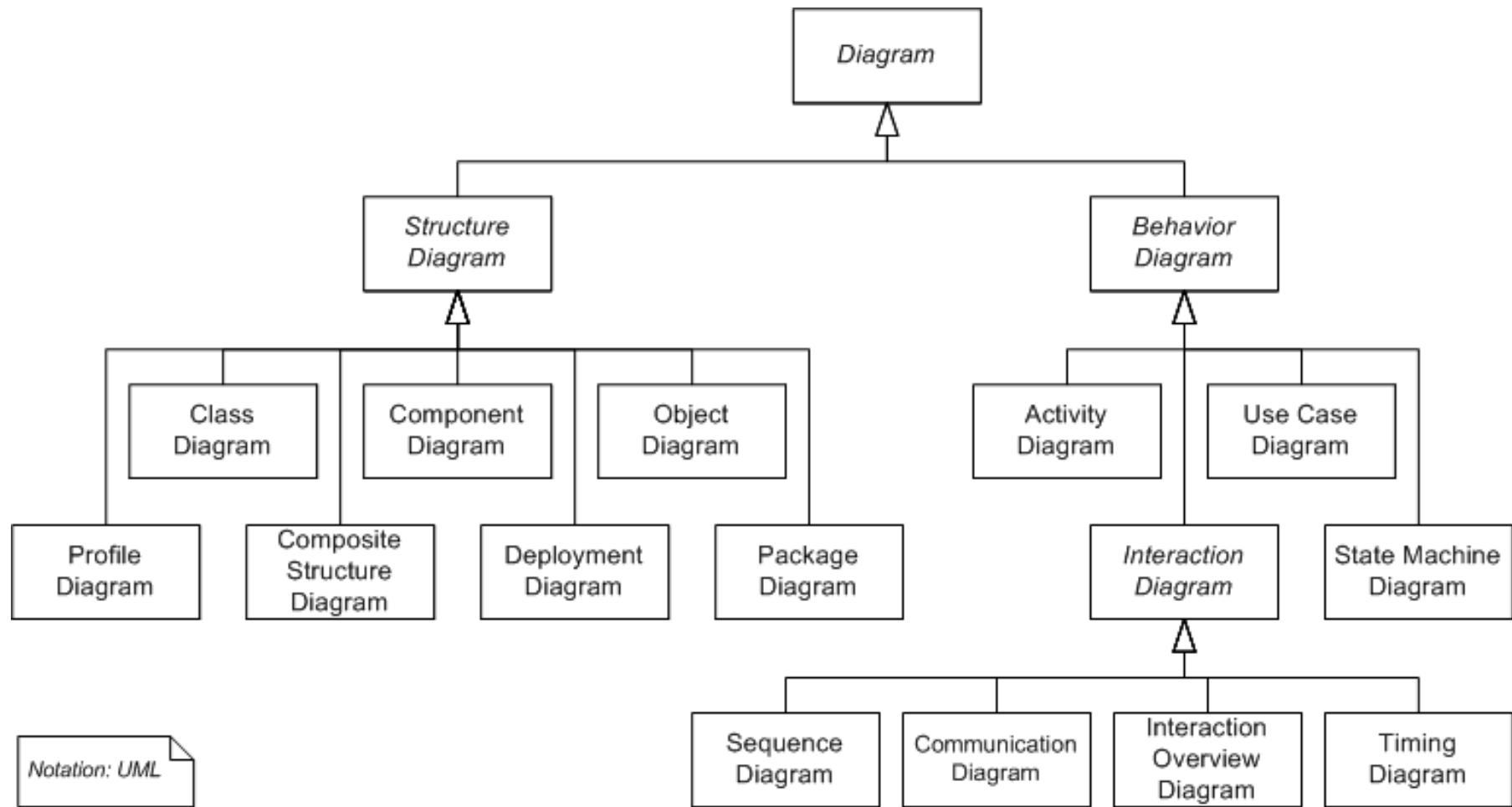
Диаграммы поведения:

- Диаграммы вариантов использования
- Диаграммы деятельности
- Диаграммы состояний
- Диаграммы взаимодействия:

- Диаграммы последовательности
- Диаграммы коммуникации
- Диаграммы синхронизации
- Обзорные диаграммы взаимодействия

Структурные диаграммы:

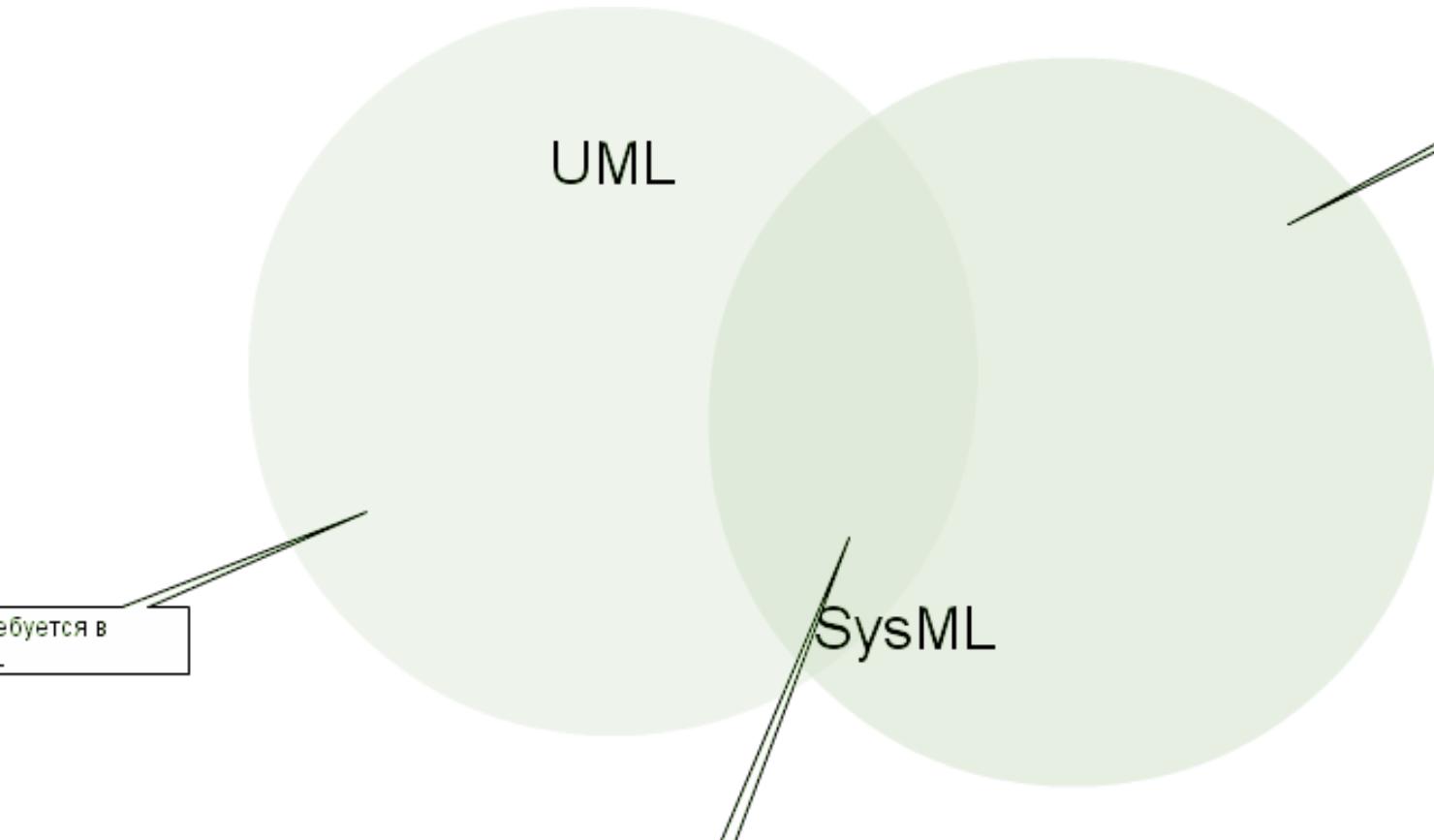
1. Диаграммы классов
2. Диаграммы объектов
3. Диаграммы пакетов
4. Диаграммы компонентов
5. Диаграммы составной структуры
6. Диаграммы размещения
7. Диаграммы профиля



Состав диаграмм UML 2.x

SYSTEMS MODELING LANGUAGE (SYSML)

SysML (англ. *The Systems Modeling Language*, язык моделирования систем) — предметно-ориентированный язык моделирования систем. Поддерживает определение, анализ, проектирование, проверку и подтверждение соответствия широкого спектра систем.

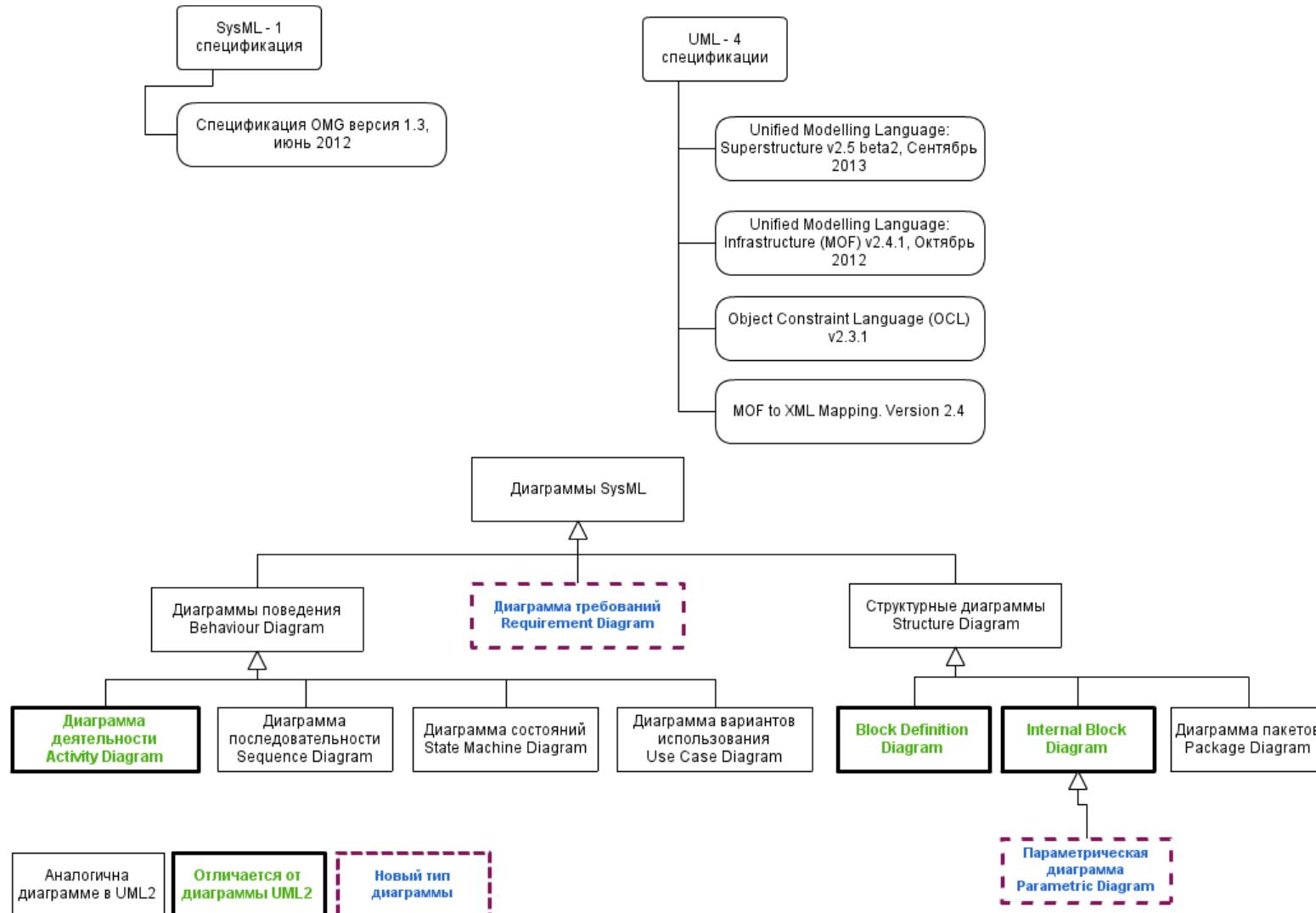


UML vs SysML

ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ SYSML

- Дополнительные диаграммы: диаграмма требований и параметрическая диаграмма.
- SysML - более компактный язык.
- Конструкции для управления моделями. поддерживает модели, представления (views) и точки зрения (viewpoints).

СПЕЦИФИКАЦИИ И СОСТАВ ДИАГРАММ SYSML



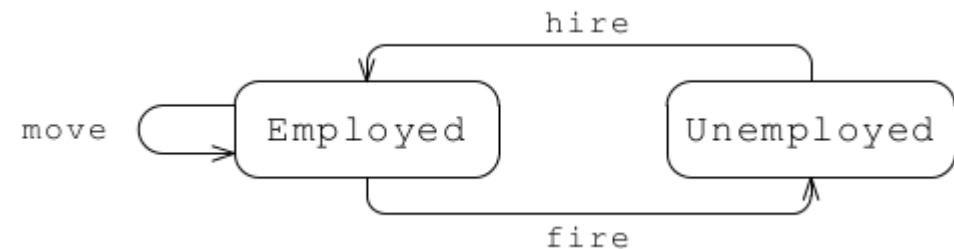
Одним из ключевых этапов разработки приложения является определение того, каким требованиям должно удовлетворять разрабатываемое приложение. В результате этого этапа появляется формальный или неформальный документ (артефакт), который называют по-разному, имея в виду примерно одно и то же: постановка задачи, требования, техническое задание, внешние спецификации и др.

Спецификация — это декларативное описание того, как нечто устроено или работает.

НАЗНАЧЕНИЕ UML

Язык UML — это графический язык моделирования общего назначения, предназначенный для спецификации, визуализации, проектирования и документирования всех артефактов, создаваемых при разработке программных систем.

1. Спецификация
2. Визуализация
3. Проектирование
4. Документирование



МОДЕЛЬ И ЕЕ ЭЛЕМЕНТЫ

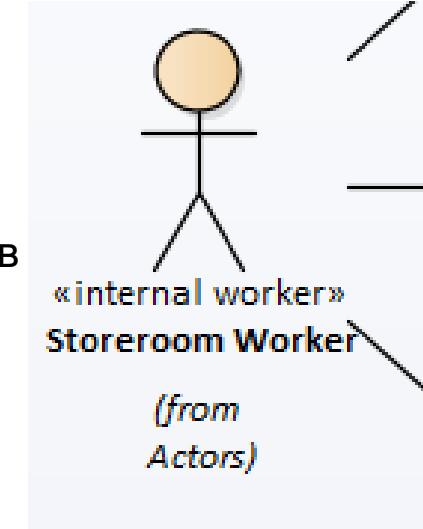
Модель UML (**UML model**) — это совокупность конечного множества конструкций языка, главные из которых — это сущности и отношения между ними.

- структурные
- поведенческие
- группирующие
- аннотационные

СТЕРЕОТИП

Стереотип – это определение нового элемента моделирования в UML на основе существующего элемента моделирования.

Определение стереотипа производится следующим образом. Взяв за основу некоторый существующий элемент модели, к нему добавляют новые помеченные значения (расширяя тем самым внутреннее представление), новые ограничения (расширяя семантику) и дополнения, то есть новые графические элементы (расширяя нотацию).



После того, как стереотип определен, его можно использовать как элемент модели нового типа.

Если при создании стереотипа не использовались дополнения и графическая нотация взята от базового элемента модели, на основе которого определен стереотип, то стереотип элемента обозначается при помощи имени стереотипа, заключенного в двойные угловые скобки (типоврафские кавычки), которое помещается перед именем элемента модели.

Если же для стереотипа определена своя нотация, например, новый графический символ, то указывается этот символ.

СТРУКТУРНЫЕ СУЩНОСТИ UML

- Объект
- Класс
- Интерфейс
- Кооперация
- Действующее лицо
- Компонент
- Артефакт
- Узел

ПОВЕДЕНЧЕСКИЕ И ДРУГИЕ СУЩНОСТИ

Поведенческие сущности:

- **Состояние**
- **Деятельность**
- **Действие**

Структурная и поведенческая сущность:

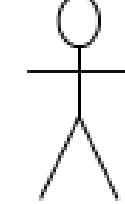
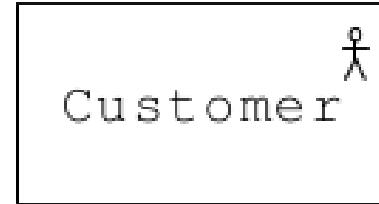
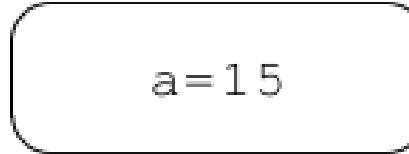
- **Вариант использования (ВИ)**

Группирующая сущность:

- **Пакет**

Аннотационная сущность:

- **Аннотация**

Название	Графическая нотация	
Артефакт	<p>«artifact»</p> <p>Requirement Specification</p>	<p>«library»</p> <p>QT</p>
Вариант использования	 <p>Make Order</p>	 <p>Make Order</p>
Действующее лицо	 <p>Customer</p>	 <p>Customer</p>
Деятельность и действие	 <p>Display main menu</p>	 <p>a=15</p>

Нотации основных сущностей

Название	Графическая нотация	
Интерфейс	IAudio	«interface» IAudio
Класс	Product	Order (abstract)
Компонент	DataBase	«component» DataBase
Кооперация	Visitor	

Нотации основных сущностей

Название	Графическая нотация
Объект	:Rectangle
Пакет	Analysis Model
Примечание	Здесь находится комментарий
Состояние	Logged entry/OpenLog exit/CloseLog
Узел	Server

Нотации основных сущностей

ТИПЫ ОТНОШЕНИЙ В UML

- зависимость
- ассоциация
- обобщение
- реализация

ДИАГРАММЫ ВАРИАНТОВ ИСПОЛЬЗОВАНИЯ И СЦЕНАРИИ

КЛАССИФИКАЦИЯ ДИАГРАММ

Диаграммы поведения:

- Диаграммы вариантов использования
- Диаграммы деятельности
- Диаграммы состояний
- Диаграммы взаимодействия:

- Диаграммы последовательности
- Диаграммы коммуникации
- Диаграммы синхронизации
- Обзорные диаграммы взаимодействия

Структурные диаграммы:

1. Диаграммы классов
2. Диаграммы объектов
3. Диаграммы пакетов
4. Диаграммы компонентов
5. Диаграммы составной структуры
6. Диаграммы размещения
7. Диаграммы профиля

ПРОЦЕСС ВИЗУАЛЬНОГО МОДЕЛИРОВАНИЯ

Визуальное моделирование с использованием нотации UML можно представить как процесс поуровневого спуска от наиболее общей и абстрактной концептуальной модели исходной бизнес-системы к логической, а затем и к физической модели соответствующей программной системы.

ДИАГРАММА ВАРИАНТОВ ИСПОЛЬЗОВАНИЯ (USE CASE DIAGRAM)

- 1) Исходное концептуальное представление или концептуальная модель системы в процессе ее проектирования и разработки.**
- 2) Диаграмма, на которой изображаются отношения между действующими лицами и вариантами использования.**

ЦЕЛИ СОЗДАНИЯ USE CASE DIAGRAM

- Определить общие границы и контекст моделируемой предметной области на начальных этапах проектирования системы
- Сформулировать общие требования к функциональному поведению проектируемой системы
- Разработать исходную концептуальную модель системы для ее последующей детализации в форме логических и физических моделей
- Подготовить исходную документацию для взаимодействия разработчиков системы с ее заказчиками и пользователями

ФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ И USE CASE

Диаграмма вариантов использования - общее представление функциональных требований к системе.

Детализация функциональных требований - описания вариантов использования. Каждый документ содержит один и более сценариев или потоков событий варианта использования.

ОБЯЗАТЕЛЬНЫЕ ЭЛЕМЕНТЫ ДИАГРАММ ВИ

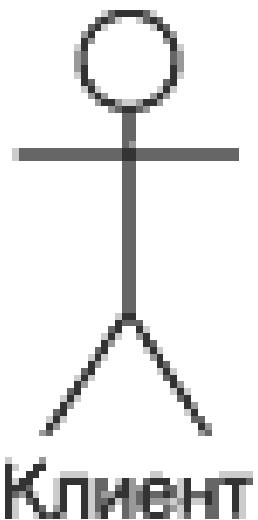
- Название моделируемой системы
- Границы системы
- Варианты использования
- Действующие лица
- Связи

ОСНОВНЫЕ ПОНЯТИЯ

- **Actor** – пользователь, действующее лицо (инициатор или исполнитель цели)
- **UseCase** – прецедент, цель, которую хочет достичь пользователь
- **Association** – ассоциация, связь между элементами (акторами и целями)
- **Include** – включения; возможные варианты реализации цели (или набор целей для реализации общей цели)
- **Extend** – расширения; новая цель, которая расширяет существующую

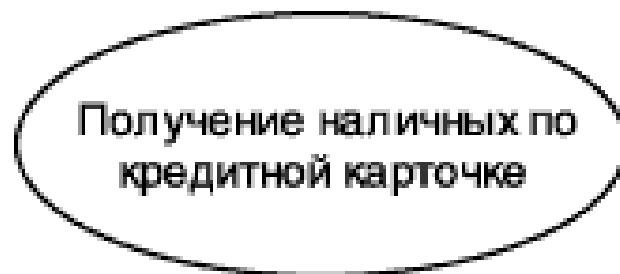
ДЕЙСТВУЮЩЕЕ ЛИЦО (дл)

Действующее лицо (actor) — согласованное множество ролей, которые играют внешние сущности по отношению к вариантам использования при взаимодействии с ними.



ВАРИАНТ ИСПОЛЬЗОВАНИЯ ИЛИ ПРЕЦЕДЕНТ (USE CASE)

Внешняя спецификация последовательности действий, которые система или другая сущность могут выполнять в процессе взаимодействия с действующими лицами.



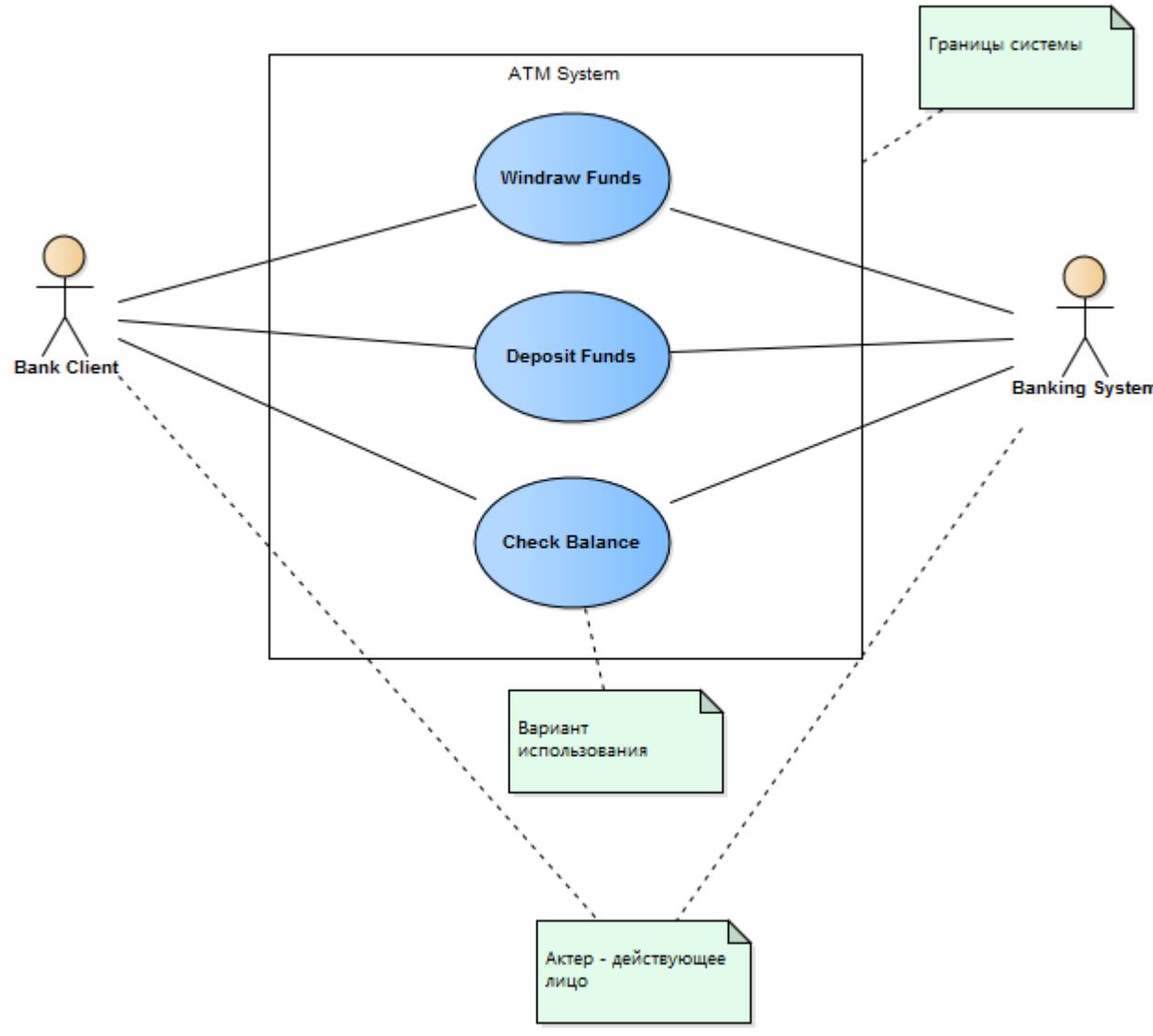
(a)



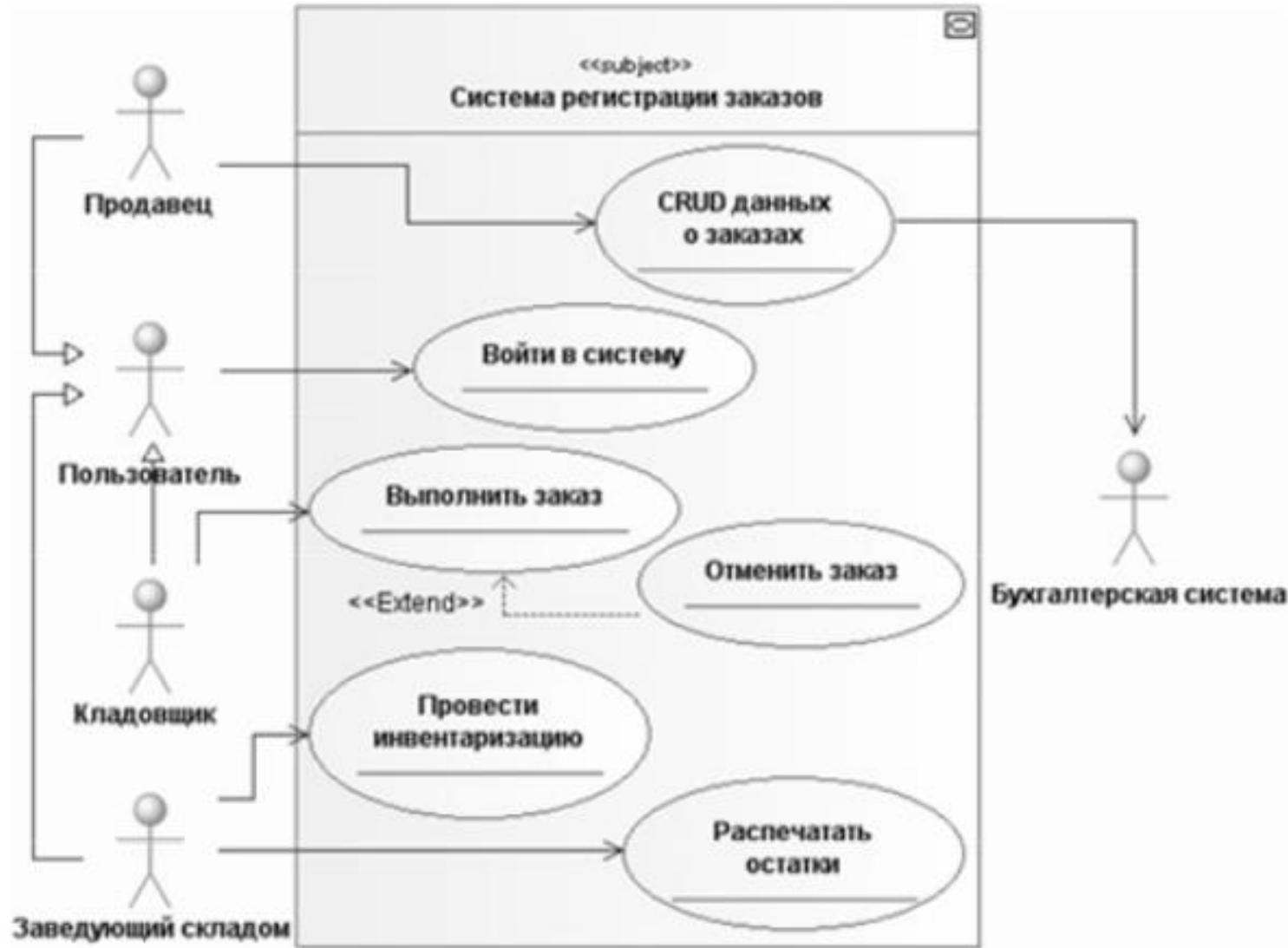
(б)

ПРИМЕРЫ ВИ

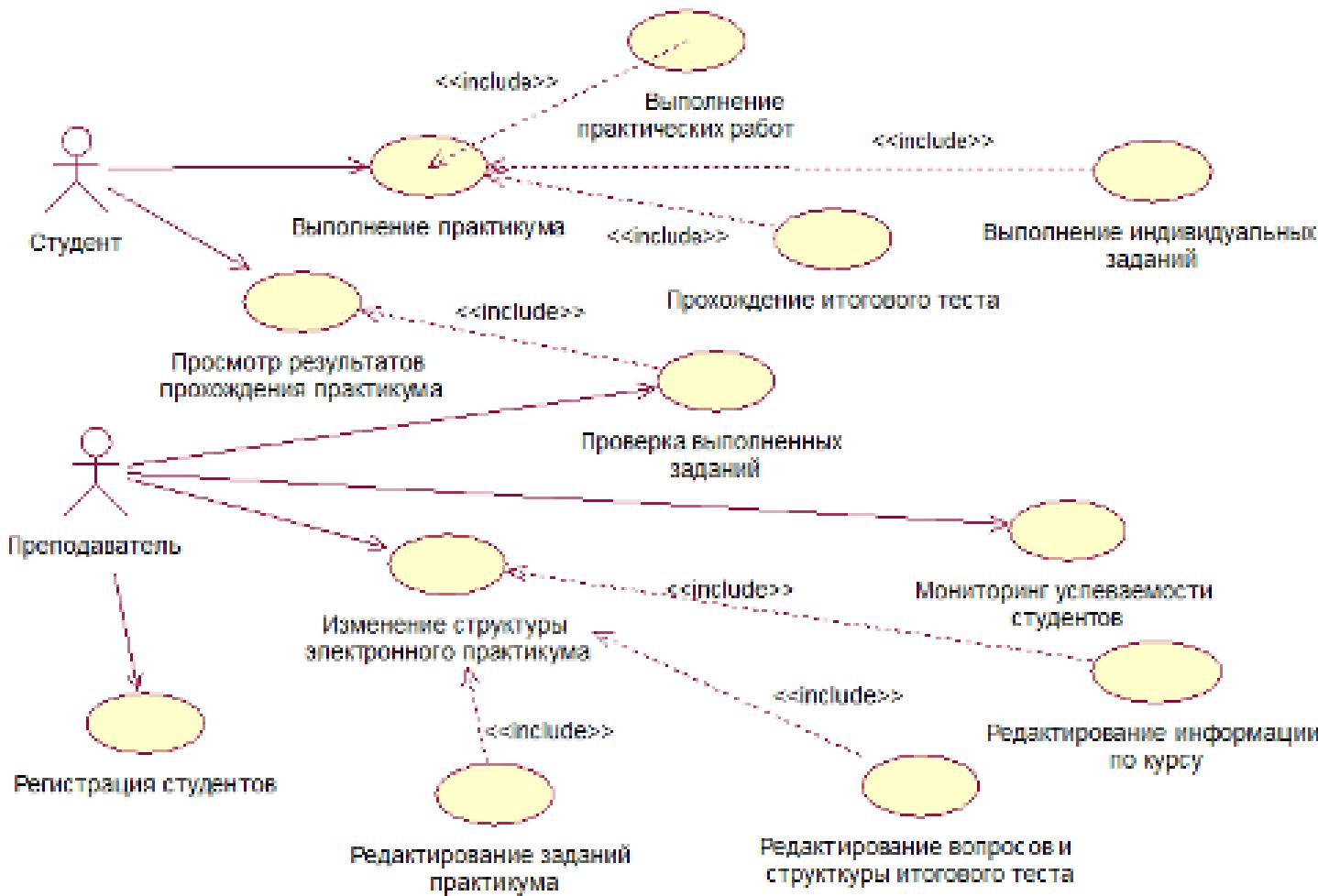
- Проверка состояния текущего счета клиента,
- Оформление заказа на покупку товара,
- Получение дополнительной информации о кредитоспособности клиента,
- Отображение графической формы на экране монитора.



Пример диаграммы вариантов использования



Пример диаграммы вариантов использования



Пример диаграммы вариантов использования

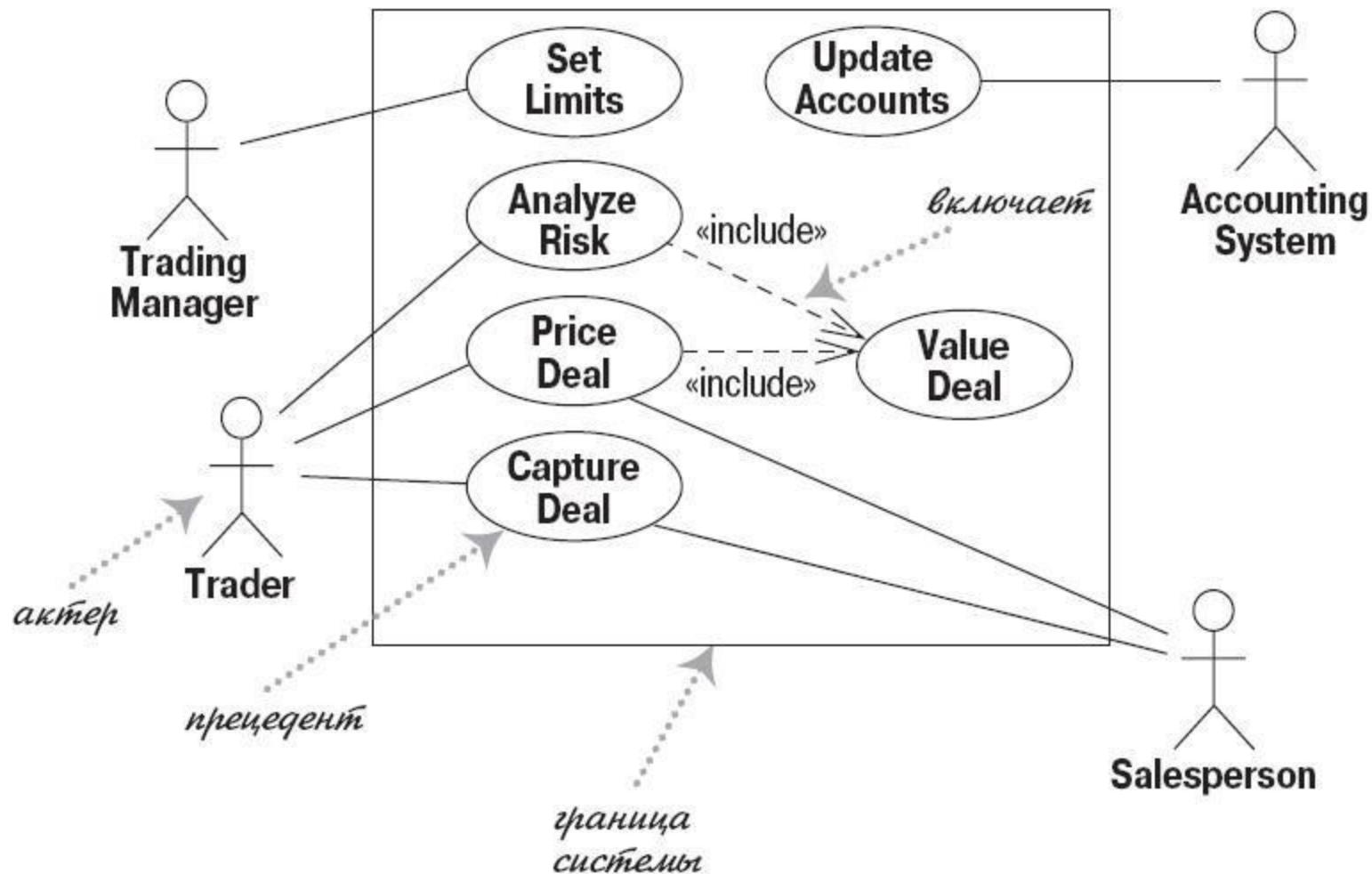
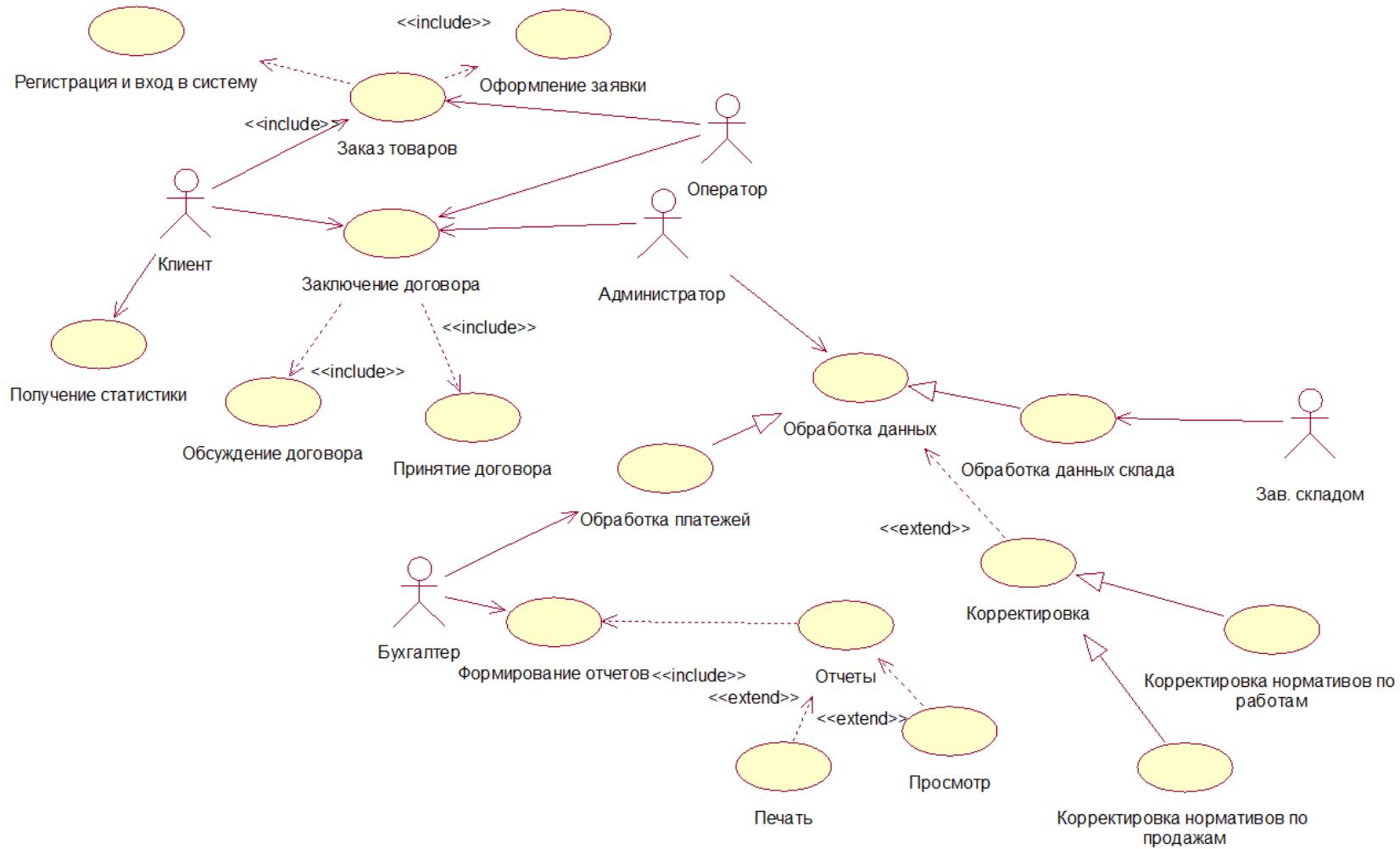
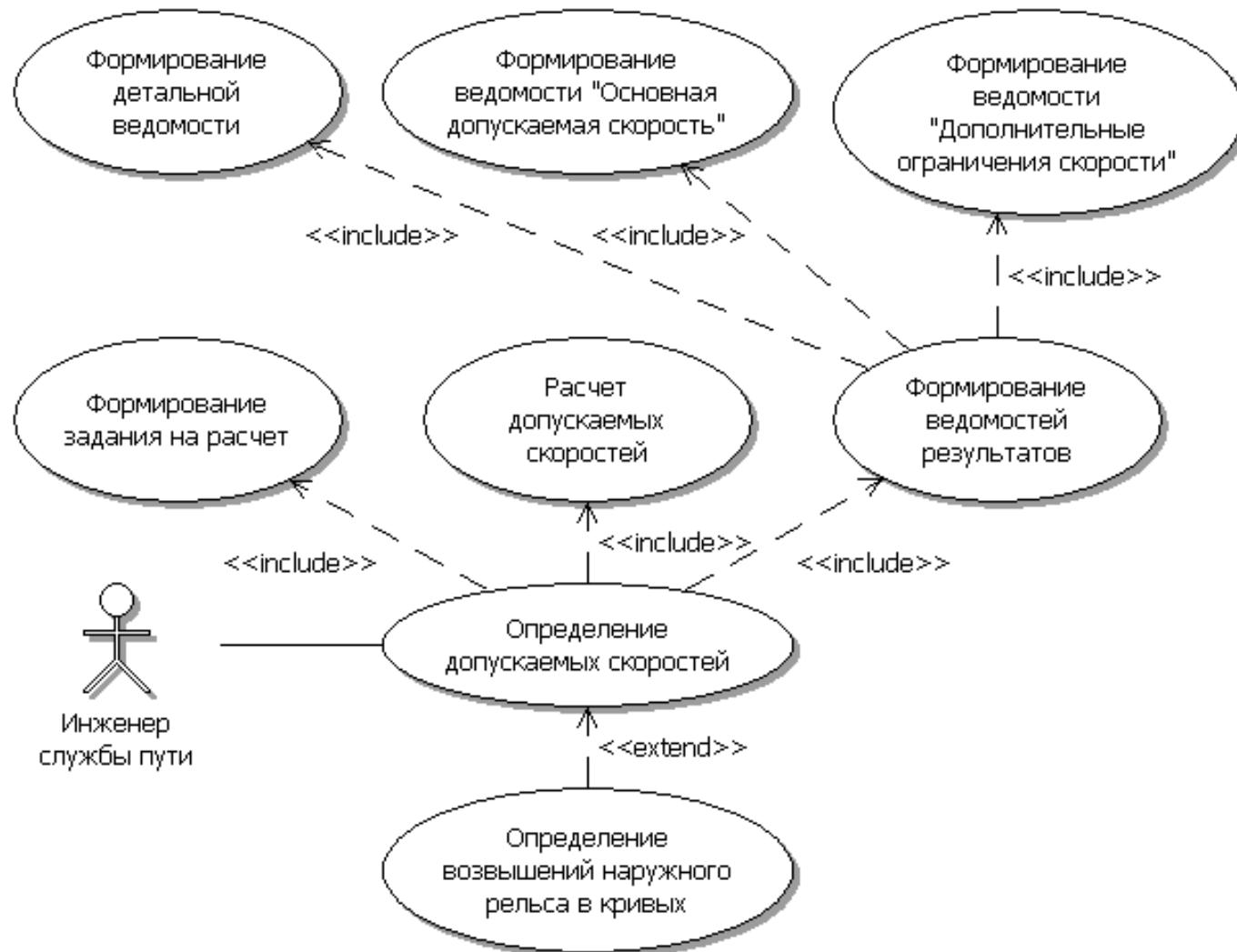


Рис. 9.2. Диаграмма прецедентов

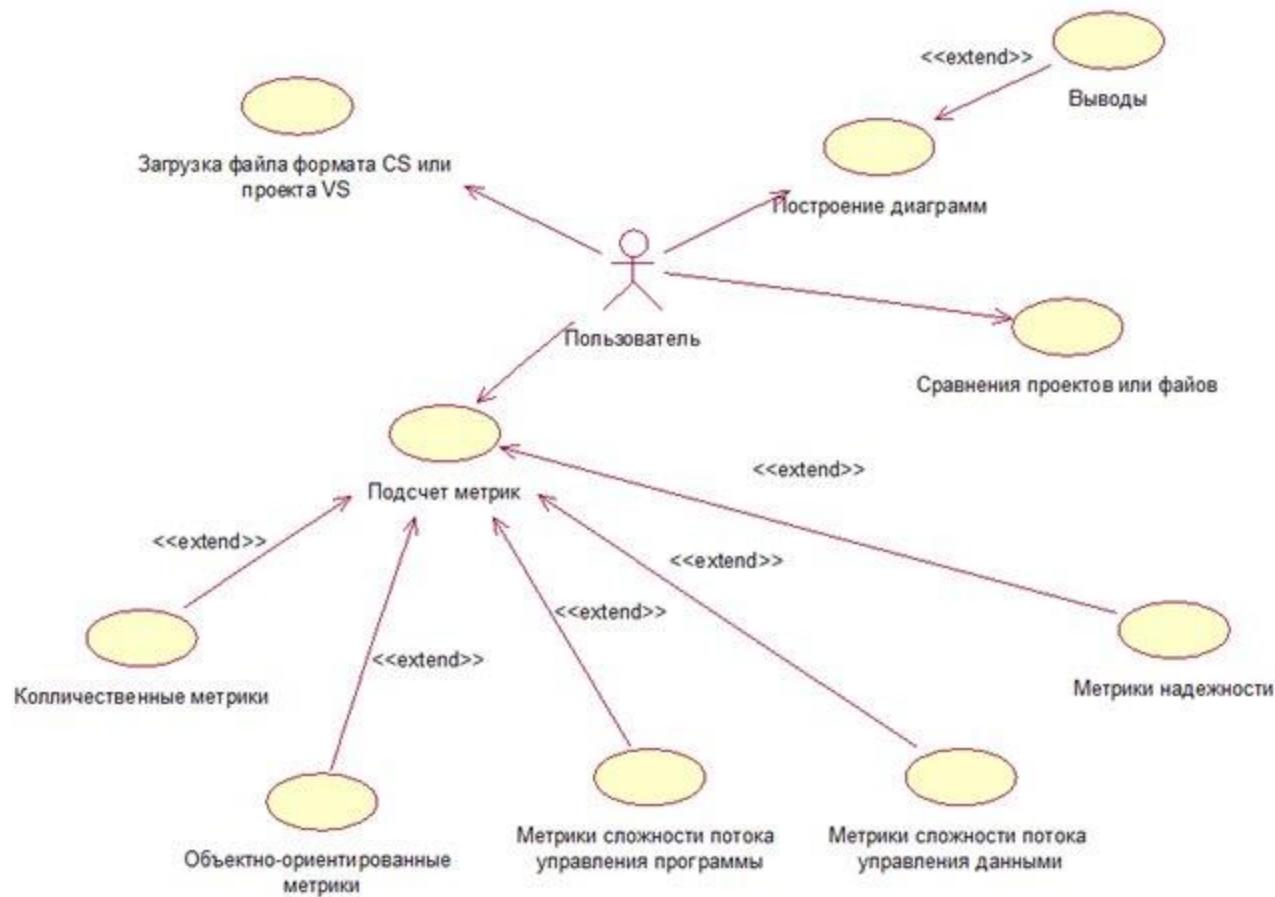
Пример диаграммы вариантов использования



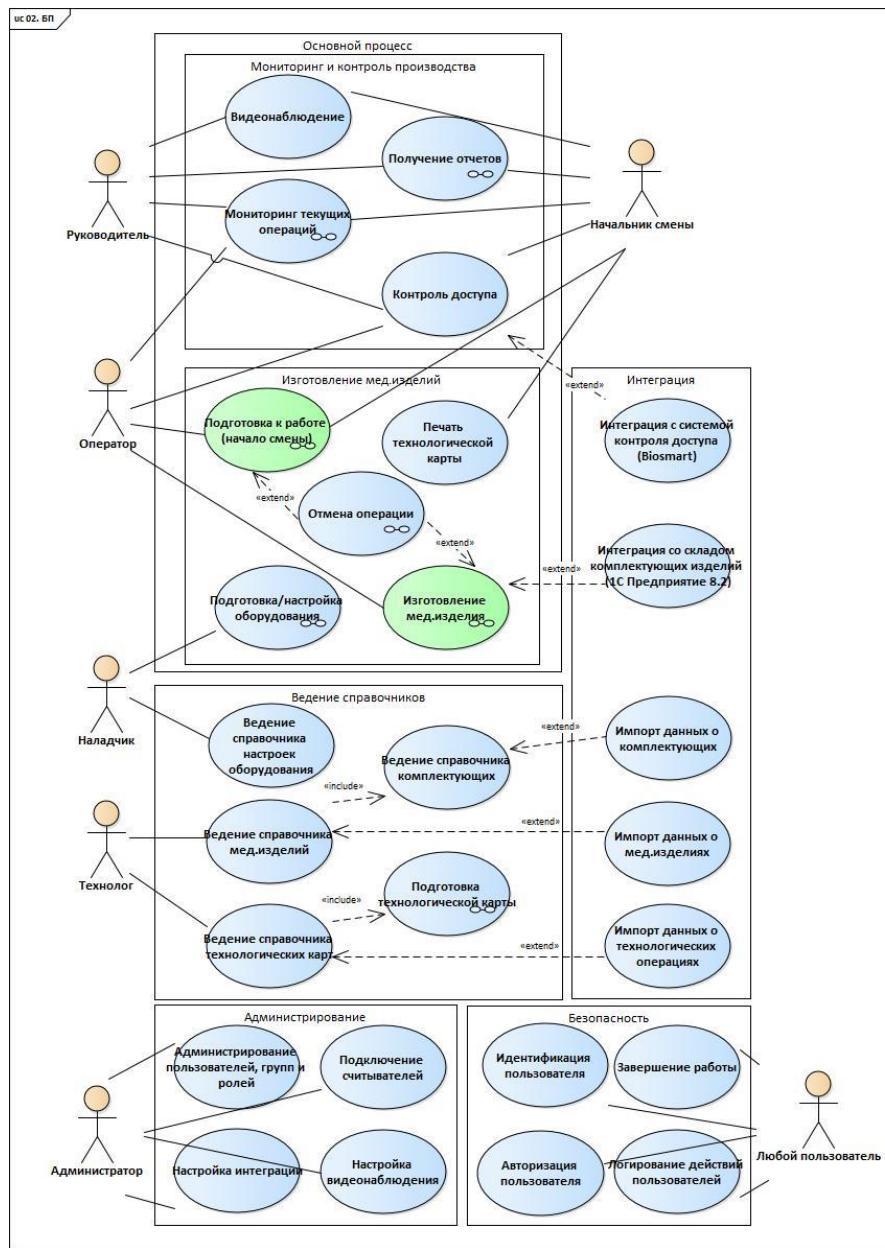
Пример диаграммы вариантов использования



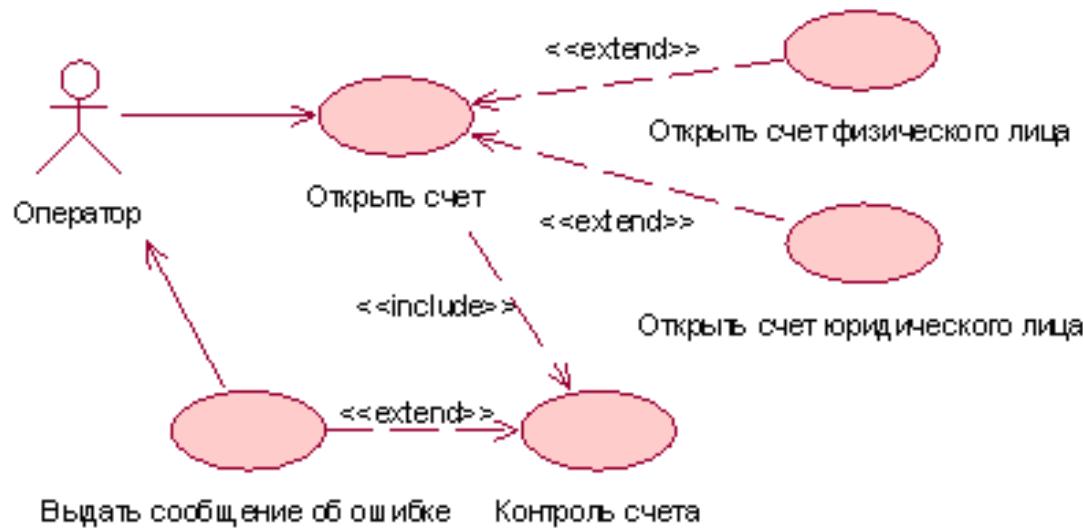
Пример диаграммы вариантов использования



Пример диаграммы вариантов использования



Пример диаграммы вариантов использования



Пример диаграммы вариантов использования

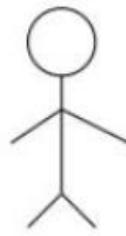
Вариант использования (use case) — внешняя спецификация последовательности действий, которые система или другая сущность могут выполнять в процессе взаимодействия с действующими лицами.

Вариант использования представляет собой спецификацию общих особенностей поведения или функционирования моделируемой системы без рассмотрения внутренней структуры этой системы.

Содержание варианта использования может быть представлено в форме дополнительного пояснительного текста, который раскрывает смысл или семантику действия при выполнении данного варианта использования. Такой пояснительный текст получил название **текста-сценария или просто **сценария**.**

ЦЕЛЬ СПЕЦИФИКАЦИИ (СЦЕНАРИЯ) ВИ

Цель спецификации варианта использования заключается в том, чтобы зафиксировать некоторый аспект или фрагмент поведения проектируемой системы без указания особенностей реализации данной функциональности.



Актер



ВИ на диаграмме



ВИ в текстовом виде

ВИ и спецификации (сценарии)

ТИПЫ СЦЕНАРИЕВ

Основной сценарий представляет собой последовательность действий, при успешном выполнении которых достигается цель варианта использования.

Расширения основного сценария описывают действия при возникновении исключительных ситуаций (действий, не предусмотренных основным сценарием, ошибках, внешних событиях и т.д.)

ОПИСАНИЕ СЦЕНАРИЕВ

Основной сценарий описывается в виде:

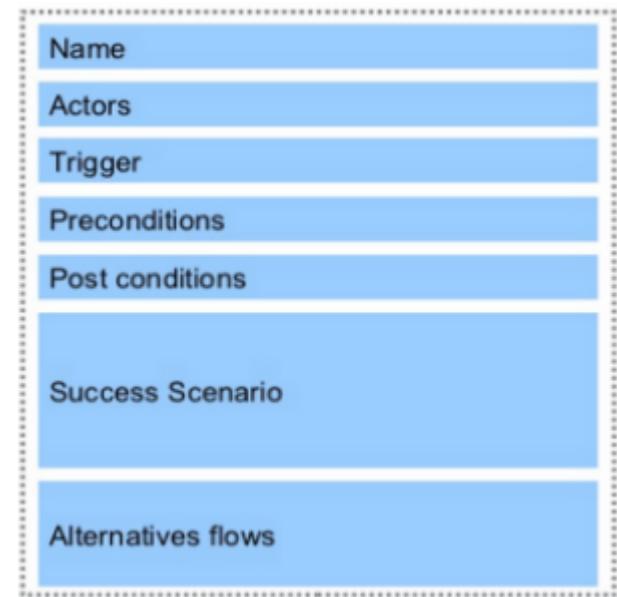
[Номер шага]. [Действие]

Расширения привязываются к определенным шагам основного сценария и представлены в виде:

**[Номер шага+идентификатор расширения]. [условие]:
[Действие или вложенный вариант использования]**

СПЕЦИФИКАЦИИ

- 1. Название**
- 2. Основные актеры**
- 3. Триггер (с чего начинается ВИ?)**
- 4. Предусловия**
- 5. Постусловия**
- 6. Область действия**
- 7. Уровень цели**
- 8. Минимальные гарантии**
- 9. Контекст использования**



Вариант использования:	Выполнить вход в систему
Контекст использования:	Пользователь совершает вход в систему
Область:	Система
Уровень:	Цель пользователя
Основной актер:	Пользователь
Предусловие:	Нет
Успешное постусловие:	Пользователю предоставлен доступ в систему
Минимальные гарантии:	Пользователю не предоставлен доступ в систему
Триггер:	Окно входа в систему

Основной сценарий

1. Пользователь вводит логин и пароль
2. Пользователь запускает проверку
3. Система проверяет логин
4. Система проверяет пароль
5. Система предоставляет пользователю доступ

Расширения

3.а. Не найдена учетная запись с таким логином:

- 3.а.1. Система уведомляет об ошибке
- 3.а.2. Возврат сценария на пункт 1

4.а. Пароль не верный:

- 4.а.1. Система увеличивает счетчик неудачных попыток входа.
- 4.а.2. Система проверяет количество неудачных попыток входа

4.а.1.а. Количество неудачных попыток больше установленного предела:

- 4.а.1.а.1. Система уведомляет о блокировке
- 4.а.1.а.2. Завершение сценария

- 4.а.3. Система уведомляет об ошибке
- 4.а.4. Возврат сценария на пункт 1

Пример спецификации варианта использования по Коберну

Ребекка Вирфс-Брок (Rebecca Wirfs-Brock)

American software engineer and consultant in object-oriented programming and object-oriented design, the founder of the information technology consulting firm Wirfs-Brock Associates, and inventor of Responsibility-Driven Design, the first behavioral approach to object design.



Ребеккой Вирфс-Брок было предложено оформлять сценарии варианта использования в виде диалога между основным актером и системой (не предполагается участие в одном варианте использования более двух актеров). Сценарий записывается в форме таблицы, состоящей из двух колонок:

Действия основного актера

Действия системы

Ребекка Вирфс-Брок

Двухколоночная таблица Вирфс-Брок. Пример описания основного сценария варианта использования «Выполнить вход в систему»

Действия пользователя

Действия системы

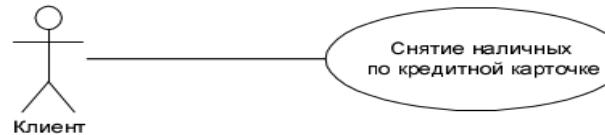
1. Вводит логин и пароль
2. Запускаем проверку

3. Проверяет логин
 4. Проверяет пароль
 5. Предоставляет доступ
-

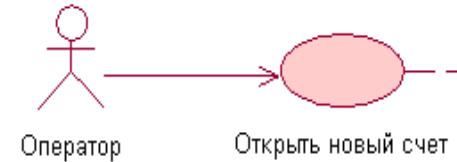
Пример описания ВИ по Вирфс-Брок

СВЯЗИ НА ДИАГРАММАХ ВИ

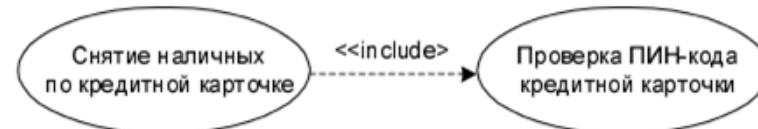
- Ассоциация



- Направленная ассоциация



- Отношение включения (include)

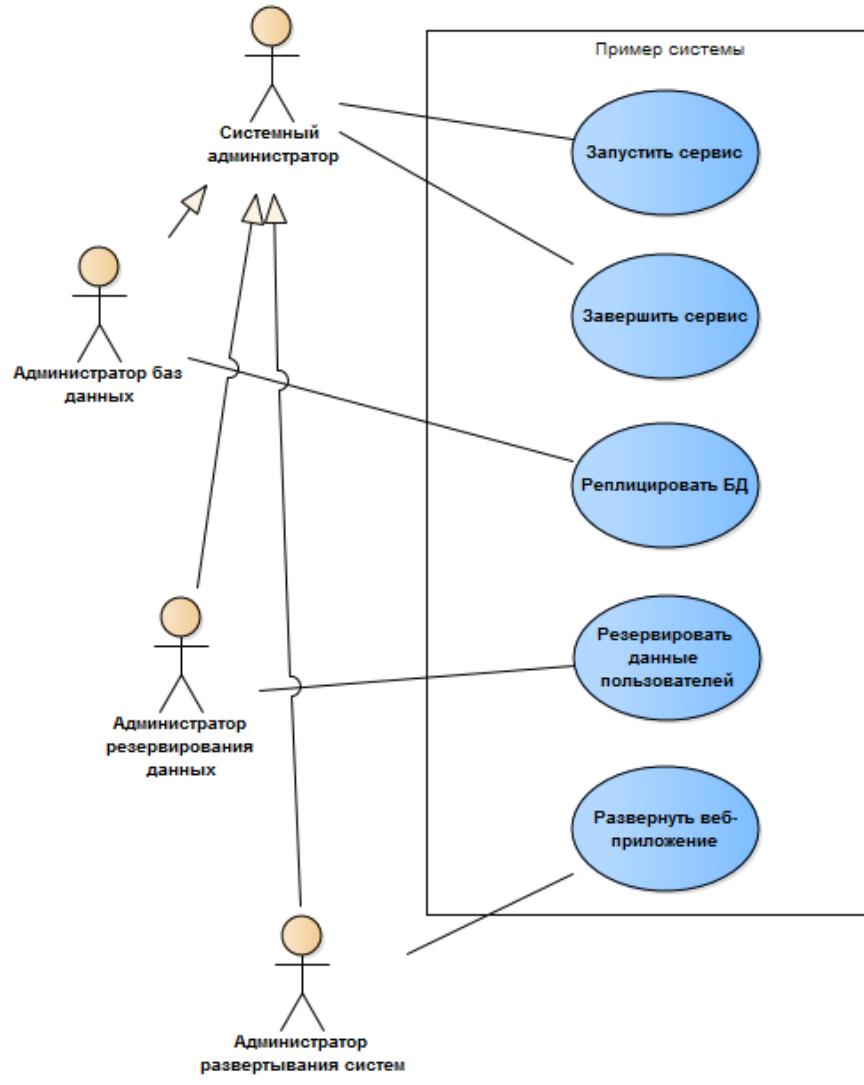


- Отношение расширения (extend)



- Отношение обобщения





Пример обобщения актеров

ПРИМЕЧАНИЯ (NOTES) НА ДИАГРАММАХ ВИ

- предназначены для включения в модель произвольной текстовой информации, имеющей непосредственное отношение к контексту разрабатываемого проекта. В качестве такой информации могут быть комментарии разработчика (например, дата и версия разработки диаграммы или ее отдельных компонентов), ограничения (например, на значения отдельных связей или экземпляры сущностей) и помеченные значения. Применительно к диаграммам вариантов использования примечание может носить самую общую информацию, относящуюся к общему контексту системы.
- Графически примечания обозначаются прямоугольником с "загнутым" верхним правым углом . Внутри прямоугольника содержится текст примечания. Примечание может относиться к любому элементу диаграммы, в этом случае их соединяет пунктирная линия. Если примечание относится к нескольким элементам, то от него проводятся, соответственно, несколько линий.



СПАСИБО ЗА ВНИМАНИЕ!

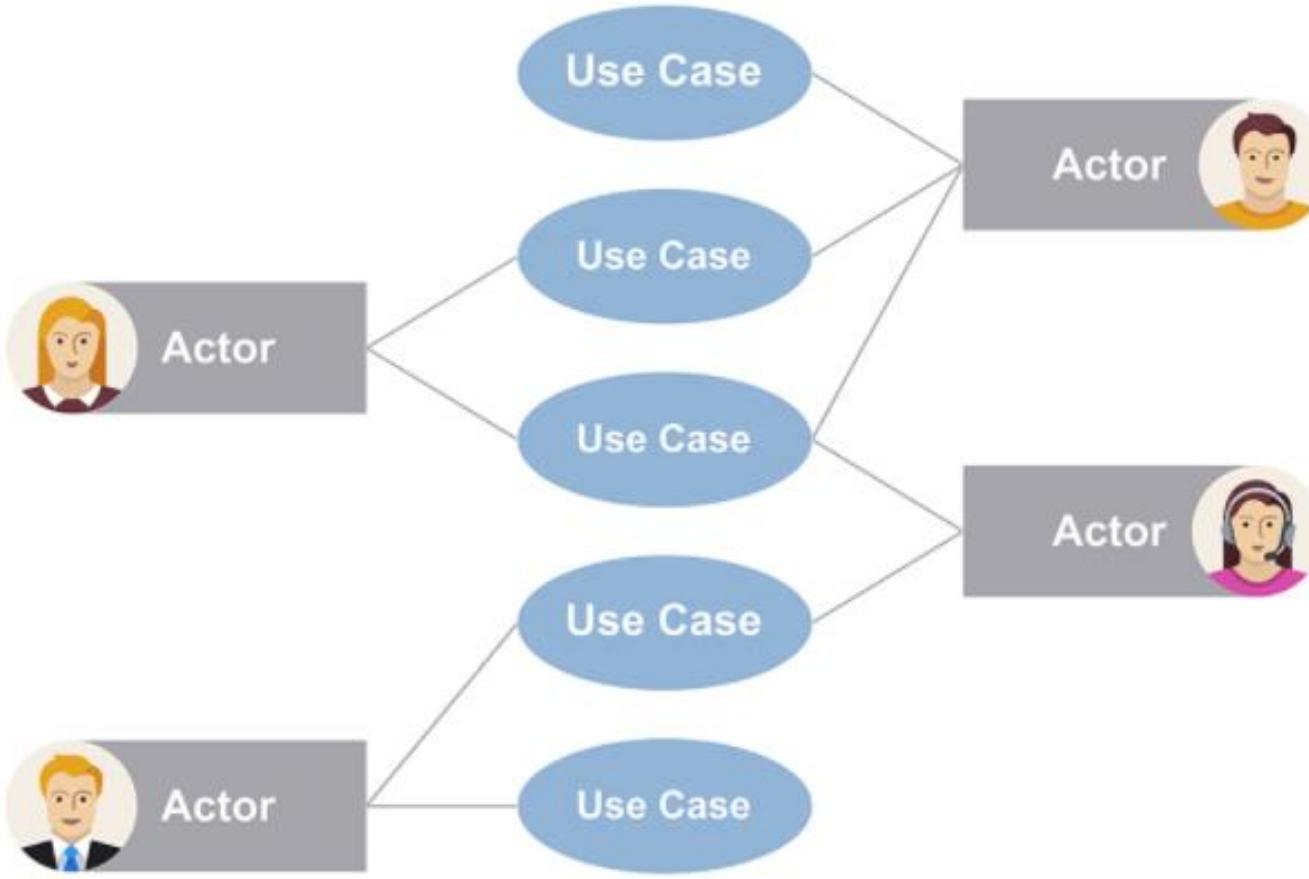
ПРОЕКТИРОВАНИЕ ПРОГРАММНЫХ СИСТЕМ

УНИФИЦИРОВАННЫЙ ЯЗЫК МОДЕЛИРОВАНИЯ UML

ЛЕКЦИЯ 3

ИНСТРУМЕНТЫ ДЛЯ РАЗРАБОТКИ НА UML

- Онлайн инструмент
- Отдельные программы
- Плагины
- Специальные IDE



В use-case диаграмме показывается, что делает система, теперь нужно определить, как это делается. Это обычно называется **реализацией вариантов использования**.

РЕАЛИЗАЦИЯ ВАРИАНТОВ ИСПОЛЬЗОВАНИЯ

Вариант использования – это описание множества последовательностей событий или действий (сценариев), доставляющих значимый для действующего лица результат.

- **Текстовые описания**
- **Реализация программой на псевдокоде**
- **Реализация диаграммами деятельности**
- **Реализация диаграммами взаимодействия**

ТЕКСТОВЫЕ ОПИСАНИЯ

Исторически самый заслуженный и до сих пор один из самых популярных способов: составить текстовое описание типичного сценария варианта использования. Рассмотрим следующий ниже текст в качестве примера.

Увольнение по собственному желанию

1. Сотрудник пишет заявление
2. Начальник подписывает заявление
3. Если есть неиспользованный отпуск, то бухгалтерия рассчитывает компенсацию
4. Бухгалтерия рассчитывает выходное пособие
5. Системный администратор удаляет учетную запись
6. Менеджер штатного расписания обновляет базу данных

РЕАЛИЗАЦИЯ ПРОГРАММОЙ НА ПСЕВДОКОДЕ

**Этот способ хорош тем, что понятен, привычен
и доступен любому.**

Use case Pay Compensation

if (add_payment)

 if (from Self Fire)

 начислить за неиспользованный

отпуск

 else

 if (from Adm Fire)

 начислить выходное пособие

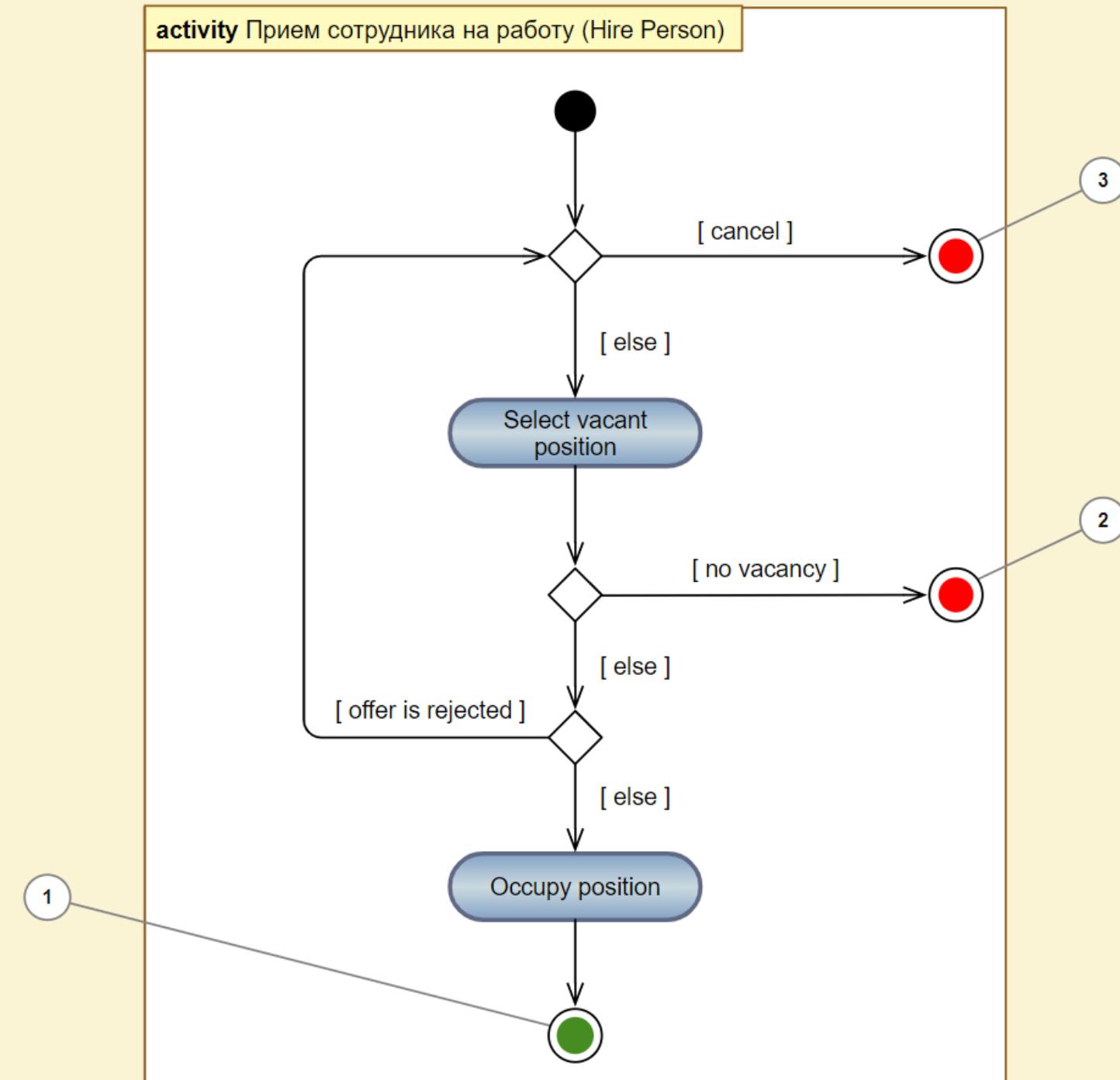
РЕАЛИЗАЦИЯ ПРОГРАММОЙ НА ПСЕВДОКОДЕ

Реализация на псевдокоде плохо согласуется с современной парадигмой объектно-ориентированного программирования.

**При использовании псевдокода теряются все преимущества использования UML:
наглядная визуализация с помощью картинки,
строгость и точность языка проектирования
и реализации, поддержка распространенными
инструментальными средствами.**

Решения на псевдокоде практически невозможно использовать повторно

РЕАЛИЗАЦИЯ ДИАГРАММАМИ ДЕЯТЕЛЬНОСТИ



РЕАЛИЗАЦИЯ ДИАГРАММАМИ ДЕЯТЕЛЬНОСТИ

Применение диаграмм деятельности для реализации вариантов использования не слишком приближает к появлению целевого артефакта – программного кода, однако может привести к более глубокому пониманию существа задачи и даже открыть неожиданные возможности улучшения приложения, которые было трудно усмотреть в первоначальной постановке задачи.

КЛАССИФИКАЦИЯ ДИАГРАММ

Диаграммы поведения:

- Диаграммы вариантов использования
- **Диаграммы деятельности**
- Диаграммы состояний
- **Диаграммы взаимодействия:**

- Диаграммы последовательности
- Диаграммы коммуникации
- Диаграммы синхронизации
- Обзорные диаграммы взаимодействия

Структурные диаграммы:

1. Диаграммы классов
2. Диаграммы объектов
3. Диаграммы пакетов
4. Диаграммы компонентов
5. Диаграммы составной структуры
6. Диаграммы размещения
7. Диаграммы профиля

ДИАГРАММЫ ДЕЯТЕЛЬНОСТИ (DIAGRAM ACTIVITY)

Диаграммы деятельности – это технология, позволяющая описывать логику процедур, бизнес процессы и потоки работ.

ОСНОВНЫЕ ПОНЯТИЯ ДД

Узлы управления:

- **входной узел**
- **финальный узел**
- **узел разделения**
- **узел слияния**
- **узел разветвления**
- **узел объединения**
- **узел действия**
- **Поток управления**
- **поток объектов**

Узлы управления:

Входной узел

Финальный узел

Узел разветвления

Узлы действий

Поток управления

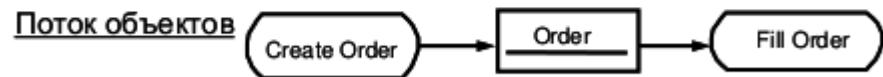
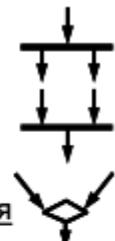
Поток объектов

Order o;
o = new Order;
FillOrder(o);

Узел разделения

Узел слияния

Узел объединения



ДИАГРАММЫ ДЕЯТЕЛЬНОСТИ (DIAGRAM ACTIVITY)

Диаграмма деятельности позволяет определить поведение с помощью последовательного исполнения поведений более низкого уровня. Исполнение следующего действия может начинаться в результате наступления одного из следующих событий:

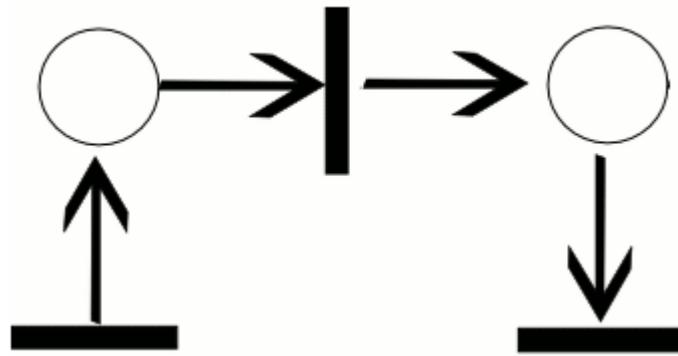
1. Завершение исполнения предыдущего действия
2. Появление необходимых данных
3. Наступление определенного события

ДИАГРАММА ДЕЯТЕЛЬНОСТИ

– это, фактически, старая добрая блок-схема алгоритма, в которой модернизированы обозначения, а семантика согласована с современным объектно-ориентированным подходом, что позволило органично включить диаграммы деятельности в UML.

На диаграмме деятельности применяют один основной тип сущностей – **деятельность**, и один тип отношений – **переходы** (передачи управления), а также графические обозначения (развилки, слияния и ветвления).

СЕТИ ПЕТРИ



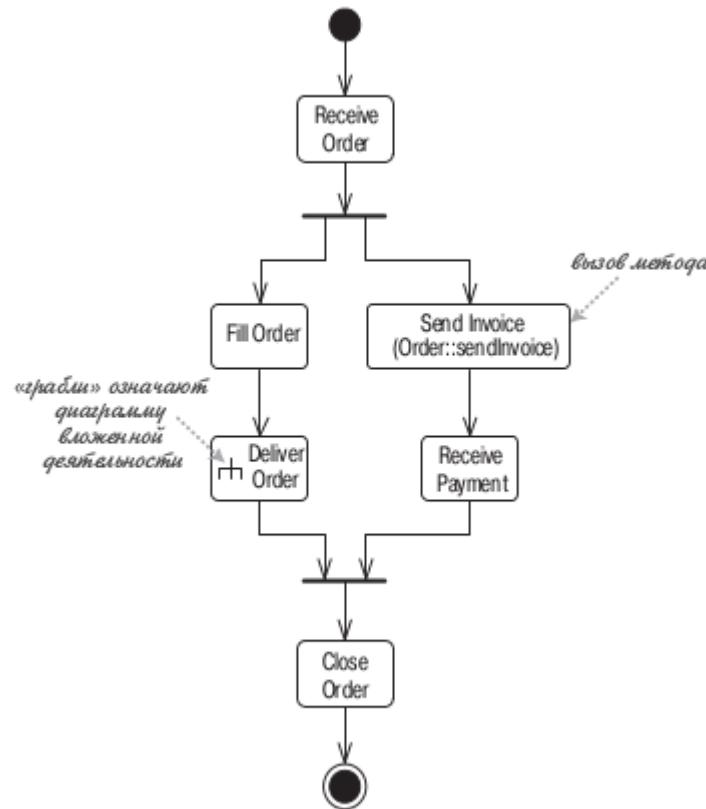
Сети Петри – математический аппарат для моделирования динамических дискретных систем. Впервые описаны Карлом Петри в 1962 году.

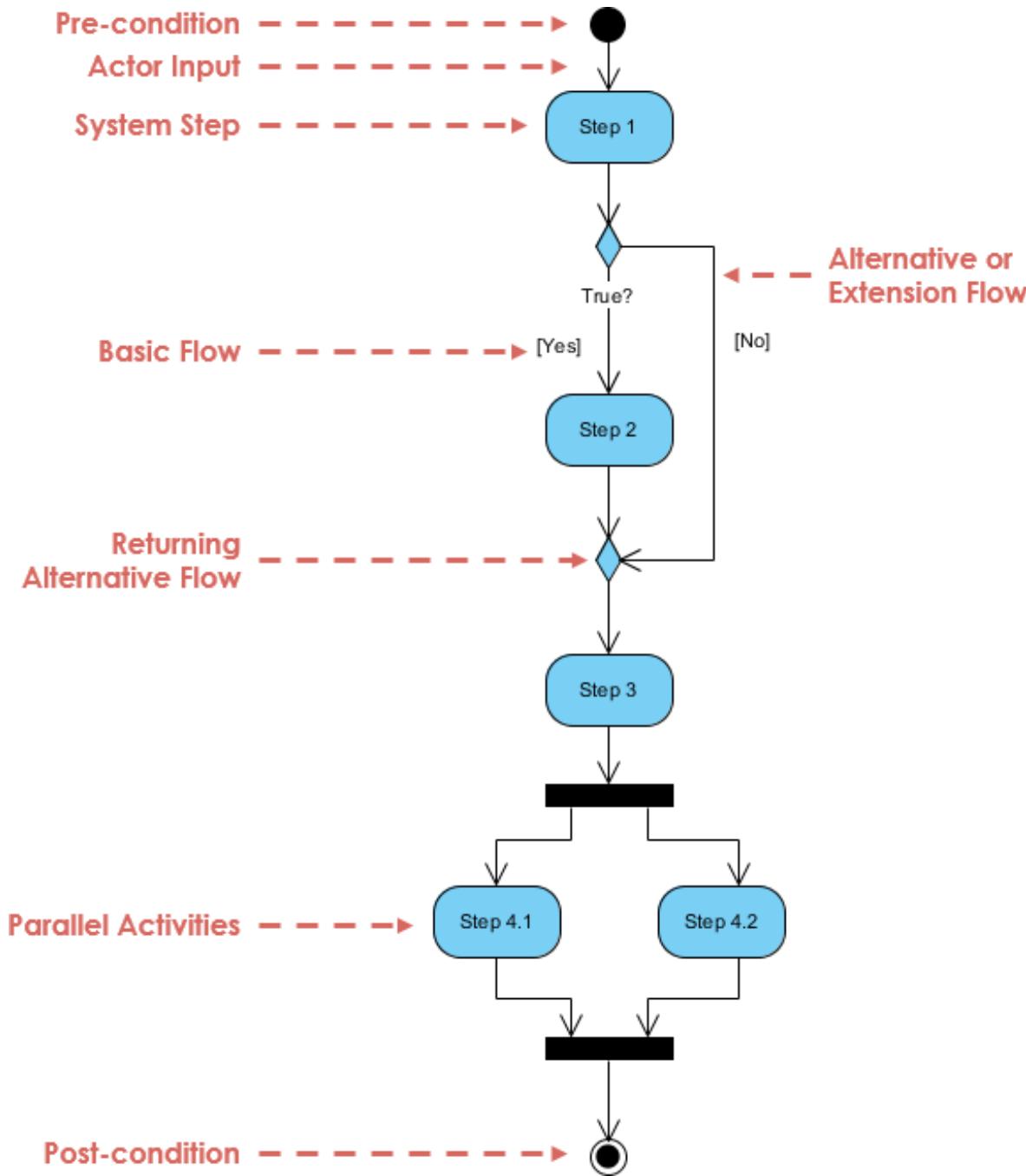
Сеть Петри представляет собой двудольный ориентированный граф, состоящий из вершин двух типов – позиций и переходов, соединённых между собой дугами. Вершины одного типа не могут быть соединены непосредственно. В позициях могут размещаться метки (маркеры), способные перемещаться по сети.

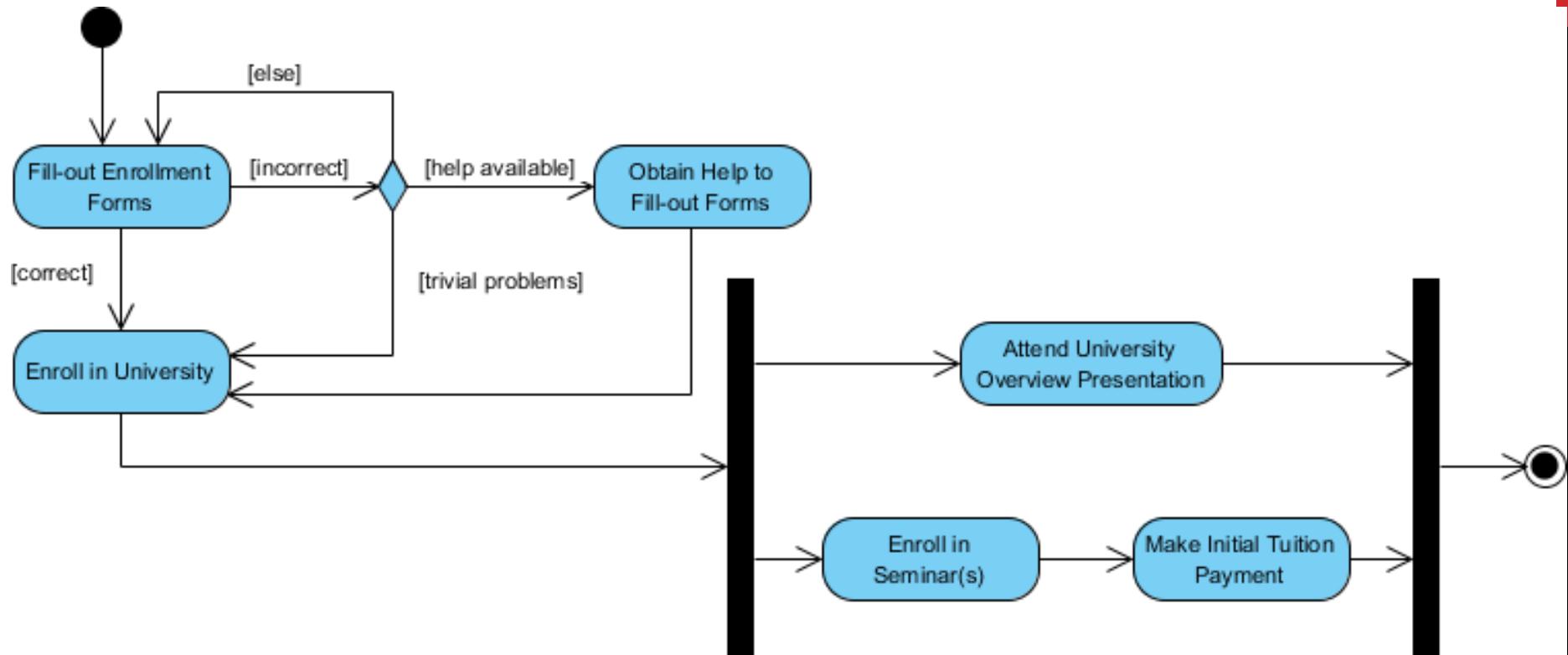
ДИАГРАММА ДЕЯТЕЛЬНОСТИ

- Описание одного процесса (алгоритма, активности, бизнес-процесса)
- Похожа на классическую блок-схему, но с некоторыми отличиями, дополнениями
- Показывает процесс в динамике, ход (текущие) работы в зависимости от условий
- Легче разрабатывать, если уже создана Use Case диаграмма
- Изображается в терминах действий с условиями (также имеется возможность указания объектов)

**Деятельности могут быть разбиты на вложенные
деятельности (subactivities) и определены как
самостоятельная деятельность:**

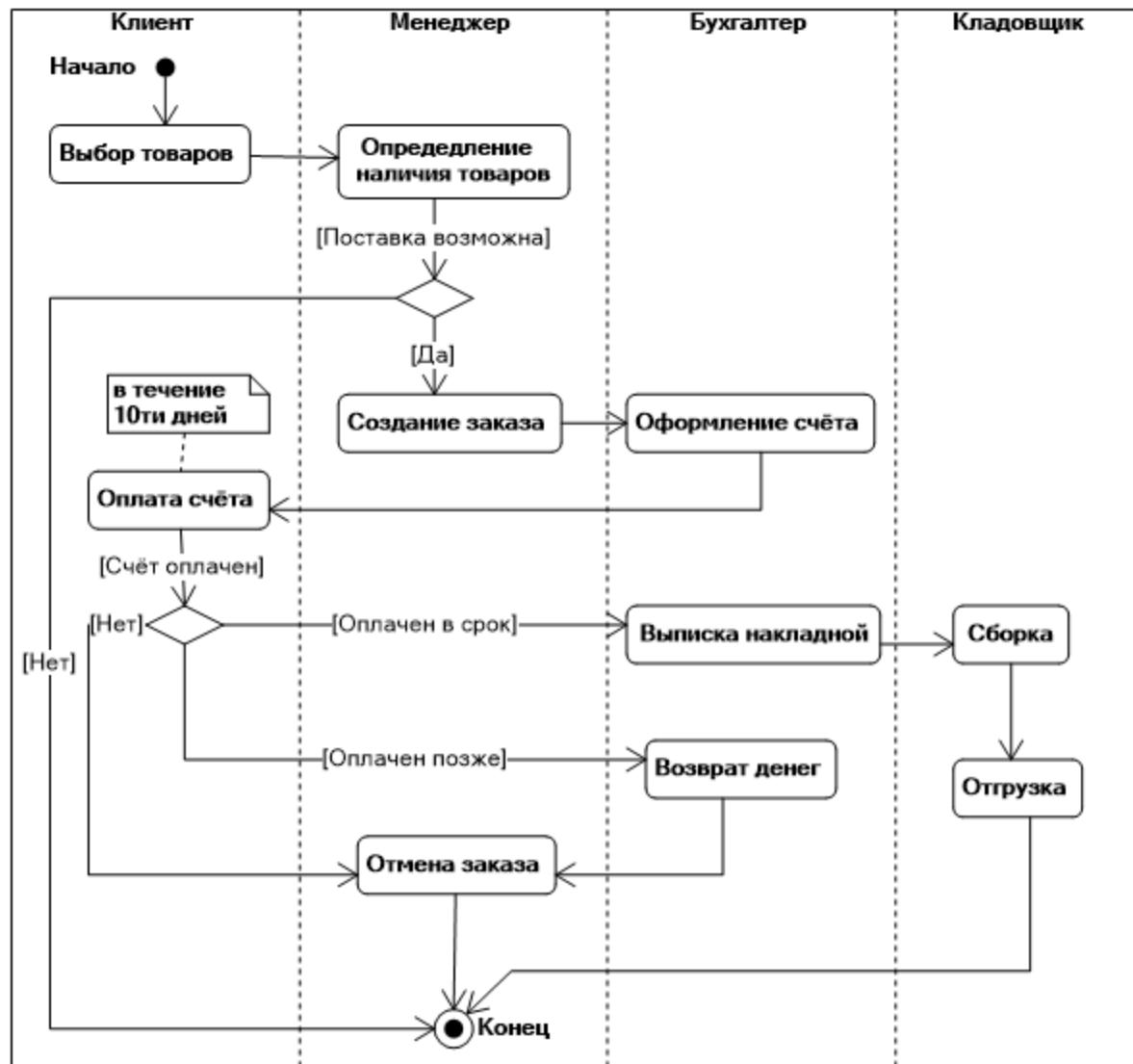






Кроме того, на диаграмме деятельности можно применить специальный графический комментарий – так называемые дорожки – подчеркивающие, что некоторые деятельности отличаются друг от друга, например, выполняются в разных местах.

Графически дорожки изображаются в виде прямоугольников с названиями.



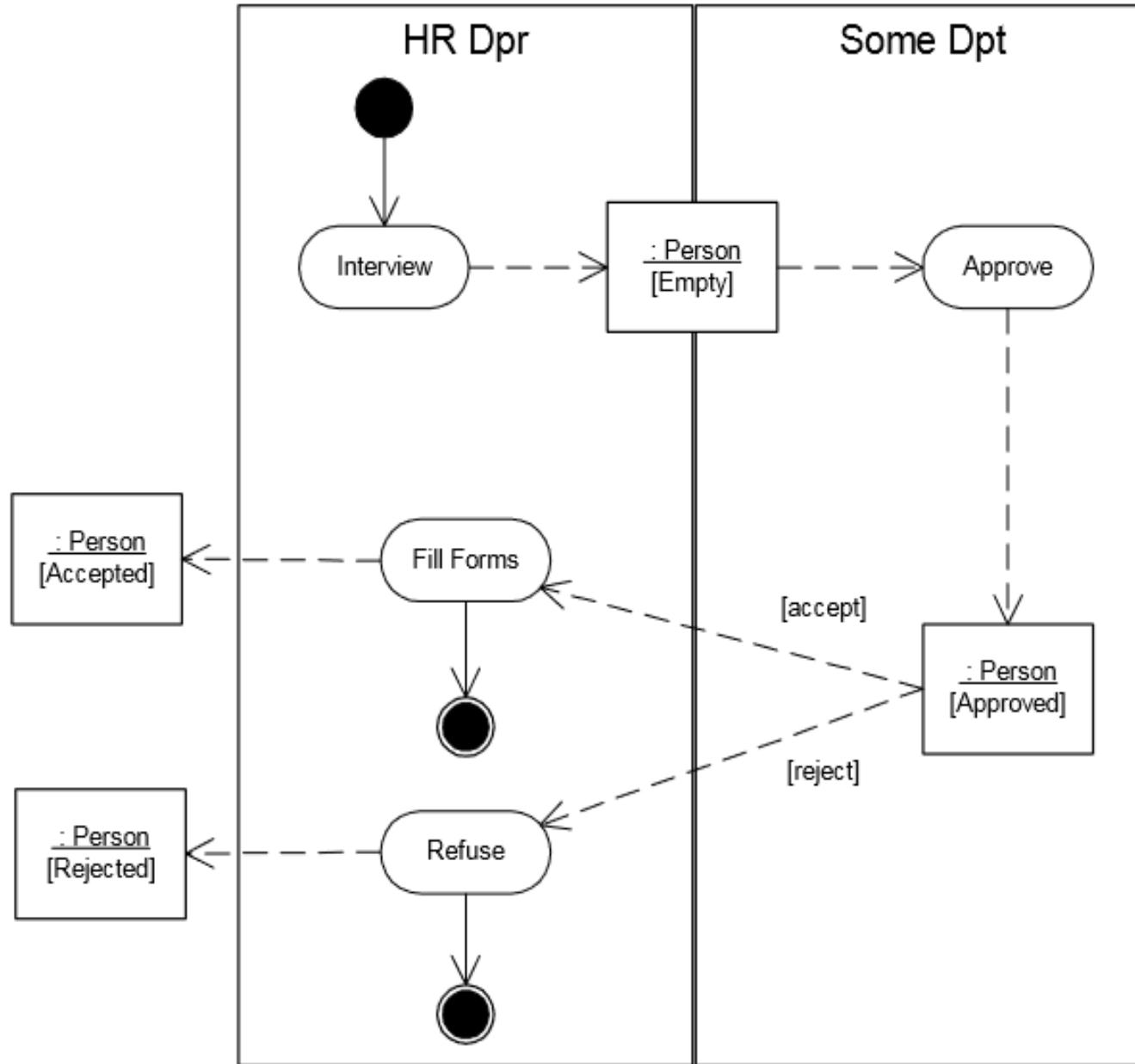
Помимо потока управления на диаграмме деятельности можно показать и поток данных, используя такую сущность, как объект (в определенном состоянии) и соответствующую зависимость.

Объект в состоянии – это объект некоторого класса, про который известно, что он находится в определенном состоянии в данной точке вычислительного процесса.

Синтаксически объект в состоянии изображается, как обычно, в виде прямоугольника и его имя подчеркивается, но дополнительно после имени объекта в квадратных скобках пишется имя состояния, в котором в данной точке вычислительного процесса находится объект. В некоторых случаях состояние объекта не важно, например, если достаточно указать, что в данной точке вычислительного процесса создается новый объект данного класса, и в этом случае применяется обычная нотация для изображения объектов. Важно подчеркнуть, что объект в состоянии на диаграммах деятельности "по определению" считается состоянием, т. е. вершиной графа модели, которая может быть инцидентна переходам, правда переходам особого рода.

Траектория объекта – это переход особого рода, исходным и/или целевым состоянием которого является объект в состоянии.

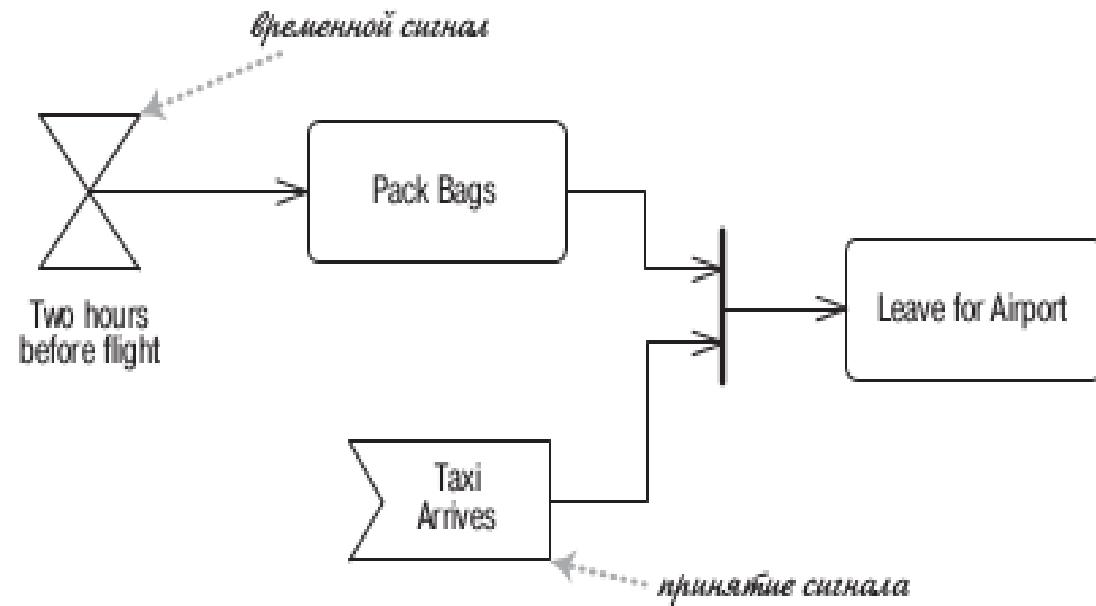
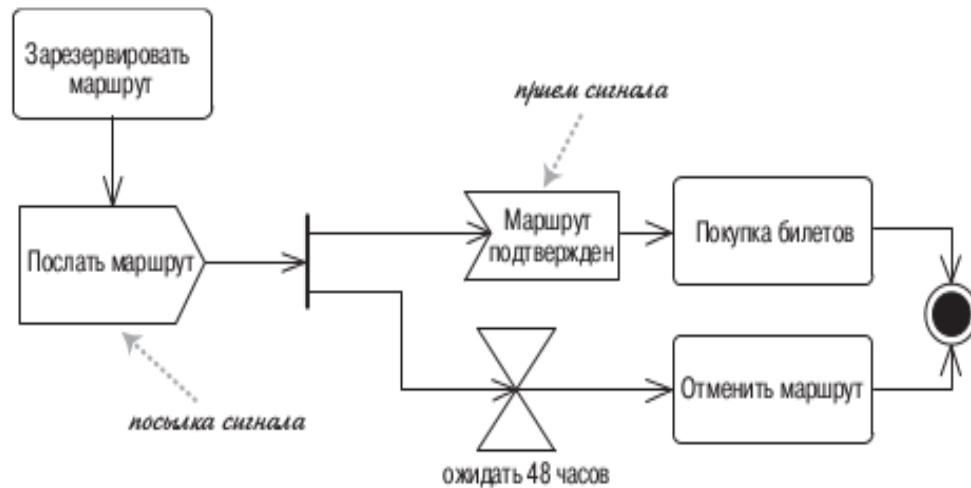
Траектория объекта изображается в виде пунктирной стрелки (в отличии от сплошной стрелки обычного перехода).



СИГНАЛЫ

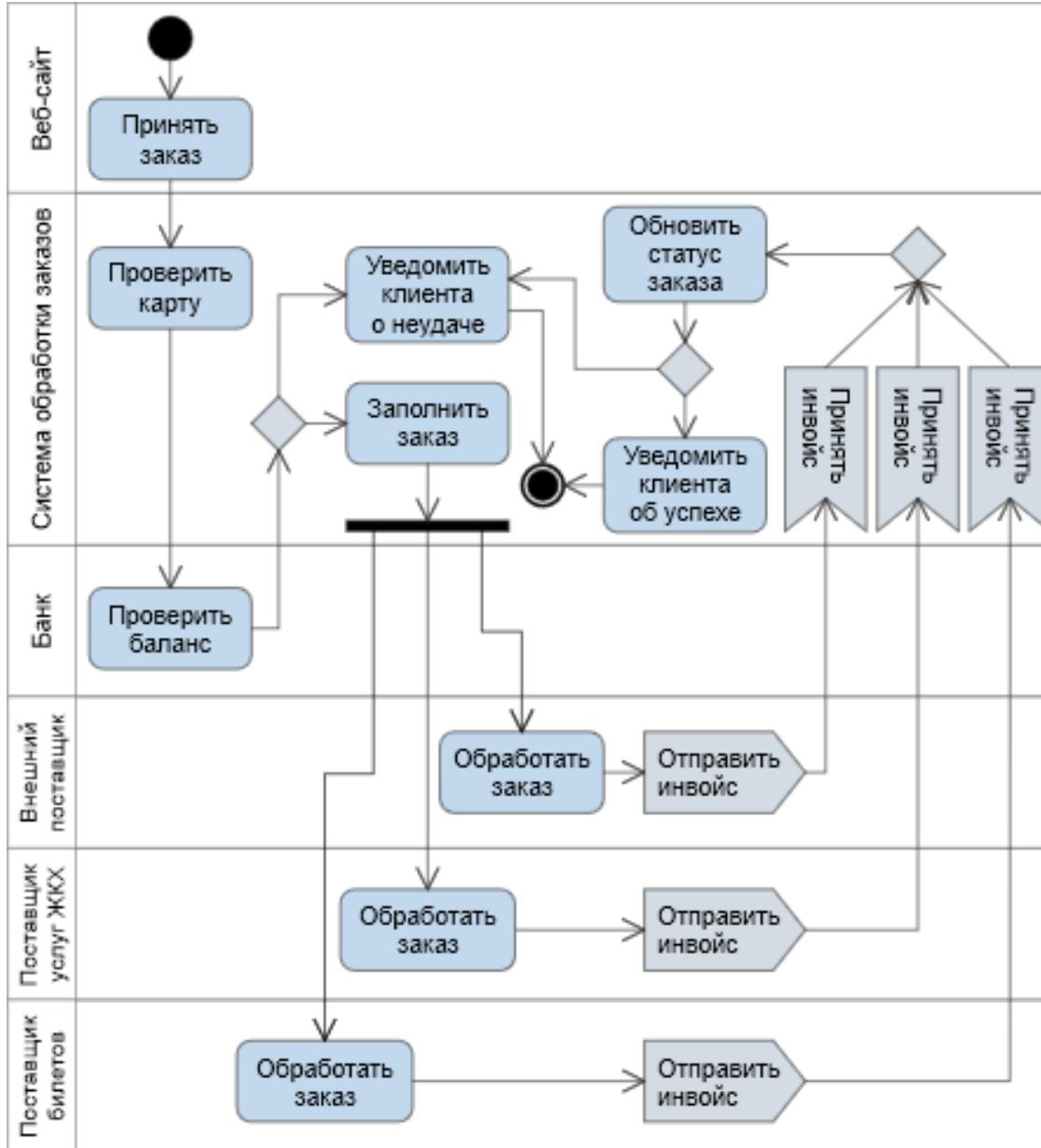
Диаграммы деятельности имеют четко определенную стартовую точку, соответствующую вызову программы или процедуры. Кроме того, операции могут отвечать на сигналы.

Временной сигнал (time signal) приходит по прошествии времени.

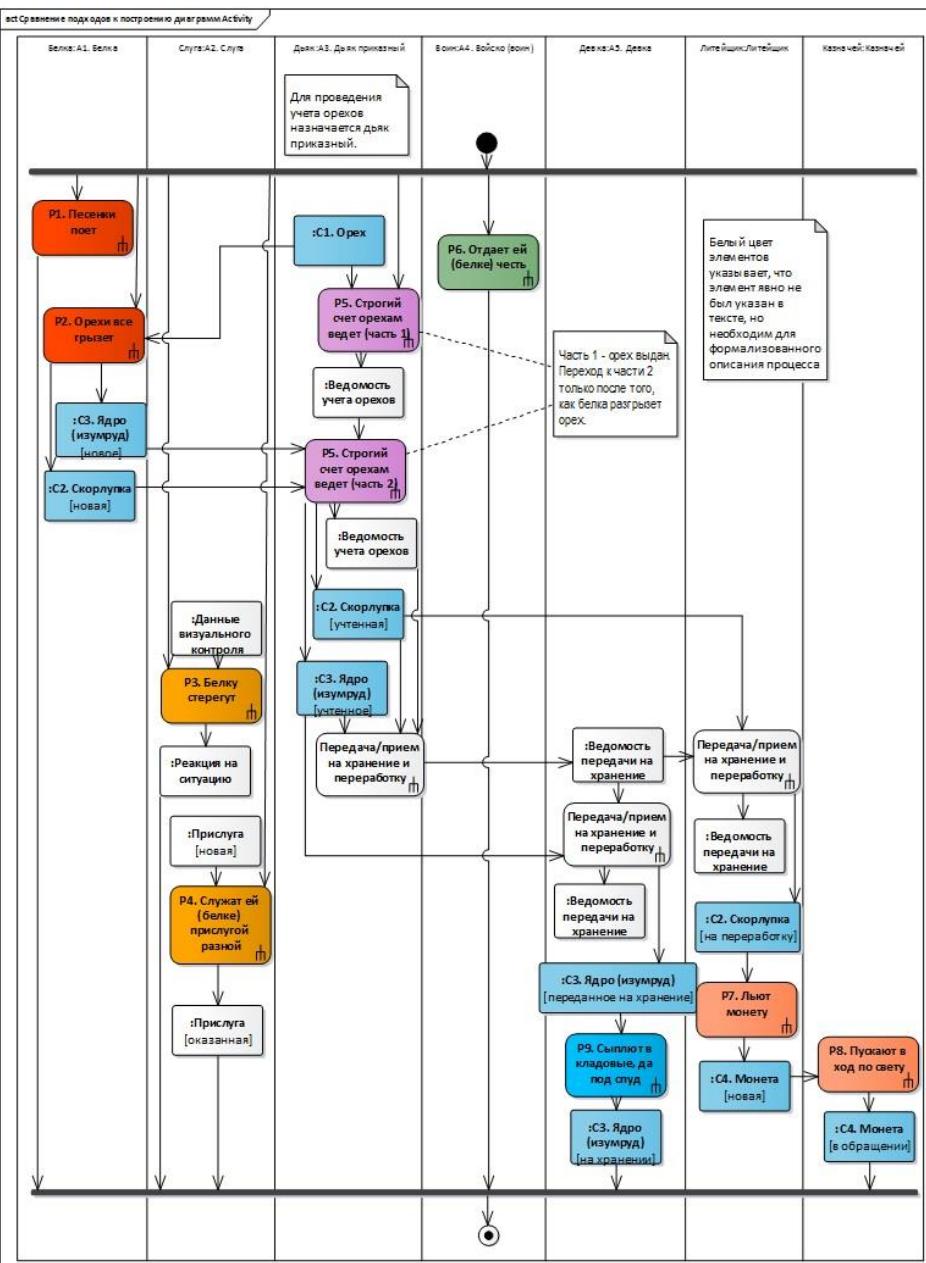




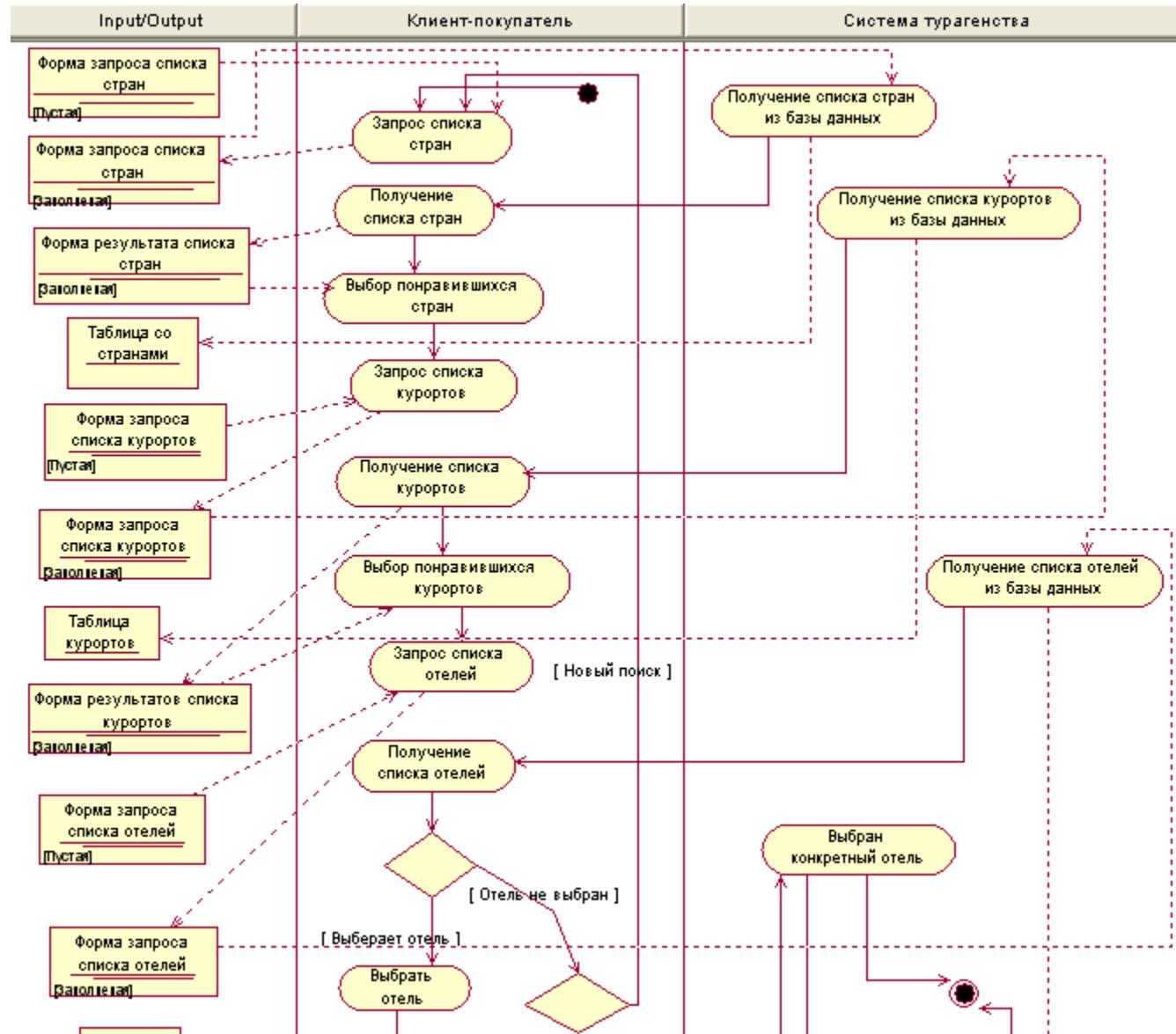
Диаграммы деятельности применяются для описания поведения на самом высоком уровне абстракции, наиболее удаленном от программной реализации и на самом низком уровне, практически на уровне программного кода.



Пример диаграммы деятельности

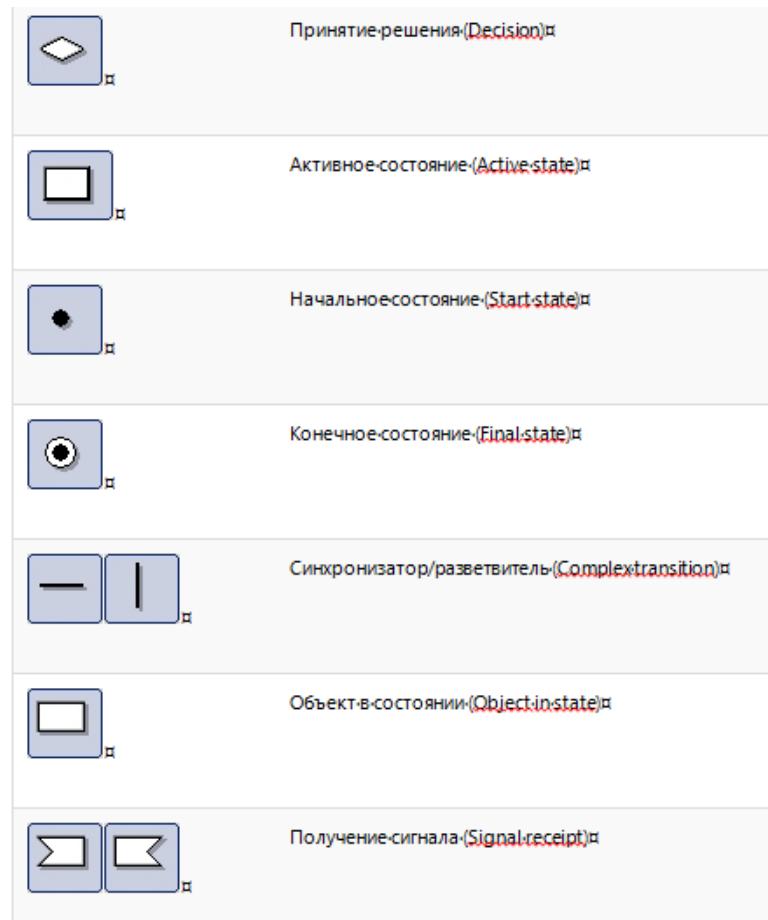


Пример диаграммы деятельности



Пример диаграммы деятельности

ОСНОВНЫЕ ЭЛЕМЕНТЫ ДИАГРАММЫ ДЕЯТЕЛЬНОСТИ



ОСНОВНЫЕ ЭЛЕМЕНТЫ ДИАГРАММЫ ДЕЯТЕЛЬНОСТИ

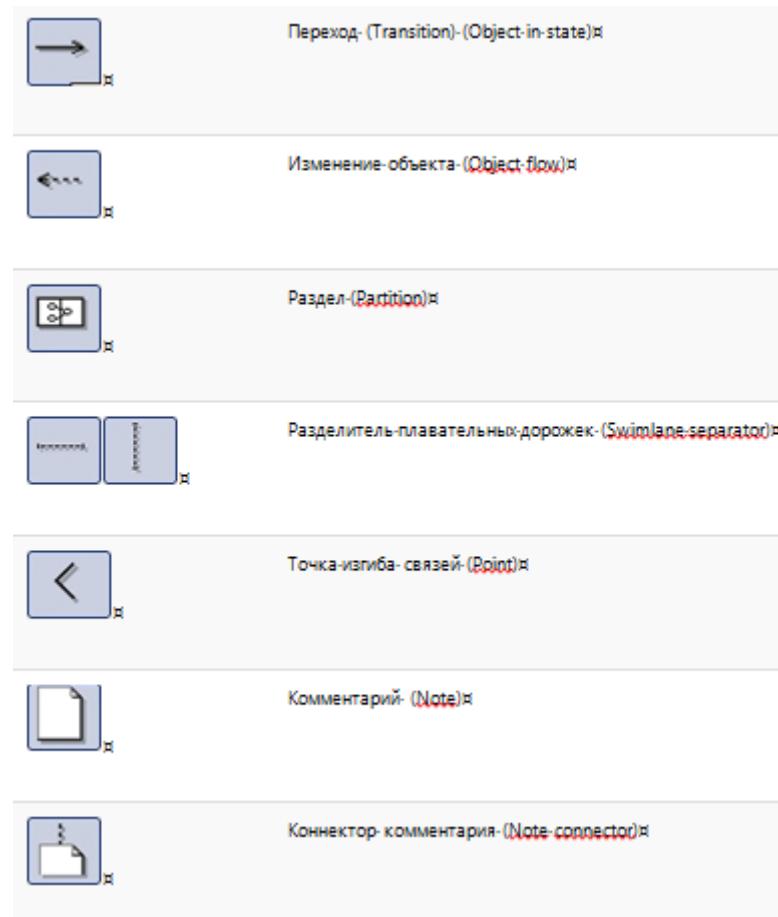


ДИАГРАММА СОСТОЯНИЙ (STATE MACHINE DIAGRAMS)

ДИАГРАММЫ СОСТОЯНИЙ

В основе конечных автоматов UML лежит работа Харела . С их помощью обычно моделируется предыстория жизненного цикла одного реактивного объекта. Она представляется в виде конечного автомата – автомата, который может существовать в конечном числе состояний. В ответ на события конечный автомат четко осуществляет переходы между этими состояниями.

Конечный автомат моделирует динамическое поведение реактивного объекта.

ДИАГРАММЫ СОСТОЯНИЙ

Для описания жизненного цикла конкретного объекта, поведение которого зависит от истории этого объекта, или же требует обработки асинхронных стимулов, используется конечный автомат в форме диаграммы состояний. При этом состояния конечного автомата соответствуют состояниям объекта, т.е. различным наборам значений атрибутов, а переходы соответствуют выполнению операций.

Выполнение конструктора объекта моделируется переходом из начального состояния, а выполнение деструктора — переходом в заключительное состояние. Диаграммы состояний можно составить не только для программных объектов — экземпляров отдельных классов, но и для более крупных конструкций, в частности, для всей модели приложения в целом или для более мелких — отдельных операций.

ДИАГРАММЫ СОСТОЯНИЙ

Конечные автоматы в UML реализованы довольно своеобразно. С одной стороны, в основу положено классическое представление автомата в форме графа состояний-переходов. С другой стороны, к классической форме добавлено большое число различных расширений и вспомогательных обозначений, которые, строго говоря, не обязательны – без них в принципе можно было бы обойтись – но весьма удобны и наглядны при составлении диаграмм.

На диаграммах состояний применяется всего один тип сущностей — состояния, и всего один тип отношений — переходы.

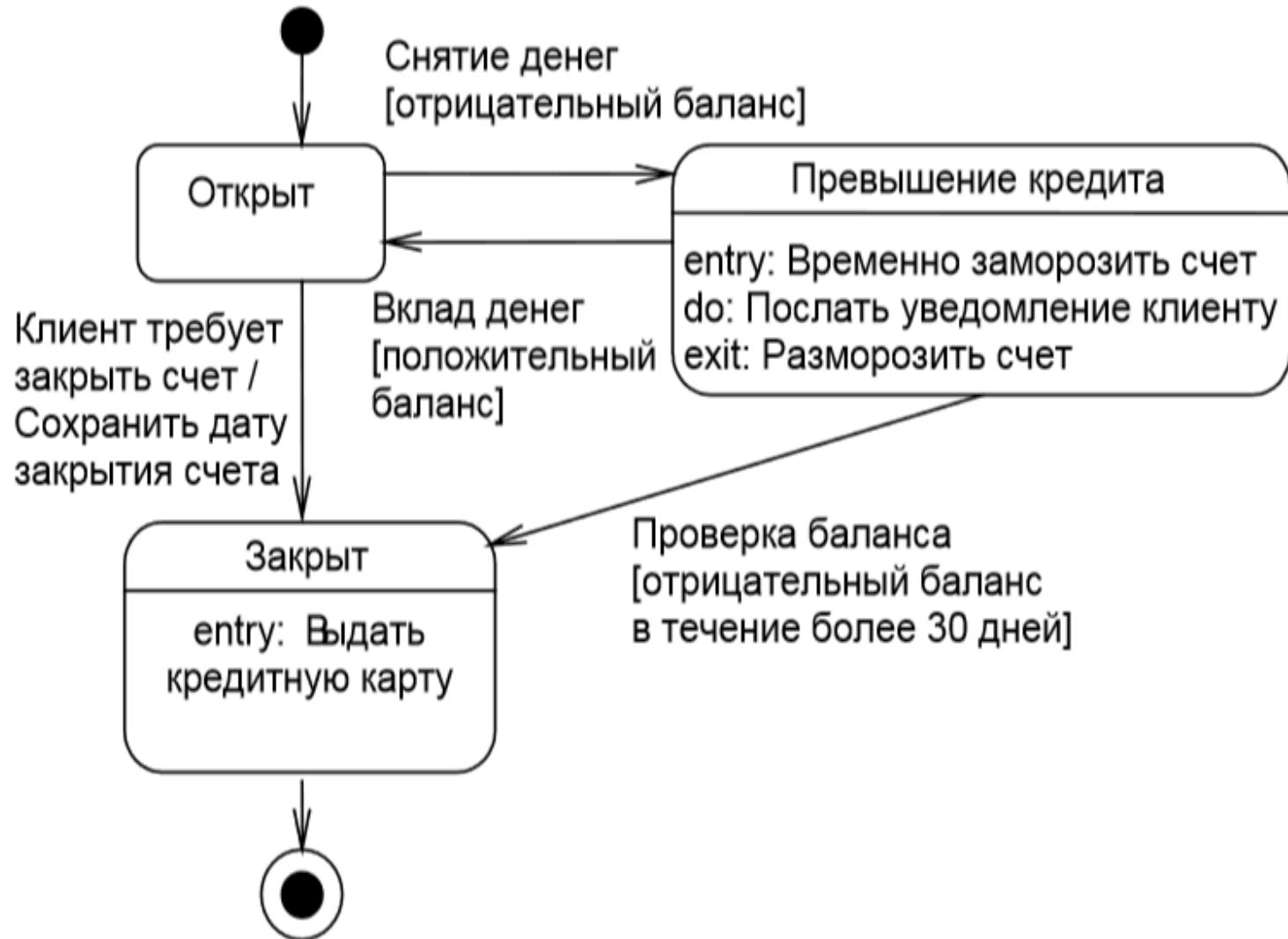
ДИАГРАММЫ СОСТОЯНИЙ

Три основных элемента автоматов – состояния, события и переходы:

- состояние (state) – «условие или ситуация в жизни объекта, при которых он удовлетворяет некоторому условию, осуществляет некоторую деятельность или ожидает некоторого события»
- событие (event) – «описание заслуживающего внимания происшествия, занимающего определенное положение во времени и пространстве»
- переход (transition) – переход из одного состояния в другое в ответ на событие.

СОСТОЯНИЯ

- Начальное состояние
- Финальное состояние
- Деятельность состояния
- Входное состояние
- Выходное состояние
- Внутренние переходы



Процессы, происходящие, когда объект находится в определенном состоянии, называются действиями (actions).

С состоянием можно связывать данные пяти типов:

- **деятельность,**
- **входное действие,**
- **выходное действие,**
- **событие и**
- **история состояния.**



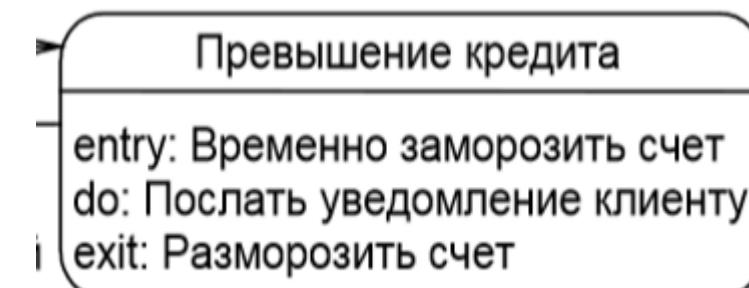
синтаксис действия: имяСобытия/некотороеДействие

синтаксис деятельности: do/ некотораяДеятельность

Синтаксис состояния

ДЕЯТЕЛЬНОСТЬ

Деятельностью (activity) называется поведение, реализуемое объектом, пока он находится в данном состоянии. Деятельность – это прерываемое поведение. Оно может выполняться до своего завершения, пока объект находится в данном состоянии, или может быть прервано переходом объекта в другое состояние. Деятельность изображают внутри самого состояния, ей должно предшествовать слово do (делать) и двоеточие.



ВХОДНОЕ ДЕЙСТВИЕ

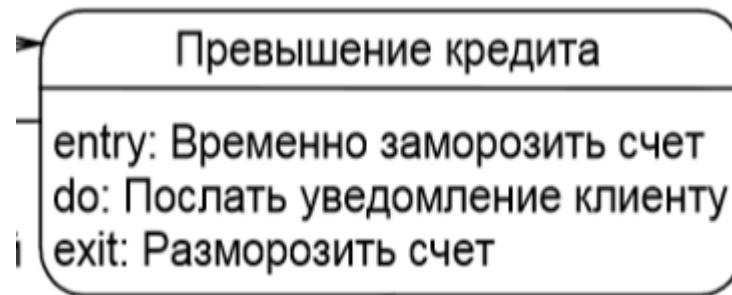
Входным действием (entry action) называется поведение, которое выполняется, когда объект переходит в данное состояние. В примере счета в банке, когда он переходит в состояние «Превышен счет», выполняется действие «Временно заморозить счет», независимо от того, откуда объект перешел в это состояние. Таким образом, данное действие осуществляется не после того, как объект перешел в это состояние, а, скорее, как часть этого перехода. В отличие от деятельности, входное действие рассматривается как непрерываемое. Входное действие также показывают внутри состояния. ему предшествует слово entry (вход) и двоеточие.

Превышение кредита

entry: Временно заморозить счет
do: Послать уведомление клиенту
exit: Разморозить счет

ВЫХОДНОЕ ДЕЙСТВИЕ

Выходное действие (exit action) подобно входному. Однако, оно осуществляется как составная часть процесса выхода из данного состояния. В нашем примере при выходе объекта Account из состояния «Превышен счет», независимо от того, куда он переходит, выполняется действие «Разморозить счет». Оно является частью процесса такого перехода. Как и входное, выходное действие является непрерываемым. Выходное действие изображают внутри состояния, ему предшествует слово exit (выход) и двоеточие.



DO: ^ЦЕЛЬ.СОБЫТИЕ(АРГУМЕНТЫ)

Поведение объекта во время деятельности, при входных и выходных действиях может включать отправку события другому объекту. Например, объект account (счет) может посыпать событие объекту card reader (устройство чтения карты). В этом случае описанию деятельности, входного действия или выходного действия предшествует знак « ^ ». Соответствующая строка на диаграмме выглядит как

Do: ^Цель.Событие(Аргументы)

Здесь Цель – это объект, получающий событие, Событие – это посыпаемое сообщение, а Аргументы являются параметрами посыпаемого сообщения.

ПЕРЕХОД

Переходом (Transition) называется перемещение из одного состояния в другое. Совокупность переходов диаграммы показывает, как объект может перемещаться между своими состояниями. На диаграмме все переходы изображают в виде стрелки, начинающейся на первоначальном состоянии и заканчивающейся последующим. Переходы могут быть рефлексивными. Объект может перейти в то же состояние, в котором он в настоящий момент находится. Рефлексивные переходы изображают в виде стрелки, начинающейся и завершающейся на одном и том же состоянии.

У перехода существует несколько спецификаций. Они включают события, аргументы, ограждающие условия, действия и посылаемые события.

СОБЫТИЯ

Событие (event) – это то, что вызывает переход из одного состояния в другое. В нашем примере событие «Клиент требует закрыть» вызывает переход счета из открытого в закрытое состояние.

Событие размещают на диаграмме вдоль линии перехода. На диаграмме для отображения события можно использовать как имя операции, так и обычную фразу. В нашем примере события описаны обычными фразами.

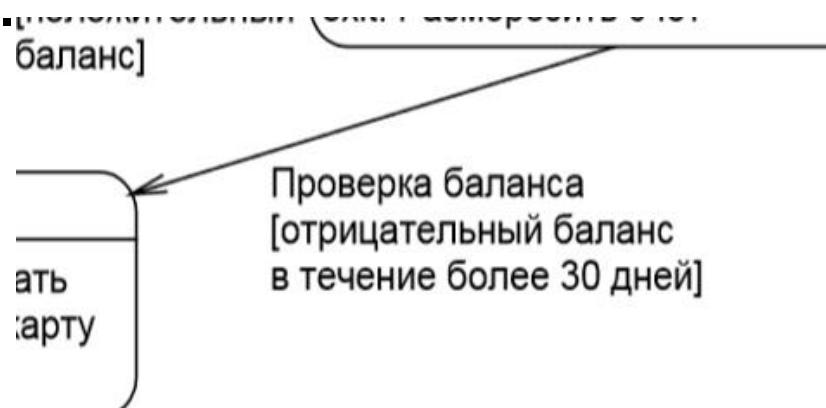
Большинство переходов должны иметь события, так как именно они, прежде всего, заставляют переход осуществиться. Тем не менее, бывают и автоматические переходы, не имеющие событий. При этом объект сам перемещается из одного состояния в другое со скоростью, позволяющей осуществляться входным действиям, деятельности и выходным действиям.



ОГРАЖДАЮЩИЕ УСЛОВИЯ

Ограждающие условия (guard conditions) определяют, когда переход может, а когда не может осуществиться. Ограждающие условия изображают на диаграмме вдоль линии перехода после имени события, заключая их в квадратные скобки.

Ограждающие условия задавать необязательно. Однако если существует несколько автоматических переходов из состояния, необходимо определить для них взаимно исключающие ограждающие условия. Это поможет читателю диаграммы понять, какой путь перехода будет автоматически выбран.



ДЕЙСТВИЕ

Действием (action) является непрерывное поведение, осуществляющееся как часть перехода.

Входные и выходные действия показывают внутри состояний, поскольку они определяют, что происходит, когда объект входит или выходит из него. Большую часть действий, однако, изображают вдоль линии перехода, так как они не должны осуществляться при входе или выходе из состояния.

Действие рисуют вдоль линии перехода после имени события, ему предшествует косая черта. Событие или действие могут быть поведением внутри объекта, а могут представлять собой сообщение, посыпаемое другому объекту. Если событие или действие посыпается другому объекту, перед ним на диаграмме помещают знак « ^ ».



СОСТОЯНИЯ И ПЕРЕХОДЫ

Состояния бывают:

- простые,
- составные,
- специальные
- и каждый тип состояний имеет дополнительные подтипы и различные составляющие элементы.

Переходы бывают *простые и составные*, и каждый переход содержит от двух до пяти составляющих:

- исходное состояние,
- событие перехода,
- сторожевое условие,
- действие на переходе,
- целевое состояние.

СОБЫТИЕ ПЕРЕХОДА

Событие перехода – это тот входной стимул, который вкупе с текущим состоянием автомата определяет следующее состояние.

В UML используются четыре типа событий:

- событие вызова,
- событие сигнала,
- событие таймера,
- событие изменения.

Составное состояние может быть

- последовательным или
- параллельным (ортогональным).

Специальные состояния, в свою очередь, бывают следующих типов:

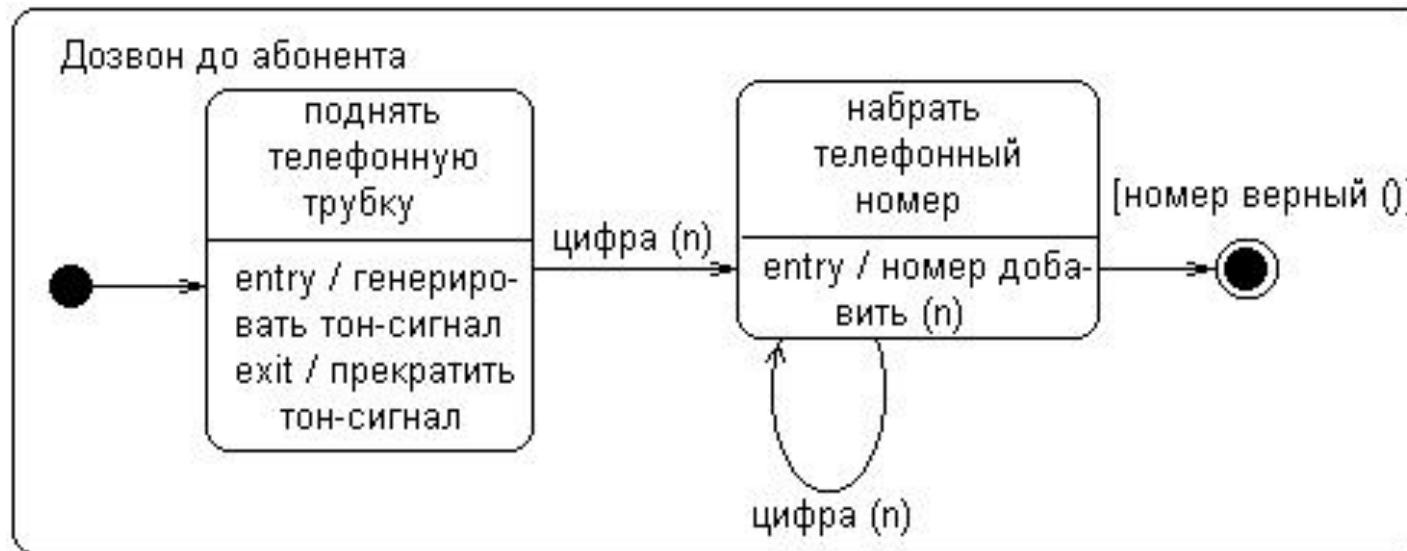
- начальное состояние,
- заключительное состояние,
- переходное состояние,
- историческое состояние,
- синхронизирующее состояние,
- ссылочное состояние,
- состояние "заглушка".

СОСТАВНОЕ СОСТОЯНИЕ

Составное состояние – это состояние, в которое вложена машина состояний. Если вложена только одна машина, то состояние называется последовательным, если несколько – параллельным. Глубина вложенности в UML неограничена, т.е. состояния вложенной машины состояний также могут быть составными.



СОСТАВНОЕ СОСТОЯНИЕ



Пример составного состояния с двумя вложенными последовательными подсостояниями.

ИСТОРИЧЕСКОЕ СОСТОЯНИЕ

Историческое состояние – это специальное состояние, подобное начальному состоянию, но обладающее дополнительной семантикой.

Историческое состояние может использоваться во вложенной машине состояний внутри составного состояния.

Историческое состояние имеет две разновидности.

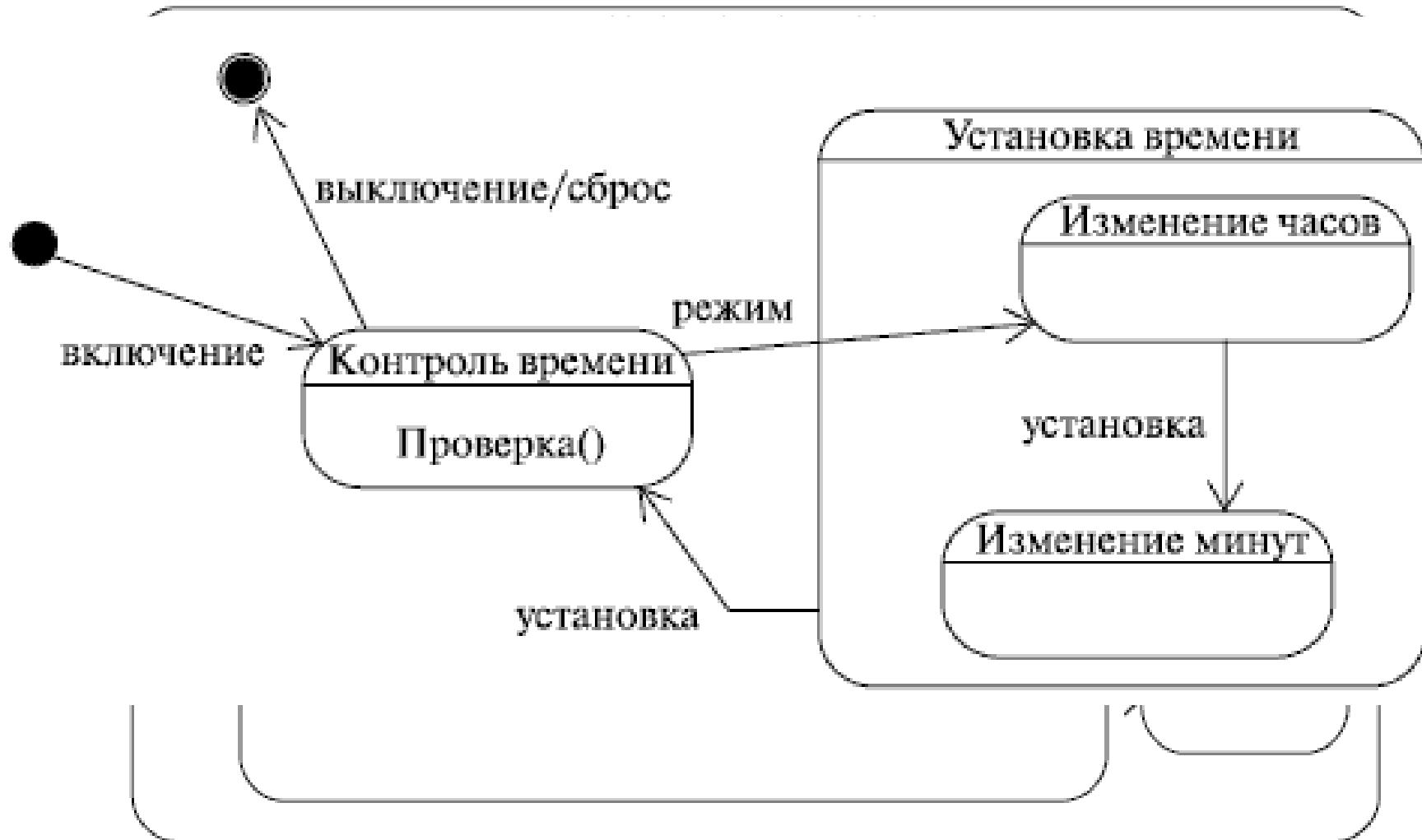
Поверхностное историческое состояние запоминает, какое состояние было активным на том же уровне вложенности, на каком находится само историческое состояние.

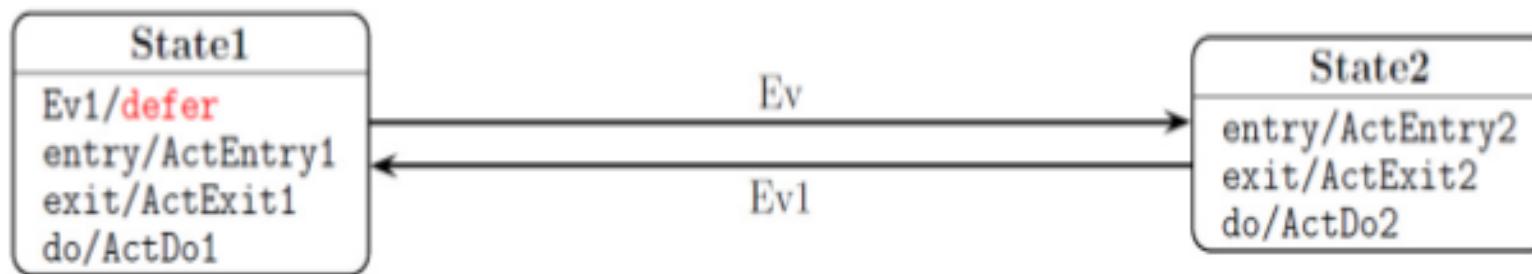
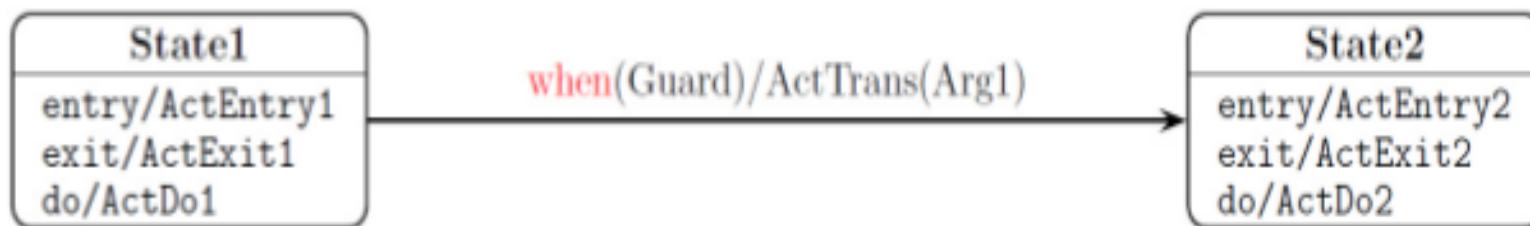
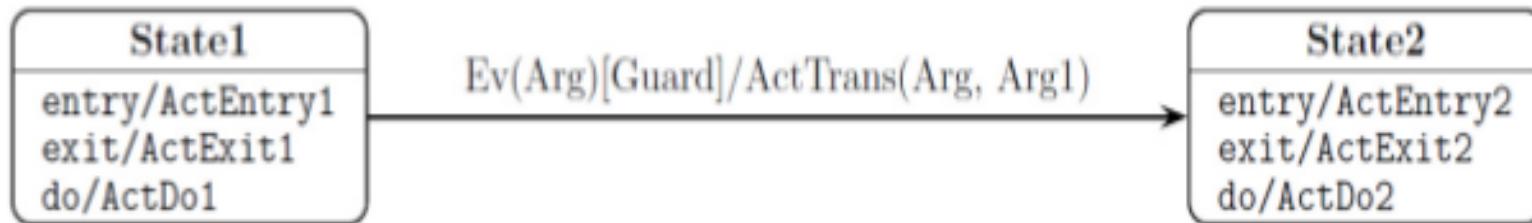
Глубинное историческое состояние помнит не только активное состояние на данном уровне, но и на вложенных уровнях.

ИСТОРИЧЕСКОЕ СОСТОЯНИЕ

При первом запуске машины состояний историческое состояние означает в точности тоже, что и начальное: оно указывает на состояние, в котором находится машина в начале работы. Если в данной машине состояний используется историческое состояние, то при выходе из объемлющего составного состояния запоминается то состояние, в котором находилась вложенная машина при выходе. При повторном входе в данное составное состояние в качестве текущего состояния восстанавливается то состояние, в котором машина находилась при выходе. Проще говоря, историческое состояние заставляет машину помнить, в каком состоянии ее прервали прошлый раз и "продолжать начатое".

ПРИМЕРЫ ДИАГРАММ СОСТОЯНИЙ



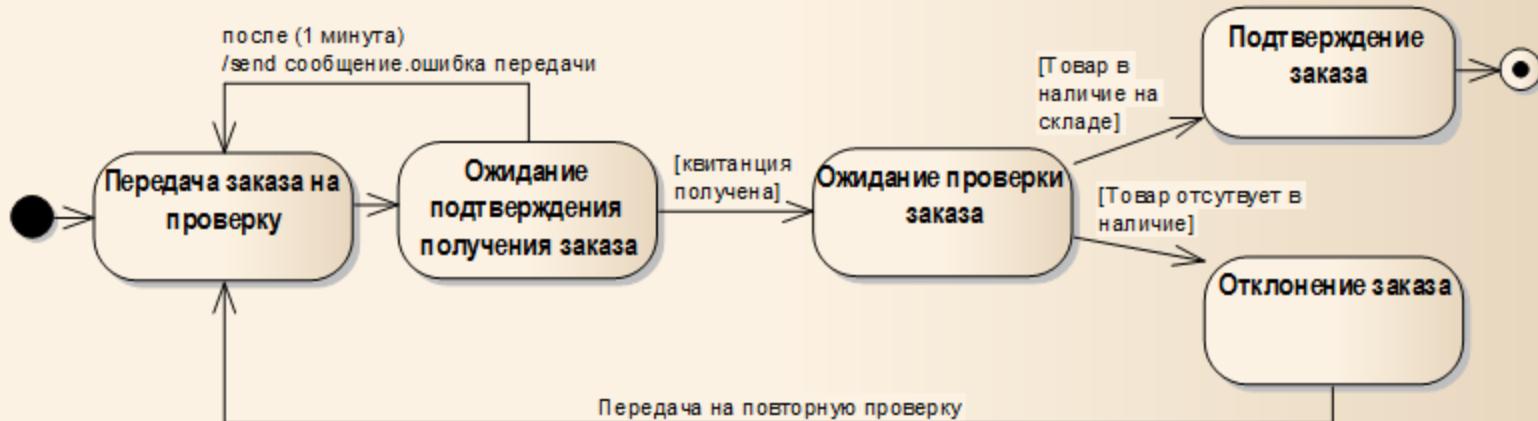


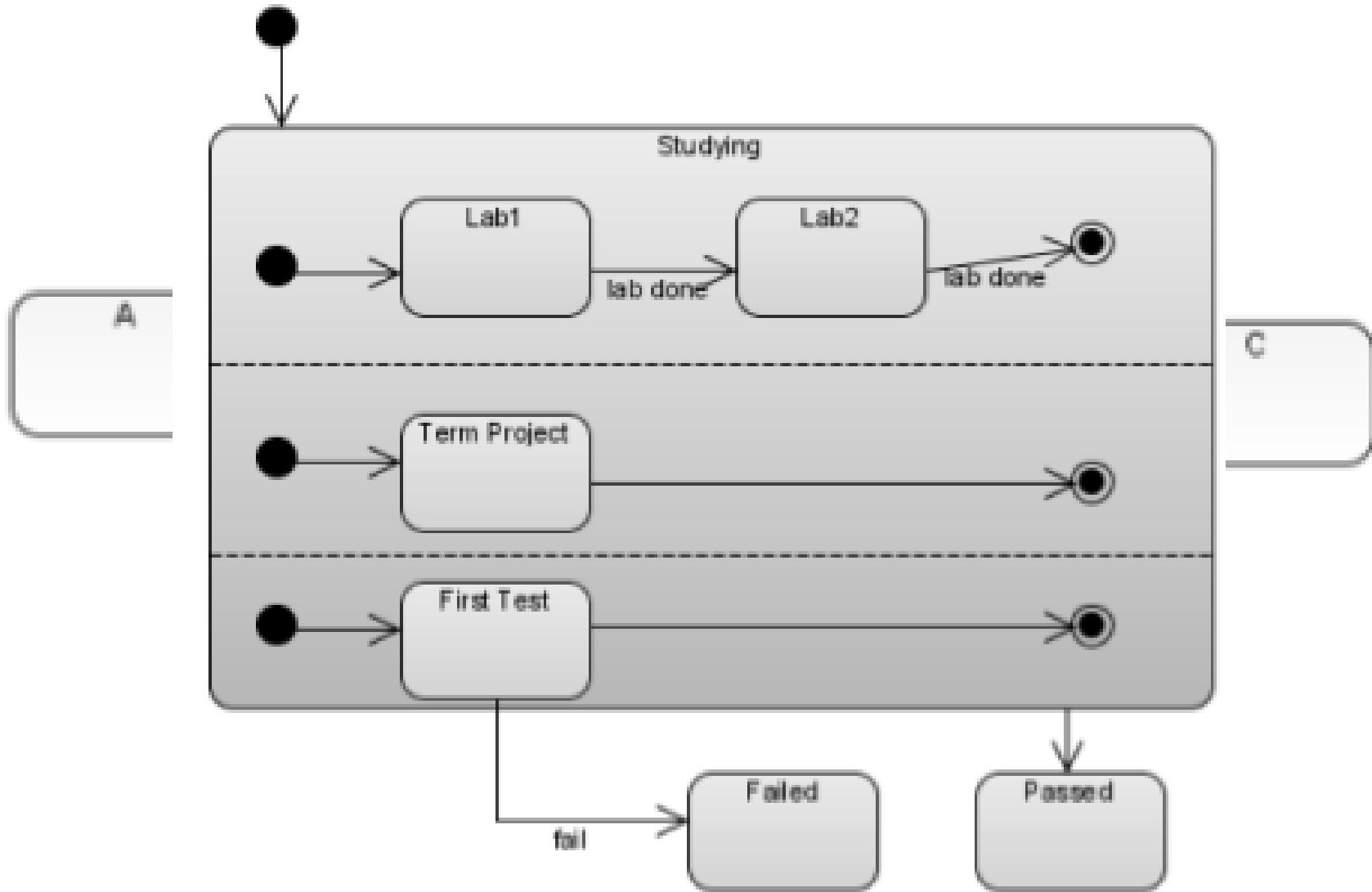
Примеры переходов

ПРИМЕРЫ ПЕРЕХОДОВ

Проверка заказа

- + entry / Создание нового заказа
- + exit / Изменение статуса заказа на "Подтвержден"

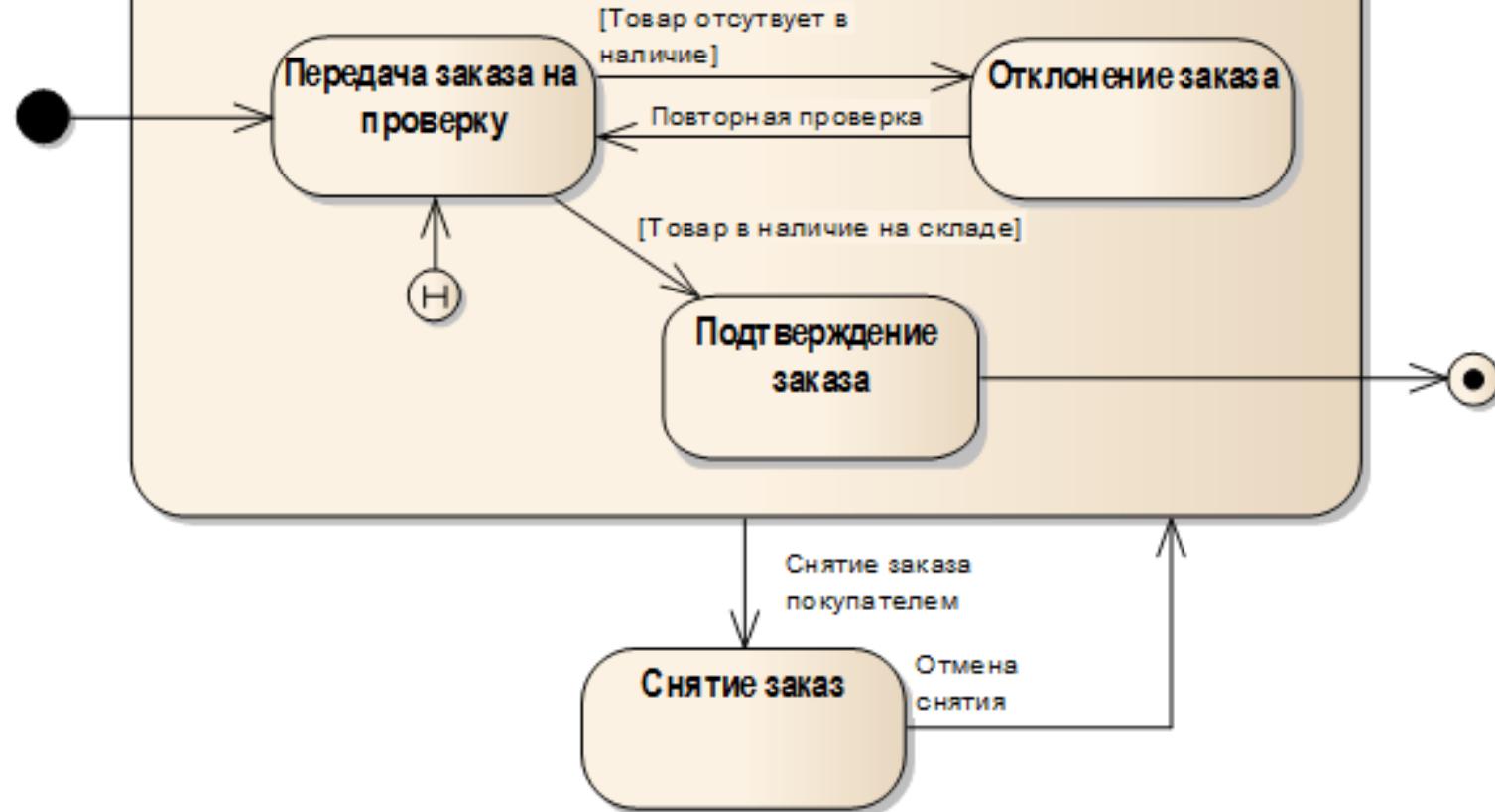




Композитные состояния и вложенные подсостояния

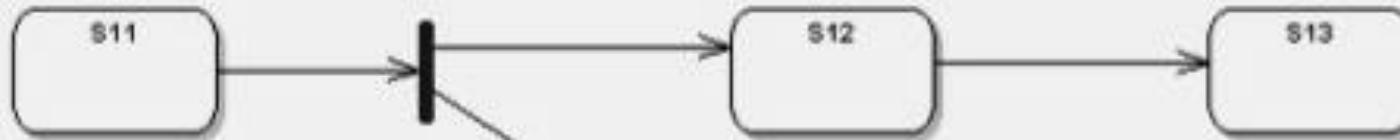
Проверка заказа

- + entry / Создание нового заказа
- + exit / Изменение статуса заказа на "Подтверждён"



Пример композитного состояния

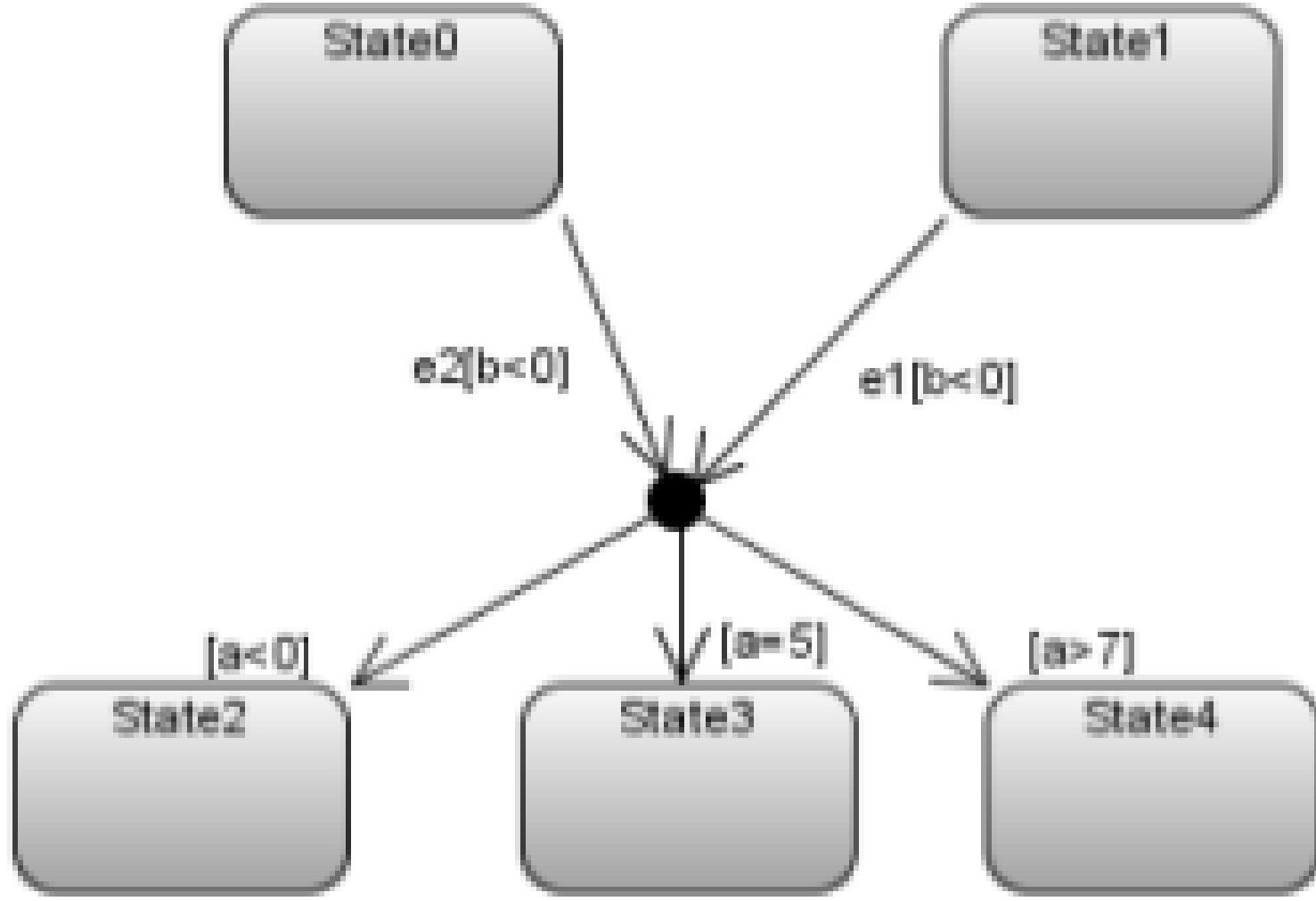
[S1]



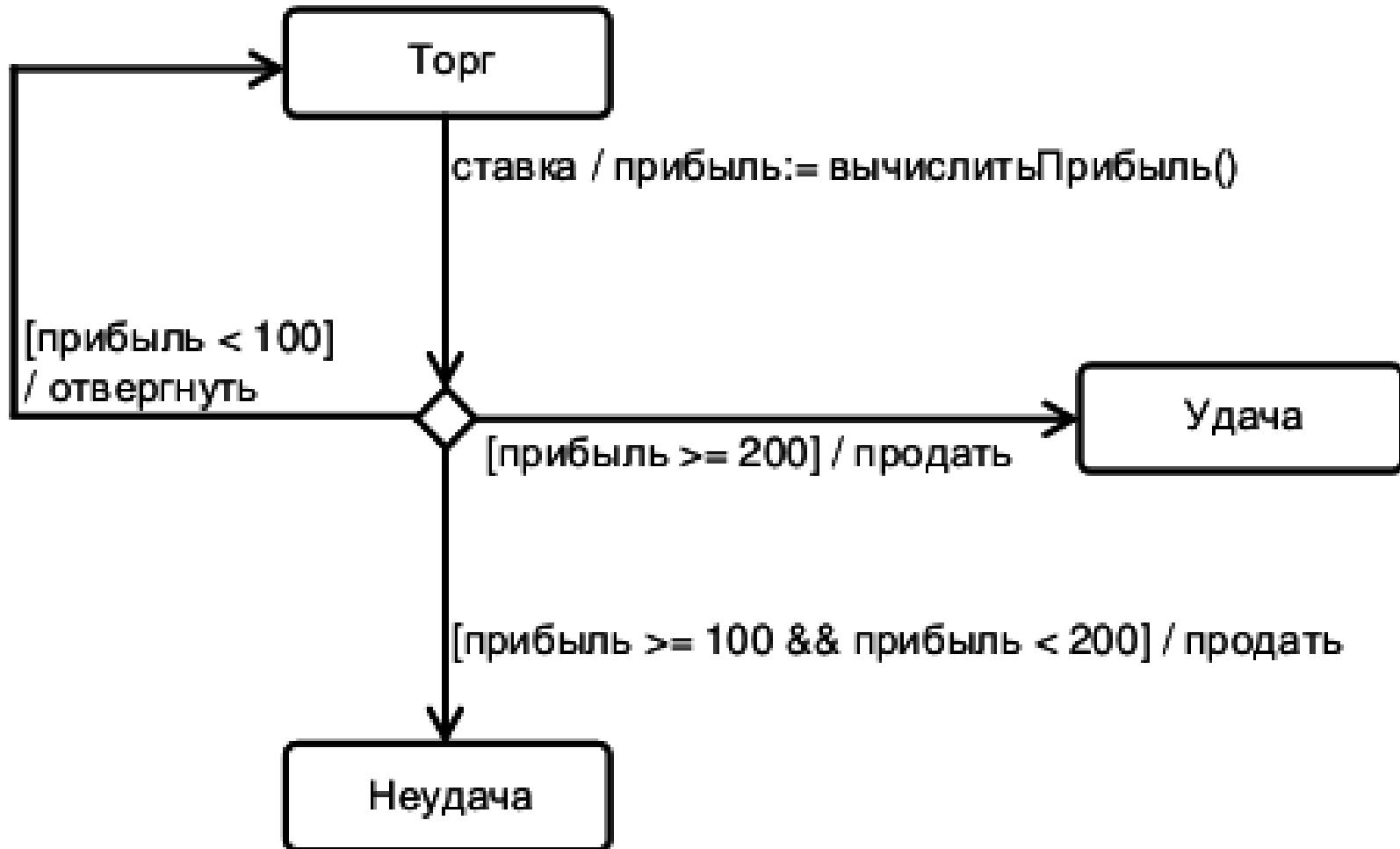
[S2]



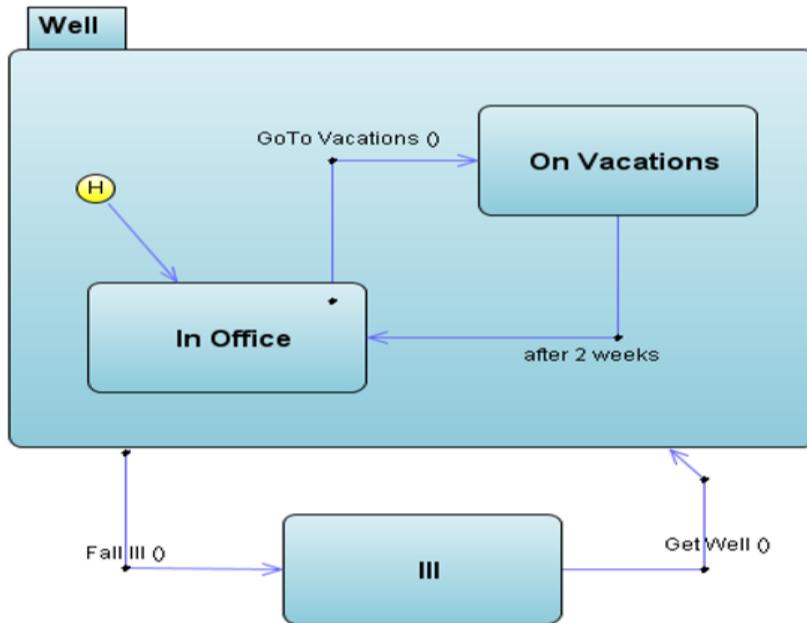
*Слияние и разделение переходов, сложные
переходы*



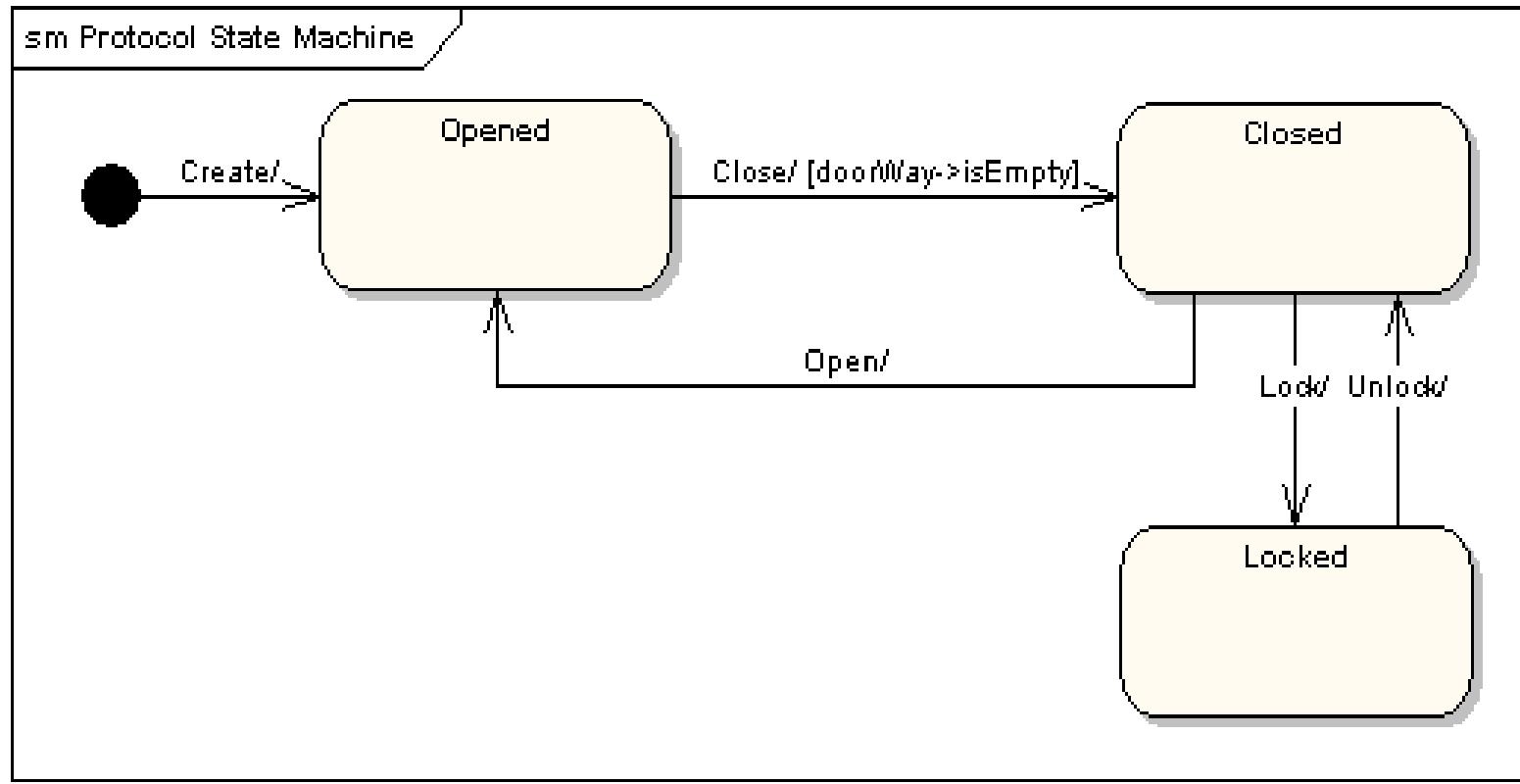
*Слияние и разделение переходов, сложные
переходы*



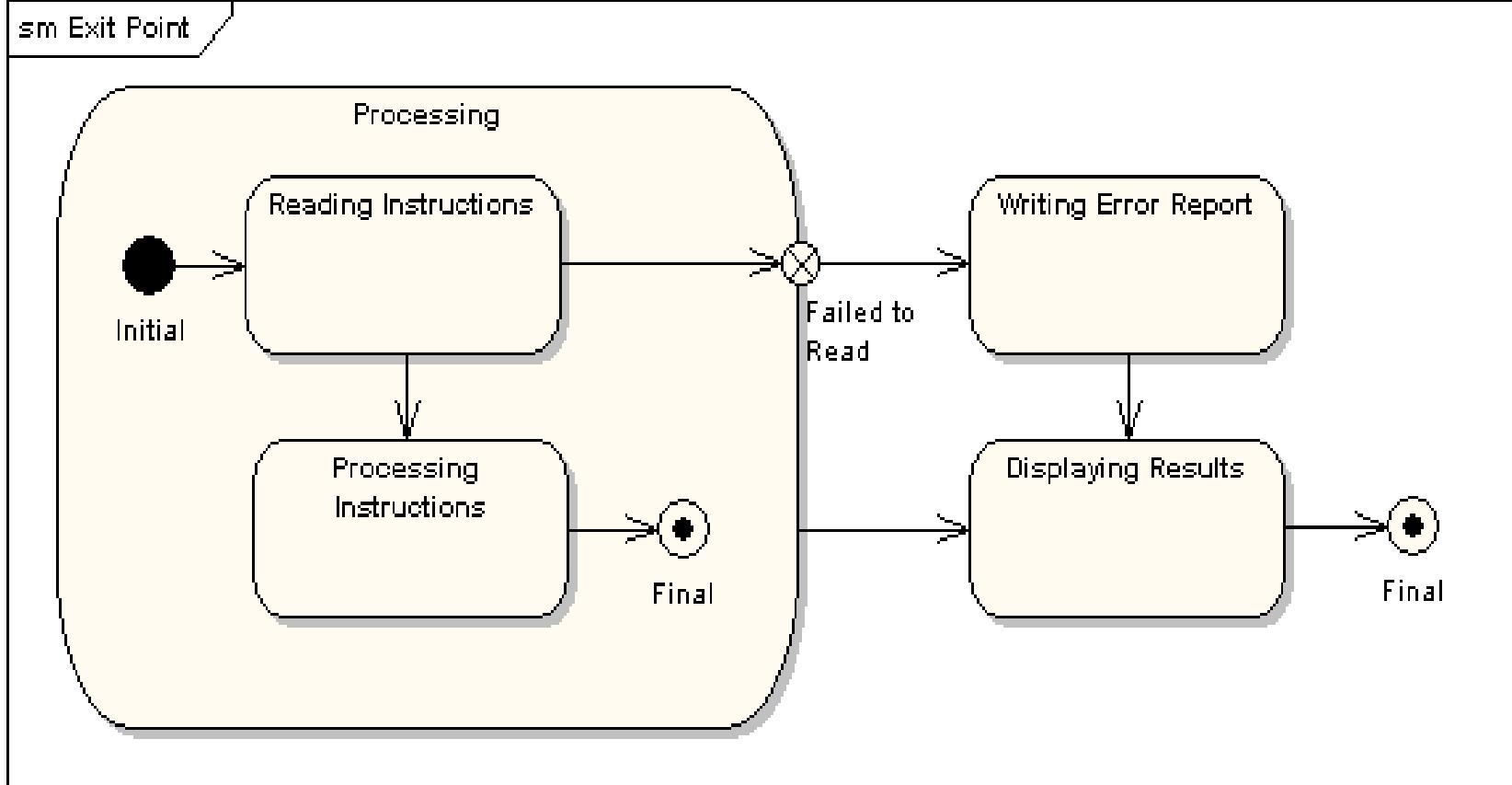
Псевдосостояния выбора



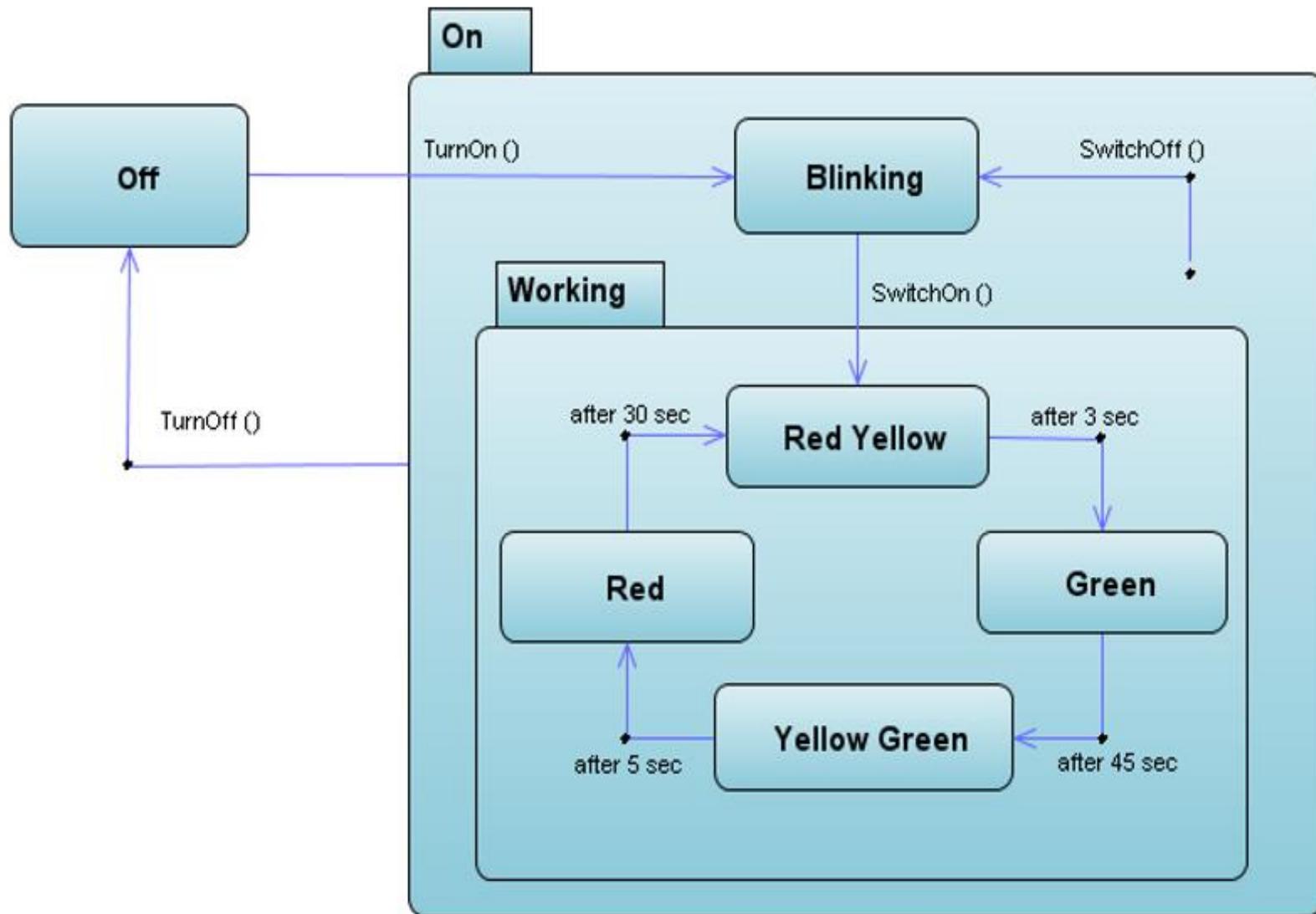
Работающий сотрудник, если все идет хорошо, пребывает в одном из двух взаимоисключающих состояний: либо он на работе (*inOffice*), либо в отпуске (*onVacations*). Но может случиться такая неприятность, как болезнь (*III*). Но в какое состояние переходит сотрудник по выздоровлении? Допустим, что в нашей информационной системе отдела кадров действует положение старого Комплекса законов о труде – если сотрудник заболел, находясь в отпуске, то отпуск прерывается, а по выздоровлении возобновляется. Для того, чтобы построить модель такого поведения, нужно воспользоваться историческим состоянием. В данном случае достаточно поверхности исторического состояния, поскольку на данном уровне вложенности все состояния уже простые.



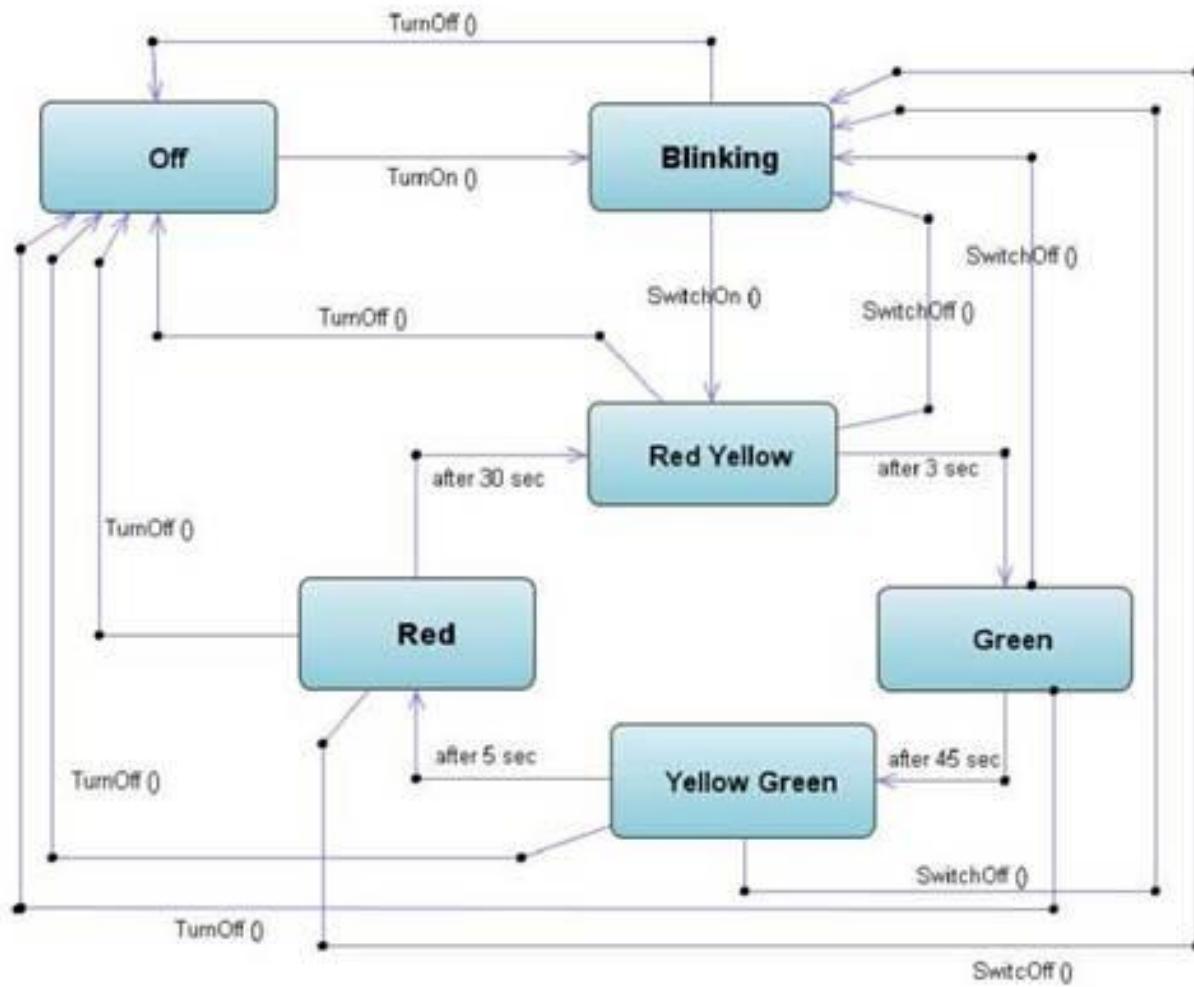
sm Exit Point



Выбор псевдосостояния



Работа Светофора



Работа Светофора (без составных состояний)

СПАСИБО ЗА ВНИМАНИЕ!

ПРОЕКТИРОВАНИЕ ПРОГРАММНЫХ СИСТЕМ

УНИФИЦИРОВАННЫЙ ЯЗЫК МОДЕЛИРОВАНИЯ UML

ЛЕКЦИЯ 4

КЛАССИФИКАЦИЯ ДИАГРАММ

Диаграммы поведения:

**Диаграммы вариантов
использования**

Диаграммы деятельности

Диаграммы состояний

**Диаграммы
взаимодействия:**

- Диаграммы последовательности
- Диаграммы коммуникации
- Диаграммы синхронизации
- Обзорные диаграммы взаимодействия

Структурные диаграммы:

- 1. Диаграммы классов**
- 2. Диаграммы объектов**
- 3. Диаграммы пакетов**
- 4. Диаграммы компонентов**
- 5. Диаграммы составной
структуры**
- 6. Диаграммы размещения**
- 7. Диаграммы профиля**

ОБЪЕКТНО- ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОЕКТИРОВАНИЕ

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОЕКТИРОВАНИЕ

Объектный подход к разработке сложных программных систем безусловно предполагает, что непосредственное программирование (написание кода) начинается далеко не сразу, а этапы анализа и моделирования предметной области, предшествующие программированию, не менее важны и сложны.

Объектный подход применяется на всех основных стадиях жизненного цикла ПО и включает в себя три ключевых понятия:

- ОOA (object oriented analysis) – объектно-ориентированный анализ.
- OOD (object oriented design) – объектно-ориентированное проектирование.
- OOP (object oriented programming) – объектно-ориентированное программирование.

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОЕКТИРОВАНИЕ

Объектно-ориентированный анализ – это методология анализа предметной области, при которой требования к проектируемой системе воспринимаются с точки зрения классов и объектов, выявленных в предметной области.

Объектно-ориентированное проектирование – это методология проектирования, соединяющая в себе процесс объектной декомпозиции и приемы представления логической и физической, а также статической и динамической моделей проектируемой системы.

Объектно-ориентированное программирование – это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

* *Методология - учение о методах, способах и стратегиях исследования предмета.*

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОЕКТИРОВАНИЕ

Предметом рассмотрения нашей дисциплины являются первые две составляющие объектного подхода – объектно-ориентированный анализ и объектно-ориентированное проектирование (далее – ООАП).

Последовательное применение ООАП позволяет получить "хороший" проект программной системы:

- удовлетворяющий требованиям заказчика;
- удобный для коллективной разработки, отладки и тестирования;
- прозрачный;
- развиваемый;
- допускающий повторное использование компонентов.

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОЕКТИРОВАНИЕ

**Базовыми принципами ООАП
являются:**

- **Декомпозиция**
- **Абстрагирование**
- **Иерархичность**
- **Многомодельность**

ПРИНЦИП ДЕКОМПОЗИЦИИ

Декомпозиция – это разбиение целого на составные элементы. В рамках объектного подхода рассматривают два вида декомпозиции: алгоритмическую и объектную.

В соответствии с алгоритмической декомпозицией предметной области при анализе задачи разработчик пытается понять, какие алгоритмы необходимо разработать для ее решения, каковы спецификации этих алгоритмов (вход, выход), и как эти алгоритмы связаны друг с другом. В языках программирования данный подход в полной мере поддерживается средствами модульного программирования (библиотеки, модули, подпрограммы).

Объектная декомпозиция предполагает выделение основных содержательных элементов задачи, разбиение их на типы (классы), определение свойств (данные) и поведения (операции) для каждого класса его, а также взаимодействия классов друг с другом. Объектная декомпозиция поддерживаются всеми современными объектно-ориентированными языками программирования.

ПРИНЦИП АБСТРАГИРОВАНИЯ

Абстрагирование применяется при решении многих задач – любая модель позволяет абстрагироваться от реального объекта, подменяя его изучение исследованием формальной модели.

Абстрагирование в ООП позволяет выделить основные элементы предметной области, обладающие одинаковой структурой и поведением. Такое разбиение предметной области на абстрактные классы позволяет существенно облегчить анализ и проектирование системы.

Согласно этому принципу в модель включаются только те аспекты проектируемой системы, которые имеют непосредственное отношение к выполнению системой своих функций.

ПРИНЦИП ИЕРАРХИЧНОСТИ

Принцип иерархичности предписывает рассматривать процесс построения модели на разных уровнях абстрагирования (детализации) в рамках фиксированных представлений.

Иерархия упорядочивает абстракции, помогает разбить задачу на уровни и постепенно ее решать по принципу "сверху – вниз" или "от общего – к частному", увеличивая детализацию ее рассмотрения на каждом очередном уровне.

ПРИНЦИП МНОГОМОДЕЛЬНОСТИ

Принцип многомодельности утверждает, что никакая единственная модель не может с достаточной степенью адекватности описывать различные аспекты сложной системы, и допускающий использование нескольких взаимосвязанных представлений, отражающих отдельные аспекты поведения или структуры систем.

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОЕКТИРОВАНИЕ

В целом же процесс ООАП можно рассматривать как последовательный переход от разработки наиболее общих моделей и представлений концептуального уровня к более частным и детальным представлениям логического и физического уровня.

При этом на каждом этапе ООАП данные модели последовательно дополняются все большим количеством деталей, что позволяет им более адекватно отражать различные аспекты конкретной реализации сложной системы.

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОЕКТИРОВАНИЕ

На рисунке приведена схема взаимосвязей представлений и моделей сложных систем в процессе ООАП: на двух уровнях иерархии (концептуальном и физическом) используются статические и динамические представления сложной системы.



ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОЕКТИРОВАНИЕ

Методология ООАП тесно связана с концепцией автоматизированной разработки ПО. Именно на этом фоне появление унифицированного языка моделирования (UML), ориентированного на комплексное решение задачи построения модели программной системы, которую, согласно современным концепциям ООАП, следует считать результатом первых двух этапов ЖЦ ПО, было в свое время воспринято сообществом корпоративных программистов с большим оптимизмом.

ОБЪЕКТНО- ОРИЕНТИРОВАННОЕ МОДЕЛИРОВАНИЕ СТРУКТУРЫ СИСТЕМЫ

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ МОДЕЛИРОВАНИЕ СТРУКТУРЫ СИСТЕМЫ

Моделируя структуру, мы описываем составные части системы и отношения между ними.

UML является объектно-ориентированным языком моделирования, поэтому не удивительно, что основным видом составных частей, из которых состоит система, являются объекты.

В каждый конкретный момент функционирования системы можно указать конечный набор конкретных объектов и связей между ними, образующих систему.

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ МОДЕЛИРОВАНИЕ СТРУКТУРЫ СИСТЕМЫ

Кроме самого компактного описания модели, подразумевается известным набор правил интерпретации описания, позволяющих построить по описанию множества любой его элемент.

Сами правила и способ их задания различны в разных случаях, но принцип один и тот же.

В UML этот принцип формализован в виде понятия дескриптора. Дескриптор имеет две стороны: это само описание множества (*intent*) и множество значений, описываемых дескриптором (*extent*).

Антонимом для дескриптора является понятие литерала. Литерал описывает сам себя.

Тип данных *integer* является дескриптором: он описывает множество целых чисел, потенциально бесконечное (или конечное, но достаточно большое, если речь идет о машинной арифметике). Изображение числа 1 описывает само число "один" и более ничего – это литерал.

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ МОДЕЛИРОВАНИЕ СТРУКТУРЫ СИСТЕМЫ

Почти все элементы моделей UML являются дескрипторами – именно поэтому средствами UML удается создавать представительные модели достаточно сложных систем.

Варианты использования и действующие лица – дескрипторы, классы, ассоциации, компоненты, узлы – также дескрипторы. Примечание же является литералом — оно описывает само себя.

Важнейшим типом дескрипторов являются классификаторы.

Классификатор – это дескриптор множества однотипных объектов.

ПАРАДИГМА ПРОГРАММИРОВАНИЯ

Парадигма программирования — это собрание основополагающих принципов, которые служат методической основой конкретных технологий и инструментальных средств программирования.

Центральной идеей парадигмы объектно-ориентированного программирования является инкапсуляция, т. е. структурирование программы на структуры особого вида, объединяющего данные и процедуры их обработки, причем внутренние данные структуры не могут быть обработаны иначе, кроме как предусмотренными для этого процедурами.

ПАРАДИГМА ПРОГРАММИРОВАНИЯ

Объект – структура из данных и процедур их обработки, существующая в памяти компьютера во время выполнения программы.

Класс – описание множества однотипных объектов в тексте программы.

За редкими исключениями в современных объектно-ориентированных системах программирования классы являются дескрипторами.

ПАРАДИГМА ПРОГРАММИРОВАНИЯ

Кроме основной идеи инкапсуляции, с объектно-ориентированным программированием принято ассоциировать также понятия наследования и полиморфизма.

Наследование – это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью..

Полиморфизм – это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

UML (ПРОДОЛЖЕНИЕ)

ДИАГРАММЫ КЛАССОВ

КЛАССИФИКАЦИЯ ДИАГРАММ

Диаграммы поведения:

**Диаграммы вариантов
использования**

Диаграммы деятельности

Диаграммы состояний

**Диаграммы
взаимодействия:**

- Диаграммы последовательности
- Диаграммы коммуникации
- Диаграммы синхронизации
- Обзорные диаграммы взаимодействия

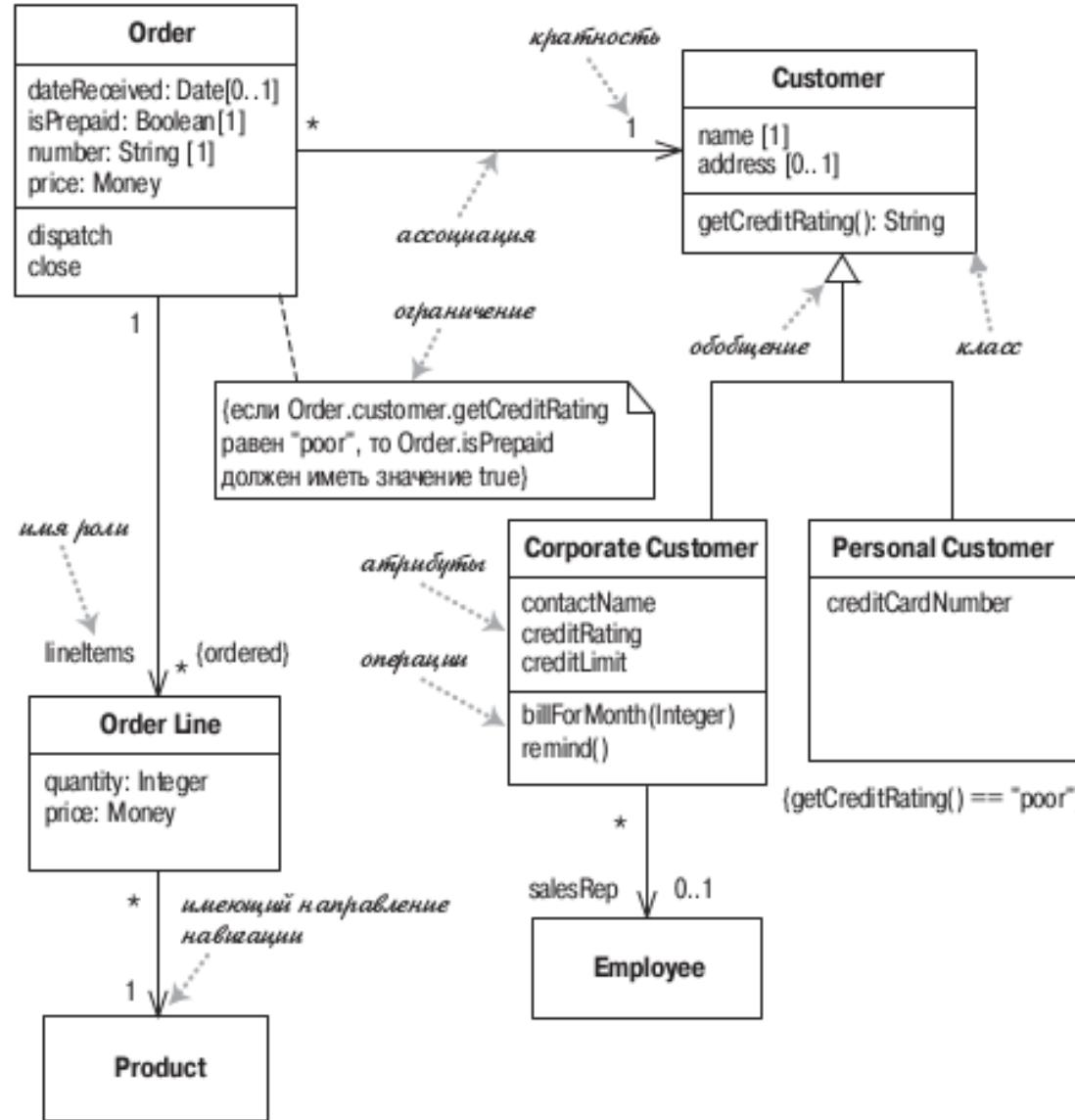
Структурные диаграммы:

- 1. Диаграммы классов**
- 2. Диаграммы объектов**
- 3. Диаграммы пакетов**
- 4. Диаграммы компонентов**
- 5. Диаграммы составной
структуры**
- 6. Диаграммы размещения**
- 7. Диаграммы профиля**

ДИАГРАММА КЛАССОВ(CLASS DIAGRAM) –

основной способ описания структуры системы.
На диаграмме классов применяются один основной тип сущностей: классы (включая многочисленные частные случаи классов: интерфейсы, типы, классы ассоциации и многие другие), между которыми устанавливаются следующие основные типы отношений:

- **ассоциация между классами (с множеством дополнительных подробностей);**
- **обобщение между классами;**
- **зависимость (различных типов) между классами;**
- **реализация.**



Пример диаграммы классов

ИНТЕРФЕЙСЫ И АБСТРАКТНЫЕ КЛАССЫ

Абстрактный класс (*abstract class*) – это класс, который нельзя реализовать непосредственно.

Абстрактная операция (*abstract operation*) - это чистое объявление, которое клиенты могут привязать к абстрактному классу.

Интерфейс – это класс, не имеющий реализаций, то есть вся его функциональность абстрактна.

КЛАСС

Класс – один из самых "богатых" элементов моделирования UML. Описание класса может включать множество различных элементов, их группируют по разделам.

Стандартных разделов три:

- раздел имени – наряду с обязательным именем может содержать также стереотип, кратность (т.е. ограничение на количество экземпляров) и список свойств;**
- раздел атрибутов – содержит список описаний атрибутов класса;**
- раздел операций – содержит список описаний операций класса.**

КЛАСС

Имя

Имя

Атрибуты

Имя

Атрибуты

Операции

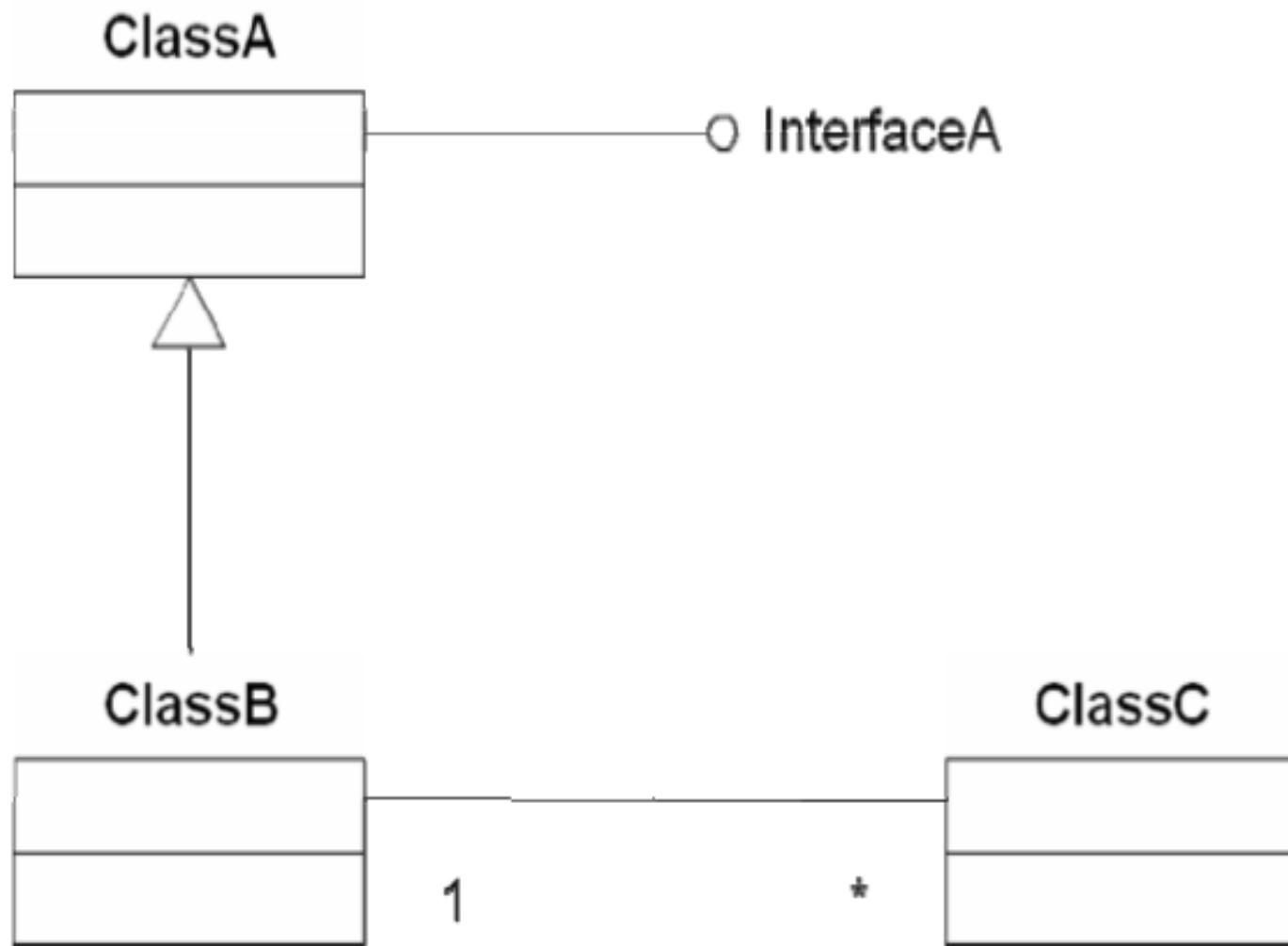
КЛАСС

Нотация классов очень проста – это всегда прямоугольник. Если разделов более одного, то внутренность прямоугольника делится горизонтальными линиями на части, соответствующие разделам. Содержимым раздела в любом случае является текст. Текст внутри стандартных разделов должен иметь определенный синтаксис.

Некоторые инструменты позволяют помещать в разделы класса не только тексты, но также фигуры и значки.

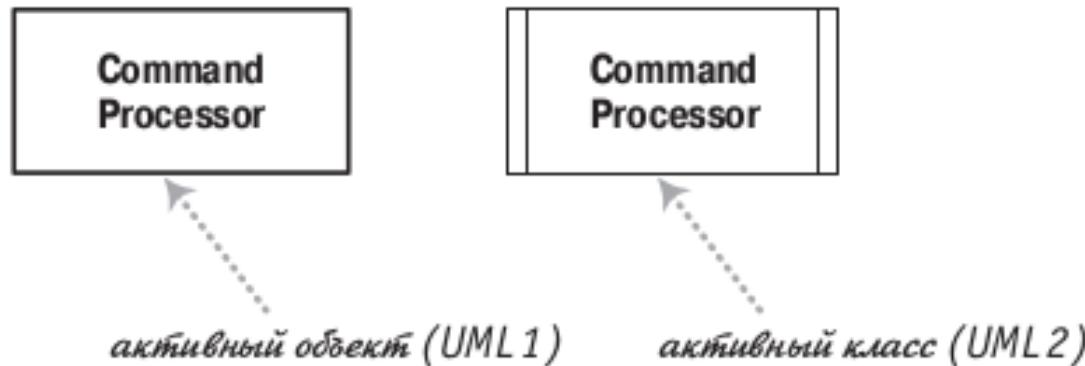
Раздел имени класса в общем случае имеет следующий синтаксис:

«стереотип» ИМЯ {свойства} кратность



АКТИВНЫЙ КЛАСС

Активный класс (active class) имеет экземпляры, каждый из которых выполняет и управляет собственным потоком управления.



ГРАНИЧНЫЙ КЛАСС

Граничные классы

Граничными классами (boundary classes) называются такие классы, которые расположены на границе системы и всей окружающей среды. Это экранные формы, отчеты, интерфейсы с аппаратурой (такой как принтеры или сканеры) и интерфейсы с другими системами.

Чтобы найти граничные классы, надо исследовать диаграммы вариантов использования. Каждому взаимодействию между действующим лицом и вариантом использования должен соответствовать, по крайней мере, один граничный класс. Именно такой класс позволяет действующему лицу взаимодействовать с системой.

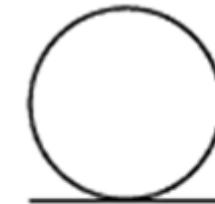


<<boundary>>
Имя класса

КЛАССЫ-СУЩНОСТИ

Классы-сущности

Классы-сущности (entity classes) содержат хранимую информацию. Они имеют наибольшее значение для пользователя, и потому в их названиях часто используют термины из предметной области. Обычно для каждого класса-сущности создают таблицу в базе данных.



класс-сущность

<<entity>>
Имя класса

УПРАВЛЯЮЩИЕ КЛАССЫ

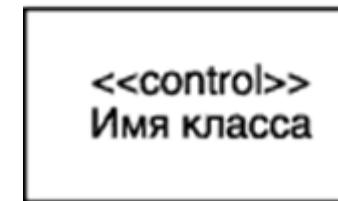
Управляющие классы

Управляющие классы (control classes) отвечают за координацию действий других классов. Обычно у каждого варианта использования имеется один управляющий класс, контролирующий последовательность событий этого варианта использования.

Управляющий класс отвечает за координацию, но сам не несет в себе никакой функциональности, так как остальные классы не посыпают ему большого количества сообщений. Вместо этого он сам посыпает множество сообщений. Управляющий класс просто делегирует ответственность другим классам, по этой причине его часто называют классом-менеджером.



управляющий класс



СТАНДАРТНЫЕ СТЕРЕОТИПЫ КЛАССОВ

Стереотип	Описание
actor	действующее лицо
enumeration	перечислимый тип данных
exception	сигнал, распространяемый по иерархии обобщений
implementation	реализация класса
Class	
interface	нет атрибутов и все операции абстрактные
metaclass	экземпляры являются классами
powertype	метакласс, экземплярами которого являются все наследники данного класса
process, thread	активные классы
signal	класс, экземплярами которого являются сообщения
stereotype	стереотип
type (datatype)	тип данных
utility	нет экземпляров = служба

СВОЙСТВА КЛАССА

Свойства представляют структурную функциональность класса.

Свойства представляют единое понятие, воплощающееся в двух совершенно различных сущностях: в атрибутах и в ассоциациях.

АТРИБУТЫ

Атрибут – это именованное место (или, как говорят, слот), в котором может храниться значение.

Атрибуты класса перечисляются в разделе атрибутов. В общем случае описание атрибута имеет следующий синтаксис.

видимость ИМЯ кратность: тип = начальное_значение {свойства}

Пример:

-имя: String [1] = "Без имени" {readOnly}

-Видимость, как обычно, обозначается знаками +, -, # (соответственно открыт, закрыт, защищен). Если видимость не указана, то никакого значения видимости по умолчанию не подразумевается.

Если имя атрибута подчеркнуто, то это означает, что областью действия данного атрибута является класс, а не экземпляр класса, как обычно. Другими словами, все объекты – экземпляры этого класса совместно используют одно и тоже значение данного атрибута, общее для всех экземпляров. В обычной ситуации (нет подчеркивания) каждый экземпляр класса хранит свое индивидуальное значение атрибута.

ЗАМЕЧАНИЕ Подчеркивание описания атрибута соответствует описателю static в языке С++.

АТРИБУТЫ

видимость ИМЯ кратность: тип = начальное_значение {свойства}

Кратность, если она присутствует, определяет данный атрибут как массив (определенной или неопределенной длины).

Тип атрибута – это либо примитивный (встроенный) тип, либо тип, определенный пользователем. Начальное значение имеет очевидный смысл: при создании экземпляра данного класса атрибут получает указанное значение. Если начальное значение не указано, то никакого значения по умолчанию не подразумевается.

Как и любой другой элемент модели, атрибут может быть наделен дополнительными свойствами в форме ограничений и именованных значений.

У атрибутов имеется еще одно стандартное свойство: **изменяемость**.

ЗНАЧЕНИЯ СВОЙСТВА ИЗМЕНЯЕМОСТИ АТРИБУТА

Значение	Описание
changeable	Никаких ограничений на изменение значения атрибута не накладывается. Данное значение имеет место по умолчанию, поэтому указывать в модели его излишне.
addOnly	Это значение применимо только к атрибутам, кратность которых больше единицы. При изменении значения атрибута новое значение добавляется в массив значений, но старые значения не меняются и не исчезают. Такой атрибут "помнит" историю своего изменения.
frozen	Значение атрибута задается при инициализации объекта и не может меняться.

ПРИМЕР

Например, в информационной системе отдела кадров класс Person, скорее всего, должен иметь атрибут, хранящий имя сотрудника.

В табл. приведен список примеров описаний такого атрибута. Все описания синтаксически допустимы и могут быть использованы в соответствии с текущим уровнем детализации модели.

Пример

name

Пояснение

Минимальное возможное описание — указано только имя атрибута

+name

Указаны имя и открытая видимость — предполагается, манипуляции с именем будут производится непосредственно

-name : String

Указаны имя, тип и закрытая видимость — манипуляции с именем будут производится с помощью специальных операций

-name [1..3] : String

Указана кратность (для хранения трех составляющих; фамилии, имени и отчества)

-name : String = "Novikov"

Указано начальное значение

+name : String {frozen}

Атрибут объявлен не меняющим своего значения после начального присваивания и открытым

КРАТНОСТЬ АТРИБУТА

[0..1] означает, что кратность атрибута может принимать значение 0 или 1. При этом 0 означает отсутствие значения для данного атрибута.

[0..*] означает, что кратность атрибута может принимать любое положительное целое значение большее или равное 0. Эта кратность может быть записана короче в виде простого символа - [*].

[1..*] означает, что кратность атрибута может принимать любое положительное целое значение большее или равное 1.

[1..5] означает, что кратность атрибута может принимать любое значение из чисел: 1, 2, 3, 4, 5.

[1..3,5,7] означает, что кратность атрибута может принимать любое значение из чисел: 1, 2, 3, 5, 7.

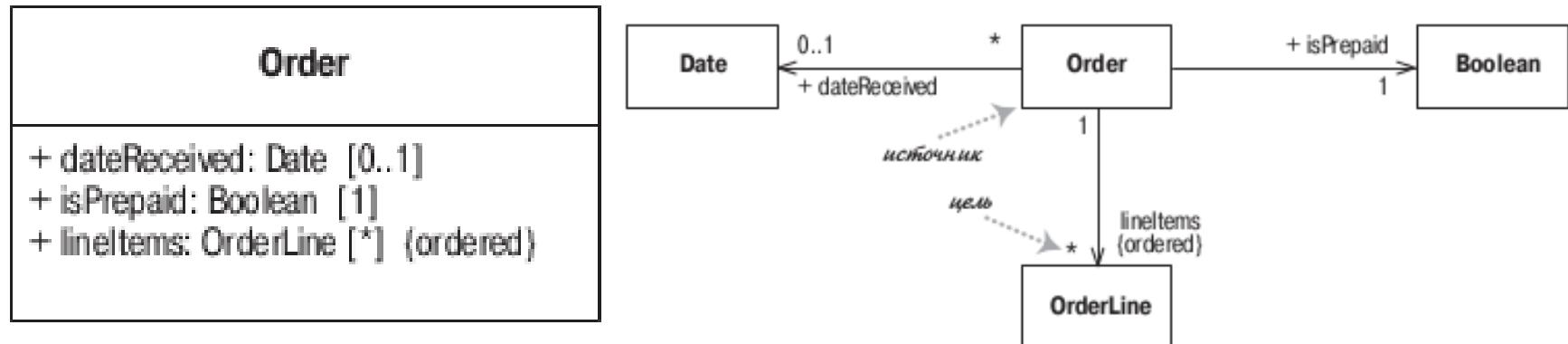
[1..3,7.. 10] означает, что кратность атрибута может принимать любое значение из чисел: 1, 2, 3, 7, 8, 9, 10.

[1..3,7..*] означает, что кратность атрибута может принимать любое значение из чисел: 1, 2, 3, а также любое положительное целое значение большее или равное 7.

Если кратность атрибута не указана, то по умолчанию принимается ее значение равное 1..1, т. е. в точности 1.

АССОЦИАЦИИ

Ассоциация – это непрерывная линия между двумя классами, направленная от исходного класса к целевому классу. Имя свойства (вместе с кратностью) располагается на целевом конце ассоциации.



ЗАВИСИМОСТИ

Всего в UML определено 17 стандартных стереотипов отношения зависимости, которые можно разделить на 6 групп:

между классами и объектами на диаграмме классов;

между пакетами;

между вариантами использования;

между объектами на диаграмме взаимодействия;

между состояниями автомата;

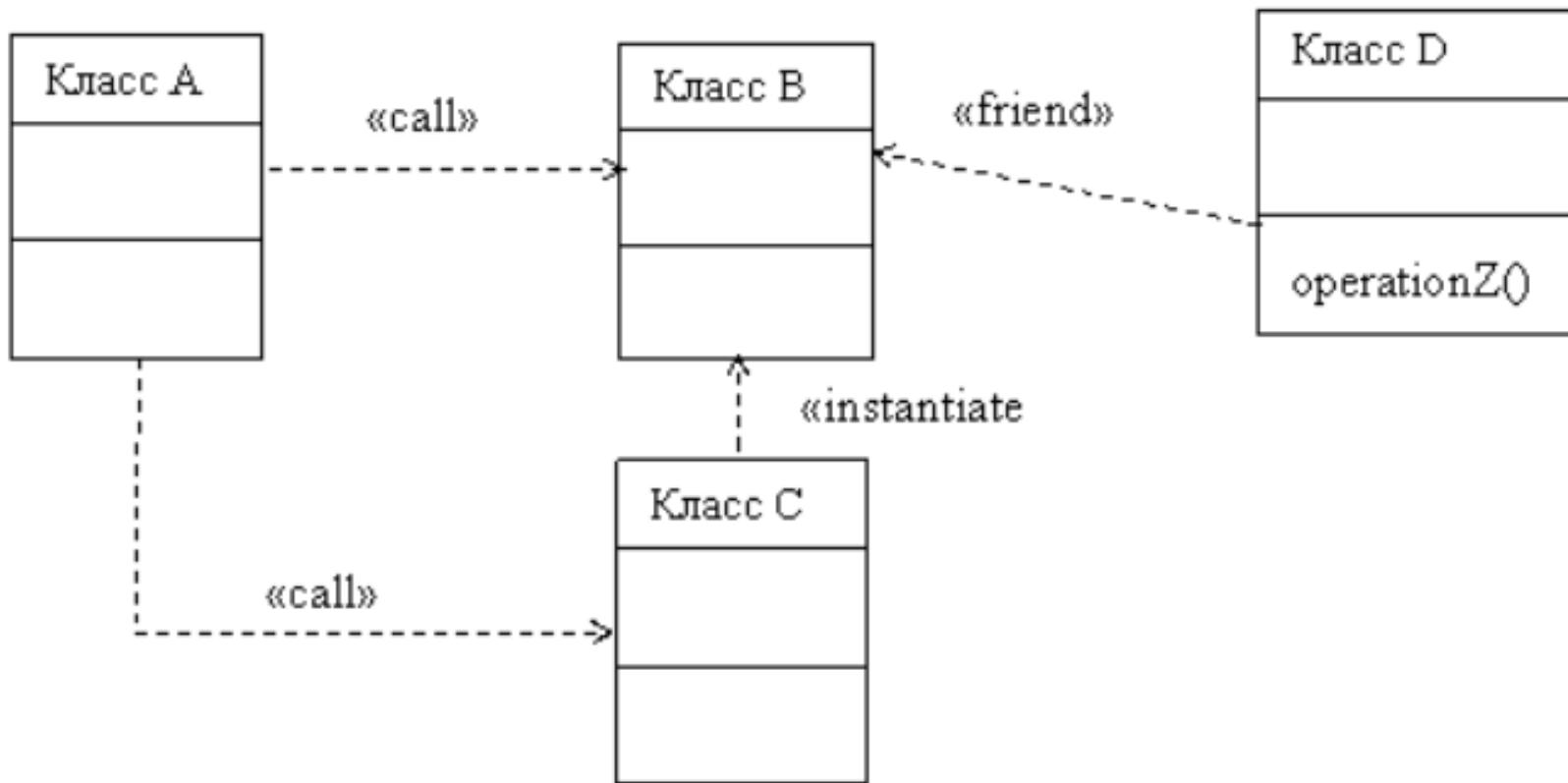
между подсистемами и моделями.

Зависимости на диаграммах классов используются сравнительно редко, потому что имеют более расплывчатую семантику по сравнению с ассоциациями и обобщением.

СТАНДАРТНЫЕ СТЕРЕОТИПЫ ЗАВИСИМОСТЕЙ НА ДИАГРАММЕ КЛАССОВ

Стереотип	Применение
bind	Подстановка параметров в шаблон. Независимой сущностью является шаблон (класс с параметрами), а зависимой — класс, который получается из шаблона заданием аргументов.
derive	Буквально означает "может быть вычислен по". Зависимость с данным стереотипом применяется не только к классам, но и к другим элементам модели: атрибутам, ассоциациям и др. Суть состоит в том, зависимый элемент может быть восстановлен по информации, содержащейся в независимом элементе. Таким образом, данная зависимость показывает, что зависимый элемент, вообще говоря, излишен и введен в модель из соображений удобства, наглядности и т.д.
friend	Назначает специальные права видимости. Зависимый класс имеет доступ к составляющим независимого класса, даже если по общим правилам видимости он не имеет на это прав.
instanceOf	Указывает, что зависимый объект (или класс) является экземпляром независимого класса (метакласса).
instantiate	Указывает, что операции независимого класса создают экземпляры зависимого класса.
powertype	Показывает, что экземплярами зависимого класса являются подклассы независимого класса. Таким образом, в данном случае зависимый класс является метаклассом.
refine	Указывает, что зависимый класс уточняет (конкретизирует) независимый. Данная зависимость показывает, что связанные классы концептуально совпадают, но находятся на разных уровнях абстракции.
use	Зависимость самого общего вида, показывающая, что зависимый класс каким-либо образом использует независимый класс.

ЗАВИСИМОСТИ



классификатор (в частности, класс) использует интерфейс – это показывается с помощью зависимости со стереотипом «call»

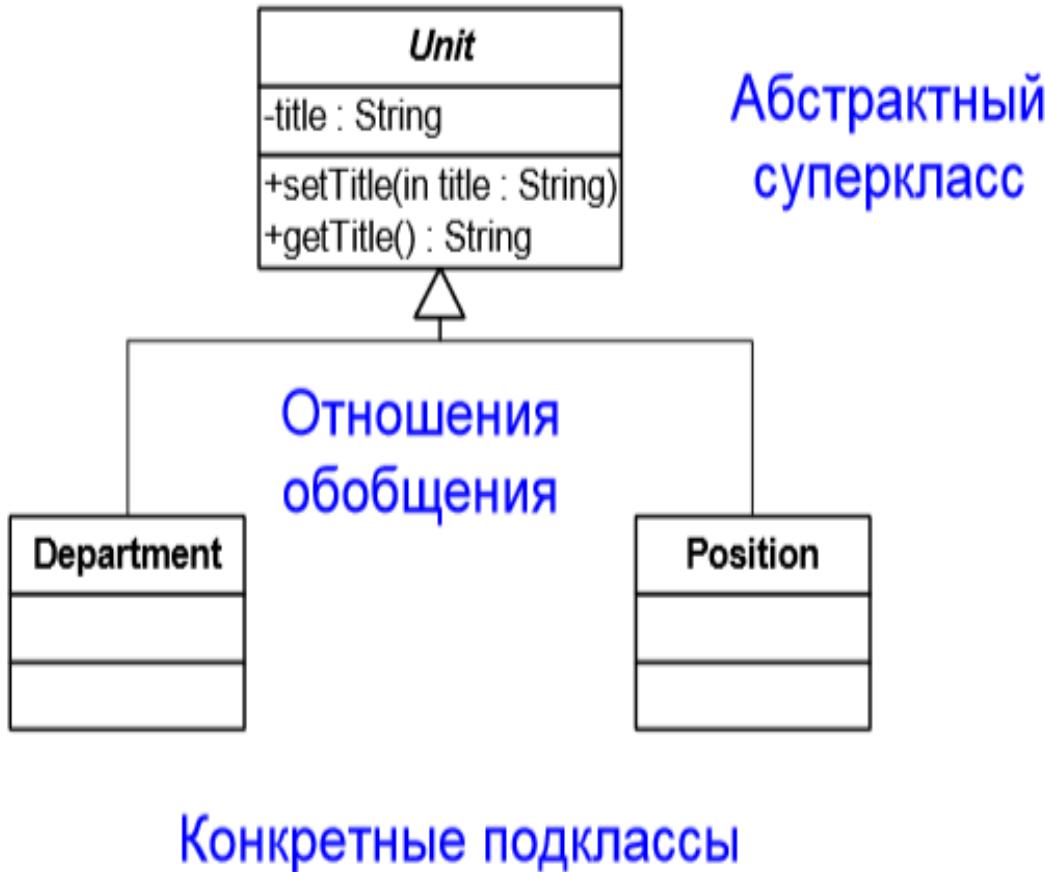
ОБОБЩЕНИЕ

Отношение обобщения часто применяется на диаграмме классов. . Как правило, между объектами в одной системе общее есть и это общее целесообразно выделить в отдельный класс. При этом общие составляющие, собранные в суперклассе, автоматически наследуются подклассами. Таким образом, сокращается общее количество описаний, а значит, уменьшается вероятность допустить ошибку.

Использование обобщений не ограничивает свободу проектировщика системы, поскольку унаследованные составляющие можно переопределить в подклассе, если нужно. При обобщении выполняется принцип подстановочности. Фактически это означает увеличение гибкости и универсальности программного кода при одновременном сохранении надежности, обеспечиваемой контролем типов.

Если, например, в качестве типа параметра некоторой процедуры указать суперкласс, то процедура будет с равным успехом работать в случае, когда в качестве аргумента ей передан объект любого подкласса данного суперкласса. Суперкласс может быть конкретным, идентифицированным, а может быть абстрактным, введенным именно для построения отношений обобщения.

ОБОБЩЕНИЕ



В UML допускается, чтобы класс был подклассом нескольких суперклассов (множественное наследование), не требуется, чтобы у базовых классов был общий суперкласс (несколько иерархий обобщения) и вообще не накладывается никаких ограничений, кроме частичной упорядоченности (т. е. отсутствия циклов в цепочках обобщений).

АССОЦИАЦИИ

Отношение ассоциации является самым важным на диаграмме классов. В общем случае ассоциация, которая обозначается сплошной линией, соединяющей классы, означает, что экземпляры одного класса связаны с экземплярами другого класса. Поскольку экземпляров может быть много, и каждый может быть связан с несколькими, ясно, что ассоциация является дескриптором, который описывает множество связанных объектов.

В UML ассоциация является классификатором, экземпляры которого называются связями. Связь между объектами (экземплярами классов) в программе может быть организована самыми разными способами.

При моделировании на UML техника реализации связи между объектами не имеет значения. Ассоциация в UML подразумевает лишь то, что связанные объекты обладают достаточной информацией для организации взаимодействия. Возможность взаимодействия означает, что объект одного класса может послать сообщение объекту другого класса, в частности, вызвать операцию или же прочитать, или изменить значение открытого атрибута.

АССОЦИАЦИИ

Моделирование структуры взаимосвязей объектов (т.е. выявление ассоциаций) является одной из ключевых задач при разработке.

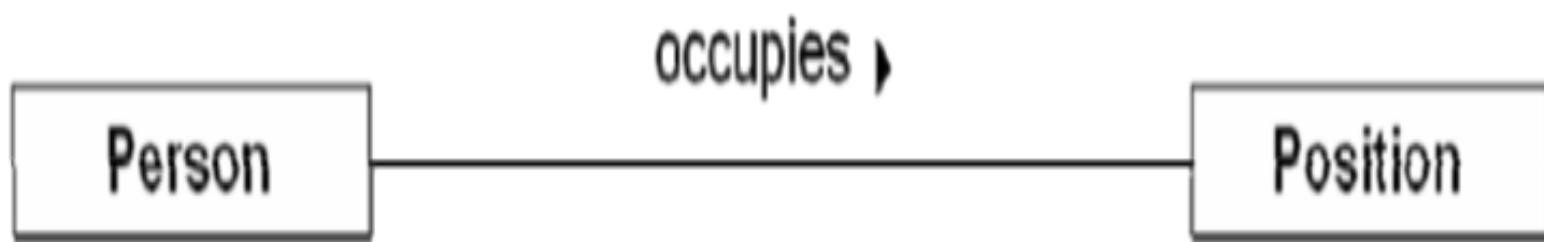
Базовая нотация ассоциации (сплошная линия) позволяет указать, что объекты ассоциированных классов могут взаимодействовать во время выполнения.

Но это только малая часть того, что можно моделировать с помощью отношения ассоциации. Для ассоциации в UML предусмотрено наибольшее количество различных дополнений.

Дополнения не являются обязательными: их используют при необходимости, в различных ситуациях по-разному. Если использовать все дополнения сразу, то диаграмма становится настолько перегруженной, что ее трудно читать.

АССОЦИАЦИИ

Имя ассоциации указывается в виде строки текста над (или под, или рядом с) линией ассоциации. Имя не несет дополнительной семантической нагрузки, а просто позволяет различать ассоциации в модели. Обычно имя не указывают, за исключением многополюсных ассоциаций или случая, когда одна и та же группа классов связана несколькими различными ассоциациями. Дополнительно можно указать направление чтения имени ассоциации.



АССОЦИАЦИИ

Итак, для ассоциации определены следующие дополнения:

- *имя ассоциации (возможно, вместе с направлением чтения);*
- *кратность полюса ассоциации;*
- *вид агрегации полюса ассоциации;*
- *роль полюса ассоциации;*
- *направление навигации полюса ассоциации;*
- *упорядоченность объектов на полюсе ассоциации;*
- *изменяемость множества объектов на полюсе ассоциации;*
- *квалификатор полюса ассоциации;*
- *класс ассоциации;*
- *видимость полюса ассоциации;*
- *многополюсные ассоциации.*
- *Полюсом называется конец линии ассоциации. Обычно используются двухполюсные ассоциации, но могут быть и многополюсные.*

ПОЛЮС АССОЦИАЦИИ

Полюс ассоциации находится у края прямоугольника, обозначающего класс. Свойства полюса ассоциации передаются в виде различных обозначений, отображаемых около соответствующего конца маршрута ассоциации.

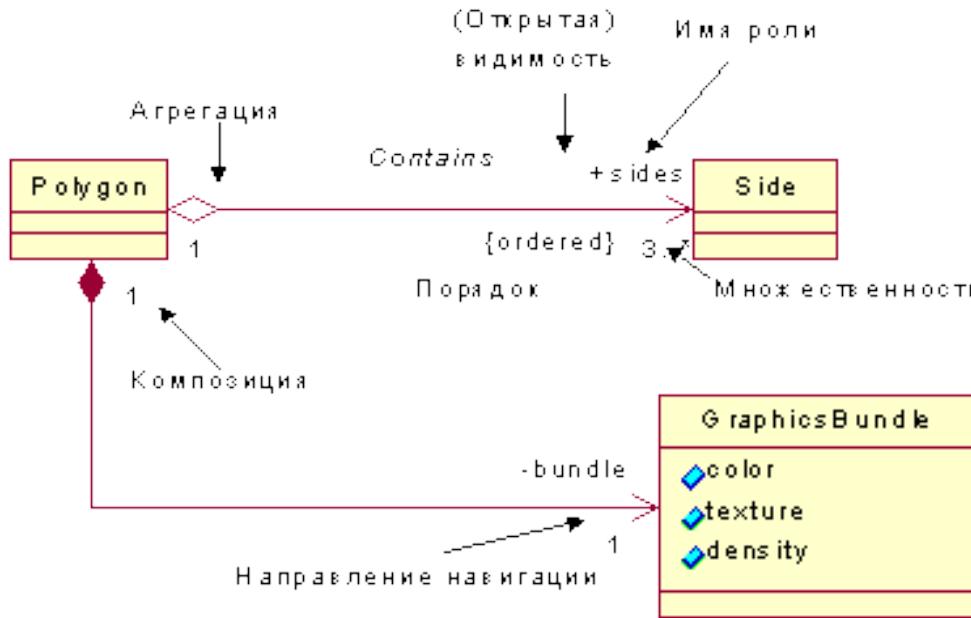
Роль полюса ассоциации, называемая также спецификатором интерфейса – это способ указать, как именно участвует классификатор (присоединенный к данному полюсу ассоциации) в ассоциации. В общем случае данное дополнение имеет следующий синтаксис:

видимость ИМЯ : тип

СПИСОК СВОЙСТВ ПОЛЮСОВ АССОЦИАЦИИ

aggregation (агрегация)	Маленький не закрашенный ромб на полюсе агрегации; для композиции используется закрашенный ромб.
changeability (изменяемость)	Пометки {frozen} или {addQOnly} около целевого полюса ассоциации. Значение по умолчанию {changeable} часто опускается.
interface specifier (спецификатор типа интерфейса)	Текстовый суффикс для имени роли - : тип.
multiplicity (множественность)	Текстовая наметка у полюса ассоциации - min..max.
navigability (возможность навигации)	Наконечник стрелки на конце маршрута ассоциации, показывающий возможность навигации в этом направлении. Если указатель возможности навигации отсутствует, то принято считать, что навигация возможна в обоих направлениях (так как нечасто возникает потребность ассоциации, в которой навигация вообще невозможна).
ordering (порядок)	Текстовая пометка {ordered} возле целевого полюса ассоциации, которая обозначает упорядоченность списка экземпляров целевого класса.
qualifier (квалифиликатор)	Маленький прямоугольник между концом маршрута и исходным классом. Прямоугольник содержит один или несколько атрибутов ассоциации - квалифиликаторов.
rolename (имя роли)	Текстовая пометка у целевого полюса ассоциации.
target scope (целевая область действия)	Указывает на то, что имя роли действительно в области действия класса. В противном случае - в области действия его экземпляра.
visibility (видимость)	Один из символов видимости ("+", "#", "-"), стоящий перед именем

ПОЛЮС АССОЦИАЦИИ



Различные указатели, применяемые на полюсах ассоциации

КРАТНОСТЬ ПОЛЮСА АССОЦИАЦИИ

Кратность полюса ассоциации указывает, сколько объектов данного класса (со стороны данного полюса) участвуют в связи. Кратность может быть задана как конкретное число, и тогда в каждой связи со стороны данного полюса участвуют ровно столько объектов, сколько указано. Более распространен случай, когда кратность указывается как диапазон возможных значений, и тогда число объектов, участвующих в связи должно находиться в пределах указанного диапазона. При указании кратности можно использовать символ *, который обозначает неопределенное число.

Например, если в информационной системе отдела кадров не предусматривается дробление ставок и совмещение должностей, то (работающему)

соответствует о

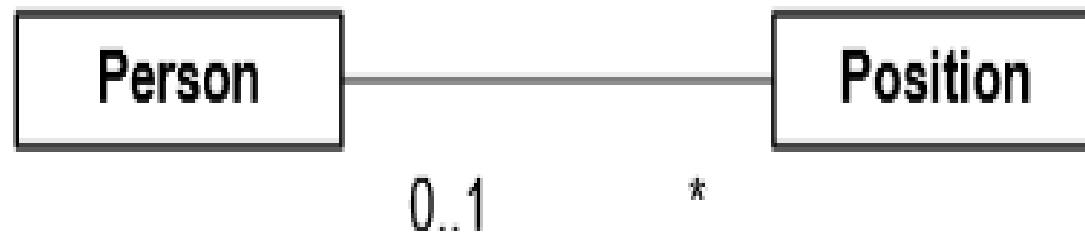


Кратность
задана
диапазоном

Кратность
задана
числом

АССОЦИАЦИИ

Более сложные случаи также легко моделируются с помощью кратности полюсов. Например, если мы хотим предусмотреть совмещение должностей и хранить информацию даже о неработающих сотрудниках, то диаграмма примет вид:



АССОЦИАЦИИ

В UML используются два частных, но очень важных случая отношения ассоциации, которые называются агрегацией и композицией.

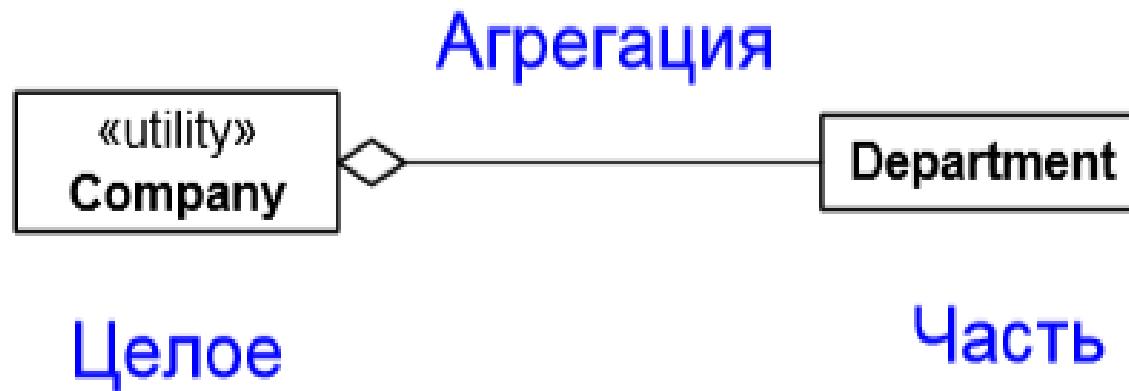
В обоих случаях речь идет о моделировании отношения типа «часть – целое». Ясно, что отношения такого типа следует отнести к отношениям ассоциации, поскольку части и целое обычно взаимодействуют.

Агрегация от класса A к классу B означает, что объекты (один или несколько) класса A входят в состав объекта класса B.

Это отмечается с помощью специального графического дополнения: на полюсе ассоциации, присоединенному к «целому», т. е., в данном случае, к классу B, изображается ромб.

АССОЦИАЦИИ

Например, на рис. указано, что подразделение является частью компании (агрегация).

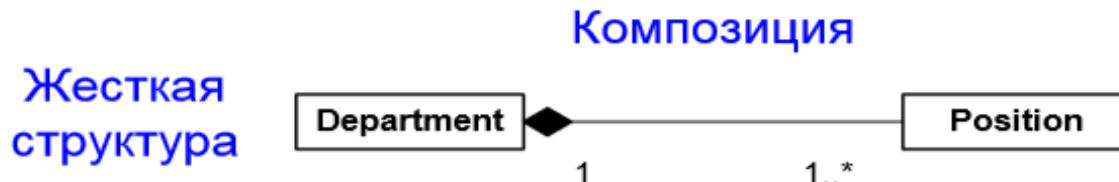


При этом никаких *дополнительных ограничений не накладывается*: объект класса А (часть) может быть связан отношениями агрегации с другими объектами (т. е. участвовать в нескольких агрегациях), создаваться и уничтожаться независимо от объекта класса В (целого).

АССОЦИАЦИИ

Композиция накладывает более сильные ограничения: композиционно часть может входить только в одно целое, часть существует только пока существует целое и прекращает свое существование вместе с целым.

Графически отношение композиции отображается закрашенным ромбом.



Жесткая
структура

Мягкая
структура

Агрегация



Приведены два возможных взгляда на отношения между подразделениями и должностями в информационной системе отдела кадров. В первом случае (вверху), мы считаем, что в организации принята жесткая («армейская») структура: каждая должность входит ровно в одно подразделение, в каждом подразделении есть по меньшей мере одна должность (начальник). Во втором случае (внизу) структура организации более аморфна: возможны «висящие в воздухе» должности, бывают «пустые» подразделения и т. д.

АССОЦИАЦИИ

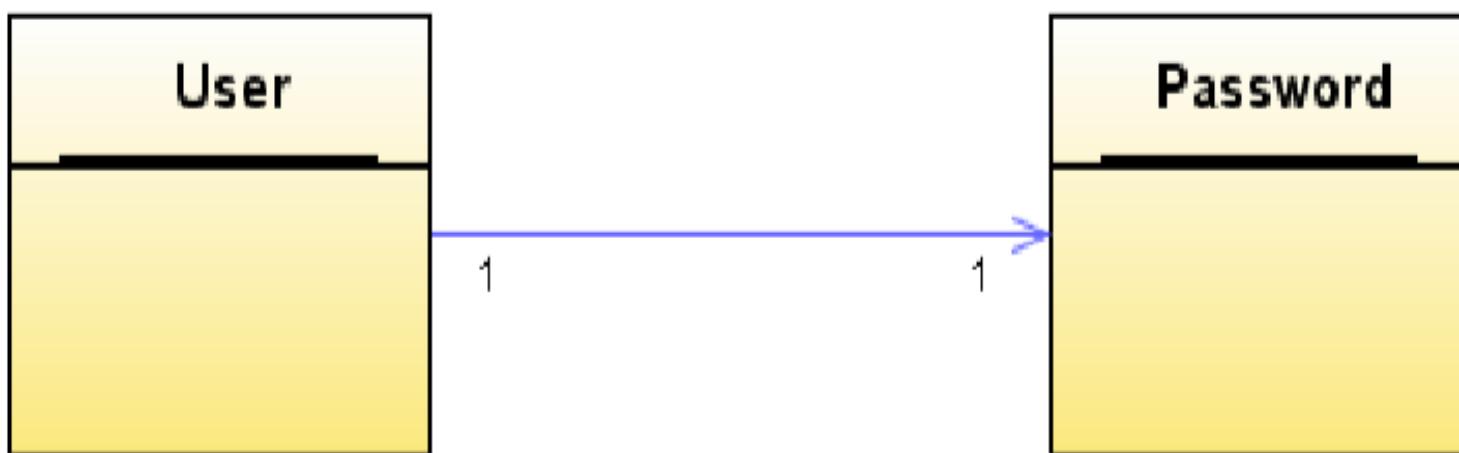
Направление навигации полюса ассоциации – это свойство полюса, имеющее значение типа Boolean, и определяющее, можно ли получить с помощью данной ассоциации доступ к объектам класса, присоединенному к данному полюсу ассоциации. По умолчанию это свойство имеет значение true, т. е. доступ возможен.

Если все полюса ассоциации (обычно их два) обеспечивают доступ, то это никак не отражается на диаграмме (потому, что данный случай наиболее распространенный и предполагается по умолчанию).

Если же навигация через некоторые полюса возможна, а через другие нет, то те полюса, через которые навигация возможна, отмечаются стрелками на концах линии ассоциации. Таким образом, если никаких стрелок не изображается, то это означает, что подразумеваются стрелки во всех возможных направлениях. Если же некоторые стрелки присутствуют, то это означает, что доступ возможен только в направлениях, указанных стрелками.

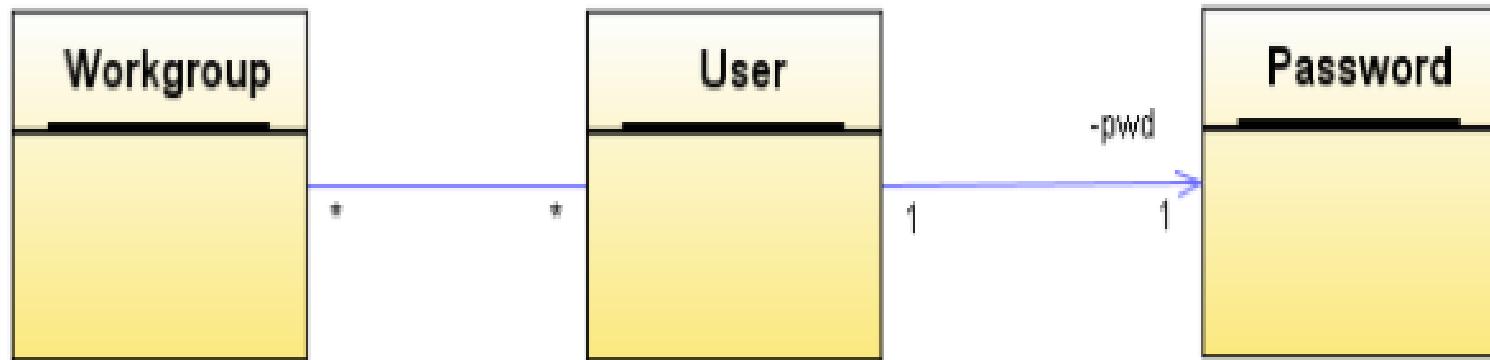
АССОЦИАЦИИ

Имеются два класса: **User** (содержит информацию о пользователе) и **Password** (содержит пароль — информацию, необходимую для аутентификации пользователя). Мы хотим отразить в модели следующую ситуацию: имеется взаимно однозначное соответствие между пользователями и паролями, зная пользователя можно получить доступ к его паролю, но обратное неверно: по имеющемуся паролю нельзя определить, кому он принадлежит.



АССОЦИАЦИИ

Видимость полюса ассоциации — это указание того, является ли классификатор присоединенный к данному полюсу ассоциации, видимым для других классификаторов вдоль данной ассоциации, помимо тех классификаторов, которые присоединены к другим полюсам ассоциации.



АССОЦИАЦИИ

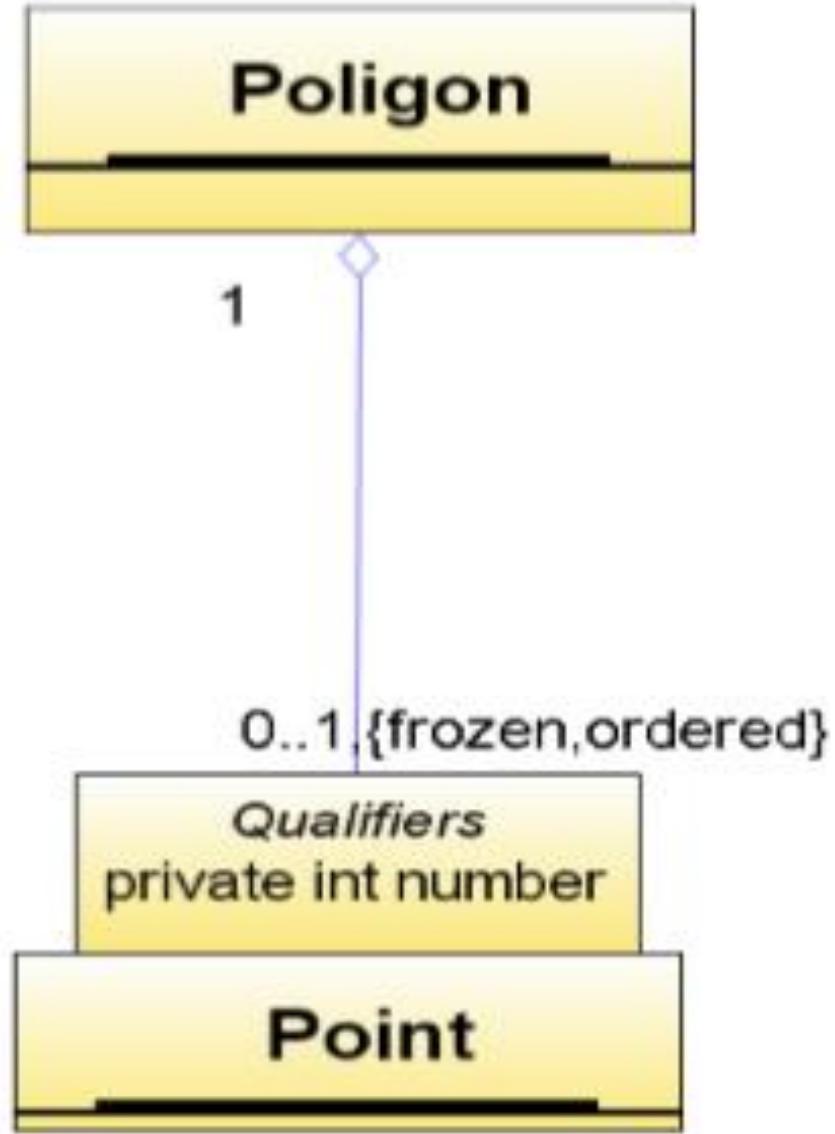
Квалификатор полюса ассоциации – это атрибут (или несколько атрибутов) ассоциации, значение которого (которых) позволяет выделить один (или несколько) объектов класса, присоединенного к данному полюсу ассоциации.

Квалификатор изображается в виде небольшого прямоугольника на полюсе ассоциации, примыкающего к прямоугольнику класса. Внутри этого прямоугольника (или рядом с ним) указываются имена и, возможно, типы атрибутов квалификатора. Описание квалифицирующего атрибута ассоциации имеет такой же синтаксис, что и описание обычного атрибута класса, только оно не может содержать начального значения.

Квалификатор может присутствовать только на том полюсе ассоциации, который имеет кратность "много", поэтому, если на полюсе ассоциации с квалификатором задана кратность, то она указывает не допустимую мощность множества объектов, присоединенных к полюсу связи, а допустимую мощность того подмножества, которое определяется при задании значений атрибутов квалификатора.

АССОЦИАЦИИ

Фрагмент модели для примера с многоугольниками, в котором использован квалификатор (в данном случае с именем number).



АССОЦИАЦИИ

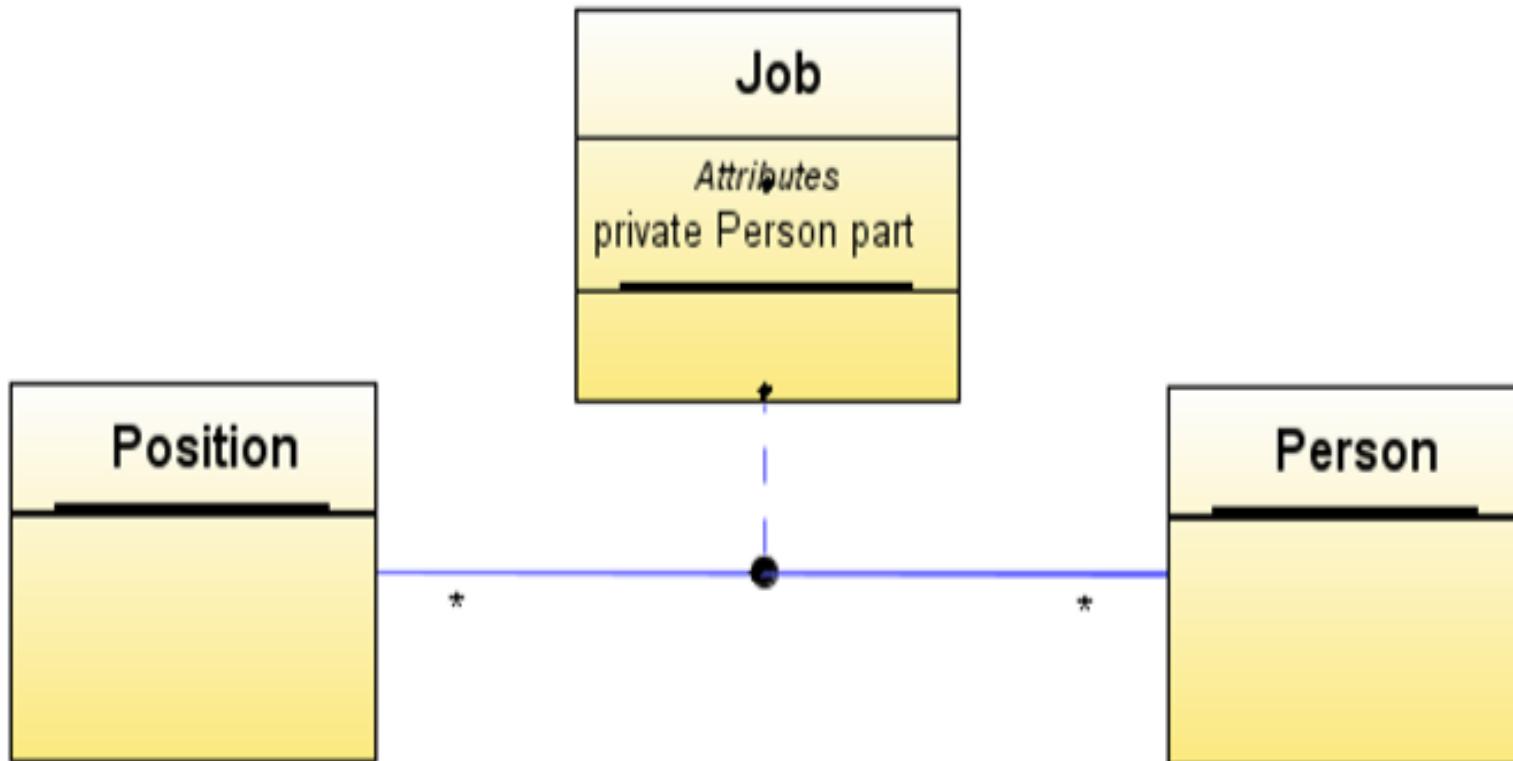
Ассоциация имеет экземпляры (связи), стало быть, является классификатором и может обладать соответствующими свойствами и составляющими классификатора.

В распространенных случаях ассоциации не обладают никакими собственными составляющими. Грубо говоря, ассоциация между классами А и В — это просто множество пар (a,b), где a — объект класса А, а b — объект класса В.

Подчеркнем еще раз, что это именно множество: двух одинаковых пар (a,b) быть не может. Однако возможны и более сложные ситуации, когда ассоциация имеет собственные атрибуты (и даже операции), значения которых хранятся в экземплярах ассоциации — связях.

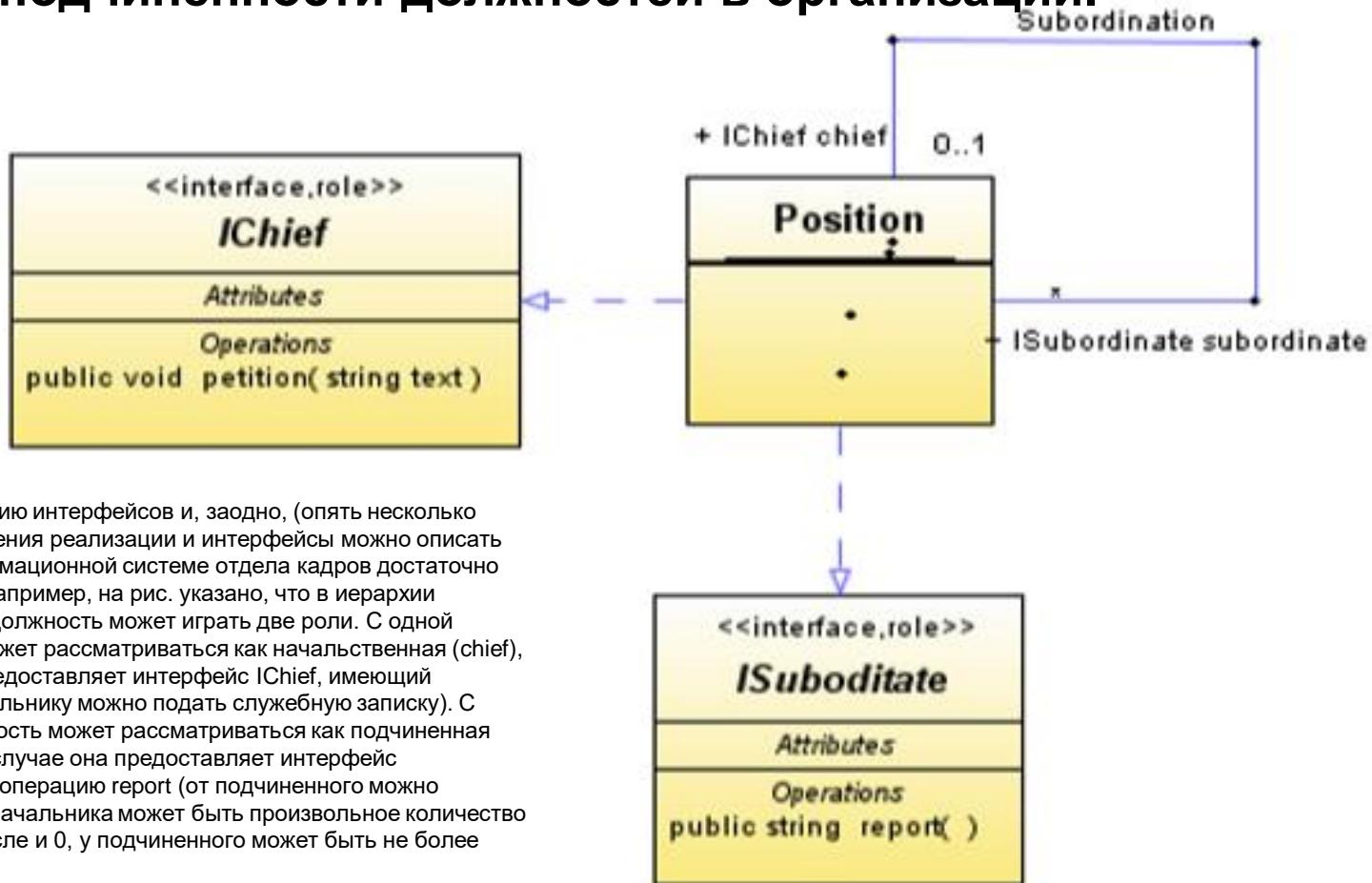
В таком случае применяется специальный элемент моделирования — класс ассоциации. Класс ассоциации — это сущность, которая имеет как свойства класса, так и свойства ассоциации. Класс ассоциации изображается в виде символа класса, присоединенного пунктирной линией к линии ассоциации.

АССОЦИАЦИИ



АССОЦИАЦИИ

На этом рисунке изображена ассоциация класса **Position** с самим собой. Эта ассоциация призвана отразить наличие иерархии подчиненности должностей в организации.



Используя спецификацию интерфейсов и, заодно, (опять несколько забегая вперед) отношения реализации и интерфейсы можно описать субординацию в информационной системе отдела кадров достаточно лаконично, но точно. Например, на рис. указано, что в иерархии субординации каждая должность может играть две роли. С одной стороны, должность может рассматриваться как начальственная (chief), и в этом случае она предоставляет интерфейс **IChief**, имеющий операцию **petition** (начальнику можно подать служебную записку). С другой стороны, должность может рассматриваться как подчиненная (subordinate), и в этом случае она предоставляет интерфейс **ISubordinate**, имеющий операцию **report** (от подчиненного можно потребовать отчет). У начальника может быть произвольное количество подчиненных, в том числе и 0, у подчиненного может быть не более одного начальника.

ОПЕРАЦИИ

Операция — это описание способа выполнить какие-то действия с объектом: изменить значения его атрибутов, вычислить новое значение по информации хранящейся в объекте и т. д. Выполнение действий, определяемых операцией, инициируется вызовом операции. При выполнении операция может, в свою очередь, вызывать операции этого и других классов. Описания операций класса перечисляются в разделе операций и имеют следующий синтаксис.

видимость ИМЯ (параметры): тип {свойства}

Здесь слово параметры обозначает последовательность описаний параметров операции, каждое из которых имеет вид следующий формат.

направление ПАРАМЕТР : тип = значение

ОПЕРАЦИИ

Видимость ИМЯ (параметры): тип {свойства}

Видимость, как обычно, обозначается с помощью знаков +, -, #.

Видимость можно обозначать с помощью ключевых слов private, public, protected.

Подчеркивание имени означает, что область действия операции — класс, а не объект. Например, конструкторы имеют область действия класс.

Курсивное написание имени означает, что операция абстрактная, т. е. в данном классе ее реализация не задана и должна быть задана в подклассах данного класса. После имени в скобках может быть указан список описаний параметров.

Описания параметров в списке разделяются запятой. Для каждого параметра обязательно указывается имя, а также могут быть указаны направление передачи параметра, его тип и значение аргумента по умолчанию.

ОПЕРАЦИИ

направление ПАРАМЕТР : тип = значение

Направление передачи параметра в UML описывает семантическое назначение параметров, не конкретизируя конкретный механизм передачи. Как именно следует трактовать указанные в модели направления передачи параметров зависит от используемой системы

Ключевое слово

Назначение параметра

In

Входной параметр — аргумент должен быть значением, которое используется в операции, но не изменяется

Out

Выходной параметр — аргумент должен быть хранилищем, в которое операция помещает значение

Inout

Входной и выходной параметр — аргумент должен быть хранилищем, содержащим значение. Операция использует переданное значение аргумента и помещает в хранилище результат

Return

Значение, возвращаемое операцией. Никакого аргумента не требуется

ОПЕРАЦИИ

Видимость ИМЯ (параметры): тип {свойства}

Типом параметра операции, равно как и тип возвращаемого операцией значения может быть любой встроенный тип или определенный в модели класс. Все вместе (имя операции, параметры и тип результата) обычно называют сигнатурой.

Операция имеет два важных свойства, которые указываются в списке свойств как именованные значения.

Во-первых, это `concurrency` – свойство, определяющее семантику одновременного (параллельного) вызова данной операции.

Во-вторых, операция имеет свойство `isQuery`, значение которого указывает, обладает ли операция побочным эффектом. Если значение данного свойства `true`, то выполнение операции не меняет состояния системы – операция только вычисляет значения, возвращаемые в точку вызова. В противном случае, т. е. при значение `false`, операция меняет состояние системы: присваивает новые значения атрибутам, создает или уничтожает объекты и т. п.

ЗНАЧЕНИЯ СВОЙСТВА CONCURRENCY

Значение	Описание
sequential	Операция не допускает параллельного вызова (не является повторно-входимой). Если параллельный вызов происходит, то дальнейшее поведение системы не определено.
Guarded	Параллельные вызовы допускаются, но только один из них выполняется — остальные блокируются и их выполнение задерживается до тех пор, пока не завершится выполнение данного вызова.
concurrent	Операция допускает произвольное число параллельных вызовов и гарантирует правильность своего выполнения. Такие операции называются повторно-входимыми (reenterable).

ПРИМЕРЫ ОПИСАНИЯ ОПЕРАЦИЙ

Пример

move

+move(in from, in to)

+move(in from : Dpt, in to : Dpt)

+getName() : String {isQuery}

+setPwd(in pwd : String = "password")

Пояснение

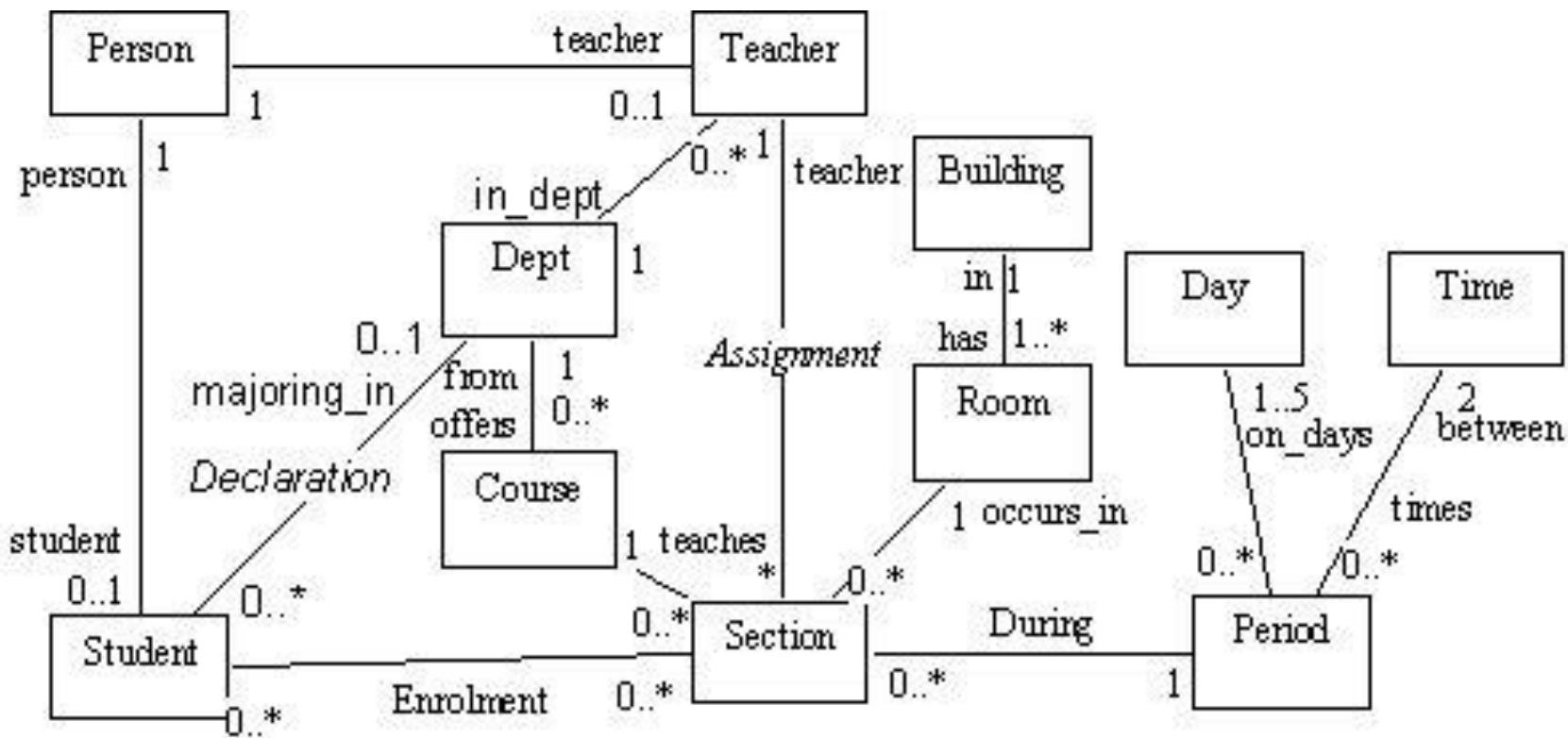
Минимальное возможное описание — указано только имя операции

Указаны видимость операции, направления передачи и имена параметров

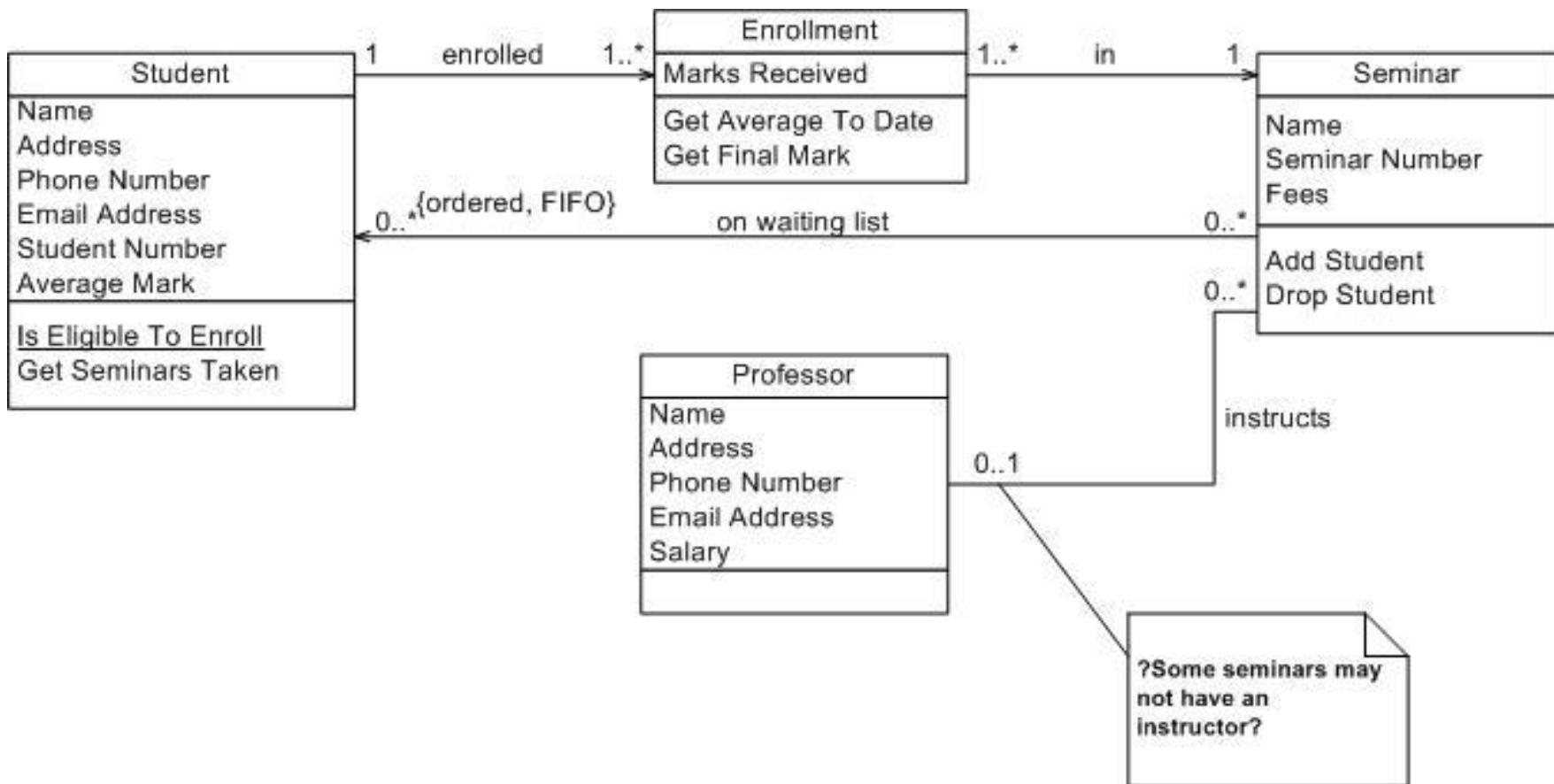
Подробное описание сигнатуры: указаны видимость операции, направления передачи, имена и типы параметров

Функция, возвращающая значение атрибута и не имеющая побочных эффектов

Процедура, для которой указано значение аргумента по умолчанию



Пример Диаграммы классов. Система классов для университета.



Пример Диаграммы классов

ИНТЕРФЕЙСЫ

Интерфейс – это именованный набор абстрактных операций. Другими словами, интерфейс – это абстрактный класс, в котором нет атрибутов и все операции абстрактны.

Поскольку интерфейс – это абстрактный класс, он не может иметь непосредственных экземпляров. Между интерфейсами и другими классификаторами, в частности классами, на диаграмме классов применяются два отношения:

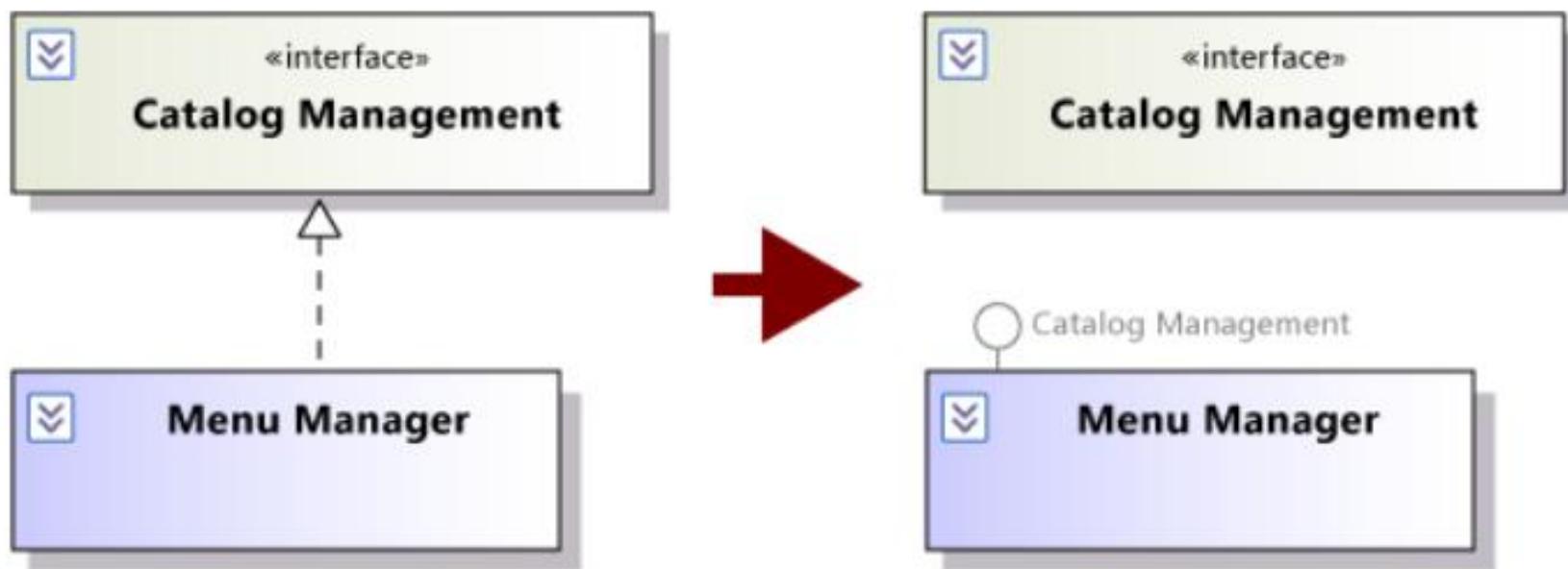
- классификатор (в частности, класс) использует интерфейс – это показывается с помощью зависимости со стереотипом «call»;
- классификатор (в частности, класс) реализует интерфейс – это показывается с помощью отношения реализации.

Никаких ограничений на использование отношения реализации не накладывается: класс может реализовывать много интерфейсов, и наоборот, интерфейс может быть реализован многими классами. Нет ограничений и на использование зависимостей со стереотипом «call» – класс может вызывать любые операции любых видимых интерфейсов.

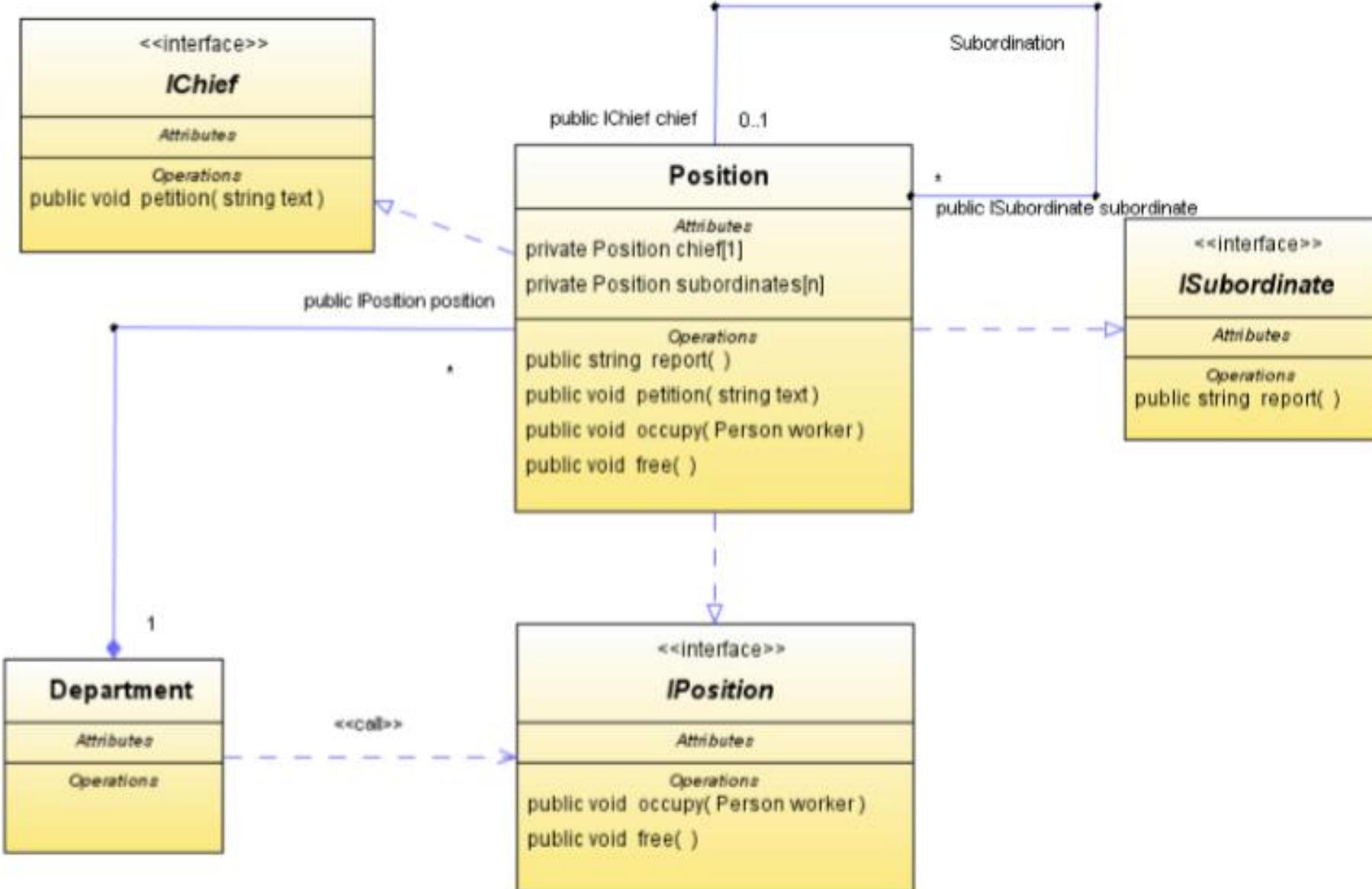
РЕАЛИЗАЦИЯ

Реализация означает, что класс реализует атрибуты и операции, заданные в интерфейсе.

Интерфейс находится на окончании соединителя с наконечником стрелки.



ИНТЕРФЕЙСЫ



ШАБЛОНЫ

Шаблон – это класс с параметрами. Параметром может быть любой элемент описания класса – тип составляющей, кратность атрибута и т. д. На диаграмме шаблон изображается с помощью прямоугольника класса, к которому в правом верхнем углу присоединен пунктирный прямоугольник с параметрами шаблона. Описания параметров перечисляются в этом прямоугольнике через запятую. Описание каждого параметра имеет вид:

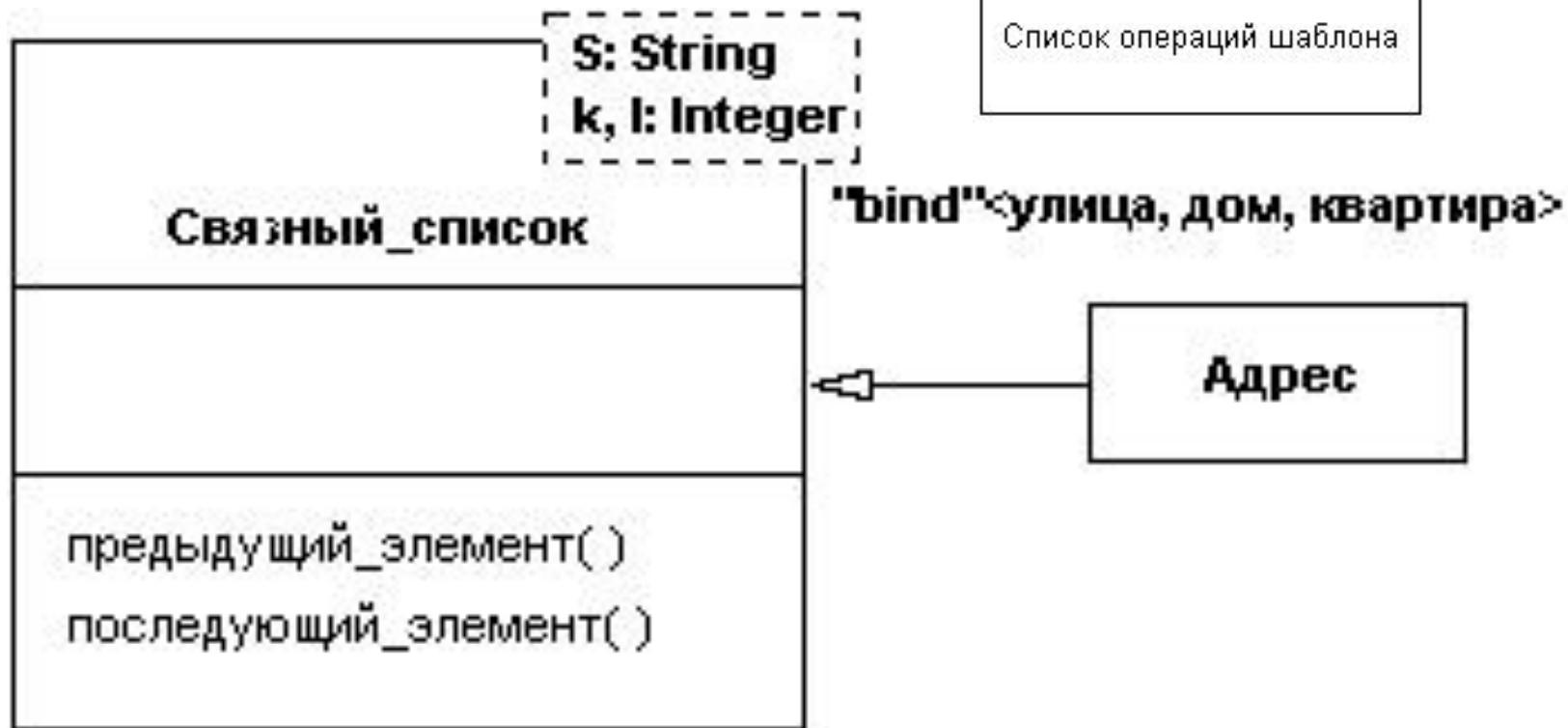
ИМЯ : ТИП

Сам по себе шаблон не может непосредственно использоваться в модели. Для того, чтобы на основе шаблона получить конкретный класс, который может использоваться в модели, нужно указать явные значения аргументов. Такое указание называется связыванием.

В UML применяются два способа связывания:

- явное связывание — зависимость со стереотипом «bind», в которой указаны значения аргументов;
- неявное связывание — определение класса, имя которого имеет формат **имя_шаблона < аргументы >**

ШАБЛОНЫ



В данном примере отмечен тот факт, что класс «Адрес» может быть получен из шаблона Связный_список на основе актуализации формальных параметров «S, k, l» фактическими атрибутами «улица, дом, квартира».

СПАСИБО ЗА ВНИМАНИЕ!

ПРОЕКТИРОВАНИЕ ПРОГРАММНЫХ СИСТЕМ

УНИФИЦИРОВАННЫЙ ЯЗЫК МОДЕЛИРОВАНИЯ UML

ЛЕКЦИЯ 5

UML (ПРОДОЛЖЕНИЕ)

ДИАГРАММА ОБЪЕКТОВ

КЛАССИФИКАЦИЯ ДИАГРАММ

Диаграммы поведения:

**Диаграммы вариантов
использования**

Диаграммы деятельности

Диаграммы состояний

**Диаграммы
взаимодействия:**

- Диаграммы последовательности
- Диаграммы коммуникации
- Диаграммы синхронизации (временные диаграммы)
- Обзорные диаграммы взаимодействия

Структурные диаграммы:

1. **Диаграммы классов**
2. **Диаграммы объектов**
3. **Диаграммы пакетов**
4. **Диаграммы компонентов**
5. **Диаграммы составной структуры**
6. **Диаграммы размещения**
7. **Диаграммы кооперации**
8. **Диаграммы профиля**

ДИАГРАММА ОБЪЕКТОВ

Диаграмма объектов – это частный случай диаграммы классов. Диаграммы объектов имеют вспомогательный характер – по сути это примеры, показывающие, какие имеются объекты и связи между ними в некоторый конкретный момент функционирования системы. На диаграмме объектов применяют один основной тип сущностей: объекты (экземпляры классов), между которыми указываются конкретные связи (экземпляры ассоциаций).

ДИАГРАММА ОБЪЕКТОВ

Имена объектов

Каждый отдельный объект представлен, например, прямоугольной формой, которая предоставляет имя через объект, а также подчеркнутый класс и разделяемый с помощью двоеточия.

ДИАГРАММА ОБЪЕКТОВ

Связи

Ссылки часто встречаются, связанные с отношениями. Вы можете нарисовать ссылку при использовании линий, примененных к диаграммам классов.

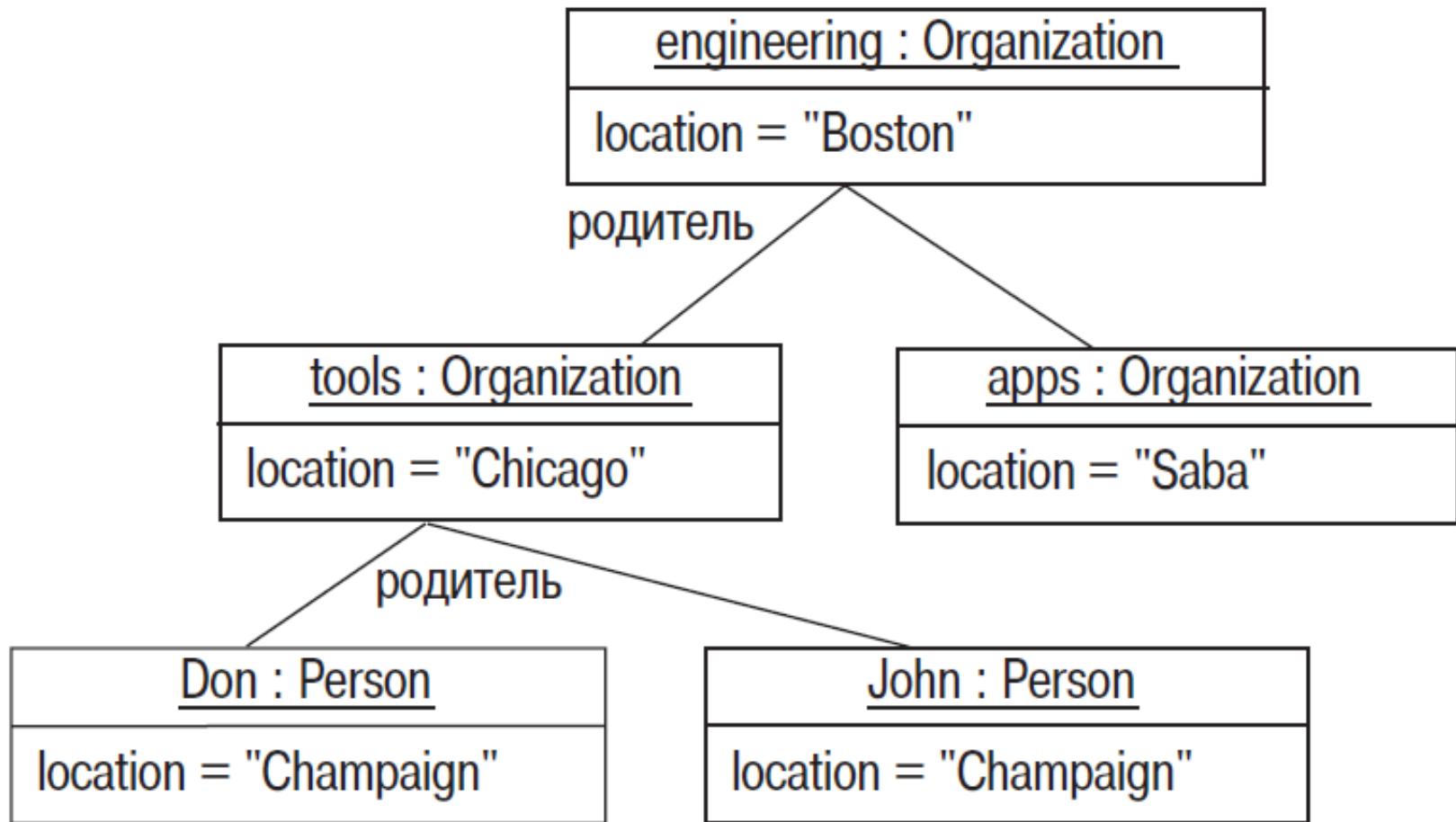


ДИАГРАММА ПАКЕТОВ (PACKAGE DIAGRAM)

ПАКЕТЫ (PACKAGES)

Визуализация, спецификация, конструирование и документирование больших систем предполагает работу с потенциально большим количеством классов, интерфейсов, узлов, компонентов, диаграмм и других элементов. Масштабируя такие системы, вы столкнетесь с необходимостью организовывать эти сущности в более крупные блоки.

В языке UML для организации моделирующих элементов в группы применяют пакеты (Packages).

ПАКЕТЫ (PACKAGES)

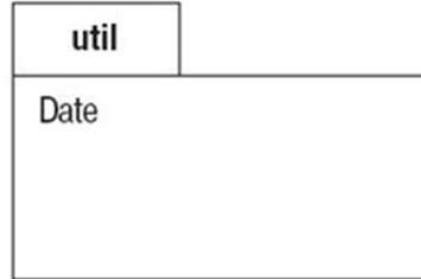
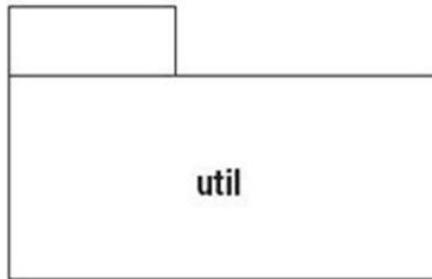
Пакет – это способ организации элементов модели в более крупные блоки, которыми впоследствии позволяет манипулировать как единым целым. Можно управлять видимостью входящих в пакет сущностей, так что некоторые будут видимы извне, а другие – нет. Кроме того, с помощью пакетов можно представлять различные виды архитектуры системы (различные виды разбиения системы на более мелкие части).

ПАКЕТЫ (PACKAGES)

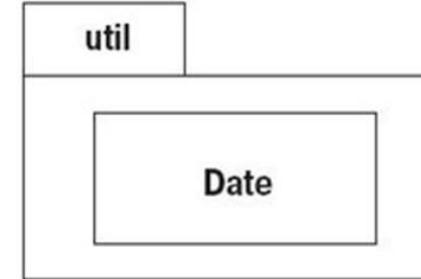
В языке UML организующие модель блоки называют пакетами. Пакет является универсальным механизмом организации элементов в группы, благодаря которому понимание модели упрощается. Пакеты позволяют также контролировать доступ к своему содержимому, что облегчает работу со стыковочными узлами в архитектуре системы.

Такая нотация позволяет визуализировать группы элементов, с которыми можно обращаться как с единым целым, контролируя при этом видимость и возможность доступа к отдельным элементам.

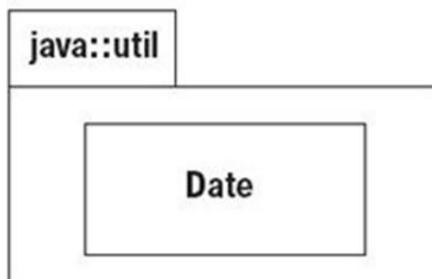
ГРАФИЧЕСКОЕ ПРЕДСТАВЛЕНИЕ ПАКЕТОВ В ЯЗЫКЕ UML:



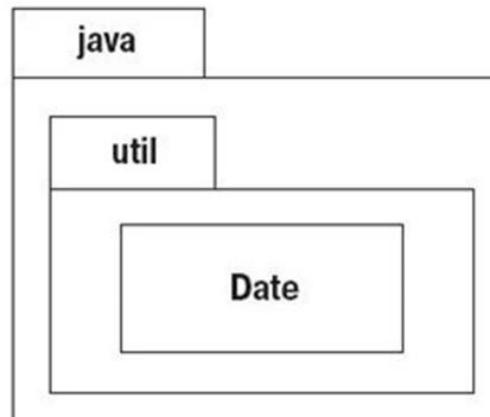
Содержимое, перечисленное в прямоугольнике



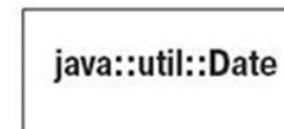
Содержимое в виде диаграммы в прямоугольнике



Полностью определенное имя пакета



Вложенные пакеты



Полностью определенное имя класса

... Способы изображения пакетов на диаграммах

ИМЕНА ПАКЕТОВ

У каждого пакета должно быть имя, отличающее его от других пакетов. Имя – это текстовая строка.

Взятое само по себе, оно называется простым.

К составному имени спереди добавлено имя объемлющего его пакета, если таковой существует.

Имя пакета должно быть **уникальным** в объемлющем пакете. Обычно при изображении компонента указывают только его имя, но, как и в случае с классами, вы можете дополнять пакеты помеченными значениями или дополнительными разделами, чтобы показать детали.



простые имена



расширенные пакеты

имя объемлющего пакета



ЭЛЕМЕНТЫ, ПРИНАДЛЕЖАЩИЕ ПАКЕТУ

Пакет может владеть другими элементами, в том числе классами, интерфейсами, компонентами, узлами, кооперациями, диаграммами и даже прочими пакетами.

Владение – это композитное отношение, означающее, что элемент объявлен внутри пакета.

Если пакет удаляется, то уничтожается и принадлежащий ему элемент.

Элемент может принадлежать только одному пакету.

ЭЛЕМЕНТЫ, ПРИНАДЛЕЖАЩИЕ ПАКЕТУ

Пакет определяет свое пространство имен; это значит, что элементы одного вида должны иметь имена, уникальные в контексте объемлющего пакета. Например, в одном пакете не может быть двух классов Queue, но может быть один класс Queue в пакете P1, а другой - в пакете P2. P1 :: Queue и P2 :: Queue имеют разные составные имена и поэтому являются различными классами.

Элементы различного вида могут иметь одинаковые имена в пределах пакета. Так, допустимо наличие класса Timer и компонента Timer. Однако на практике, во избежание недоразумений, лучше всем элементам давать уникальные имена в пакете.

ЭЛЕМЕНТЫ, ПРИНАДЛЕЖАЩИЕ ПАКЕТУ

Пакету могут принадлежать другие пакеты, а значит, позволительно осуществить иерархическую декомпозицию модели. Например, может существовать класс Camera, принадлежащий пакету Vision, который, в свою очередь, содержится в пакете Sensors.

Составное имя этого класса будет Sensors :: Vision :: Camera.

Лучше, однако, избегать слишком глубокой вложенности пакетов – два-три уровня являются пределом.

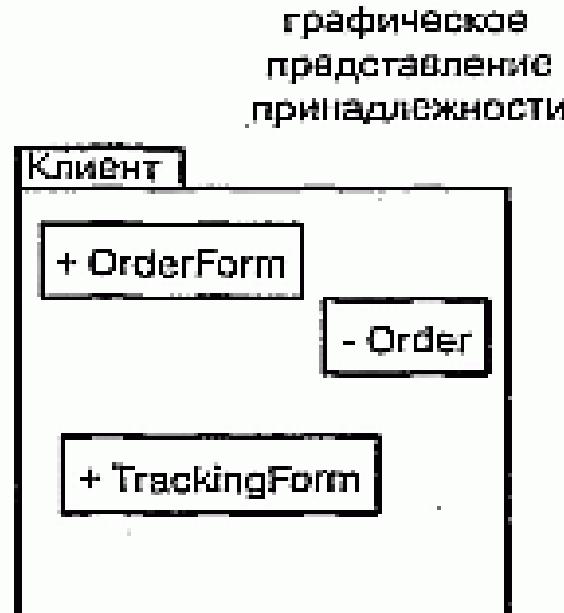
ЭЛЕМЕНТЫ, ПРИНАДЛЕЖАЩИЕ ПАКЕТУ

Пакеты позволяют контролировать элементы системы, даже если они разрабатываются в разном темпе.

Содержание пакета можно представить графически или в текстовом виде. Обратите внимание, что если вы изображаете принадлежащие пакету элементы, то имя пакета пишется внутри закладки



текстовое
представление
принадлежности



ВИДИМОСТЬ

Видимость принадлежащих пакету элементов можно контролировать точно так же, как видимость атрибутов и операций класса. По умолчанию такие элементы являются открытыми, то есть видимы для всех элементов, содержащихся в любом пакете, импортирующем данный. Защищенные элементы видимы только для потомков, а закрытые вообще невидимы вне своего пакета.

ВИДИМОСТЬ

Видимость принадлежащих пакету элементов обозначается с помощью символа видимости перед именем этого элемента. Для открытых элементов используется символ + (плюс), как и в случае с OrderForm. Все открытые части пакета составляют его интерфейс.

Аналогично классам для имен защищенных элементов используют символ # (диез), а для закрытых добавляют символ - (минус). Напомним, что защищенные элементы будут видны только для пакетов, наследующих данному, а закрытые вообще не видны вне пакета, в котором объявлены.

Примечание: Пакеты, связанные с некоторым пакетом отношениями зависимости со стереотипом friend, могут "видеть" все элементы данного пакета, независимо от объявленной для них видимости.

ВИДИМОСТЬ

Допустим, что класс А расположен в одном пакете, а класс В – в другом, причем оба пакета равноправны. Допустим также, что как А, так и В объявлены открытыми частями в своих пакетах.

Хотя оба класса объявлены открытыми, свободный доступ одного из них к другому невозможен, ибо границы пакетов непрозрачны. Однако если пакет, содержащий класс А, импортирует пакет-владелец класса В, то А сможет "видеть" В, хотя В по-прежнему не будет "видеть" А.

Импорт дает элементам одного пакета односторонний доступ к элементам другого. На языке UML отношения импорта моделируют как зависимости, дополненные стереотипом *import*.

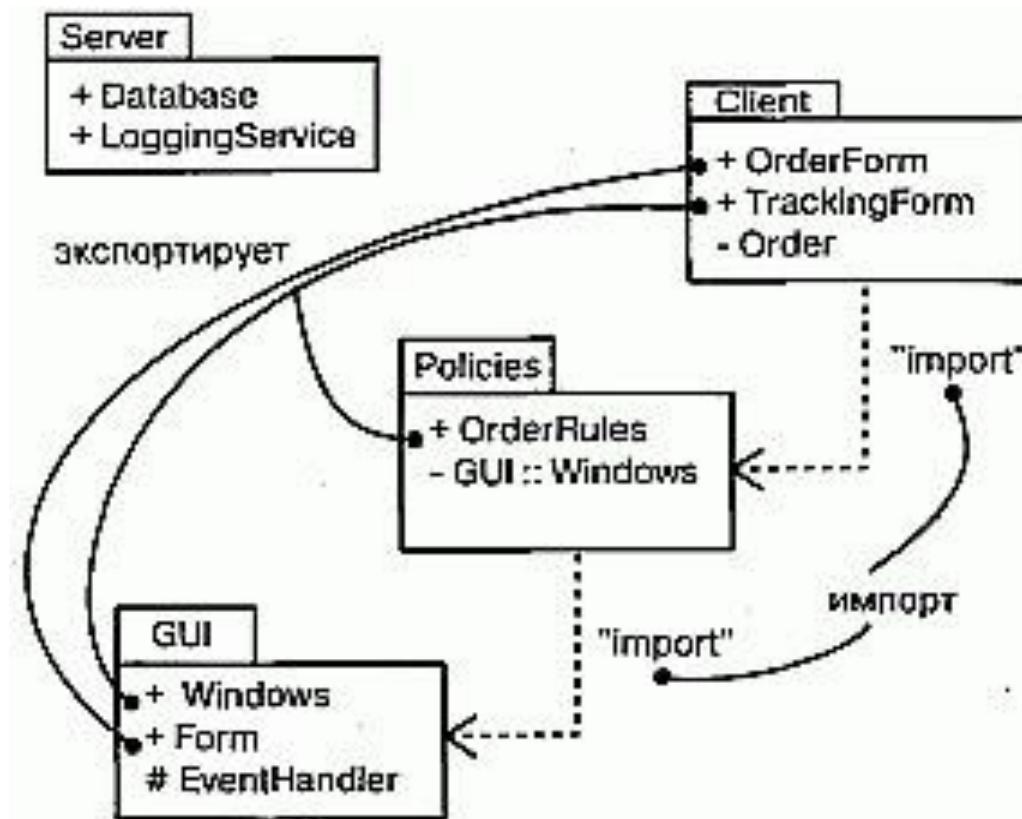
Упаковывая абстракции в семантически осмыслиенные блоки и контролируя доступ к ним с помощью импорта, вы можете управлять сложностью систем, насчитывающих множество абстракций.

ВИДИМОСТЬ

Открытые элементы пакета называются экспортируемыми.

Экспортируемые элементы будут видны только тем пакетам, которые явно импортируют данный.

Зависимости импорта и доступа не являются транзитивными.

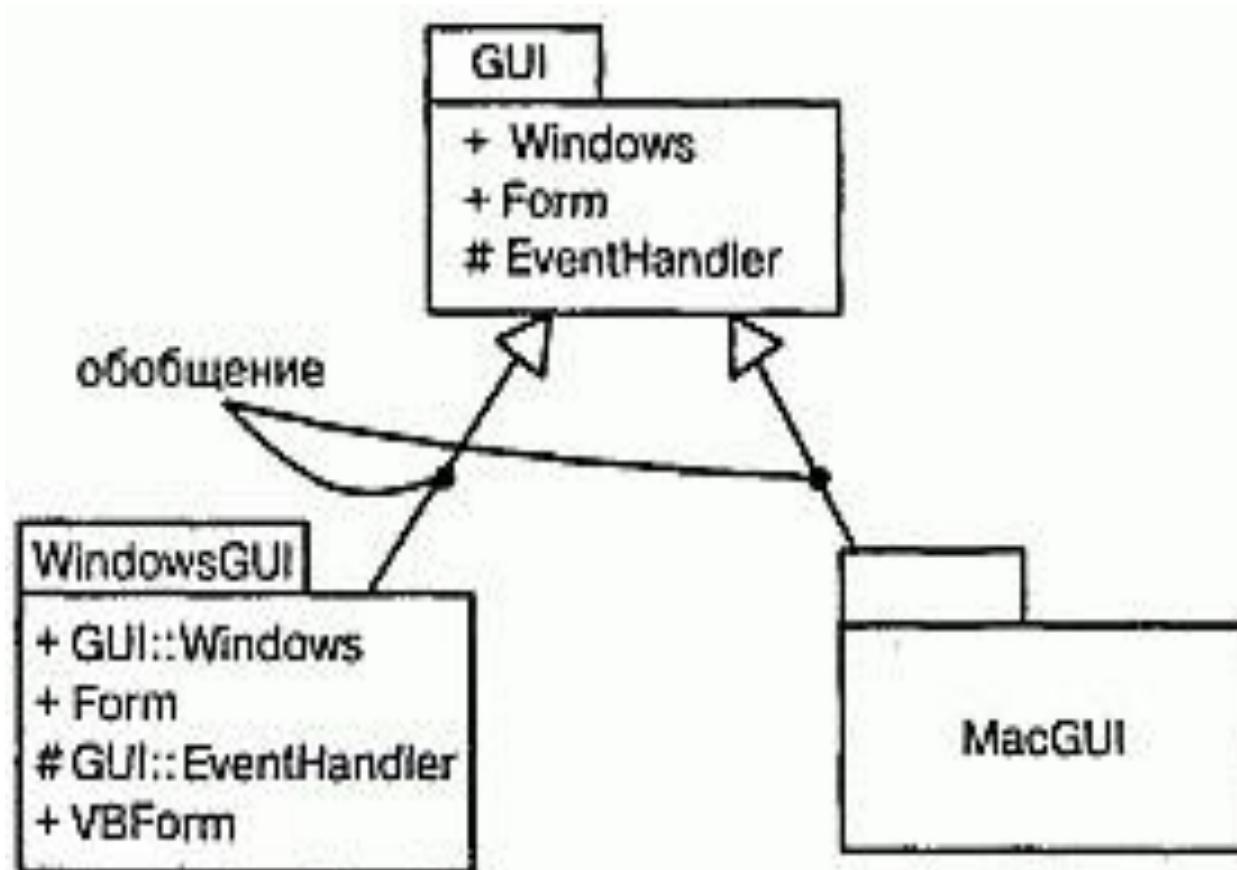


ОБОБЩЕНИЯ

Между пакетами определены два типа отношений: зависимости импорта и доступа, применяемые для импорта в пакет элементов, экспортируемых другим пакетом, и обобщения, используемые для спецификации семейств пакетов.

Отношения обобщения между пакетами очень похожи на отношения обобщения между классами.

ОБОБЩЕНИЯ



СТАНДАРТНЫЕ ЭЛЕМЕНТЫ

К пакетам применимы все механизмы расширения. Чаще всего используют помеченные значения для определения новых свойств (например, указания имени автора) и стереотипы для определения новых видов пакетов (например, пакетов, инкапсулирующих сервисы операционных систем).

В языке UML определены пять стандартных стереотипов, применимых к пакетам:

facade (фасад) – определяет пакет, являющийся всего лишь представлением какого-то другого пакета;

framework (каркас) – определяет пакет, состоящий в основном из образцов (паттернов);

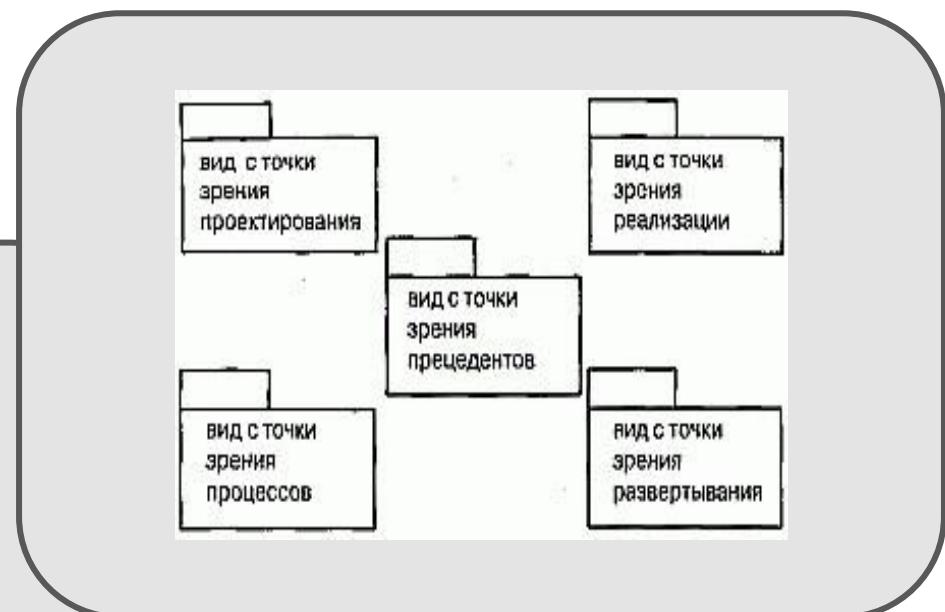
stub (стаб, заглушка) – определяет пакет, служащий заместителем открытого содержимого другого пакета;

subsystem (подсистема) – определяет пакет, представляющий независимую часть моделируемой системы;

system (система) – определяет пакет, представляющий всю моделируемую систему.

ТИПИЧНЫЕ ПРИЕМЫ МОДЕЛИРОВАНИЯ

Группы элементов Архитектурные виды



ГРУППЫ ЭЛЕМЕНТОВ

Чаще всего пакеты применяют для организации элементов моделирования в именованные группы, с которыми затем можно будет работать как с единым целым. Создавая простое приложение, можно вообще обойтись без пакетов, поскольку все ваши абстракции прекрасно разместятся в единственном пакете. В более сложных системах вы вскоре обнаружите, что многие классы, компоненты, узлы, интерфейсы и даже диаграммы естественным образом разделяются на группы. Эти группы и моделируют в виде пакетов.

ГРУППЫ ЭЛЕМЕНТОВ

Чаще всего с помощью пакетов элементы одинаковых типов организуют в группы. Например, классы и их отношения в представлении системы с точки зрения проектирования можно разбить на несколько пакетов и контролировать доступ к ним с помощью зависимостей импорта. Компоненты вида системы с точки зрения реализации допустимо организовать таким же образом.

Пакеты также применяются для группирования элементов различных типов. Например, если система создается несколькими коллективами разработчиков, расположенными в разных местах, то пакеты можно использовать для управления конфигурацией, размещая в них все классы и диаграммы, так чтобы члены разных коллективов могли независимо извлекать их из хранилища и помещать обратно.

На практике пакеты часто применяют для группирования элементов моделирования и ассоциированных с ними диаграмм.

ГРУППЫ ЭЛЕМЕНТОВ

Моделирование группы элементов производится так:

Просмотрите моделирующие элементы в некотором представлении архитектуры системы с целью поиска групп семантически или концептуально близких элементов.

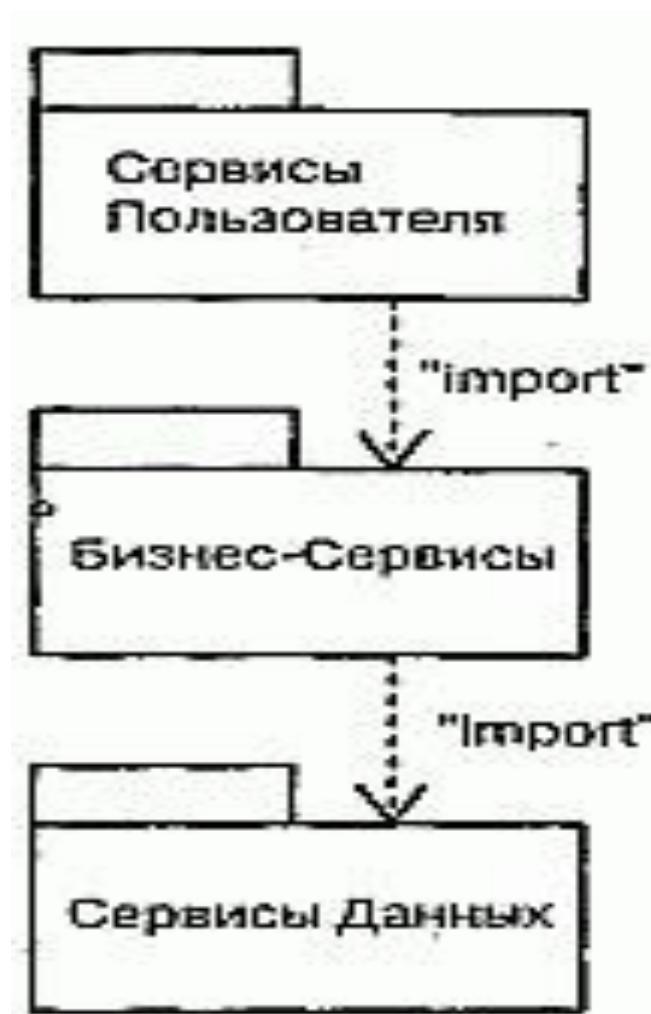
Поместите каждую такую группу в пакет.

Для каждого пакета определите, какие элементы должны быть доступны извне. Пометьте их как открытые, а остальные - как защищенные или закрытые. Если сомневаетесь, скройте элемент.

Явно соедините пакеты отношениями импорта с теми пакетами, от которых они зависят.

Если имеются в наличии семейства пакетов, соедините специализированные пакеты с общими отношениями обобщения.

ГРУППЫ ЭЛЕМЕНТОВ



АРХИТЕКТУРНЫЕ ВИДЫ

При рассмотрении архитектурных видов программных систем возникает потребность в еще более крупных блоках. Архитектурные виды тоже можно моделировать с помощью пакетов.

Видом или представлением называется проекция организации и структуры системы, в которой внимание акцентируется на одном из конкретных аспектов этой системы

Из этого определения вытекают два следствия. Во-первых, систему можно разложить на почти ортогональные пакеты, каждый из которых имеет дело с набором архитектурно значимых решений (например, можно создать виды с точки зрения проектирования, процессов, реализации, развертывания и прецедентов). Во-вторых, этим пакетам будут принадлежать все абстракции, относящиеся к данному виду. Так, все компоненты модели принадлежат пакету, который представляет вид системы с точки зрения реализации.

АРХИТЕКТУРНЫЕ ВИДЫ

Моделирование архитектурных видов осуществляется следующим образом:

Определите, какие виды важны для решения вашей проблемы. Обычно это виды с точки зрения проектирования, процессов, реализации, развертывания и прецедентов.

Поместите в соответствующие пакеты элементы и диаграммы, необходимые и достаточные для визуализации, специфирования, конструирования, документирования семантики каждого вида.

При необходимости сгруппируйте элементы каждого вида в более мелкие пакеты.

Между элементами из различных видов, скорее всего, будут существовать отношения зависимости, поэтому в общем случае стоит открыть все виды на верхнем уровне системы для всех остальных видов того же уровня.

АРХИТЕКТУРНЫЕ ВИДЫ

Каноническая декомпозиция верхнего уровня, пригодная даже для самых сложных систем

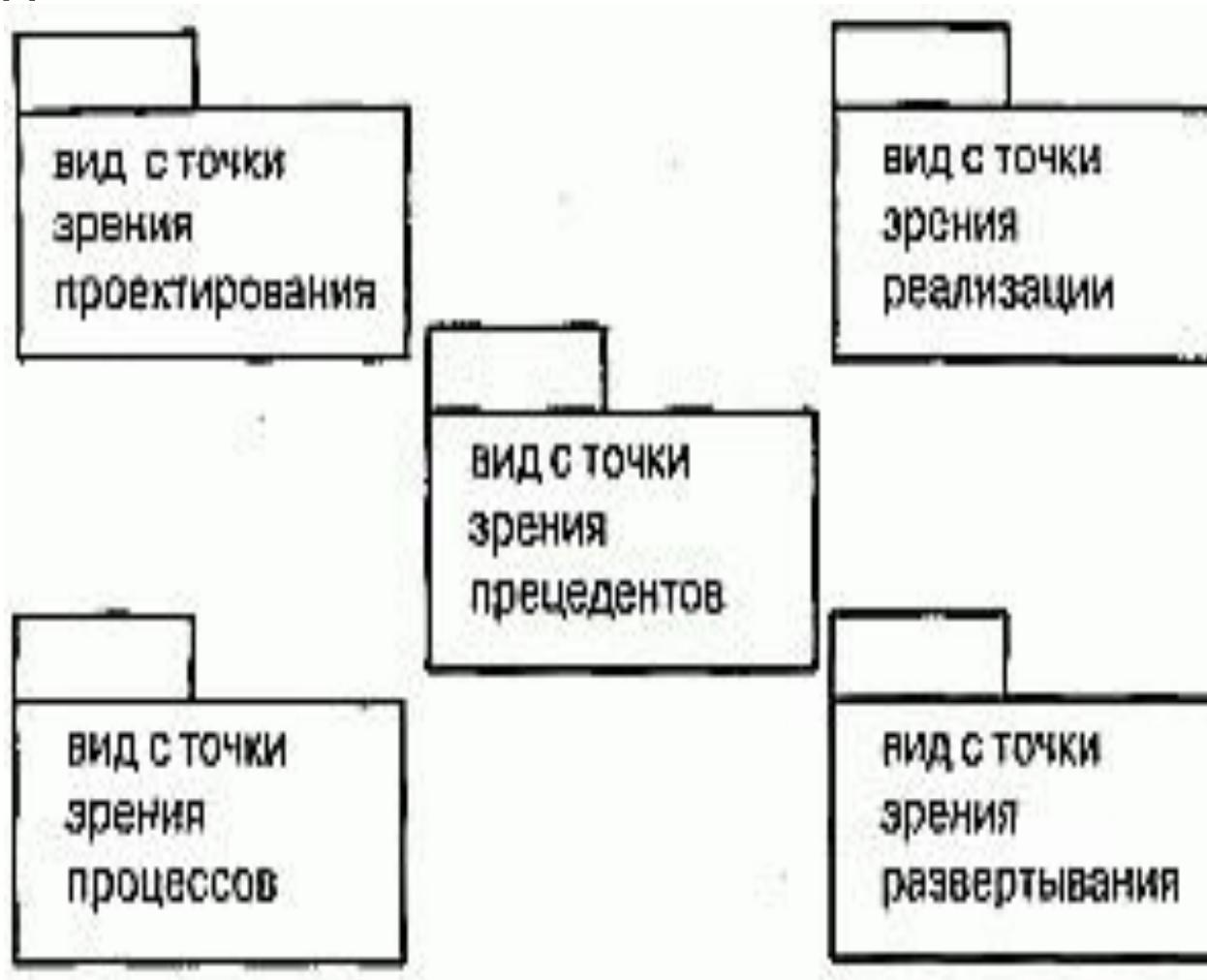


ДИАГРАММА ПАКЕТОВ

Моделируя пакеты в UML нужно помнить, что они нужны только для организации элементов модели. Если имеются абстракции, непосредственно материализуемые как объект в системе, не пользуйтесь пакетами. Вместо этого применяйте такие элементы моделирования, как классы или компоненты.

Хорошо структурированный пакет характеризуется следующими свойствами:

он внутренне согласован и очерчивает четкую границу вокруг группы родственных элементов;

он слабо связан и экспортирует в другие пакеты только те элементы, которые они действительно должны "видеть", а импортирует лишь элементы, которые необходимы и достаточны для того, чтобы его собственные элементы могли работать;

глубина вложенности пакета невелика, поскольку человек не способен воспринимать слишком глубоко вложенные структуры;

владея сбалансированным набором элементов, пакет по отношению к другим пакетам в системе не должен быть ни слишком большим (если надо, расщепляйте его на более мелкие), ни слишком маленьким (объединяйте элементы, которыми можно манипулировать как единым целым).

Изображая пакет в UML, руководствуйтесь следующими принципами:

применяйте простую форму пиктограммы пакета, если не требуется явно раскрыть его содержимое;

раскрывая содержимое пакета, показывайте только те элементы, которые абсолютно необходимы для понимания его назначения в данном контексте;

моделируя с помощью пакетов сущности, относящиеся к управлению конфигурацией, раскрывайте значения меток, связанных с номерами версий.

ДИАГРАММЫ ВЗАЙМОДЕЙСТВИЯ (INTERACTION DIAGRAM)

Диаграммы взаимодействия описывают поведение совместно действующих групп объектов в рамках потока событий.

Сообщение – это средство, с помощью которого объект-отправитель запрашивает у объекта-получателя выполнение одной из его операций.

ВИДЫ ДИАГРАММ ВЗАИМОДЕЙСТВИЯ

- Диаграммы последовательности,
- Коммуникационные диаграммы,
- Обзорные диаграммы взаимодействия,
- Временные диаграммы (диаграммы синхронизации)

ДИАГРАММЫ ПОСЛЕДОВАТЕЛЬНОСТИ (SEQUENCE DIAGRAM)

Диаграммы последовательности отражают временную последовательность событий, происходящих в рамках варианта использования.

Более детально описывают логику сценариев использования. Это отличное средство документирования проекта с точки зрения сценариев использования!

Аттестационная
комиссия

Математик

Экономист

Юрист

Сколько будет 2×2 ?

Четыре

Сколько будет 2×2 ?

Подумал 5 минут

Что-то в пределах от 3-х до 5-ти

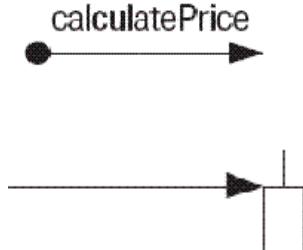
Сколько будет 2×2 ?

Столько, сколько нужно

Пример диаграммы

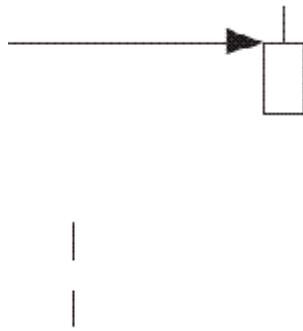
ЭЛЕМЕНТЫ ДИАГРАММЫ ПОСЛЕДОВАТЕЛЬНОСТИ

Найденное сообщение



У первого сообщения нет участника, пославшего его, поскольку оно приходит из неизвестного источника. Оно называется найденным сообщением (found message).

Сообщение



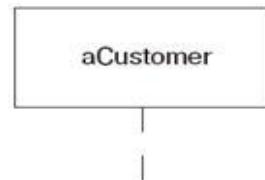
Команда, отправляемая другому участнику. Может содержать только передаваемые данные.

Линия жизни



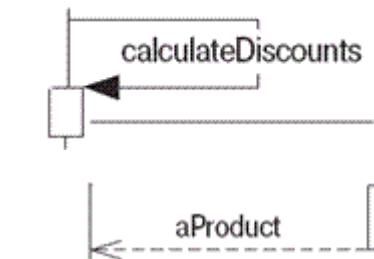
Каждая линия жизни имеет полосу активности, которая показывает интервал активности участника при взаимодействии. Она соответствует времени нахождения в стеке одного из методов участника.

Участник



В большинстве случаев можно считать участников диаграммы взаимодействия объектами, как это и было в действительности в UML 1. Но в UML 2 их роль значительно сложнее. Поэтому здесь употребляется термин участники (participants), который формально не входит в спецификацию UML.

Самовызов



Участник отправляет сообщение (команду) самому себе.

Возврат



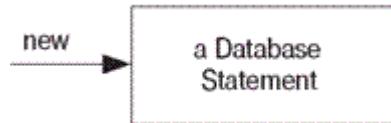
Передача управления обратно участнику, который до этого инициировал сообщение.

ЭЛЕМЕНТЫ ДИАГРАММЫ ПОСЛЕДОВАТЕЛЬНОСТИ

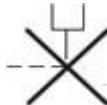
Активация



Создание



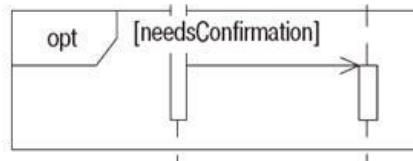
Самоудаление



Удаление из другого объекта



фрейм



На изображении это — белый вертикальный прямоугольник. Момент, когда участник начинает действовать в ответ на принятое сообщение.

В случае создания участника надо нарисовать стрелку сообщения, направленную к прямоугольнику участника. Если применяется конструктор, то имя сообщения не обязательно, но можно маркировать его словом «*new*» в любом случае. Если участник выполняет что-нибудь непосредственно после создания, например команду запроса, то надо начать активацию сразу после прямоугольника участника.

Удаление участника обозначается большим крестом (X). X в конце линии жизни показывает, что участник удаляет сам себя.

Удаление участника обозначается большим крестом (X). Стрелка сообщения, идущая в X, означает, что один участник явным образом удаляет другого.

ФОРМАТЫ СООБЩЕНИЙ

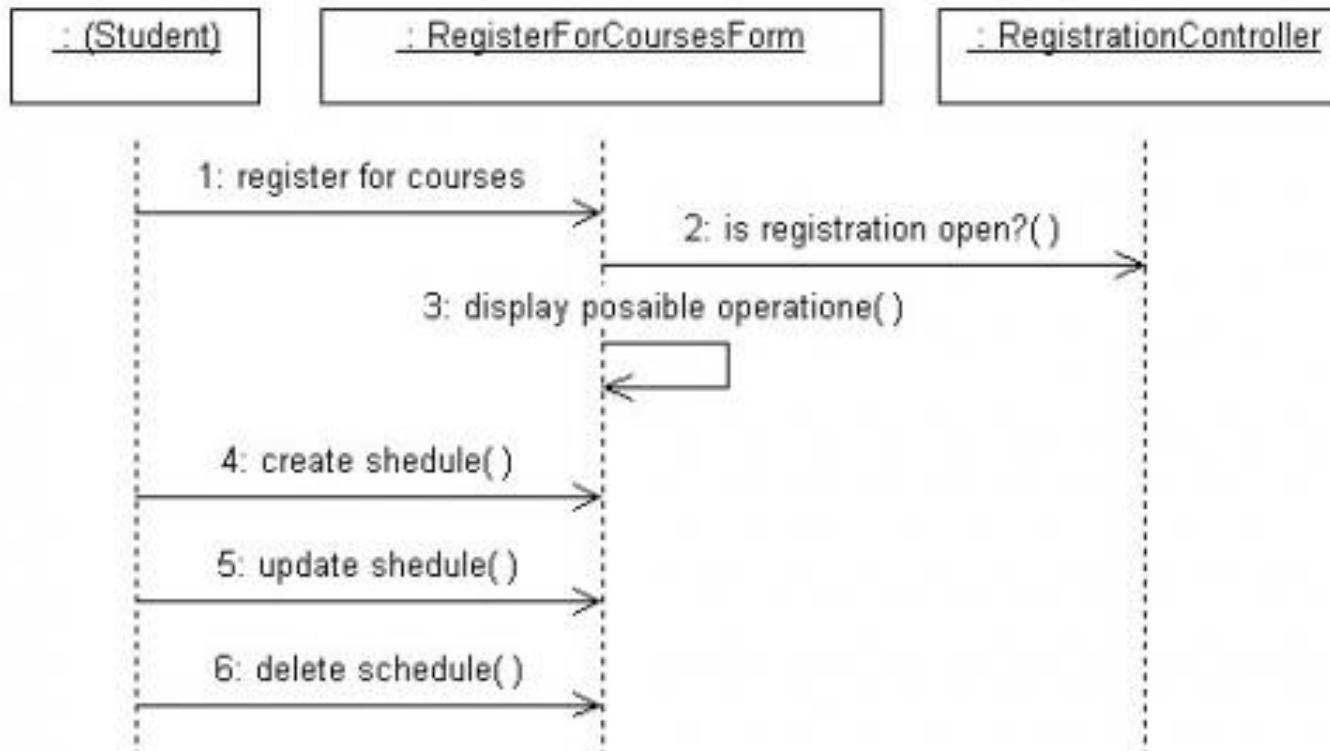
Типы сообщений:

- **Синхронное** - клиент посылает сообщение серверу и ждет, пока тот примет и обработает сообщение . Обозначается: →
- **Асинхронное** - клиент посылает сообщение серверу и, не дожидаясь ответа, продолжает выполнять следующие операции. Обозначается: →
- **Возврат**- обозначающее возврат значения или управления от сервера обратно клиенту. Стрелки этого вида зачастую отсутствуют на диаграммах, поскольку неявно предполагается их существование после окончания процесса выполнения операции. Обозначается: <--

ФОРМАТЫ СООБЩЕНИЙ

Структура сообщения:

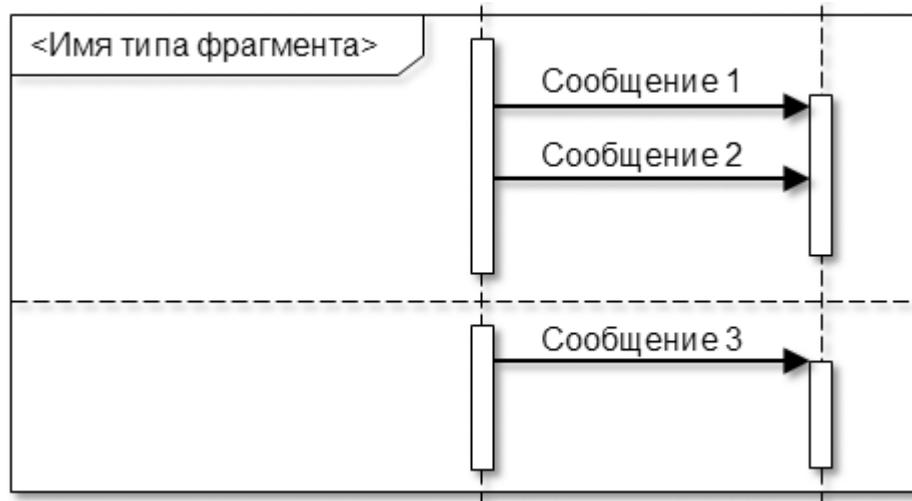
Номер : [переменная =] имя([список параметров]) [:возвращаемое значение].



ФРЕЙМЫ ВЗАИМОДЕЙСТВИЯ

В основном фреймы состоят из некоторой области диаграммы последовательности, разделенной на несколько фрагментов.

Каждый фрейм имеет оператор, а каждый фрагмент может иметь защиту.



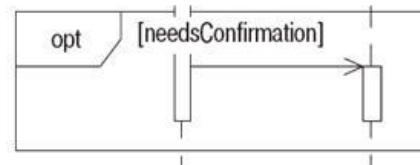
ЭЛЕМЕНТЫ ДИАГРАММЫ ПОСЛЕДОВАТЕЛЬНОСТИ. ФРЕЙМЫ ВЗАИМОДЕЙСТВИЯ

Общая проблема диаграмм последовательности заключается в том, как отображать циклы и условные конструкции.

Прежде всего надо усвоить, что диаграмма последовательности для этого не предназначена. Подобные управляющие структуры лучше показывать с помощью диаграммы деятельности или собственно кода.

Диаграмма последовательности применяется для визуализации процесса взаимодействия объектов, а не как средство моделирования алгоритма управления.

Как было сказано, существуют дополнительные обозначения. И для циклов, и для условий используются фреймы взаимодействий (inter action frames), представляющие собой средство разметки диаграммы взаимодействия.



ОПЕРАТОРЫ ВЗАИМОДЕЙСТВИЯ

Отдельные фрагменты диаграммы взаимодействия можно выделить с помощью фрейма. Фрейм должен содержать метку оператора взаимодействия. UML содержит следующие операнды:

Alt - Несколько альтернативных фрагментов (alternative); выполняется только тот фрагмент, условие которого истинно

Opt - Необязательный (optional) фрагмент; выполняется, только если условие истинно. Эквивалентно **alt** с одной веткой

Par - Параллельный (parallel); все фрагменты выполняются параллельно

loop - Цикл (loop); фрагмент может выполняться несколько раз, а защищает тело итерации

region - Критическая область (critical region); фрагмент может иметь только один поток, выполняющийся за один прием

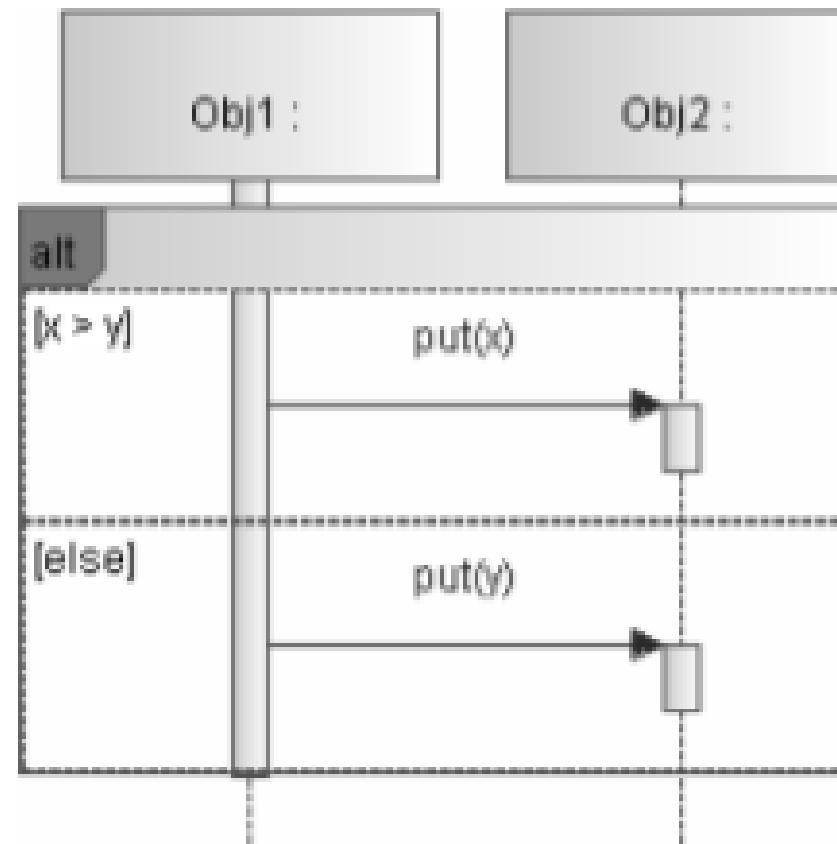
Neg - Отрицательный (negative) фрагмент; обозначает неверное взаимодействие

ref - Ссылка (reference); ссылается на взаимодействие, определенное на другой диаграмме. Фрейм рисуется, чтобы охватить линии жизни, вовлеченные во взаимодействие. Можно определять параметры и возвращать значение

Sd - Диаграмма последовательности (sequence diagram); используется для очерчивания всей диаграммы последовательности, если это необходимо.

ОПЕРАТОРЫ ВЗАИМОДЕЙСТВИЯ

alt



ОПЕРАТОРЫ ВЗАИМОДЕЙСТВИЯ

Loop - цикл

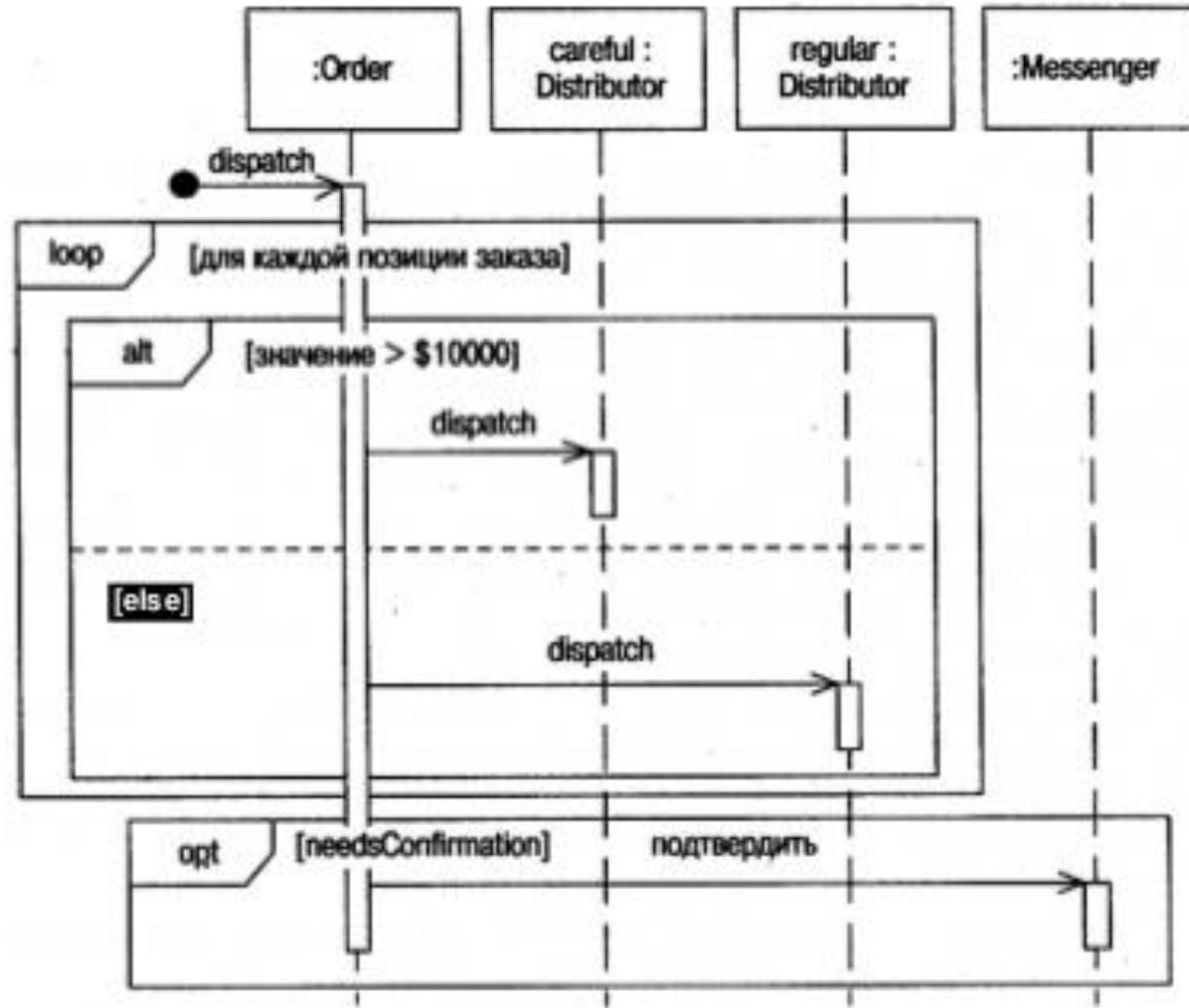
Alt - выполняется

только тот фрагмент,
условие которого

Истинно

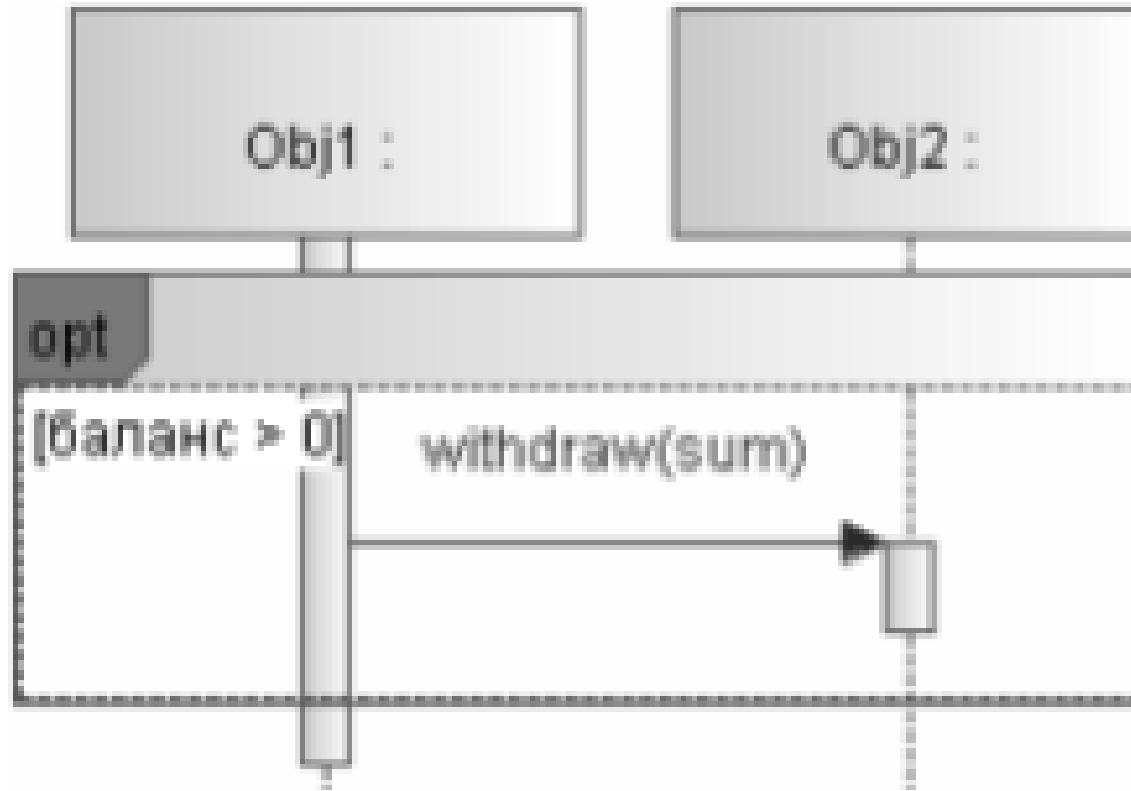
Opt -

Необязательный
(optional) фрагмент;
выполняется,
только если
условие истинно.
Эквивалентно alt с
одной веткой



ОПЕРАТОРЫ ВЗАИМОДЕЙСТВИЯ

Opt - Необязательный (optional) фрагмент; выполняется, только если условие истинно. Эквивалентно **alt** с одной веткой

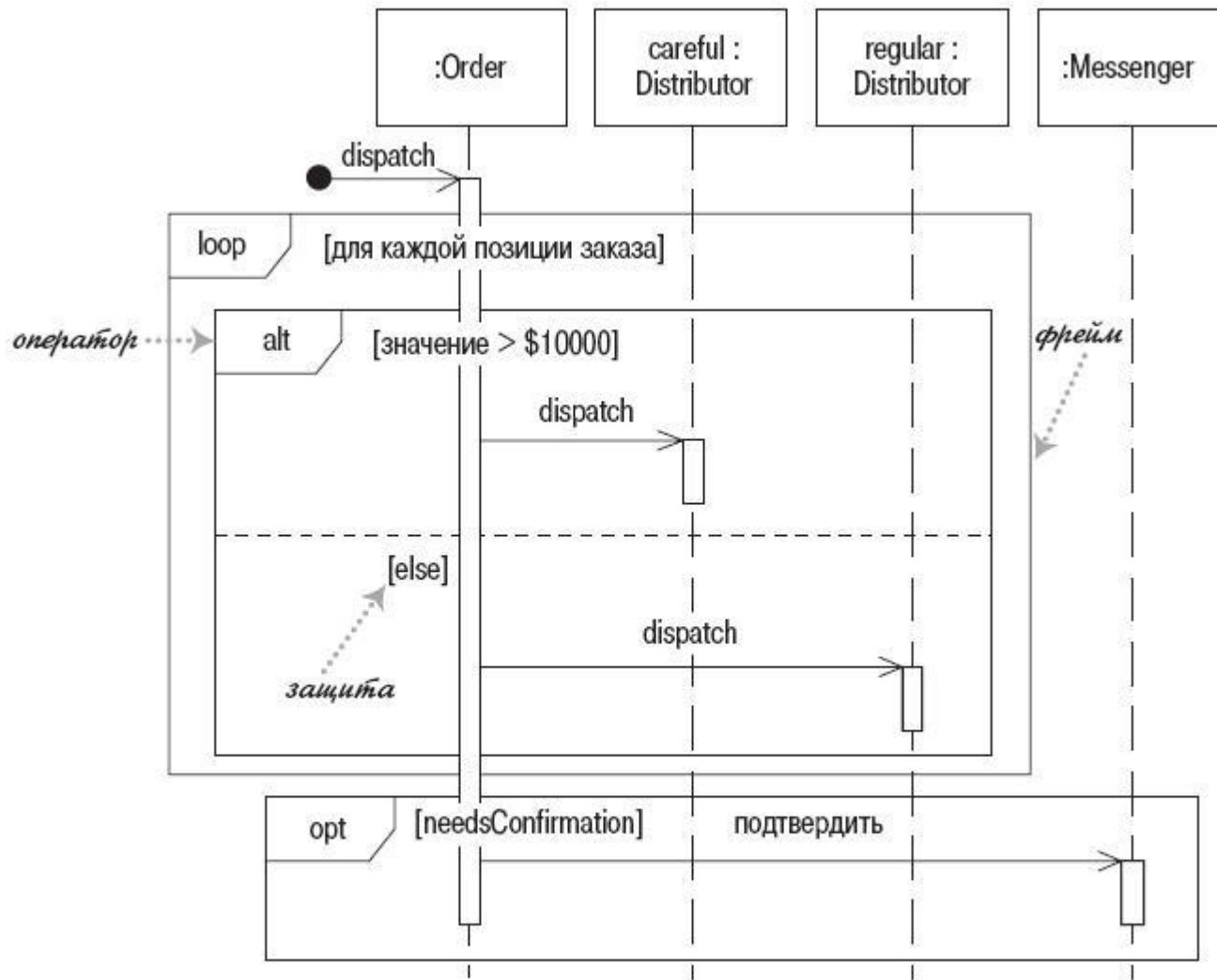


ОПЕРАТОРЫ ВЗАИМОДЕЙСТВИЯ

Par - Параллельный (parallel); все фрагменты выполняются параллельно



ОПЕРАТОРЫ ВЗАИМОДЕЙСТВИЯ



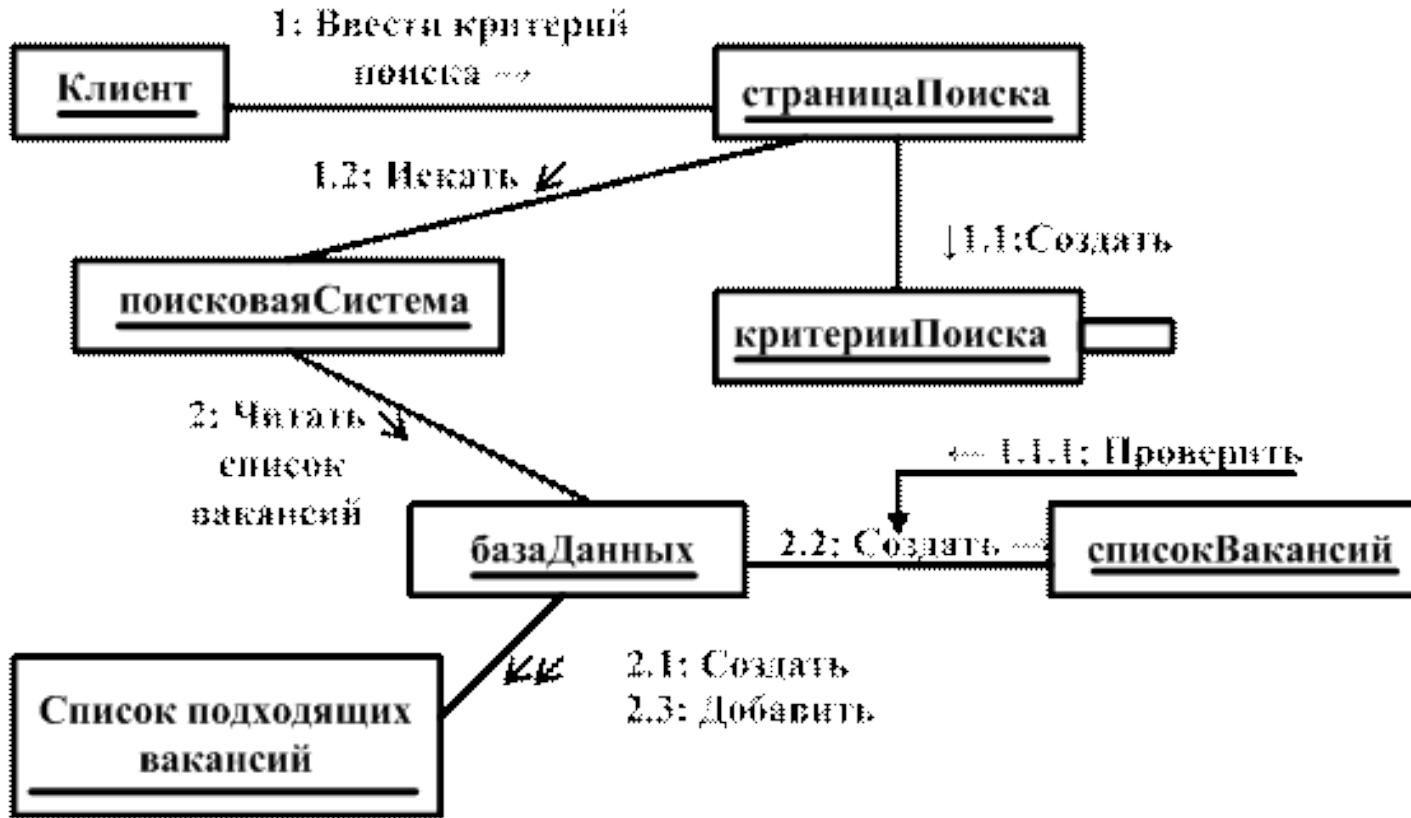
Фреймы взаимодействия

КОММУНИКАЦИОННЫЕ ДИАГРАММЫ (COMMUNICATION DIAGRAM)

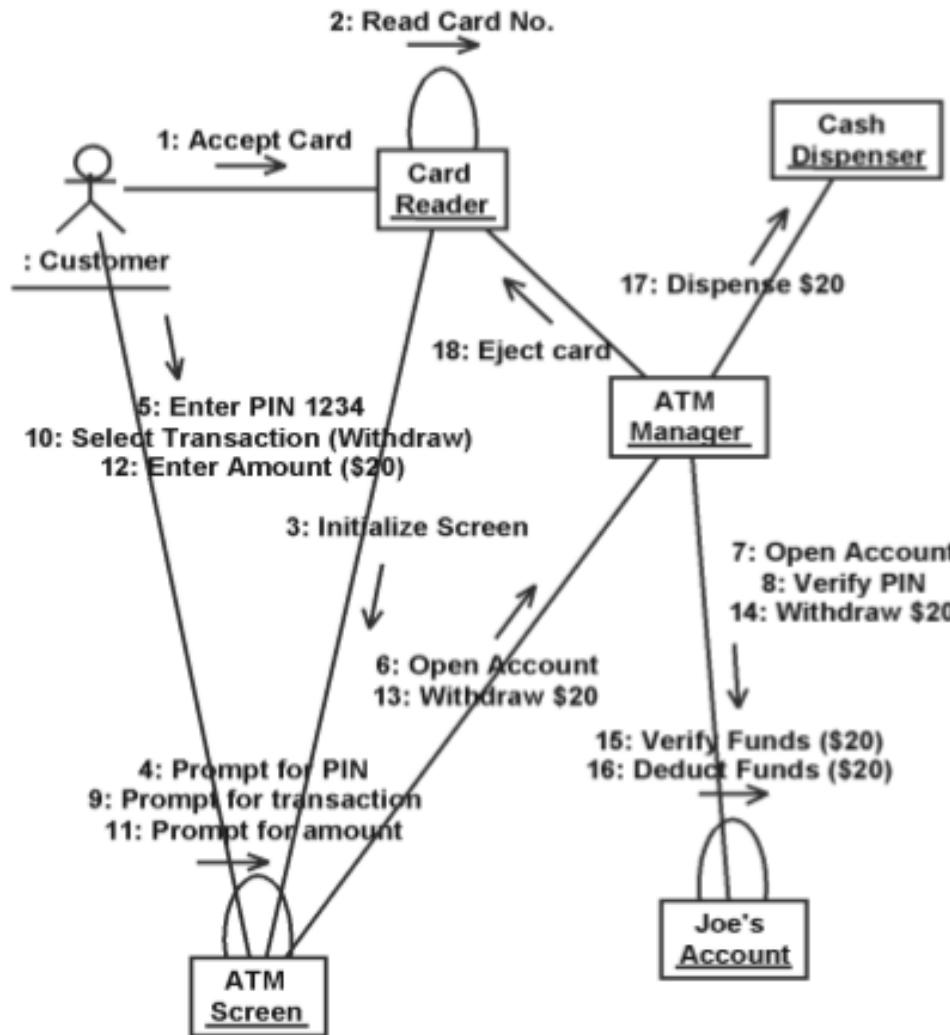
Коммуникационные диаграммы (*communication diagrams*) – это особый вид диаграмм взаимодействия, акцентированных на обмене данными между различными участниками взаимодействия.

КОММУНИКАЦИОННЫЕ ДИАГРАММЫ (COMMUNICATION DIAGRAM)

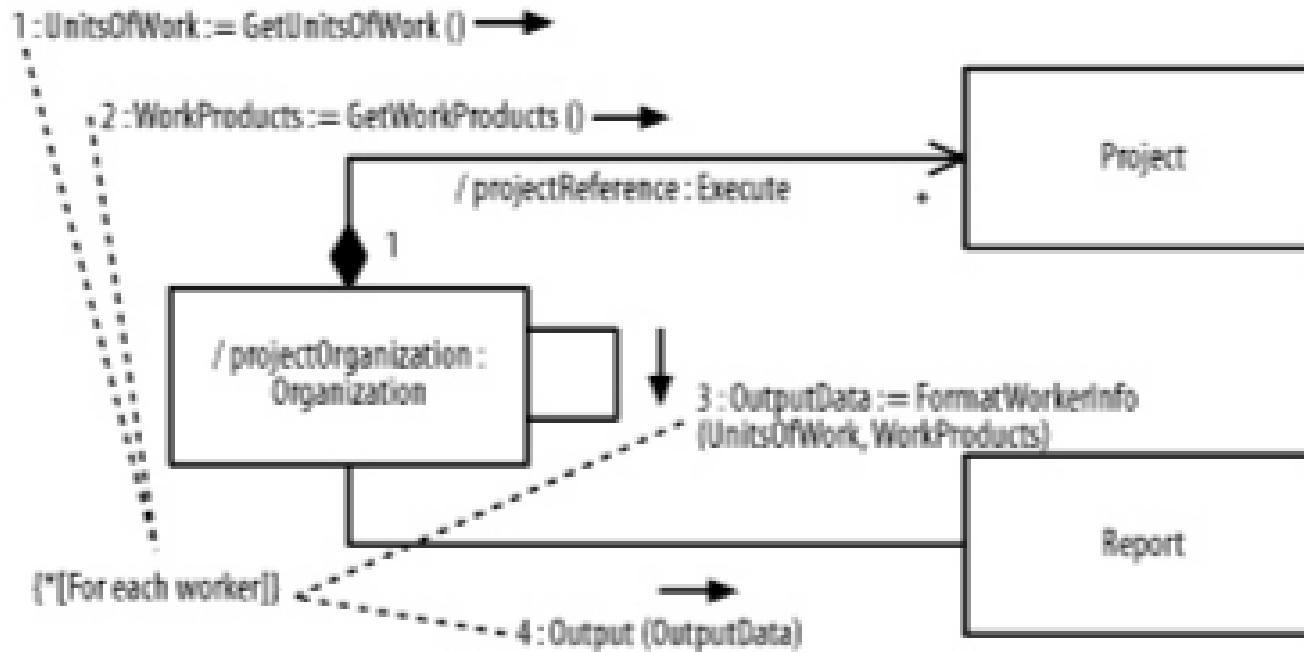
Вместо того, чтобы рисовать каждого участника в виде линии жизни и показывать последовательность сообщений, располагая их по вертикали, как это делается в диаграммах последовательности, коммуникационные диаграммы допускают произвольное размещение участников, позволяя рисовать связи между ними, и использовать вложенную десятичную нумерацию для представления последовательности сообщений.



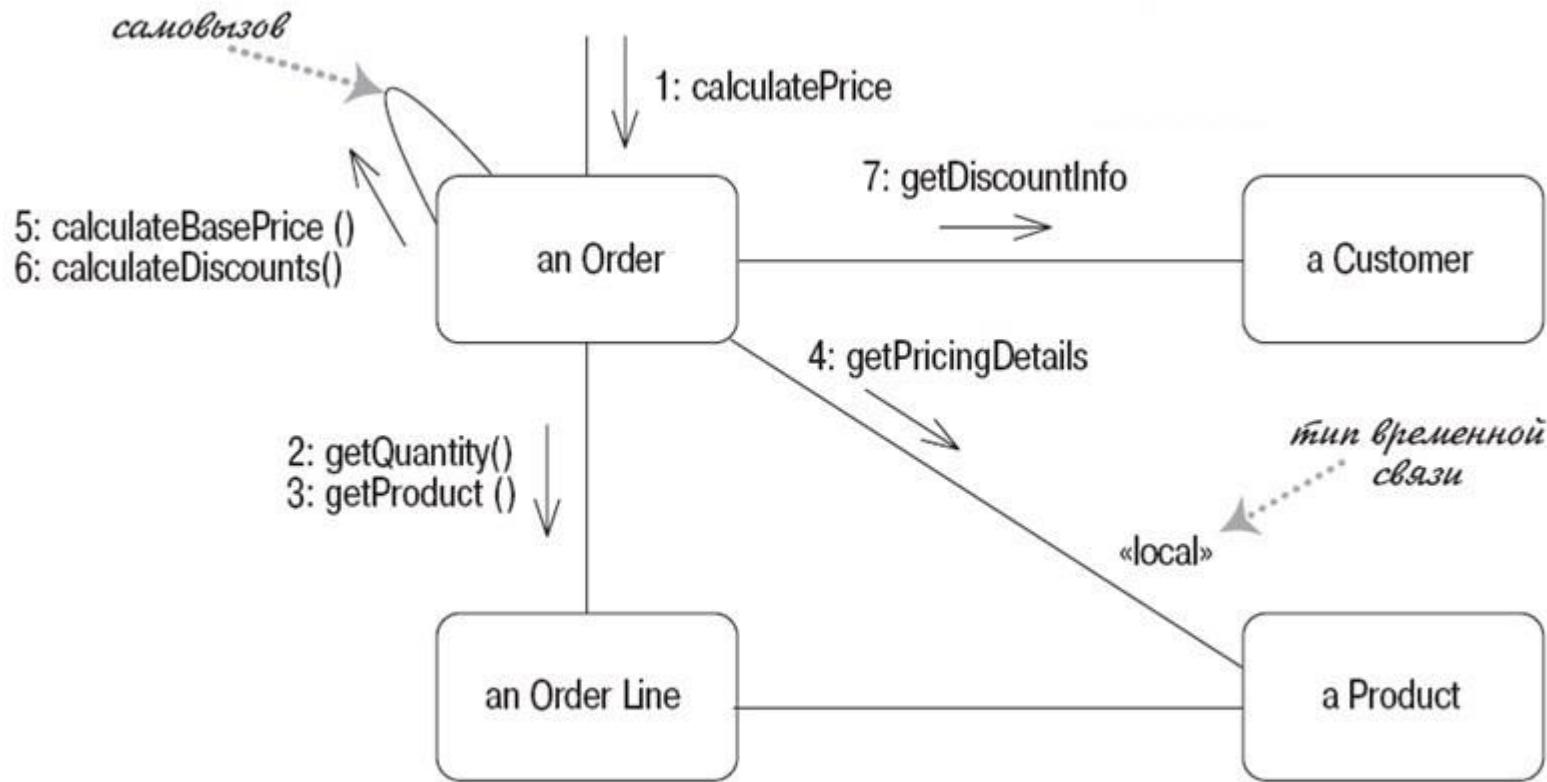
Пример Коммуникационной диаграммы для функции поиска вакансий работ на сайте.



Рефлексивные сообщения и циклы – обозначаются дугой над циклом



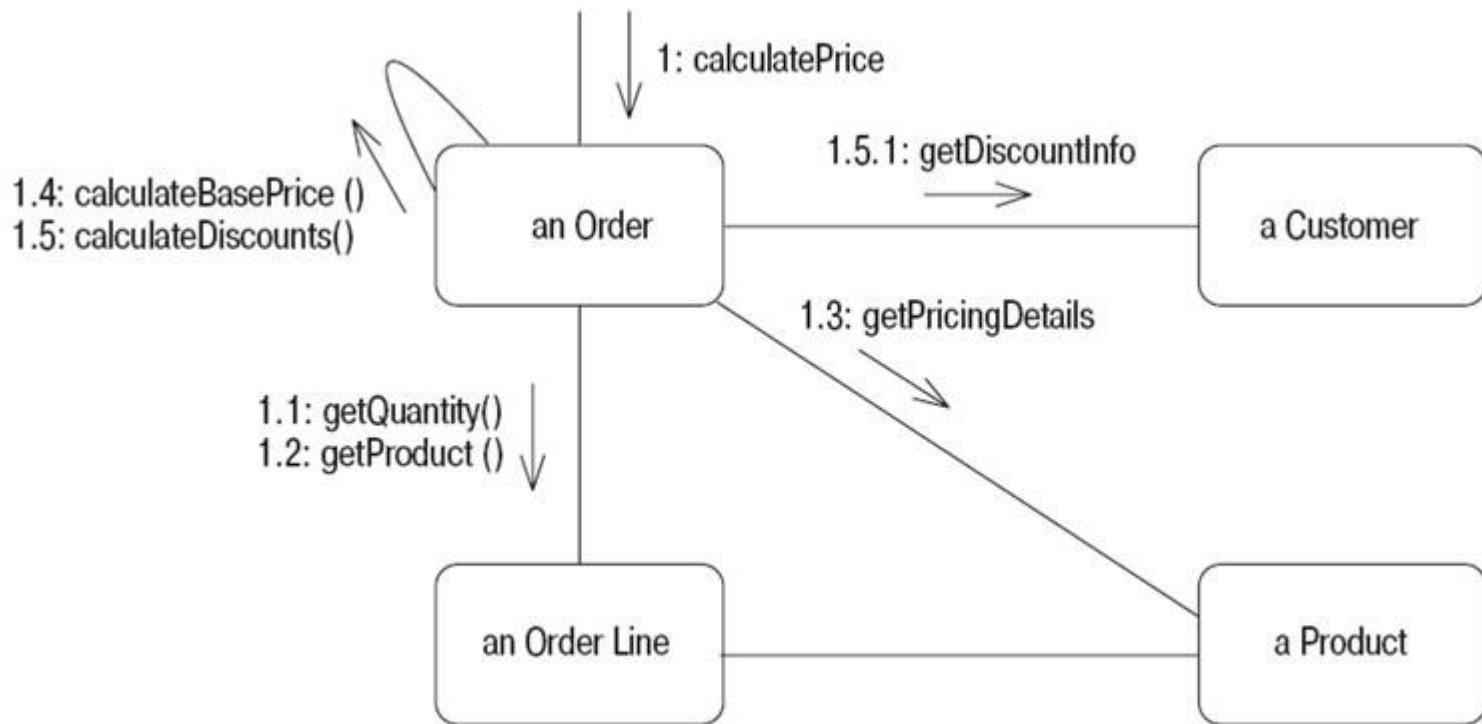
Рефлексивные сообщения и циклы



Коммуникационная диаграмма системы централизованного управления

КД для централизованного управления.

Кроме отображения связей, которые представляют собой экземпляры ассоциаций, можно также показать временные связи, возникающие только в контексте взаимодействия. В данном случае связь **«local»** (локальная) от объекта **Order** (**Заказ**) к объекту **Product** (**Продукт**) – это локальная переменная.



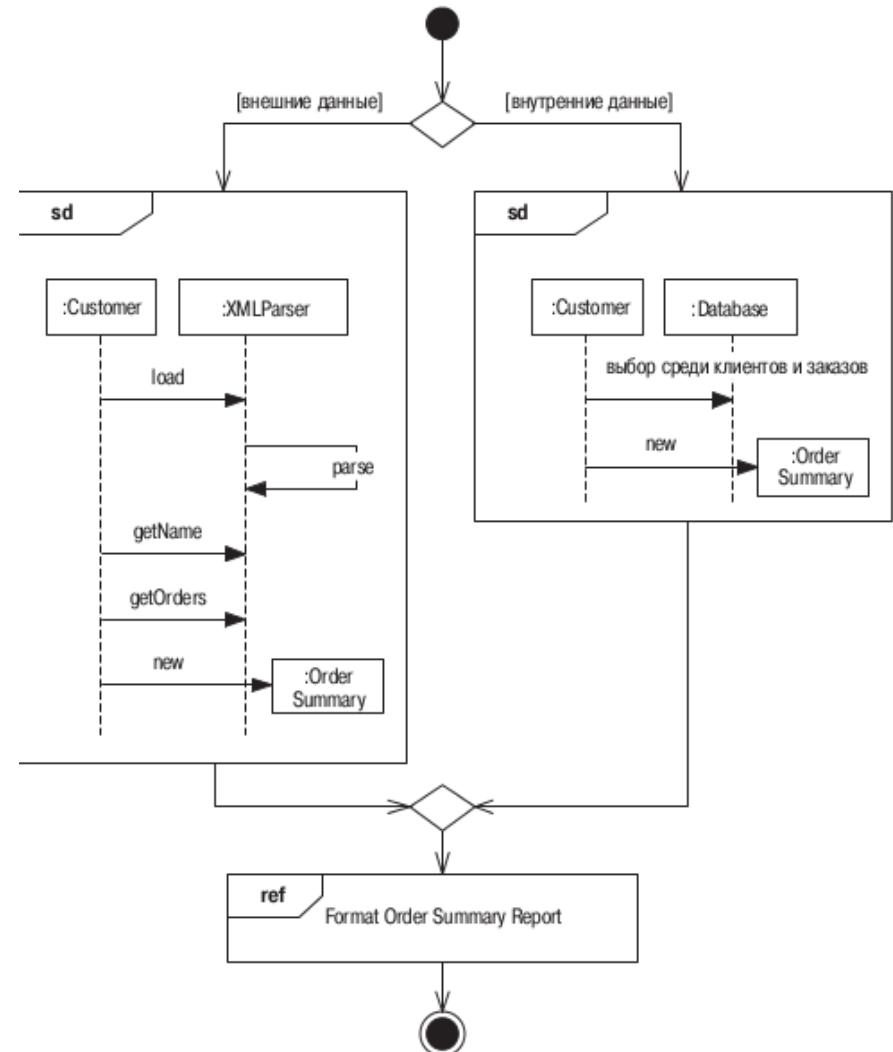
Коммуникационная диаграмма с вложенной десятичной нумерацией

КД для централизованного управления.

В соответствии с правилами UML необходимо придерживаться вложенной десятичной нумерации

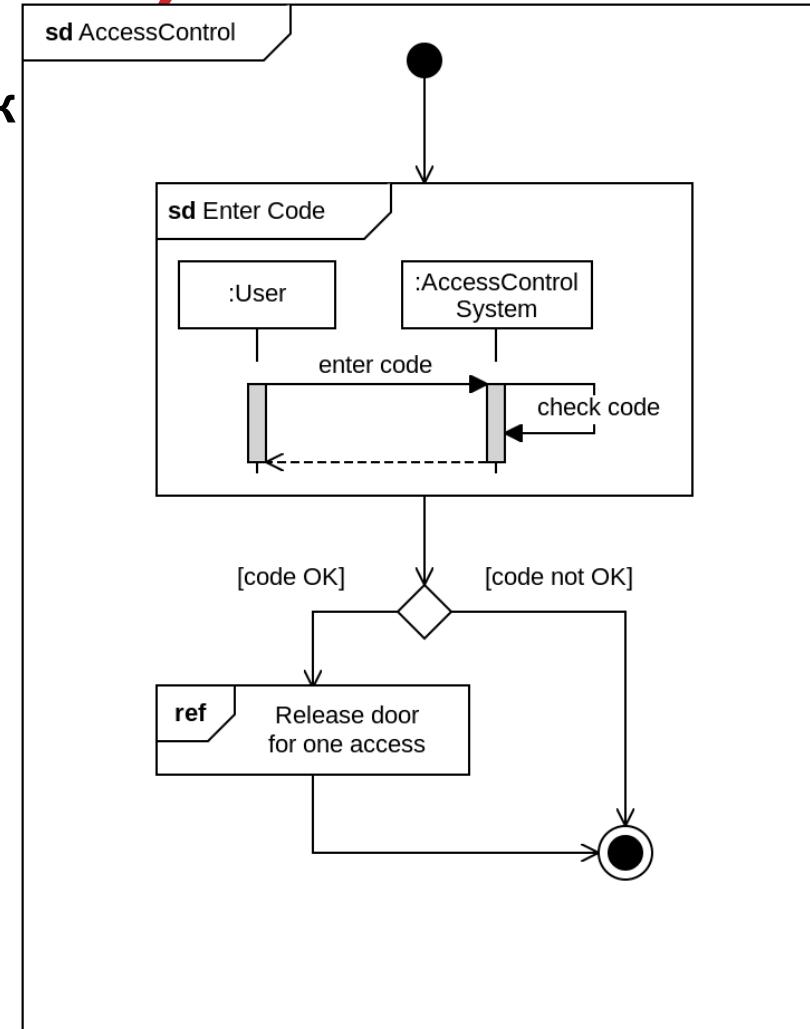
ОБЗОРНЫЕ ДИАГРАММЫ ВЗАЙМОДЕЙСТВИЯ (*INTERACTION OVERVIEW DIAGRAM*)

Обзорные диаграммы взаимодействия – это комбинация диаграмм деятельности и диаграмм последовательности.



ОБЗОРНЫЕ ДИАГРАММЫ ВЗАЙМОДЕЙСТВИЯ (*INTERACTION OVERVIEW DIAGRAM*)

Цель её создания ставится как увязывание в единое целое потока управления между узлами из диаграмм деятельности с последовательностью сообщений между линиями выполнения диаграмм последовательности.: Расширение синтаксиса осуществляется за счёт использования ссылок на взаимодействия, которые основаны на диаграмме последовательности.



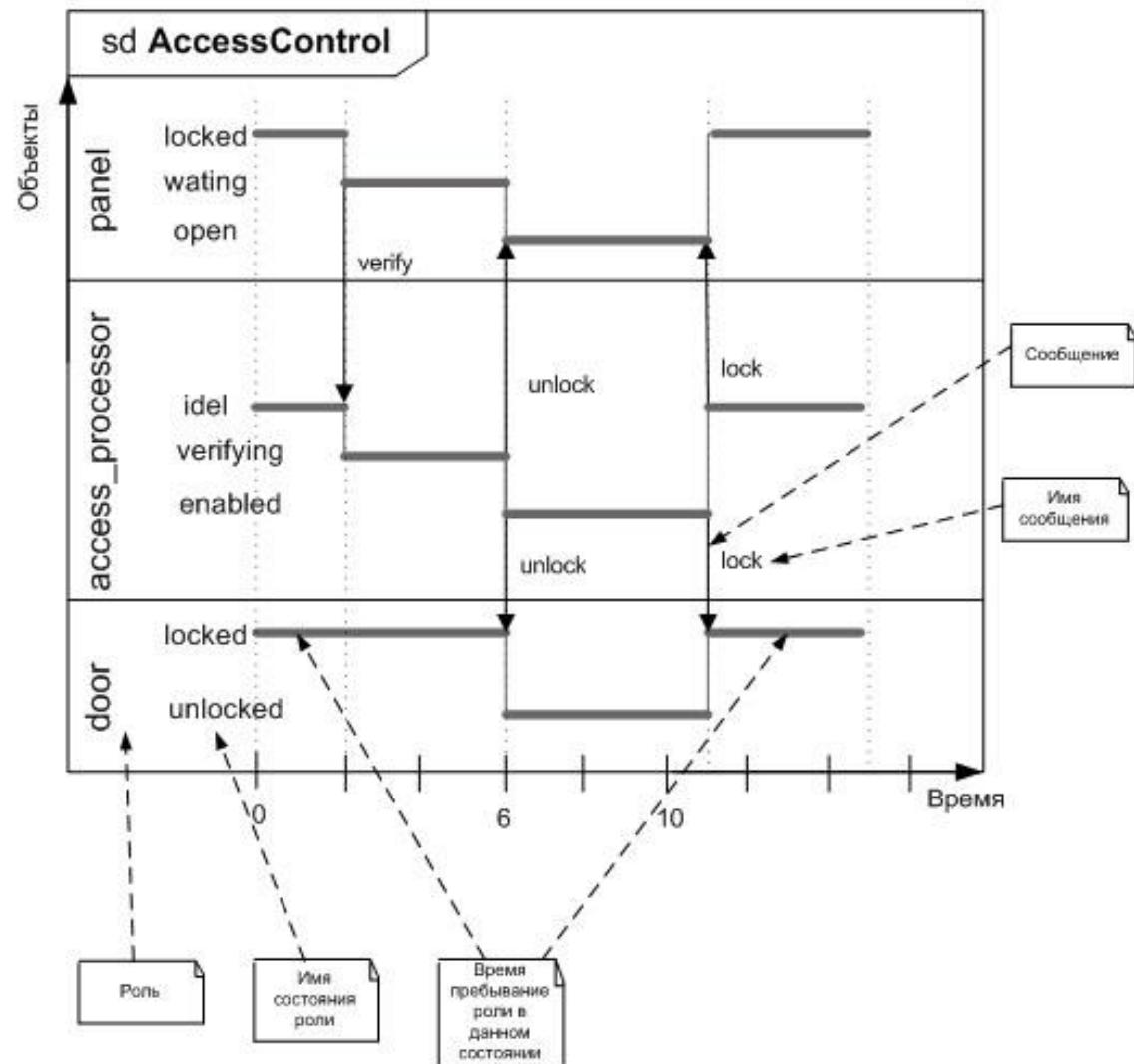
ВРЕМЕННЫЕ ДИАГРАММЫ

Временные диаграммы – это еще одна форма диаграмм взаимодействия, которая акцентирована на временных ограничениях: либо для одиночного объекта, либо, что более полезно, для группы объектов.

Последние изображаются не вертикально, а горизонтально, и основной упор делается на наглядное изображение их состояний, точнее, того, как они меняются во времени. Такая возможность полезна, например, при моделировании встроенных систем.

ВРЕМЕННЫЕ ДИАГРАММЫ

Пример работы системы AccessControl, которая управляет открытием/блокированием двери в помещение по предъявлению человеком электронного ключа. На рисунке показано три компонента этой системы.



ДИАГРАММЫ КОМПОНЕНТОВ (COMPONENT DIAGRAM)

Диаграммы компонентов моделируют физический уровень системы. На них изображаются компоненты ПО и связи между ними.

Во многих средах разработки модуль или компонент соответствует файлу.

Пунктирные стрелки, соединяющие модули, показывают отношения взаимозависимости, аналогичные тем, которые имеют место при компиляции исходных текстов программ.

Основными графическими элементами диаграммы компонентов являются компоненты, интерфейсы и зависимости между ними.

ЦЕЛИ РАЗРАБОТКИ ДИАГРАММ КОМПОНЕНТОВ

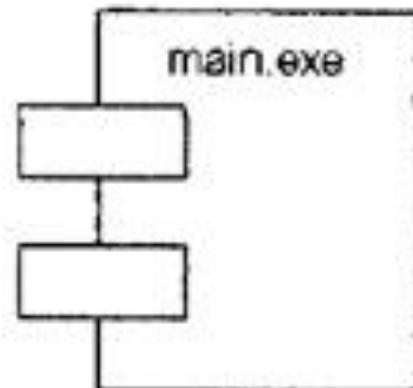
- визуализации общей структуры исходного кода программной системы;
- спецификации исполняемого варианта программной системы;
- обеспечения многократного использования отдельных фрагментов программного кода;
- представления концептуальной и физической схем баз данных.

КОМПОНЕНТ

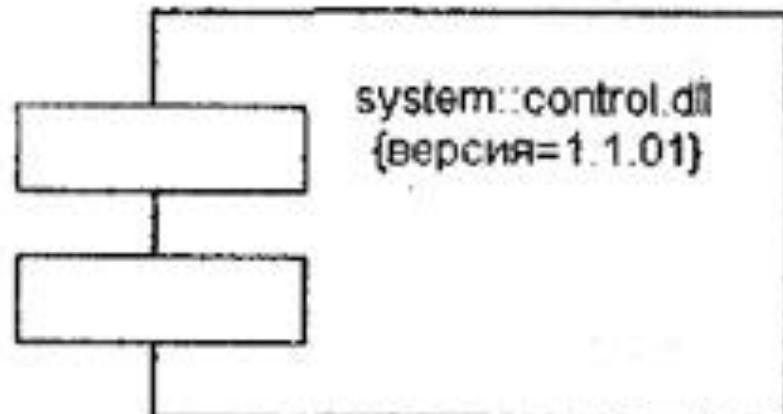
Имя компонента подчиняется общим правилам именования элементов модели в языке UML.

Компонент на уровне экземпляра:

<имя компонента>' : '<имя типаX>



(а)



(б)

ГРАФИЧЕСКИЕ ПУТИ И УЗЛЫ

Графические узлы и пути

Тип графического элемента	Нотация
Компонент (component) с текстовым стереотипом	
Компонент (component) с пиктограммой стереотипа	
Компонент с предоставляемым интерфейсом (provided interface)	
Компонент имеет порт (port) с предоставляемым интерфейсом	
Компонент с требуемым интерфейсом (required interface)	
Компонент имеет порт (port) с требуемым интерфейсом	

ГРАФИЧЕСКИЕ ПУТИ И УЗЛЫ

Компонент имеет не-
сколько портов (ports)
с предоставляемыми
и требуемыми интер-
фейсами



Класс (class)

Имя класса

Часть (part)

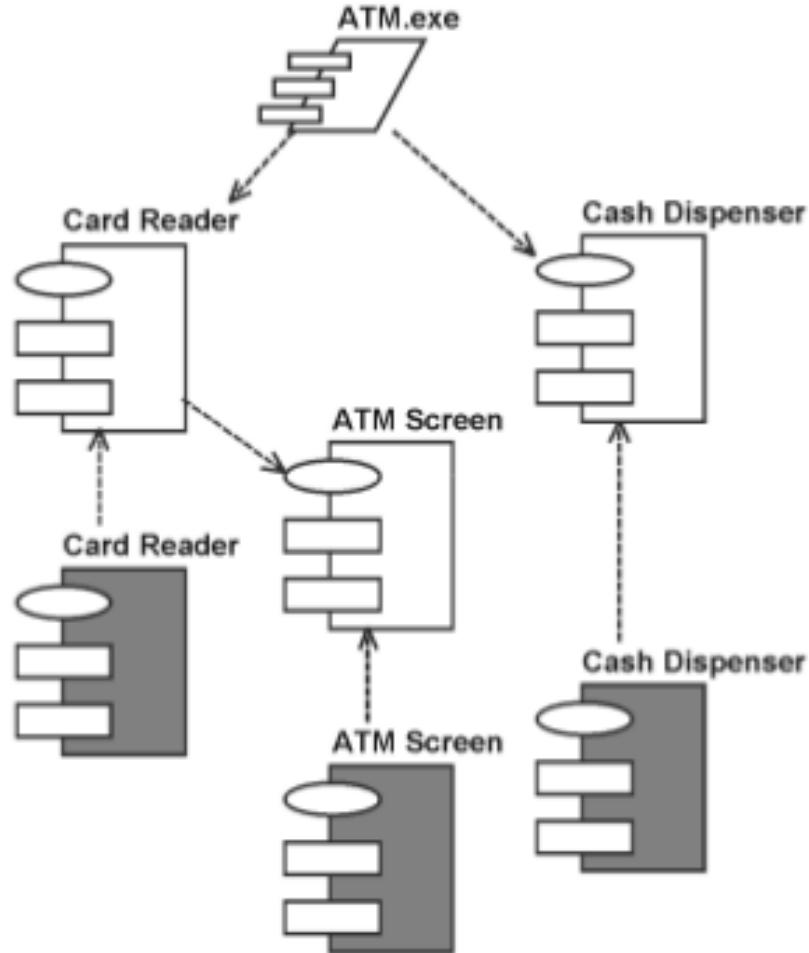
имя части: Имя класса

Собирающий
соединитель
(assembly connector)



СТЕРЕОТИПЫ ДЛЯ КОМПОНЕНТОВ

- библиотека (library)
- таблица (table)
- файл (file)
- документ (document)
- исполняемый (executable)



Пример диаграммы компонентов

ДИАГРАММЫ РАЗВЕРТЫВАНИЯ / РАЗМЕЩЕНИЯ (DEPLOYMENT DIAGRAM)

Это один из видов диаграмм реализации в UML, показывающий физическую конфигурации аппаратного и программного обеспечения. Можно построить несколько диаграмм развертывания для данной системы, каждая из них будет фокусироваться на разных аспектах системы или показывать разные аспекты модели.

ДИАГРАММЫ РАЗВЕРТЫВАНИЯ / РАЗМЕЩЕНИЯ (DEPLOYMENT DIAGRAM)

Диаграмма развертывания (deployment diagram) - диаграмма, на которой представлены узлы выполнения программных компонентов реального времени, а также процессов и объектов.

Диаграмма развёртывания показывает топологию системы и распределение компонентов системы по ее узлам, а также соединения - маршруты передачи информации между аппаратными узлами. Это единственная диаграмма, на которой применяются “трехмерные” обозначения: узлы системы обозначаются кубиками. Все остальные обозначения в UML - плоские фигуры обязательно компьютерных.

ДИАГРАММЫ РАЗВЕРТЫВАНИЯ / РАЗМЕЩЕНИЯ (DEPLOYMENT DIAGRAM)

Польза диаграмм развёртывания

- Графическое представление ИТ-инфраструктуры может помочь более рационально распределить компоненты системы по узлам сети, от чего зависит в том числе и производительность системы.
- Такая диаграмма может помочь решить множество вспомогательных задач, связанных, например, с обеспечением безопасности.

ДИАГРАММЫ РАЗМЕЩЕНИЯ

Диаграмма размещения (развертывания) отражает физические взаимосвязи между программными и аппаратными компонентами системы.

Основные элементы:

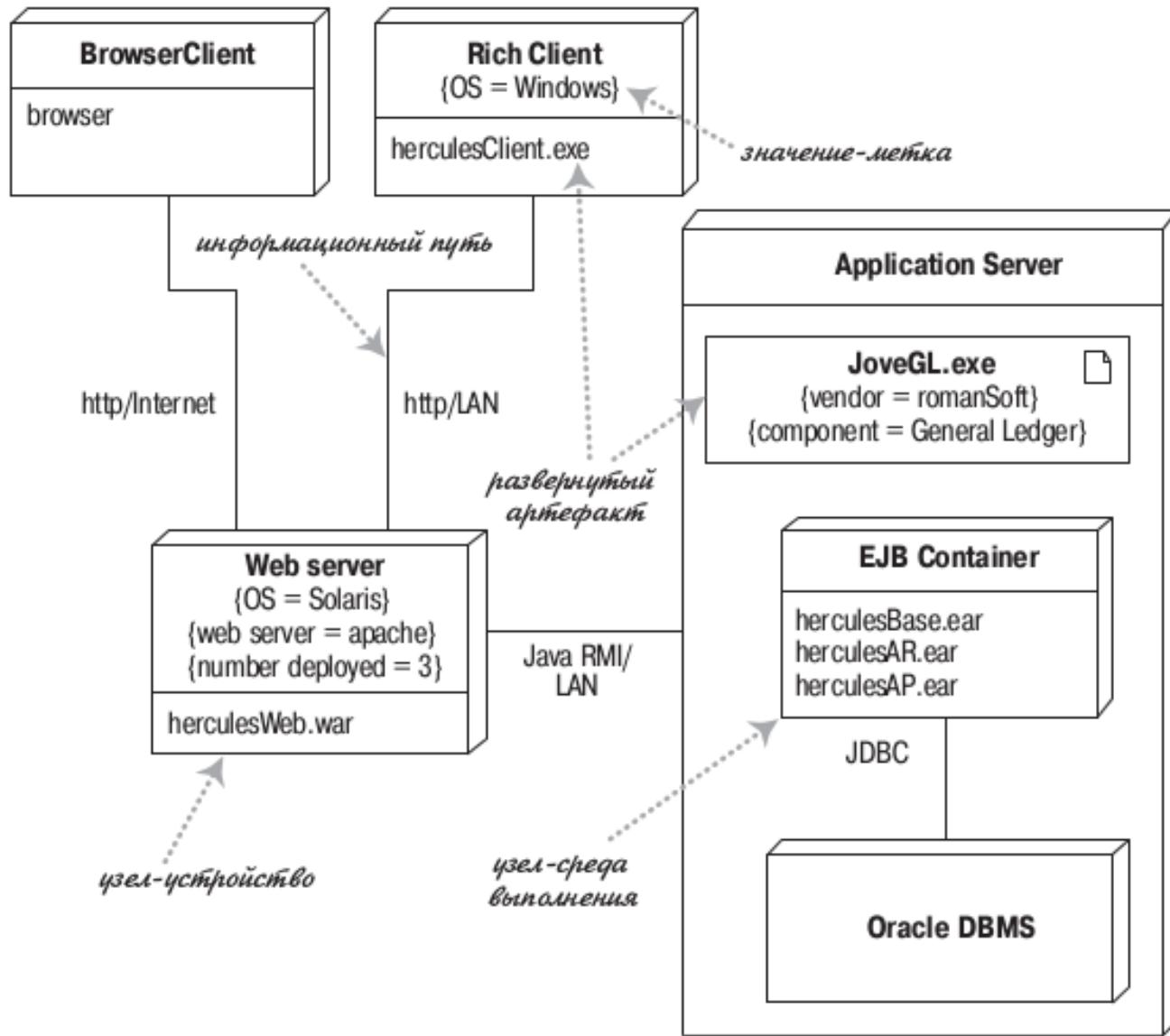
1. узел (node):

- устройство (device)
- среда выполнения (execution environment)

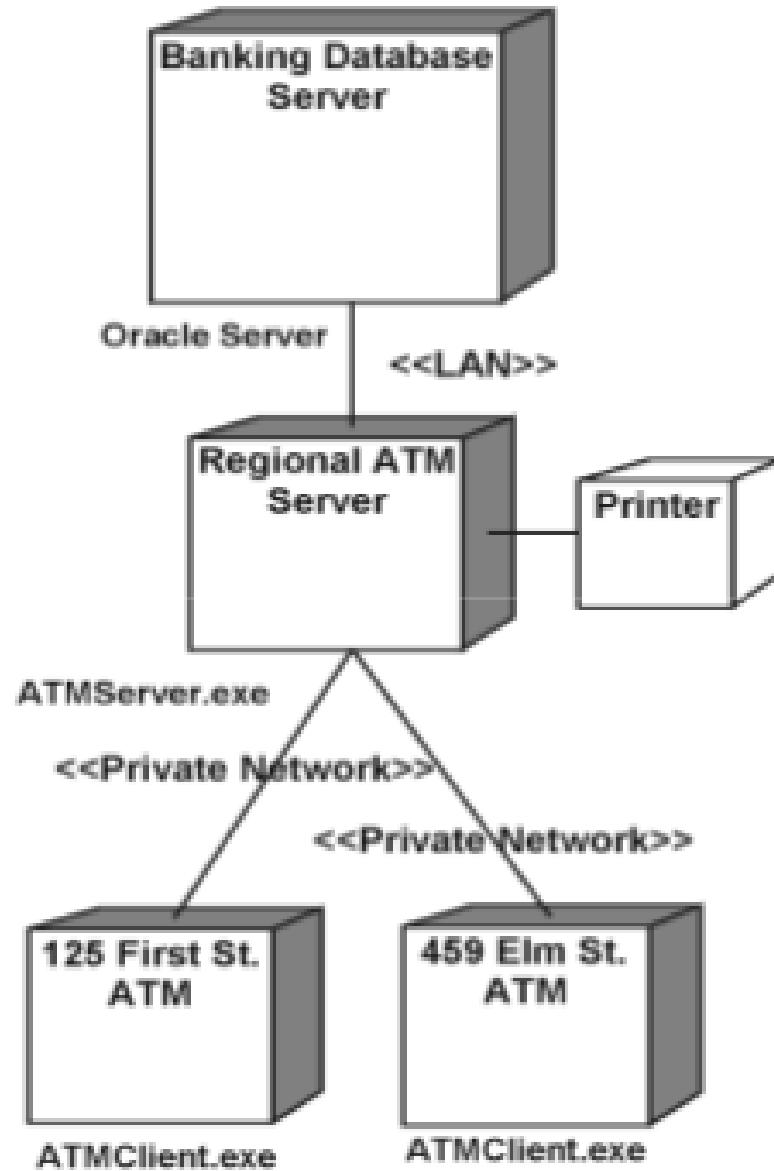
2. линия коммуникации (связи) (communication line)

ОСНОВНЫЕ ЭЛЕМЕНТЫ

	Компонент (Component)
	Экземпляр компонента (Component instance)
	Интерфейс (Interface)
	Узел (Node)
	Экземпляр узла (Node instance)
	Объект (Object)
	Активный объект (Active object)
	Зависимость (Dependency)
	Пример Связь (Connection)
	Точка изгиба связей (Point)
	Комментарий (Note)
	Коннектор комментария (Note connector)



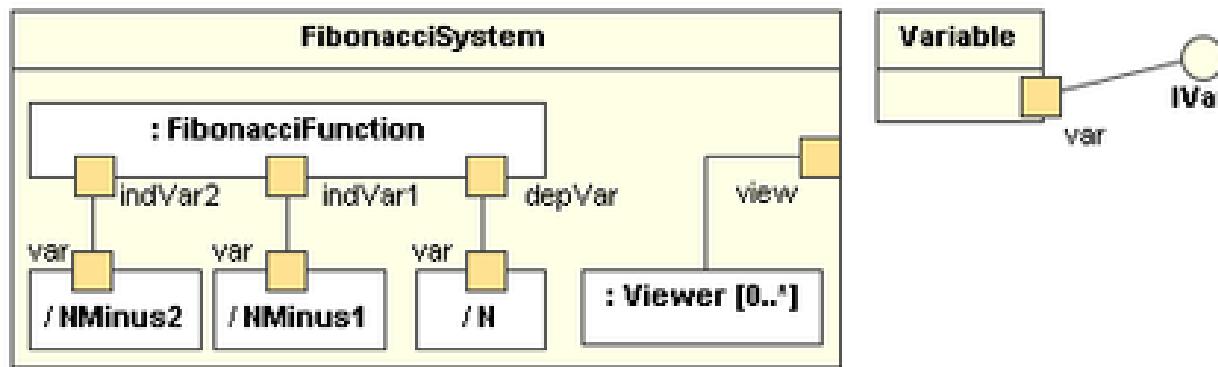
Пример диаграммы размещения



Пример диаграммы размещения

ДИАГРАММЫ СОСТАВНОЙ СТРУКТУРЫ

Диаграммы составных структур (Composite Structure Diagram)
- статическая структурная диаграмма, демонстрирует внутреннюю структуру классов и, по возможности, взаимодействие элементов (частей) внутренней структуры класса.

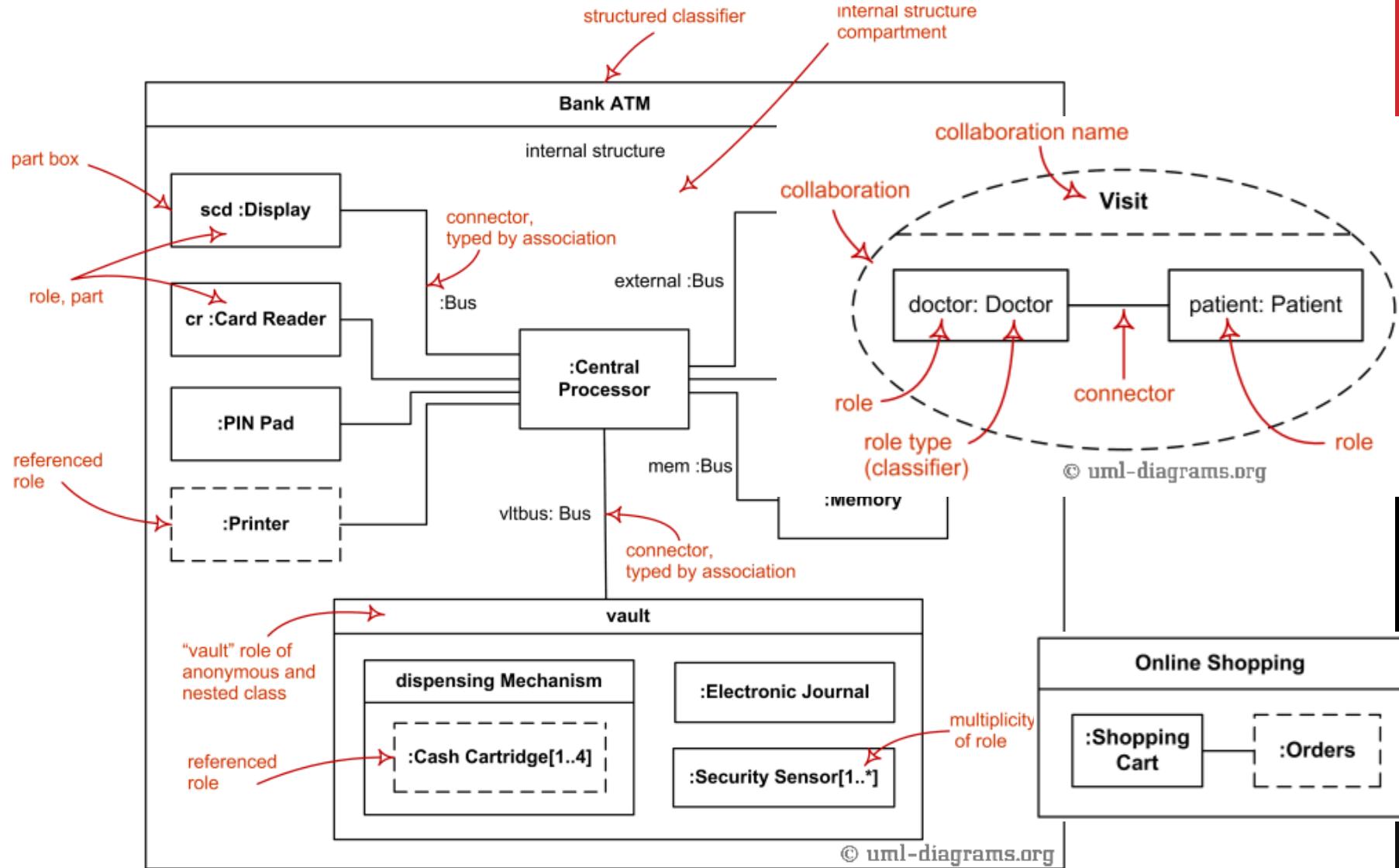


НАПРАВЛЕНИЯ ИСПОЛЬЗОВАНИЯ

Внутренняя структура классификатора

Классификатор взаимодействия

Поведение при взаимодействии

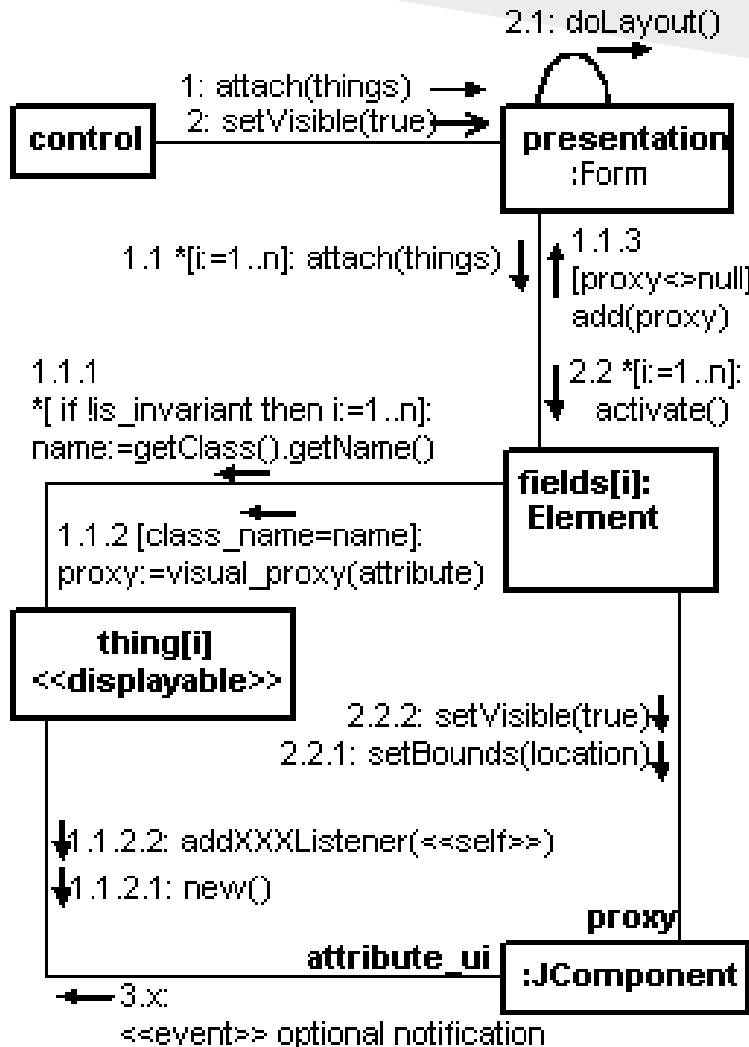


Примеры диаграмм составной структуры

ДИАГРАММА КООПЕРАЦИИ (COLLABORATION DIAGRAM)

Диаграмма кооперации - диаграмма взаимодействий, в которой основной акцент сделан на структурной организации объектов, посылающих и получающих сообщения. На этой диаграмме изображено, как организованы взаимодействия между экземплярами и какие между ними существуют связи. Это, по сути, альтернативная форма диаграммы последовательностей, более компактная, но и более сложная для чтения.

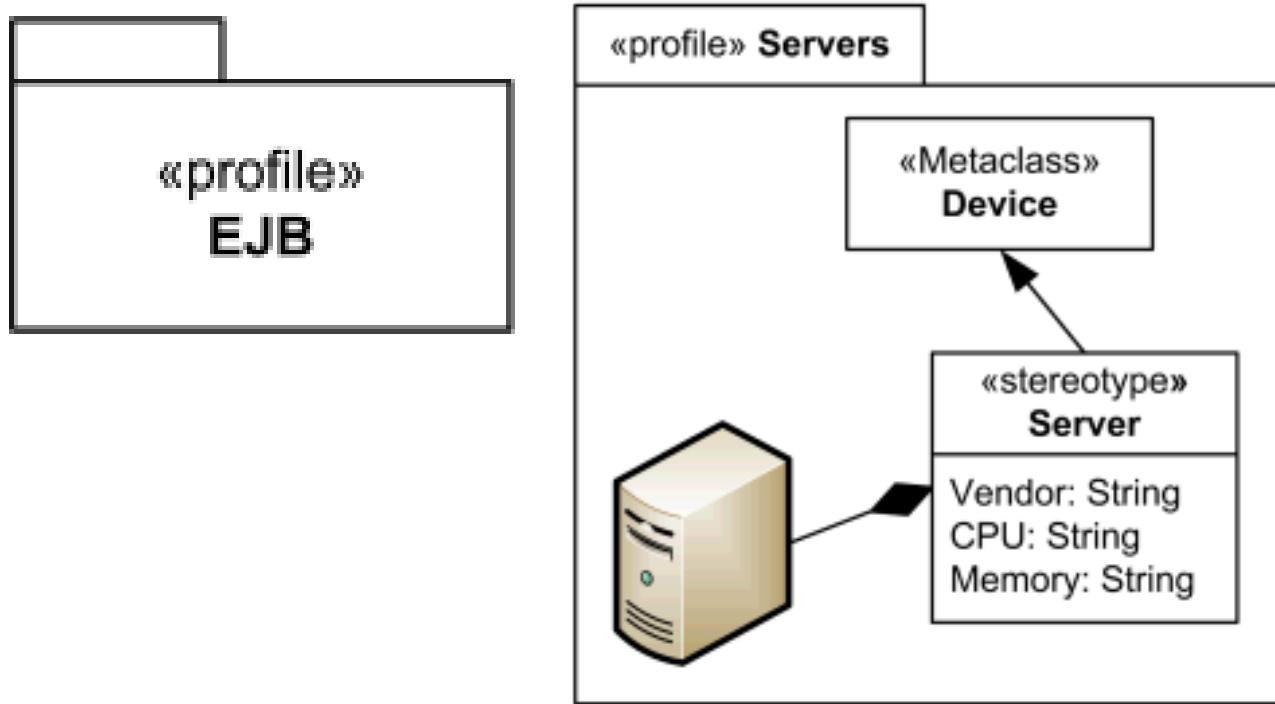
ДИАГРАММА КООПЕРАЦИИ (COLLABORATION DIAGRAM)



ДИАГРАММЫ ПРОФИЛЯ

Диаграмма профиля (*profile diagram*) — структурная диаграмма, которая описывает простой механизм расширения в UML, определив собственные стереотипы, поименованные значения и ограничения. Профили позволяют адаптировать UML метамодели под разные:

- платформы (J2EE или .NET)
- домены (в режиме реального времени или моделирования бизнес-процессов).



Пример диаграмм профиля

СПАСИБО ЗА ВНИМАНИЕ!

ПРОЕКТИРОВАНИЕ ПРОГРАММНЫХ СИСТЕМ

ОСНОВНЫЕ ТЕМЫ КУРСА

- **Основы программной инженерии**
- **Язык UML**
- **Архитектура программных систем.**
Моделирование систем. Объектная модель
- **Определение требований к ПО**
- **Анализ и проектирование. Проектирование баз данных**
- **Шаблоны проектирования**
- **Управление проектами**
- **Методологии разработки ПО**

ИТОГОВАЯ АТТЕСТАЦИЯ

Лекции - 34 часа

Лабораторные занятия - 26 часов

Управляемая самостоятельная работа - 8 часов

Форма аттестации - зачет

Зачет при условии:

- 1. сданы и зачтены преподавателем все лабораторные работы,**
- 2. Сданы все тесты ≥ 6 баллов**

ГДЕ ПРИМЕНЯТЬ?

- Аналитик (Системный, бизнес-аналитик)
- Архитектор (системный, информационный)
- Проектировщик

(Analyst; Architects, Project Manager, Tech Leader)

ЛЕКЦИЯ 1

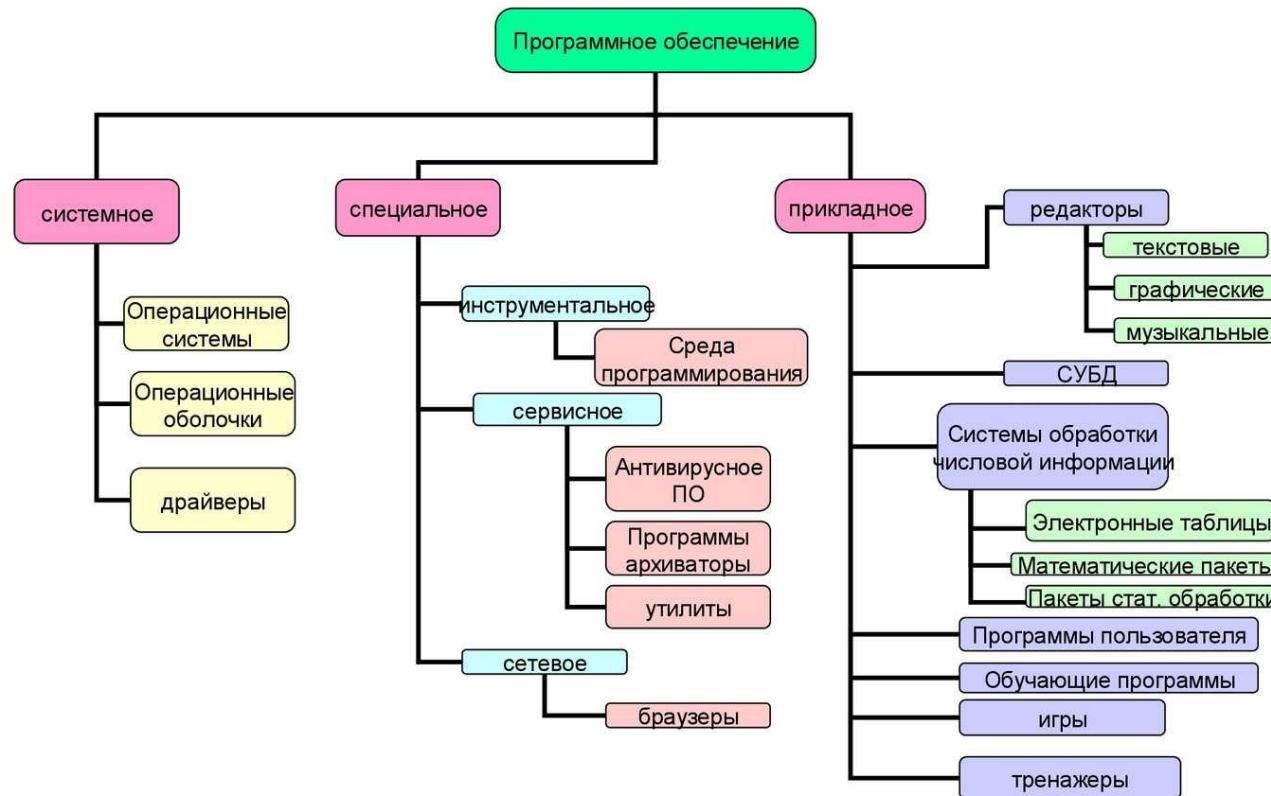
ОСНОВЫ
ПРОГРАММНОЙ
ИНЖЕНЕРИИ

ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Программное обеспечение (ПО) – это не только программы, но и вся сопутствующая документация, а также конфигурационные данные, необходимые для корректной работы программ.

КЛАССИФИКАЦИЯ ПО

КЛАССИФИКАЦИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ПК



СИСТЕМНЫЙ ПОДХОД

Основой проектирования программного обеспечения является системный подход.

Системный подход – это методология исследования объекта любой природы как системы.

СИСТЕМА

- это совокупность взаимосвязанных частей, работающих совместно для достижения некоторого результата. Определяющий признак системы – поведение системы в целом не сводимо к совокупности поведения частей системы.**
- это совокупность элементов и связей между ними. Ее можно представить в виде модели ниже. В рамках данной модели объекты моделируются сущностями, а связи моделируются отношениями. В рамках данного курса будем изучать методы моделирования и проектирования программных систем.**

ПРОГРАММНЫЕ СИСТЕМЫ –

это большие, сложные совокупности программных компонентов, которые предназначены для управления целым рядом, как принято говорить, ресурсных контуров – это людские ресурсы, финансовые ресурсы, различные другие виды производственных ресурсов, специфичные нефтегазовые ресурсы, документы и тому подобные ресурсы.

ПРОГРАММНЫЕ СИСТЕМЫ

Программные системы состоят из :

- совокупности программ,**
- файлов конфигурации, необходимых для установки этих программ,**
- и документации, которая описывает структуру системы, а также содержит инструкции для пользователей, объясняющие работу с системой,**
- и часто адрес web-узла, где пользователь может найти самую последнюю информацию о данном программном продукте.**

ТИПЫ ПРОГРАММНЫХ ПРОДУКТОВ

Специалисты по программному обеспечению разрабатывают программные продукты, т.е. такое ПО, которое можно продать потребителю. Программные продукты делятся на два типа.

- 1. Общие программные продукты.** Это автономные программные системы, которые созданы компанией по производству ПО и продаются на открытом рынке программных продуктов любому потребителю, способному их купить. Примерами этого типа программных продуктов могут служить системы управления базами данных, текстовые процессоры, графические пакеты и средства управления проектами.
- 2. Программные продукты, созданные на заказ.** Это программные системы, которые создаются по заказу определенного потребителя. Такое ПО разрабатывается специально для данного потребителя согласно заключенному контракту. Программные продукты этого типа включают системы управления для электронных устройств, системы поддержки определенных производственных или бизнес-процессов, системы управления воздушным транспортом и т.п.

ПРИКЛАДНАЯ СИСТЕМА

**При этом, если мы говорим о прикладной системе или
приложении, речь идет о том, чтобы решать задачи
прикладного плана.**

**Например, управление корпоративными ресурсами в аспекте
людских ресурсов:**

- прием на работу, перевод, увольнение,
- зачисление, обучение, повышение квалификации и т.д.

ПРОГРАММНАЯ ИНЖЕНЕРИЯ

Для решения такого рода задач по созданию программных систем, как систем прикладных, так и систем, имеющих другое назначение, как раз и возникла дисциплина, которая названа программной инженерией.

СТАНОВЛЕНИЕ И РАЗВИТИЕ ПРОГРАММНОЙ ИНЖЕНЕРИИ

Термин – software engineering (программная инженерия) – впервые был озвучен в октябре 1968 года на конференции подкомитета НАТО по науке и технике (г. Гармиш, Германия).

Присутствовало 50 профессиональных разработчиков ПО из 11 стран.

Рассматривались проблемы проектирования, разработки, распространения и поддержки программ.

СТАНОВЛЕНИЕ И РАЗВИТИЕ ПРОГРАММНОЙ ИНЖЕНЕРИИ

Становление и развитие этой области деятельности было вызвано рядом проблем, связанных с

- высокой стоимостью программного обеспечения, его создания**
- необходимостью управления и прогнозирования процессов разработки.**

СТАНОВЛЕНИЕ И РАЗВИТИЕ ПРОГРАММНОЙ ИНЖЕНЕРИИ

В конце 60-х – начале 70-х годов прошлого века произошло событие, которое вошло в историю как первый кризис программирования.

Событие состояло в том, что стоимость программного обеспечения стала приближаться к стоимости аппаратуры («железа»)

СТАНОВЛЕНИЕ И РАЗВИТИЕ ПРОГРАММНОЙ ИНЖЕНЕРИИ

ПРЕДПОСЫЛКИ

- Повторное использование кода (модульное программирование)
- Рост сложности программ (структурное программирование)
- Модификация программ (ООП)

НЕКОТОРЫЕ ИТОГИ

Программная инженерия (или технология промышленного программирования) как некоторое направление возникло и формировалось под давлением роста стоимости создаваемого программного обеспечения.

Главная цель этой области знаний – сокращение стоимости и сроков разработки программ. Программная инженерия прошла несколько этапов развития, в процессе которых были сформулированы фундаментальные принципы и методы разработки программных продуктов.

Основной принцип программной инженерии состоит в том, что программы создаются в результате выполнения нескольких взаимосвязанных этапов (анализ требований, проектирование, разработка, внедрение, сопровождение), составляющих жизненный цикл программного продукта.

Фундаментальными методами проектирования и разработки являются модульное, структурное и объектно-ориентированное проектирование и программирование.

ПРОДОЛЖЕНИЕ КРИЗИСА ПРОГРАММИРОВАНИЯ

Рубеж 80–90-х годов отмечается как начало информационно-технологической революции, вызванной взрывным ростом использования информационных средств: персональный компьютер, локальные и глобальные вычислительные сети, мобильная связь, электронная почту, Internet и т.д.

Цена успеха – кризис программирования принимает хронические формы.

	2004	2006	2008	2010	2012	2013
УСПЕШНЫЕ	29%	35%	32%	37%	39%	36%
ПРОВАЛЬНЫЕ	18%	19%	24%	21%	18%	16%
СПОРНЫЕ	53%	46%	44%	42%	43%	48%



По некоторым исследованиям каждый шестой ИТ-проект оказывается провальным.

Это подразумевает разрастание изначального бюджета (чуть ли не до 200% от запланированного) и затягивание сроков (почти на 70%).



ФРЕДЕРИК БРУКС

- Характерные вопросы и задачи инженерии программирования, изложенные Фредериком Бруксом (л-т премии Тьюринга) еще в 1999 г, актуальны и по сегодняшний день.
- Как проектировать и строить программы, образующие системы?
- Как проектировать и строить программы и системы, являющиеся надежным, отлаженным, документированным и сопровождаемым продуктом?
- Как осуществлять интеллектуальный контроль в условиях большой сложности?



Фредерик Филипс Брукс мл. (Frederick Phillips Brooks, Jr.), род. 19 апреля 1931 года — американский менеджер, инженер и ученый, наиболее известен как руководитель разработки операционной системы OS/360. В 1975 году, обобщая опыт этой работы, написал книгу "Мифический человекомесяц". Брукс насмешливо называл свою книгу "библией программной инженерии": "все ее читали, но никто ей не следует!".

ОПРЕДЕЛЕНИЕ ПРОГРАММНОЙ ИНЖЕНЕРИИ

(из книги Липаева «Программная инженерия»)

Программная инженерия – это комплекс задач, методов, средств и технологий создания (проектирования и реализации) сложных, расширяемых, тиражируемых, высококачественных программных систем, возможно включающих базы данных.

ФУНДАМЕНТАЛЬНАЯ ИДЕЯ ПРОГРАММНОЙ ИНЖЕНЕРИИ:

**Проектирование ПО является формальным
процессом, который можно изучать и
совершенствовать.**

**Освоение и правильное применение методов и
средств программной инженерии позволяет
повысить качество, обеспечить управляемость
процесса проектирования.**

ПРОБЛЕМЫ РАЗРАБОТКИ ПО

- **Достижение высокого качества при**
 - Нехватке ресурсов
 - Возрастающей сложности программных систем
 - Часто изменяющихся требованиях
 - Большом количестве участников
- **Устаревание ПО**
- **Высокие темпы разработки**
- **Информатизация все новых отраслей хозяйства**
- **Высокая конкуренция**

ОГРАНИЧЕНИЯ ДЛЯ РЕАЛИЗАЦИИ ПРОГРАММНЫХ СИСТЕМ:

- **размер реализуемых программных систем комплексов и компонент, из которых строятся такого рода системы;**
- **сложность, т.е. количество модулей и связей между ними;**
- **программную среду, в рамках которой происходит реализация;**
- **и определенные ограничения, связанные с человеческим фактором, с людьми, с людскими ресурсами.**

**По сути дела, при реализации
программных систем мы
сталкиваемся с задачей
многофакторной оптимизации.**

СИСТЕМНЫЙ ПОДХОД КАК ОСНОВА ПРОЕКТИРОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

СИСТЕМНЫЙ ПОДХОД –

**общенаучный обобщенный эвроритм,
предусматривающий всестороннее исследование
сложного объекта с использованием
компонентного, структурного, функционального,
параметрического и генетического видов анализа.**

(Эвроритм – порядок действия человека при выполнении какой-то деятельности. В отличие от алгоритма может изменяться в процессе исполнения благодаря разумности исполнителя.)

Компонентный анализ – рассмотрение объекта, включающего в себя составные элементы и входящего, в свою очередь, в систему более высокого ранга.

Структурный анализ – выявление элементов объекта и связей между ними.

Функциональный анализ – рассмотрение объекта как комплекса выполняемых им полезных и вредных функций.

Параметрический анализ – установление качественных пределов развития объекта – физических, экономических, экологических и др.

Применительно к программам параметрами могут быть: время выполнения какого-нибудь алгоритма, размер занимаемой памяти и т.д.

При этом выявляются ключевые технические противоречия, мешающие дальнейшему развитию объекта, и ставится задача их устранения за счет новых технических решений.

Генетический анализ – исследование объекта на его соответствие законам развития программных систем. В процессе анализа изучается история развития (генезис) исследуемого объекта: конструкции аналогов и возможных частей, технологии изготовления, объемы тиражирования, языки программирования и т.д.

ОСОБЕННОСТИ И КЛАССИФИКАЦИЯ СОВРЕМЕННЫХ ПРОГРАММНЫХ ПРОЕКТОВ

ОСОБЕННОСТИ СОВРЕМЕННЫХ ПРОГРАММНЫХ ПРОЕКТОВ

- сложность – неотъемлемая характеристика создаваемого ПО;
- отсутствие полных аналогов и высокая доля вновь разрабатываемого ПО;
- наличие унаследованного ПО и необходимость его интеграции с разрабатываемым ПО;
- территориально распределенная и неоднородная среда функционирования;
- большое количество участников проектирования, разобщенность и разнородность отдельных групп разработчиков по уровню квалификации и опыту.

СПЕЦИФИЧЕСКИЕ ОСОБЕННОСТИ РАЗРАБОТКИ ПО

- неформальный характер требований к ПО и формализованный основной объект разработки – программы;
- творческий характер разработки;
- дуализм ПО, которое, с одной стороны, является статическим объектом – совокупностью текстов, с другой стороны, – динамическим, поскольку при эксплуатации порождаются процессы обработки данных;
- при своем использовании (эксплуатации) ПО не расходуется и не изнашивается;
- «неощущимость», «воздушность» ПО, что подталкивает к безответственному переделыванию, поскольку легко стереть и переписать, чего не сделаешь при проектировании зданий и аппаратуры.

КЛАССИФИКАЦИЯ ПРОЕКТОВ

Институтом качества программного обеспечения SQI (Software Quality Institute, США) специально для выбора модели ЖЦ разработана схема классификации проектов по разработке ПС. Основу данной классификации составляют четыре категории критериев:

- **Характеристики требований к проекту.**
- **Характеристики команды разработчиков.**
- **Характеристики пользователей (заказчиков).**
- **Характеристики типов проектов и рисков.**

КЛАССИФИКАЦИЯ ПРОЕКТОВ

Набор критериев данной категории, предложенный институтом SQI, является недостаточно полным и может быть дополнен критериями из других общепринятых классификаций:

- масштаб продукта проекта (размер или сложность продукта проекта);
- стабильность продукта проекта (эволюция продукта проекта);
- критичность продукта проекта (степень влияния на общество повреждений продукта проекта).

КЛАССИФИКАЦИЯ ПРОЕКТОВ ПО РАЗМЕРУ

- **небольшие проекты – проектная команда (ПК) менее 10 чел., срок от 3 до 6 мес.;**
- **средние проекты – ПК от 20 до 30 чел., протяженность проекта 1-2 года;**
- **крупномасштабные проекты – ПК от 100 до 300 чел., длительность проекта 3-5 лет;**
- **гигантские проекты – ПК от 1000 до 2000 чел. и более (включая консультантов и соисполнителей), протяженность проекта от 7 до 10 лет.**

ЖИЗНЕННЫЙ ЦИКЛ ПО

В ЭТОМ РАЗДЕЛЕ ЛЕКЦИИ ИСПОЛЬЗОВАНЫ
МАТЕРИАЛЫ ЭЛЕКТРОННОГО КУРСА
«РАЗРАБОТКА КОРПОРАТИВНЫХ СИСТЕМ. ЧАСТЬ
1. МОДЕЛИ ЖИЗНЕННОГО ЦИКЛА» НИЯУ МИФИ.

ЖИЗНЕННЫЙ ЦИКЛ ПО –

это достаточно протяженный во времени процесс, который начинается с концепции, может быть с неформальной идеи, только с общего представления самых предварительных контуров программного продукта и заканчивается с полным выводом из эксплуатации этого программного продукта или программной системы

(книга "Быстрая разработка" Стива Мак Коннелла).

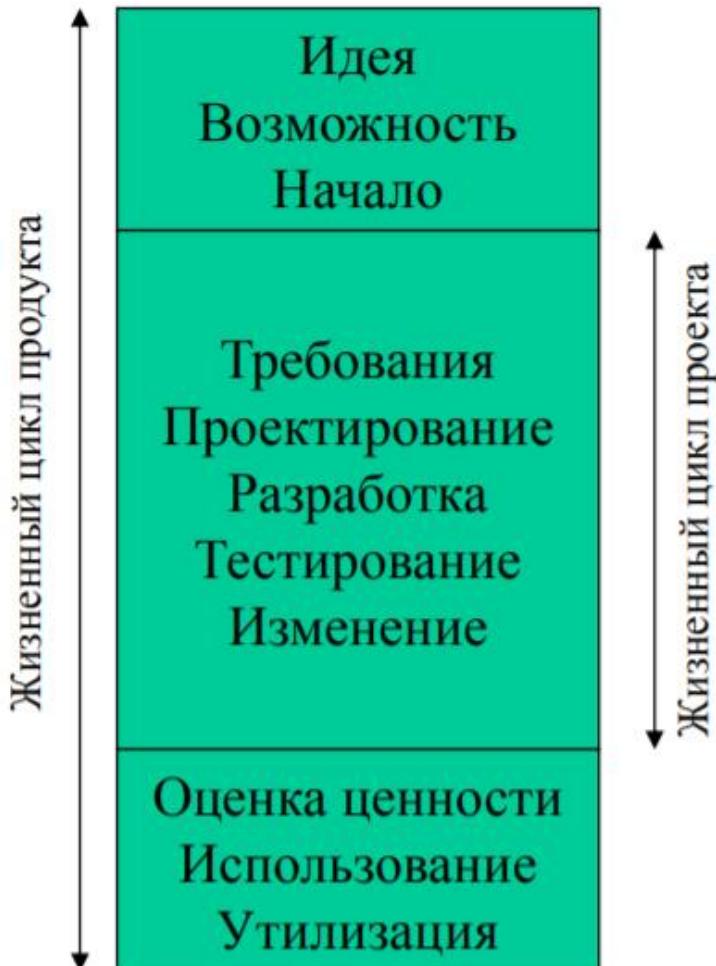
ОСОБЕННОСТИ ЖЦ РАЗРАБОТКИ ПО

- Программная система разрабатывается постепенно и развивается, начиная от зарождения идеи ПО до реализации и сдачи пользователю, и далее.
- Каждый этап завершается разработкой части системы или связанной с ней документации (план тестирования, руководство пользователя и т.д.).
- Теоретически, для каждого этапа четко определены начальные и конечные точки, а также известно, что он должен передать следующему этапу.
- На практике все сложнее.

ЦЕЛИ ИЗУЧЕНИЯ ЖЦ

- **Организация и управление разработкой ПО.**
- **Основа для анализа разработки ПО.**
- **Основа для планирования разработки ПО.**
- **Корректная постановка процессов разработки ПО.**
- **Анализ ЖЦ обязателен для сложных проектов.**

СВЯЗЬ ЖЦ ПРОДУКТА И ПРОЕКТА



Проект – это нечто измеримое, его жизненный цикл выглядит несколько иначе, более конкретно и включает разработку требований, проектных спецификаций, реализацию, проектирование, интеграцию, тестирование и управление изменениями с учетом тех возможных коррективов в требованиях, которые могут потребоваться заказчику.

МОДЕЛЬ ЖИЗНЕННОГО ЦИКЛА

- **Модель ЖЦ ПО** – это структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач на протяжении всего ЖЦ.
- ***В любой модели ЖЦ рассматривается как совокупность стадий.***
- **Стадия ЖЦ** – это часть ЖЦ ограниченная временными рамками, по завершении которой достигается определенный важный результат в соответствии с требованиями для данной стадии ЖЦ.
- **Между двумя стадиями, идущими друг за другом, находится контрольная точка (веха).** Так называют момент времени, разделяющий стадии жизненного цикла (или итерации, если они предусмотрены в модели ЖЦ), по наступлении которого должны достигаться результаты важные для всего проекта и должны приниматься решения о дальнейшем управлении проектом.

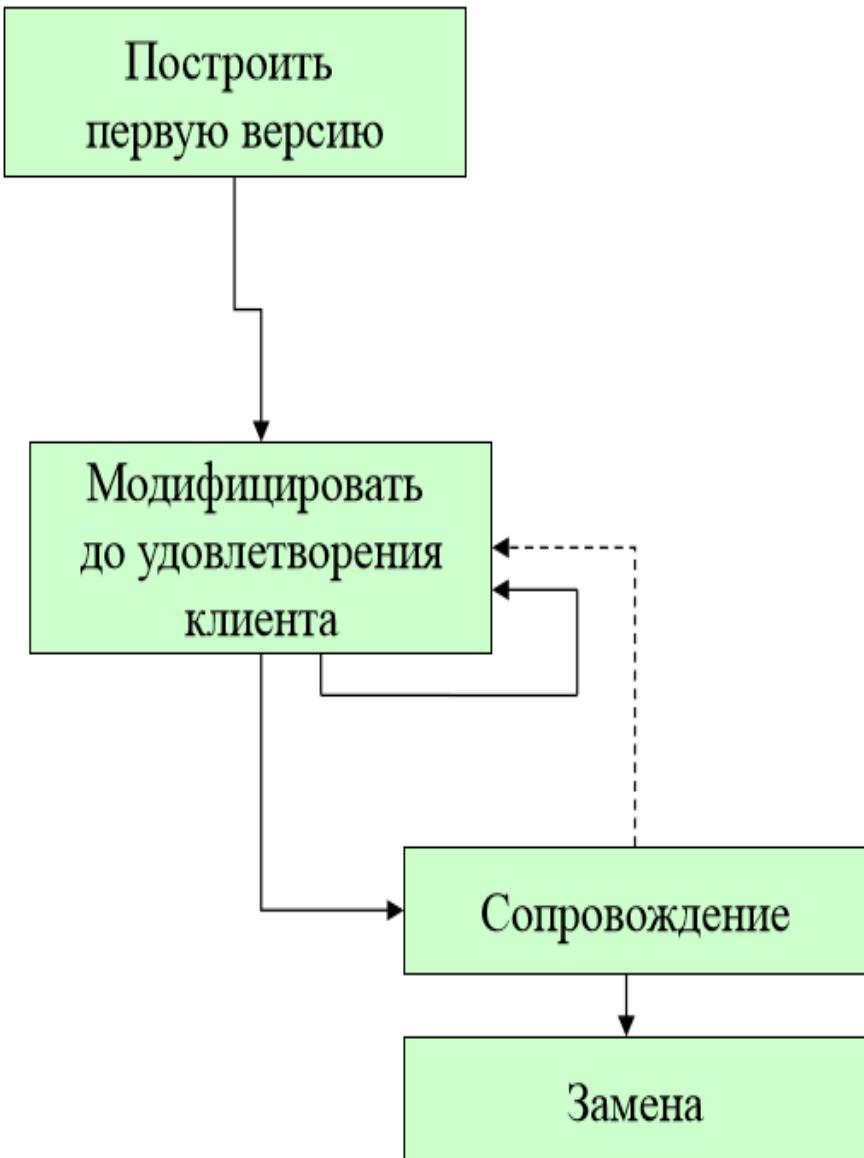
МОДЕЛИ ЖЦ ПО ОПРЕДЕЛЯЮТ:

- **Характер и масштаб проекта (объем, сроки, риски, ...)**
- **Экономику проекта**
- **Степень сопровождаемости**
- **Перспективы развития (прогноз запросов клиента)**
- **Архитектуру проекта (стабильная эволюция, революционные усовершенствования)**
- **Скорость поиска и устранения ошибок**
- **Управление рисками проекта**
- **Степень полноты реализации (прототип, промежуточное решение, готовый продукт)**

ОГРАНИЧЕННЫЕ МОДЕЛИ ЖИЗНЕННОГО ЦИКЛА

МОДЕЛИ, КОТОРЫЕ НЕСКОЛЬКО ОГРАНИЧЕНЫ В СИЛУ РАЗЛИЧНЫХ ПРИЧИН: ПРОСТОТА, НЕПРИГОДНОСТЬ ДЛЯ БОЛЬШИХ И СЛОЖНЫХ ПРОЕКТОВ, НЕСАМОСТОЯТЕЛЬНОСТЬ.

МОДЕЛЬ CODE-AND-FIX (BUILD-AND- FIX)



По сути дела, речь идет о модели проб и ошибок.

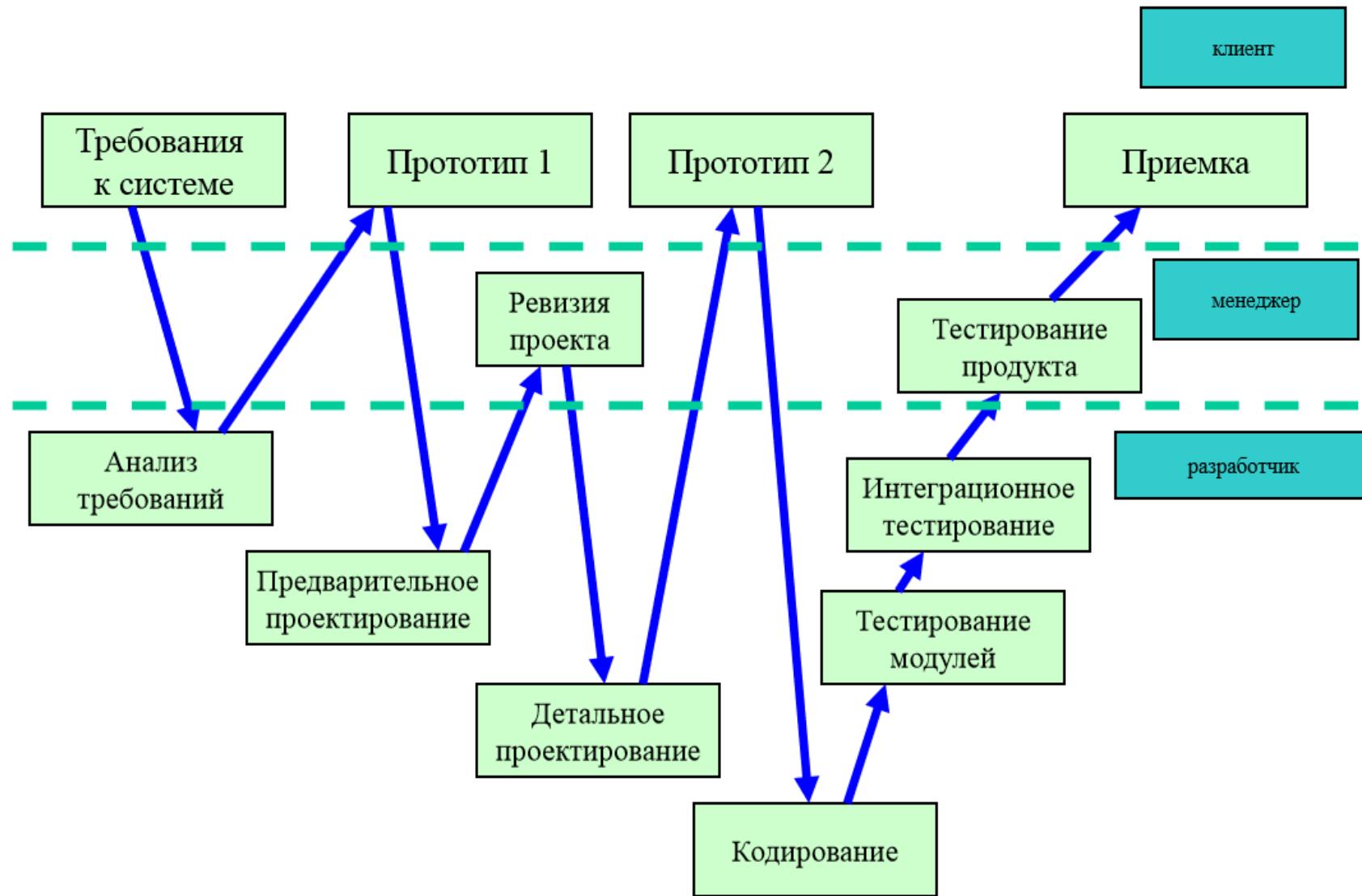
Здесь приходится строить продукт заново каждый раз до тех пор, пока клиент не будет доволен, не будет удовлетворен.

ВОДОПАДНАЯ/КАСКАДНАЯ МОДЕЛЬ



Существенной особенностью этой модели является однопроходная разработка без циклических повторений. Другим существенным аспектом на каждой стадии является непременное условие верификации.

МОДЕЛЬ БЫСТРОГО ПРОТОТИПИРОВАНИЯ/«ЗУБЬЯ АКУЛЫ»



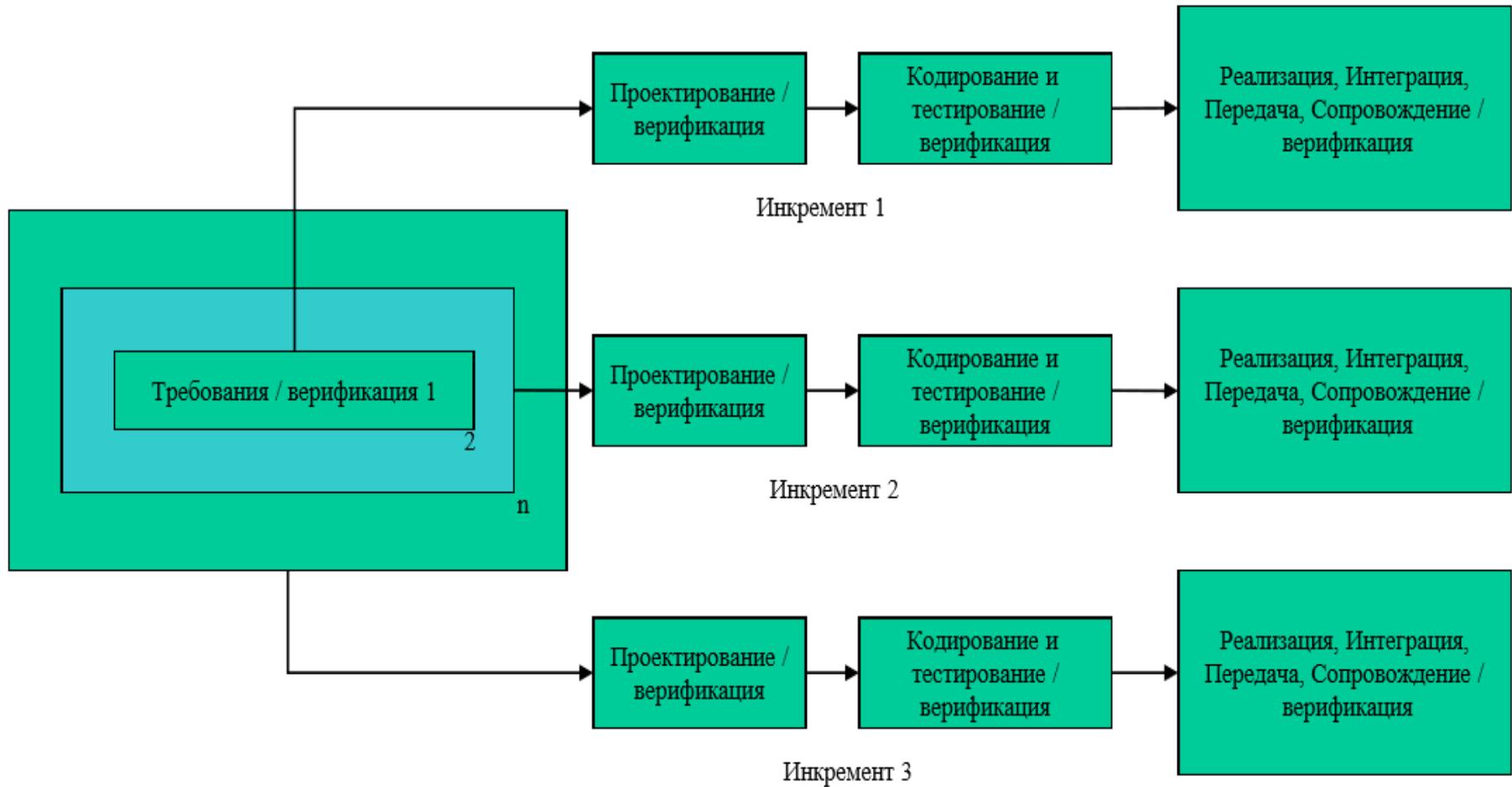
ОГРАНИЧЕННЫЕ МОДЕЛИ ЖЦ ПО: СРАВНИТЕЛЬНЫЙ АНАЛИЗ

Модель ЖЦ	Преимущества	Недостатки
Build-and-Fix (Code – and –Fix)	Хороша для небольших, не требующих сопровождения проектов	Абсолютно непригодна для нетривиальных проектов
Водопадная	Четкая дисциплина проекта, документно-управляемая	ПО может не соответствовать требованиям клиента
Быстрого прототипирования	Обеспечивает соответствие ПО требованиям клиента	Вызывает соблазн повторного использования кода, который следует заново реализовать

ЦИКЛИЧЕСКИЕ МОДЕЛИ ЖИЗНЕННОГО ЦИКЛА

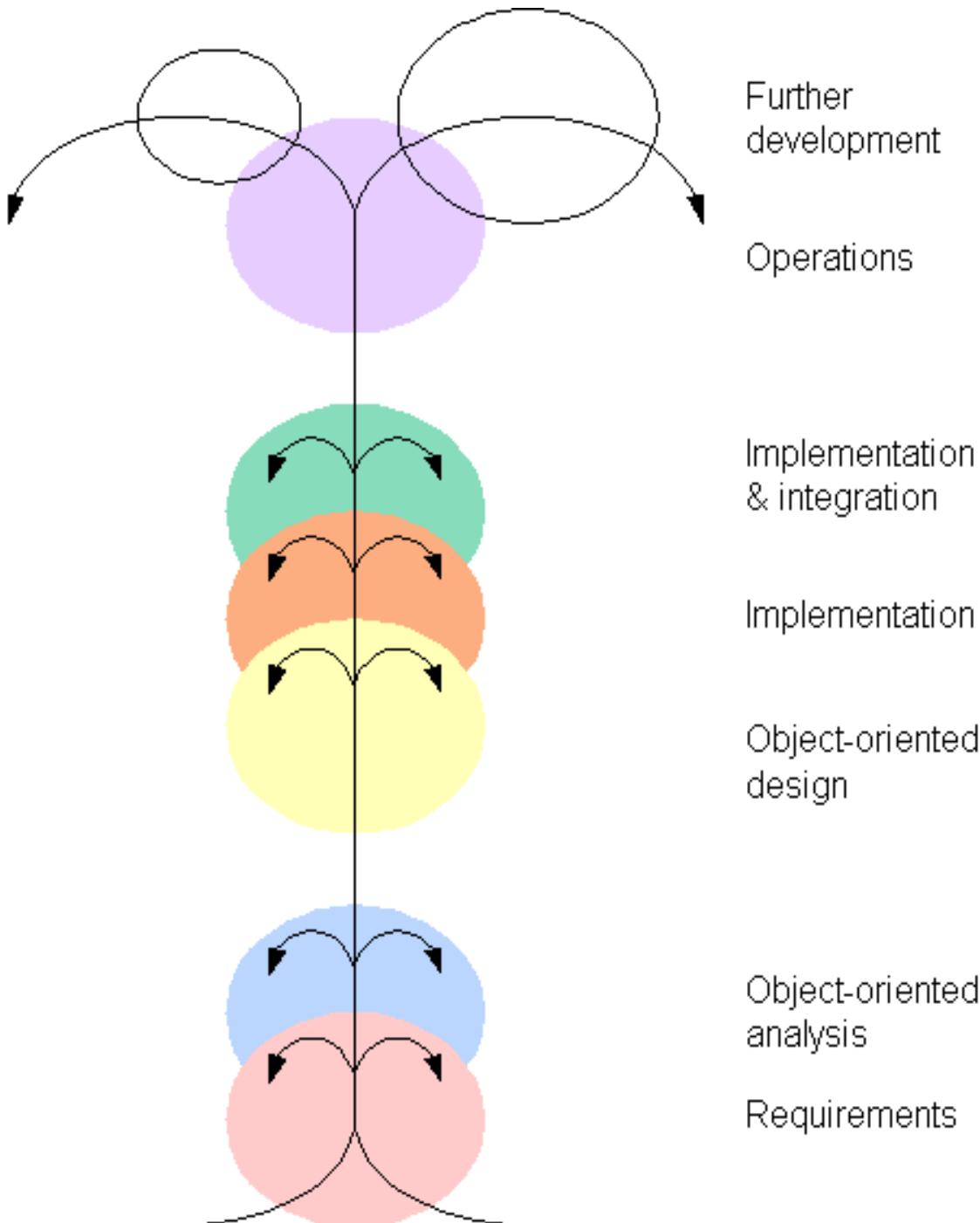
МОДЕЛИ, В КОТОРЫХ СТАДИИ ЖИЗНЕННОГО ЦИКЛА
СМЕНЯЮТ ДРУГ ДРУГА ИТЕРАТИВНО ИЛИ ЦИКЛИЧЕСКИ

ИНКРЕМЕНТНАЯ ИЛИ ИНКРЕМЕНТАЛЬНАЯ МОДЕЛЬ



*Каждый релиз включает детальное проектирование ,
реализацию, интеграцию, тестирование и передачу*

ОБ'ЄКТНО- ОРИЕНТИРОВАННА Я МОДЕЛЬ



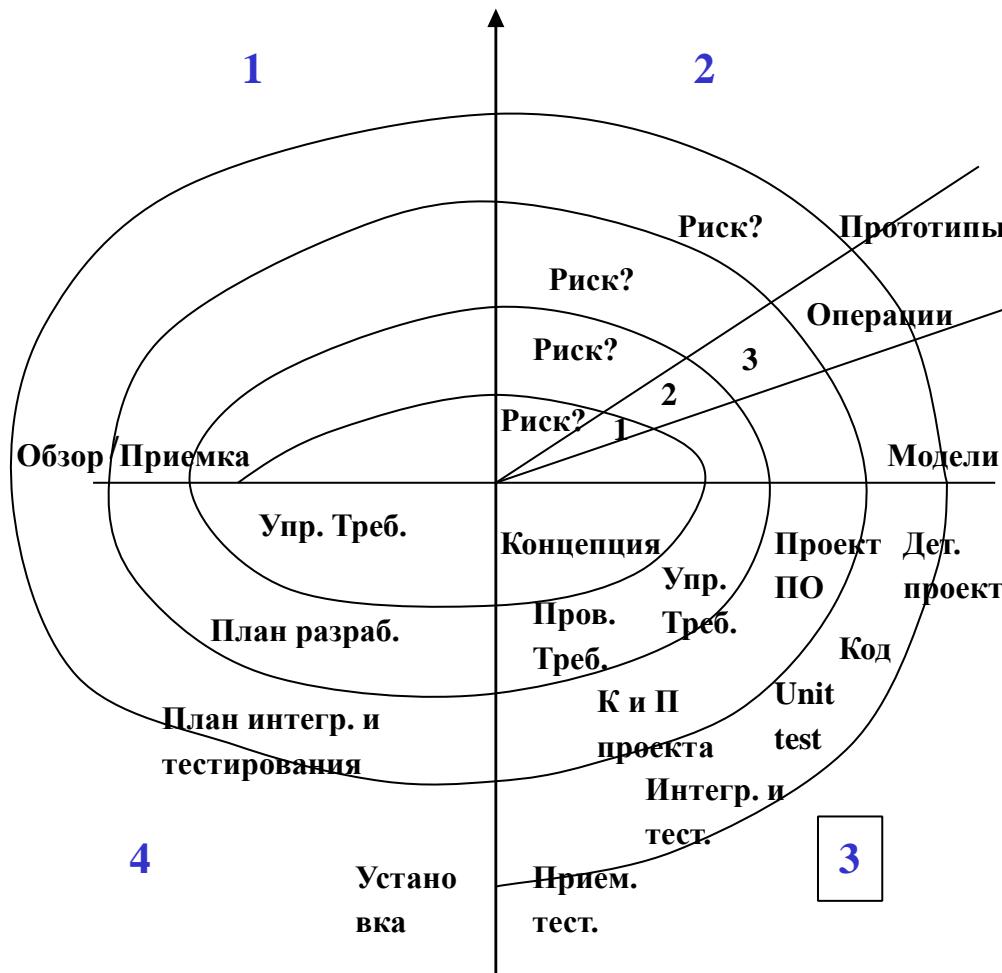
ЦИКЛИЧЕСКИЕ МОДЕЛИ ЖЦ ПО: СРАВНИТЕЛЬНЫЙ АНАЛИЗ

Модель ЖЦ	Преимущества	Недостатки
Инкрементная	Максимально ранний возврат инвестиций; способствует сопровождаемости	Требует открытой архитектуры; может выродиться в Build-and-fix
ОО-модель	Обеспечивает итерацию внутри фаз и параллелизм между фазами	Может выродиться в Build-and-fix

СПЕЦИАЛИЗИРОВАННЫЕ МОДЕЛИ ЖИЗНЕННОГО ЦИКЛА

СПИРАЛЬНАЯ МОДЕЛЬ ВКЛЮЧАЕТ В СЕБЯ ОЦЕНКУ РИСКОВ, ЧТО САМО ПО СЕБЕ ЯВЛЯЕТСЯ ДОСТАТОЧНО СЛОЖНЫМ И ЗАТРАТНЫМ МЕРОПРИЯТИЕМ, А МОДЕЛЬ СИНХРОНИЗАЦИИ И СТАБИЛИЗАЦИИ (ВКЛЮЧАЕТ ДВА ДОСТАТОЧНО СЛОЖНЫХ ПРОЦЕССА: СИНХРОНИЗАЦИЮ И СТАБИЛИЗАЦИЮ)

СПИРАЛЬНАЯ МОДЕЛЬ ЖЦ (ВОЕНМ, 1987)



Каждый виток состоит из 4 фаз:

1. Определить цели: определить продукт, определить деловые цели, понять ограничения, предложить альтернативы
2. Оценить альтернативы: анализ риска, прототипирование
3. Разработать продукт: детальный проект, код, unit test, интеграция
4. Спланировать следующий цикл: оценка клиентом, планирование проектирования, поставка клиенту, внедрение

МОДЕЛЬ СИНХРОНИЗАЦИИ И СТАБИЛИЗАЦИИ (MICROSOFT)

Особенности:

3-4 инкрементных версии ПО, включающих:

- Синхронизацию (проверка, сборка, тестирование)
- Стабилизацию(устранение ошибок, найденных тестами)
- «Заморозку» - работающий «срез» ПО

Преимущества:

- «частое и раннее» тестирование (и выявление ошибок)
- Постоянная интероперабельность (модули тестируются в сборе => всегда есть работающая версия ПО => связи между модулями легко тестировать)
- Ранняя коррекция проекта (полная «сборка» ПО первых версий позволяет выявить недочеты проекта до полно-масштабной реализации и снизить стоимость редизайна)

СПЕЦИАЛИЗИРОВАННЫЕ МОДЕЛИ ЖЦ ПО : СРАВНИТЕЛЬНЫЙ АНАЛИЗ

Модель ЖЦ	Преимущества	Недостатки
Синхронизации и стабилизации	Удовлетворяет будущим потребностям клиента; обеспечивает интеграцию компонент	Не получила широкого применения вне Microsoft
Сpirальная	Объединяет характеристики всех перечисленных выше моделей	Пригодна лишь для крупных внутренних проектов; разработчики должны владеть управлением рисками

МОДЕЛЬ ФОРМАЛЬНОЙ РАЗРАБОТКИ СИСТЕМЫ

**Основана на разработке формальной
математической спецификации программной
системы и преобразования этой спецификации
посредством специальных математических
методов в исполняемые программы.**

**Проверка соответствия спецификации и
системных компонентов также выполняется
математическими методами**

МЕТОДОЛОГИЯ РАЗРАБОТКИ ПО

Методология – это система принципов, а также совокупность идей, понятий, методов, способов и средств, определяющих стиль разработки программного обеспечения.

Методологии могут включать различные модели в своей основе.

В полном научном смысле понятие методологии в области разработки ПО не совсем правильно. Это больше набор практических приемов, нет математических моделей, а часто и экономического обоснования.

**ОБЩИЕ
СВЕДЕНИЯ О
СЕМЕЙСТВЕ
СТАНДАРТОВ
12207**

В основе практически всех современных промышленных технологий создания ПС лежит международный стандарт ISO/IEC 12207 «Системная и программная инженерия. Процессы жизненного цикла программных средств.»

В состав семейства входят:

ISO/IEC 12207:1995 «Information technology–Software life cycle processes» с дополнениями и изменениями ISO/IEC 12207:1995/AMD 1:2002 и ISO/IEC 12207:2002/AMD 2:2004 (принят в новой редакции в 2008 году) ISO/IEC 12207:2008 «Systems and software engineering–Software life cycle processes»

ISO/IEC TR 15271:1998 Information technology – Guide for ISO/IEC 12207 (Software Life Cycle Processes)

ISO/IEC TR 16326:1999 Software engineering – Guide for the application of ISO/IEC 12207 to project management

СТАНДАРТЫ И ГОСТЫ В ОБЛАСТИ ИНФОРМАТИЗАЦИИ РБ

<http://nmo.basnet.by/documents/normative/standarts.php>

**Структура стандарта имеет гибкий,
модульный скелет, что позволяет быть
адаптируемым к потребностям детализации
для любого пользователя**

ОРГАНИЗАЦИЯ РАЗРАБОТКИ ТРЕБОВАНИЙ К СЛОЖНЫМ ПРОГРАММНЫМ СРЕДСТВАМ

ГРУППЫ ПРОЦЕССОВ

- **основные**
(заказ, поставка, разработка, эксплуатация, сопровождение);
- **вспомогательные**
(документирование, управление конфигурацией, обеспечение качества, верификация, аттестация, совместный анализ, аудит, разрешение проблем);
- **организационные**
(управление, создание инфраструктуры, усовершенствование, обучение).

Проекты программных средств различаются по

- уровню сложности,
- масштабу
- и необходимому качеству.

Они имеют различное

- назначение,
- содержание
- и относятся к разным областям применения.

Поэтому существует потребность в четко организованном процессе, методах формализации и управления требованиями к конкретным программным продуктам.

Чаще всего проблемами, с которыми встретились не достигшие своих целей проекты программных продуктов, являются:

- недостаток информации от пользователя или заказчика о функциях проекта,*
- неполные, некорректные требования,*
- а также многочисленные изменения требований и*



Формализация и управление требованиями – это систематический метод выявления, организации и документирования требований к ПС и/или ПО,

а также процесс, в ходе которого вырабатывается и обеспечивается соглашение между заказчиком и выполняющими проект специалистами, в условиях меняющихся требований к системе.

- Команда разработчиков должна применить методы и процессы для того, чтобы *понять решаемую проблему заказчика до начала разработки ПС.*
- Для этого следует использовать *метод анализа, выявления и освоения проблемы и интересов заказчика:*
- **достигнуть соглашения между заказчиком и разработчиком по определению проблемы, целей и задач проекта;**
- **выделить основные причины – проблемы, являющиеся её источниками и стоящие за основной проблемой проекта системы и ПС;**
- **выявить заинтересованных лиц и пользователей, чье коллективное мнение и оценка в конечном итоге определяет успех или неудачу проекта;**
- **определить, где приблизительно находятся область и границы возможных решений проблем;**
- **понять ограничения, которые будут наложены на проект, команду и решения проблем.**

Для сложных систем требуются стратегии управления информацией о требованиях.

Для этого применяется информационная иерархия; она начинается с потребностей пользователей, описанных с помощью функций, которые затем превращаются в более подробные требования к ПС, выраженные посредством прецедентов или традиционных форм описания и стандартизованных характеристик.

Эта иерархия отражает уровень абстракции при рассмотрении взаимосвязи области проблемы и области решения.

Концепцию требований, модифицированную в соответствии с конкретным содержанием комплекса программ, необходимо иметь в каждом проекте. В требованиях к ПС следует указывать, какие функции должны осуществляться, а не то, как они могут реализоваться. Они используются для задания функциональных и конструктивных требований, а также ограничений ресурсов проектирования.

Концепция требований к проекту (или системный проект) должна быть “живым” документом, чтобы было легче его использовать и пересматривать.

Следует сделать концепцию *официальным каналом* изменения функций так, чтобы проект всегда имел достоверный, соответствующий современному состоянию документ.

Проекты, как правило, инициируются с объемом функциональных возможностей, значительно превышающим тот, который разработчик может реализовать, обеспечив приемлемое качество при заданных ресурсах. Тем не менее, необходимо ограничиваться, чтобы иметь возможность предоставить в срок достаточно целостный и качественный продукт.

Существуют различные методы задания очередности выполнения (приоритетов) требований и понятие базового уровня – совместно согласованного представления о том, в чем будут состоять ключевые функции системы как продукта проекта – понятие состава требований, задающих *ориентир* для принятия решений и их оценки.

Если масштаб проекта и сопутствующие требования заказчика превышают реальные ресурсы, в любом случае придется ограничиваться в функциях и качестве ПС.

Поэтому следует определять, что обязательно должно быть сделано в версии программного продукта при имеющихся ресурсах проекта.

Для этого придется вести переговоры.

ПРОЦЕССЫ РАЗРАБОТКИ ТРЕБОВАНИЙ К ХАРАКТЕРИСТИКАМ СЛОЖНЫХ ПРОГРАММНЫХ СИСТЕМ

ПРИ ПРОЕКТИРОВАНИИ РЕКОМЕНДУЕТСЯ НАБОР КРИТЕРИЕВ КАЧЕСТВА ТРЕБОВАНИЙ К КОМПЛЕКСАМ ПРОГРАММ, КОТОРЫЙ ВКЛЮЧАЕТ:

- **корректность** – отсутствуют дефекты и ошибки в формулировках требований к комплексу программ;
- **недвусмысленность** – каждое требование должно быть однозначно и не допускать различного понимания и толкования специалистами;
- **полнота** – состав и содержание требований должны быть достаточны для производства и применения корректного комплекса программ и компонентов;
- **непротиворечивость** – между разными требованиями к компонентам и комплексу программ отсутствуют конфликты и противоречия;
- **модифицируемость** – каждое требование допускает возможность его простого и согласованного изменения и развития при производстве комплекса программ;
- **трассируемость** – требование имеет однозначный идентификатор и возможность детализации и перехода к производству компонента или комплекса программ.

**ПРИ ПЛАНИРОВАНИИ, РАЗРАБОТКЕ И РЕАЛИЗАЦИИ
ТРЕБОВАНИЙ К ХАРАКТЕРИСТИКАМ КАЧЕСТВА ПС
НЕОБХОДИМО, В ПЕРВУЮ ОЧЕРЕДЬ, УЧИТЬ ВАТЬ
СЛЕДУЮЩИЕ ОСНОВНЫЕ ФАКТОРЫ:**

- **функциональную пригодность
(функциональность) конкретного проекта
ПС;**
- **возможные конструктивные
характеристики качества комплекса
программ, необходимые для улучшения
функциональной пригодности;**
- **доступные ресурсы для создания и
обеспечения всего жизненного цикла ПО с
требуемым качеством.**

Требования к характеристикам комплексов программ, определяющие их функциональную пригодность, разделяются на две группы:

- *требования к количественным, измеряемым, функциональным характеристикам, непосредственно влияющим на оперативное функционирование и возможность применения программного продукта, в которые входят требования к надежности, безопасности, производительности, допустимым рискам применения;*
- *требования к структурным характеристикам, определяющим архитектуру комплекса программ, влияющие на возможности его модификации и сопровождения версий, на мобильность и переносимость на различные платформы, на документированность, удобство практического освоения и применения программного продукта вне оперативного функционирования.*

*Разработку и утверждение требований к
характеристикам и атрибутам качества без учета рисков,
целесообразно проводить итерационно на этапах
системного и детального проектирования ПС.*

На этапах проектирования последовательно должны определяться требования:

- при проектировании концепции – предварительные требования к назначению, функциональной пригодности и к номенклатуре необходимых конструктивных характеристик качества ПС;
- при предварительном проектировании – требования к шкалам и мерам применяемых атрибутов характеристик качества с учетом общих ограничений ресурсов;
- при детальном проектировании – подробные требования к атрибутам качества с детальным учетом и распределением реальных ограниченных ресурсов, а также, возможно, их оптимизация по критерию качество/затраты.

ЗАКАЗЧИК



ДИЗАЙНЕР



ВЕРСТАЛЬЩИК



ПРОГРАММИСТ



Эти требования закрепляются в контракте и техническом задании, по которым разработчик впоследствии должен отчитываться перед заказчиком при завершении проекта или его версии.

Выбор требований к характеристикам и атрибутам качества при проектировании программных систем начинается с *определения исходных данных*. Для корректного выбора и установления требований к характеристикам качества, прежде всего, необходимо *определить особенности проекта*:

- **класс, назначение и основные функции создаваемой ПС;**
- **комплект стандартов и их содержание, которые целесообразно использовать при выборе характеристик качества ПС;**
- **состав потребителей характеристик качества ПС, для которых важны соответствующие атрибуты качества;**
- **реальные ограничения всех видов ресурсов проекта.**



Этап разработки концепции проекта целесообразно начинать с формализации и обоснования набора исходных данных, отражающих общие особенности класса, потребителей и этапов жизненного цикла проекта ПС, каждый из которых влияет на выбор определенных характеристик качества комплекса программ.

Из конструктивных характеристик и атрибутов качества, прежде всего, следует выделять те, на которые в наибольшей степени воздействует класс, назначение и функции ПС.

Требования стандартов к функциональной пригодности должны выполняться для любых классов и назначения ПС. Однако номенклатура учитываемых требований к конструктивным характеристикам качества существенно зависит от назначения и функций комплексов программ. Так, например, при проектировании:

систем управления объектами в реальном времени наибольшее значение имеет защищенность, корректность и надежность функционирования стабильного комплекса программ и менее важно может быть качество обеспечения сопровождения и конфигурационного управления, способность к взаимодействию и практичность;

административных систем кроме корректности, важно обеспечивать практичность применения, комфортное взаимодействие с пользователями и внешней средой и может не иметь особого значение эффективность использования вычислительных ресурсов и обеспечение мобильности комплекса программ;

операционных систем наиболее жесткие требования предъявляются к корректности, эффективности использования вычислительных ресурсов и защищенности, и не всегда могут учитываться мобильность и унификация способности к взаимодействию её компонентов;

пакетов прикладных программ для вычислений или моделирования процессов, возможно не учитывать их мобильность, защищенность и временную эффективность, но особенно важна корректность.

СТРУКТУРА ОСНОВНЫХ ДОКУМЕНТОВ, ОТРАЖАЮЩИХ ТРЕБОВАНИЯ К ПРОГРАММНЫМ СИСТЕМАМ

При разработке требований к проектам программных средств, кроме основных целей, назначения и функций важно учесть и сформулировать содержание достаточно полного множества характеристик, каждая из которых может влиять на успех проекта программного продукта. Для уменьшения вероятности случайного пропуска важного требования заказчикам и пользователям целесообразно иметь типовые проекты перечней (шаблоны) наборов требований, которые можно целеустремленно сокращать и адаптировать, обеспечивая целостность требований для конкретных проектов ПС.

СОСТАВ КОНЦЕПЦИИ ОСНОВНЫХ ТРЕБОВАНИЙ К ПРОГРАММНОМУ СРЕДСТВУ:

- **описание обобщенных результатов обследования и изучения существующей системы и внешней среды;**
- **описание целей, назначения программного продукта и потребностей заказчика и потенциальных пользователей к нему в заданной среде применения;**
- **перечень базовых стандартов предполагаемого проекта программного продукта;**
- **общие требования к характеристикам комплекса задач ПС:**
 - цели создания программного продукта и назначение комплекса функциональных задач;
 - перечень объектов среды применения ПС (технологических объектов управления, подразделений предприятия и т. п.), при управлении которыми должен решаться комплекс задач;
 - периодичность и продолжительность решения комплекса задач;
 - связи и взаимодействие комплекса задач с внешней средой и другими компонентами системы;
 - распределение функций между персоналом, программными и техническими средствами при различных ситуациях решения требуемого комплекса функциональных задач;



СОСТАВ КОНЦЕПЦИИ ОСНОВНЫХ ТРЕБОВАНИЙ К ПРОГРАММНОМУ СРЕДСТВУ:

- **сопоставительный анализ требований заказчика и пользователей к программному продукту и набора функций в концепции ПС для удовлетворения требований заказчика и пользователей;**
- **обоснование выбора оптимального варианта требований к содержанию и приоритетам комплекса функций ПС в концепции;**
- **общие требования к структуре, составу компонентов и интерфейсам с внешней средой;**
- **ожидаемые результаты и возможная эффективность реализации выбранного варианта требований в концепции ПС;**
- **ориентировочный план реализации выбранного варианта требований концепции ПС;**
- **общие требования к составу и содержанию документации проекта ПС;**
- **оценка необходимых затрат ресурсов на разработку, ввод в действие и обеспечение функционирования ПС;**
- **предварительный состав требований, гарантирующих качество применения ПС;**
- **предварительные требования к условиям испытаний и приемки системы и ПС.**

СОСТАВ КОНЦЕПЦИИ ОСНОВНЫХ ТРЕБОВАНИЙ К ПРОГРАММНОМУ СРЕДСТВУ:

требования к входной информации:

- источники информации и их идентификаторы;
- перечень и описание входных сообщений (идентификаторы, формы представления, регламент, сроки и частота поступления);
- перечень и описание структурных единиц информации входных сообщений или ссылка на документы, содержащие эти данные;

требования к выходной информации:

- потребители и назначение выходной информации;
- перечень и описание выходных сообщений;
- регламент и периодичность их выдачи;
- допустимое время задержки решения определенных задач;

описание и оценка преимуществ и недостатков разработанных альтернативных вариантов функций в концепции создания проекта ПС;

СПЕЦИФИКАЦИЯ ТРЕБОВАНИЙ К СИСТЕМЕ И К КОМПЛЕКСУ ПРОГРАММ НА ЭТАПЕ ДЕТАЛЬНОГО ПРОЕКТИРОВАНИЯ:

- **требования проекта системы к комплексу программ, как к целому в общей архитектуре системы;**
- **требования к унификации интерфейсов и базы данных комплекса программ;**
- **требования и обоснование выбора проектных решений уровня системы, состава компонентов системы, описание функций системы и ПС с точки зрения пользователя;**
- **спецификация требований верхнего уровня комплекса программ, производные требования к компонентам ПС и требования к интерфейсам между системными компонентами, элементами конфигурации ПС и аппаратуры;**
- **описание распределения системных требований по компонентам ПС с учетом требований, которые обеспечивают заданные характеристики качества;**
- **требования к архитектуре системы, содержащей идентификацию и функции компонентов системы, их назначение, статус разработки, аппаратные и программные ресурсы;**
- **требования совместного целостного функционирования компонентов ПС, описание и характеристики их динамических связей;**

СПЕЦИФИКАЦИЯ ТРЕБОВАНИЙ К СИСТЕМЕ И К КОМПЛЕКСУ ПРОГРАММ НА ЭТАПЕ ДЕТАЛЬНОГО ПРОЕКТИРОВАНИЯ

требования анализа трассируемости функций компонентов программного средства к требованиям проекта системы;

требования для системы или/и подсистем и методы, которые должны быть использованы для гарантии того, что каждое требование к комплексу программ будет выполнено и прослеживаемо к конкретным требованиям системы:

- к режимам работы;
- к производительности системы;
- к внешнему и пользовательскому интерфейсу системы;
- к внутреннему интерфейсу компонентов и к внутренним данным системы;
- по возможности адаптации ПС к внешней среде;
- по обеспечению безопасности системы, ПС и внешней среды;
- по обеспечению защиты, безопасности и секретности данных;
- по ограничениям доступных ресурсов проекта ПС;
- по обучению и уровню квалификации персонала;
- по возможностям средств аттестации результатов и компонентов, включающие в себя демонстрацию, тестирование, анализ, инспекцию и требуемые специальные методы для контроля функций, и качества конкретной системы или компонента ПС.

СПАСИБО ЗА ВНИМАНИЕ!

ПРОЕКТИРОВАНИЕ ПРОГРАММНЫХ СИСТЕМ

УНИФИЦИРОВАННЫЙ ЯЗЫК МОДЕЛИРОВАНИЯ UML

ЛЕКЦИЯ 2

- **Важный принцип создания проектов на многих ООП языках программирования – сначала проектирование, а потом программирование**
- **На этапе проектирования и происходит использование UML-диаграмм (самый популярный инструмент)**

ЧТО ТАКОЕ UML

- **UML – аббревиатура полного названия Unified Modeling Language.**
- **Правильный перевод этого названия на русский язык – унифицированный язык моделирования.** Таким образом, обсуждаемый предмет характеризуется тремя словами, каждое из которых является точным термином.

ЧТО ТАКОЕ UML

Язык — это знаковая система для хранения и передачи информации.

UML — язык графического описания для объектного моделирования в области разработки программного обеспечения. UML является языком широкого профиля, это — открытый стандарт, использующий графические обозначения для создания абстрактной модели системы, называемой UML-моделью.

ЧТО ТАКОЕ UML

Спецификация – это декларативное описание того, как нечто устроено или работает.

Основное назначение UML – предоставить, с одной стороны, достаточно формальное, с другой стороны, достаточно удобное, и, с третьей стороны, достаточно универсальное средство, позволяющее до некоторой степени снизить риск расхождений в толковании спецификаций

КТО МОЖЕТ ИСПОЛЬЗОВАТЬ UML

- Заказчик – общие цели и задачи проекта, что будет выполнять система
- Аналитик – проверяет подходы, правильность работы системы и ее частей, «слои» приложения
- Разработчик/архитектор – чаще всего дизайн кода, архитектура классов, объектов, взаимодействий
- Тестировщик – может проверять все уровни, также производительность по диаграммам времени
- Менеджер – общая картина проекта, самый верхний уровень

- UML задает описание или поведение процесса, но не показывает реализацию
- Цели UML – проектирование, документирование, визуальное описание основных моментов проекта
- Ключевое понятие – диаграмма (есть разные типы) – визуальное описание процесса, классов, взаимодействия и пр.
- На каждый процесс или тип задачи – своя диаграмма
- UML не является языком программирования (но можно генерировать готовый код из диаграмм)
- Хранение информации в электронном виде (легче программировать, так как не надо держать в голове проект)
- Позволяет посмотреть на проект с верхнего уровня – сразу видны ключевые моменты
- Дополняет стандартные документы типа ТЗ
- Нужно хорошо знать ООП

UML - ЭТО ЯЗЫК

Язык — это знаковая система для хранения и передачи информации.

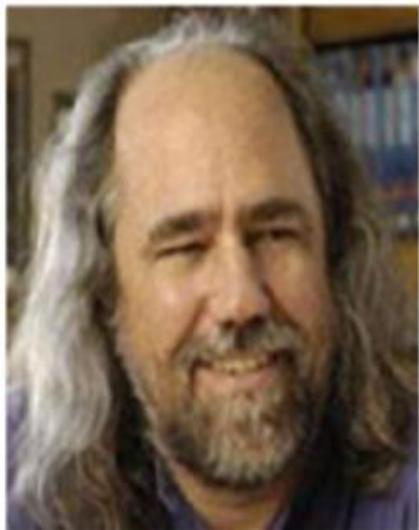
Формальный искусственный язык описан, если описание содержит:

- Синтаксис (*syntax*).
- Семантика (*semantics*).
- Прагматика (*pragmatics*).

УНИФИЦИРОВАННЫЙ ЯЗЫК МОДЕЛИРОВАНИЯ UML

Unified Modeling Language (UML) - это язык для определения, представления, проектирования и документирования программных систем, организационно-экономических систем, технических систем и других систем различной природы.

ИСТОРИЯ РАЗВИТИЯ UML



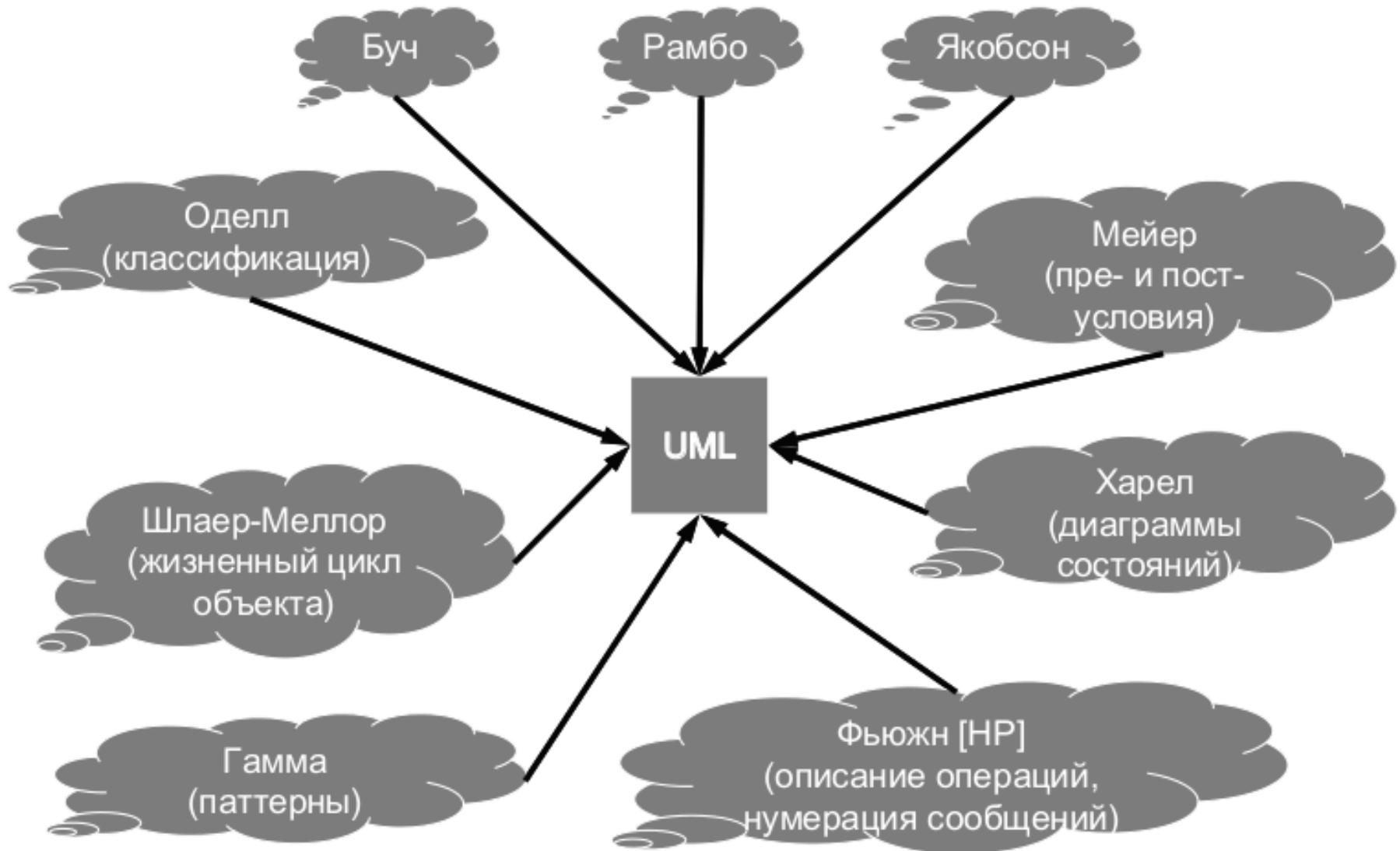
Grady Booch
Грэди Буч



James Rumbaugh
Джеймс Рамбо



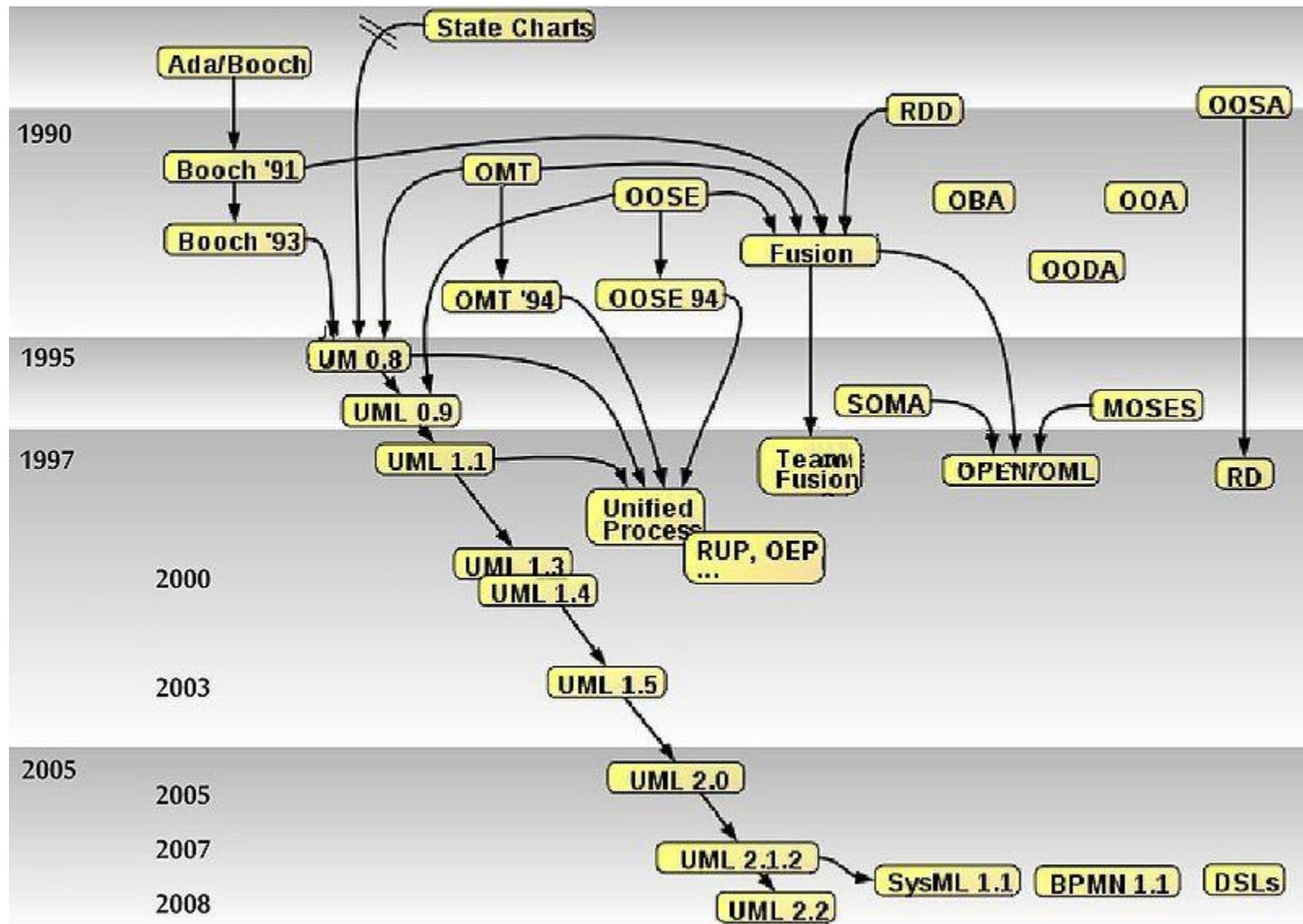
Ivar Jacobson
Айвар Якобсон



Вклад в развитие UML

ЦЕЛИ РАЗВИТИЯ UML

- предоставить разработчикам единый язык визуального моделирования;
- предусмотреть механизмы расширения и специализации языка;
- обеспечить независимость языка от языков программирования и процессов разработки;
- интегрировать накопленный практический опыт.



История развития UML

ИНСТРУМЕНТЫ ДЛЯ РАЗРАБОТКИ НА UML

- Онлайн инструмент
- Отдельные программы
- Плагины
- Специальные IDE

БЛОКИ UML

- **элементы модели (классы, интерфейсы, компоненты, варианты использования и др.);**
- **связи (ассоциации, обобщения, зависимости и др.);**
- **механизмы расширения (стереотипы, ограничения, метасвойства, примечания);**
- **диаграммы.**

СОСТАВ ДИАГРАММ UML 1.X

★ структурные:

- диаграммы классов
- диаграммы компонентов
- диаграммы размещения

★ поведенческие:

- диаграммы вариантов использования
- диаграммы взаимодействия:
 - диаграммы последовательности
 - коммуникационные диаграммы
- диаграммы состояний
- диаграммы деятельности

КЛАССИФИКАЦИЯ ДИАГРАММ В UML 2.X

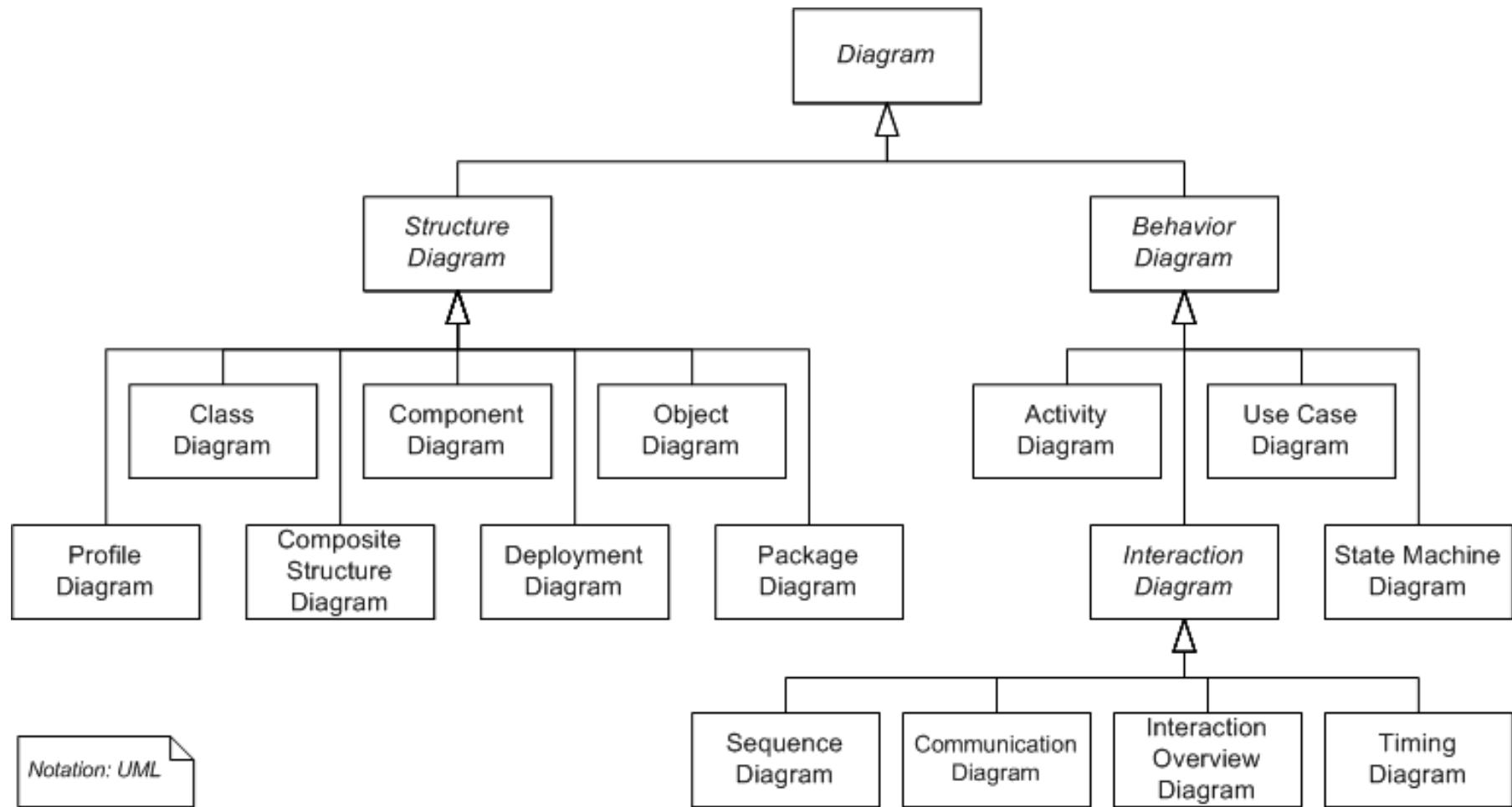
Диаграммы поведения:

- Диаграммы вариантов использования
- Диаграммы деятельности
- Диаграммы состояний
- Диаграммы взаимодействия:

- Диаграммы последовательности
- Диаграммы коммуникации
- Диаграммы синхронизации
- Обзорные диаграммы взаимодействия

Структурные диаграммы:

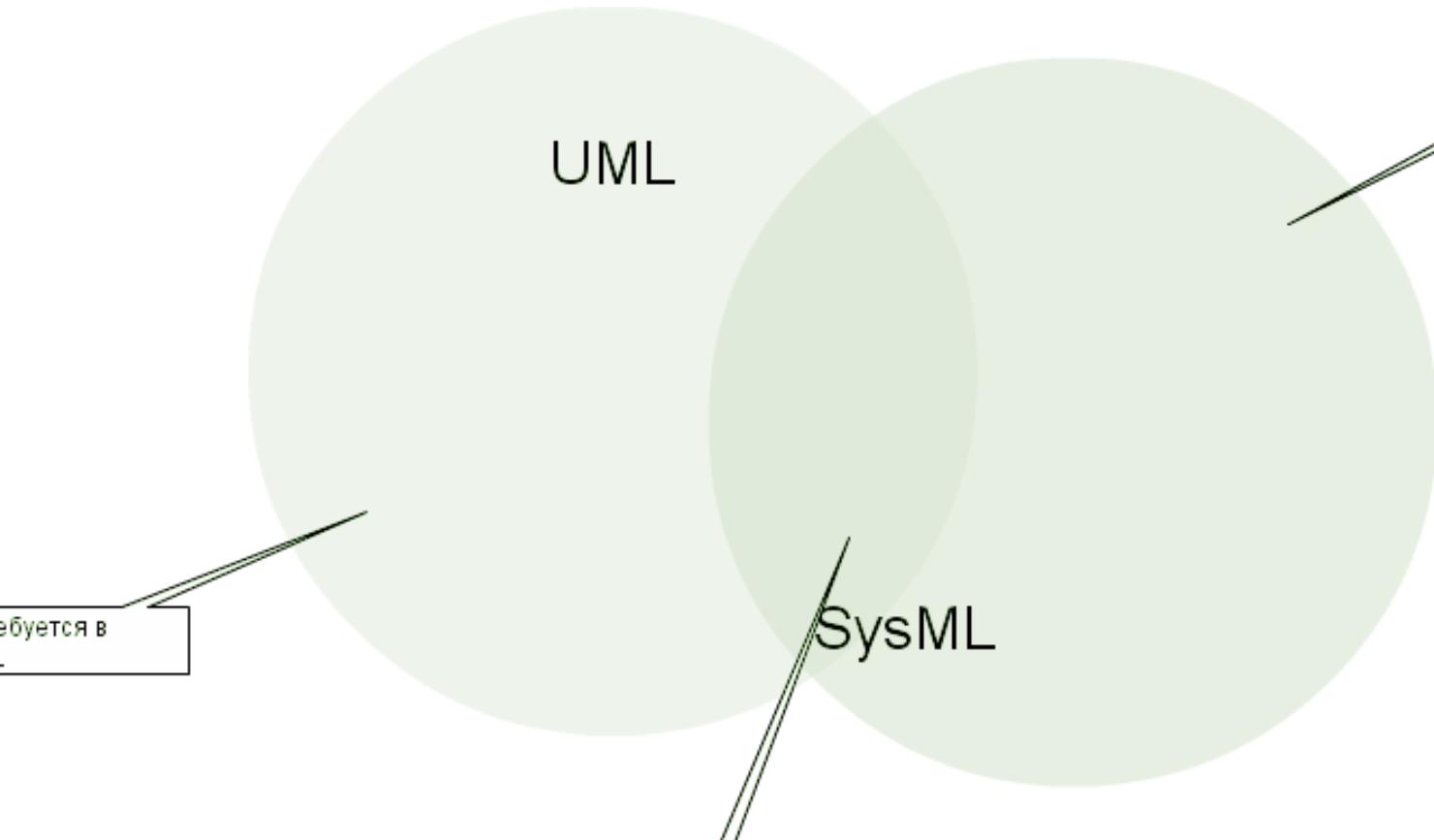
1. Диаграммы классов
2. Диаграммы объектов
3. Диаграммы пакетов
4. Диаграммы компонентов
5. Диаграммы составной структуры
6. Диаграммы размещения
7. Диаграммы профиля



Состав диаграмм UML 2.x

SYSTEMS MODELING LANGUAGE (SYSML)

SysML (англ. *The Systems Modeling Language*, язык моделирования систем) — предметно-ориентированный язык моделирования систем. Поддерживает определение, анализ, проектирование, проверку и подтверждение соответствия широкого спектра систем.

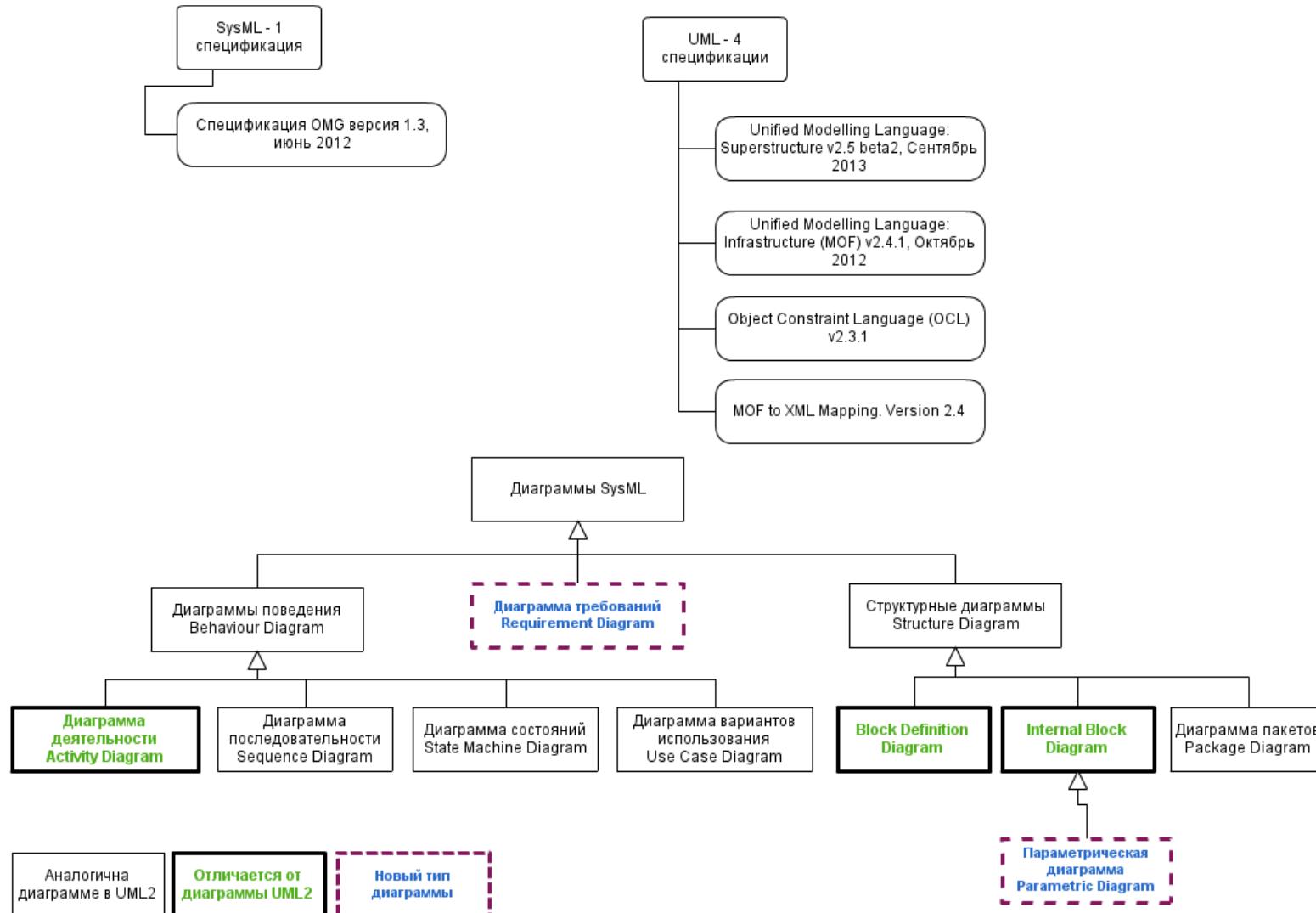


UML vs SysML

ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ SYSML

- Дополнительные диаграммы: диаграмма требований и параметрическая диаграмма.
- SysML - более компактный язык.
- Конструкции для управления моделями. поддерживает модели, представления (views) и точки зрения (viewpoints).

СПЕЦИФИКАЦИИ И СОСТАВ ДИАГРАММ SYSML



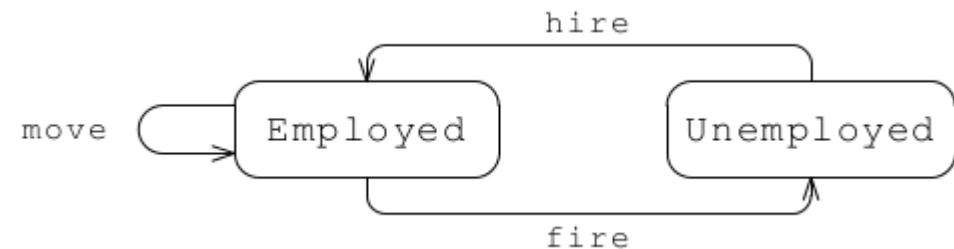
Одним из ключевых этапов разработки приложения является определение того, каким требованиям должно удовлетворять разрабатываемое приложение. В результате этого этапа появляется формальный или неформальный документ (артефакт), который называют по-разному, имея в виду примерно одно и то же: постановка задачи, требования, техническое задание, внешние спецификации и др.

Спецификация — это декларативное описание того, как нечто устроено или работает.

НАЗНАЧЕНИЕ UML

Язык UML — это графический язык моделирования общего назначения, предназначенный для спецификации, визуализации, проектирования и документирования всех артефактов, создаваемых при разработке программных систем.

1. Спецификация
2. Визуализация
3. Проектирование
4. Документирование



МОДЕЛЬ И ЕЕ ЭЛЕМЕНТЫ

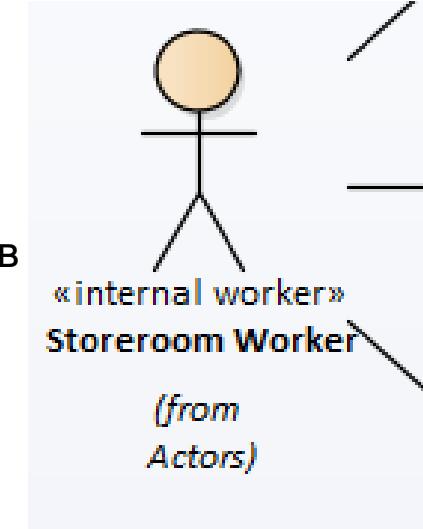
Модель UML (**UML model**) — это совокупность конечного множества конструкций языка, главные из которых — это сущности и отношения между ними.

- структурные
- поведенческие
- группирующие
- аннотационные

СТЕРЕОТИП

Стереотип – это определение нового элемента моделирования в UML на основе существующего элемента моделирования.

Определение стереотипа производится следующим образом. Взяв за основу некоторый существующий элемент модели, к нему добавляют новые помеченные значения (расширяя тем самым внутреннее представление), новые ограничения (расширяя семантику) и дополнения, то есть новые графические элементы (расширяя нотацию).



После того, как стереотип определен, его можно использовать как элемент модели нового типа.

Если при создании стереотипа не использовались дополнения и графическая нотация взята от базового элемента модели, на основе которого определен стереотип, то стереотип элемента обозначается при помощи имени стереотипа, заключенного в двойные угловые скобки (типографские кавычки), которое помещается перед именем элемента модели.

Если же для стереотипа определена своя нотация, например, новый графический символ, то указывается этот символ.

СТРУКТУРНЫЕ СУЩНОСТИ UML

- Объект
- Класс
- Интерфейс
- Кооперация
- Действующее лицо
- Компонент
- Артефакт
- Узел

ПОВЕДЕНЧЕСКИЕ И ДРУГИЕ СУЩНОСТИ

Поведенческие сущности:

- **Состояние**
- **Деятельность**
- **Действие**

Структурная и поведенческая сущность:

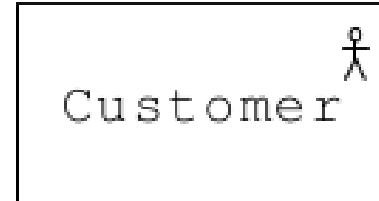
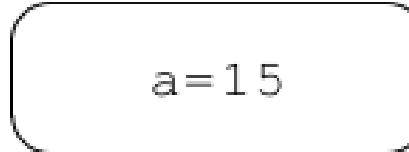
- **Вариант использования (ВИ)**

Группирующая сущность:

- **Пакет**

Аннотационная сущность:

- **Аннотация**

Название	Графическая нотация	
Артефакт	<p>«artifact»</p> <p>Requirement Specification</p>	<p>«library»</p> <p>QT</p>
Вариант использования	 <p>Make Order</p>	 <p>Make Order</p>
Действующее лицо	 <p>Customer</p>	 <p>Customer</p>
Деятельность и действие	 <p>Display main menu</p>	 <p>a=15</p>

Нотации основных сущностей

Название	Графическая нотация	
Интерфейс	IAudio	«interface» IAudio
Класс	Product	Order (abstract)
Компонент	DataBase	«component» DataBase
Кооперация	Visitor	

Нотации основных сущностей

Название	Графическая нотация
Объект	:Rectangle
Пакет	Analysis Model
Примечание	Здесь находится комментарий
Состояние	Logged entry/OpenLog exit/CloseLog
Узел	Server

Нотации основных сущностей

ТИПЫ ОТНОШЕНИЙ В UML

- зависимость
- ассоциация
- обобщение
- реализация

ДИАГРАММЫ ВАРИАНТОВ ИСПОЛЬЗОВАНИЯ И СЦЕНАРИИ

КЛАССИФИКАЦИЯ ДИАГРАММ

Диаграммы поведения:

- Диаграммы вариантов использования
- Диаграммы деятельности
- Диаграммы состояний
- Диаграммы взаимодействия:

- Диаграммы последовательности
- Диаграммы коммуникации
- Диаграммы синхронизации
- Обзорные диаграммы взаимодействия

Структурные диаграммы:

1. Диаграммы классов
2. Диаграммы объектов
3. Диаграммы пакетов
4. Диаграммы компонентов
5. Диаграммы составной структуры
6. Диаграммы размещения
7. Диаграммы профиля

ПРОЦЕСС ВИЗУАЛЬНОГО МОДЕЛИРОВАНИЯ

Визуальное моделирование с использованием нотации UML можно представить как процесс поуровневого спуска от наиболее общей и абстрактной концептуальной модели исходной бизнес-системы к логической, а затем и к физической модели соответствующей программной системы.

ДИАГРАММА ВАРИАНТОВ ИСПОЛЬЗОВАНИЯ (USE CASE DIAGRAM)

- 1) Исходное концептуальное представление или концептуальная модель системы в процессе ее проектирования и разработки.**
- 2) Диаграмма, на которой изображаются отношения между действующими лицами и вариантами использования.**

ЦЕЛИ СОЗДАНИЯ USE CASE DIAGRAM

- Определить общие границы и контекст моделируемой предметной области на начальных этапах проектирования системы
- Сформулировать общие требования к функциональному поведению проектируемой системы
- Разработать исходную концептуальную модель системы для ее последующей детализации в форме логических и физических моделей
- Подготовить исходную документацию для взаимодействия разработчиков системы с ее заказчиками и пользователями

ФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ И USE CASE

Диаграмма вариантов использования - общее представление функциональных требований к системе.

Детализация функциональных требований - описания вариантов использования. Каждый документ содержит один и более сценариев или потоков событий варианта использования.

ОБЯЗАТЕЛЬНЫЕ ЭЛЕМЕНТЫ ДИАГРАММ ВИ

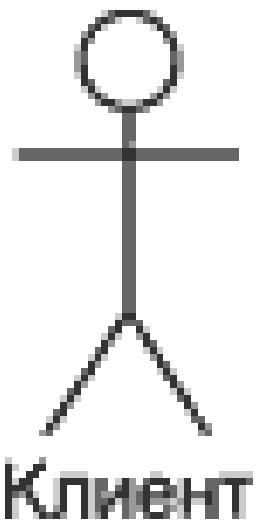
- Название моделируемой системы
- Границы системы
- Варианты использования
- Действующие лица
- Связи

ОСНОВНЫЕ ПОНЯТИЯ

- **Actor** – пользователь, действующее лицо (инициатор или исполнитель цели)
- **UseCase** – прецедент, цель, которую хочет достичь пользователь
- **Association** – ассоциация, связь между элементами (акторами и целями)
- **Include** – включения; возможные варианты реализации цели (или набор целей для реализации общей цели)
- **Extend** – расширения; новая цель, которая расширяет существующую

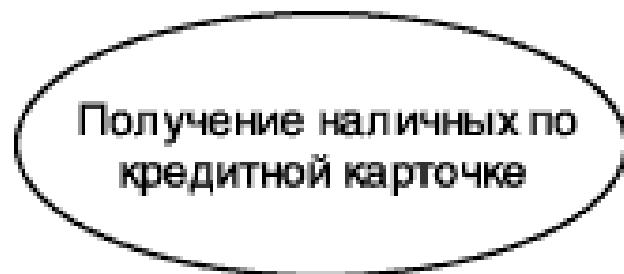
ДЕЙСТВУЮЩЕЕ ЛИЦО (дл)

Действующее лицо (actor) — согласованное множество ролей, которые играют внешние сущности по отношению к вариантам использования при взаимодействии с ними.

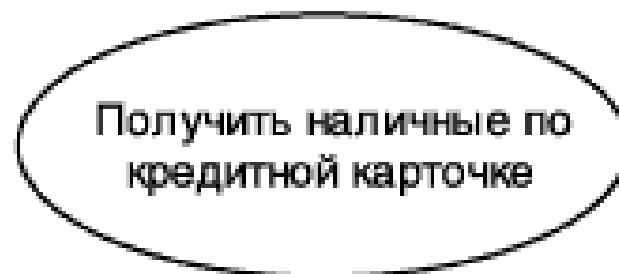


ВАРИАНТ ИСПОЛЬЗОВАНИЯ ИЛИ ПРЕЦЕДЕНТ (USE CASE)

Внешняя спецификация последовательности действий, которые система или другая сущность могут выполнять в процессе взаимодействия с действующими лицами.



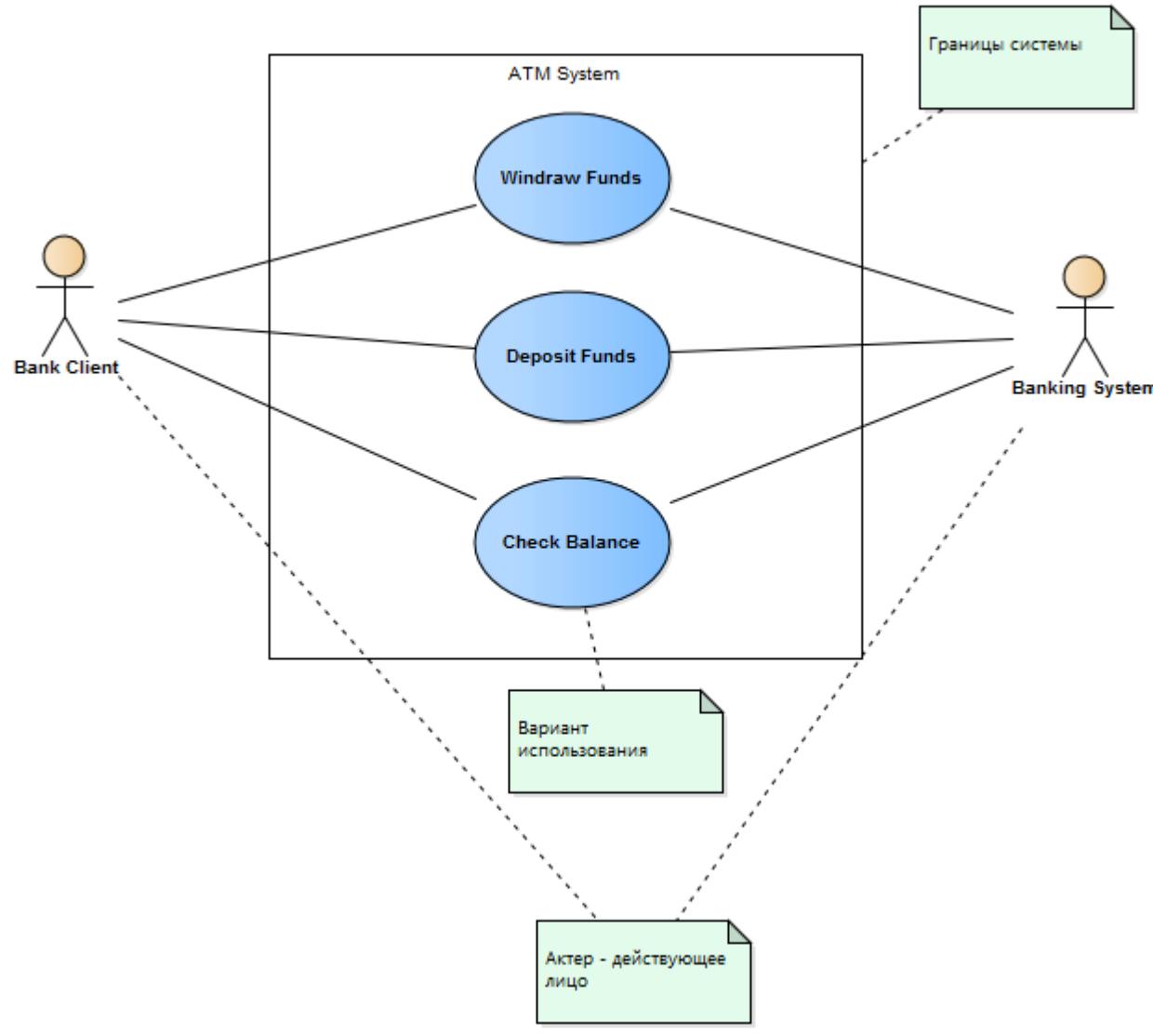
(a)



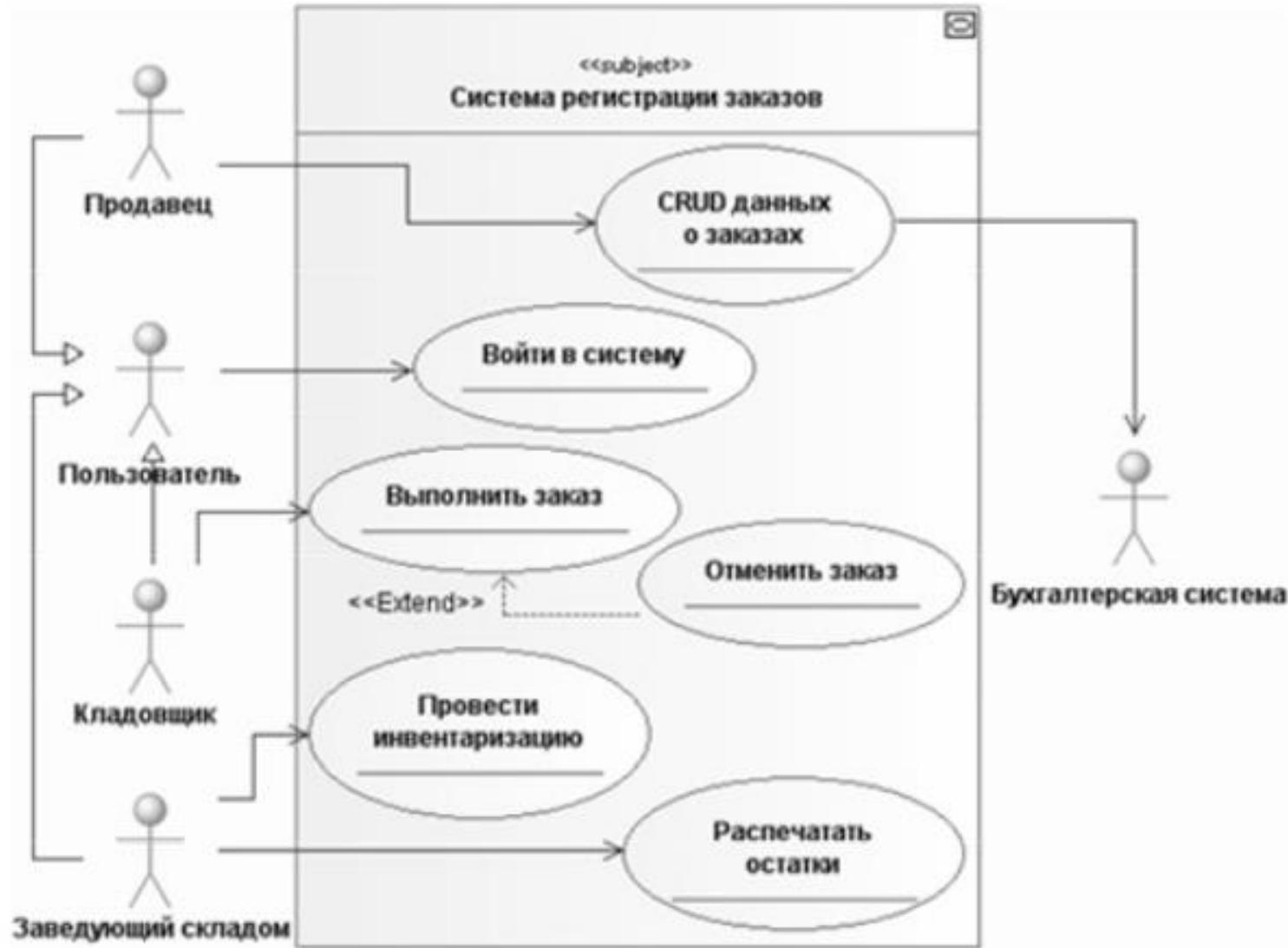
(б)

ПРИМЕРЫ ВИ

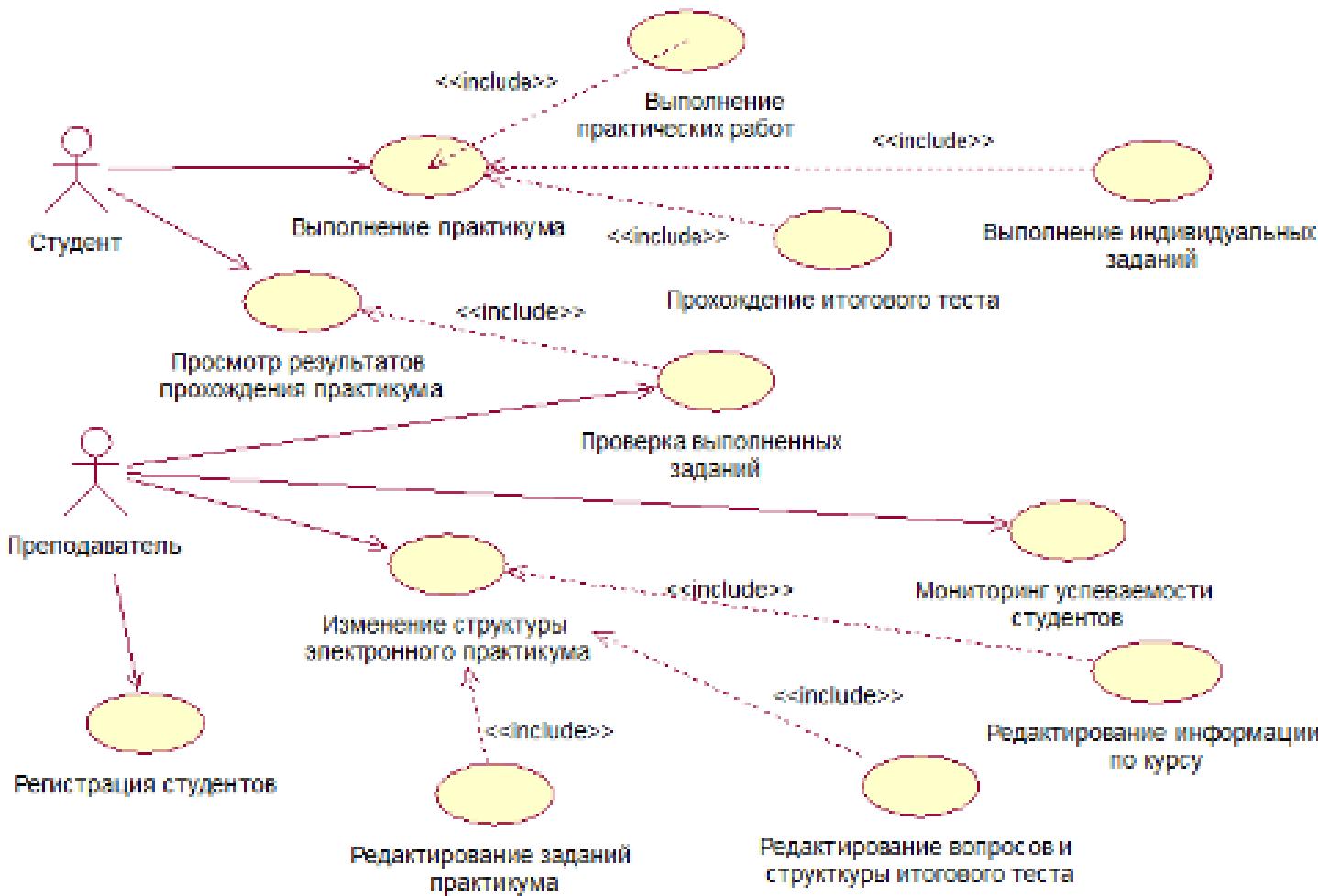
- Проверка состояния текущего счета клиента,
- Оформление заказа на покупку товара,
- Получение дополнительной информации о кредитоспособности клиента,
- Отображение графической формы на экране монитора.



Пример диаграммы вариантов использования



Пример диаграммы вариантов использования



Пример диаграммы вариантов использования

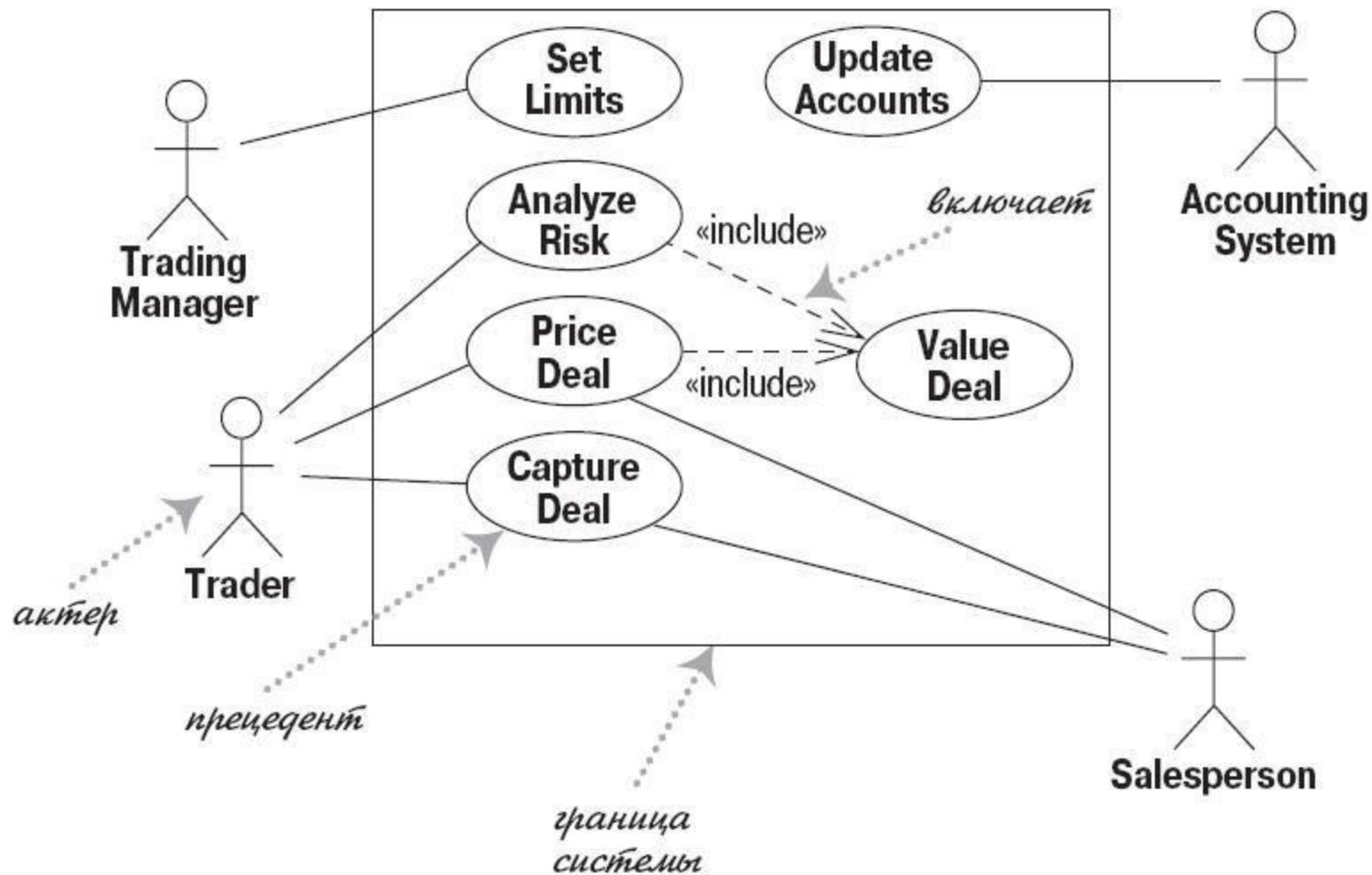
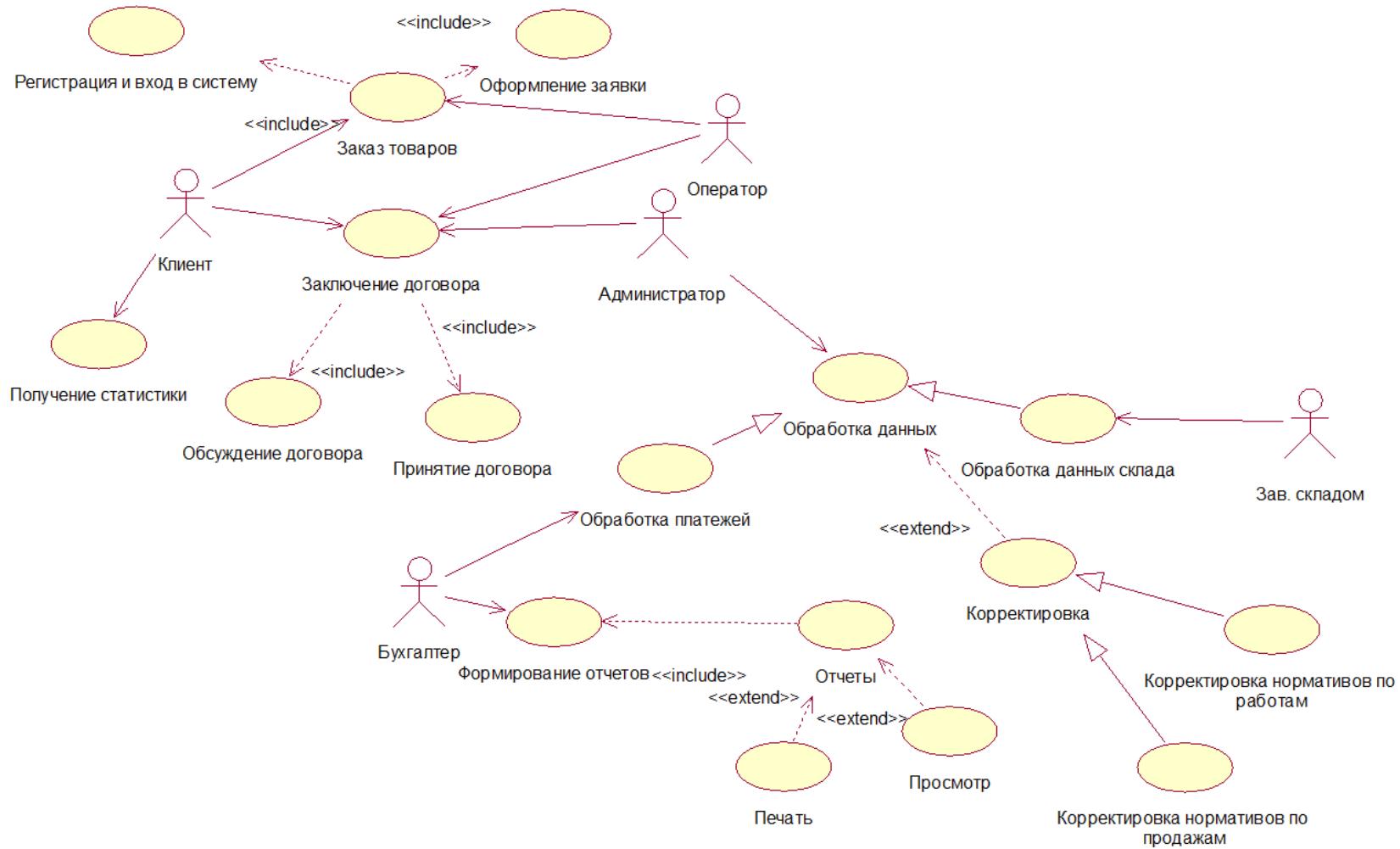
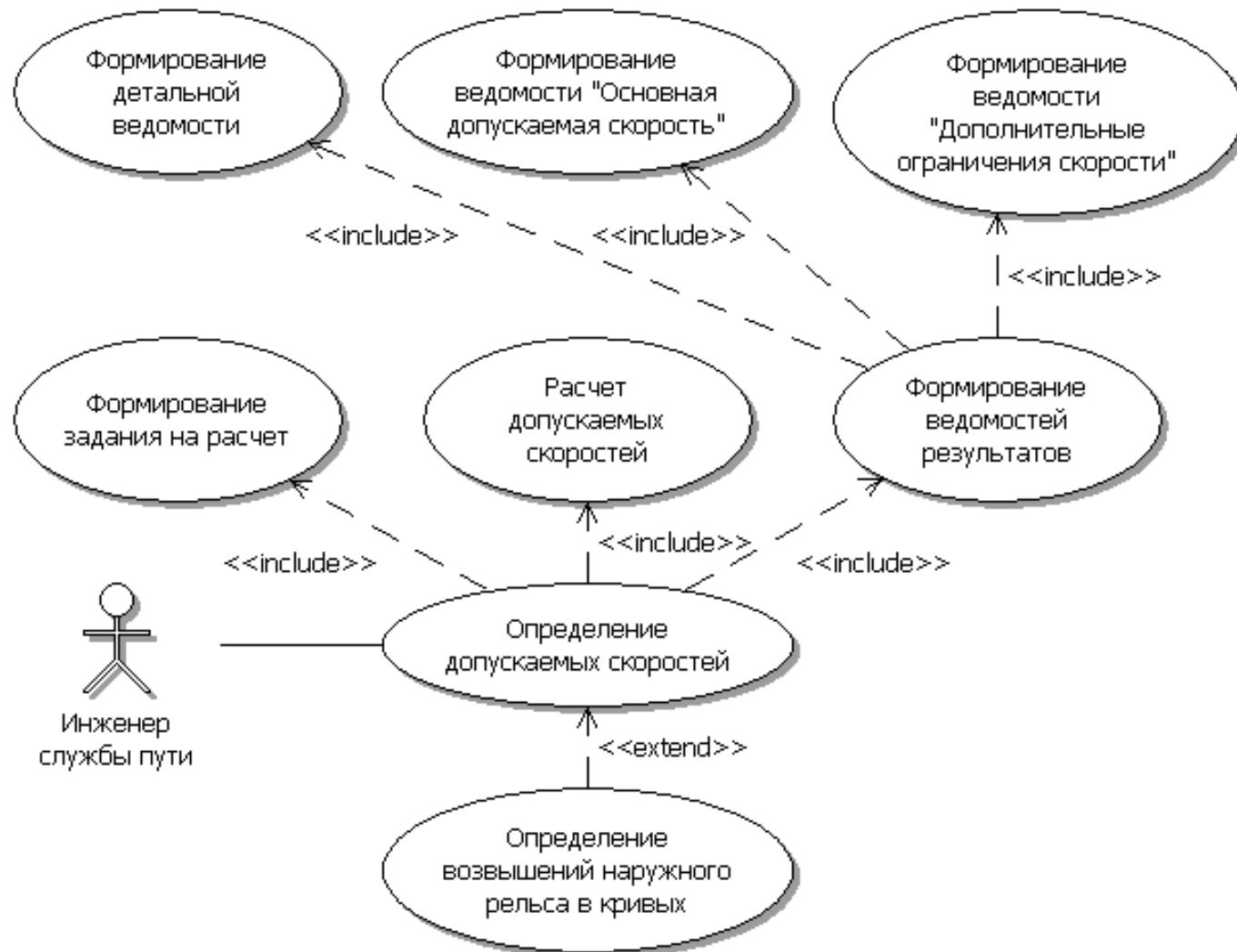


Рис. 9.2. Диаграмма прецедентов

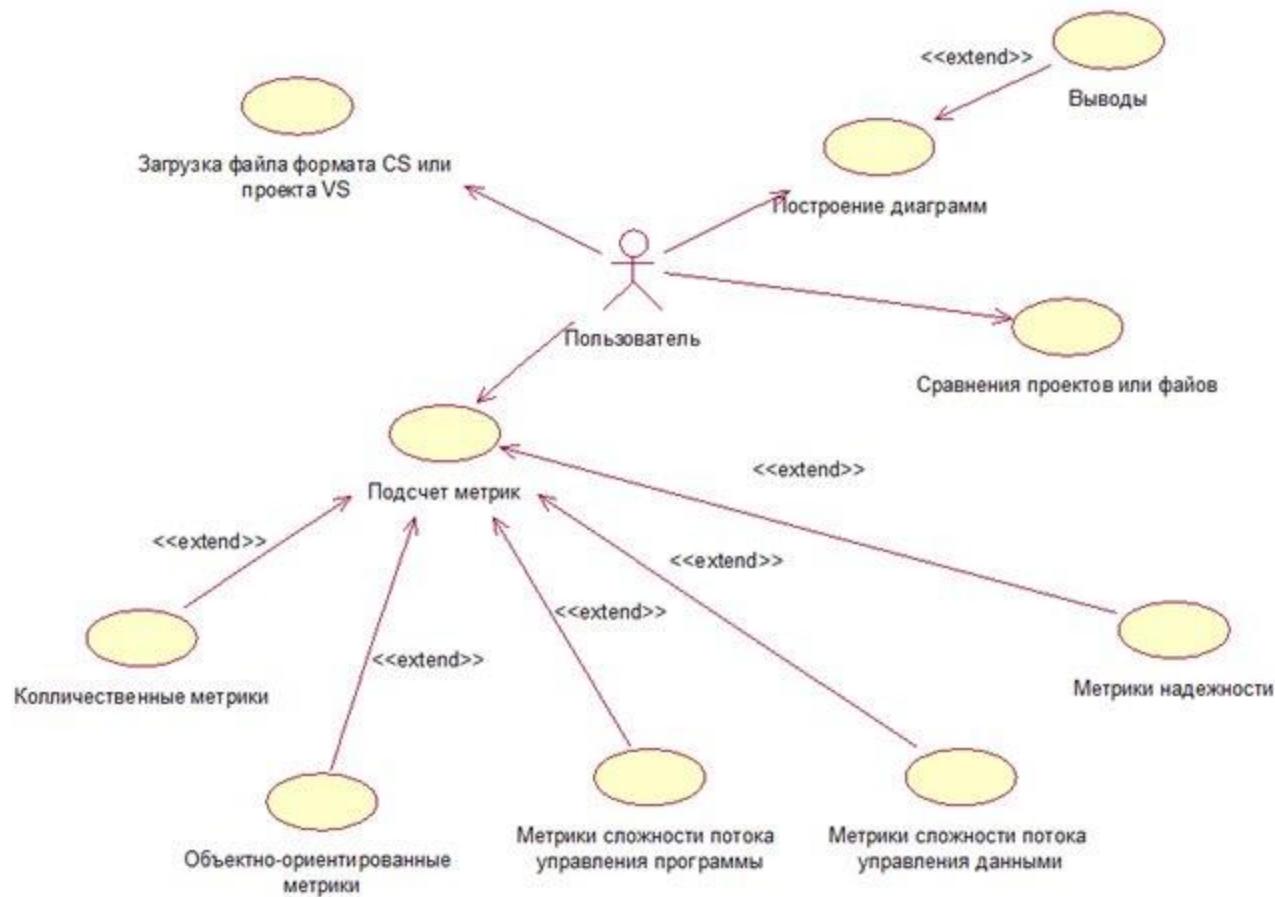
Пример диаграммы вариантов использования



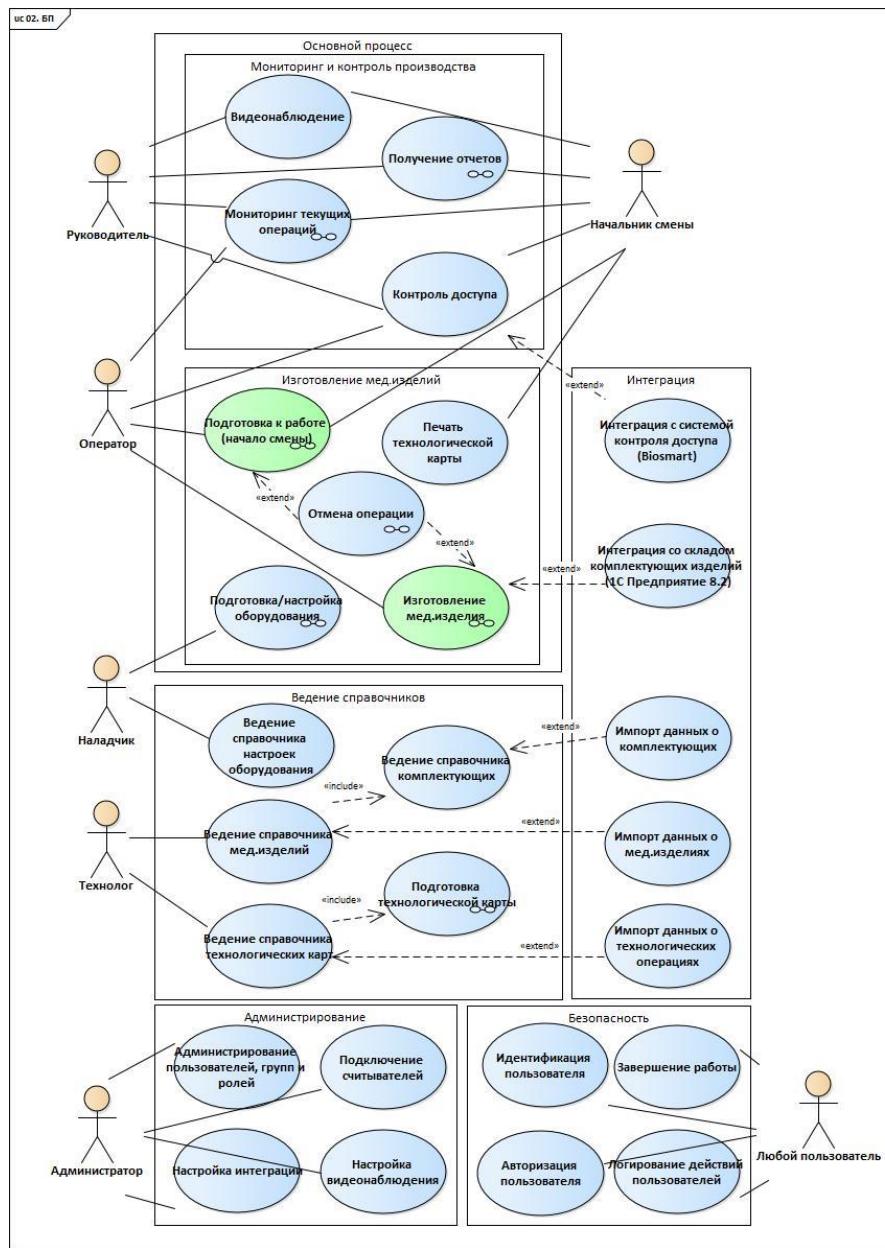
Пример диаграммы вариантов использования



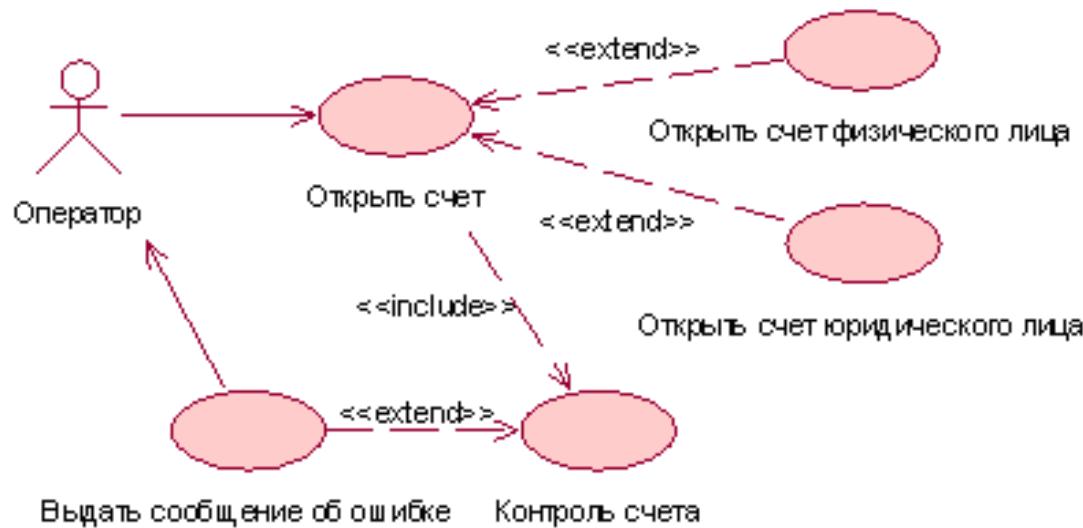
Пример диаграммы вариантов использования



Пример диаграммы вариантов использования



Пример диаграммы вариантов использования



Пример диаграммы вариантов использования

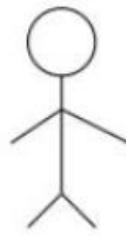
Вариант использования (use case) — внешняя спецификация последовательности действий, которые система или другая сущность могут выполнять в процессе взаимодействия с действующими лицами.

Вариант использования представляет собой спецификацию общих особенностей поведения или функционирования моделируемой системы без рассмотрения внутренней структуры этой системы.

Содержание варианта использования может быть представлено в форме дополнительного пояснительного текста, который раскрывает смысл или семантику действия при выполнении данного варианта использования. Такой пояснительный текст получил название **текста-сценария или просто **сценария**.**

ЦЕЛЬ СПЕЦИФИКАЦИИ (СЦЕНАРИЯ) ВИ

Цель спецификации варианта использования заключается в том, чтобы зафиксировать некоторый аспект или фрагмент поведения проектируемой системы без указания *особенностей реализации данной функциональности.*



Актер



ВИ на диаграмме



ВИ в текстовом виде

ВИ и спецификации (сценарии)

ТИПЫ СЦЕНАРИЕВ

Основной сценарий представляет собой последовательность действий, при успешном выполнении которых достигается цель варианта использования.

Расширения основного сценария описывают действия при возникновении исключительных ситуаций (действий, не предусмотренных основным сценарием, ошибках, внешних событиях и т.д.)

ОПИСАНИЕ СЦЕНАРИЕВ

Основной сценарий описывается в виде:

[Номер шага]. [Действие]

Расширения привязываются к определенным шагам основного сценария и представлены в виде:

**[Номер шага+идентификатор расширения]. [условие]:
[Действие или вложенный вариант использования]**

СПЕЦИФИКАЦИИ

- 1. Название**
- 2. Основные актеры**
- 3. Триггер (с чего начинается ВИ?)**
- 4. Предусловия**
- 5. Постусловия**
- 6. Область действия**
- 7. Уровень цели**
- 8. Минимальные гарантии**
- 9. Контекст использования**



Вариант использования:	Выполнить вход в систему
Контекст использования:	Пользователь совершает вход в систему
Область:	Система
Уровень:	Цель пользователя
Основной актер:	Пользователь
Предусловие:	Нет
Успешное постусловие:	Пользователю предоставлен доступ в систему
Минимальные гарантии:	Пользователю не предоставлен доступ в систему
Триггер:	Окно входа в систему

Основной сценарий

1. Пользователь вводит логин и пароль
2. Пользователь запускает проверку
3. Система проверяет логин
4. Система проверяет пароль
5. Система предоставляет пользователю доступ

Расширения

3.а. Не найдена учетная запись с таким логином:

- 3.а.1. Система уведомляет об ошибке
- 3.а.2. Возврат сценария на пункт 1

4.а. Пароль не верный:

- 4.а.1. Система увеличивает счетчик неудачных попыток входа.
- 4.а.2. Система проверяет количество неудачных попыток входа

4.а.1.а. Количество неудачных попыток больше установленного предела:

- 4.а.1.а.1. Система уведомляет о блокировке
- 4.а.1.а.2. Завершение сценария

- 4.а.3. Система уведомляет об ошибке
- 4.а.4. Возврат сценария на пункт 1

Пример спецификации варианта использования по Коберну

Ребекка Вирфс-Брок (Rebecca Wirfs-Brock)

American software engineer and consultant in object-oriented programming and object-oriented design, the founder of the information technology consulting firm Wirfs-Brock Associates, and inventor of Responsibility-Driven Design, the first behavioral approach to object design.



Ребеккой Вирфс-Брок было предложено оформлять сценарии варианта использования в виде диалога между основным актером и системой (не предполагается участие в одном варианте использования более двух актеров). Сценарий записывается в форме таблицы, состоящей из двух колонок:

Действия основного актера

Действия системы

Ребекка Вирфс-Брок

Двухколоночная таблица Вирфс-Брок. Пример описания основного сценария варианта использования «Выполнить вход в систему»

Действия пользователя

Действия системы

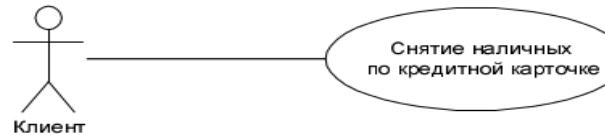
1. Вводит логин и пароль
2. Запускаем проверку

3. Проверяет логин
 4. Проверяет пароль
 5. Предоставляет доступ
-

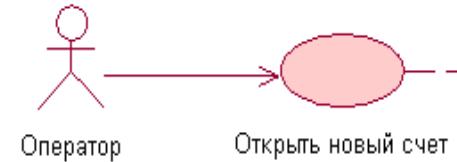
Пример описания ВИ по Вирфс-Брок

СВЯЗИ НА ДИАГРАММАХ ВИ

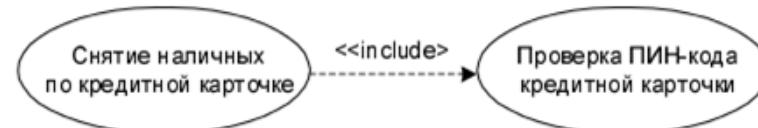
- Ассоциация



- Направленная ассоциация



- Отношение включения (include)

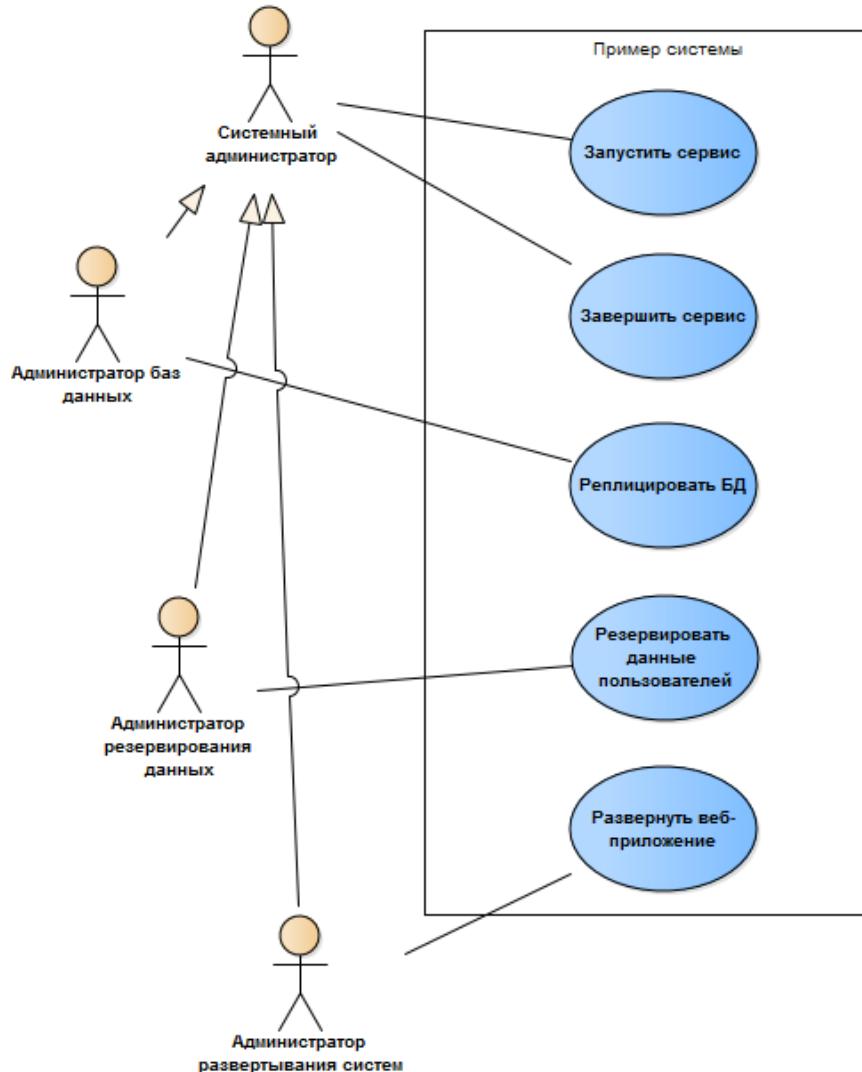


- Отношение расширения (extend)



- Отношение обобщения





Пример обобщения актеров

ПРИМЕЧАНИЯ (NOTES) НА ДИАГРАММАХ ВИ

- предназначены для включения в модель произвольной текстовой информации, имеющей непосредственное отношение к контексту разрабатываемого проекта. В качестве такой информации могут быть комментарии разработчика (например, дата и версия разработки диаграммы или ее отдельных компонентов), ограничения (например, на значения отдельных связей или экземпляры сущностей) и помеченные значения. Применительно к диаграммам вариантов использования примечание может носить самую общую информацию, относящуюся к общему контексту системы.
- Графически примечания обозначаются прямоугольником с "загнутым" верхним правым углом . Внутри прямоугольника содержится текст примечания. Примечание может относиться к любому элементу диаграммы, в этом случае их соединяет пунктирная линия. Если примечание относится к нескольким элементам, то от него проводятся, соответственно, несколько линий.



СПАСИБО ЗА ВНИМАНИЕ!

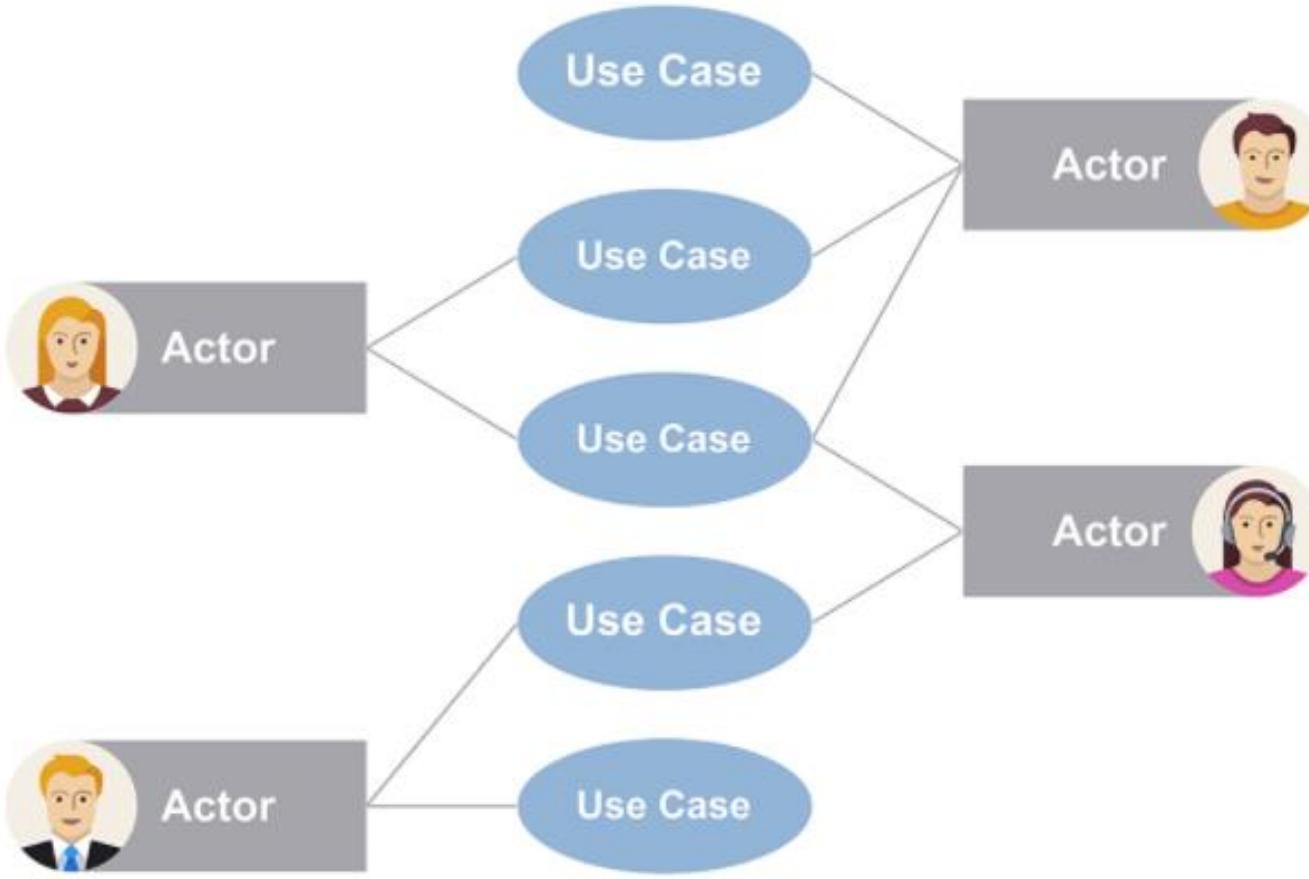
ПРОЕКТИРОВАНИЕ ПРОГРАММНЫХ СИСТЕМ

УНИФИЦИРОВАННЫЙ ЯЗЫК МОДЕЛИРОВАНИЯ UML

ЛЕКЦИЯ 3

ИНСТРУМЕНТЫ ДЛЯ РАЗРАБОТКИ НА UML

- Онлайн инструмент
- Отдельные программы
- Плагины
- Специальные IDE



В use-case диаграмме показывается, что делает система, теперь нужно определить, как это делается. Это обычно называется **реализацией вариантов использования**.

РЕАЛИЗАЦИЯ ВАРИАНТОВ ИСПОЛЬЗОВАНИЯ

Вариант использования – это описание множества последовательностей событий или действий (сценариев), доставляющих значимый для действующего лица результат.

- **Текстовые описания**
- **Реализация программой на псевдокоде**
- **Реализация диаграммами деятельности**
- **Реализация диаграммами взаимодействия**

ТЕКСТОВЫЕ ОПИСАНИЯ

Исторически самый заслуженный и до сих пор один из самых популярных способов: составить текстовое описание типичного сценария варианта использования. Рассмотрим следующий ниже текст в качестве примера.

Увольнение по собственному желанию

1. Сотрудник пишет заявление
2. Начальник подписывает заявление
3. Если есть неиспользованный отпуск, то бухгалтерия рассчитывает компенсацию
4. Бухгалтерия рассчитывает выходное пособие
5. Системный администратор удаляет учетную запись
6. Менеджер штатного расписания обновляет базу данных

РЕАЛИЗАЦИЯ ПРОГРАММОЙ НА ПСЕВДОКОДЕ

**Этот способ хорош тем, что понятен, привычен
и доступен любому.**

Use case Pay Compensation

if (add_payment)

 if (from Self Fire)

 начислить за неиспользованный

отпуск

 else

 if (from Adm Fire)

 начислить выходное пособие

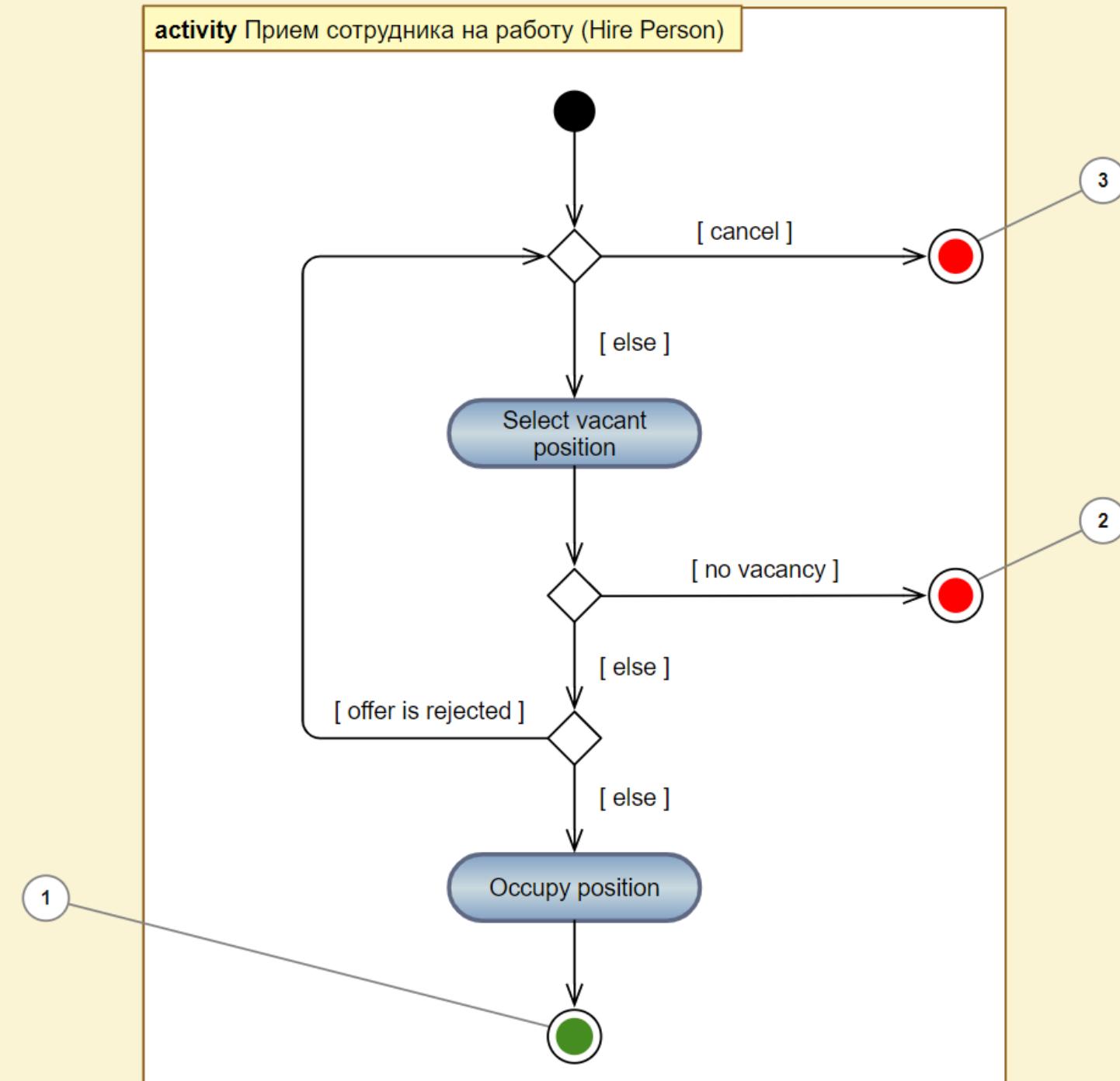
РЕАЛИЗАЦИЯ ПРОГРАММОЙ НА ПСЕВДОКОДЕ

Реализация на псевдокоде плохо согласуется с современной парадигмой объектно-ориентированного программирования.

**При использовании псевдокода теряются все преимущества использования UML:
наглядная визуализация с помощью картинки,
строгость и точность языка проектирования
и реализации, поддержка распространенными
инструментальными средствами.**

Решения на псевдокоде практически невозможно использовать повторно

РЕАЛИЗАЦИЯ ДИАГРАММАМИ ДЕЯТЕЛЬНОСТИ



РЕАЛИЗАЦИЯ ДИАГРАММАМИ ДЕЯТЕЛЬНОСТИ

Применение диаграмм деятельности для реализации вариантов использования не слишком приближает к появлению целевого артефакта – программного кода, однако может привести к более глубокому пониманию существа задачи и даже открыть неожиданные возможности улучшения приложения, которые было трудно усмотреть в первоначальной постановке задачи.

КЛАССИФИКАЦИЯ ДИАГРАММ

Диаграммы поведения:

- Диаграммы вариантов использования
- **Диаграммы деятельности**
- Диаграммы состояний
- **Диаграммы взаимодействия:**

- Диаграммы последовательности
- Диаграммы коммуникации
- Диаграммы синхронизации
- Обзорные диаграммы взаимодействия

Структурные диаграммы:

1. Диаграммы классов
2. Диаграммы объектов
3. Диаграммы пакетов
4. Диаграммы компонентов
5. Диаграммы составной структуры
6. Диаграммы размещения
7. Диаграммы профиля

ДИАГРАММЫ ДЕЯТЕЛЬНОСТИ (DIAGRAM ACTIVITY)

Диаграммы деятельности – это технология, позволяющая описывать логику процедур, бизнес процессы и потоки работ.

ОСНОВНЫЕ ПОНЯТИЯ ДД

Узлы управления:

- **входной узел**
- **финальный узел**
- **узел разделения**
- **узел слияния**
- **узел разветвления**
- **узел объединения**
- **узел действия**
- **Поток управления**
- **поток объектов**

Узлы управления:

Входной узел

Финальный узел

Узел разветвления

Узлы действий

Поток управления

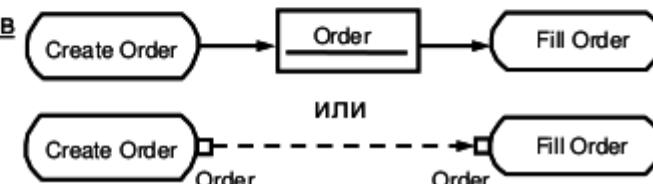
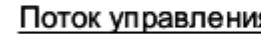
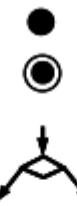
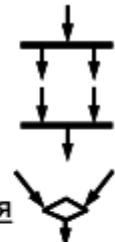
Поток объектов

Order o;
o = new Order;
FillOrder(o);

Узел разделения

Узел слияния

Узел объединения



или



ДИАГРАММЫ ДЕЯТЕЛЬНОСТИ (DIAGRAM ACTIVITY)

Диаграмма деятельности позволяет определить поведение с помощью последовательного исполнения поведений более низкого уровня. Исполнение следующего действия может начинаться в результате наступления одного из следующих событий:

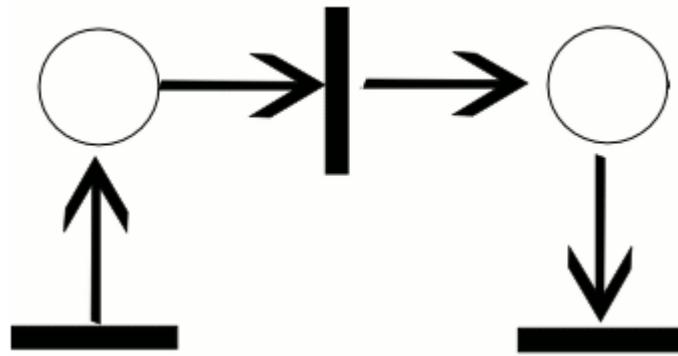
1. Завершение исполнения предыдущего действия
2. Появление необходимых данных
3. Наступление определенного события

ДИАГРАММА ДЕЯТЕЛЬНОСТИ

– это, фактически, старая добрая блок-схема алгоритма, в которой модернизированы обозначения, а семантика согласована с современным объектно-ориентированным подходом, что позволило органично включить диаграммы деятельности в UML.

На диаграмме деятельности применяют один основной тип сущностей – **деятельность**, и один тип отношений – **переходы** (передачи управления), а также графические обозначения (развилки, слияния и ветвления).

СЕТИ ПЕТРИ



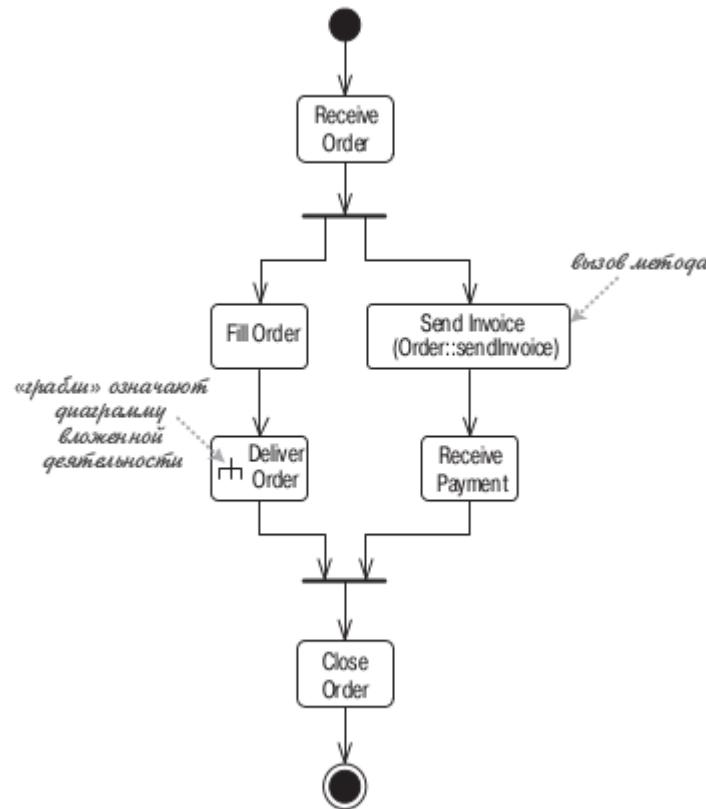
Сети Петри – математический аппарат для моделирования динамических дискретных систем. Впервые описаны Карлом Петри в 1962 году.

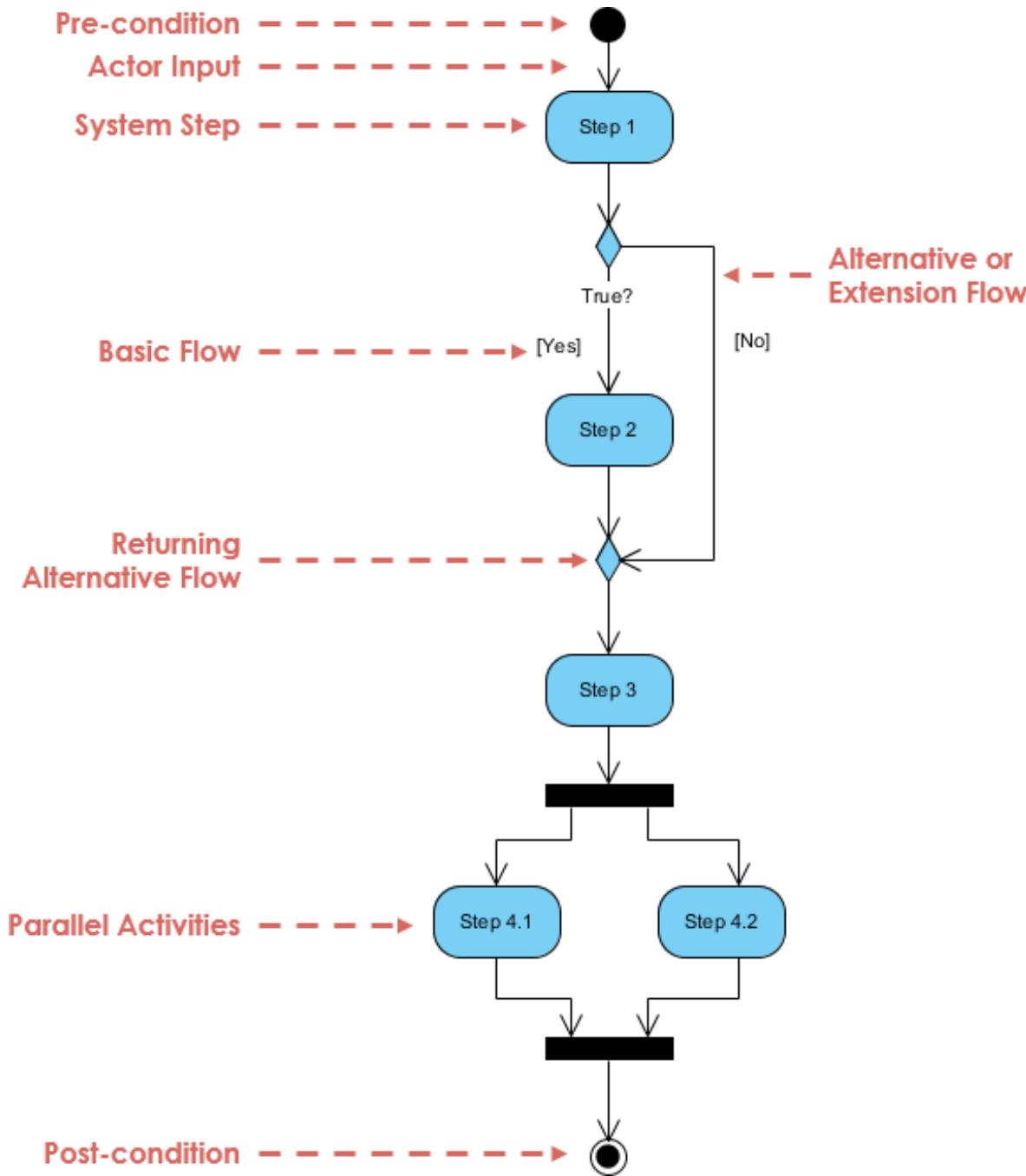
Сеть Петри представляет собой двудольный ориентированный граф, состоящий из вершин двух типов – позиций и переходов, соединённых между собой дугами. Вершины одного типа не могут быть соединены непосредственно. В позициях могут размещаться метки (маркеры), способные перемещаться по сети.

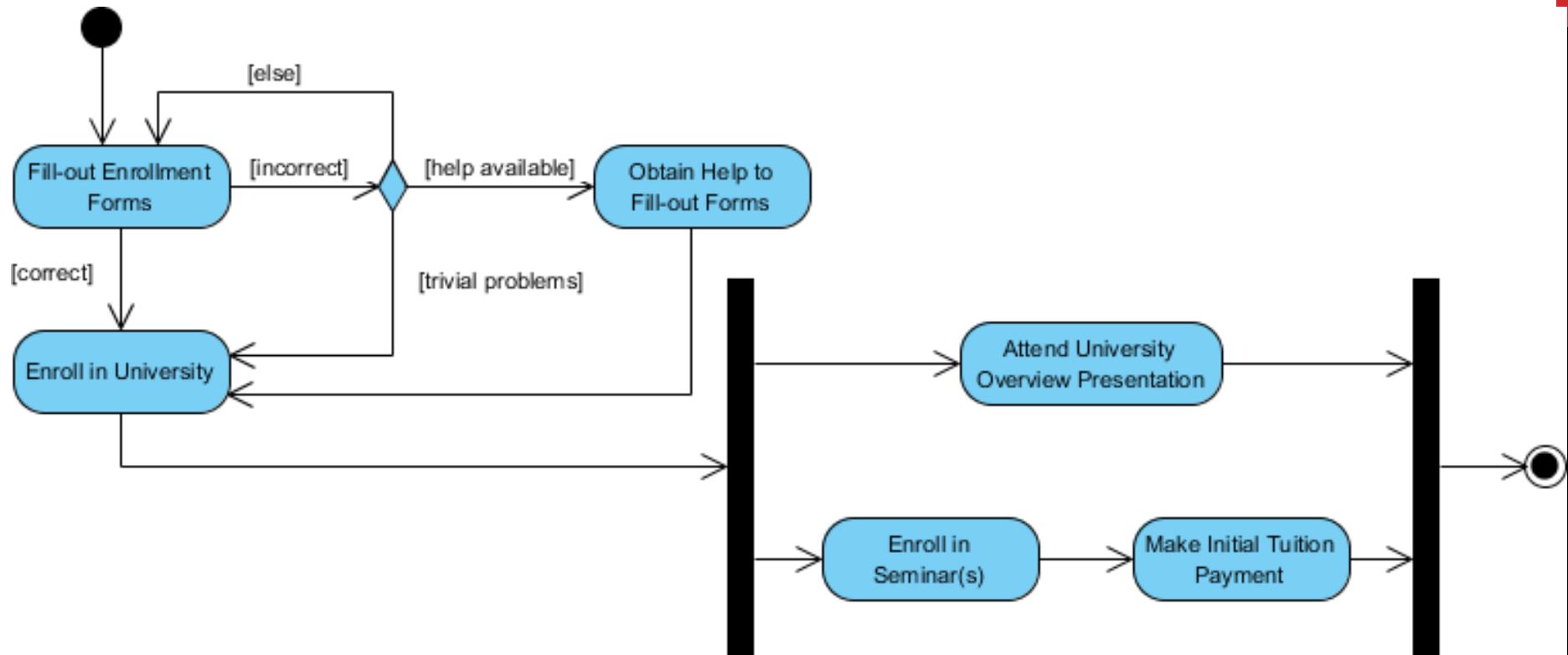
ДИАГРАММА ДЕЯТЕЛЬНОСТИ

- Описание одного процесса (алгоритма, активности, бизнес-процесса)
- Похожа на классическую блок-схему, но с некоторыми отличиями, дополнениями
- Показывает процесс в динамике, ход (текущие) работы в зависимости от условий
- Легче разрабатывать, если уже создана Use Case диаграмма
- Изображается в терминах действий с условиями (также имеется возможность указания объектов)

**Деятельности могут быть разбиты на вложенные
деятельности (subactivities) и определены как
самостоятельная деятельность:**

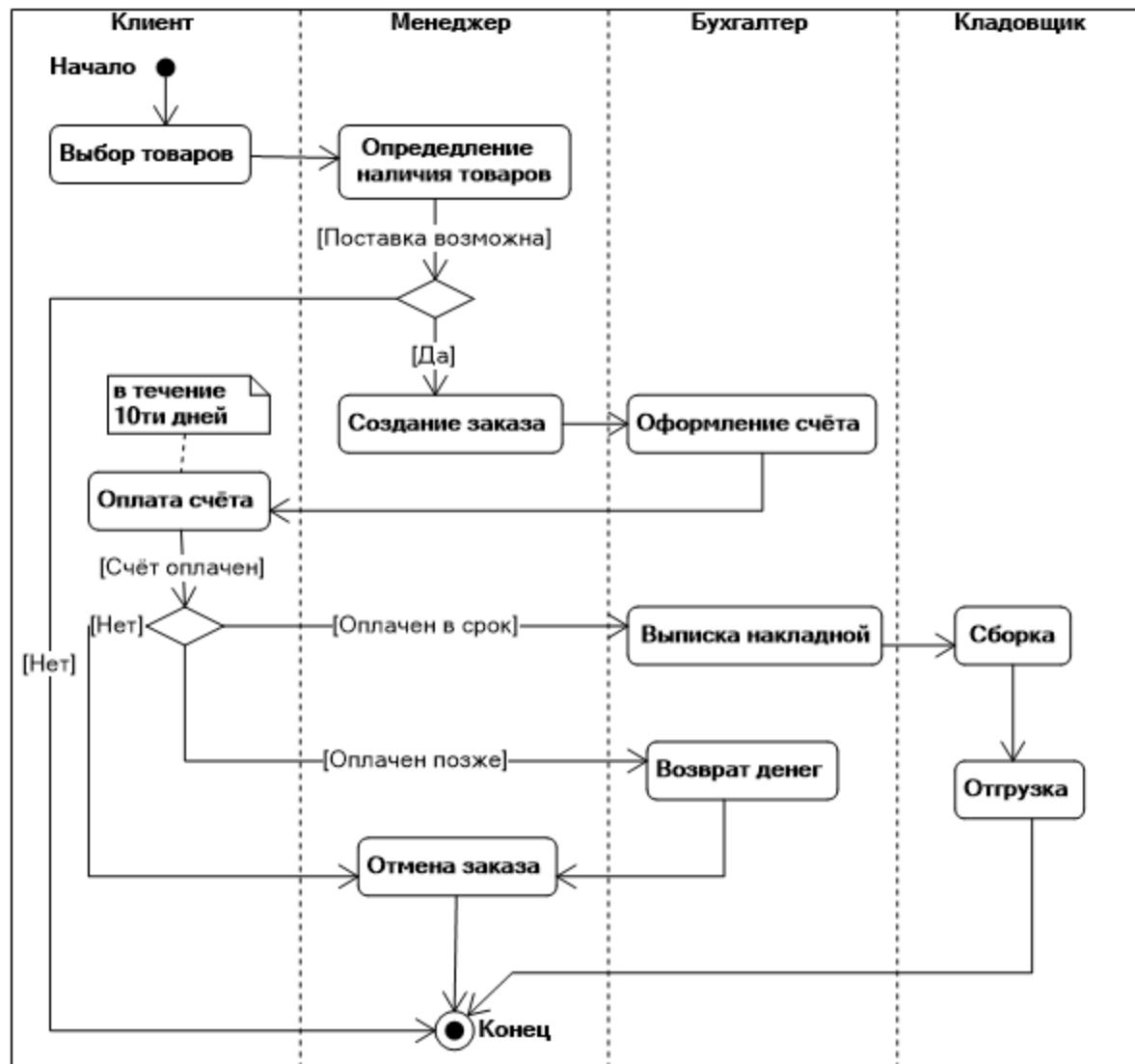






Кроме того, на диаграмме деятельности можно применить специальный графический комментарий – так называемые дорожки – подчеркивающие, что некоторые деятельности отличаются друг от друга, например, выполняются в разных местах.

Графически дорожки изображаются в виде прямоугольников с названиями.



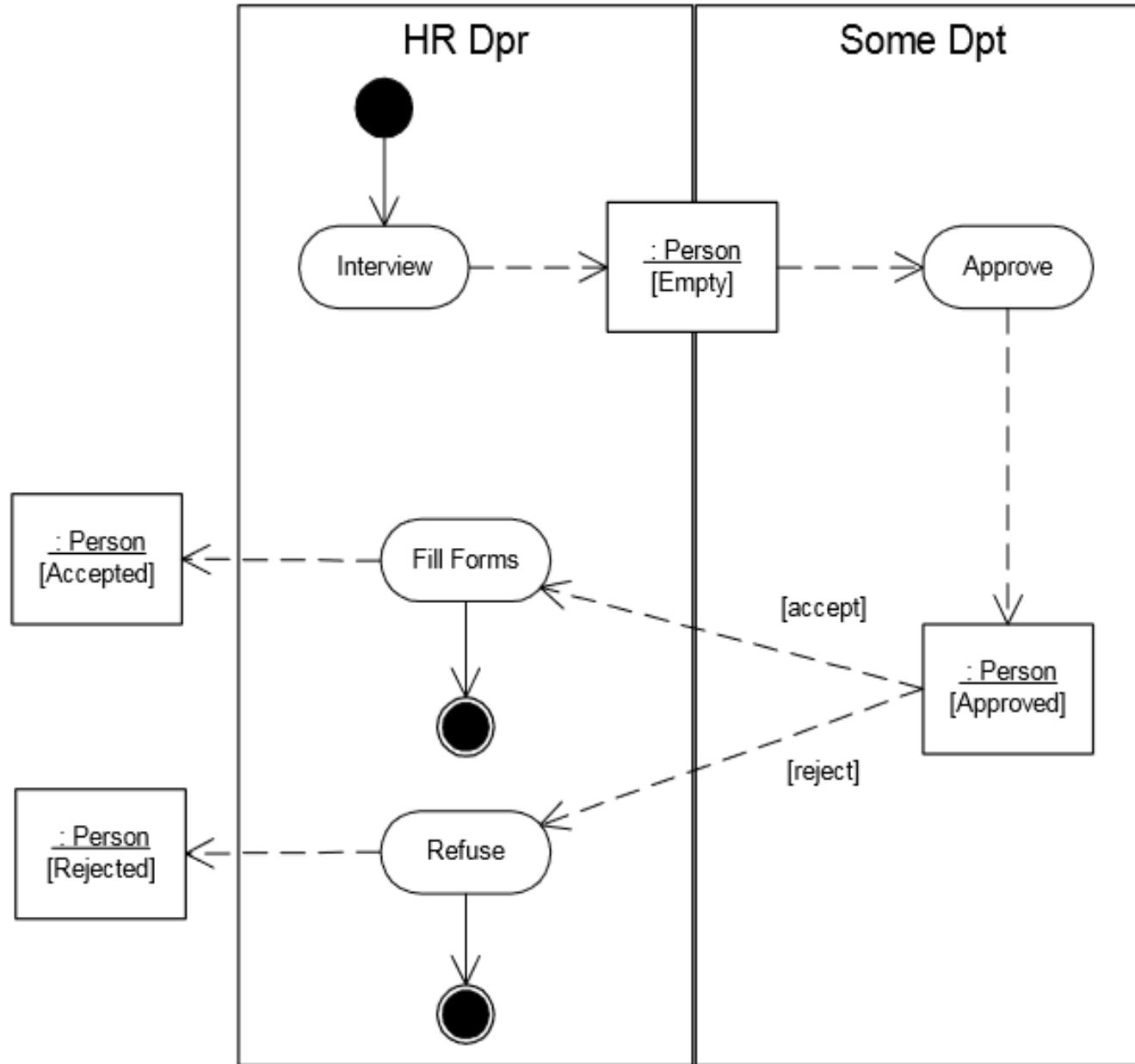
Помимо потока управления на диаграмме деятельности можно показать и поток данных, используя такую сущность, как объект (в определенном состоянии) и соответствующую зависимость.

Объект в состоянии – это объект некоторого класса, про который известно, что он находится в определенном состоянии в данной точке вычислительного процесса.

Синтаксически объект в состоянии изображается, как обычно, в виде прямоугольника и его имя подчеркивается, но дополнительно после имени объекта в квадратных скобках пишется имя состояния, в котором в данной точке вычислительного процесса находится объект. В некоторых случаях состояние объекта не важно, например, если достаточно указать, что в данной точке вычислительного процесса создается новый объект данного класса, и в этом случае применяется обычная нотация для изображения объектов. Важно подчеркнуть, что объект в состоянии на диаграммах деятельности "по определению" считается состоянием, т. е. вершиной графа модели, которая может быть инцидентна переходам, правда переходам особого рода.

Траектория объекта – это переход особого рода, исходным и/или целевым состоянием которого является объект в состоянии.

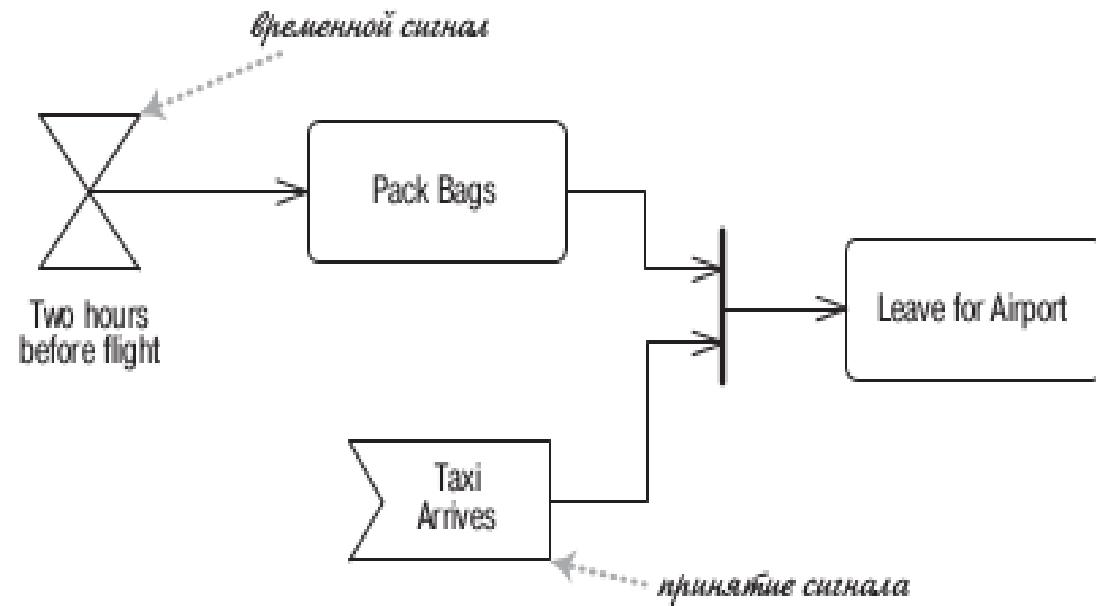
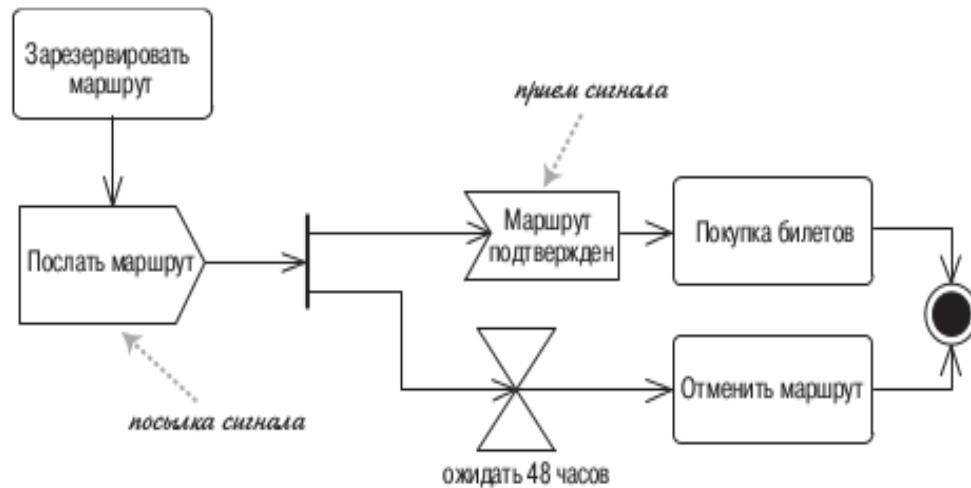
Траектория объекта изображается в виде пунктирной стрелки (в отличии от сплошной стрелки обычного перехода).



СИГНАЛЫ

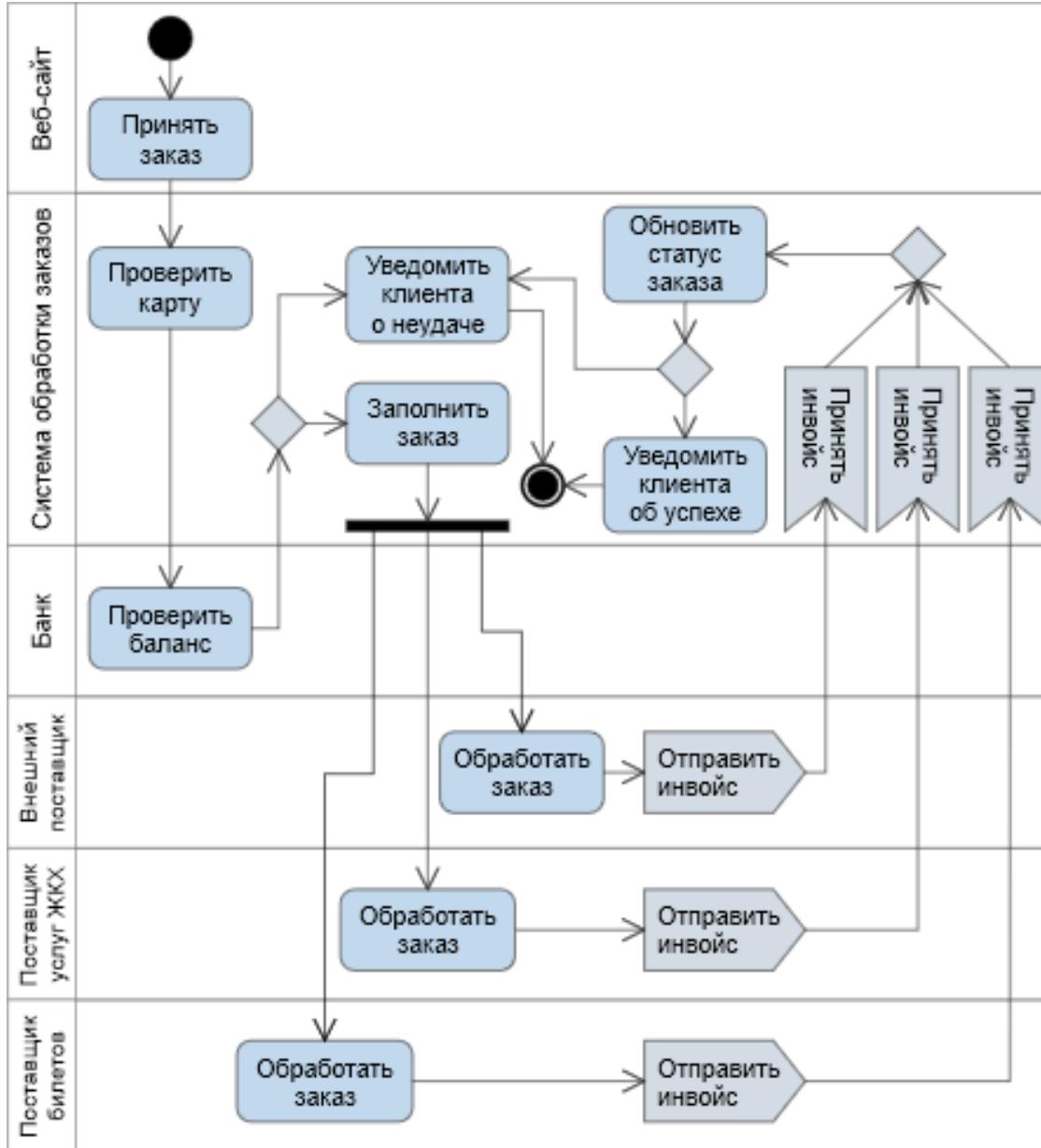
Диаграммы деятельности имеют четко определенную стартовую точку, соответствующую вызову программы или процедуры. Кроме того, операции могут отвечать на сигналы.

Временной сигнал (time signal) приходит по прошествии времени.

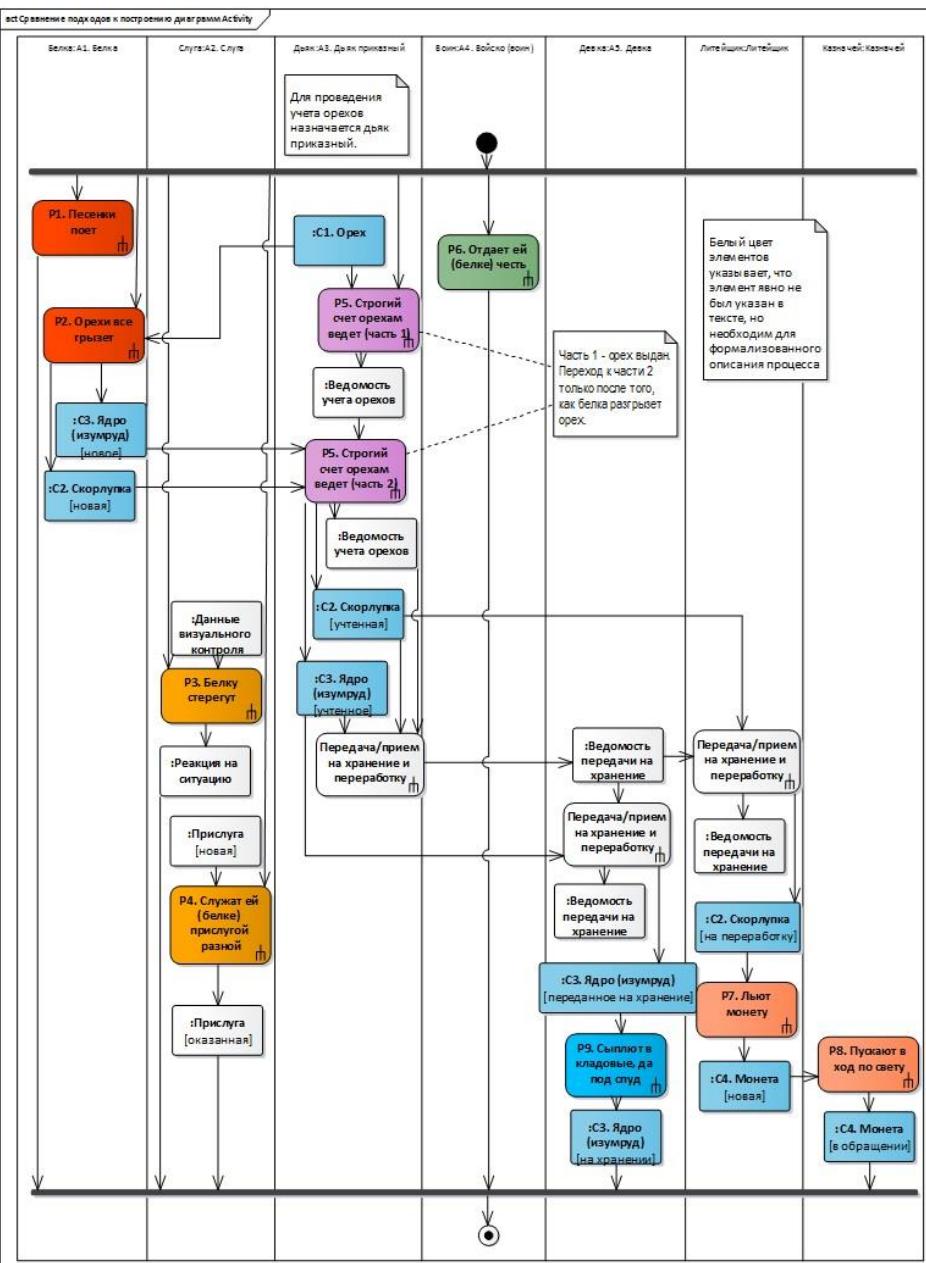




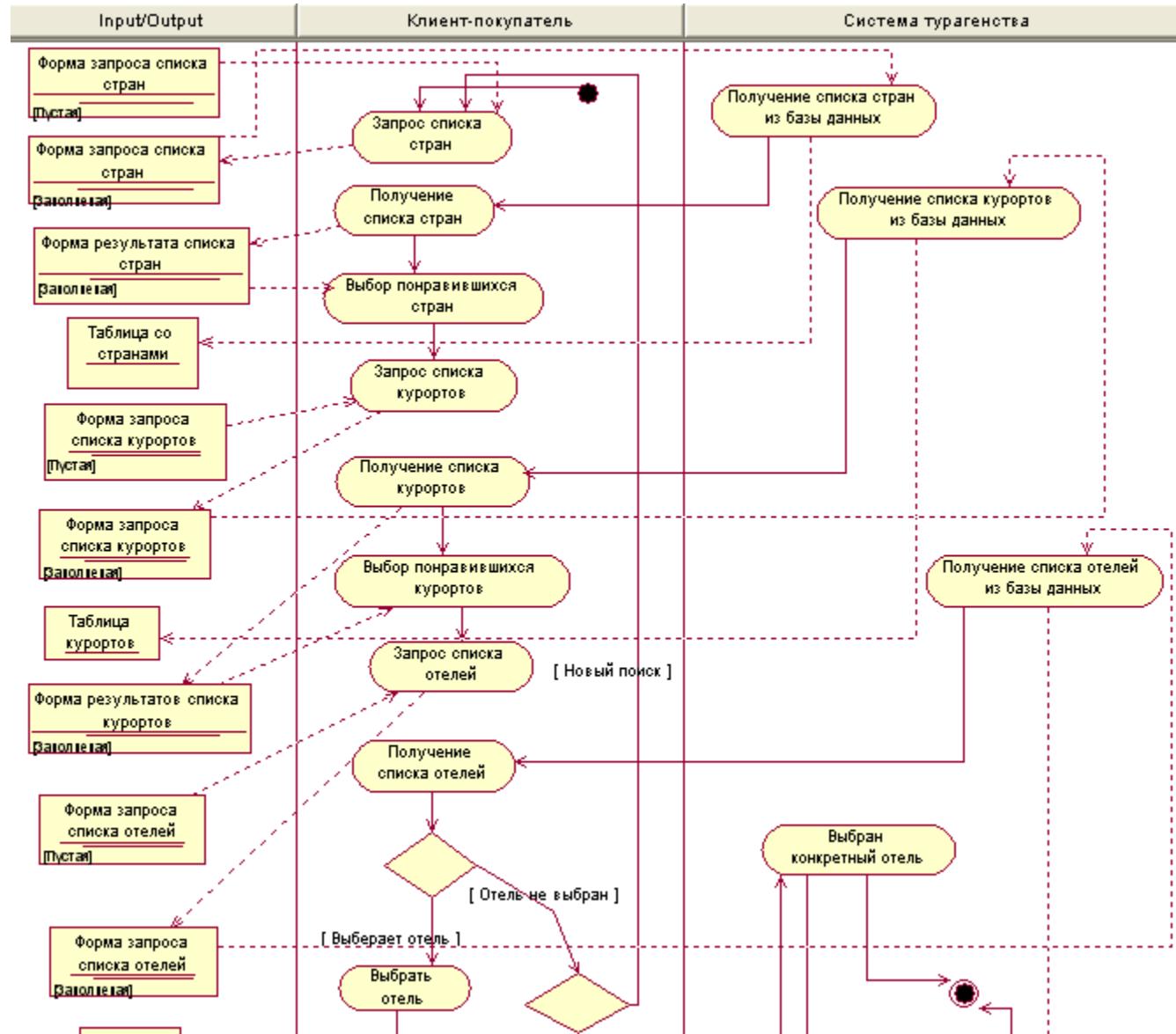
Диаграммы деятельности применяются для описания поведения на самом высоком уровне абстракции, наиболее удаленном от программной реализации и на самом низком уровне, практически на уровне программного кода.



Пример диаграммы деятельности

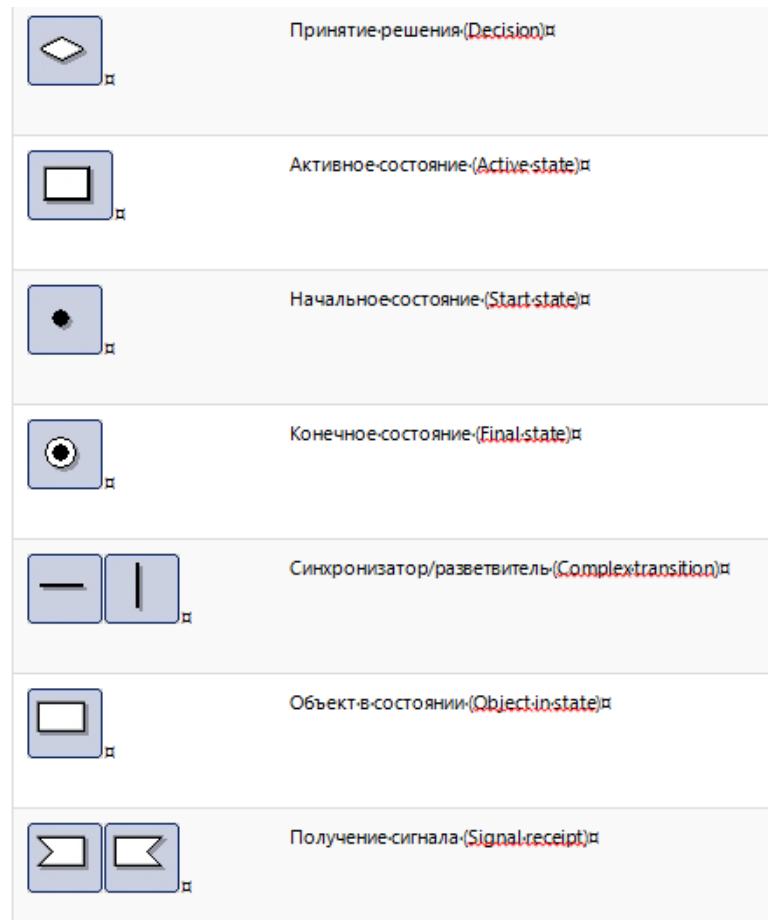


Пример диаграммы деятельности



Пример диаграммы деятельности

ОСНОВНЫЕ ЭЛЕМЕНТЫ ДИАГРАММЫ ДЕЯТЕЛЬНОСТИ



ОСНОВНЫЕ ЭЛЕМЕНТЫ ДИАГРАММЫ ДЕЯТЕЛЬНОСТИ

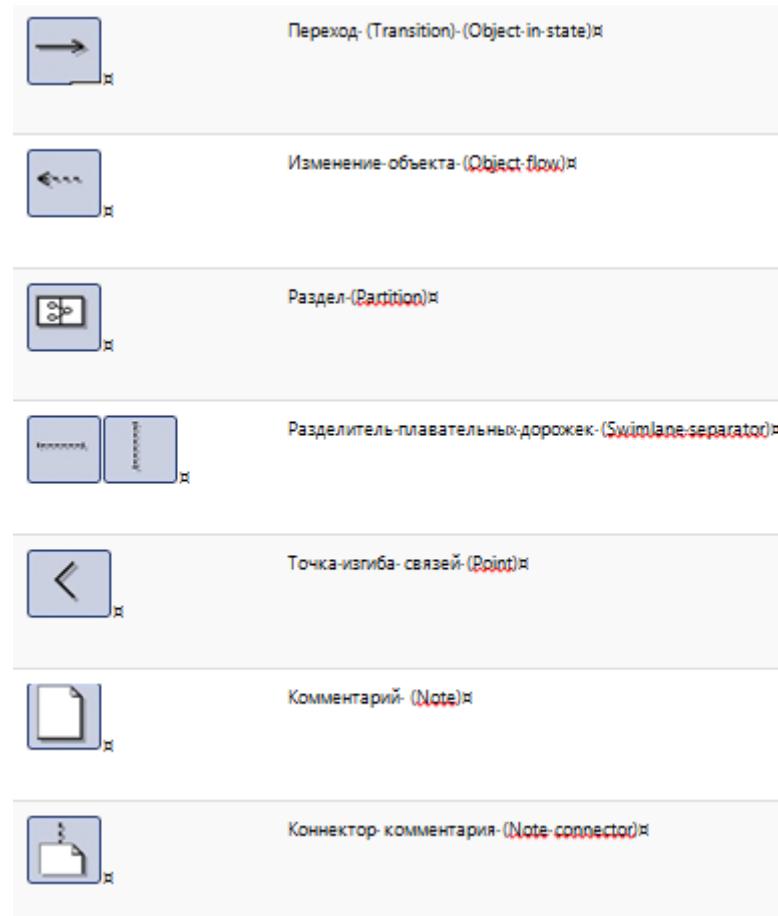


ДИАГРАММА СОСТОЯНИЙ (STATE MACHINE DIAGRAMS)

ДИАГРАММЫ СОСТОЯНИЙ

В основе конечных автоматов UML лежит работа Харела . С их помощью обычно моделируется предыстория жизненного цикла одного реактивного объекта. Она представляется в виде конечного автомата – автомата, который может существовать в конечном числе состояний. В ответ на события конечный автомат четко осуществляет переходы между этими состояниями.

Конечный автомат моделирует динамическое поведение реактивного объекта.

ДИАГРАММЫ СОСТОЯНИЙ

Для описания жизненного цикла конкретного объекта, поведение которого зависит от истории этого объекта, или же требует обработки асинхронных стимулов, используется конечный автомат в форме диаграммы состояний. При этом состояния конечного автомата соответствуют состояниям объекта, т.е. различным наборам значений атрибутов, а переходы соответствуют выполнению операций.

Выполнение конструктора объекта моделируется переходом из начального состояния, а выполнение деструктора — переходом в заключительное состояние. Диаграммы состояний можно составить не только для программных объектов — экземпляров отдельных классов, но и для более крупных конструкций, в частности, для всей модели приложения в целом или для более мелких — отдельных операций.

ДИАГРАММЫ СОСТОЯНИЙ

Конечные автоматы в UML реализованы довольно своеобразно. С одной стороны, в основу положено классическое представление автомата в форме графа состояний-переходов. С другой стороны, к классической форме добавлено большое число различных расширений и вспомогательных обозначений, которые, строго говоря, не обязательны – без них в принципе можно было бы обойтись – но весьма удобны и наглядны при составлении диаграмм.

На диаграммах состояний применяется всего один тип сущностей — состояния, и всего один тип отношений — переходы.

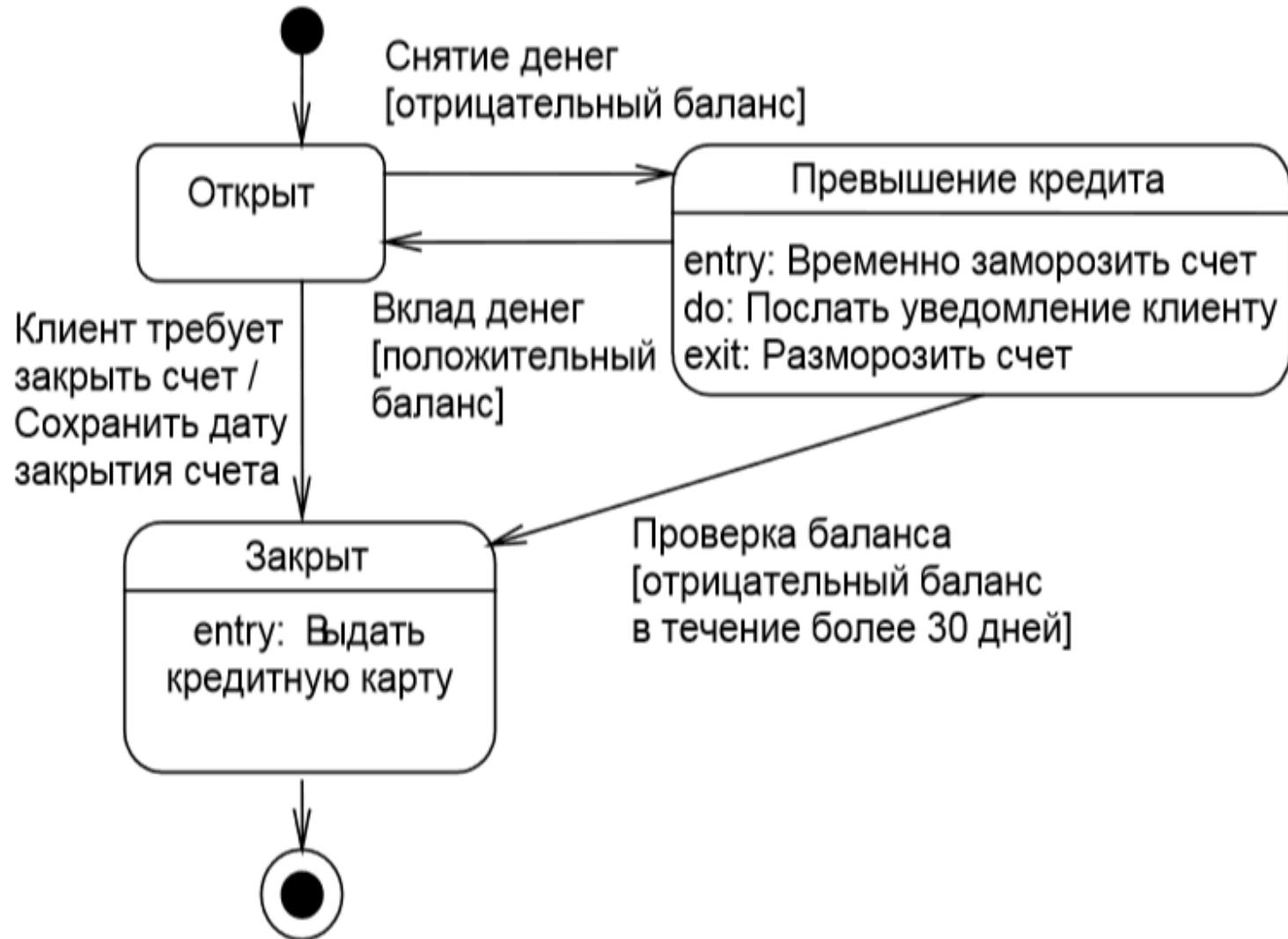
ДИАГРАММЫ СОСТОЯНИЙ

Три основных элемента автоматов – состояния, события и переходы:

- состояние (state) – «условие или ситуация в жизни объекта, при которых он удовлетворяет некоторому условию, осуществляет некоторую деятельность или ожидает некоторого события»
- событие (event) – «описание заслуживающего внимания происшествия, занимающего определенное положение во времени и пространстве»
- переход (transition) – переход из одного состояния в другое в ответ на событие.

СОСТОЯНИЯ

- Начальное состояние
- Финальное состояние
- Деятельность состояния
- Входное состояние
- Выходное состояние
- Внутренние переходы



Процессы, происходящие, когда объект находится в определенном состоянии, называются действиями (actions).

С состоянием можно связывать данные пяти типов:

- **деятельность,**
- **входное действие,**
- **выходное действие,**
- **событие и**
- **история состояния.**



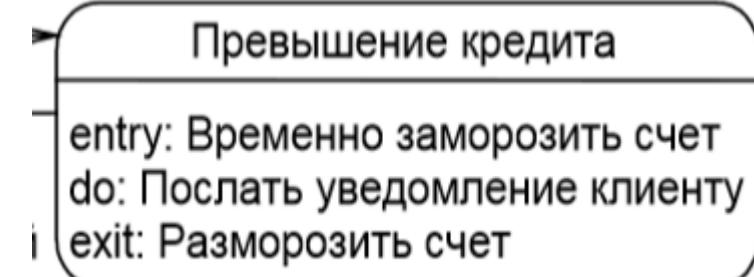
синтаксис действия: `имяСобытия/некотороеДействие`

синтаксис деятельности: `до/некотораяДеятельность`

Синтаксис состояния

ДЕЯТЕЛЬНОСТЬ

Деятельностью (activity) называется поведение, реализуемое объектом, пока он находится в данном состоянии. Деятельность – это прерываемое поведение. Оно может выполняться до своего завершения, пока объект находится в данном состоянии, или может быть прервано переходом объекта в другое состояние. Деятельность изображают внутри самого состояния, ей должно предшествовать слово do (делать) и двоеточие.



ВХОДНОЕ ДЕЙСТВИЕ

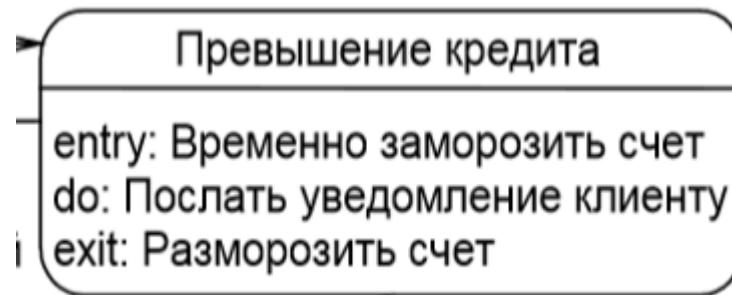
Входным действием (entry action) называется поведение, которое выполняется, когда объект переходит в данное состояние. В примере счета в банке, когда он переходит в состояние «Превышен счет», выполняется действие «Временно заморозить счет», независимо от того, откуда объект перешел в это состояние. Таким образом, данное действие осуществляется не после того, как объект перешел в это состояние, а, скорее, как часть этого перехода. В отличие от деятельности, входное действие рассматривается как непрерываемое. Входное действие также показывают внутри состояния. ему предшествует слово entry (вход) и двоеточие.

Превышение кредита

entry: Временно заморозить счет
do: Послать уведомление клиенту
exit: Разморозить счет

ВЫХОДНОЕ ДЕЙСТВИЕ

Выходное действие (exit action) подобно входному. Однако, оно осуществляется как составная часть процесса выхода из данного состояния. В нашем примере при выходе объекта Account из состояния «Превышен счет», независимо от того, куда он переходит, выполняется действие «Разморозить счет». Оно является частью процесса такого перехода. Как и входное, выходное действие является непрерываемым. Выходное действие изображают внутри состояния, ему предшествует слово exit (выход) и двоеточие.



DO: ^ЦЕЛЬ.СОБЫТИЕ(АРГУМЕНТЫ)

Поведение объекта во время деятельности, при входных и выходных действиях может включать отправку события другому объекту. Например, объект account (счет) может посыпать событие объекту card reader (устройство чтения карты). В этом случае описанию деятельности, входного действия или выходного действия предшествует знак « ^ ». Соответствующая строка на диаграмме выглядит как

Do: ^Цель.Событие(Аргументы)

Здесь Цель – это объект, получающий событие, Событие – это посыпаемое сообщение, а Аргументы являются параметрами посыпаемого сообщения.

ПЕРЕХОД

Переходом (Transition) называется перемещение из одного состояния в другое. Совокупность переходов диаграммы показывает, как объект может перемещаться между своими состояниями. На диаграмме все переходы изображают в виде стрелки, начинающейся на первоначальном состоянии и заканчивающейся последующим. Переходы могут быть рефлексивными. Объект может перейти в то же состояние, в котором он в настоящий момент находится. Рефлексивные переходы изображают в виде стрелки, начинающейся и завершающейся на одном и том же состоянии.

У перехода существует несколько спецификаций. Они включают события, аргументы, ограждающие условия, действия и посылаемые события.

СОБЫТИЯ

Событие (event) – это то, что вызывает переход из одного состояния в другое. В нашем примере событие «Клиент требует закрыть» вызывает переход счета из открытого в закрытое состояние.

Событие размещают на диаграмме вдоль линии перехода. На диаграмме для отображения события можно использовать как имя операции, так и обычную фразу. В нашем примере события описаны обычными фразами.

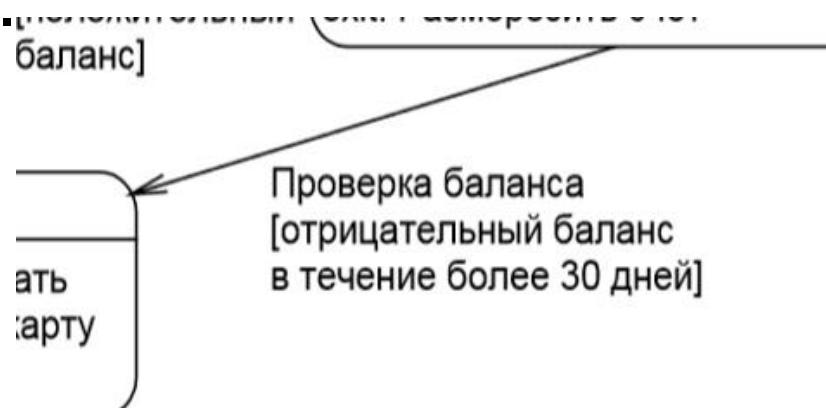
Большинство переходов должны иметь события, так как именно они, прежде всего, заставляют переход осуществиться. Тем не менее, бывают и автоматические переходы, не имеющие событий. При этом объект сам перемещается из одного состояния в другое со скоростью, позволяющей осуществляться входным действиям, деятельности и выходным действиям.



ОГРАЖДАЮЩИЕ УСЛОВИЯ

Ограждающие условия (guard conditions) определяют, когда переход может, а когда не может осуществиться. Ограждающие условия изображают на диаграмме вдоль линии перехода после имени события, заключая их в квадратные скобки.

Ограждающие условия задавать необязательно. Однако если существует несколько автоматических переходов из состояния, необходимо определить для них взаимно исключающие ограждающие условия. Это поможет читателю диаграммы понять, какой путь перехода будет автоматически выбран.

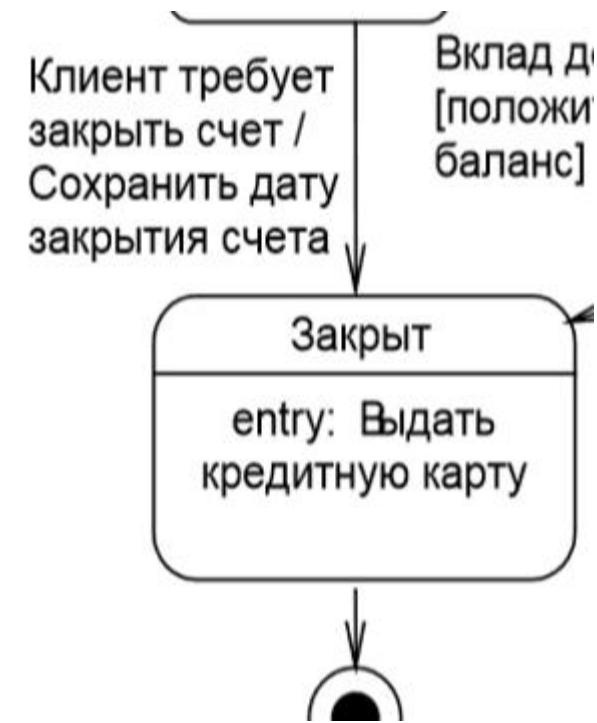


ДЕЙСТВИЕ

Действием (action) является непрерывное поведение, осуществляющееся как часть перехода.

Входные и выходные действия показывают внутри состояний, поскольку они определяют, что происходит, когда объект входит или выходит из него. Большую часть действий, однако, изображают вдоль линии перехода, так как они не должны осуществляться при входе или выходе из состояния.

Действие рисуют вдоль линии перехода после имени события, ему предшествует косая черта. Событие или действие могут быть поведением внутри объекта, а могут представлять собой сообщение, посыпаемое другому объекту. Если событие или действие посыпается другому объекту, перед ним на диаграмме помещают знак « ^ ».



СОСТОЯНИЯ И ПЕРЕХОДЫ

Состояния бывают:

- простые,
- составные,
- специальные
- и каждый тип состояний имеет дополнительные подтипы и различные составляющие элементы.

Переходы бывают *простые и составные*, и каждый переход содержит от двух до пяти составляющих:

- исходное состояние,
- событие перехода,
- сторожевое условие,
- действие на переходе,
- целевое состояние.

СОБЫТИЕ ПЕРЕХОДА

Событие перехода – это тот входной стимул, который вкупе с текущим состоянием автомата определяет следующее состояние.

В UML используются четыре типа событий:

- событие вызова,
- событие сигнала,
- событие таймера,
- событие изменения.

Составное состояние может быть

- последовательным или
- параллельным (ортогональным).

Специальные состояния, в свою очередь, бывают следующих типов:

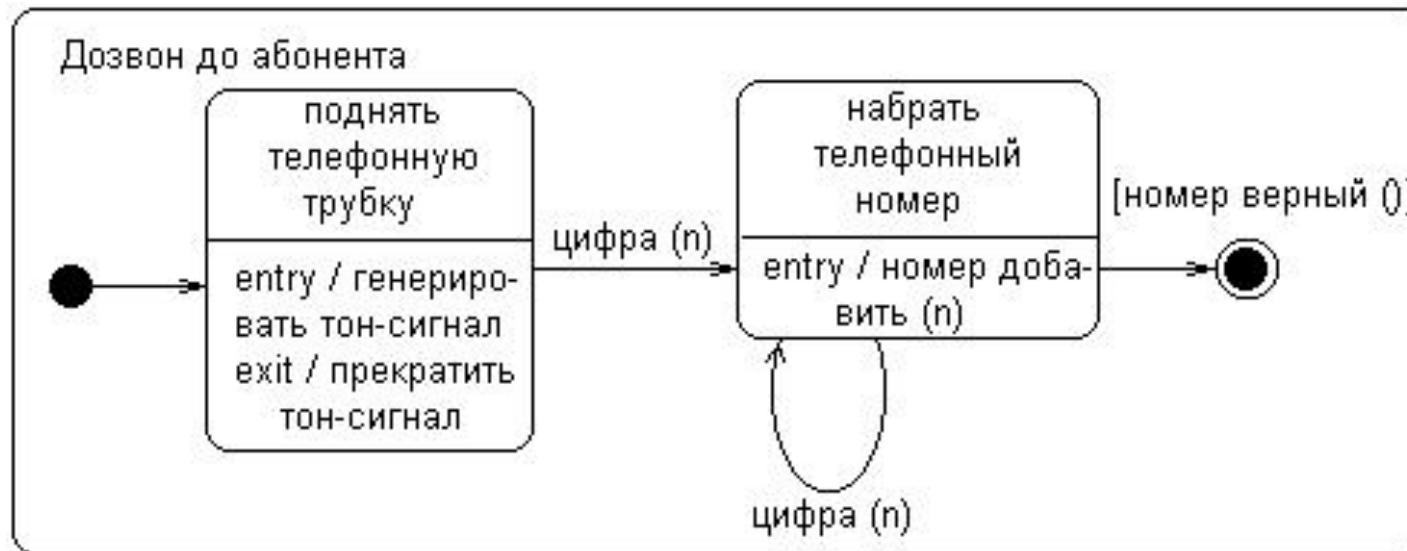
- начальное состояние,
- заключительное состояние,
- переходное состояние,
- историческое состояние,
- синхронизирующее состояние,
- ссылочное состояние,
- состояние "заглушка".

СОСТАВНОЕ СОСТОЯНИЕ

Составное состояние – это состояние, в которое вложена машина состояний. Если вложена только одна машина, то состояние называется последовательным, если несколько – параллельным. Глубина вложенности в UML неограничена, т.е. состояния вложенной машины состояний также могут быть составными.



СОСТАВНОЕ СОСТОЯНИЕ



Пример составного состояния с двумя вложенными последовательными подсостояниями.

ИСТОРИЧЕСКОЕ СОСТОЯНИЕ

Историческое состояние – это специальное состояние, подобное начальному состоянию, но обладающее дополнительной семантикой.

Историческое состояние может использоваться во вложенной машине состояний внутри составного состояния.

Историческое состояние имеет две разновидности.

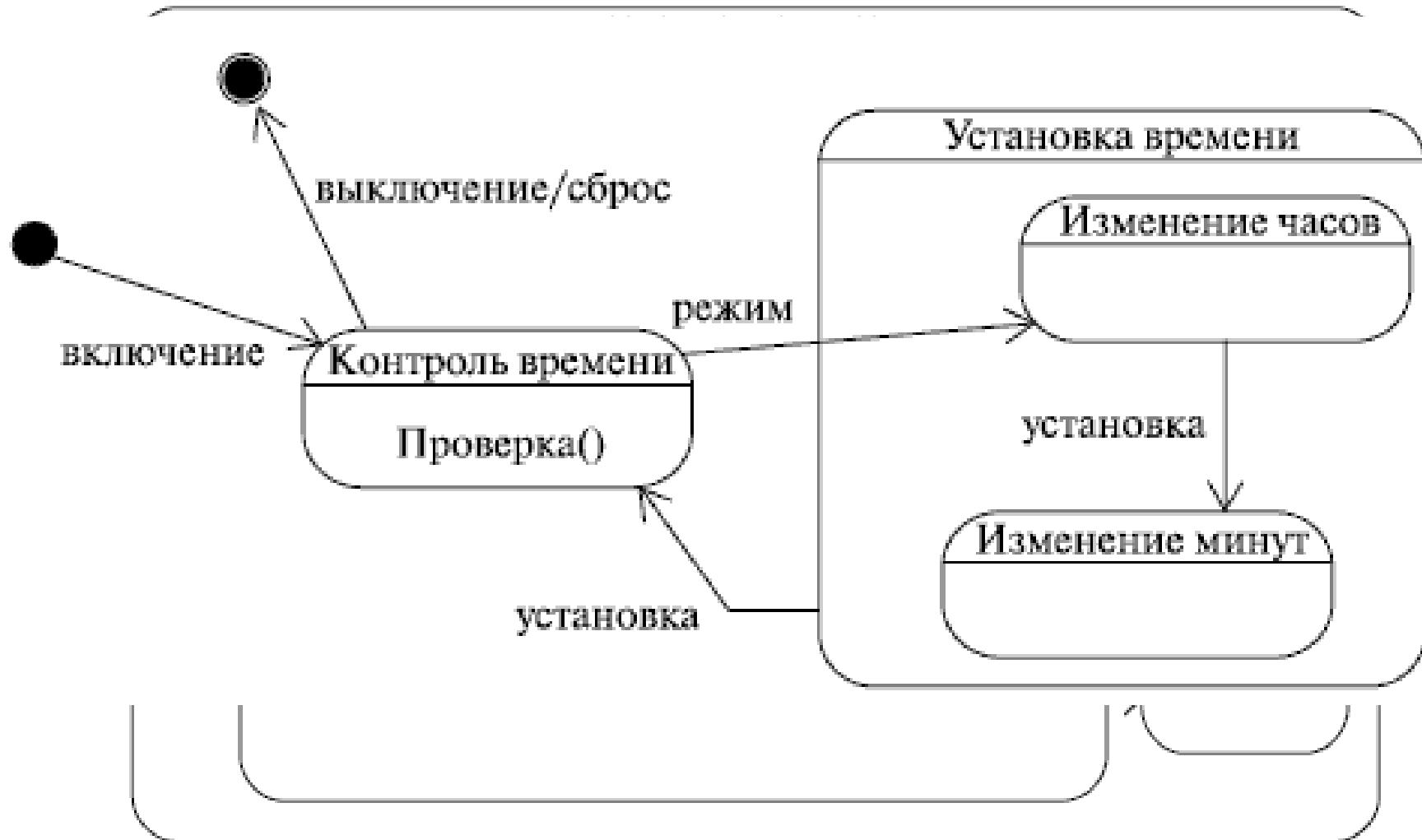
Поверхностное историческое состояние запоминает, какое состояние было активным на том же уровне вложенности, на каком находится само историческое состояние.

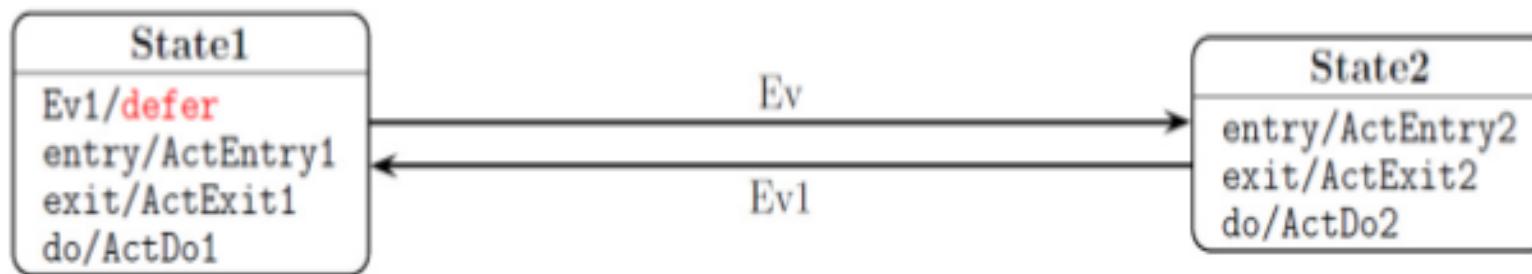
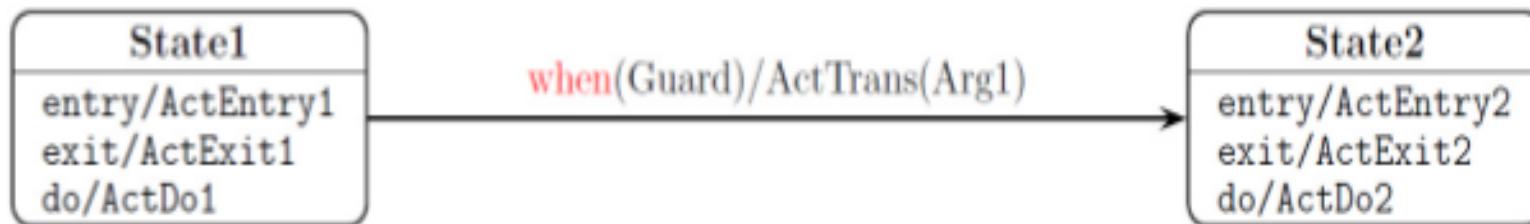
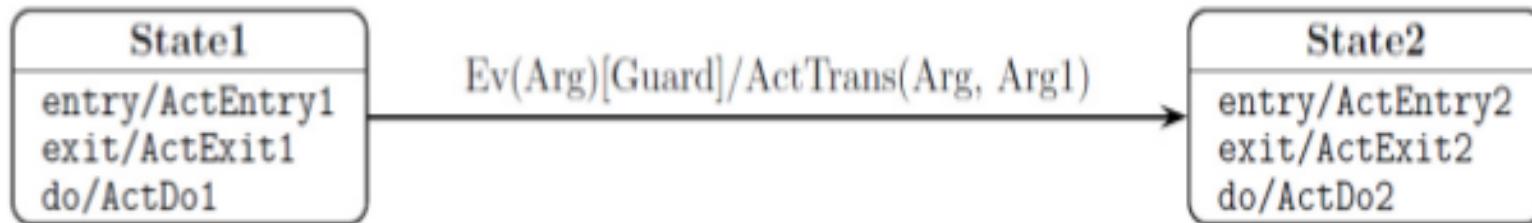
Глубинное историческое состояние помнит не только активное состояние на данном уровне, но и на вложенных уровнях.

ИСТОРИЧЕСКОЕ СОСТОЯНИЕ

При первом запуске машины состояний историческое состояние означает в точности тоже, что и начальное: оно указывает на состояние, в котором находится машина в начале работы. Если в данной машине состояний используется историческое состояние, то при выходе из объемлющего составного состояния запоминается то состояние, в котором находилась вложенная машина при выходе. При повторном входе в данное составное состояние в качестве текущего состояния восстанавливается то состояние, в котором машина находилась при выходе. Проще говоря, историческое состояние заставляет машину помнить, в каком состоянии ее прервали прошлый раз и "продолжать начатое".

ПРИМЕРЫ ДИАГРАММ СОСТОЯНИЙ



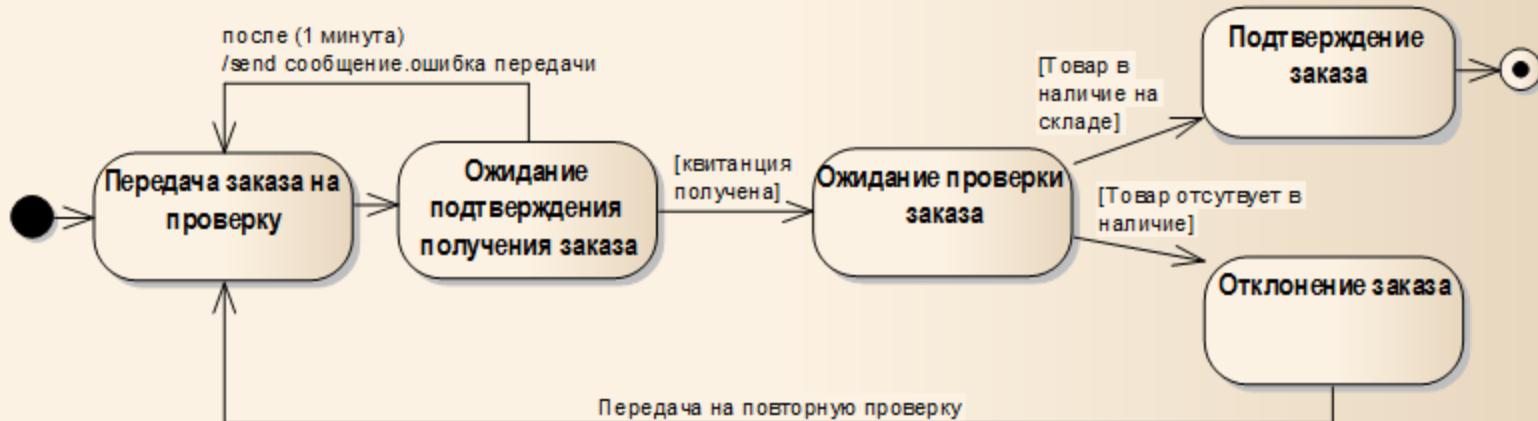


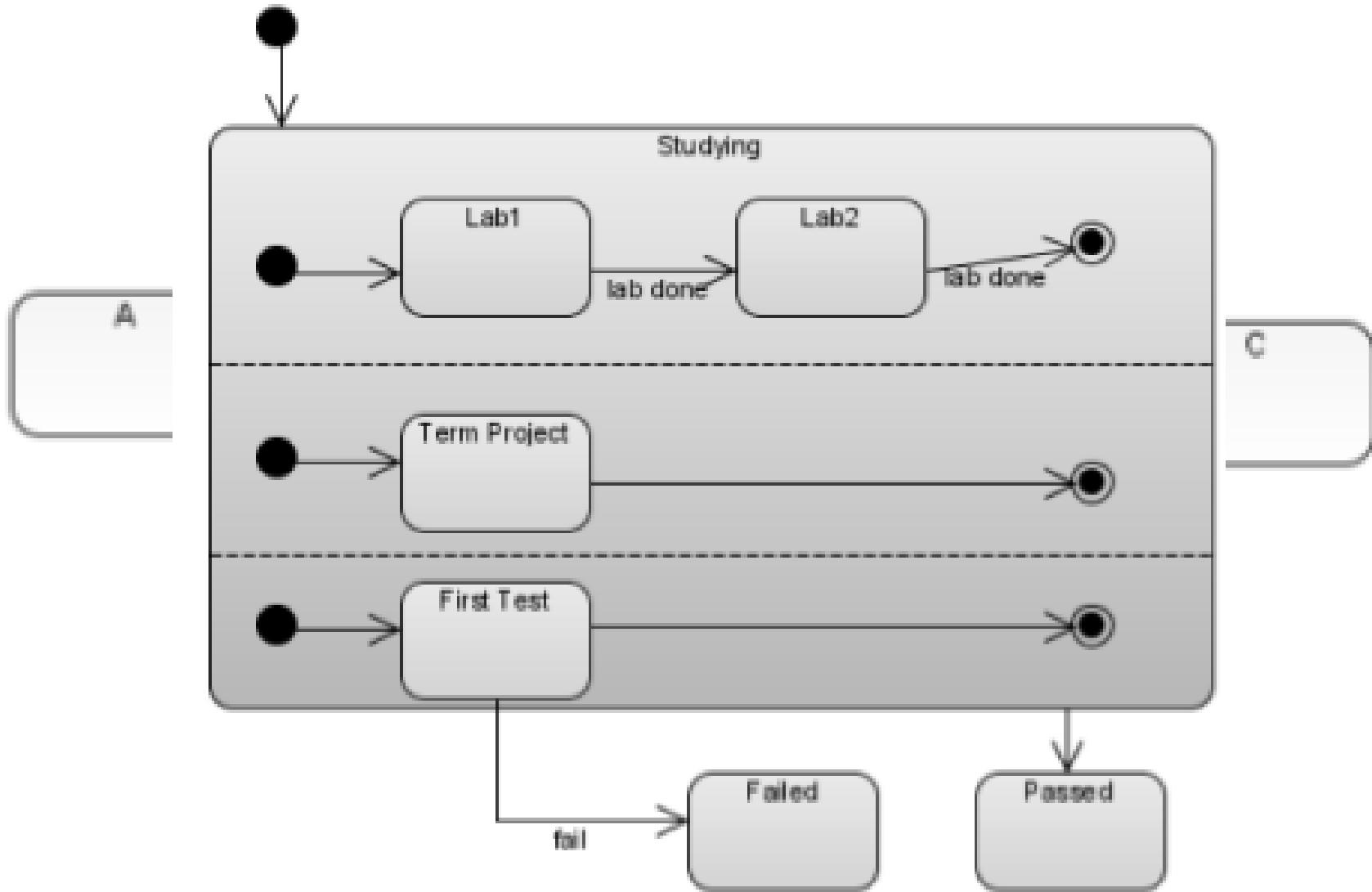
Примеры переходов

ПРИМЕРЫ ПЕРЕХОДОВ

Проверка заказа

- + entry / Создание нового заказа
- + exit / Изменение статуса заказа на "Подтвержден"

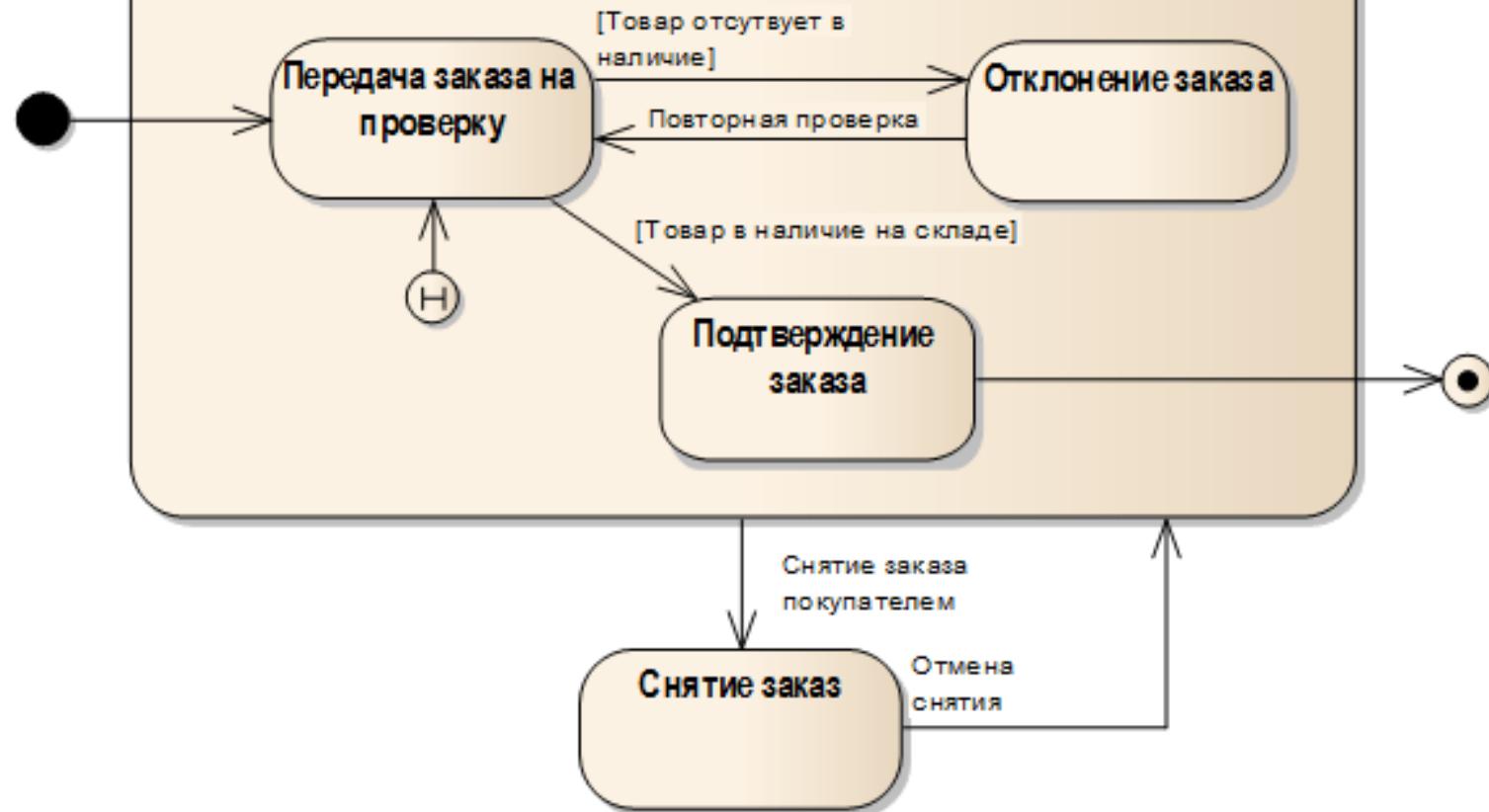




Композитные состояния и вложенные подсостояния

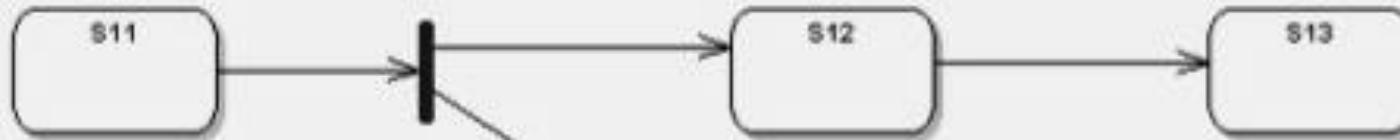
Проверка заказа

- + entry / Создание нового заказа
- + exit / Изменение статуса заказа на "Подтверждён"



Пример композитного состояния

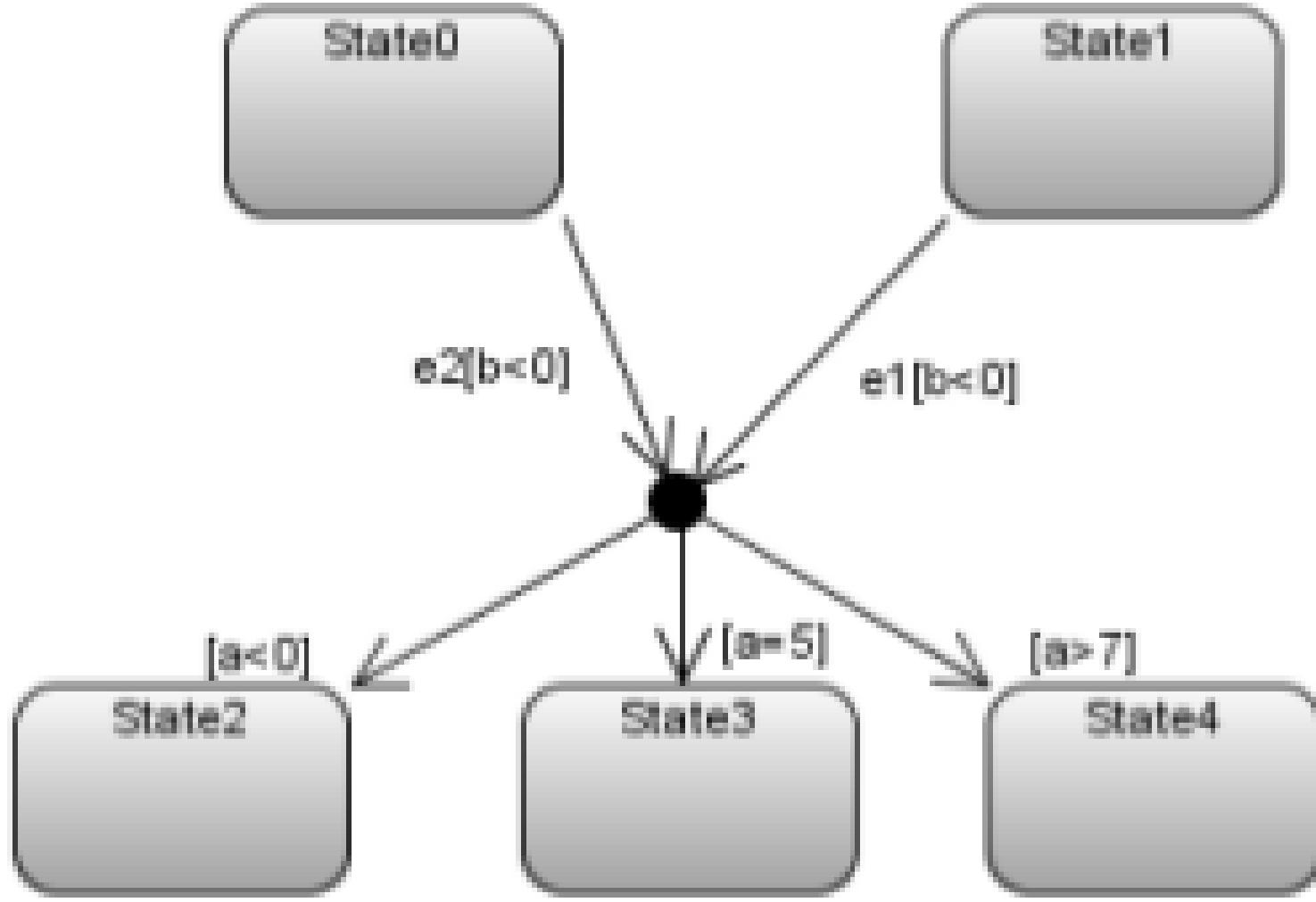
[S1]



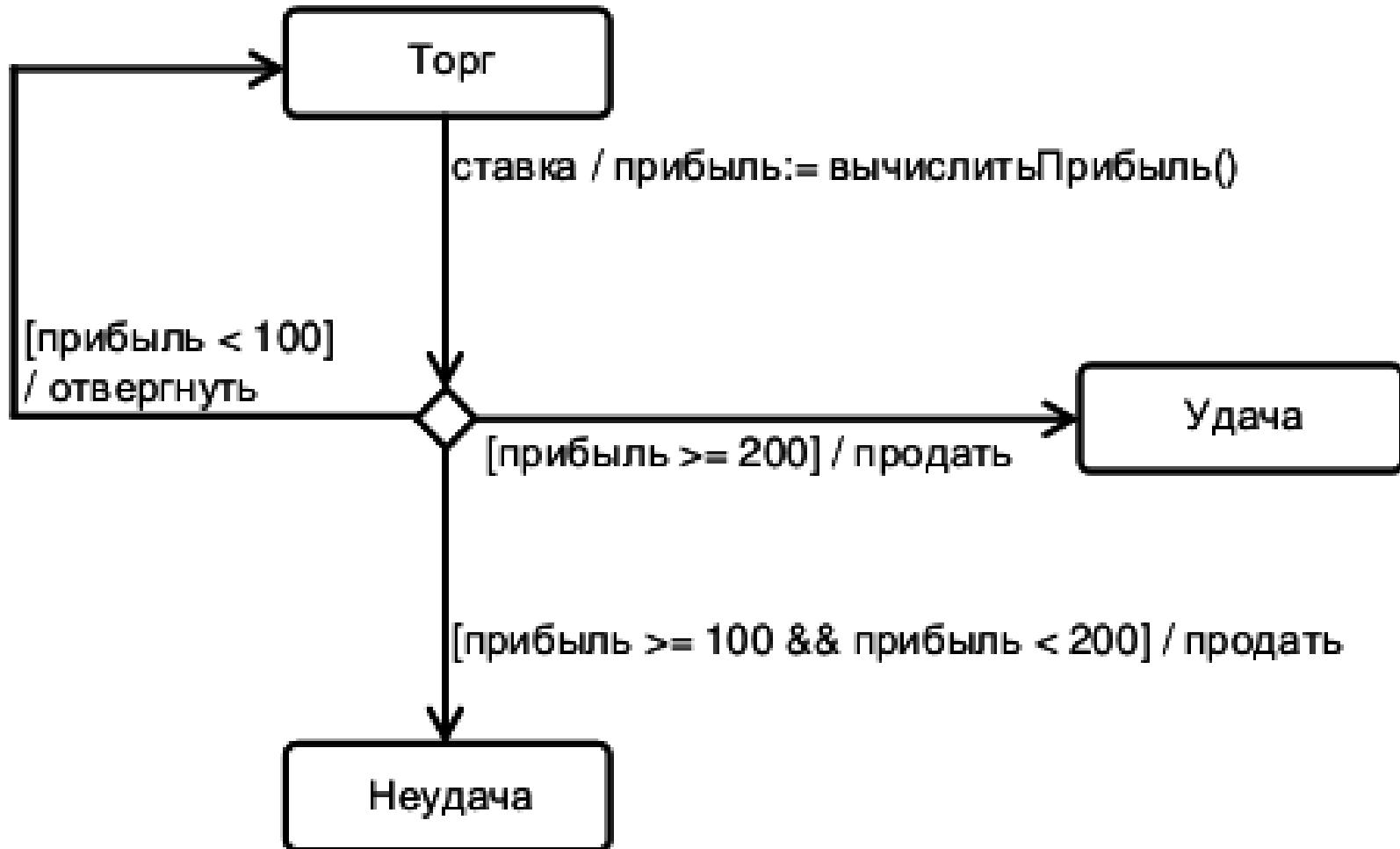
[S2]



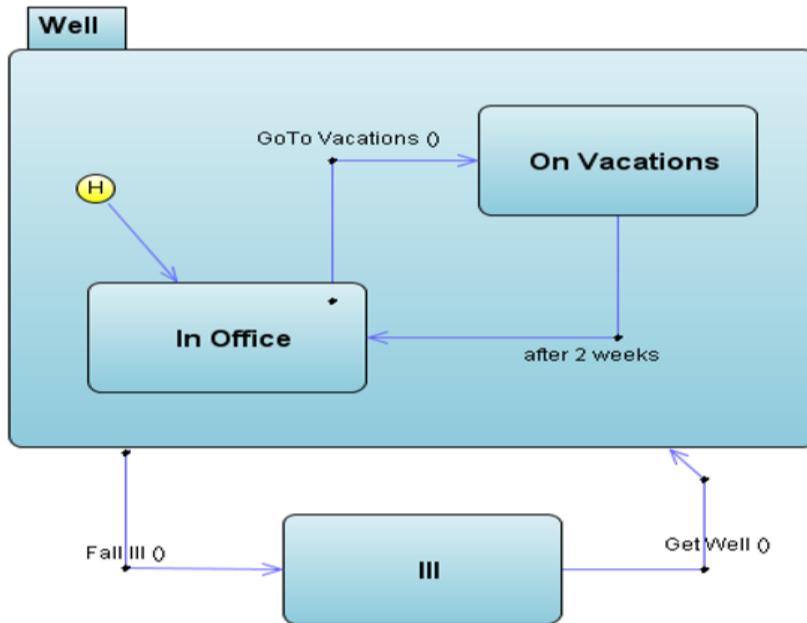
*Слияние и разделение переходов, сложные
переходы*



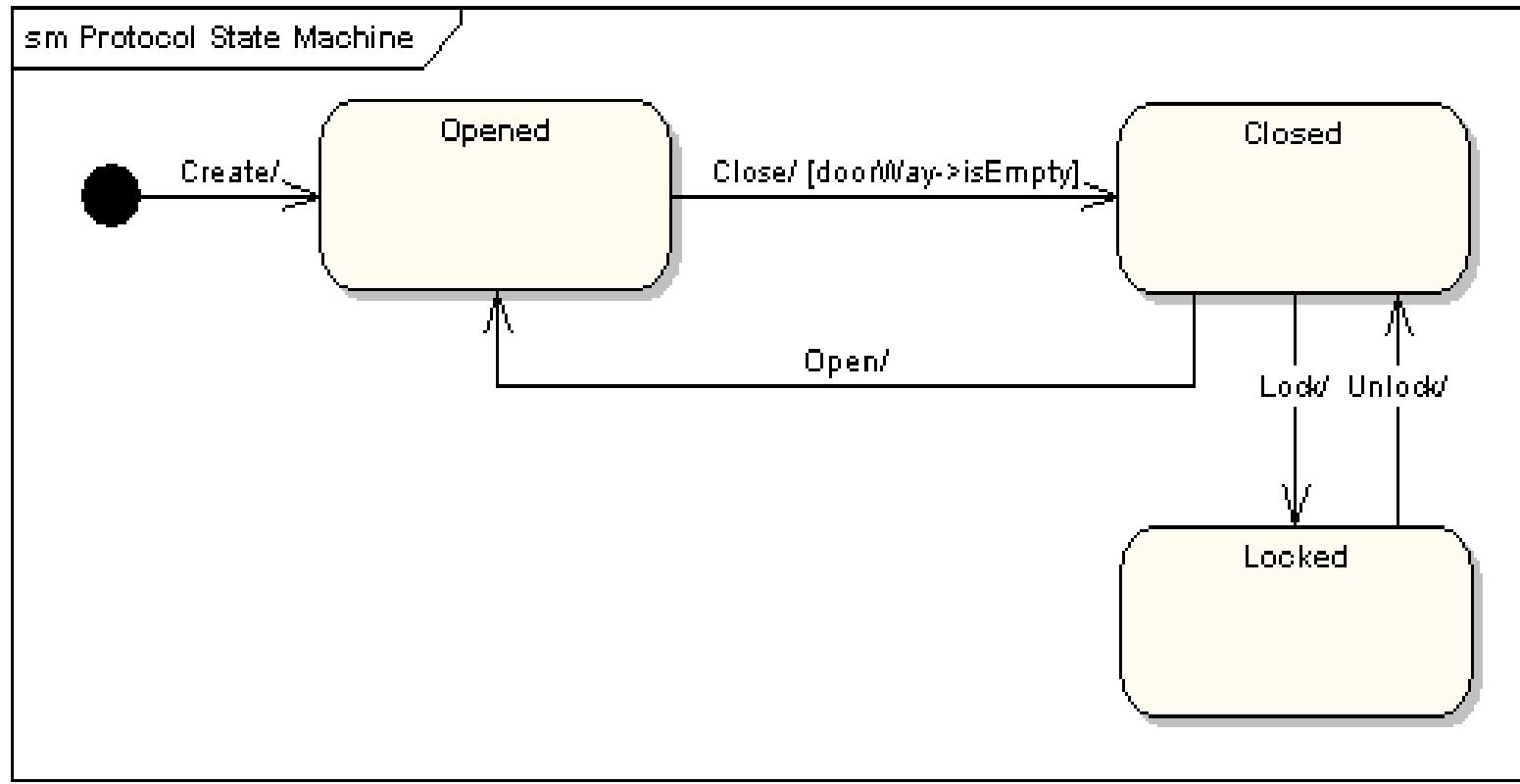
*Слияние и разделение переходов, сложные
переходы*



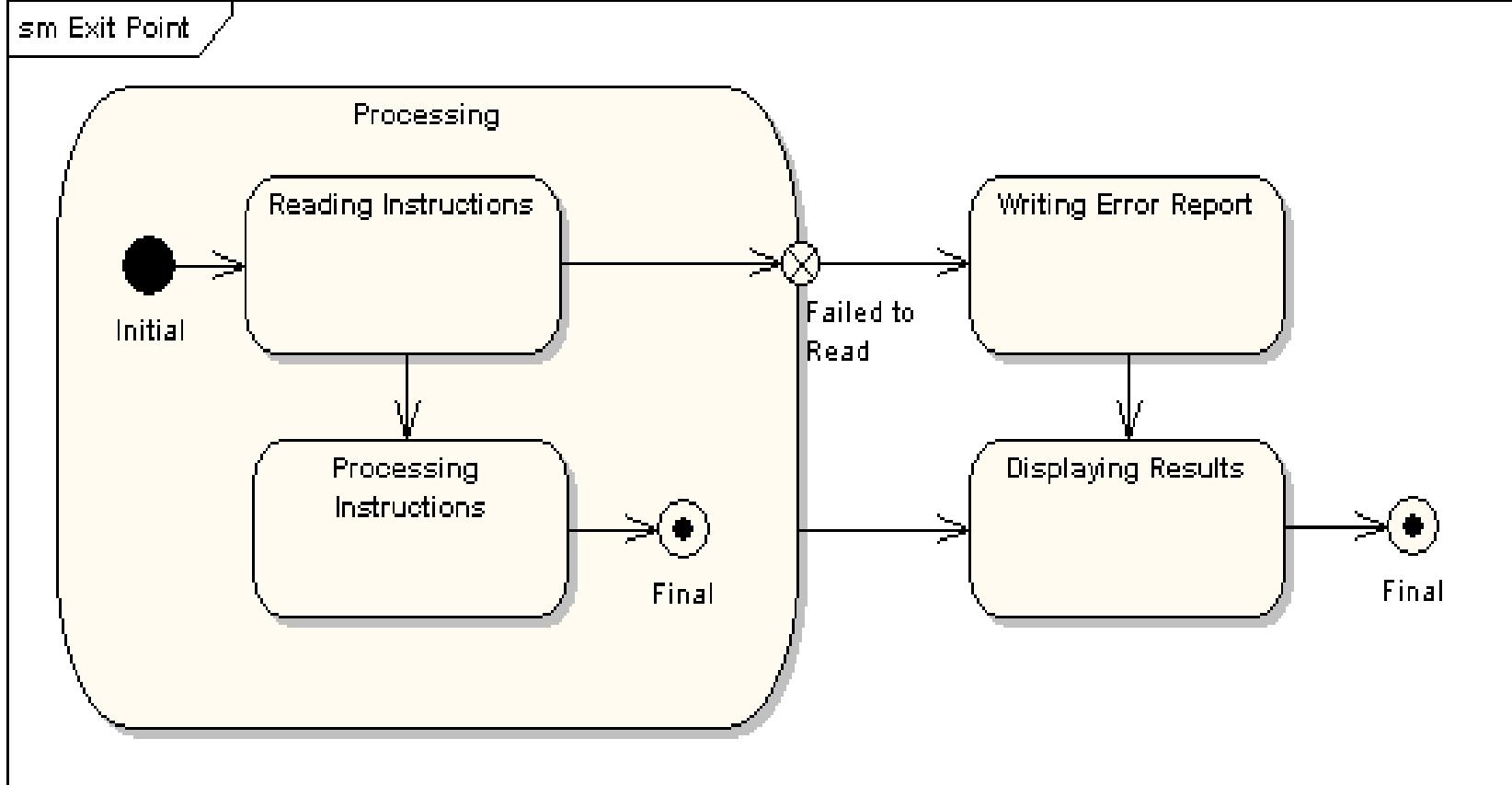
Псевдосостояния выбора



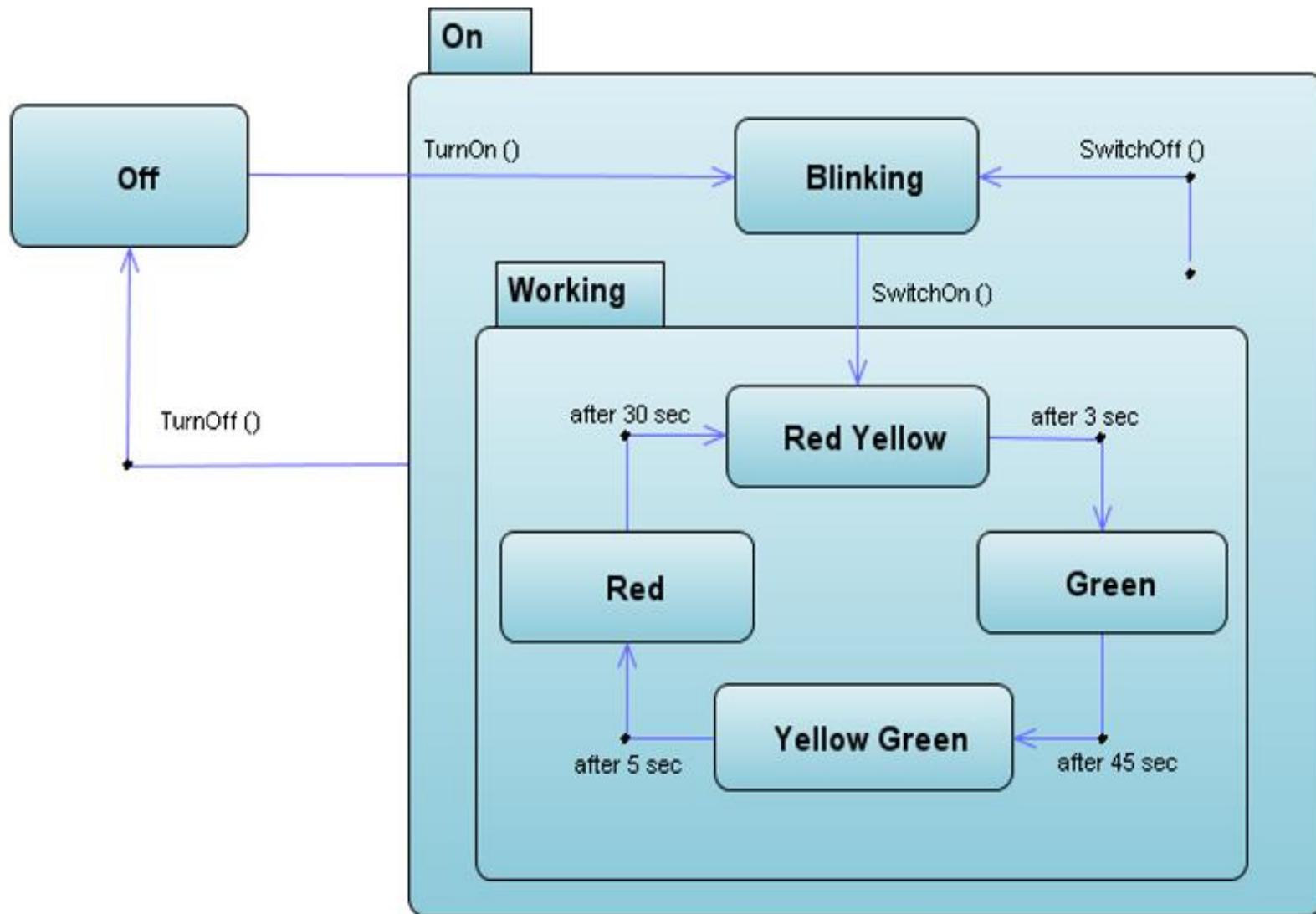
Работающий сотрудник, если все идет хорошо, пребывает в одном из двух взаимоисключающих состояний: либо он на работе (*inOffice*), либо в отпуске (*onVacations*). Но может случиться такая неприятность, как болезнь (*III*). Но в какое состояние переходит сотрудник по выздоровлении? Допустим, что в нашей информационной системе отдела кадров действует положение старого Комплекса законов о труде – если сотрудник заболел, находясь в отпуске, то отпуск прерывается, а по выздоровлении возобновляется. Для того, чтобы построить модель такого поведения, нужно воспользоваться историческим состоянием. В данном случае достаточно поверхности исторического состояния, поскольку на данном уровне вложенности все состояния уже простые.



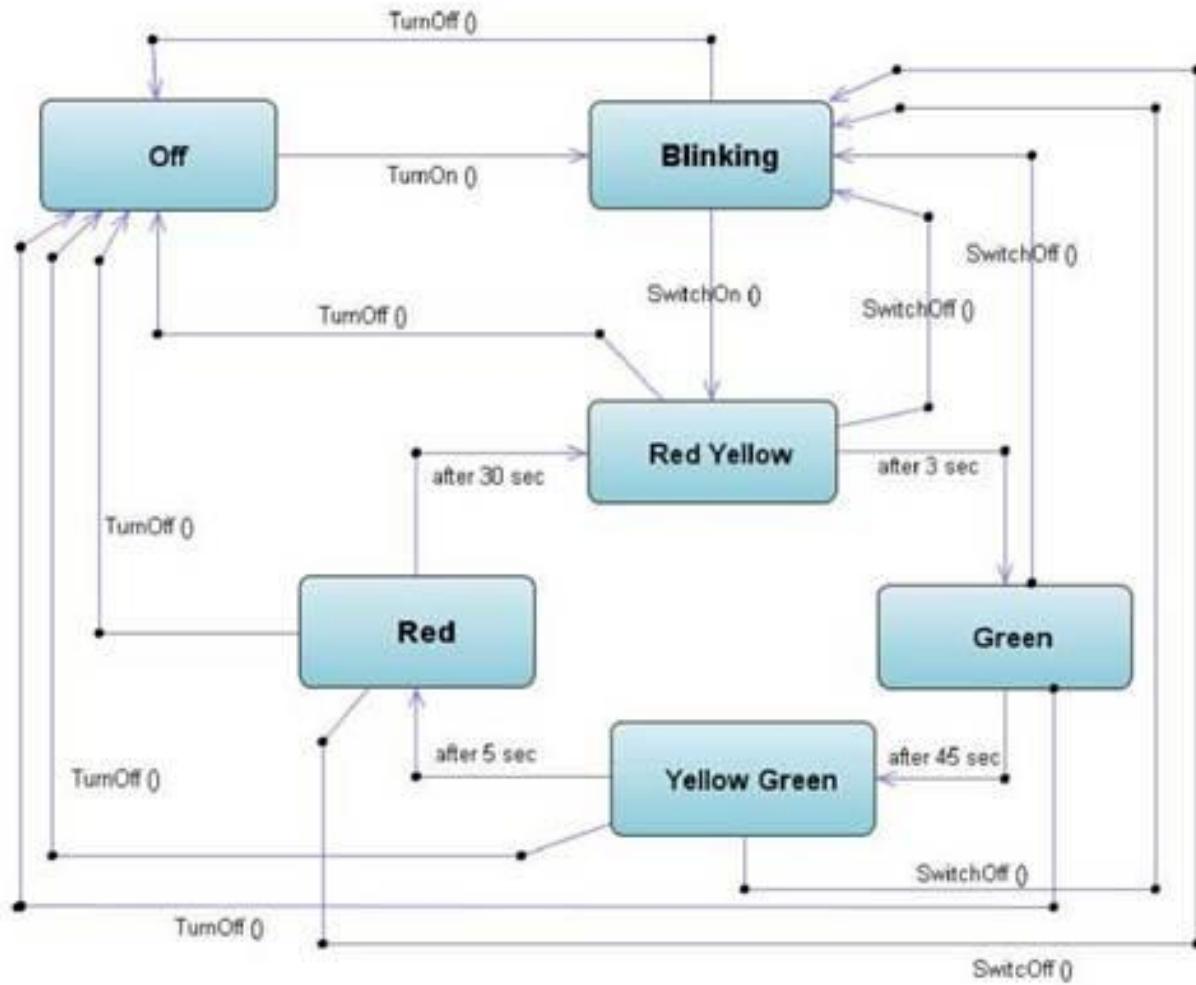
sm Exit Point



Выбор псевдосостояния



Работа Светофора



Работа Светофора (без составных состояний)

СПАСИБО ЗА ВНИМАНИЕ!

ПРОЕКТИРОВАНИЕ ПРОГРАММНЫХ СИСТЕМ

УНИФИЦИРОВАННЫЙ ЯЗЫК МОДЕЛИРОВАНИЯ UML

ЛЕКЦИЯ 4

КЛАССИФИКАЦИЯ ДИАГРАММ

Диаграммы поведения:

**Диаграммы вариантов
использования**

Диаграммы деятельности

Диаграммы состояний

**Диаграммы
взаимодействия:**

- Диаграммы последовательности
- Диаграммы коммуникации
- Диаграммы синхронизации
- Обзорные диаграммы взаимодействия

Структурные диаграммы:

- 1. Диаграммы классов**
- 2. Диаграммы объектов**
- 3. Диаграммы пакетов**
- 4. Диаграммы компонентов**
- 5. Диаграммы составной
структуры**
- 6. Диаграммы размещения**
- 7. Диаграммы профиля**

ОБЪЕКТНО- ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОЕКТИРОВАНИЕ

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОЕКТИРОВАНИЕ

Объектный подход к разработке сложных программных систем безусловно предполагает, что непосредственное программирование (написание кода) начинается далеко не сразу, а этапы анализа и моделирования предметной области, предшествующие программированию, не менее важны и сложны.

Объектный подход применяется на всех основных стадиях жизненного цикла ПО и включает в себя три ключевых понятия:

- ОOA (object oriented analysis) – объектно-ориентированный анализ.
- OOD (object oriented design) – объектно-ориентированное проектирование.
- OOP (object oriented programming) – объектно-ориентированное программирование.

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОЕКТИРОВАНИЕ

Объектно-ориентированный анализ – это методология анализа предметной области, при которой требования к проектируемой системе воспринимаются с точки зрения классов и объектов, выявленных в предметной области.

Объектно-ориентированное проектирование – это методология проектирования, соединяющая в себе процесс объектной декомпозиции и приемы представления логической и физической, а также статической и динамической моделей проектируемой системы.

Объектно-ориентированное программирование – это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

* *Методология - учение о методах, способах и стратегиях исследования предмета.*

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОЕКТИРОВАНИЕ

Предметом рассмотрения нашей дисциплины являются первые две составляющие объектного подхода – объектно-ориентированный анализ и объектно-ориентированное проектирование (далее – ООАП).

Последовательное применение ООАП позволяет получить "хороший" проект программной системы:

- удовлетворяющий требованиям заказчика;
- удобный для коллективной разработки, отладки и тестирования;
- прозрачный;
- развиваемый;
- допускающий повторное использование компонентов.

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОЕКТИРОВАНИЕ

**Базовыми принципами ООАП
являются:**

- **Декомпозиция**
- **Абстрагирование**
- **Иерархичность**
- **Многомодельность**

ПРИНЦИП ДЕКОМПОЗИЦИИ

Декомпозиция – это разбиение целого на составные элементы. В рамках объектного подхода рассматривают два вида декомпозиции: алгоритмическую и объектную.

В соответствии с алгоритмической декомпозицией предметной области при анализе задачи разработчик пытается понять, какие алгоритмы необходимо разработать для ее решения, каковы спецификации этих алгоритмов (вход, выход), и как эти алгоритмы связаны друг с другом. В языках программирования данный подход в полной мере поддерживается средствами модульного программирования (библиотеки, модули, подпрограммы).

Объектная декомпозиция предполагает выделение основных содержательных элементов задачи, разбиение их на типы (классы), определение свойств (данные) и поведения (операции) для каждого класса его, а также взаимодействия классов друг с другом. Объектная декомпозиция поддерживаются всеми современными объектно-ориентированными языками программирования.

ПРИНЦИП АБСТРАГИРОВАНИЯ

Абстрагирование применяется при решении многих задач – любая модель позволяет абстрагироваться от реального объекта, подменяя его изучение исследованием формальной модели.

Абстрагирование в ООП позволяет выделить основные элементы предметной области, обладающие одинаковой структурой и поведением. Такое разбиение предметной области на абстрактные классы позволяет существенно облегчить анализ и проектирование системы.

Согласно этому принципу в модель включаются только те аспекты проектируемой системы, которые имеют непосредственное отношение к выполнению системой своих функций.

ПРИНЦИП ИЕРАРХИЧНОСТИ

Принцип иерархичности предписывает рассматривать процесс построения модели на разных уровнях абстрагирования (детализации) в рамках фиксированных представлений.

Иерархия упорядочивает абстракции, помогает разбить задачу на уровни и постепенно ее решать по принципу "сверху – вниз" или "от общего – к частному", увеличивая детализацию ее рассмотрения на каждом очередном уровне.

ПРИНЦИП МНОГОМОДЕЛЬНОСТИ

Принцип многомодельности утверждает, что никакая единственная модель не может с достаточной степенью адекватности описывать различные аспекты сложной системы, и допускающий использование нескольких взаимосвязанных представлений, отражающих отдельные аспекты поведения или структуры систем.

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОЕКТИРОВАНИЕ

В целом же процесс ООАП можно рассматривать как последовательный переход от разработки наиболее общих моделей и представлений концептуального уровня к более частным и детальным представлениям логического и физического уровня.

При этом на каждом этапе ООАП данные модели последовательно дополняются все большим количеством деталей, что позволяет им более адекватно отражать различные аспекты конкретной реализации сложной системы.

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОЕКТИРОВАНИЕ

На рисунке приведена схема взаимосвязей представлений и моделей сложных систем в процессе ООАП: на двух уровнях иерархии (концептуальном и физическом) используются статические и динамические представления сложной системы.



ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОЕКТИРОВАНИЕ

Методология ООАП тесно связана с концепцией автоматизированной разработки ПО. Именно на этом фоне появление унифицированного языка моделирования (UML), ориентированного на комплексное решение задачи построения модели программной системы, которую, согласно современным концепциям ООАП, следует считать результатом первых двух этапов ЖЦ ПО, было в свое время воспринято сообществом корпоративных программистов с большим оптимизмом.

ОБЪЕКТНО- ОРИЕНТИРОВАННОЕ МОДЕЛИРОВАНИЕ СТРУКТУРЫ СИСТЕМЫ

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ МОДЕЛИРОВАНИЕ СТРУКТУРЫ СИСТЕМЫ

Моделируя структуру, мы описываем составные части системы и отношения между ними.

UML является объектно-ориентированным языком моделирования, поэтому не удивительно, что основным видом составных частей, из которых состоит система, являются объекты.

В каждый конкретный момент функционирования системы можно указать конечный набор конкретных объектов и связей между ними, образующих систему.

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ МОДЕЛИРОВАНИЕ СТРУКТУРЫ СИСТЕМЫ

Кроме самого компактного описания модели, подразумевается известным набор правил интерпретации описания, позволяющих построить по описанию множества любой его элемент.

Сами правила и способ их задания различны в разных случаях, но принцип один и тот же.

В UML этот принцип формализован в виде понятия дескриптора. Дескриптор имеет две стороны: это само описание множества (*intent*) и множество значений, описываемых дескриптором (*extent*).

Антонимом для дескриптора является понятие литерала. Литерал описывает сам себя.

Тип данных *integer* является дескриптором: он описывает множество целых чисел, потенциально бесконечное (или конечное, но достаточно большое, если речь идет о машинной арифметике). Изображение числа 1 описывает само число "один" и более ничего – это литерал.

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ МОДЕЛИРОВАНИЕ СТРУКТУРЫ СИСТЕМЫ

Почти все элементы моделей UML являются дескрипторами – именно поэтому средствами UML удается создавать представительные модели достаточно сложных систем.

Варианты использования и действующие лица – дескрипторы, классы, ассоциации, компоненты, узлы – также дескрипторы. Примечание же является литералом — оно описывает само себя.

Важнейшим типом дескрипторов являются классификаторы.

Классификатор – это дескриптор множества однотипных объектов.

ПАРАДИГМА ПРОГРАММИРОВАНИЯ

Парадигма программирования — это собрание основополагающих принципов, которые служат методической основой конкретных технологий и инструментальных средств программирования.

Центральной идеей парадигмы объектно-ориентированного программирования является инкапсуляция, т. е. структурирование программы на структуры особого вида, объединяющего данные и процедуры их обработки, причем внутренние данные структуры не могут быть обработаны иначе, кроме как предусмотренными для этого процедурами.

ПАРАДИГМА ПРОГРАММИРОВАНИЯ

Объект – структура из данных и процедур их обработки, существующая в памяти компьютера во время выполнения программы.

Класс – описание множества однотипных объектов в тексте программы.

За редкими исключениями в современных объектно-ориентированных системах программирования классы являются дескрипторами.

ПАРАДИГМА ПРОГРАММИРОВАНИЯ

Кроме основной идеи инкапсуляции, с объектно-ориентированным программированием принято ассоциировать также понятия наследования и полиморфизма.

Наследование – это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью..

Полиморфизм – это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

UML (ПРОДОЛЖЕНИЕ)

ДИАГРАММЫ КЛАССОВ

КЛАССИФИКАЦИЯ ДИАГРАММ

Диаграммы поведения:

**Диаграммы вариантов
использования**

Диаграммы деятельности

Диаграммы состояний

**Диаграммы
взаимодействия:**

- Диаграммы последовательности
- Диаграммы коммуникации
- Диаграммы синхронизации
- Обзорные диаграммы взаимодействия

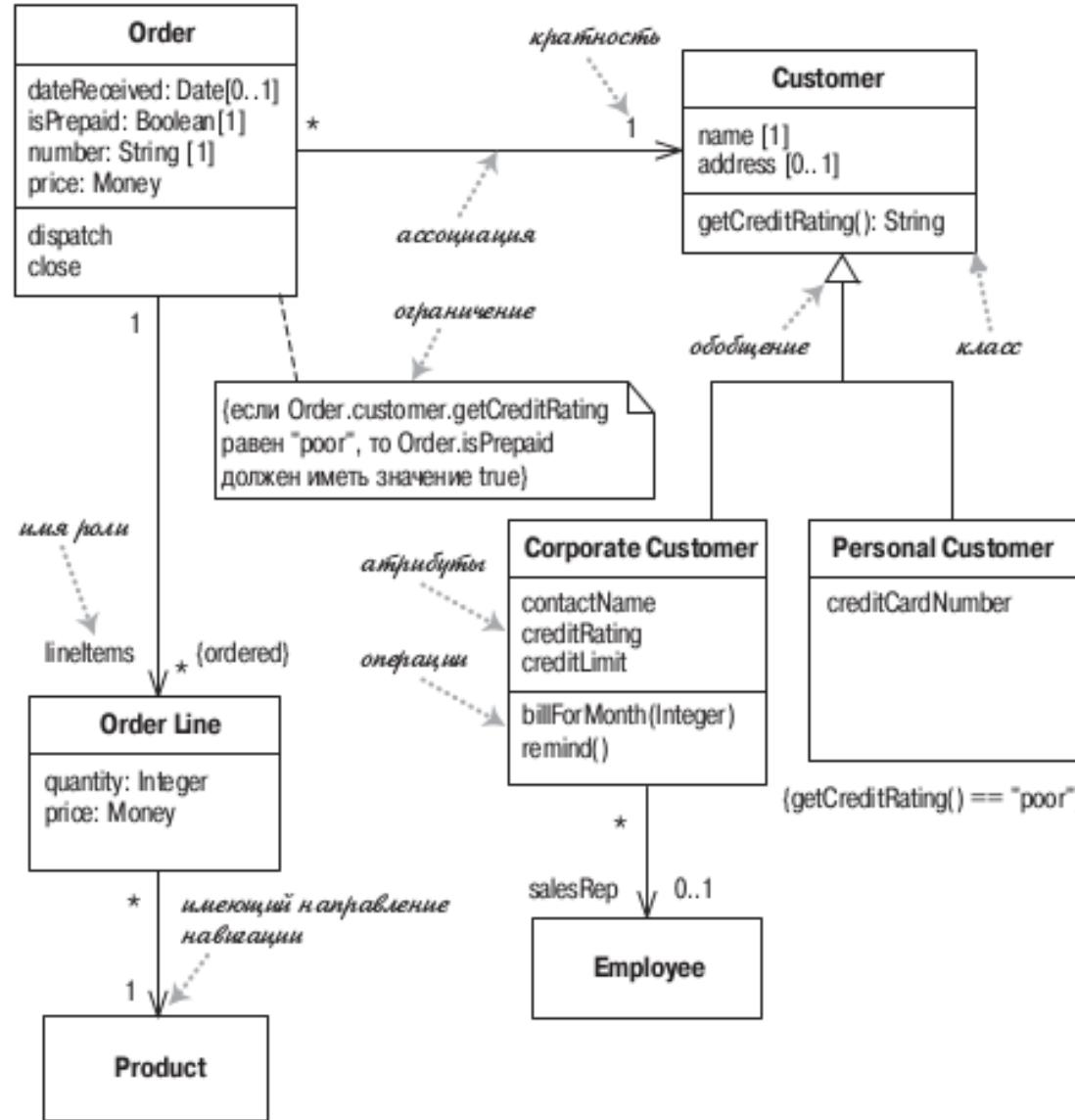
Структурные диаграммы:

- 1. Диаграммы классов**
- 2. Диаграммы объектов**
- 3. Диаграммы пакетов**
- 4. Диаграммы компонентов**
- 5. Диаграммы составной
структуры**
- 6. Диаграммы размещения**
- 7. Диаграммы профиля**

ДИАГРАММА КЛАССОВ(CLASS DIAGRAM) –

основной способ описания структуры системы.
На диаграмме классов применяются один основной тип сущностей: классы (включая многочисленные частные случаи классов: интерфейсы, типы, классы ассоциации и многие другие), между которыми устанавливаются следующие основные типы отношений:

- **ассоциация между классами (с множеством дополнительных подробностей);**
- **обобщение между классами;**
- **зависимость (различных типов) между классами;**
- **реализация.**



Пример диаграммы классов

ИНТЕРФЕЙСЫ И АБСТРАКТНЫЕ КЛАССЫ

Абстрактный класс (*abstract class*) – это класс, который нельзя реализовать непосредственно.

Абстрактная операция (*abstract operation*) - это чистое объявление, которое клиенты могут привязать к абстрактному классу.

Интерфейс – это класс, не имеющий реализаций, то есть вся его функциональность абстрактна.

КЛАСС

Класс – один из самых "богатых" элементов моделирования UML. Описание класса может включать множество различных элементов, их группируют по разделам.

Стандартных разделов три:

- раздел имени – наряду с обязательным именем может содержать также стереотип, кратность (т.е. ограничение на количество экземпляров) и список свойств;**
- раздел атрибутов – содержит список описаний атрибутов класса;**
- раздел операций – содержит список описаний операций класса.**

КЛАСС

Имя

Имя

Атрибуты

Имя

Атрибуты

Операции

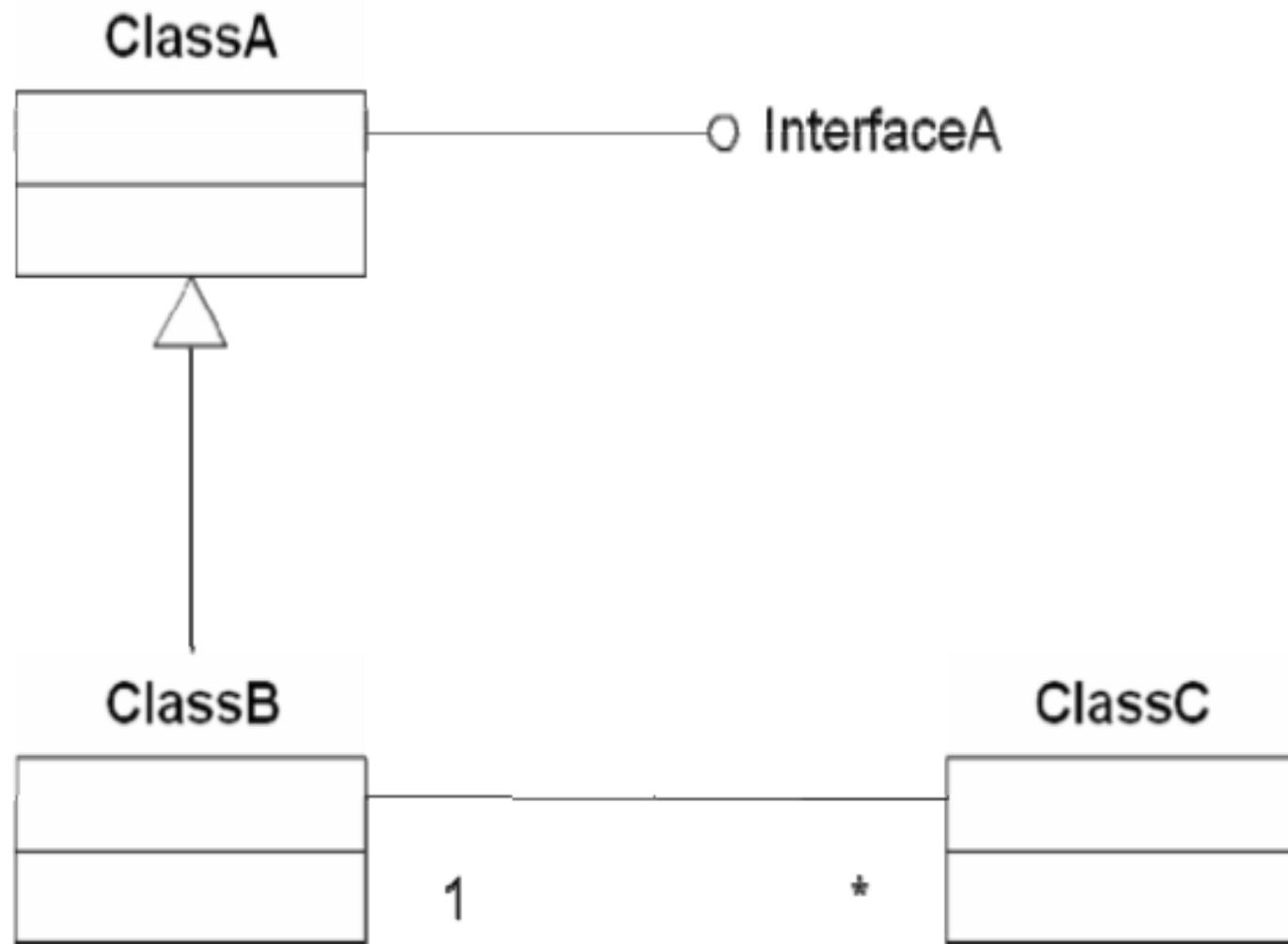
КЛАСС

Нотация классов очень проста – это всегда прямоугольник. Если разделов более одного, то внутренность прямоугольника делится горизонтальными линиями на части, соответствующие разделам. Содержимым раздела в любом случае является текст. Текст внутри стандартных разделов должен иметь определенный синтаксис.

Некоторые инструменты позволяют помещать в разделы класса не только тексты, но также фигуры и значки.

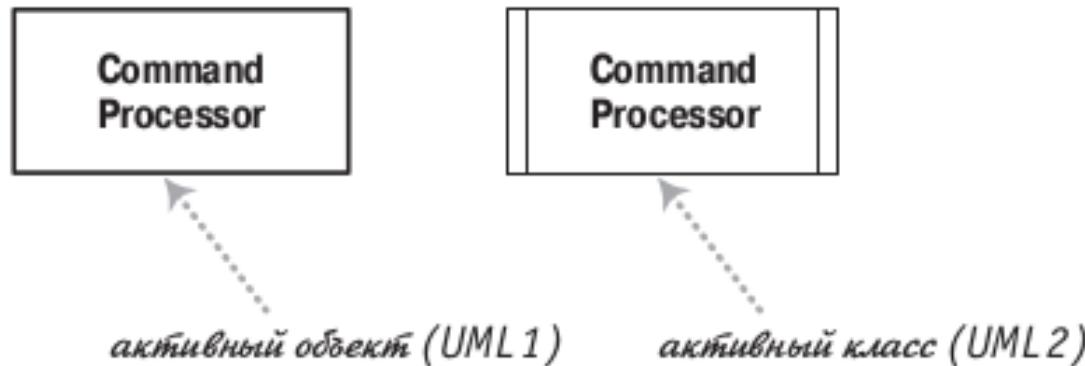
Раздел имени класса в общем случае имеет следующий синтаксис:

«стереотип» ИМЯ {свойства} кратность



АКТИВНЫЙ КЛАСС

Активный класс (active class) имеет экземпляры, каждый из которых выполняет и управляет собственным потоком управления.

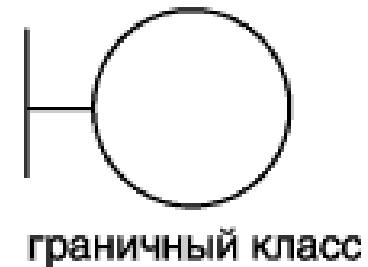


ГРАНИЧНЫЙ КЛАСС

Граничные классы

Граничными классами (boundary classes) называются такие классы, которые расположены на границе системы и всей окружающей среды. Это экранные формы, отчеты, интерфейсы с аппаратурой (такой как принтеры или сканеры) и интерфейсы с другими системами.

Чтобы найти граничные классы, надо исследовать диаграммы вариантов использования. Каждому взаимодействию между действующим лицом и вариантом использования должен соответствовать, по крайней мере, один граничный класс. Именно такой класс позволяет действующему лицу взаимодействовать с системой.



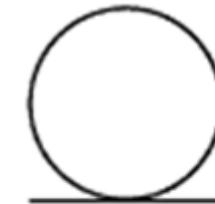
границный класс



КЛАССЫ-СУЩНОСТИ

Классы-сущности

Классы-сущности (entity classes) содержат хранимую информацию. Они имеют наибольшее значение для пользователя, и потому в их названиях часто используют термины из предметной области. Обычно для каждого класса-сущности создают таблицу в базе данных.



класс-сущность

<<entity>>
Имя класса

УПРАВЛЯЮЩИЕ КЛАССЫ

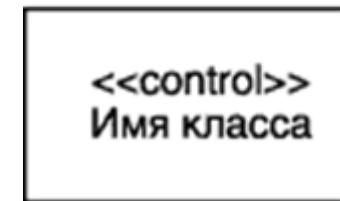
Управляющие классы

Управляющие классы (control classes) отвечают за координацию действий других классов. Обычно у каждого варианта использования имеется один управляющий класс, контролирующий последовательность событий этого варианта использования.

Управляющий класс отвечает за координацию, но сам не несет в себе никакой функциональности, так как остальные классы не посыпают ему большого количества сообщений. Вместо этого он сам посыпает множество сообщений. Управляющий класс просто делегирует ответственность другим классам, по этой причине его часто называют классом-менеджером.



управляющий класс



СТАНДАРТНЫЕ СТЕРЕОТИПЫ КЛАССОВ

Стереотип	Описание
actor	действующее лицо
enumeration	перечислимый тип данных
exception	сигнал, распространяемый по иерархии обобщений
implementation	реализация класса
Class	
interface	нет атрибутов и все операции абстрактные
metaclass	экземпляры являются классами
powertype	метакласс, экземплярами которого являются все наследники данного класса
process, thread	активные классы
signal	класс, экземплярами которого являются сообщения
stereotype	стереотип
type (datatype)	тип данных
utility	нет экземпляров = служба

СВОЙСТВА КЛАССА

Свойства представляют структурную функциональность класса.

Свойства представляют единое понятие, воплощающееся в двух совершенно различных сущностях: в атрибутах и в ассоциациях.

АТРИБУТЫ

Атрибут – это именованное место (или, как говорят, слот), в котором может храниться значение.

Атрибуты класса перечисляются в разделе атрибутов. В общем случае описание атрибута имеет следующий синтаксис.

видимость ИМЯ кратность: тип = начальное_значение {свойства}

Пример:

-имя: String [1] = "Без имени" {readOnly}

-Видимость, как обычно, обозначается знаками +, -, # (соответственно открыт, закрыт, защищен). Если видимость не указана, то никакого значения видимости по умолчанию не подразумевается.

Если имя атрибута подчеркнуто, то это означает, что областью действия данного атрибута является класс, а не экземпляр класса, как обычно. Другими словами, все объекты – экземпляры этого класса совместно используют одно и тоже значение данного атрибута, общее для всех экземпляров. В обычной ситуации (нет подчеркивания) каждый экземпляр класса хранит свое индивидуальное значение атрибута.

ЗАМЕЧАНИЕ Подчеркивание описания атрибута соответствует описателю static в языке С++.

АТРИБУТЫ

видимость ИМЯ кратность: тип = начальное_значение {свойства}

Кратность, если она присутствует, определяет данный атрибут как массив (определенной или неопределенной длины).

Тип атрибута – это либо примитивный (встроенный) тип, либо тип, определенный пользователем. Начальное значение имеет очевидный смысл: при создании экземпляра данного класса атрибут получает указанное значение. Если начальное значение не указано, то никакого значения по умолчанию не подразумевается.

Как и любой другой элемент модели, атрибут может быть наделен дополнительными свойствами в форме ограничений и именованных значений.

У атрибутов имеется еще одно стандартное свойство: **изменяемость**.

ЗНАЧЕНИЯ СВОЙСТВА ИЗМЕНЯЕМОСТИ АТРИБУТА

Значение	Описание
changeable	Никаких ограничений на изменение значения атрибута не накладывается. Данное значение имеет место по умолчанию, поэтому указывать в модели его излишне.
addOnly	Это значение применимо только к атрибутам, кратность которых больше единицы. При изменении значения атрибута новое значение добавляется в массив значений, но старые значения не меняются и не исчезают. Такой атрибут "помнит" историю своего изменения.
frozen	Значение атрибута задается при инициализации объекта и не может меняться.

ПРИМЕР

Например, в информационной системе отдела кадров класс Person, скорее всего, должен иметь атрибут, хранящий имя сотрудника.

В табл. приведен список примеров описаний такого атрибута. Все описания синтаксически допустимы и могут быть использованы в соответствии с текущим уровнем детализации модели.

Пример

name

Пояснение

Минимальное возможное описание — указано только имя атрибута

+name

Указаны имя и открытая видимость — предполагается, манипуляции с именем будут производится непосредственно

-name : String

Указаны имя, тип и закрытая видимость — манипуляции с именем будут производится с помощью специальных операций

-name [1..3] : String

Указана кратность (для хранения трех составляющих; фамилии, имени и отчества)

-name : String = "Novikov"

Указано начальное значение

+name : String {frozen}

Атрибут объявлен не меняющим своего значения после начального присваивания и открытым

КРАТНОСТЬ АТРИБУТА

[0..1] означает, что кратность атрибута может принимать значение 0 или 1. При этом 0 означает отсутствие значения для данного атрибута.

[0..*] означает, что кратность атрибута может принимать любое положительное целое значение большее или равное 0. Эта кратность может быть записана короче в виде простого символа - [*].

[1..*] означает, что кратность атрибута может принимать любое положительное целое значение большее или равное 1.

[1..5] означает, что кратность атрибута может принимать любое значение из чисел: 1, 2, 3, 4, 5.

[1..3,5,7] означает, что кратность атрибута может принимать любое значение из чисел: 1, 2, 3, 5, 7.

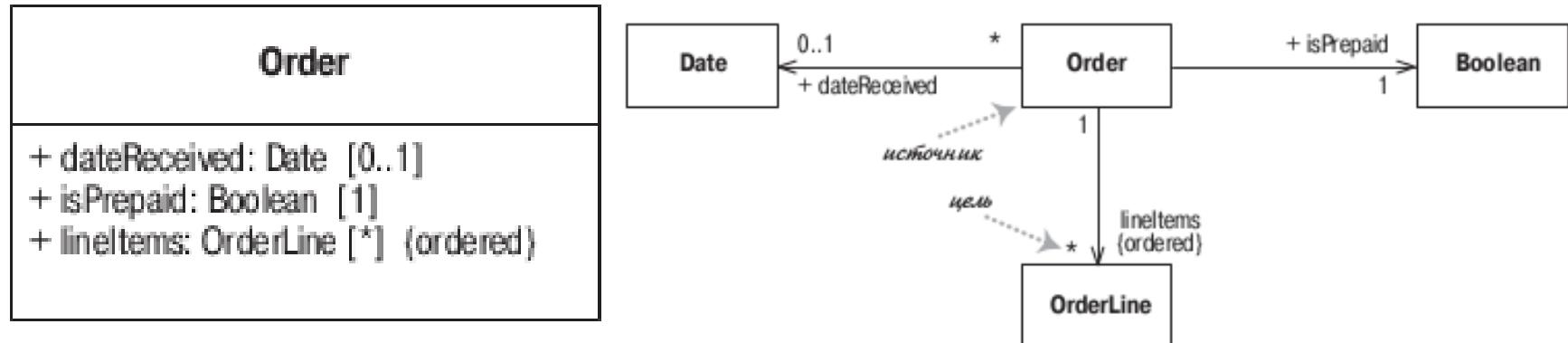
[1..3,7.. 10] означает, что кратность атрибута может принимать любое значение из чисел: 1, 2, 3, 7, 8, 9, 10.

[1..3,7..*] означает, что кратность атрибута может принимать любое значение из чисел: 1, 2, 3, а также любое положительное целое значение большее или равное 7.

Если кратность атрибута не указана, то по умолчанию принимается ее значение равное 1..1, т. е. в точности 1.

АССОЦИАЦИИ

Ассоциация – это непрерывная линия между двумя классами, направленная от исходного класса к целевому классу. Имя свойства (вместе с кратностью) располагается на целевом конце ассоциации.



ЗАВИСИМОСТИ

Всего в UML определено 17 стандартных стереотипов отношения зависимости, которые можно разделить на 6 групп:

между классами и объектами на диаграмме классов;

между пакетами;

между вариантами использования;

между объектами на диаграмме взаимодействия;

между состояниями автомата;

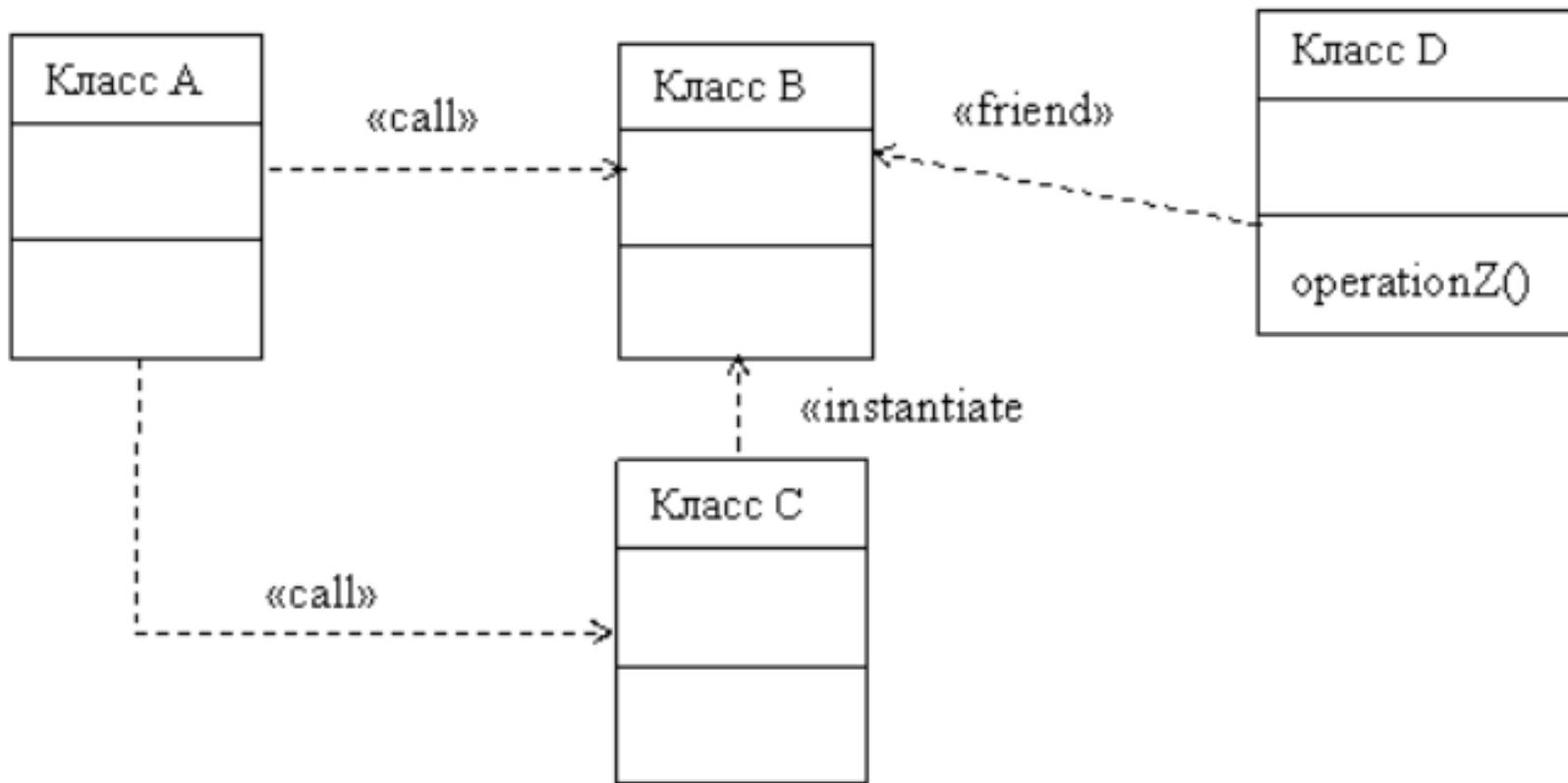
между подсистемами и моделями.

Зависимости на диаграммах классов используются сравнительно редко, потому что имеют более расплывчатую семантику по сравнению с ассоциациями и обобщением.

СТАНДАРТНЫЕ СТЕРЕОТИПЫ ЗАВИСИМОСТЕЙ НА ДИАГРАММЕ КЛАССОВ

Стереотип	Применение
bind	Подстановка параметров в шаблон. Независимой сущностью является шаблон (класс с параметрами), а зависимой — класс, который получается из шаблона заданием аргументов.
derive	Буквально означает "может быть вычислен по". Зависимость с данным стереотипом применяется не только к классам, но и к другим элементам модели: атрибутам, ассоциациям и др. Суть состоит в том, зависимый элемент может быть восстановлен по информации, содержащейся в независимом элементе. Таким образом, данная зависимость показывает, что зависимый элемент, вообще говоря, излишен и введен в модель из соображений удобства, наглядности и т.д.
friend	Назначает специальные права видимости. Зависимый класс имеет доступ к составляющим независимого класса, даже если по общим правилам видимости он не имеет на это прав.
instanceOf	Указывает, что зависимый объект (или класс) является экземпляром независимого класса (метакласса).
instantiate	Указывает, что операции независимого класса создают экземпляры зависимого класса.
powertype	Показывает, что экземплярами зависимого класса являются подклассы независимого класса. Таким образом, в данном случае зависимый класс является метаклассом.
refine	Указывает, что зависимый класс уточняет (конкретизирует) независимый. Данная зависимость показывает, что связанные классы концептуально совпадают, но находятся на разных уровнях абстракции.
use	Зависимость самого общего вида, показывающая, что зависимый класс каким-либо образом использует независимый класс.

ЗАВИСИМОСТИ



классификатор (в частности, класс) использует интерфейс – это показывается с помощью зависимости со стереотипом «call»

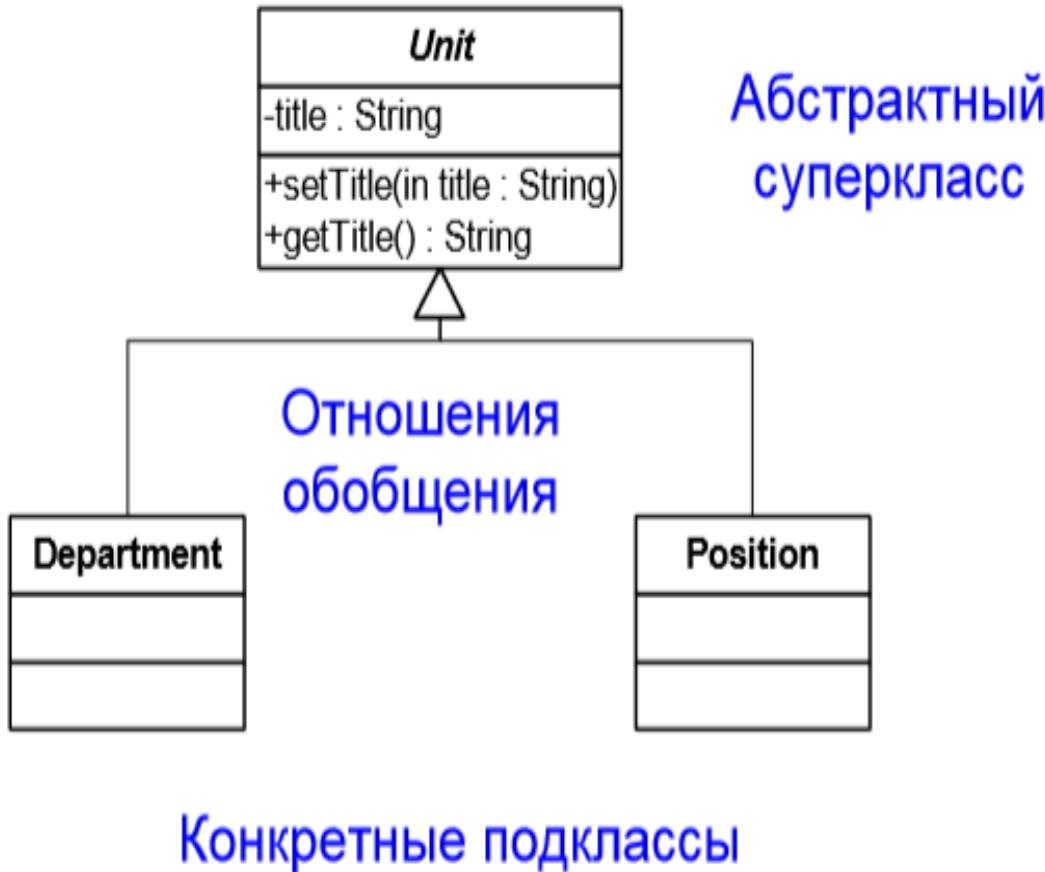
ОБОБЩЕНИЕ

Отношение обобщения часто применяется на диаграмме классов. . Как правило, между объектами в одной системе общее есть и это общее целесообразно выделить в отдельный класс. При этом общие составляющие, собранные в суперклассе, автоматически наследуются подклассами. Таким образом, сокращается общее количество описаний, а значит, уменьшается вероятность допустить ошибку.

Использование обобщений не ограничивает свободу проектировщика системы, поскольку унаследованные составляющие можно переопределить в подклассе, если нужно. При обобщении выполняется принцип подстановочности. Фактически это означает увеличение гибкости и универсальности программного кода при одновременном сохранении надежности, обеспечиваемой контролем типов.

Если, например, в качестве типа параметра некоторой процедуры указать суперкласс, то процедура будет с равным успехом работать в случае, когда в качестве аргумента ей передан объект любого подкласса данного суперкласса. Суперкласс может быть конкретным, идентифицированным, а может быть абстрактным, введенным именно для построения отношений обобщения.

ОБОБЩЕНИЕ



В UML допускается, чтобы класс был подклассом нескольких суперклассов (множественное наследование), не требуется, чтобы у базовых классов был общий суперкласс (несколько иерархий обобщения) и вообще не накладывается никаких ограничений, кроме частичной упорядоченности (т. е. отсутствия циклов в цепочках обобщений).

АССОЦИАЦИИ

Отношение ассоциации является самым важным на диаграмме классов. В общем случае ассоциация, которая обозначается сплошной линией, соединяющей классы, означает, что экземпляры одного класса связаны с экземплярами другого класса. Поскольку экземпляров может быть много, и каждый может быть связан с несколькими, ясно, что ассоциация является дескриптором, который описывает множество связанных объектов.

В UML ассоциация является классификатором, экземпляры которого называются связями. Связь между объектами (экземплярами классов) в программе может быть организована самыми разными способами.

При моделировании на UML техника реализации связи между объектами не имеет значения. Ассоциация в UML подразумевает лишь то, что связанные объекты обладают достаточной информацией для организации взаимодействия. Возможность взаимодействия означает, что объект одного класса может послать сообщение объекту другого класса, в частности, вызвать операцию или же прочитать, или изменить значение открытого атрибута.

АССОЦИАЦИИ

Моделирование структуры взаимосвязей объектов (т.е. выявление ассоциаций) является одной из ключевых задач при разработке.

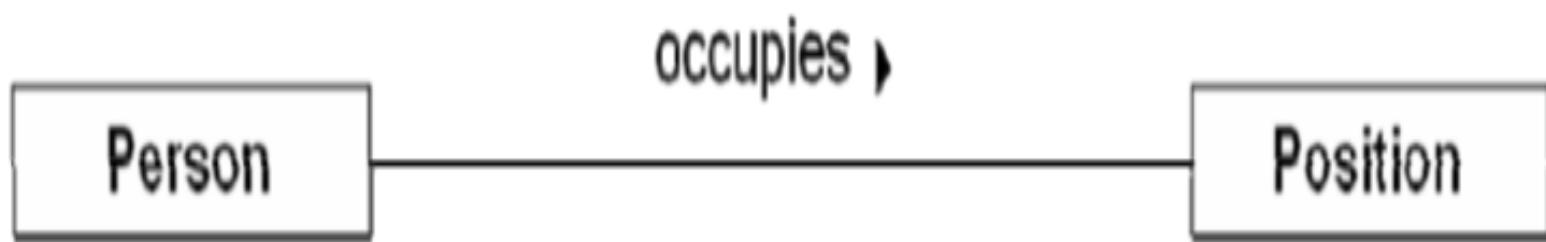
Базовая нотация ассоциации (сплошная линия) позволяет указать, что объекты ассоциированных классов могут взаимодействовать во время выполнения.

Но это только малая часть того, что можно моделировать с помощью отношения ассоциации. Для ассоциации в UML предусмотрено наибольшее количество различных дополнений.

Дополнения не являются обязательными: их используют при необходимости, в различных ситуациях по-разному. Если использовать все дополнения сразу, то диаграмма становится настолько перегруженной, что ее трудно читать.

АССОЦИАЦИИ

Имя ассоциации указывается в виде строки текста над (или под, или рядом с) линией ассоциации. Имя не несет дополнительной семантической нагрузки, а просто позволяет различать ассоциации в модели. Обычно имя не указывают, за исключением многополюсных ассоциаций или случая, когда одна и та же группа классов связана несколькими различными ассоциациями. Дополнительно можно указать направление чтения имени ассоциации.



АССОЦИАЦИИ

Итак, для ассоциации определены следующие дополнения:

- *имя ассоциации (возможно, вместе с направлением чтения);*
- *кратность полюса ассоциации;*
- *вид агрегации полюса ассоциации;*
- *роль полюса ассоциации;*
- *направление навигации полюса ассоциации;*
- *упорядоченность объектов на полюсе ассоциации;*
- *изменяемость множества объектов на полюсе ассоциации;*
- *квалификатор полюса ассоциации;*
- *класс ассоциации;*
- *видимость полюса ассоциации;*
- *многополюсные ассоциации.*
- *Полюсом называется конец линии ассоциации. Обычно используются двухполюсные ассоциации, но могут быть и многополюсные.*

ПОЛЮС АССОЦИАЦИИ

Полюс ассоциации находится у края прямоугольника, обозначающего класс. Свойства полюса ассоциации передаются в виде различных обозначений, отображаемых около соответствующего конца маршрута ассоциации.

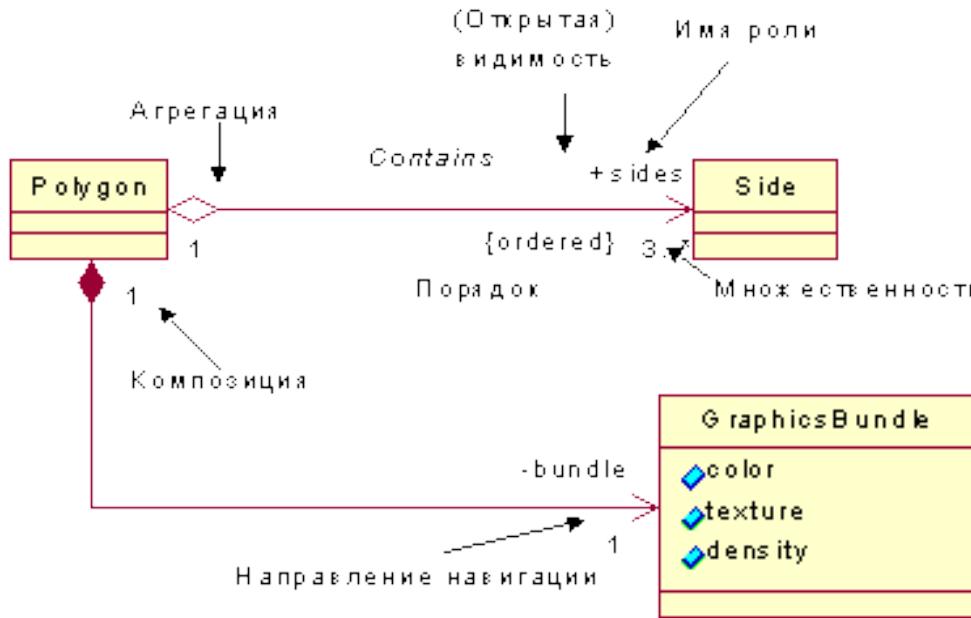
Роль полюса ассоциации, называемая также спецификатором интерфейса – это способ указать, как именно участвует классификатор (присоединенный к данному полюсу ассоциации) в ассоциации. В общем случае данное дополнение имеет следующий синтаксис:

видимость ИМЯ : тип

СПИСОК СВОЙСТВ ПОЛЮСОВ АССОЦИАЦИИ

aggregation (агрегация)	Маленький не закрашенный ромб на полюсе агрегации; для композиции используется закрашенный ромб.
changeability (изменяемость)	Пометки {frozen} или {addQOnly} около целевого полюса ассоциации. Значение по умолчанию {changeable} часто опускается.
interface specifier (спецификатор типа интерфейса)	Текстовый суффикс для имени роли - : тип.
multiplicity (множественность)	Текстовая наметка у полюса ассоциации - min..max.
navigability (возможность навигации)	Наконечник стрелки на конце маршрута ассоциации, показывающий возможность навигации в этом направлении. Если указатель возможности навигации отсутствует, то принято считать, что навигация возможна в обоих направлениях (так как нечасто возникает потребность ассоциации, в которой навигация вообще невозможна).
ordering (порядок)	Текстовая пометка {ordered} возле целевого полюса ассоциации, которая обозначает упорядоченность списка экземпляров целевого класса.
qualifier (квалифициратор)	Маленький прямоугольник между концом маршрута и исходным классом. Прямоугольник содержит один или несколько атрибутов ассоциации - квалифицираторов.
rolename (имя роли)	Текстовая пометка у целевого полюса ассоциации.
target scope (целевая область действия)	Указывает на то, что имя роли действительно в области действия класса. В противном случае - в области действия его экземпляра.
visibility (видимость)	Один из символов видимости ("+", "#", "-"), стоящий перед именем

ПОЛЮС АССОЦИАЦИИ



Различные указатели, применяемые на полюсах ассоциации

КРАТНОСТЬ ПОЛЮСА АССОЦИАЦИИ

Кратность полюса ассоциации указывает, сколько объектов данного класса (со стороны данного полюса) участвуют в связи. Кратность может быть задана как конкретное число, и тогда в каждой связи со стороны данного полюса участвуют ровно столько объектов, сколько указано. Более распространен случай, когда кратность указывается как диапазон возможных значений, и тогда число объектов, участвующих в связи должно находиться в пределах указанного диапазона. При указании кратности можно использовать символ *, который обозначает неопределенное число.

Например, если в информационной системе отдела кадров не предусматривается дробление ставок и совмещение должностей, то (работающему)

соответствует о

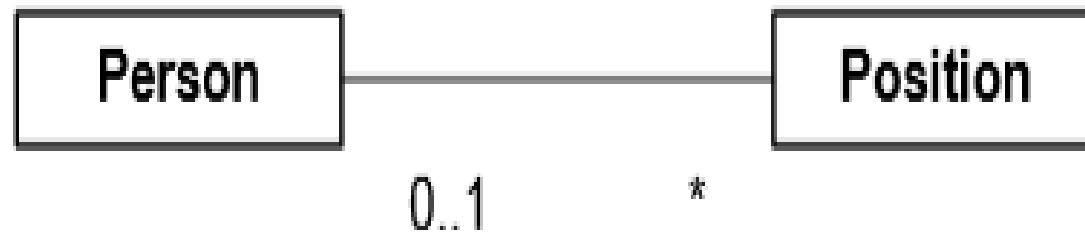


Кратность
задана
диапазоном

Кратность
задана
числом

АССОЦИАЦИИ

Более сложные случаи также легко моделируются с помощью кратности полюсов. Например, если мы хотим предусмотреть совмещение должностей и хранить информацию даже о неработающих сотрудниках, то диаграмма примет вид:



АССОЦИАЦИИ

В UML используются два частных, но очень важных случая отношения ассоциации, которые называются агрегацией и композицией.

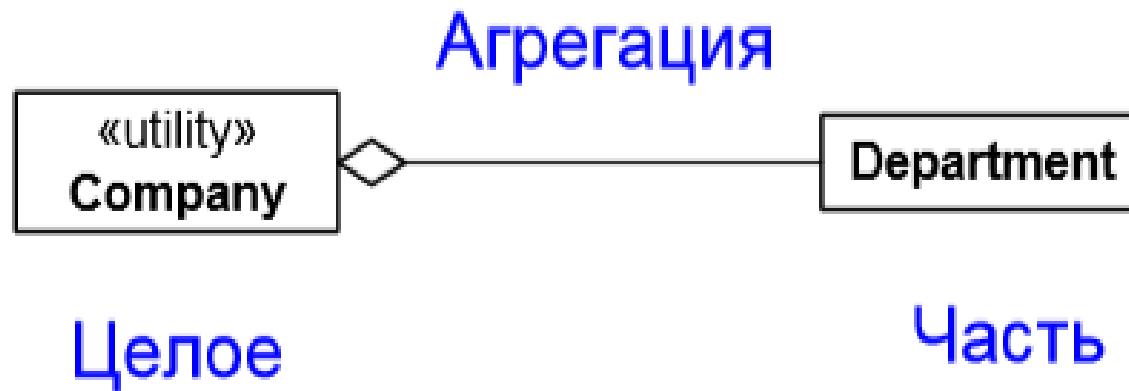
В обоих случаях речь идет о моделировании отношения типа «часть – целое». Ясно, что отношения такого типа следует отнести к отношениям ассоциации, поскольку части и целое обычно взаимодействуют.

Агрегация от класса A к классу B означает, что объекты (один или несколько) класса A входят в состав объекта класса B.

Это отмечается с помощью специального графического дополнения: на полюсе ассоциации, присоединенному к «целому», т. е., в данном случае, к классу B, изображается ромб.

АССОЦИАЦИИ

Например, на рис. указано, что подразделение является частью компании (агрегация).

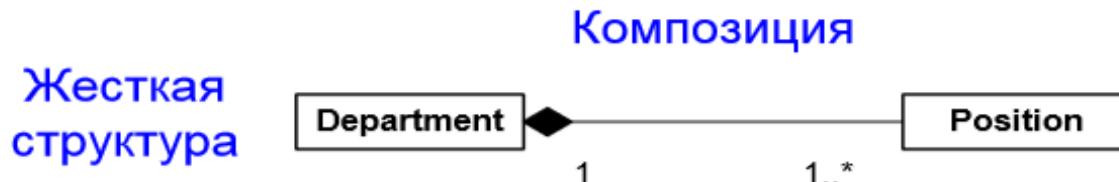


При этом никаких *дополнительных ограничений не накладывается*: объект класса А (часть) может быть связан отношениями агрегации с другими объектами (т. е. участвовать в нескольких агрегациях), создаваться и уничтожаться независимо от объекта класса В (целого).

АССОЦИАЦИИ

Композиция накладывает более сильные ограничения: композиционно часть может входить только в одно целое, часть существует только пока существует целое и прекращает свое существование вместе с целым.

Графически отношение композиции отображается закрашенным ромбом.



Жесткая
структура

Мягкая
структура

Агрегация



Приведены два возможных взгляда на отношения между подразделениями и должностями в информационной системе отдела кадров. В первом случае (вверху), мы считаем, что в организации принята жесткая («армейская») структура: каждая должность входит ровно в одно подразделение, в каждом подразделении есть по меньшей мере одна должность (начальник). Во втором случае (внизу) структура организации более аморфна: возможны «висящие в воздухе» должности, бывают «пустые» подразделения и т. д.

АССОЦИАЦИИ

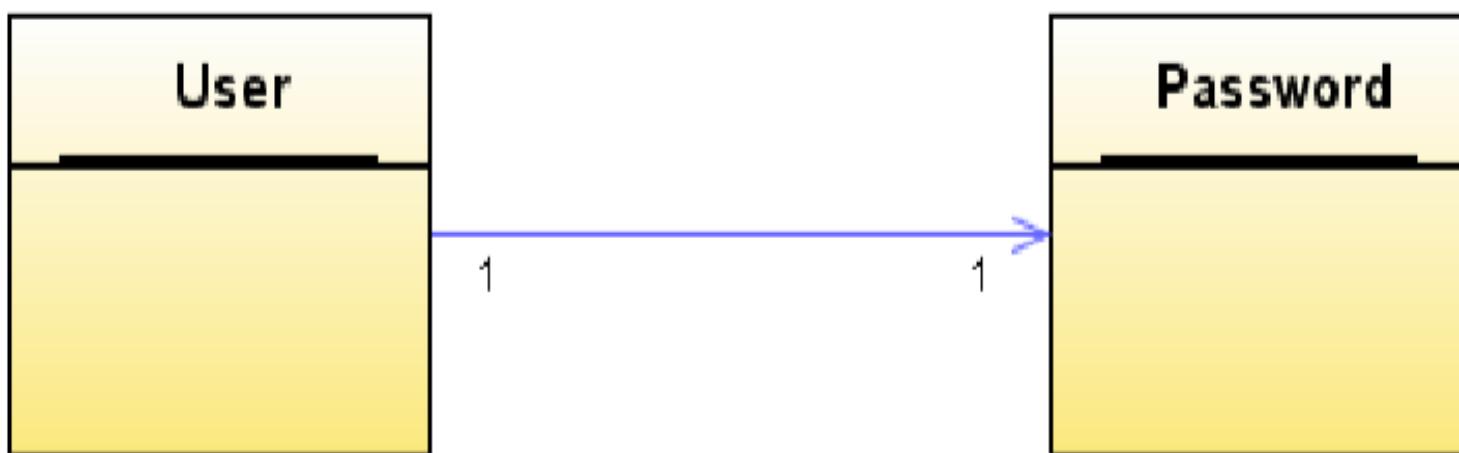
Направление навигации полюса ассоциации – это свойство полюса, имеющее значение типа Boolean, и определяющее, можно ли получить с помощью данной ассоциации доступ к объектам класса, присоединенному к данному полюсу ассоциации. По умолчанию это свойство имеет значение true, т. е. доступ возможен.

Если все полюса ассоциации (обычно их два) обеспечивают доступ, то это никак не отражается на диаграмме (потому, что данный случай наиболее распространенный и предполагается по умолчанию).

Если же навигация через некоторые полюса возможна, а через другие нет, то те полюса, через которые навигация возможна, отмечаются стрелками на концах линии ассоциации. Таким образом, если никаких стрелок не изображается, то это означает, что подразумеваются стрелки во всех возможных направлениях. Если же некоторые стрелки присутствуют, то это означает, что доступ возможен только в направлениях, указанных стрелками.

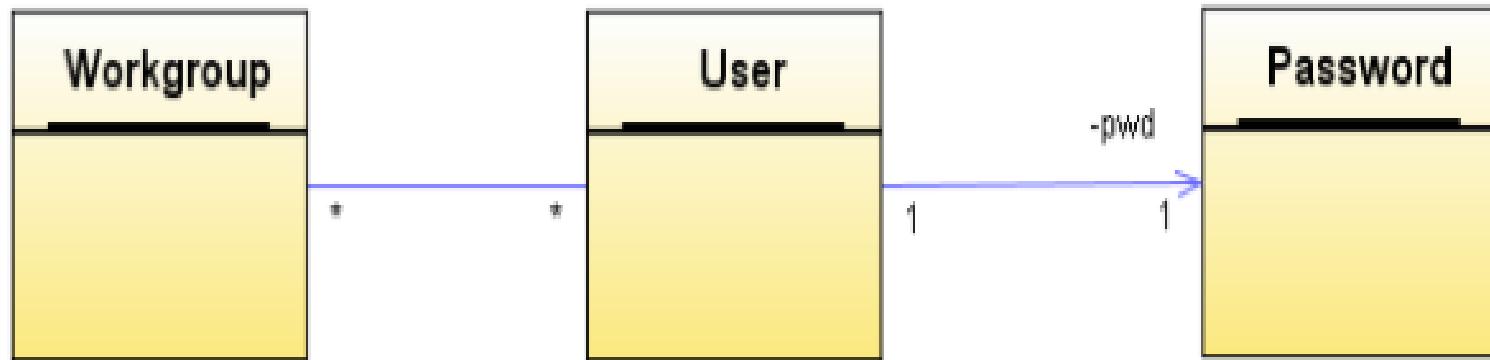
АССОЦИАЦИИ

Имеются два класса: **User** (содержит информацию о пользователе) и **Password** (содержит пароль — информацию, необходимую для аутентификации пользователя). Мы хотим отразить в модели следующую ситуацию: имеется взаимно однозначное соответствие между пользователями и паролями, зная пользователя можно получить доступ к его паролю, но обратное неверно: по имеющемуся паролю нельзя определить, кому он принадлежит.



АССОЦИАЦИИ

Видимость полюса ассоциации — это указание того, является ли классификатор присоединенный к данному полюсу ассоциации, видимым для других классификаторов вдоль данной ассоциации, помимо тех классификаторов, которые присоединены к другим полюсам ассоциации.



АССОЦИАЦИИ

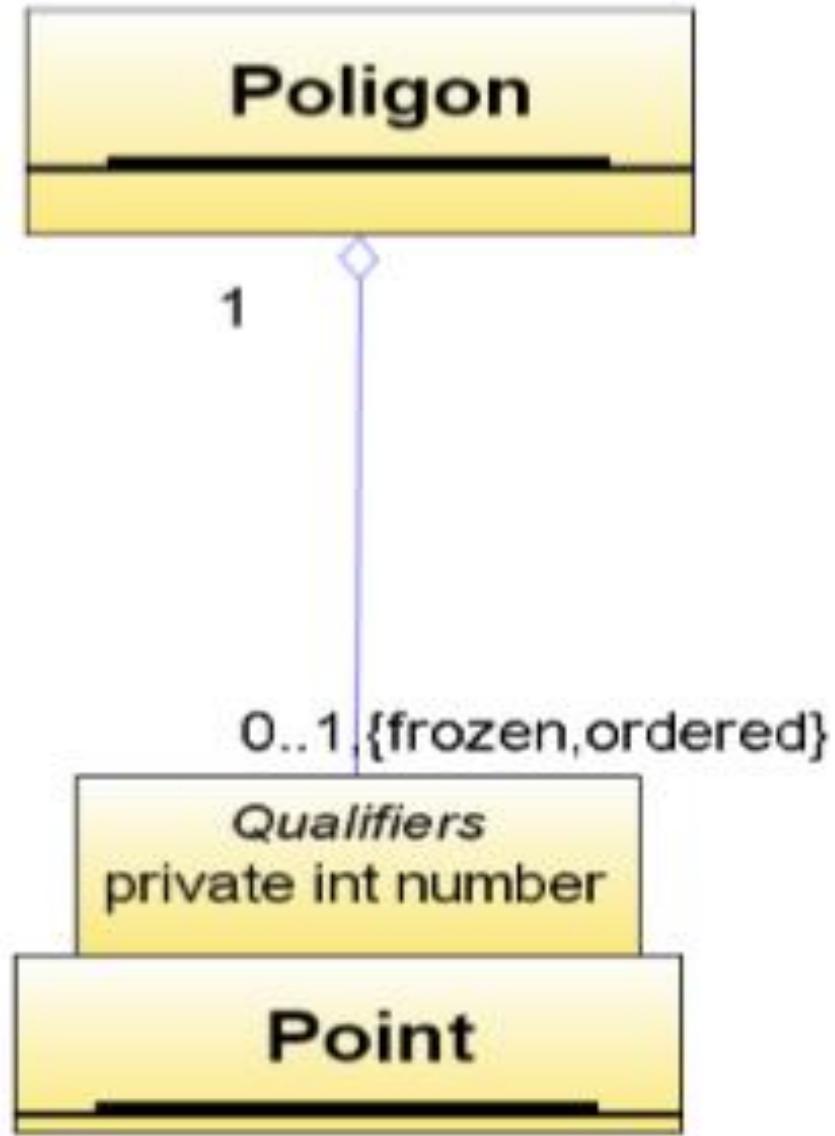
Квалификатор полюса ассоциации – это атрибут (или несколько атрибутов) ассоциации, значение которого (которых) позволяет выделить один (или несколько) объектов класса, присоединенного к данному полюсу ассоциации.

Квалификатор изображается в виде небольшого прямоугольника на полюсе ассоциации, примыкающего к прямоугольнику класса. Внутри этого прямоугольника (или рядом с ним) указываются имена и, возможно, типы атрибутов квалификатора. Описание квалифицирующего атрибута ассоциации имеет такой же синтаксис, что и описание обычного атрибута класса, только оно не может содержать начального значения.

Квалификатор может присутствовать только на том полюсе ассоциации, который имеет кратность "много", поэтому, если на полюсе ассоциации с квалификатором задана кратность, то она указывает не допустимую мощность множества объектов, присоединенных к полюсу связи, а допустимую мощность того подмножества, которое определяется при задании значений атрибутов квалификатора.

АССОЦИАЦИИ

Фрагмент модели для примера с многоугольниками, в котором использован квалификатор (в данном случае с именем number).



АССОЦИАЦИИ

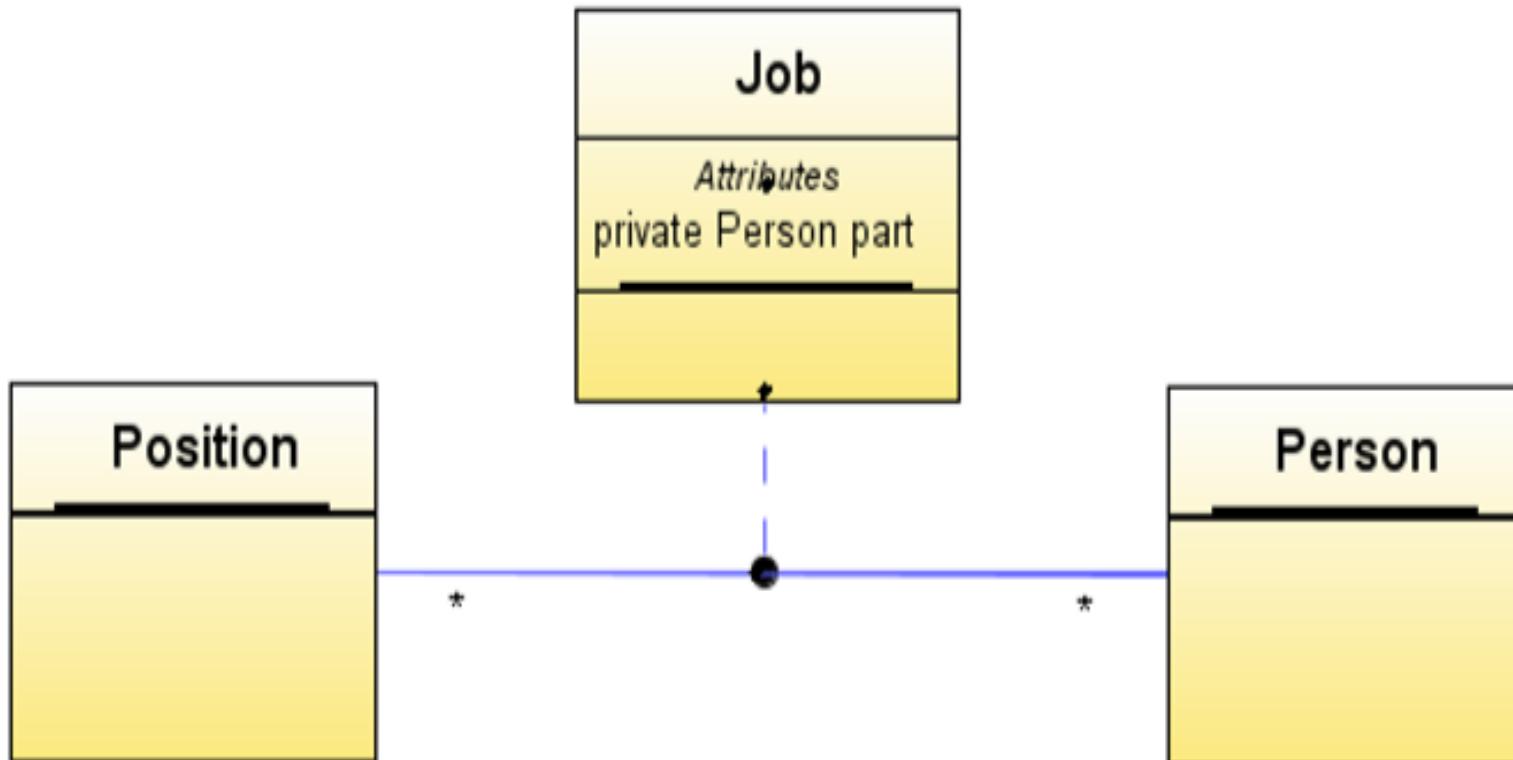
Ассоциация имеет экземпляры (связи), стало быть, является классификатором и может обладать соответствующими свойствами и составляющими классификатора.

В распространенных случаях ассоциации не обладают никакими собственными составляющими. Грубо говоря, ассоциация между классами А и В — это просто множество пар (a,b), где a — объект класса А, а b — объект класса В.

Подчеркнем еще раз, что это именно множество: двух одинаковых пар (a,b) быть не может. Однако возможны и более сложные ситуации, когда ассоциация имеет собственные атрибуты (и даже операции), значения которых хранятся в экземплярах ассоциации — связях.

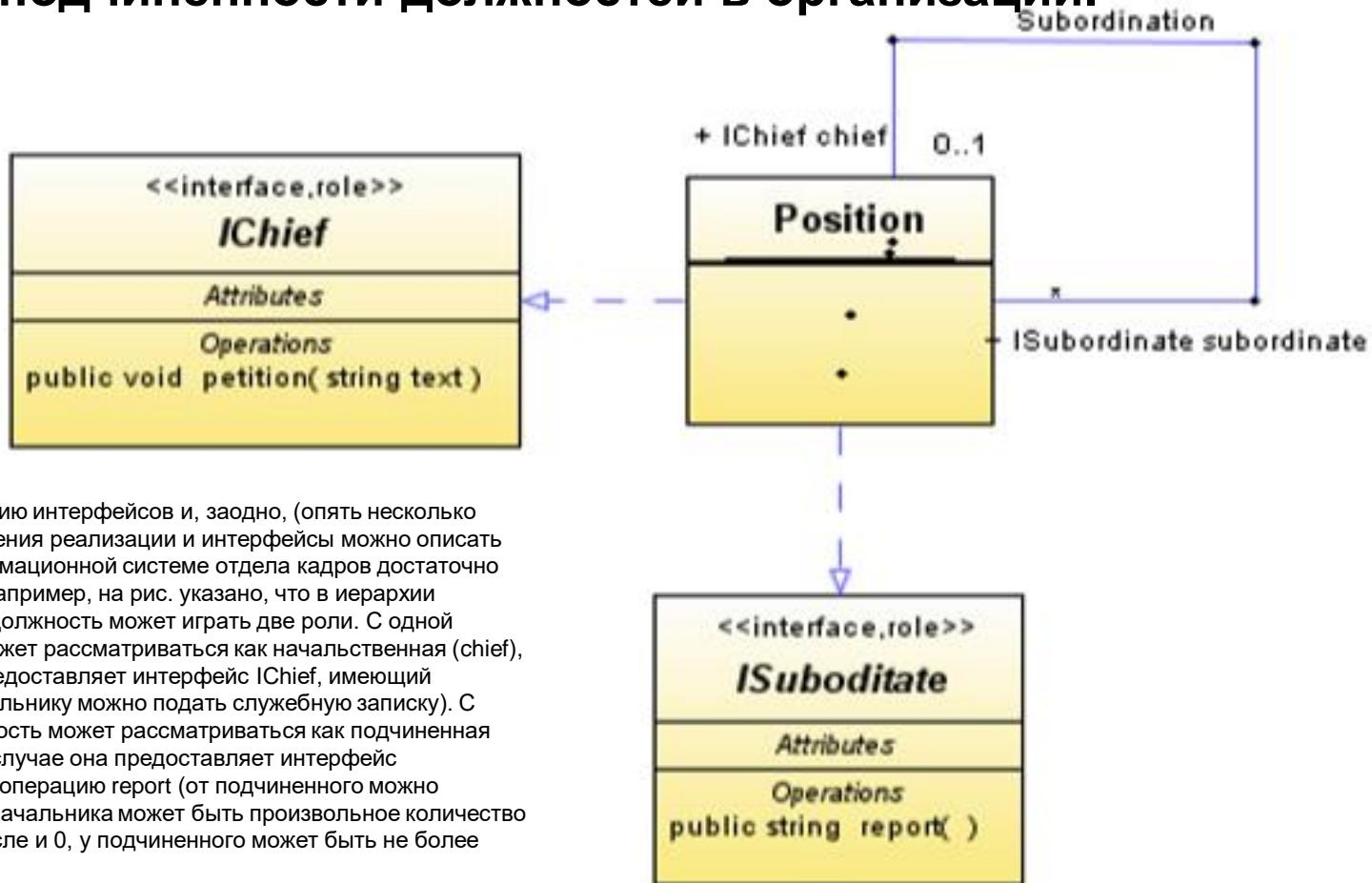
В таком случае применяется специальный элемент моделирования — класс ассоциации. Класс ассоциации — это сущность, которая имеет как свойства класса, так и свойства ассоциации. Класс ассоциации изображается в виде символа класса, присоединенного пунктирной линией к линии ассоциации.

АССОЦИАЦИИ



АССОЦИАЦИИ

На этом рисунке изображена ассоциация класса **Position** с самим собой. Эта ассоциация призвана отразить наличие иерархии подчиненности должностей в организации.



Используя спецификацию интерфейсов и, заодно, (опять несколько забегая вперед) отношения реализации и интерфейсы можно описать субординацию в информационной системе отдела кадров достаточно лаконично, но точно. Например, на рис. указано, что в иерархии субординации каждая должность может играть две роли. С одной стороны, должность может рассматриваться как начальственная (chief), и в этом случае она предоставляет интерфейс **IChief**, имеющий операцию **petition** (начальнику можно подать служебную записку). С другой стороны, должность может рассматриваться как подчиненная (subordinate), и в этом случае она предоставляет интерфейс **ISubordinate**, имеющий операцию **report** (от подчиненного можно потребовать отчет). У начальника может быть произвольное количество подчиненных, в том числе и 0, у подчиненного может быть не более одного начальника.

ОПЕРАЦИИ

Операция — это описание способа выполнить какие-то действия с объектом: изменить значения его атрибутов, вычислить новое значение по информации хранящейся в объекте и т. д. Выполнение действий, определяемых операцией, инициируется вызовом операции. При выполнении операция может, в свою очередь, вызывать операции этого и других классов. Описания операций класса перечисляются в разделе операций и имеют следующий синтаксис.

видимость ИМЯ (параметры): тип {свойства}

Здесь слово параметры обозначает последовательность описаний параметров операции, каждое из которых имеет вид следующий формат.

направление ПАРАМЕТР : тип = значение

ОПЕРАЦИИ

Видимость ИМЯ (параметры): тип {свойства}

Видимость, как обычно, обозначается с помощью знаков +, -, #.

Видимость можно обозначать с помощью ключевых слов private, public, protected.

Подчеркивание имени означает, что область действия операции — класс, а не объект. Например, конструкторы имеют область действия класс.

Курсивное написание имени означает, что операция абстрактная, т. е. в данном классе ее реализация не задана и должна быть задана в подклассах данного класса. После имени в скобках может быть указан список описаний параметров.

Описания параметров в списке разделяются запятой. Для каждого параметра обязательно указывается имя, а также могут быть указаны направление передачи параметра, его тип и значение аргумента по умолчанию.

ОПЕРАЦИИ

направление ПАРАМЕТР : тип = значение

Направление передачи параметра в UML описывает семантическое назначение параметров, не конкретизируя конкретный механизм передачи. Как именно следует трактовать указанные в модели направления передачи параметров зависит от используемой системы

**Ключевое
слово**

Назначение параметра

In

Входной параметр — аргумент должен быть значением, которое используется в операции, но не изменяется

Out

Выходной параметр — аргумент должен быть хранилищем, в которое операция помещает значение

Inout

Входной и выходной параметр — аргумент должен быть хранилищем, содержащим значение. Операция использует переданное значение аргумента и помещает в хранилище результат

Return

Значение, возвращаемое операцией. Никакого аргумента не требуется

ОПЕРАЦИИ

Видимость ИМЯ (параметры): тип {свойства}

Типом параметра операции, равно как и тип возвращаемого операцией значения может быть любой встроенный тип или определенный в модели класс. Все вместе (имя операции, параметры и тип результата) обычно называют сигнатурой.

Операция имеет два важных свойства, которые указываются в списке свойств как именованные значения.

Во-первых, это `concurrency` – свойство, определяющее семантику одновременного (параллельного) вызова данной операции.

Во-вторых, операция имеет свойство `isQuery`, значение которого указывает, обладает ли операция побочным эффектом. Если значение данного свойства `true`, то выполнение операции не меняет состояния системы – операция только вычисляет значения, возвращаемые в точку вызова. В противном случае, т. е. при значение `false`, операция меняет состояние системы: присваивает новые значения атрибутам, создает или уничтожает объекты и т. п.

ЗНАЧЕНИЯ СВОЙСТВА CONCURRENCY

Значение	Описание
sequential	Операция не допускает параллельного вызова (не является повторно-входимой). Если параллельный вызов происходит, то дальнейшее поведение системы не определено.
Guarded	Параллельные вызовы допускаются, но только один из них выполняется — остальные блокируются и их выполнение задерживается до тех пор, пока не завершится выполнение данного вызова.
concurrent	Операция допускает произвольное число параллельных вызовов и гарантирует правильность своего выполнения. Такие операции называются повторно-входимыми (reenterable).

ПРИМЕРЫ ОПИСАНИЯ ОПЕРАЦИЙ

Пример

move

+move(in from, in to)

+move(in from : Dpt, in to : Dpt)

+getName() : String {isQuery}

+setPwd(in pwd : String = "password")

Пояснение

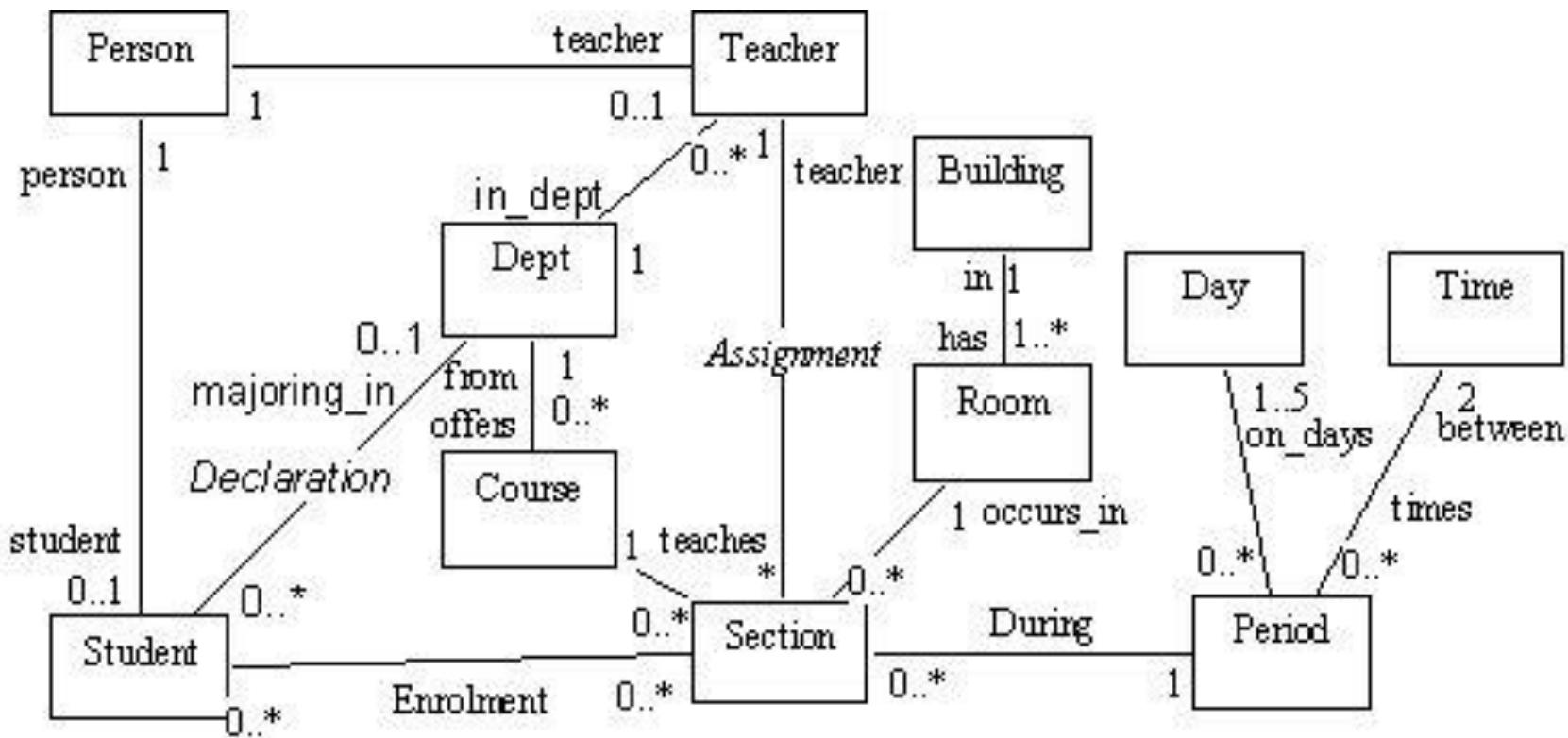
Минимальное возможное описание — указано только имя операции

Указаны видимость операции, направления передачи и имена параметров

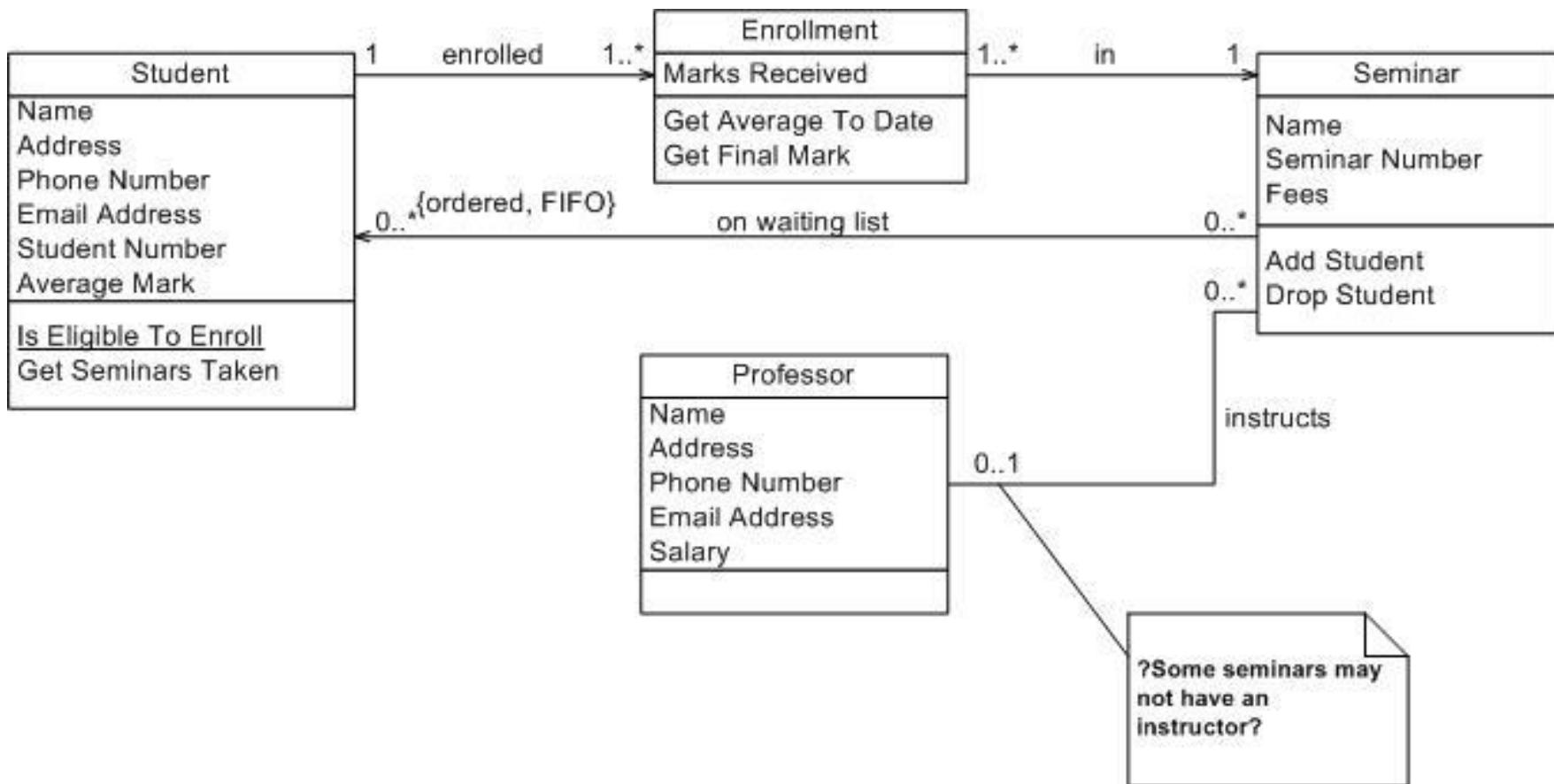
Подробное описание сигнатуры: указаны видимость операции, направления передачи, имена и типы параметров

Функция, возвращающая значение атрибута и не имеющая побочных эффектов

Процедура, для которой указано значение аргумента по умолчанию



Пример Диаграммы классов. Система классов для университета.



Пример Диаграммы классов

ИНТЕРФЕЙСЫ

Интерфейс – это именованный набор абстрактных операций. Другими словами, интерфейс – это абстрактный класс, в котором нет атрибутов и все операции абстрактны.

Поскольку интерфейс – это абстрактный класс, он не может иметь непосредственных экземпляров. Между интерфейсами и другими классификаторами, в частности классами, на диаграмме классов применяются два отношения:

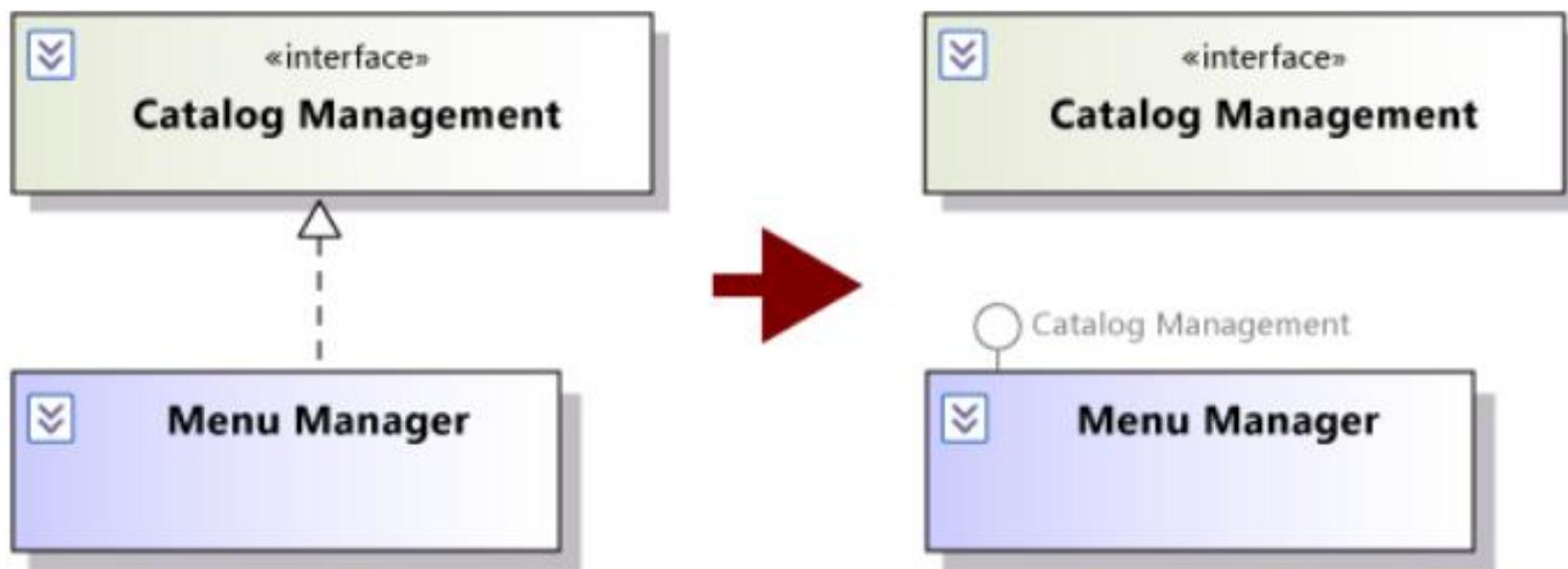
- классификатор (в частности, класс) использует интерфейс – это показывается с помощью зависимости со стереотипом «call»;
- классификатор (в частности, класс) реализует интерфейс – это показывается с помощью отношения реализации.

Никаких ограничений на использование отношения реализации не накладывается: класс может реализовывать много интерфейсов, и наоборот, интерфейс может быть реализован многими классами. Нет ограничений и на использование зависимостей со стереотипом «call» – класс может вызывать любые операции любых видимых интерфейсов.

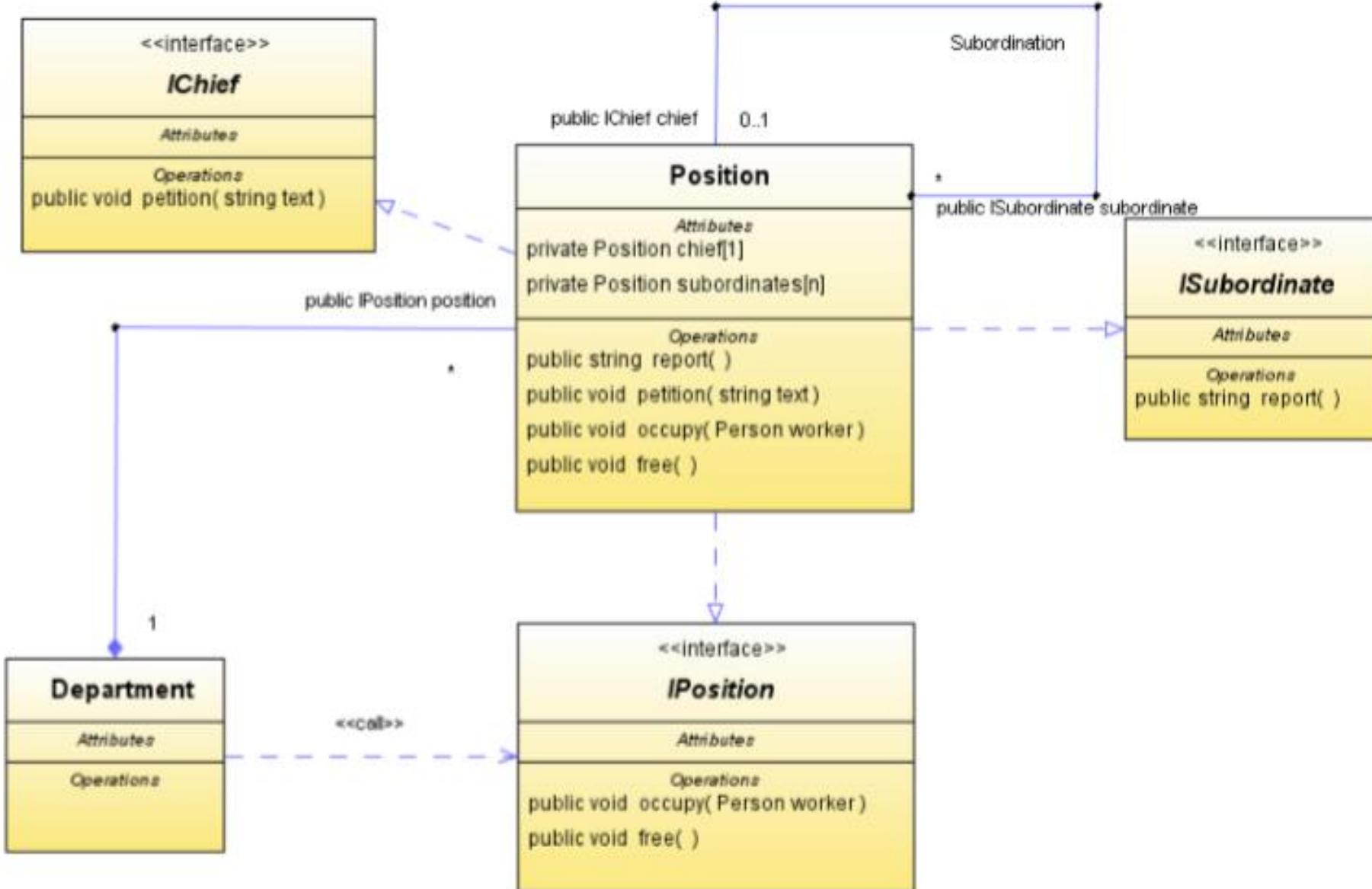
РЕАЛИЗАЦИЯ

Реализация означает, что класс реализует атрибуты и операции, заданные в интерфейсе.

Интерфейс находится на окончании соединителя с наконечником стрелки.



ИНТЕРФЕЙСЫ



ШАБЛОНЫ

Шаблон – это класс с параметрами. Параметром может быть любой элемент описания класса – тип составляющей, кратность атрибута и т. д. На диаграмме шаблон изображается с помощью прямоугольника класса, к которому в правом верхнем углу присоединен пунктирный прямоугольник с параметрами шаблона. Описания параметров перечисляются в этом прямоугольнике через запятую. Описание каждого параметра имеет вид:

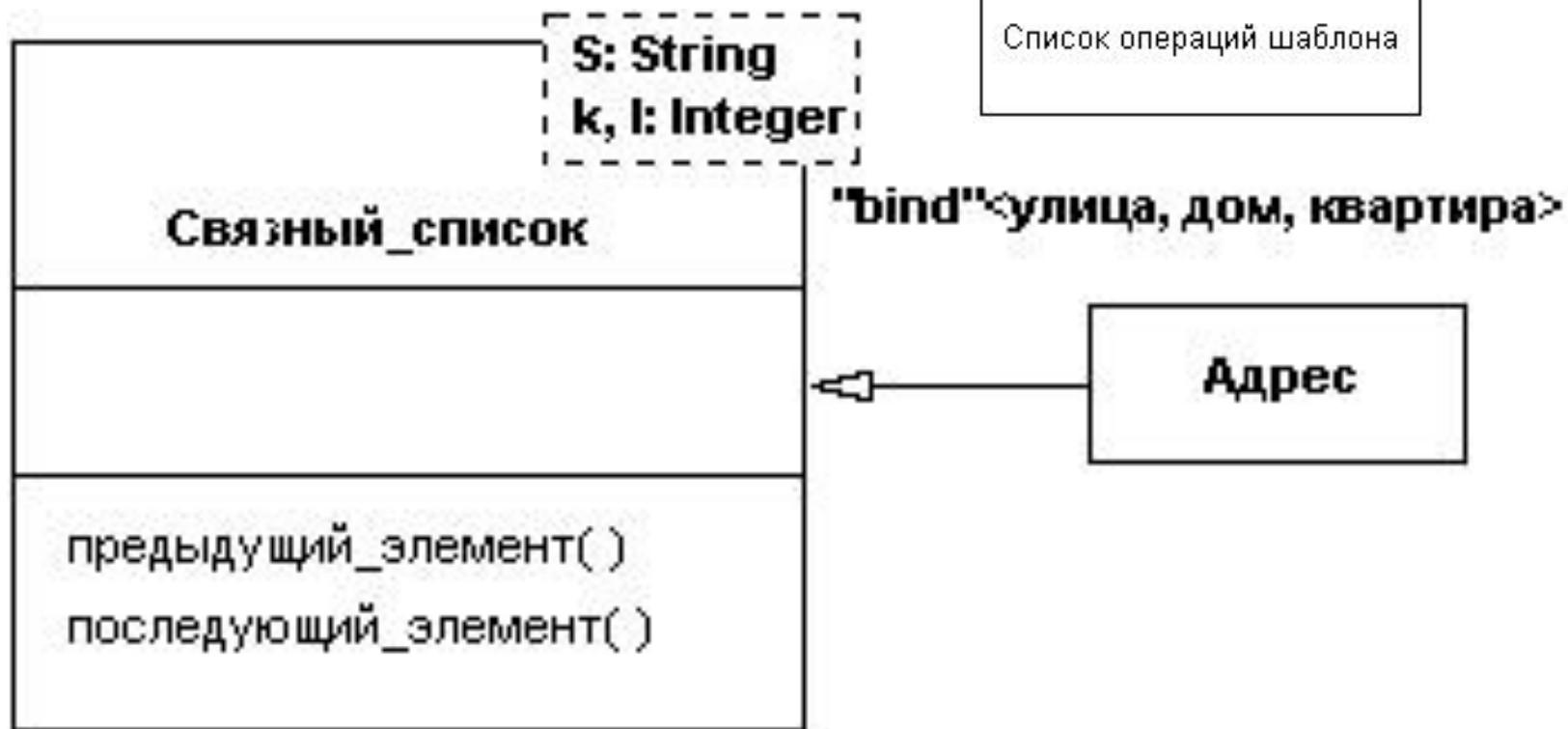
ИМЯ : ТИП

Сам по себе шаблон не может непосредственно использоваться в модели. Для того, чтобы на основе шаблона получить конкретный класс, который может использоваться в модели, нужно указать явные значения аргументов. Такое указание называется связыванием.

В UML применяются два способа связывания:

- явное связывание — зависимость со стереотипом «bind», в которой указаны значения аргументов;
- неявное связывание — определение класса, имя которого имеет формат **имя_шаблона < аргументы >**

ШАБЛОНЫ



В данном примере отмечен тот факт, что класс «Адрес» может быть получен из шаблона Связный_список на основе актуализации формальных параметров «S, k, l» фактическими атрибутами «улица, дом, квартира».

СПАСИБО ЗА ВНИМАНИЕ!

ПРОЕКТИРОВАНИЕ ПРОГРАММНЫХ СИСТЕМ

УНИФИЦИРОВАННЫЙ ЯЗЫК МОДЕЛИРОВАНИЯ UML

ЛЕКЦИЯ 5

UML (ПРОДОЛЖЕНИЕ)

ДИАГРАММА ОБЪЕКТОВ

КЛАССИФИКАЦИЯ ДИАГРАММ

Диаграммы поведения:

**Диаграммы вариантов
использования**

Диаграммы деятельности

Диаграммы состояний

**Диаграммы
взаимодействия:**

- Диаграммы последовательности
- Диаграммы коммуникации
- Диаграммы синхронизации (временные диаграммы)
- Обзорные диаграммы взаимодействия

Структурные диаграммы:

1. **Диаграммы классов**
2. **Диаграммы объектов**
3. **Диаграммы пакетов**
4. **Диаграммы компонентов**
5. **Диаграммы составной структуры**
6. **Диаграммы размещения**
7. **Диаграммы кооперации**
8. **Диаграммы профиля**

ДИАГРАММА ОБЪЕКТОВ

Диаграмма объектов – это частный случай диаграммы классов. Диаграммы объектов имеют вспомогательный характер – по сути это примеры, показывающие, какие имеются объекты и связи между ними в некоторый конкретный момент функционирования системы. На диаграмме объектов применяют один основной тип сущностей: объекты (экземпляры классов), между которыми указываются конкретные связи (экземпляры ассоциаций).

ДИАГРАММА ОБЪЕКТОВ

Имена объектов

Каждый отдельный объект представлен, например, прямоугольной формой, которая предоставляет имя через объект, а также подчеркнутый класс и разделяемый с помощью двоеточия.

ДИАГРАММА ОБЪЕКТОВ

Связи

Ссылки часто встречаются, связанные с отношениями. Вы можете нарисовать ссылку при использовании линий, примененных к диаграммам классов.

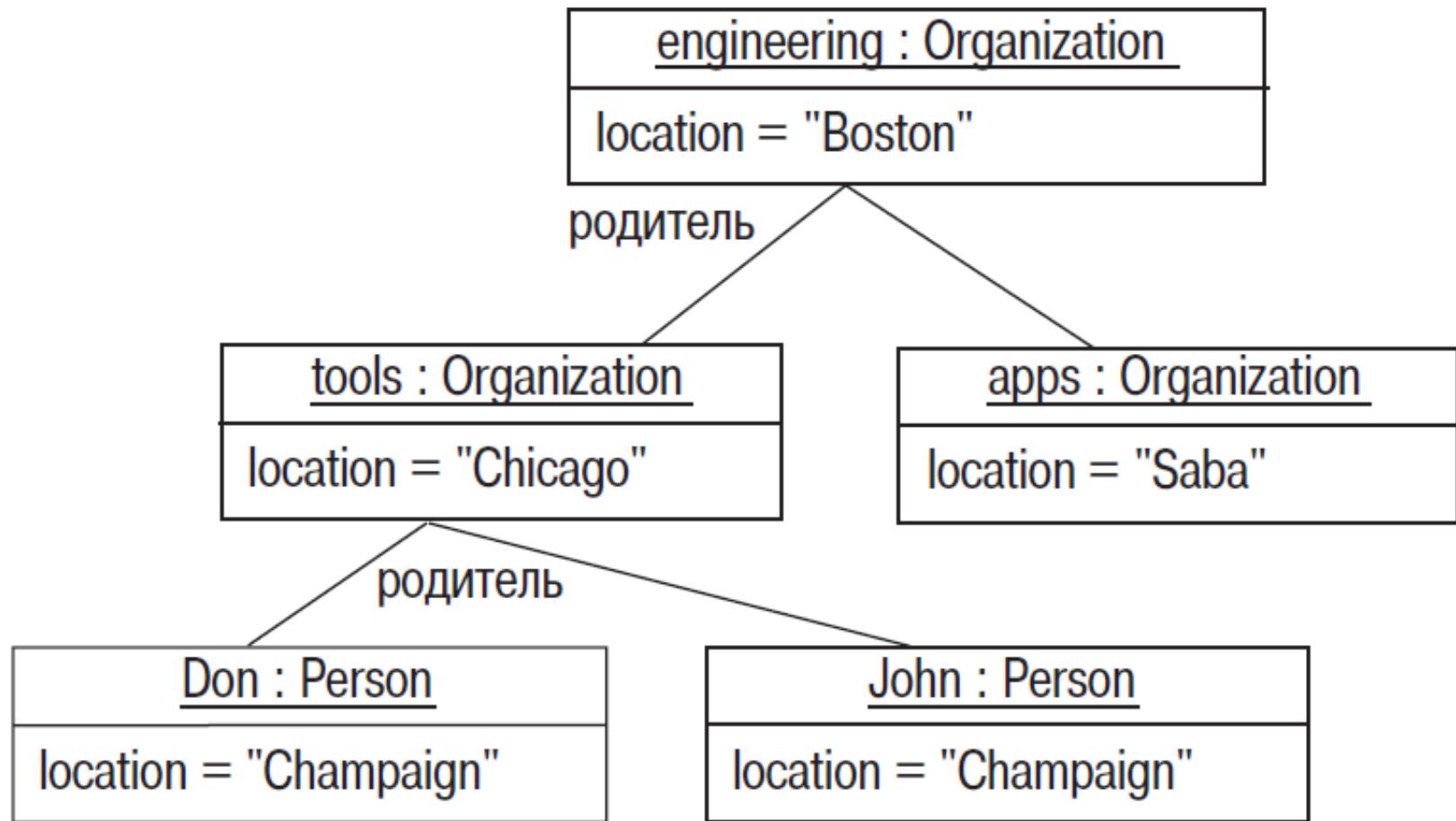


ДИАГРАММА ПАКЕТОВ (PACKAGE DIAGRAM)

ПАКЕТЫ (PACKAGES)

Визуализация, спецификация, конструирование и документирование больших систем предполагает работу с потенциально большим количеством классов, интерфейсов, узлов, компонентов, диаграмм и других элементов. Масштабируя такие системы, вы столкнетесь с необходимостью организовывать эти сущности в более крупные блоки.

В языке UML для организации моделирующих элементов в группы применяют пакеты (Packages).

ПАКЕТЫ (PACKAGES)

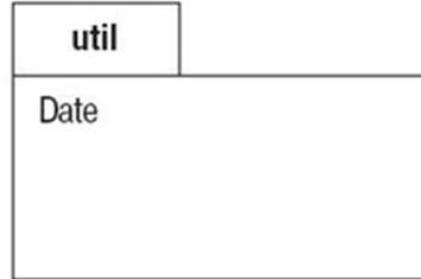
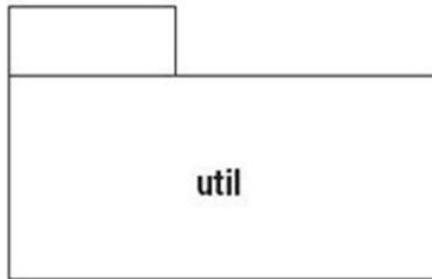
Пакет – это способ организации элементов модели в более крупные блоки, которыми впоследствии позволяет манипулировать как единым целым. Можно управлять видимостью входящих в пакет сущностей, так что некоторые будут видимы извне, а другие – нет. Кроме того, с помощью пакетов можно представлять различные виды архитектуры системы (различные виды разбиения системы на более мелкие части).

ПАКЕТЫ (PACKAGES)

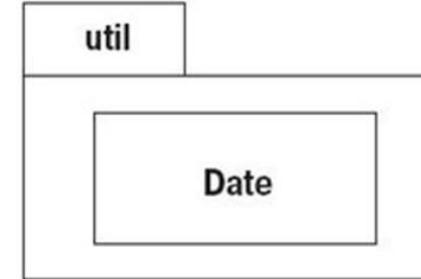
В языке UML организующие модель блоки называют пакетами. Пакет является универсальным механизмом организации элементов в группы, благодаря которому понимание модели упрощается. Пакеты позволяют также контролировать доступ к своему содержимому, что облегчает работу со стыковочными узлами в архитектуре системы.

Такая нотация позволяет визуализировать группы элементов, с которыми можно обращаться как с единым целым, контролируя при этом видимость и возможность доступа к отдельным элементам.

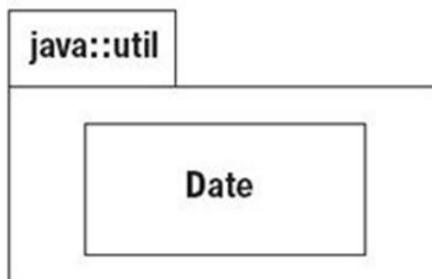
ГРАФИЧЕСКОЕ ПРЕДСТАВЛЕНИЕ ПАКЕТОВ В ЯЗЫКЕ UML:



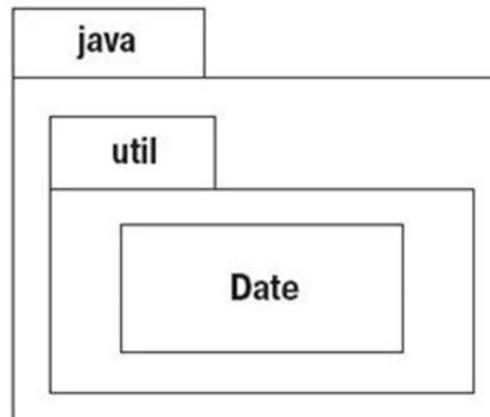
Содержимое, перечисленное в прямоугольнике



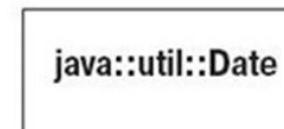
Содержимое в виде диаграммы в прямоугольнике



Полностью определенное имя пакета



Вложенные пакеты



Полностью определенное имя класса

... Способы изображения пакетов на диаграммах

ИМЕНА ПАКЕТОВ

У каждого пакета должно быть имя, отличающее его от других пакетов. Имя – это текстовая строка.

Взятое само по себе, оно называется простым.

К составному имени спереди добавлено имя объемлющего его пакета, если таковой существует.

Имя пакета должно быть уникальным в объемлющем пакете. Обычно при изображении компонента указывают только его имя, но, как и в случае с классами, вы можете дополнять пакеты помеченными значениями или дополнительными разделами, чтобы показать детали.



простые имена



расширенные пакеты

имя объемлющего пакета



ЭЛЕМЕНТЫ, ПРИНАДЛЕЖАЩИЕ ПАКЕТУ

Пакет может владеть другими элементами, в том числе классами, интерфейсами, компонентами, узлами, кооперациями, диаграммами и даже прочими пакетами.

Владение – это композитное отношение, означающее, что элемент объявлен внутри пакета.

Если пакет удаляется, то уничтожается и принадлежащий ему элемент.

Элемент может принадлежать только одному пакету.

ЭЛЕМЕНТЫ, ПРИНАДЛЕЖАЩИЕ ПАКЕТУ

Пакет определяет свое пространство имен; это значит, что элементы одного вида должны иметь имена, уникальные в контексте объемлющего пакета. Например, в одном пакете не может быть двух классов Queue, но может быть один класс Queue в пакете P1, а другой - в пакете P2. P1 :: Queue и P2 :: Queue имеют разные составные имена и поэтому являются различными классами.

Элементы различного вида могут иметь одинаковые имена в пределах пакета. Так, допустимо наличие класса Timer и компонента Timer. Однако на практике, во избежание недоразумений, лучше всем элементам давать уникальные имена в пакете.

ЭЛЕМЕНТЫ, ПРИНАДЛЕЖАЩИЕ ПАКЕТУ

Пакету могут принадлежать другие пакеты, а значит, позволительно осуществить иерархическую декомпозицию модели. Например, может существовать класс Camera, принадлежащий пакету Vision, который, в свою очередь, содержится в пакете Sensors.

Составное имя этого класса будет Sensors :: Vision :: Camera.

Лучше, однако, избегать слишком глубокой вложенности пакетов – два-три уровня являются пределом.

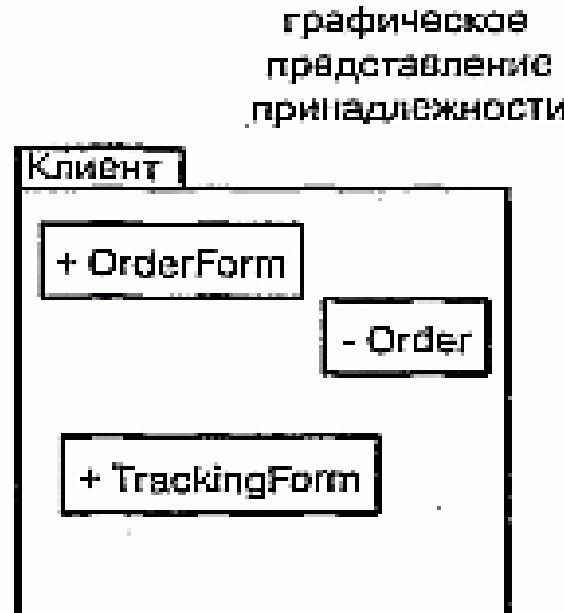
ЭЛЕМЕНТЫ, ПРИНАДЛЕЖАЩИЕ ПАКЕТУ

Пакеты позволяют контролировать элементы системы, даже если они разрабатываются в разном темпе.

Содержание пакета можно представить графически или в текстовом виде. Обратите внимание, что если вы изображаете принадлежащие пакету элементы, то имя пакета пишется внутри закладки



текстовое
представление
принадлежности



ВИДИМОСТЬ

Видимость принадлежащих пакету элементов можно контролировать точно так же, как видимость атрибутов и операций класса. По умолчанию такие элементы являются открытыми, то есть видимы для всех элементов, содержащихся в любом пакете, импортирующем данный. Защищенные элементы видимы только для потомков, а закрытые вообще невидимы вне своего пакета.

ВИДИМОСТЬ

Видимость принадлежащих пакету элементов обозначается с помощью символа видимости перед именем этого элемента. Для открытых элементов используется символ + (плюс), как и в случае с OrderForm. Все открытые части пакета составляют его интерфейс.

Аналогично классам для имен защищенных элементов используют символ # (диез), а для закрытых добавляют символ - (минус). Напомним, что защищенные элементы будут видны только для пакетов, наследующих данному, а закрытые вообще не видны вне пакета, в котором объявлены.

Примечание: Пакеты, связанные с некоторым пакетом отношениями зависимости со стереотипом friend, могут "видеть" все элементы данного пакета, независимо от объявленной для них видимости.

ВИДИМОСТЬ

Допустим, что класс А расположен в одном пакете, а класс В – в другом, причем оба пакета равноправны. Допустим также, что как А, так и В объявлены открытыми частями в своих пакетах.

Хотя оба класса объявлены открытыми, свободный доступ одного из них к другому невозможен, ибо границы пакетов непрозрачны. Однако если пакет, содержащий класс А, импортирует пакет-владелец класса В, то А сможет "видеть" В, хотя В по-прежнему не будет "видеть" А.

Импорт дает элементам одного пакета односторонний доступ к элементам другого. На языке UML отношения импорта моделируют как зависимости, дополненные стереотипом *import*.

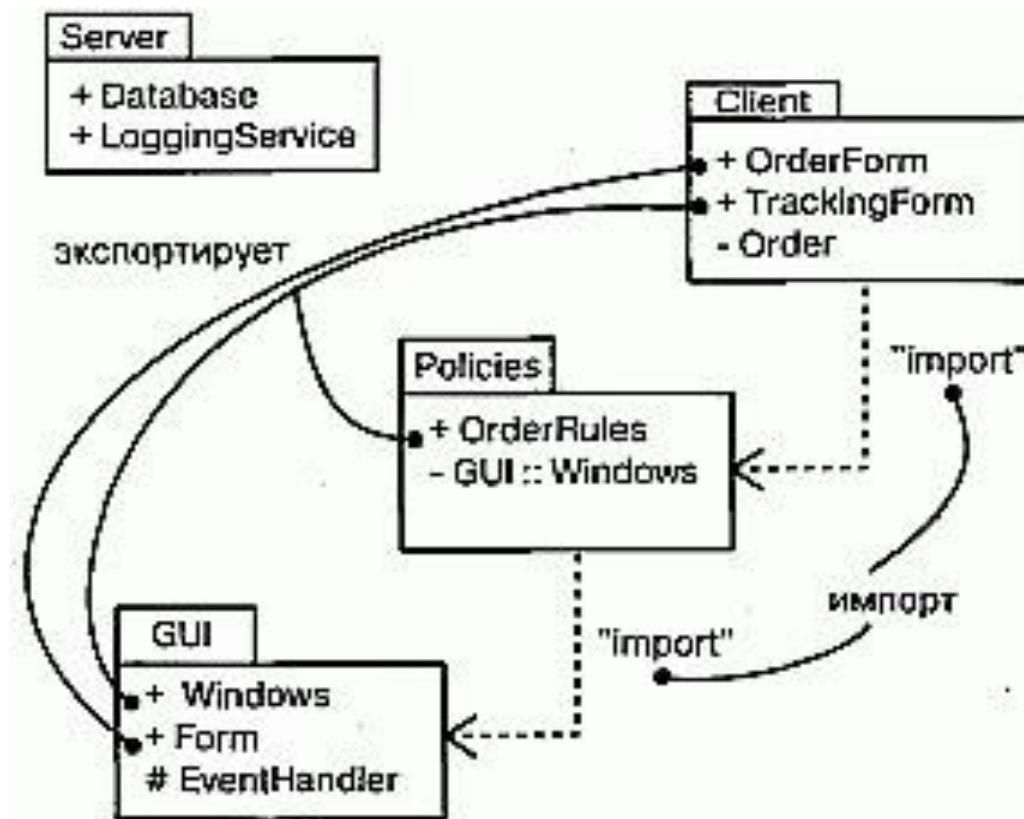
Упаковывая абстракции в семантически осмыслиенные блоки и контролируя доступ к ним с помощью импорта, вы можете управлять сложностью систем, насчитывающих множество абстракций.

ВИДИМОСТЬ

Открытые элементы пакета называются экспортируемыми.

Экспортируемые элементы будут видны только тем пакетам, которые явно импортируют данный.

Зависимости импорта и доступа не являются транзитивными.

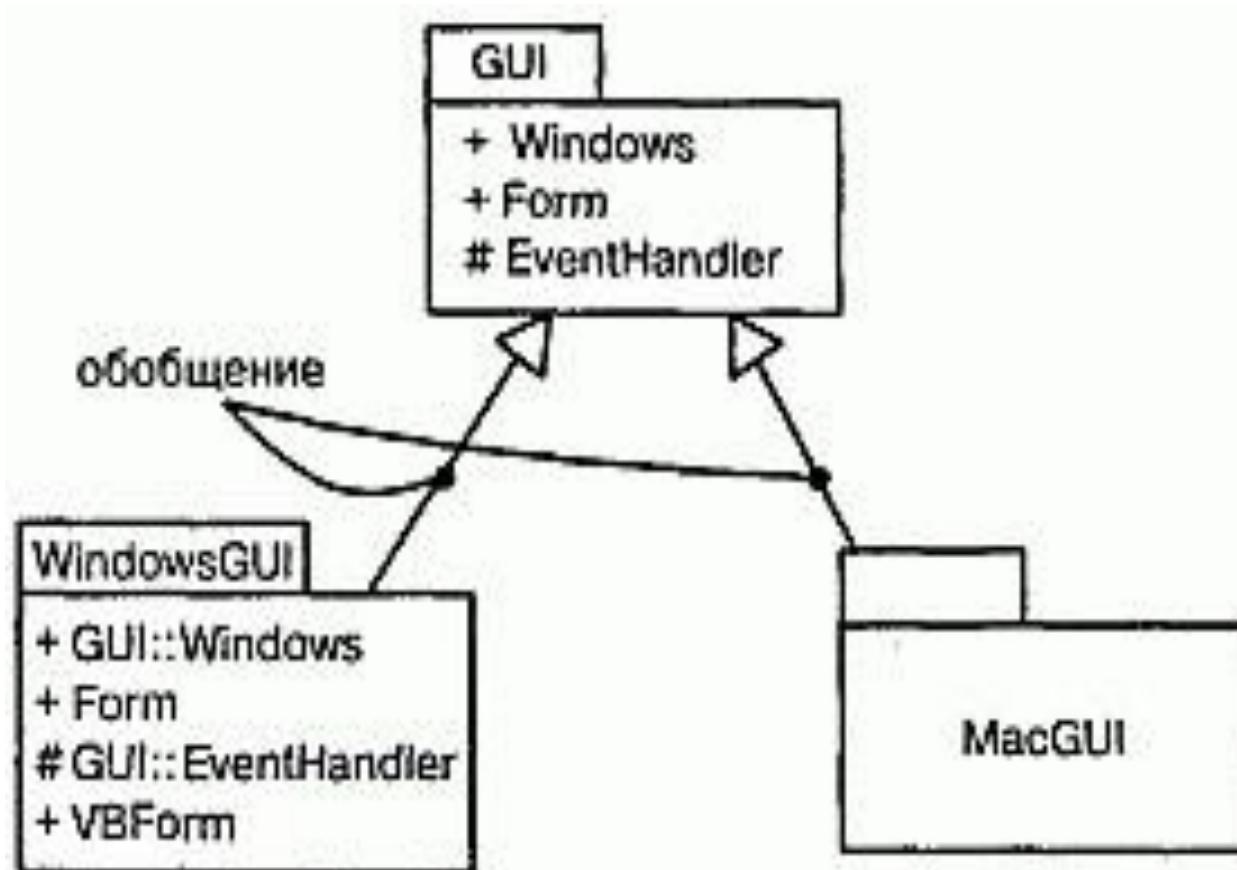


ОБОБЩЕНИЯ

Между пакетами определены два типа отношений: зависимости импорта и доступа, применяемые для импорта в пакет элементов, экспортируемых другим пакетом, и обобщения, используемые для спецификации семейств пакетов.

Отношения обобщения между пакетами очень похожи на отношения обобщения между классами.

ОБОБЩЕНИЯ



СТАНДАРТНЫЕ ЭЛЕМЕНТЫ

К пакетам применимы все механизмы расширения. Чаще всего используют помеченные значения для определения новых свойств (например, указания имени автора) и стереотипы для определения новых видов пакетов (например, пакетов, инкапсулирующих сервисы операционных систем).

В языке UML определены пять стандартных стереотипов, применимых к пакетам:

facade (фасад) – определяет пакет, являющийся всего лишь представлением какого-то другого пакета;

framework (каркас) – определяет пакет, состоящий в основном из образцов (паттернов);

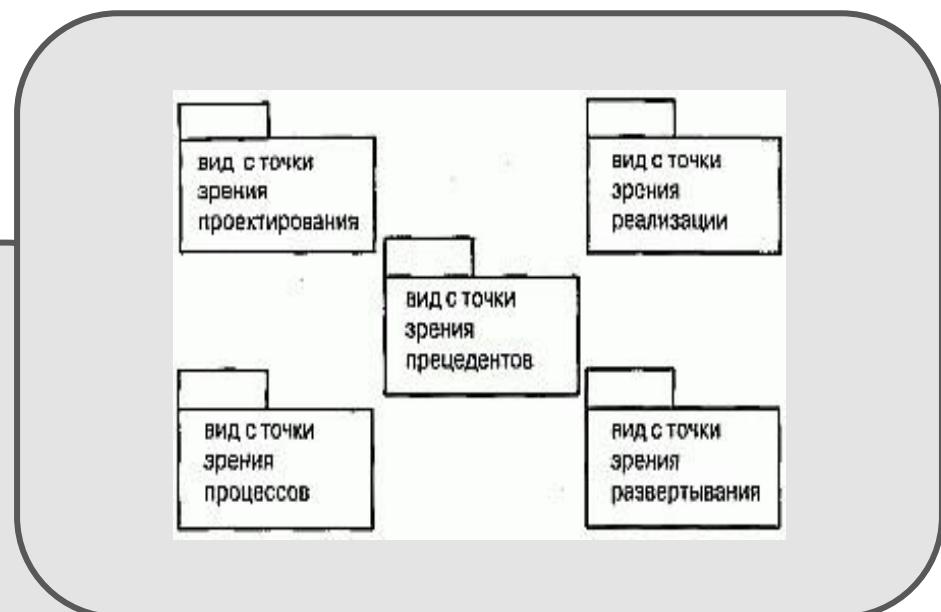
stub (стаб, заглушка) – определяет пакет, служащий заместителем открытого содержимого другого пакета;

subsystem (подсистема) – определяет пакет, представляющий независимую часть моделируемой системы;

system (система) – определяет пакет, представляющий всю моделируемую систему.

ТИПИЧНЫЕ ПРИЕМЫ МОДЕЛИРОВАНИЯ

Группы элементов Архитектурные виды



ГРУППЫ ЭЛЕМЕНТОВ

Чаще всего пакеты применяют для организации элементов моделирования в именованные группы, с которыми затем можно будет работать как с единым целым. Создавая простое приложение, можно вообще обойтись без пакетов, поскольку все ваши абстракции прекрасно разместятся в единственном пакете. В более сложных системах вы вскоре обнаружите, что многие классы, компоненты, узлы, интерфейсы и даже диаграммы естественным образом разделяются на группы. Эти группы и моделируют в виде пакетов.

ГРУППЫ ЭЛЕМЕНТОВ

Чаще всего с помощью пакетов элементы одинаковых типов организуют в группы. Например, классы и их отношения в представлении системы с точки зрения проектирования можно разбить на несколько пакетов и контролировать доступ к ним с помощью зависимостей импорта. Компоненты вида системы с точки зрения реализации допустимо организовать таким же образом.

Пакеты также применяются для группирования элементов различных типов. Например, если система создается несколькими коллективами разработчиков, расположенными в разных местах, то пакеты можно использовать для управления конфигурацией, размещая в них все классы и диаграммы, так чтобы члены разных коллективов могли независимо извлекать их из хранилища и помещать обратно.

На практике пакеты часто применяют для группирования элементов моделирования и ассоциированных с ними диаграмм.

ГРУППЫ ЭЛЕМЕНТОВ

Моделирование группы элементов производится так:

Просмотрите моделирующие элементы в некотором представлении архитектуры системы с целью поиска групп семантически или концептуально близких элементов.

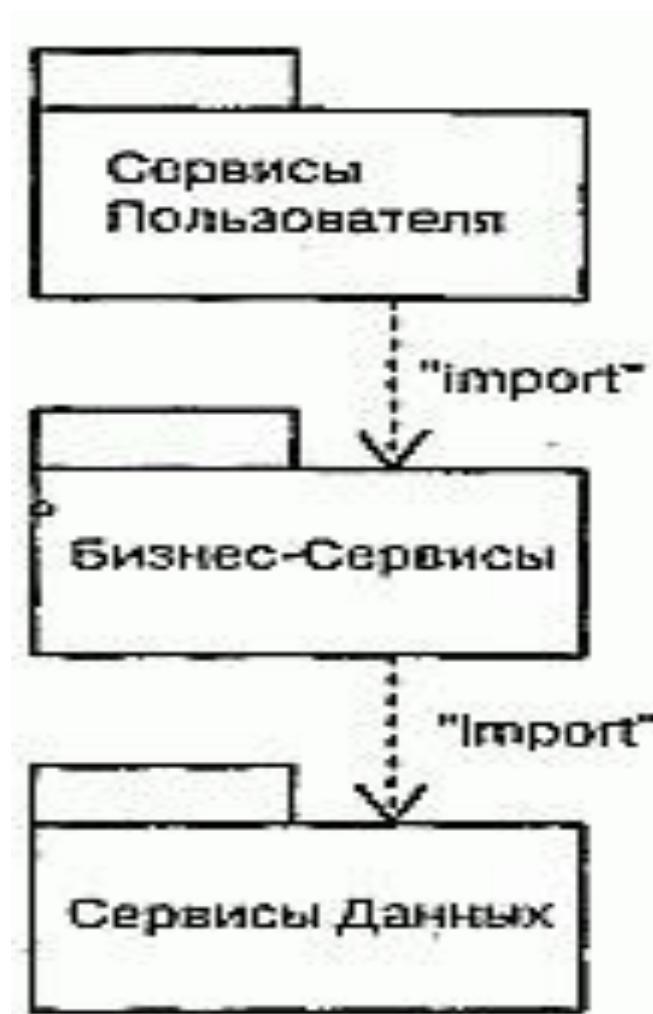
Поместите каждую такую группу в пакет.

Для каждого пакета определите, какие элементы должны быть доступны извне. Пометьте их как открытые, а остальные - как защищенные или закрытые. Если сомневаетесь, скройте элемент.

Явно соедините пакеты отношениями импорта с теми пакетами, от которых они зависят.

Если имеются в наличии семейства пакетов, соедините специализированные пакеты с общими отношениями обобщения.

ГРУППЫ ЭЛЕМЕНТОВ



АРХИТЕКТУРНЫЕ ВИДЫ

При рассмотрении архитектурных видов программных систем возникает потребность в еще более крупных блоках. Архитектурные виды тоже можно моделировать с помощью пакетов.

Видом или представлением называется проекция организации и структуры системы, в которой внимание акцентируется на одном из конкретных аспектов этой системы

Из этого определения вытекают два следствия. Во-первых, систему можно разложить на почти ортогональные пакеты, каждый из которых имеет дело с набором архитектурно значимых решений (например, можно создать виды с точки зрения проектирования, процессов, реализации, развертывания и прецедентов). Во-вторых, этим пакетам будут принадлежать все абстракции, относящиеся к данному виду. Так, все компоненты модели принадлежат пакету, который представляет вид системы с точки зрения реализации.

АРХИТЕКТУРНЫЕ ВИДЫ

Моделирование архитектурных видов осуществляется следующим образом:

Определите, какие виды важны для решения вашей проблемы. Обычно это виды с точки зрения проектирования, процессов, реализации, развертывания и прецедентов.

Поместите в соответствующие пакеты элементы и диаграммы, необходимые и достаточные для визуализации, специфирования, конструирования, документирования семантики каждого вида.

При необходимости сгруппируйте элементы каждого вида в более мелкие пакеты.

Между элементами из различных видов, скорее всего, будут существовать отношения зависимости, поэтому в общем случае стоит открыть все виды на верхнем уровне системы для всех остальных видов того же уровня.

АРХИТЕКТУРНЫЕ ВИДЫ

Каноническая декомпозиция верхнего уровня, пригодная даже для самых сложных систем



ДИАГРАММА ПАКЕТОВ

Моделируя пакеты в UML нужно помнить, что они нужны только для организации элементов модели. Если имеются абстракции, непосредственно материализуемые как объект в системе, не пользуйтесь пакетами. Вместо этого применяйте такие элементы моделирования, как классы или компоненты.

Хорошо структурированный пакет характеризуется следующими свойствами:

он внутренне согласован и очерчивает четкую границу вокруг группы родственных элементов;

он слабо связан и экспортирует в другие пакеты только те элементы, которые они действительно должны "видеть", а импортирует лишь элементы, которые необходимы и достаточны для того, чтобы его собственные элементы могли работать;

глубина вложенности пакета невелика, поскольку человек не способен воспринимать слишком глубоко вложенные структуры;

владея сбалансированным набором элементов, пакет по отношению к другим пакетам в системе не должен быть ни слишком большим (если надо, расщепляйте его на более мелкие), ни слишком маленьким (объединяйте элементы, которыми можно манипулировать как единым целым).

Изображая пакет в UML, руководствуйтесь следующими принципами:

применяйте простую форму пиктограммы пакета, если не требуется явно раскрыть его содержимое;

раскрывая содержимое пакета, показывайте только те элементы, которые абсолютно необходимы для понимания его назначения в данном контексте;

моделируя с помощью пакетов сущности, относящиеся к управлению конфигурацией, раскрывайте значения меток, связанных с номерами версий.

ДИАГРАММЫ ВЗАЙМОДЕЙСТВИЯ (INTERACTION DIAGRAM)

Диаграммы взаимодействия описывают поведение совместно действующих групп объектов в рамках потока событий.

Сообщение – это средство, с помощью которого объект-отправитель запрашивает у объекта-получателя выполнение одной из его операций.

ВИДЫ ДИАГРАММ ВЗАИМОДЕЙСТВИЯ

- Диаграммы последовательности,
- Коммуникационные диаграммы,
- Обзорные диаграммы взаимодействия,
- Временные диаграммы (диаграммы синхронизации)

ДИАГРАММЫ ПОСЛЕДОВАТЕЛЬНОСТИ (SEQUENCE DIAGRAM)

Диаграммы последовательности отражают временную последовательность событий, происходящих в рамках варианта использования.

Более детально описывают логику сценариев использования. Это отличное средство документирования проекта с точки зрения сценариев использования!

Аттестационная
комиссия

Математик

Экономист

Юрист

Сколько будет 2×2 ?

Четыре

Сколько будет 2×2 ?

Подумал 5 минут

Что-то в пределах от 3-х до 5-ти

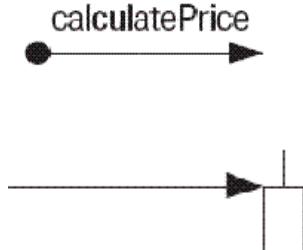
Сколько будет 2×2 ?

Столько, сколько нужно

Пример диаграммы

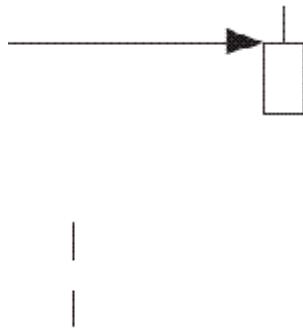
ЭЛЕМЕНТЫ ДИАГРАММЫ ПОСЛЕДОВАТЕЛЬНОСТИ

Найденное сообщение



У первого сообщения нет участника, пославшего его, поскольку оно приходит из неизвестного источника. Оно называется найденным сообщением (found message).

Сообщение



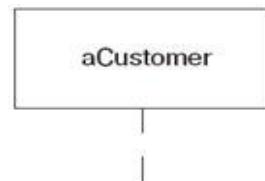
Команда, отправляемая другому участнику. Может содержать только передаваемые данные.

Линия жизни



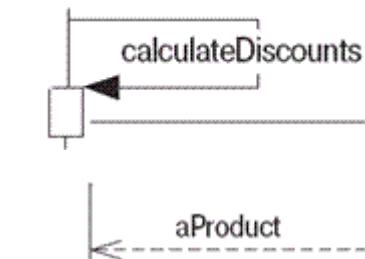
Каждая линия жизни имеет полосу активности, которая показывает интервал активности участника при взаимодействии. Она соответствует времени нахождения в стеке одного из методов участника.

Участник



В большинстве случаев можно считать участников диаграммы взаимодействия объектами, как это и было в действительности в UML 1. Но в UML 2 их роль значительно сложнее. Поэтому здесь употребляется термин участники (participants), который формально не входит в спецификацию UML.

Самовызов



Участник отправляет сообщение (команду) самому себе.

Возврат



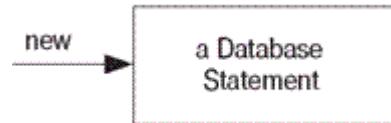
Передача управления обратно участнику, который до этого инициировал сообщение.

ЭЛЕМЕНТЫ ДИАГРАММЫ ПОСЛЕДОВАТЕЛЬНОСТИ

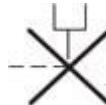
Активация



Создание



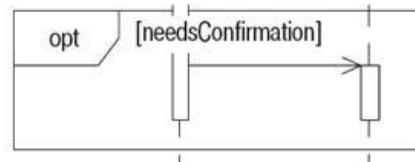
Самоудаление



Удаление из другого объекта



фрейм



На изображении это — белый вертикальный прямоугольник. Момент, когда участник начинает действовать в ответ на принятое сообщение.

В случае создания участника надо нарисовать стрелку сообщения, направленную к прямоугольнику участника. Если применяется конструктор, то имя сообщения не обязательно, но можно маркировать его словом «*new*» в любом случае. Если участник выполняет что-нибудь непосредственно после создания, например команду запроса, то надо начать активацию сразу после прямоугольника участника.

Удаление участника обозначается большим крестом (X). X в конце линии жизни показывает, что участник удаляет сам себя.

Удаление участника обозначается большим крестом (X). Стрелка сообщения, идущая в X, означает, что один участник явным образом удаляет другого.

ФОРМАТЫ СООБЩЕНИЙ

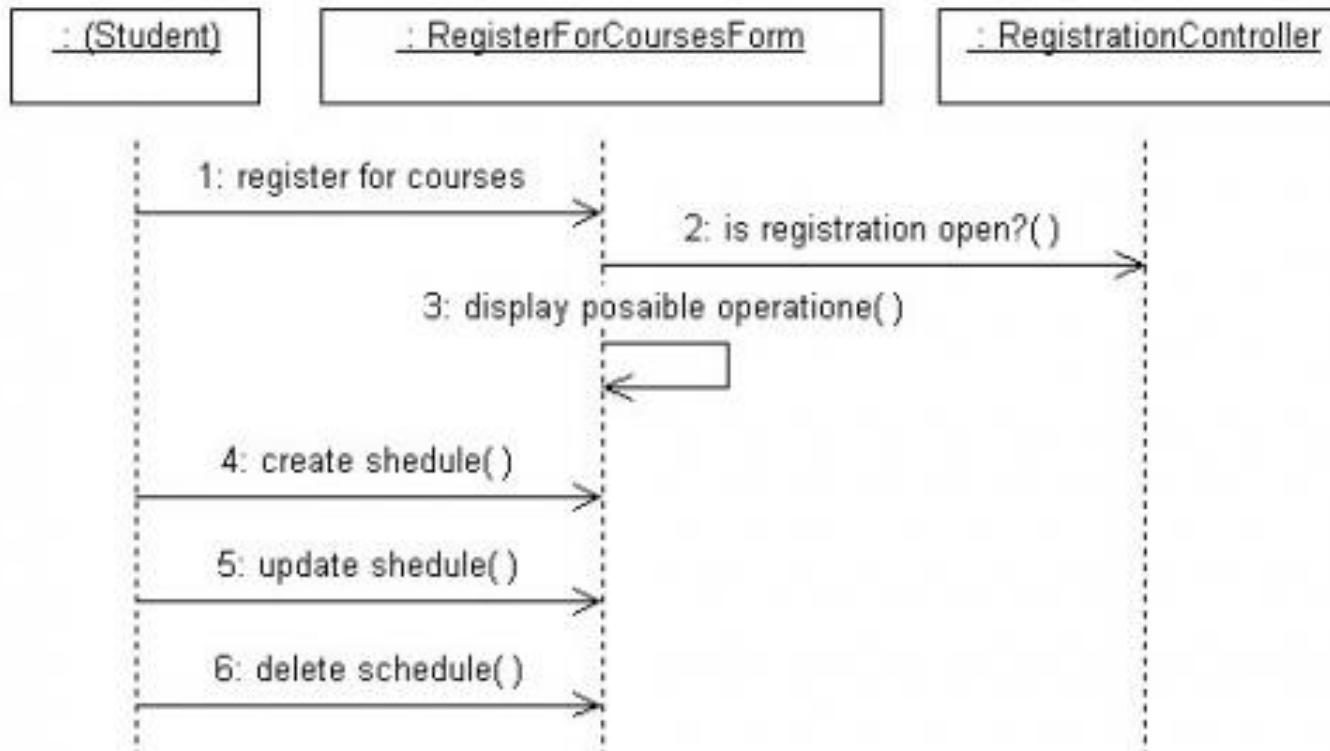
Типы сообщений:

- **Синхронное** - клиент посылает сообщение серверу и ждет, пока тот примет и обработает сообщение . Обозначается: →
- **Асинхронное** - клиент посылает сообщение серверу и, не дожидаясь ответа, продолжает выполнять следующие операции. Обозначается: →
- **Возврат**- обозначающее возврат значения или управления от сервера обратно клиенту. Стрелки этого вида зачастую отсутствуют на диаграммах, поскольку неявно предполагается их существование после окончания процесса выполнения операции. Обозначается: <--

ФОРМАТЫ СООБЩЕНИЙ

Структура сообщения:

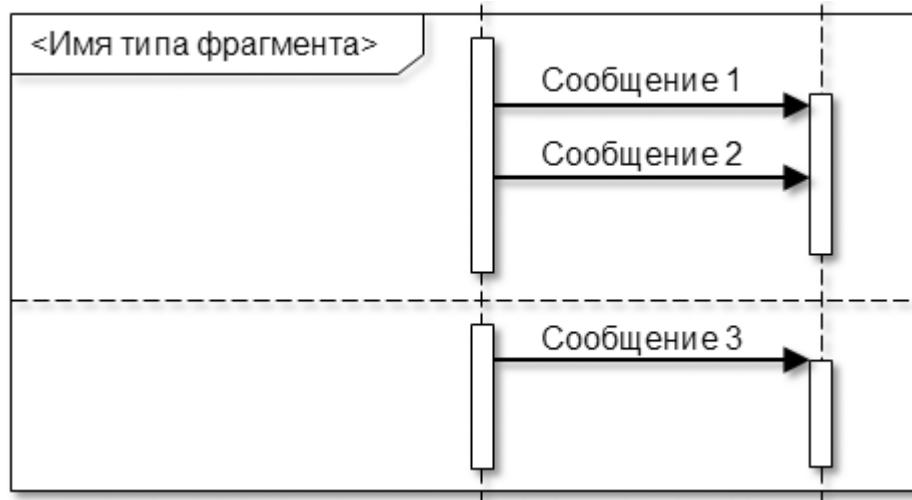
Номер : [переменная =] имя([список параметров]) [:возвращаемое значение].



ФРЕЙМЫ ВЗАИМОДЕЙСТВИЯ

В основном фреймы состоят из некоторой области диаграммы последовательности, разделенной на несколько фрагментов.

Каждый фрейм имеет оператор, а каждый фрагмент может иметь защиту.



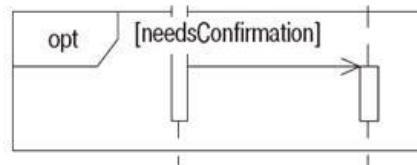
ЭЛЕМЕНТЫ ДИАГРАММЫ ПОСЛЕДОВАТЕЛЬНОСТИ. ФРЕЙМЫ ВЗАИМОДЕЙСТВИЯ

Общая проблема диаграмм последовательности заключается в том, как отображать циклы и условные конструкции.

Прежде всего надо усвоить, что диаграмма последовательности для этого не предназначена. Подобные управляющие структуры лучше показывать с помощью диаграммы деятельности или собственно кода.

Диаграмма последовательности применяется для визуализации процесса взаимодействия объектов, а не как средство моделирования алгоритма управления.

Как было сказано, существуют дополнительные обозначения. И для циклов, и для условий используются фреймы взаимодействий (*inter action frames*), представляющие собой средство разметки диаграммы взаимодействия.



ОПЕРАТОРЫ ВЗАИМОДЕЙСТВИЯ

Отдельные фрагменты диаграммы взаимодействия можно выделить с помощью фрейма. Фрейм должен содержать метку оператора взаимодействия. UML содержит следующие операнды:

Alt - Несколько альтернативных фрагментов (alternative); выполняется только тот фрагмент, условие которого истинно

Opt - Необязательный (optional) фрагмент; выполняется, только если условие истинно. Эквивалентно **alt** с одной веткой

Par - Параллельный (parallel); все фрагменты выполняются параллельно

loop - Цикл (loop); фрагмент может выполняться несколько раз, а защищает тело итерации

region - Критическая область (critical region); фрагмент может иметь только один поток, выполняющийся за один прием

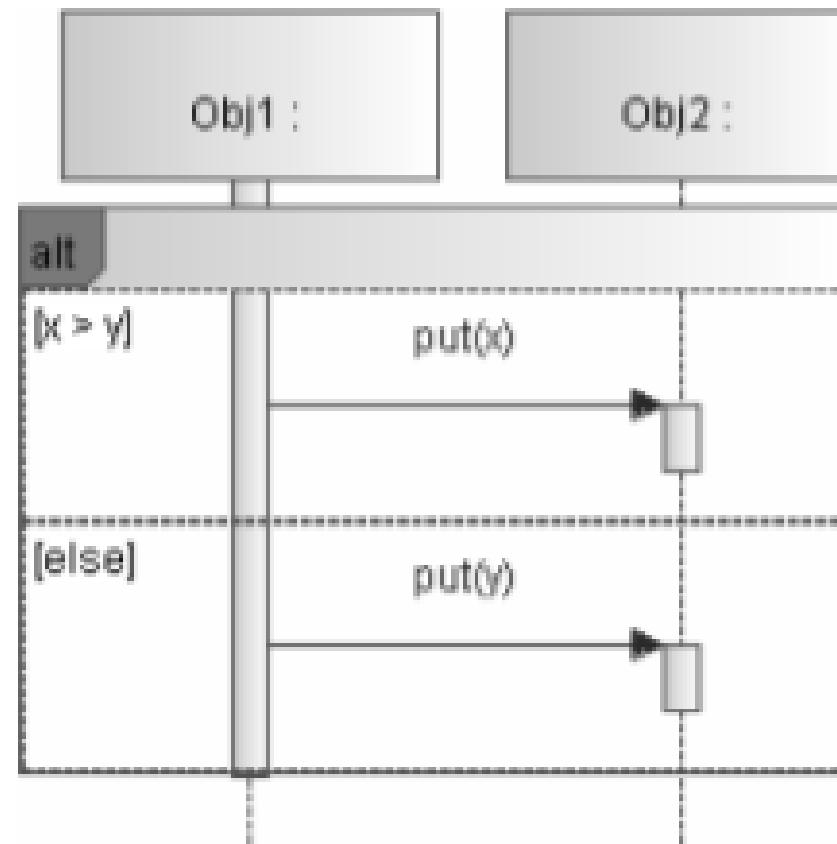
Neg - Отрицательный (negative) фрагмент; обозначает неверное взаимодействие

ref - Ссылка (reference); ссылается на взаимодействие, определенное на другой диаграмме. Фрейм рисуется, чтобы охватить линии жизни, вовлеченные во взаимодействие. Можно определять параметры и возвращать значение

Sd - Диаграмма последовательности (sequence diagram); используется для очерчивания всей диаграммы последовательности, если это необходимо.

ОПЕРАТОРЫ ВЗАИМОДЕЙСТВИЯ

alt



ОПЕРАТОРЫ ВЗАИМОДЕЙСТВИЯ

Loop - цикл

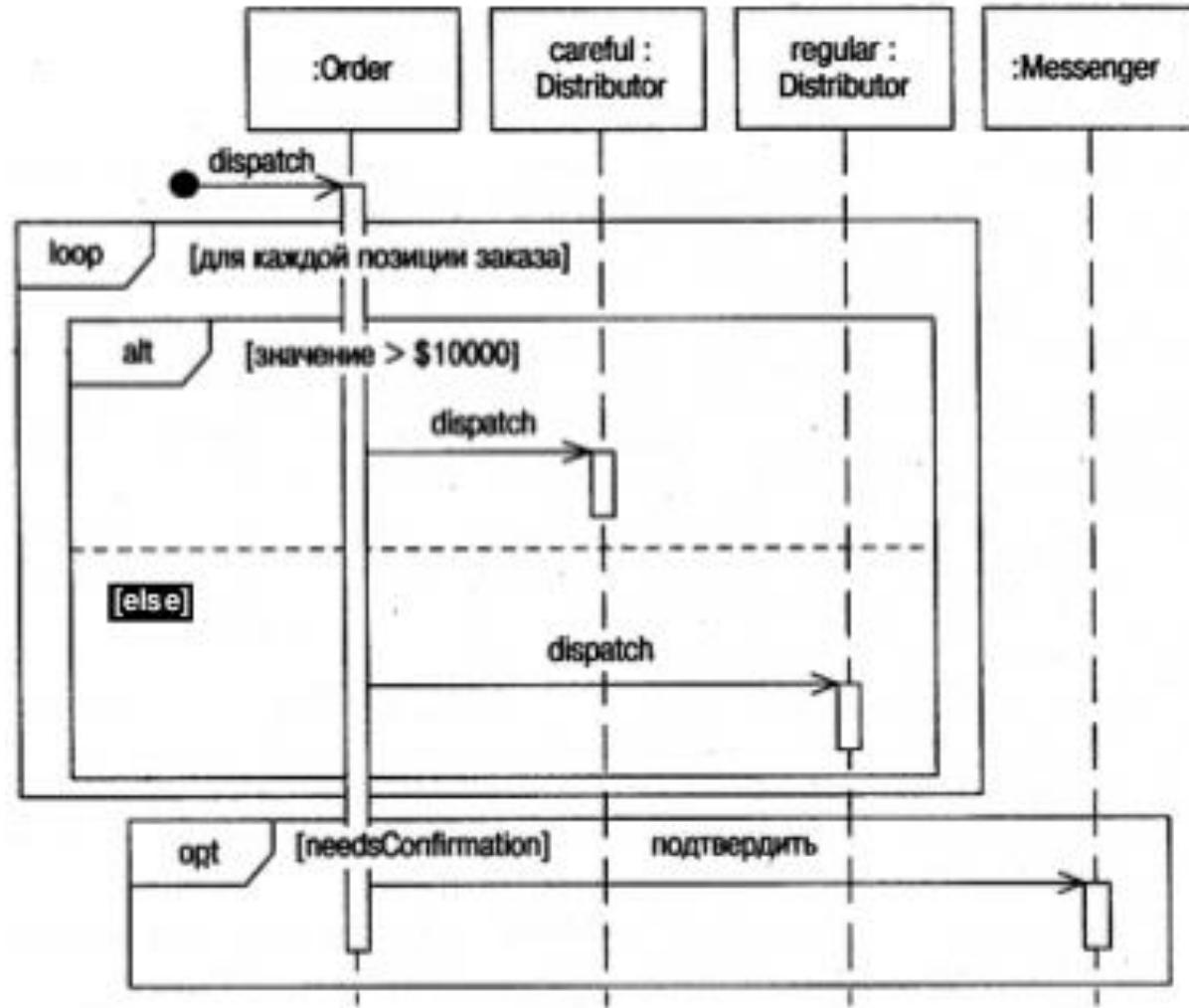
Alt - выполняется

только тот фрагмент,
условие которого

Истинно

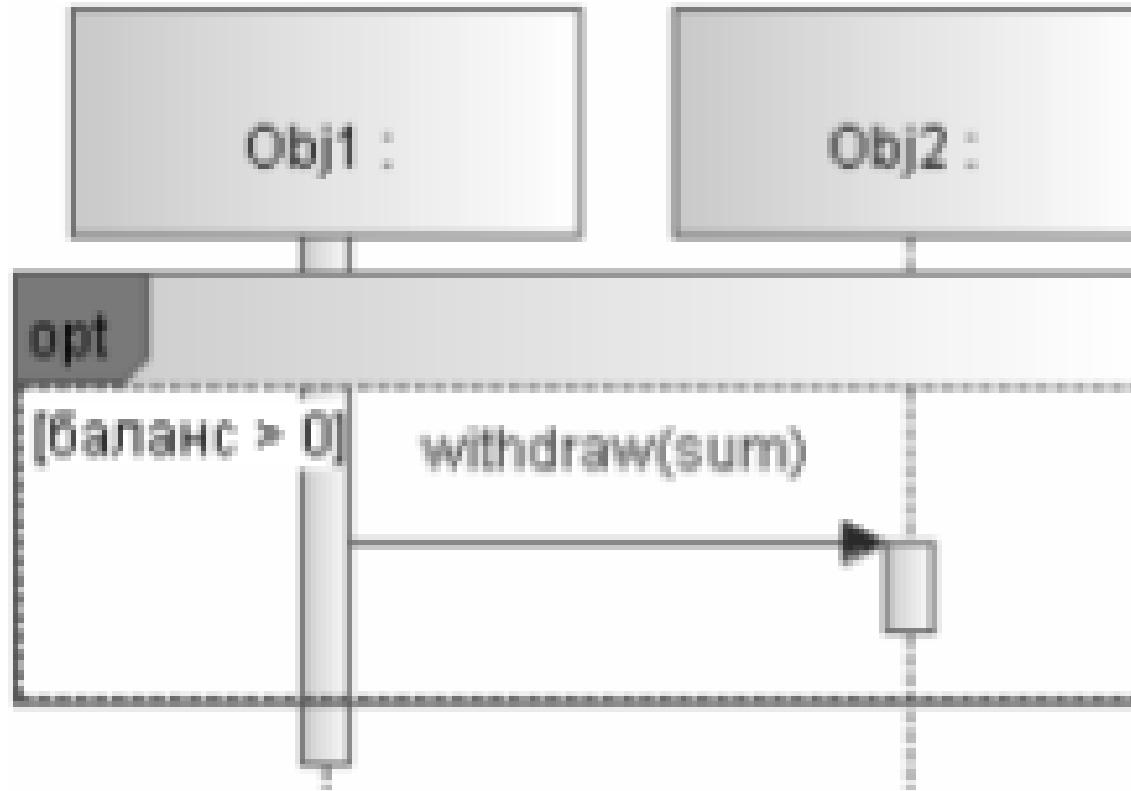
Opt -

Необязательный
(optional) фрагмент;
выполняется,
только если
условие истинно.
Эквивалентно alt с
одной веткой



ОПЕРАТОРЫ ВЗАИМОДЕЙСТВИЯ

Opt - Необязательный (optional) фрагмент; выполняется, только если условие истинно. Эквивалентно **alt** с одной веткой

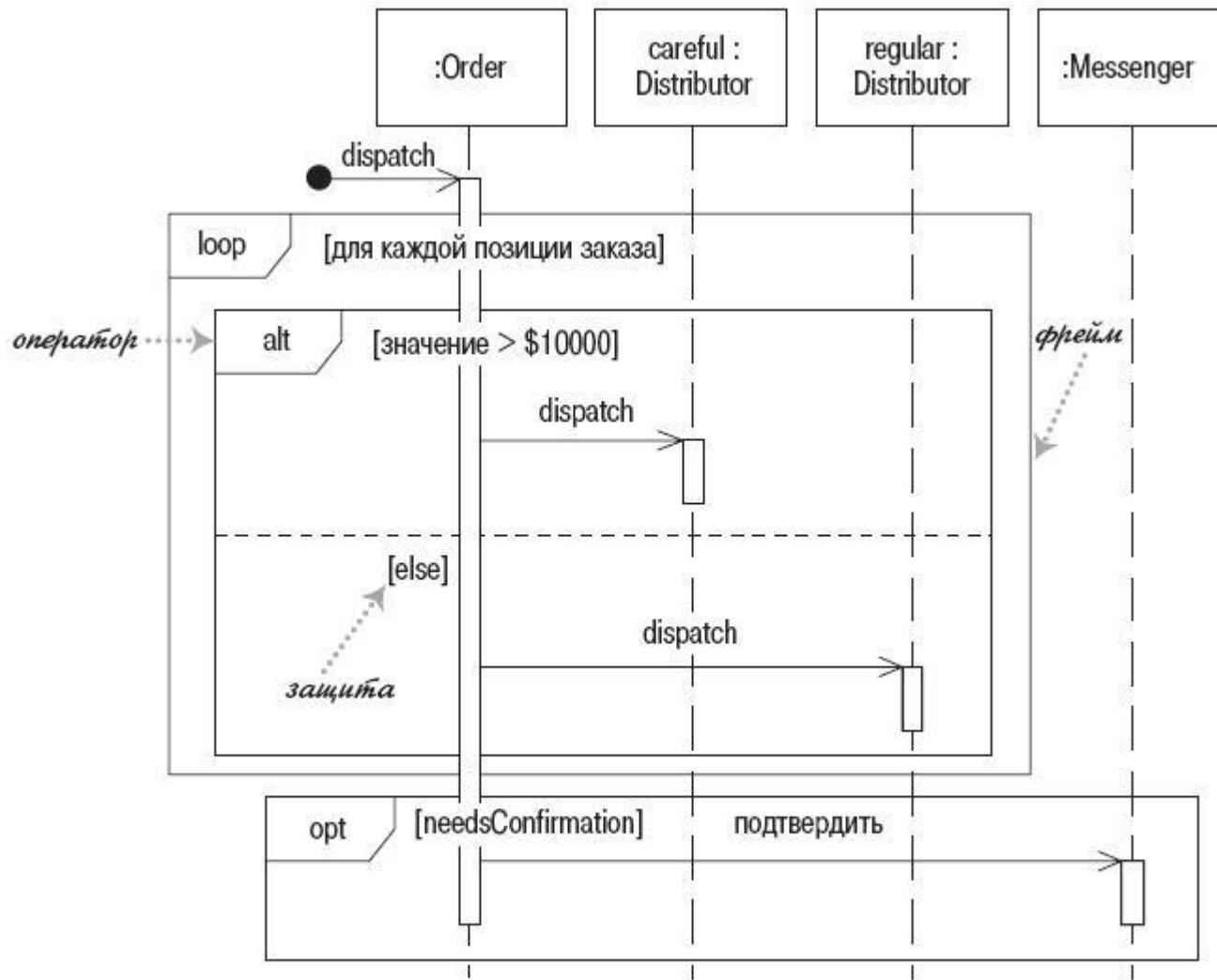


ОПЕРАТОРЫ ВЗАИМОДЕЙСТВИЯ

Par - Параллельный (parallel); все фрагменты выполняются параллельно



ОПЕРАТОРЫ ВЗАИМОДЕЙСТВИЯ



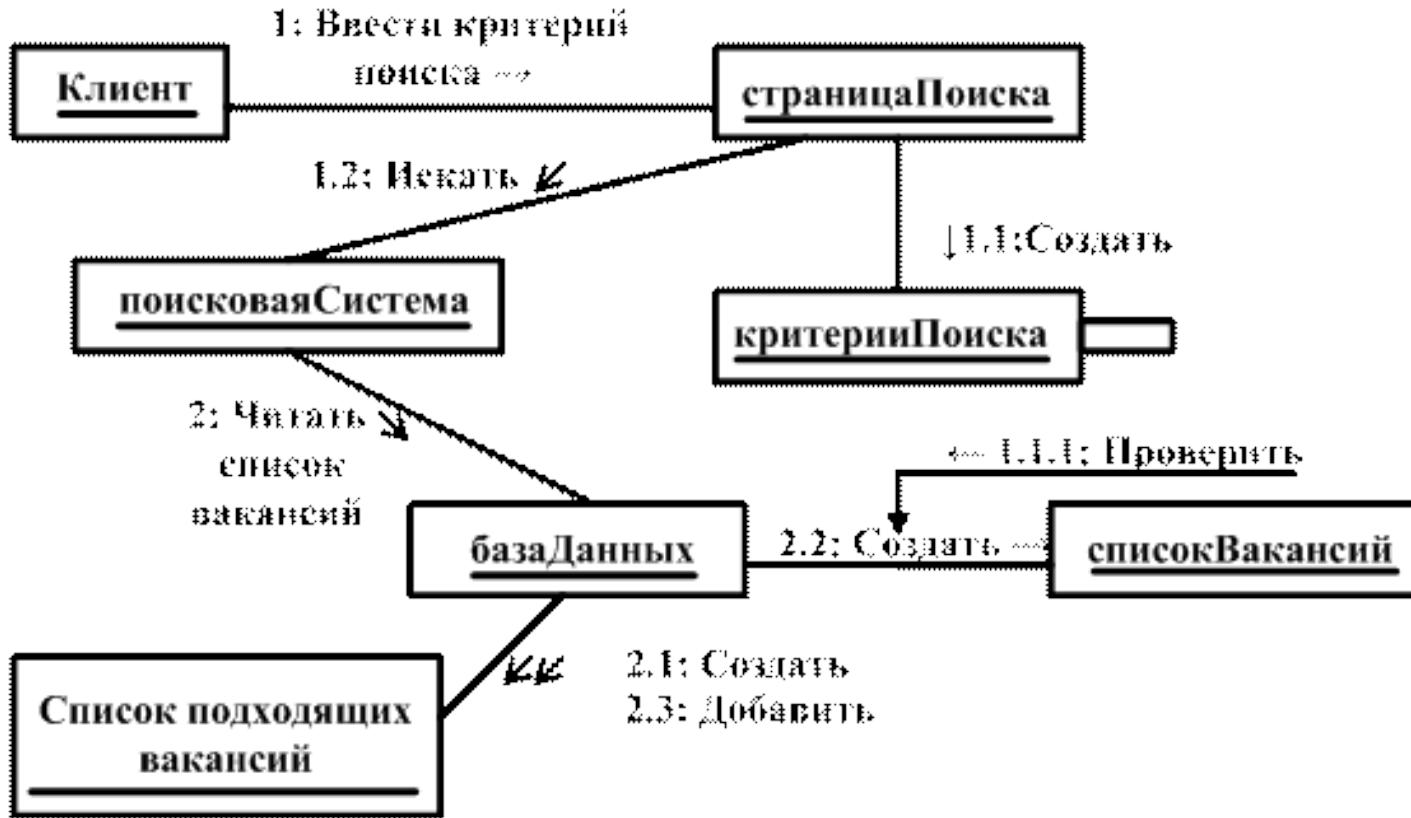
Фреймы взаимодействия

КОММУНИКАЦИОННЫЕ ДИАГРАММЫ (COMMUNICATION DIAGRAM)

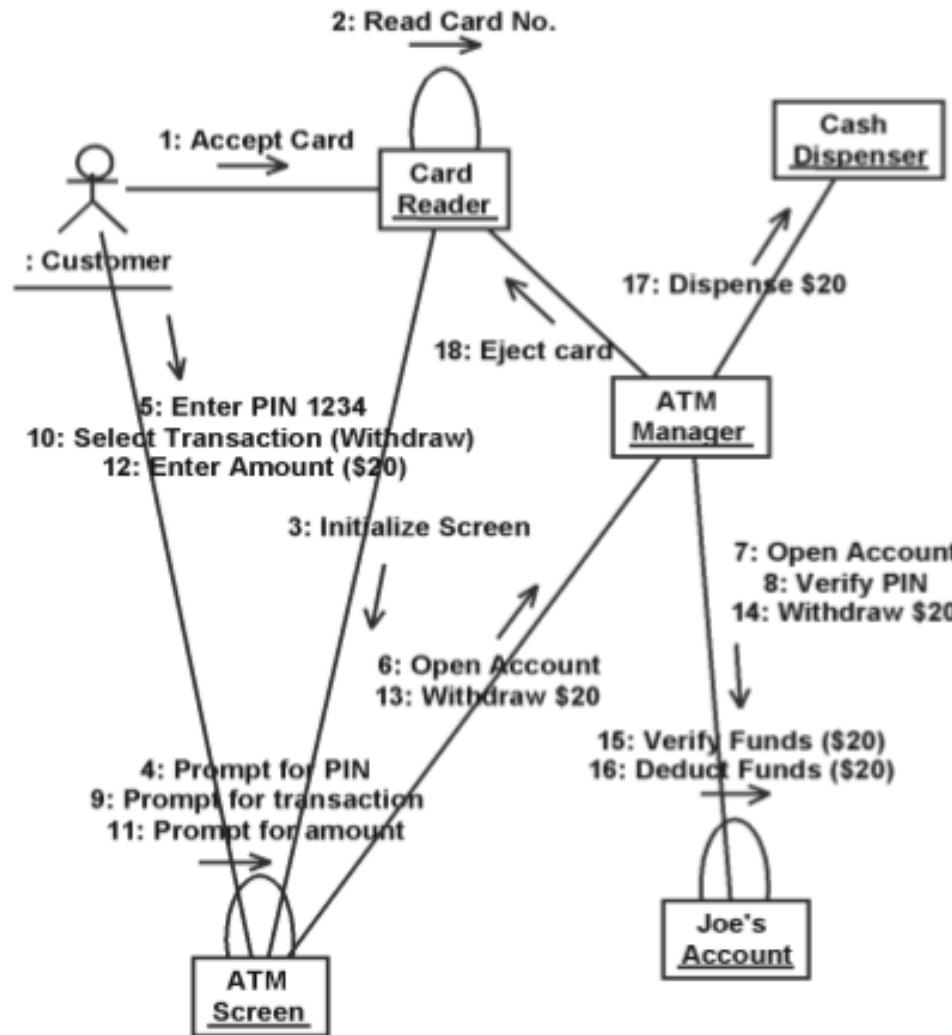
Коммуникационные диаграммы (*communication diagrams*) – это особый вид диаграмм взаимодействия, акцентированных на обмене данными между различными участниками взаимодействия.

КОММУНИКАЦИОННЫЕ ДИАГРАММЫ (COMMUNICATION DIAGRAM)

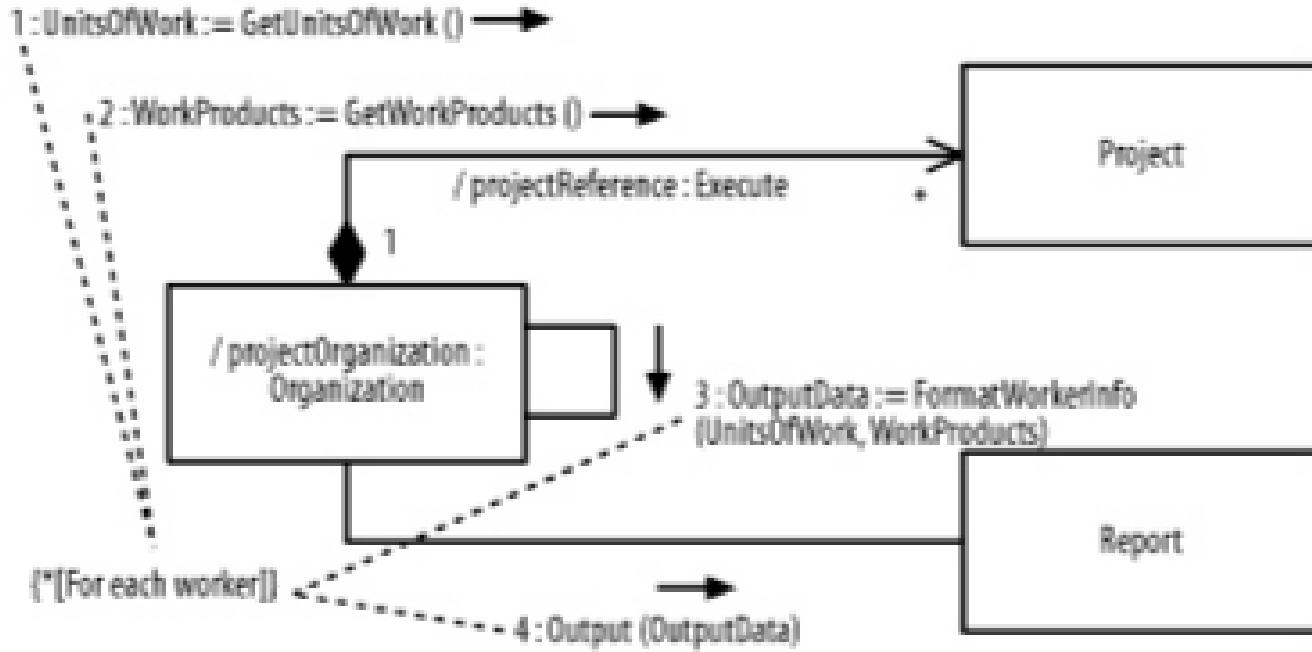
Вместо того, чтобы рисовать каждого участника в виде линии жизни и показывать последовательность сообщений, располагая их по вертикали, как это делается в диаграммах последовательности, коммуникационные диаграммы допускают произвольное размещение участников, позволяя рисовать связи между ними, и использовать вложенную десятичную нумерацию для представления последовательности сообщений.



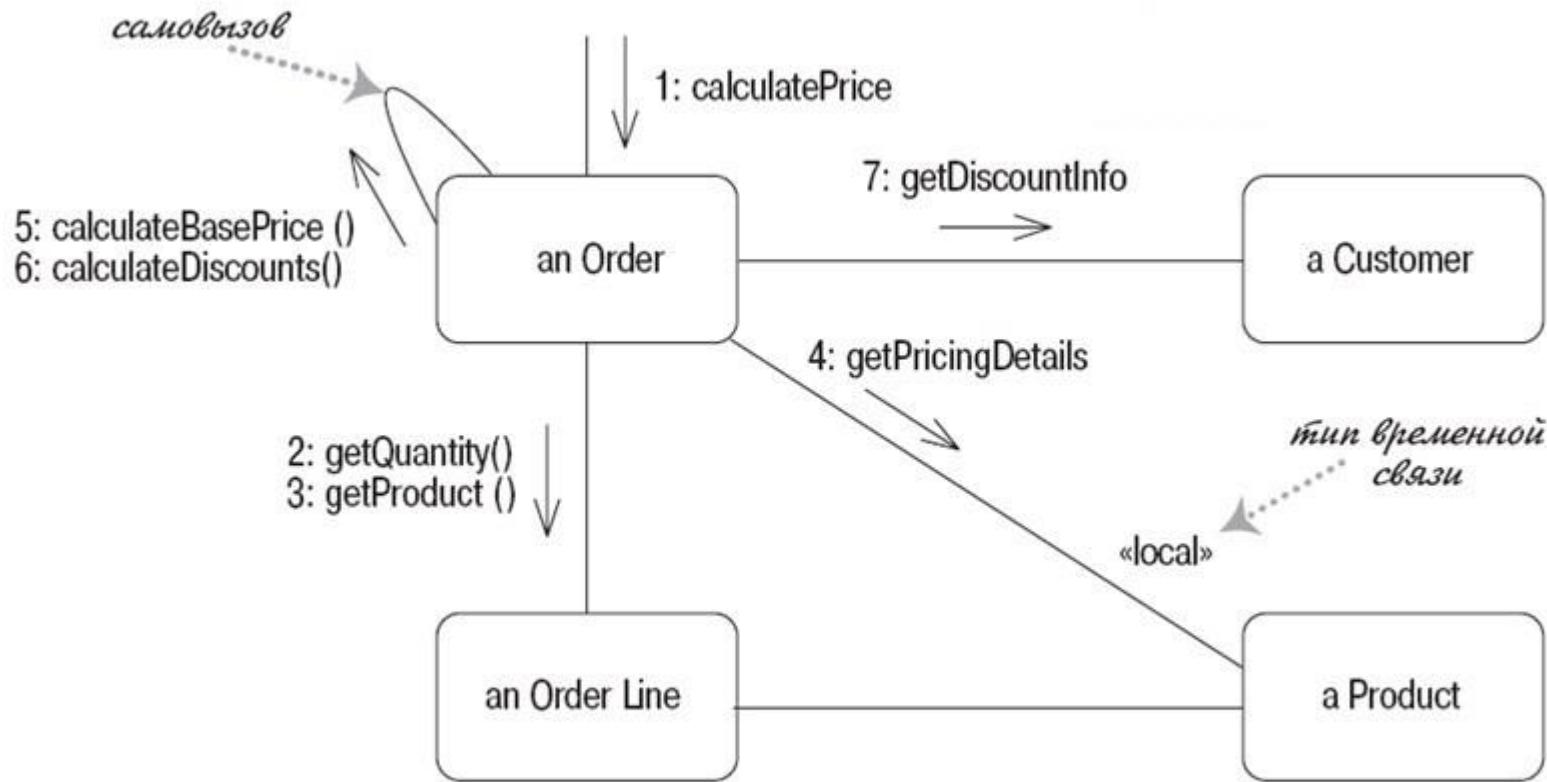
Пример Коммуникационной диаграммы для функции поиска вакансий работ на сайте.



Рефлексивные сообщения и циклы – обозначаются дугой над циклом



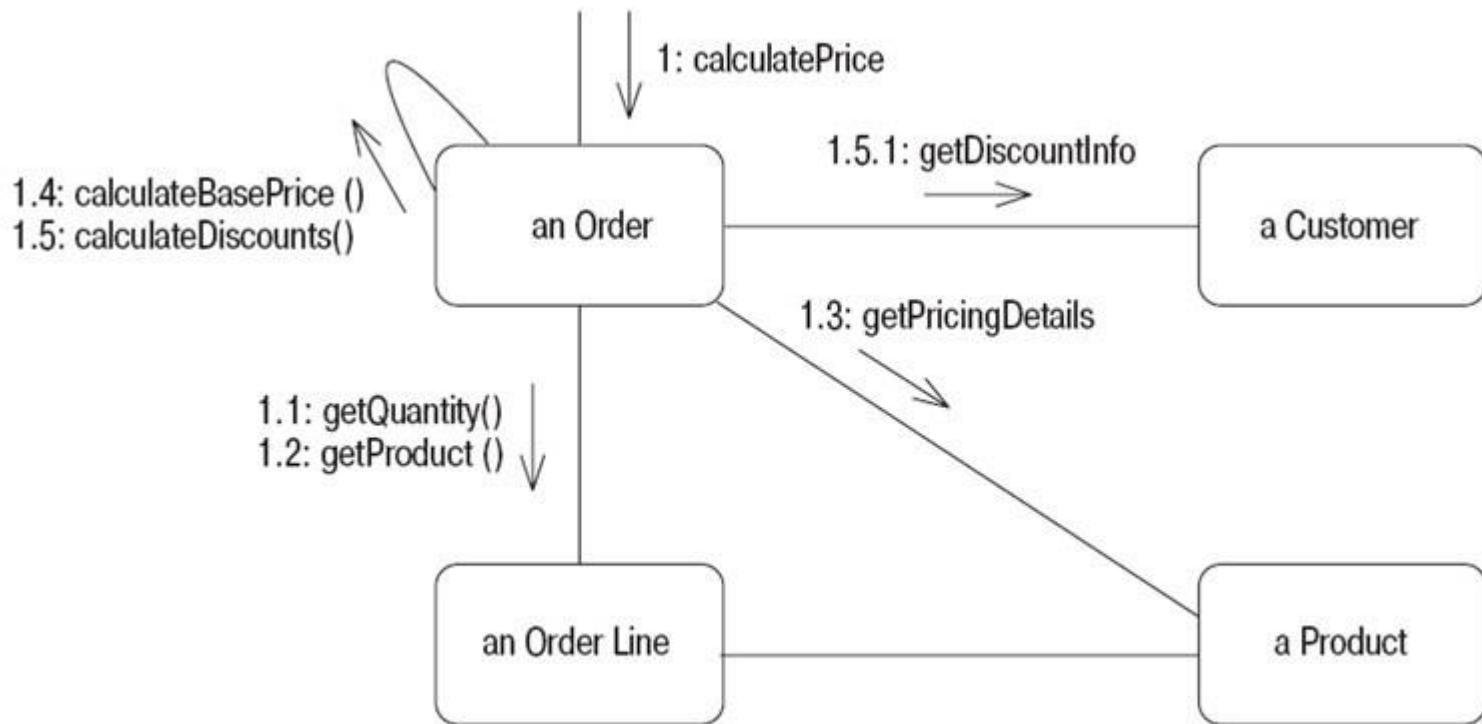
Рефлексивные сообщения и циклы



Коммуникационная диаграмма системы централизованного управления

КД для централизованного управления.

Кроме отображения связей, которые представляют собой экземпляры ассоциаций, можно также показать временные связи, возникающие только в контексте взаимодействия. В данном случае связь «**local**» (локальная) от объекта **Order** (**Заказ**) к объекту **Product** (**Продукт**) – это локальная переменная.



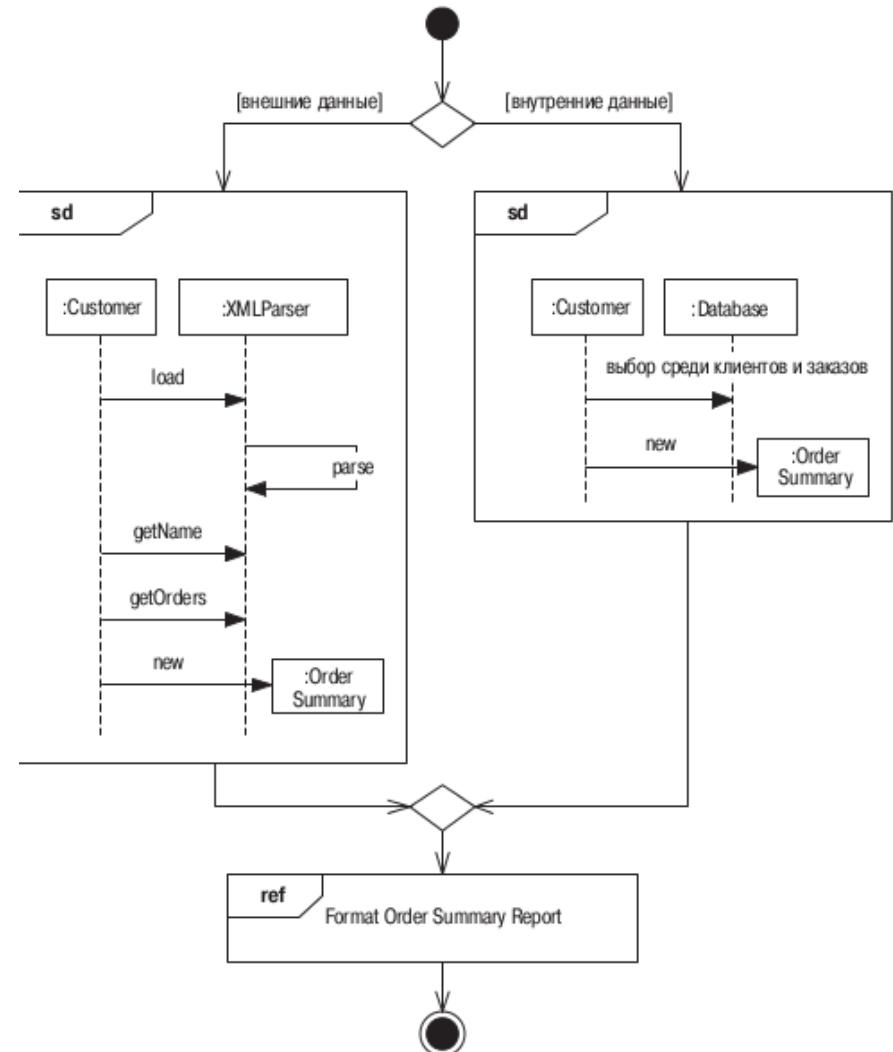
Коммуникационная диаграмма с вложенной десятичной нумерацией

КД для централизованного управления.

В соответствии с правилами UML необходимо придерживаться вложенной десятичной нумерации

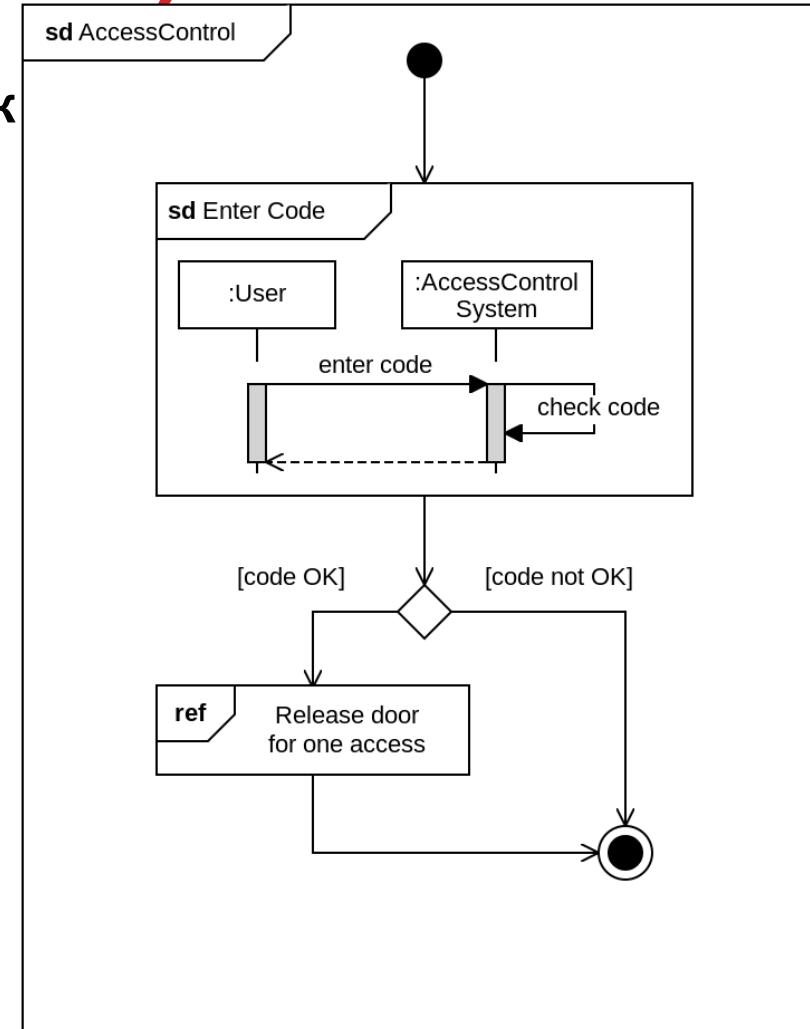
ОБЗОРНЫЕ ДИАГРАММЫ ВЗАЙМОДЕЙСТВИЯ (*INTERACTION OVERVIEW DIAGRAM*)

Обзорные диаграммы взаимодействия – это комбинация диаграмм деятельности и диаграмм последовательности.



ОБЗОРНЫЕ ДИАГРАММЫ ВЗАЙМОДЕЙСТВИЯ (*INTERACTION OVERVIEW DIAGRAM*)

Цель её создания ставится как увязывание в единое целое потока управления между узлами из диаграмм деятельности с последовательностью сообщений между линиями выполнения диаграмм последовательности.: Расширение синтаксиса осуществляется за счёт использования ссылок на взаимодействия, которые основаны на диаграмме последовательности.



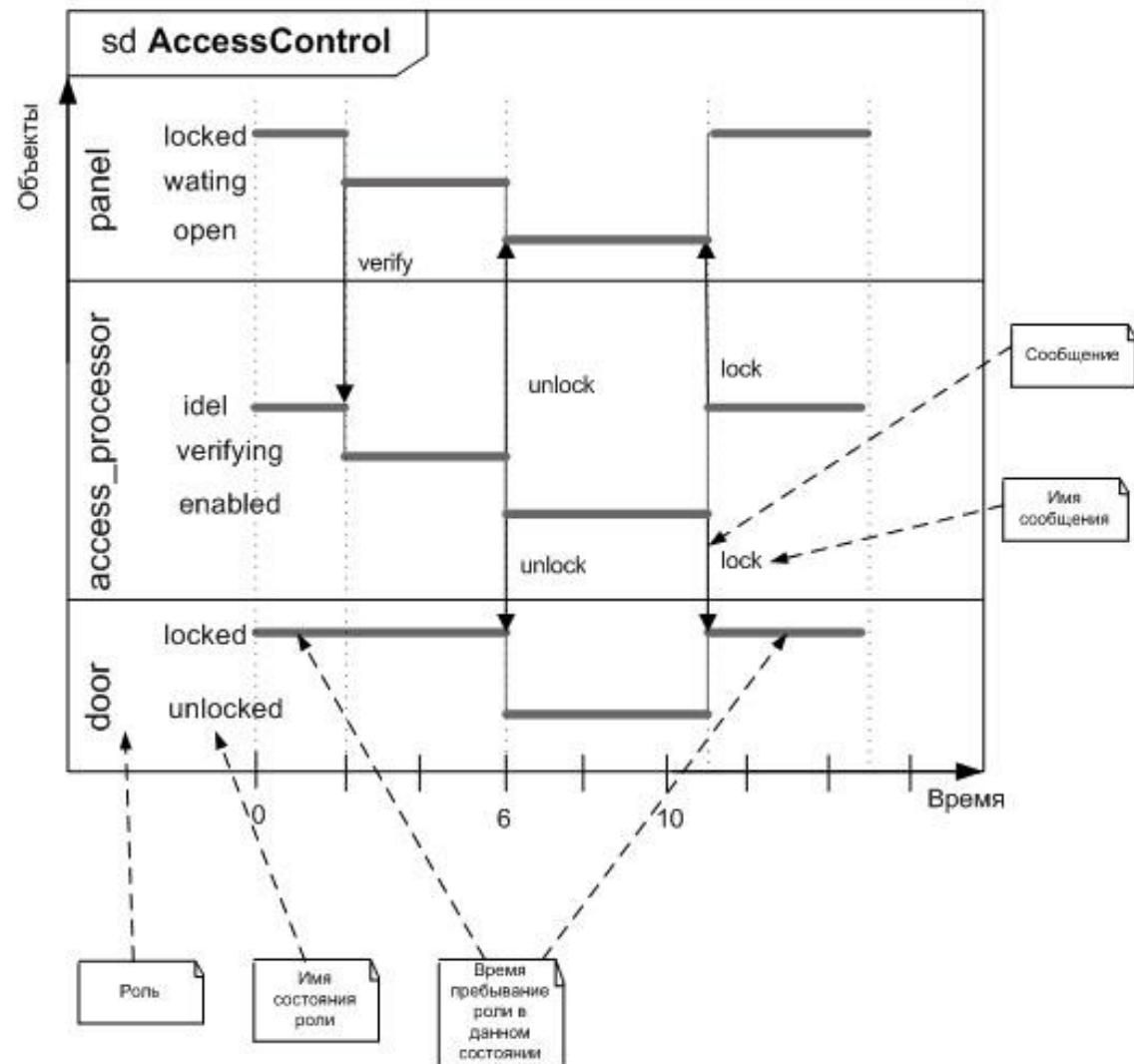
ВРЕМЕННЫЕ ДИАГРАММЫ

Временные диаграммы – это еще одна форма диаграмм взаимодействия, которая акцентирована на временных ограничениях: либо для одиночного объекта, либо, что более полезно, для группы объектов.

Последние изображаются не вертикально, а горизонтально, и основной упор делается на наглядное изображение их состояний, точнее, того, как они меняются во времени. Такая возможность полезна, например, при моделировании встроенных систем.

ВРЕМЕННЫЕ ДИАГРАММЫ

Пример работы системы AccessControl, которая управляет открытием/блокированием двери в помещение по предъявлению человеком электронного ключа. На рисунке показано три компонента этой системы.



ДИАГРАММЫ КОМПОНЕНТОВ (COMPONENT DIAGRAM)

Диаграммы компонентов моделируют физический уровень системы. На них изображаются компоненты ПО и связи между ними.

Во многих средах разработки модуль или компонент соответствует файлу.

Пунктирные стрелки, соединяющие модули, показывают отношения взаимозависимости, аналогичные тем, которые имеют место при компиляции исходных текстов программ.

Основными графическими элементами диаграммы компонентов являются компоненты, интерфейсы и зависимости между ними.

ЦЕЛИ РАЗРАБОТКИ ДИАГРАММ КОМПОНЕНТОВ

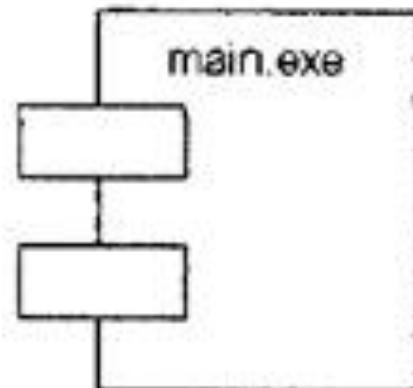
- визуализации общей структуры исходного кода программной системы;
- спецификации исполняемого варианта программной системы;
- обеспечения многократного использования отдельных фрагментов программного кода;
- представления концептуальной и физической схем баз данных.

КОМПОНЕНТ

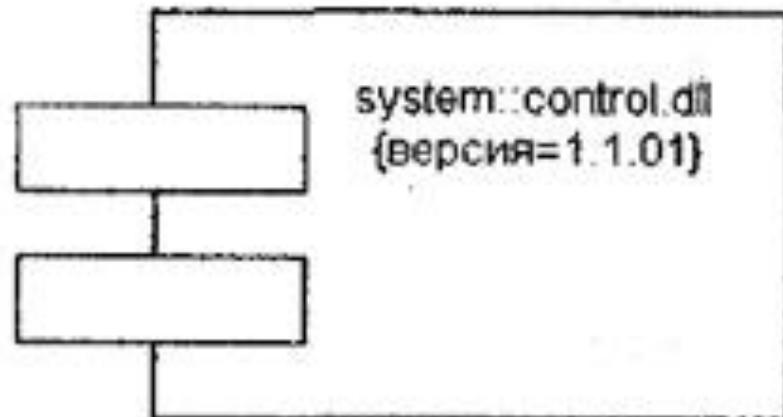
Имя компонента подчиняется общим правилам именования элементов модели в языке UML.

Компонент на уровне экземпляра:

<имя компонента>' : '<имя типаX>



(а)



(б)

ГРАФИЧЕСКИЕ ПУТИ И УЗЛЫ

Графические узлы и пути

Тип графического элемента	Нотация
Компонент (component) с текстовым стереотипом	
Компонент (component) с пиктограммой стереотипа	
Компонент с предоставляемым интерфейсом (provided interface)	
Компонент имеет порт (port) с предоставляемым интерфейсом	
Компонент с требуемым интерфейсом (required interface)	
Компонент имеет порт (port) с требуемым интерфейсом	

ГРАФИЧЕСКИЕ ПУТИ И УЗЛЫ

Компонент имеет не-
сколько портов (ports)
с предоставляемыми
и требуемыми интер-
фейсами



Класс (class)

Имя класса

Часть (part)

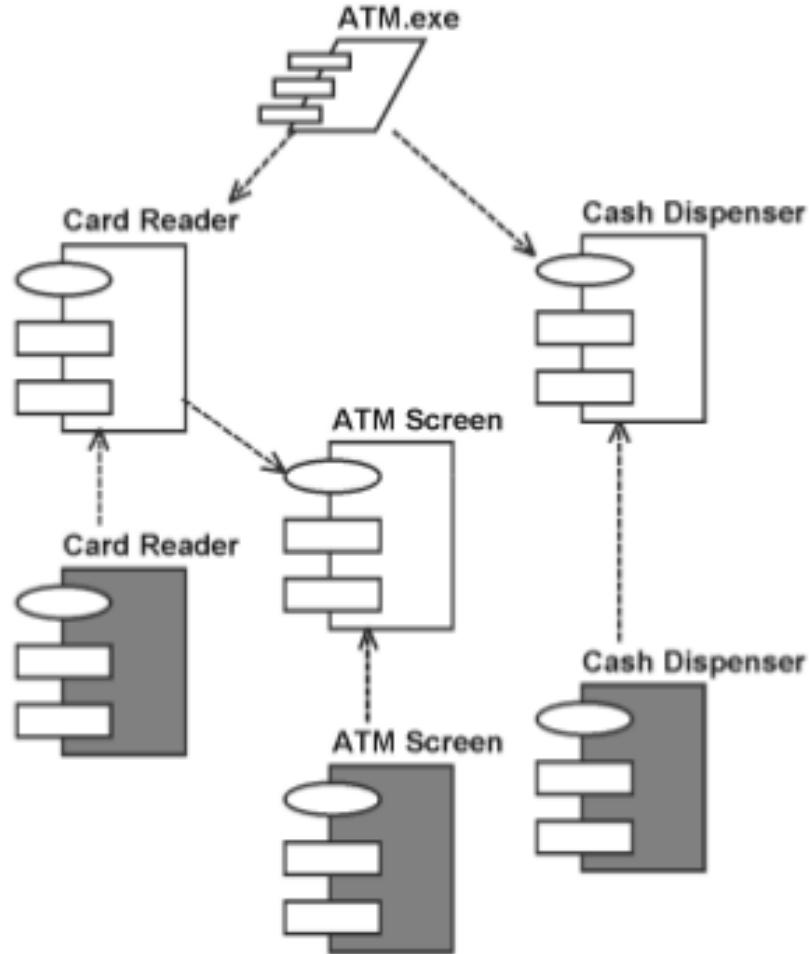
имя части: Имя класса

Собирающий
соединитель
(assembly connector)



СТЕРЕОТИПЫ ДЛЯ КОМПОНЕНТОВ

- библиотека (library)
- таблица (table)
- файл (file)
- документ (document)
- исполняемый (executable)



Пример диаграммы компонентов

ДИАГРАММЫ РАЗВЕРТЫВАНИЯ / РАЗМЕЩЕНИЯ (DEPLOYMENT DIAGRAM)

Это один из видов диаграмм реализации в UML, показывающий физическую конфигурации аппаратного и программного обеспечения. Можно построить несколько диаграмм развертывания для данной системы, каждая из них будет фокусироваться на разных аспектах системы или показывать разные аспекты модели.

ДИАГРАММЫ РАЗВЕРТЫВАНИЯ / РАЗМЕЩЕНИЯ (DEPLOYMENT DIAGRAM)

Диаграмма развертывания (deployment diagram) - диаграмма, на которой представлены узлы выполнения программных компонентов реального времени, а также процессов и объектов.

Диаграмма развёртывания показывает топологию системы и распределение компонентов системы по ее узлам, а также соединения - маршруты передачи информации между аппаратными узлами. Это единственная диаграмма, на которой применяются “трехмерные” обозначения: узлы системы обозначаются кубиками. Все остальные обозначения в UML - плоские фигуры обязательно компьютерных.

ДИАГРАММЫ РАЗВЕРТЫВАНИЯ / РАЗМЕЩЕНИЯ (DEPLOYMENT DIAGRAM)

Польза диаграмм развёртывания

- Графическое представление ИТ-инфраструктуры может помочь более рационально распределить компоненты системы по узлам сети, от чего зависит в том числе и производительность системы.
- Такая диаграмма может помочь решить множество вспомогательных задач, связанных, например, с обеспечением безопасности.

ДИАГРАММЫ РАЗМЕЩЕНИЯ

Диаграмма размещения (развертывания) отражает физические взаимосвязи между программными и аппаратными компонентами системы.

Основные элементы:

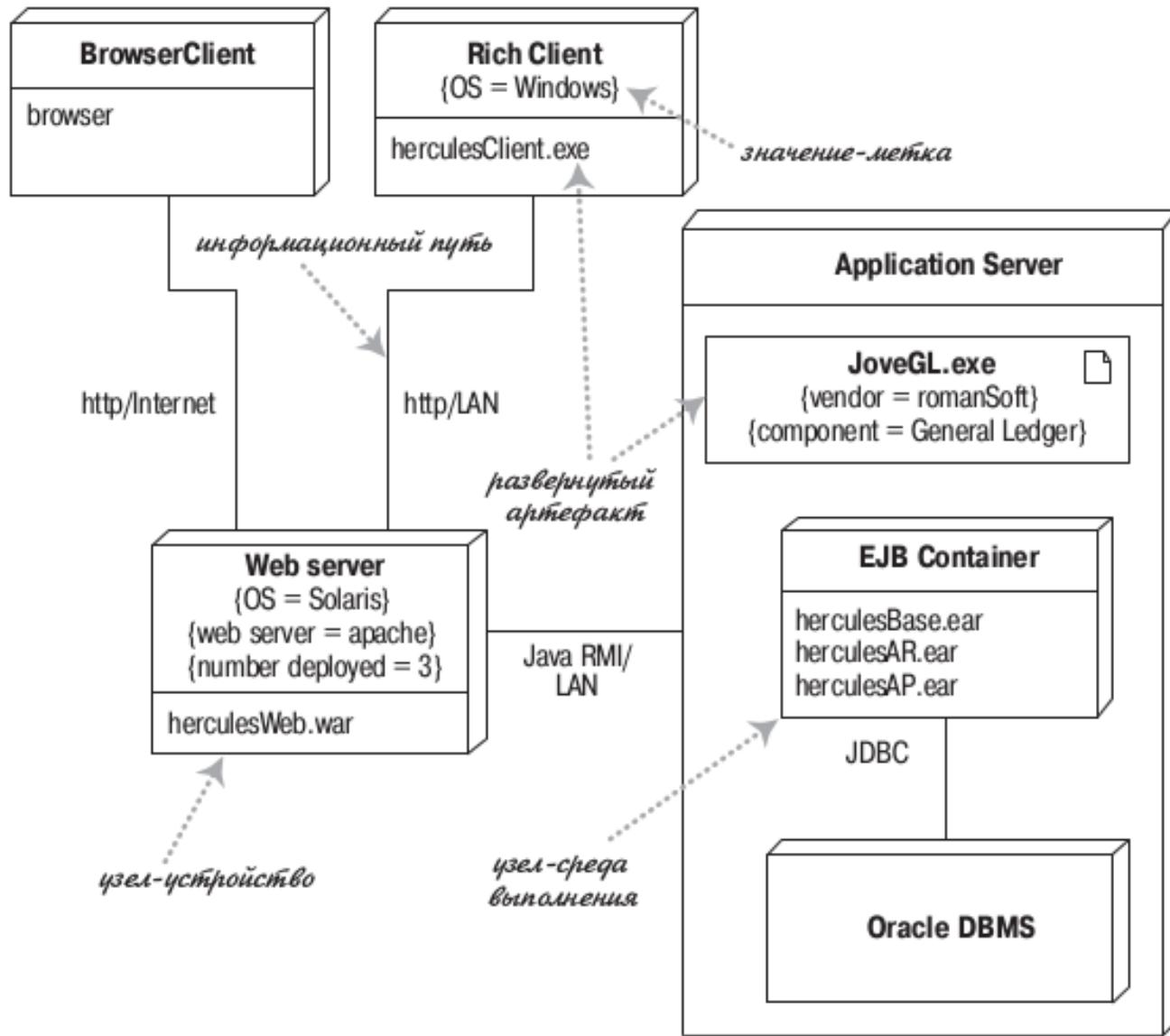
1. узел (node):

- устройство (device)
- среда выполнения (execution environment)

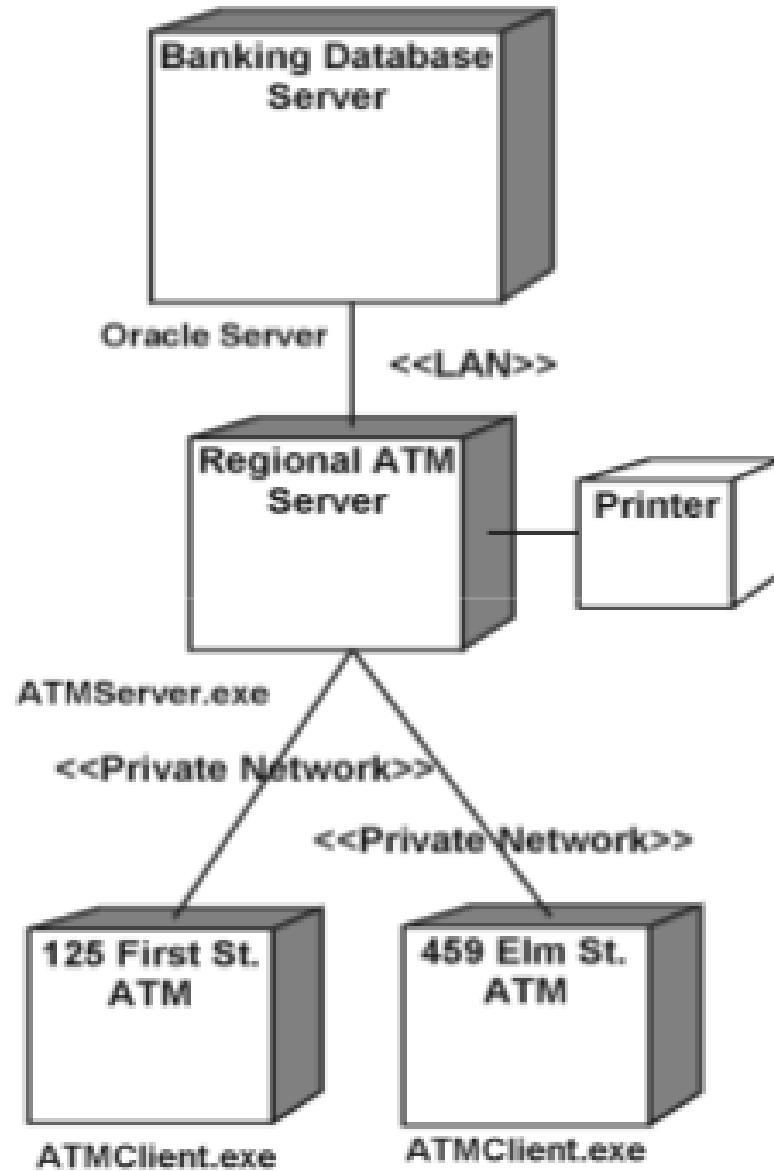
2. линия коммуникации (связи) (communication line)

ОСНОВНЫЕ ЭЛЕМЕНТЫ

	Компонент (Component)
	Экземпляр компонента (Component instance)
	Интерфейс (Interface)
	Узел (Node)
	Экземпляр узла (Node instance)
	Объект (Object)
	Активный объект (Active object)
	Зависимость (Dependency)
	Пример Связь (Connection)
	Точка изгиба связей (Point)
	Комментарий (Note)
	Коннектор комментария (Note connector)



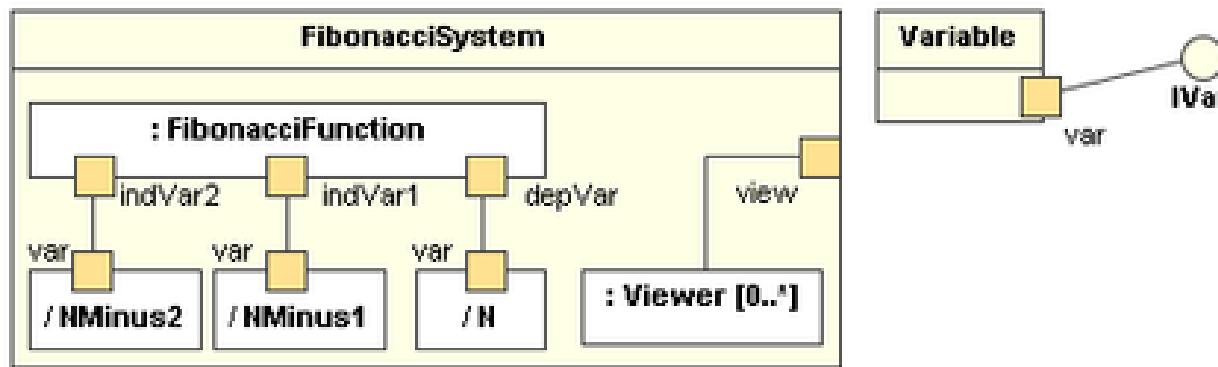
Пример диаграммы размещения



Пример диаграммы размещения

ДИАГРАММЫ СОСТАВНОЙ СТРУКТУРЫ

Диаграммы составных структур (Composite Structure Diagram)
- статическая структурная диаграмма, демонстрирует внутреннюю структуру классов и, по возможности, взаимодействие элементов (частей) внутренней структуры класса.

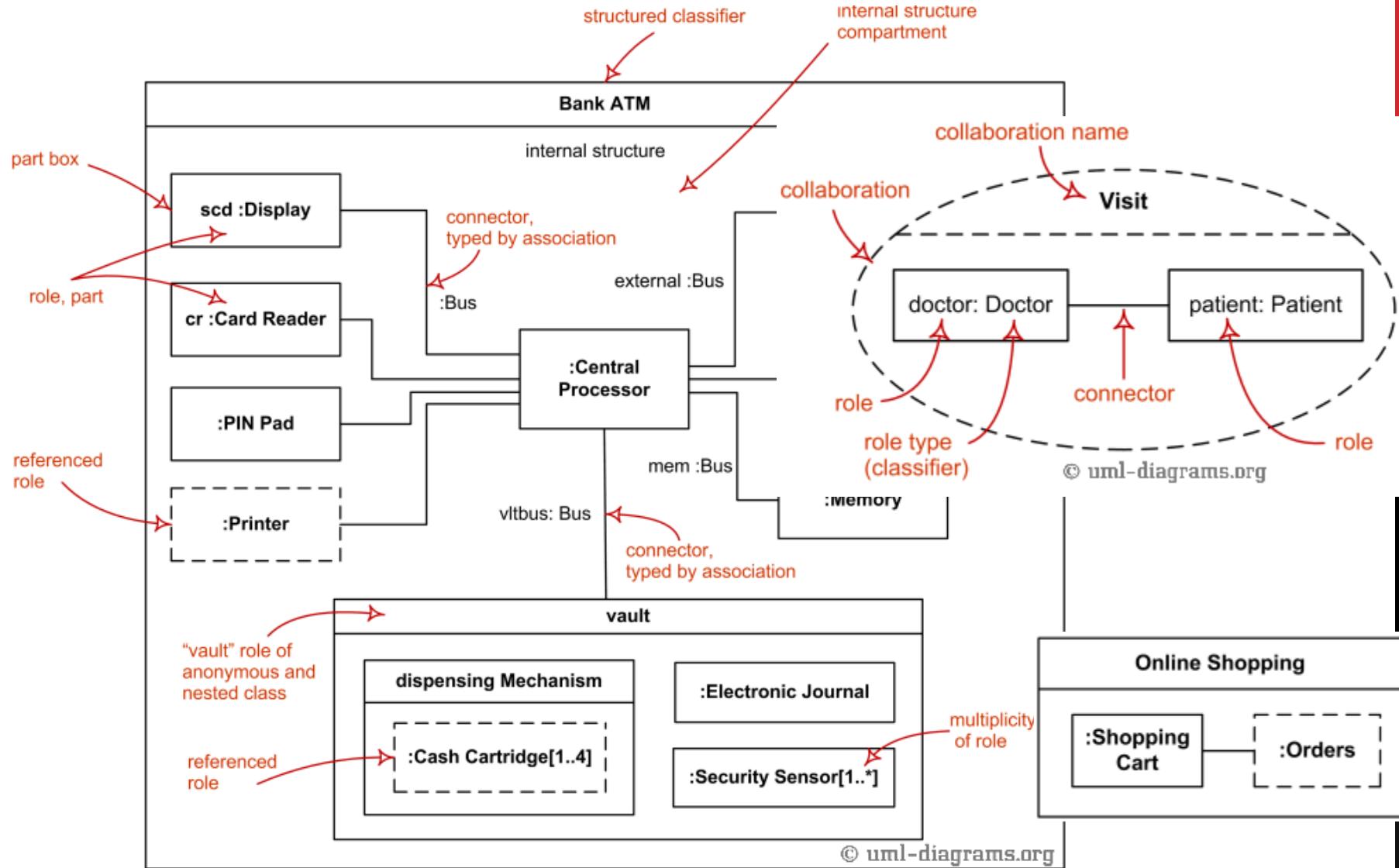


НАПРАВЛЕНИЯ ИСПОЛЬЗОВАНИЯ

Внутренняя структура классификатора

Классификатор взаимодействия

Поведение при взаимодействии

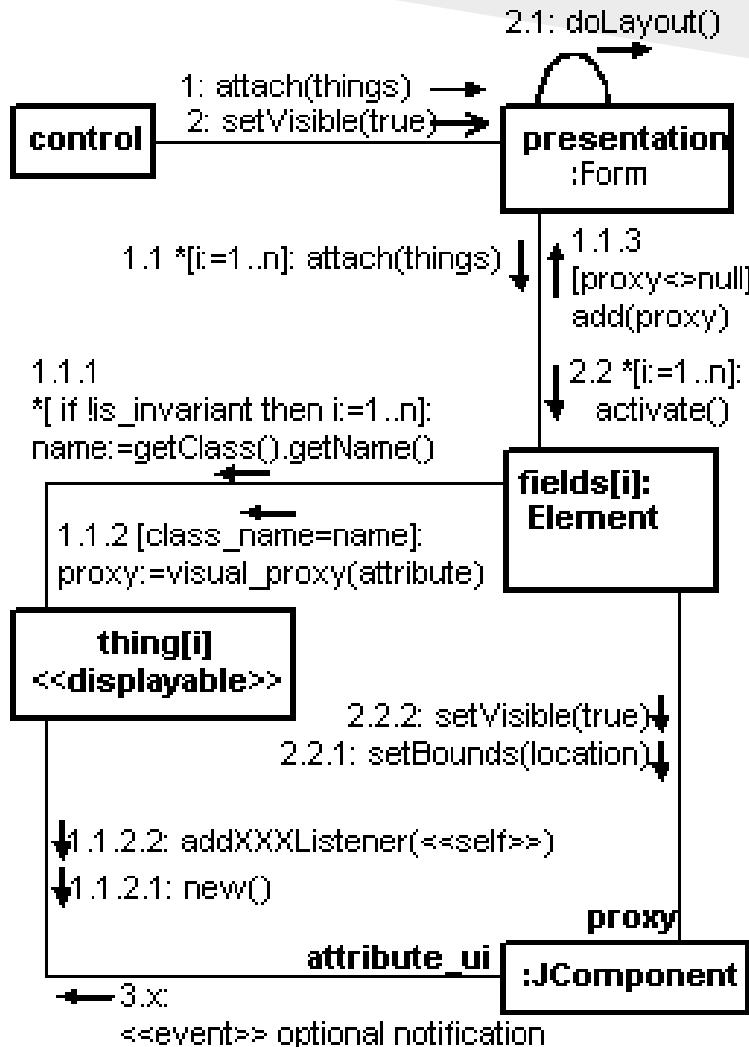


Примеры диаграмм составной структуры

ДИАГРАММА КООПЕРАЦИИ (COLLABORATION DIAGRAM)

Диаграмма кооперации - диаграмма взаимодействий, в которой основной акцент сделан на структурной организации объектов, посылающих и получающих сообщения. На этой диаграмме изображено, как организованы взаимодействия между экземплярами и какие между ними существуют связи. Это, по сути, альтернативная форма диаграммы последовательностей, более компактная, но и более сложная для чтения.

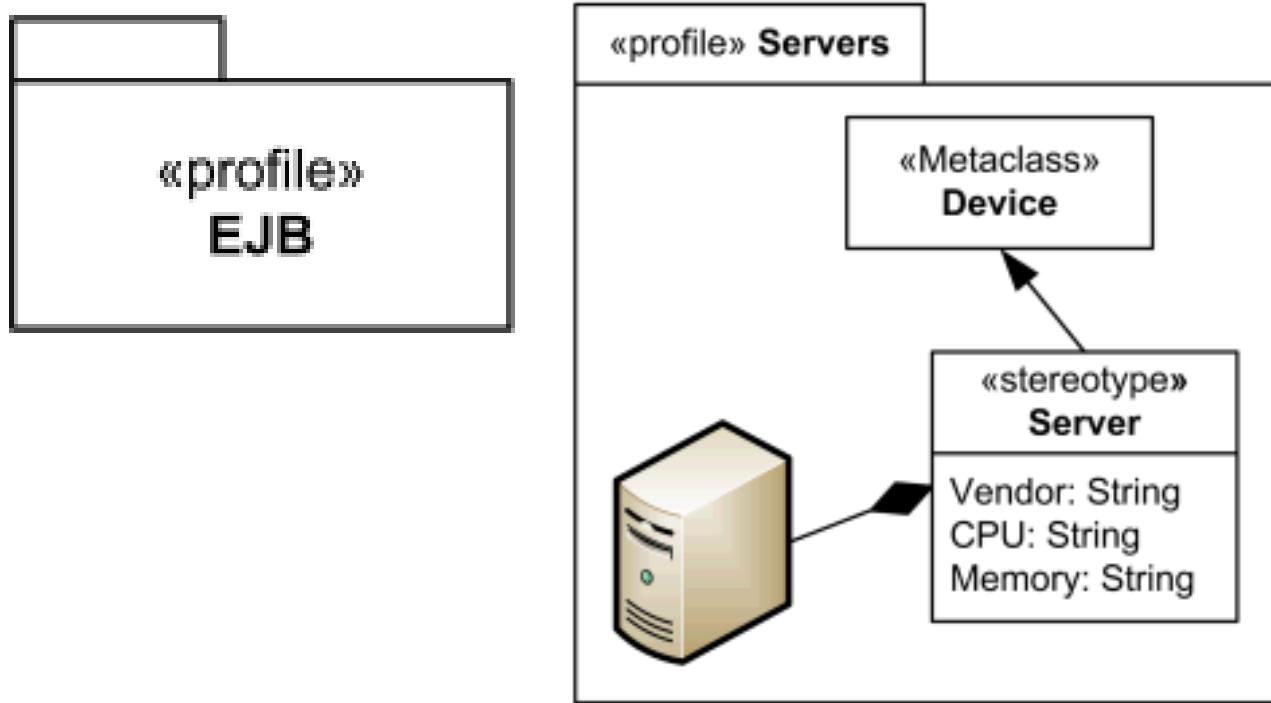
ДИАГРАММА КООПЕРАЦИИ (COLLABORATION DIAGRAM)



ДИАГРАММЫ ПРОФИЛЯ

Диаграмма профиля (*profile diagram*) — структурная диаграмма, которая описывает простой механизм расширения в UML, определив собственные стереотипы, поименованные значения и ограничения. Профили позволяют адаптировать UML метамодели под разные:

- платформы (J2EE или .NET)
- домены (в режиме реального времени или моделирования бизнес-процессов).



Пример диаграмм профиля

СПАСИБО ЗА ВНИМАНИЕ!