

## Оглавление

Предметный указатель .....	2
Конспект лекций по курсу «Объектно-ориентированный анализ и проектирование» .....	8
Лекция 1.    Основы программной инженерии .....	8
Модели ЖЦ: .....	14
Литература к лекции 1 .....	19
Лекция 2.    МОДЕЛИ И ИХ РОЛЬ В СОЗДАНИИ СИСТЕМ. ОБЪЕКТНАЯ МОДЕЛЬ .....	20
Литература к лекции 2 .....	34
Лекция 3.    Унифицированный язык моделирования (Unified Modelling Language) .....	34
Литература к лекции 3 .....	52
Ссылки: .....	52
Лекция 4.    Объектный язык ограничений (OCL) .....	52
Литература к лекции 4 .....	61
Лекция 5.    Моделирование бизнес-процессов .....	61
Литература к лекции 5 .....	70
Лекция 6.    ОПРЕДЕЛЕНИЕ ТРЕБОВАНИЙ К ПРОГРАММНОМУ ОБЕСПЕЧЕНИЮ .....	70
Литература к лекции 6 .....	79
Лекция 7.    Анализ и проектирование программного обеспечения. Анализ ПО .....	79
Литература к лекции 7 .....	95

## Предметный указатель

- access operations, 26
- agile software development, 18
- alt, 38
- AS-IS, 22, 61
- AS-TO-BE, 22, 61
- Booch, 23, 34
- boundary, 41
- Boundary, 51
- business actor, 63
- Business Actor, 63
- Business entity, 66
- Business Event, 69
- business system, 69
- Business Use Case, 62, 63
- Business Use Case Model, 62, 63
- business use case realization, 69
- Business use-case, 64
- Business worker, 66
- C++, 23, 54
- CASE-средства*, 22, 61
- CASE-технология*, 23
- choice pseudostate, 46
- Computer Aided Software Engineering, 22
- connection, 50
- constraint*, 52
- Control, 51
- Creator*, 88, 89
- death march projects, 9
- defer**, 45
- Design Guidelines, 82
- disjoint**, 33, 51
- divide et impera, 20
- do**, 43
- Entity, 51
- entry*, 43
- exit*, 43
- get-теры, 26
- helper operations, 26
- High Cohesion, 20, 88, 90, 91
- implementor operations, 26
- Information Expert*, 88, 89, 90
- inheritance, 33
- Java, 23
- Key Abstractions, 83
- link*, 27
- Liskov substitution principle, 33
- loop, 38
- Low Coupling, 20, 88, 90
- LSP, 33
- manager operations, 26
- Model-view-controller, 84
- multiplicity, 29
- MVC, 85
- neg, 38
- node, 50
- N-арной ассоциации, 30
- N-арные ассоциации, 31
- Object Constraint Language, 52, 53, 61
- Object Modeling Technique, 34
- Objective-C, 23
- Object-Oriented Software Engineering, 34
- OMT, 23, 34
- OOA, 23
- OOD, 23
- OOSE, 34
- opt, 38
- par, 38
- postcondition, 72
- private, 25
- protected, 25
- Rational Software, 34
- Rational Unified Process, 62, 72, 75, 77
- Recommended Practice for Software Requirements Specifications, 71
- ref, 38
- region, 38
- sd, 38
- SE, 10
- set-теры, 26
- Simula, 23
- Smalltalk, 23, 84
- software engineering, 10
- software lifecycle*, 11
- storyboard, 71
- tangible entity, 25
- transition, 43
- UML 1.1, 23, 35
- UML 1.x, 35
- UML 2, 30, 34, 36, 38, 47, 52, 61, 95
- UML 2.0, 30, 34, 36, 38, 52, 61, 95
- Unified Method, 34
- Unified Modeling Language, 34
- Use Case Model, 75
- View Of Participating Classes, 82, 87
- VOPC, 82, 87
- when, 42
- workflow, 78
- абстрагирование, 23
- Абстрагирование*, 24
- Агрегация*, 28
- адаптировать язык моделирования к своим конкретным нуждам, 51
- активации*, 37
- Анализ вариантов использования*, 85, 86

- Анализ поведения, 80  
 Аналитик бизнес-процессов, 62  
 анкетирование, 71  
 аппаратный интерфейс, 87  
 архитектором системы, 50, 82  
*Архитектура ПО*, 20  
*архитектурное представление*, 21  
*Архитектурные механизмы*, 82  
*Архитектурные уровни*, 83  
*Архитектурный анализ*, 82  
 архитектурный объект, 26  
 Ассоциации, 29  
 ассоциации, при помощи которого атрибуты  
 и операции можно привязать  
 непосредственно к соединению, 30  
*Ассоциация*, 28  
*Атрибут*, 25  
*Атрибуты классов анализа*, 94  
*безнадежные проекты*, 9  
 белый ящик, 63  
 библиотеки кода, 50  
 Бизнес правила, 68  
 Бизнес процесс, 64  
 бизнес-логику приложения, 41  
 Бизнес-модель, 61, 62  
 бизнес-правила, 62, 68, 84  
 бизнес-процесс, 63, 64  
*Бизнес-процесс*, 61  
 Бизнес-разработчик, 65, 68  
 Бизнес-система, 69  
 Бизнес-событие, 69  
 блок из параллельно выполняемых  
 разделов, 38  
 блок, включающий диаграмму  
 последовательности целиком, 38  
 БНФ, 51, 54, 58  
*Большое программное обеспечение*, 8  
 Брукс, 10, 19  
 в то же состояние, в котором он в настоящий  
 момент находится, 43  
 Вариант использования, 36, 72  
*Ввод в действие*, 14  
 видение бизнеса, 62, 67  
 Виды альтернативных сценариев, 73  
 Виды сложности, 20  
 вложенные подсостояния, 45  
 внешние бизнес-системы, 63  
 внутренние переходы, 43  
 временную последовательность событий, 37  
 временные диаграммы, 36  
 Вспомогательные операции, 26  
 всякий раз, когда объект переходит в данное  
 состояние, 43  
 Входное действие, 43  
*входной узел*, 47  
*выделения граничных классов*, 86  
*выделения классов-сущностей*, 87  
*выделения управляющих классов*, 87  
 высокая прочность, 20  
 Выходное действие, 43  
 вычислительный ресурс, 50  
*гигантские проекты*, 9  
 глоссарий, 62, 72, 80  
 глоссарий деятельности, 62  
 Гради, 22, 34  
 граничный, 41, 51, 86  
 группировки классов, 41  
 Действующее лицо, 36, 72, 73  
 Действующим лицом, 36  
 декомпозиции, 20, 24, 84  
 Деловая сущность, 66  
 деловое действующее лицо, 63  
 Деловое действующее лицо, 63  
 Дерево целей, 63  
 деятельности внутри события, 42  
 деятельность организации, 61  
 Деятельность состояния, 43  
 диаграмм деятельности, 47  
 Диаграмма вариантов использования, 36  
 Диаграмма классов, 41  
 диаграмма размещения, 51  
 Диаграмма размещения, 50  
 диаграммы вариантов использования, 36, 75  
 диаграммы вариантов использования,  
 моделирующие бизнес-процессы, 36  
 диаграммы взаимодействия, 36, 67, 69  
*Диаграммы взаимодействия*, 37  
 диаграммы деятельности, 36, 48, 64, 65, 67,  
 69, 75, 78  
 Диаграммы деятельности, 47, 50  
 диаграммы классов, моделирующие  
 статическую структуру классов системы  
 и связи между классами, 35  
 Диаграммы компонентов, 50  
 диаграммы компонентов, моделирующие  
 иерархии компонентов, 35  
 диаграммы обзора взаимодействия, 36  
 Диаграммы пакетов, 41  
 диаграммы последовательности, 36, 37, 38,  
 39  
*Диаграммы последовательности*, 37, 87  
 диаграммы последовательности могут  
 содержать блоки разных типов, 38  
 диаграммы размещения, моделирующие  
 физическую архитектуру системы, 35  
 диаграммы составных структур, 36

- диаграммы состояний, 36, 67, 69
- Диаграммы состояний, 42, 47
- Жизненный цикл ПО*, 11
- жирные горизонтальные линии, 47
- ЖЦ, 11, 12, 13, 14, 15, 16, 17, 23
- Зависимость*, 32
- Зависимость между классами, 33
- Идентификация основных абстракций*, 83
- иерархия, 20, 21, 23, 24
- Иерархия*, 24
- именованные значения*, 35, 51
- Индивидуальность*, 25
- Инженерия ПО*, 10
- инкапсулировать, 20
- инкапсуляция, 23, 24
- Инкапсуляция*, 24
- Интерфейс, 26, 83
- исполняемые компоненты, 50
- Использование базы данных требований для управления проектом, 71
- историческое состояние, 45
- исходящего перехода, 48
- итерации во взаимодействии, 40
- итерационных моделей*, 16, 17
- канал взаимодействия узлов, 50
- каскадной модели*, 14, 15, 16
- Квалификация механизмов анализа*, 86, 94
- Класс*, 25, 31, 41, 89, 90
- класс-сущность, 51
- Класс-сущность, 41
- классы системы, 41
- ключевых абстракций, 80, 83
- Коберну, 72
- когда вызывается операция объекта, 42
- когда наступает заданный момент времени, 42
- когда становится истинным условие, 42
- Коммуникации между вариантами использования, 37
- коммуникационные диаграммы, 36, 37, 39
- компиляцию и сборку системы, 50
- композитными состояниями, 45
- Композиция*, 28
- Компонент*, 27
- компонент исходного кода, 50
- Компонентная разработка, 27
- компоненты ПО, 50
- конечный автомат, 42
- контроллер, 41, 89, 92, 93
- кооперативные диаграммы, 87
- кооперативными, 37, 39
- косая черта, 44
- Краткая история ООП, 23
- кризис ПО*, 9
- критический участок, 38
- крупномасштабные проекты*, 9
- курсор управления, 48
- курсоры объектов, 48
- легковесный процесс*, 18
- линейки синхронизации, 47
- линия жизни, 37
- логически заверченный набор взаимосвязанных и взаимодействующих видов деятельности, 61
- логическим умножением, 46
- локализация вносимых изменений, 23
- локализация свойств и поведения, 24
- Малое программное обеспечение*, 8
- Манифест гибкой разработки, 18
- маркер, 48
- менеджером проекта, 50
- метамоделью, 51
- Механизмы расширения UML, 51
- Минимальные гарантии, 72
- Мифический человеко-месяц, 10, 19
- множества, 32, 90
- множественное наследование, 33
- модели бизнес-процессов, 22, 64
- модели деятельности организации, 22
- Модели ЖЦ, 14
- Моделирование*, 22, 61
- моделировании встроенных систем, 51
- модель «4+1», 21
- модель бизнес-анализа, 62, 63
- Модель бизнес-анализа, 65, 66, 69
- модель бизнес-объектов, 65
- модель бизнес-процессов, 62
- Модель бизнес-процессов, 63, 65
- Модель бизнес-целей, 63
- Модель ЖЦ*, 13
- Модель ПО*, 21
- модульность, 23
- Модульность*, 24
- мозговой штурм, 71
- Мощность*, 29
- мультимножества, 32
- Направление*, 29
- Наследование*, 33
- наследования, 33, 37
- начало потока управления, 47
- Начальная точка, 48
- начальное состояние, 43
- небольшие проекты*, 9
- неверное взаимодействие, 38
- необязательный блок, 38
- непрерываемое, 43

- несколько альтернатив, 38
- Нефункциональные требования, 71
- низкая связанность, 20, 90
- новые типы диаграмм, 36
- Оберон, 25
- обеспечивающие процессы, 61
- обмен сообщениями между объектами, 36
- Обобщение*, 33, 75
- Объект*, 25, 26, 43, 92, 93
- Объектная модель, 23
- объектной модели*, 25, 27, 30, 33
- объектному узлу, 47
- Объектный язык ограничений, 52
- Ограничение*, 52
- ожидание курсоров на всех входах, 48
- ООП, 23, 27
- ОО-система, 26
- Операции доступа, 26
- Операции реализации, 26
- Операции управления, 26
- Операция*, 26, 58, 60
- Описание бизнес-правил, 68
- описание физического размещения компонент, 21
- Определение типов требований, 71
- организации элементов в группы, 27
- основные процессы, 61
- Особенности современных проектов ПО, 10
- осязаемая сущность, 25
- отложенным, 45
- отображают зависимости между пакетами, 41
- оценка организации, 68
- оценку организации, 62
- очереди событий, 44
- Пакет*, 27, 41
- параллелизм, 21, 23
- Параллелизм, 25
- параллельно, 11, 38, 47
- параллельные области, 46
- Первичный сбор требований, 71
- перемещение курсора по ребру, 48
- переход из состояния в само себя, 43
- переходные псевдосостояния, 46
- Переходом, 43
- ПО, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 34, 35, 36, 37, 51, 61, 72, 77, 79, 81, 84
- Поведение*, 25, 92
- поведенческие, 36, 92
- Повторное использование, 23
- Подсистема*, 27, 41
- подсостояние, 45
- полиморфизм, 25, 33
- Полиморфизм*, 26
- пользовательский интерфейс, 87
- последовательности, 14, 32, 40, 62, 67, 87
- последовательность во времени, 39
- Поток объектов, 47
- поток событий, 36, 39, 73, 87, 90
- потоки объектов, 47
- потоки управления, 21, 22, 36, 47, 49
- правила вывода и вычислительные правила, 68
- Правила написания сценариев, 73
- правила-ограничения, 68
- предметной области, 23, 24, 26, 82, 83, 84, 86, 91, 92, 94
- представление данных, 41
- представление логической организации, 21
- представление о поведении системы с точки зрения пользователя, 36
- представление структуры потоков управления и аспектов параллельной работы, 21
- представление физической структуры программных компонент, 21
- представление функциональных возможностей, 21
- предыдущему активному подсостоянию, 45
- прерываемое поведение, 43
- При выполнении внутреннего перехода, 43
- Примечание*, 52
- принцип подстановки Лисковой, 33, 60
- Принципы быстрой разработки, 18
- проблема сложности, 20
- проведение совещаний, 71
- Программная инженерия, 11
- Программное обеспечение*, 8
- Проект ПО*, 8
- Проектирование*, 8, 14, 19, 34, 70, 79, 81, 95
- Проектирование ПО*, 8
- производственный процесс*, 61
- Процесс аттестации*, 13
- Процесс аудита*, 13
- Процесс верификации*, 13
- Процесс документирования*, 12
- Процесс обеспечения качества*, 12
- Процесс обучения*, 13
- Процесс поставки*, 12
- Процесс приобретения*, 11
- Процесс разработки*, 12
- Процесс разрешения проблем*, 13
- Процесс совместной оценки*, 13
- Процесс создания инфраструктуры*, 13
- Процесс сопровождения*, 12

- Процесс управления*, 12, 13  
*Процесс управления конфигурацией*, 12  
*Процесс усовершенствования*, 13  
*Процесс эксплуатации*, 12  
 процессы управления, 61  
 псевдосоединением – дугой над объектом, 40  
 псевдосостояния выбора, 46  
 public, 25  
 развертывание, 12, 14  
 разделяй и властвуй, 20  
 размещение объектов и компонентов в распределенной системе, 50  
 раскадровка, 71  
 распараллеливание, 48  
 реактивируется, 45  
*Реализация*, 14, 26, 34  
 Реализация бизнес-процесса, 69  
 регулярные выражения, 51  
 Рефлексивные сообщения, 40  
 ролевые игры, 71  
 Роль, возложенная на класс изображается, 41  
 Руководящие указания по проектированию, 82  
 Связи коммуникации, 37  
*Связи между классами*, 94  
 связи микропроцессоров, 51  
 связи расширения, 37  
 связь «тип – подтип», 33  
 связь включения, 37  
 связь коммуникации, 37  
 связь между вариантом использования и действующим лицом, 37  
 связь между классами, 28  
 связь между объектами, 27  
 Связь между подсистемой и интерфейсом, 41  
 связь обобщения, 37, 41  
 связью реализации, 41  
 Семантический разрыв, 23  
 семантическое ограничение, 52  
 сессии по выявлению требований, 71  
 сетях Петри, 47  
 сильная связность, 20  
 Синхронизация, 48  
*Система*, 8, 72, 73  
 системный интерфейс, 87  
*Системный подход*, 8  
 системными инженерами, 50  
 слабое зацепление, 20  
 словарь основных терминов организации, 62  
 Словарь предметной области, 72  
 сложность управления, 20  
 смена одного состояния объекта на другое, 43  
 Смена текущего состояния, 42  
 собеседование, 71  
 событие времени, 42  
 событие вызова, 42, 44  
 событие изменения, 42  
*Соглашения моделирования*, 82  
 соединение, 26, 50  
*Соединение*, 27  
*соединения* между объектами, 26  
 создание и демонстрация работающих прототипов, 71  
*Сопровождение*, 14  
*Состояние объекта*, 25  
 Спецификация требований к ПО, 70  
 спиральной модели, 17  
*средние проекты*, 9  
 ссылка на другую диаграмму, 38  
*Стадия ЖЦ*, 13  
*Стереотип*, 51  
 стереотипов, 41, 51, 63  
 стереотипы, 35, 51  
 Стереотипы классов, 51  
 сторожевое условие, 37, 38, 44, 45, 48  
 строгой типизации, 25  
 структурные, 21, 35, 66, 68, 92  
 суперсостояния, 45  
 сценарий варианта использования, 36  
*Теги*, 51  
 Текущая версия UML, 35  
 текущее состояние дел в организации, 62  
*тестирования*, 13, 14  
 техническая сложность, 20  
 технологический процесс анализа и проектирования, 79  
 типизация, 23  
 Типизация, 24  
 Типичные ошибки в сценариях, 74  
 требования к интерфейсу, 71  
 требования к надежности, 71  
 требования к применению, 71  
 требования к производительности, 71  
 требования к реализации, 71  
 тяжелые процессы, 17  
 узел, 47, 48, 50  
 узел действия, 47, 48  
 Узел действия, 47  
 узел объединения, 47, 48  
 узел объединения потоков, 47  
 узел принятия решения, 47  
 узел разветвления, 48  
*Узел разветвления*, 47

- узел разделения, 48
- узлом слияния, 48
- узлы разделения*, 47
- Узлы разделения и объединения, 49
- узлы слияния*, 47
- Унификация классов анализа*, 86, 95
- упорядоченные множества, 32
- управляющий класс, 51
- устойчивость, 23, 25
- Устойчивость, 25, 83
- физические взаимосвязи между
  - программными и аппаратными компонентами системы, 50
- физический уровень системы, 50
- физическое размещение системы, 50
- Финальное состояние, 43
- финальный узел, 48
- Финальный узел*, 47
- формализованное описание бизнес-процессов предприятия, 61
- формальных преобразованиях*, 14, 15, 16
- Формат описания варианта использования, 72
- формирования требований*, 14, 61
- Формулировки бизнес-правил, 68
- Функциональные требования, 71, 72
- Хороший набор требований, 71
- цели бизнес-моделирования, 62
- Цели бизнес-моделирования, 61
- Целью объектно-ориентированного анализа, 81
- цикл (пока истинно условие), 38
- черной точкой, 43
- черной точкой в белом кружке, 43
- черный кружок, 46
- черный ящик, 24, 62
- эксплуатационным персоналом, 50
- Эксплуатация*, 14
- элементарную единицу работы, 47
- Эскизная архитектура, 80

## Конспект лекций по курсу «Объектно-ориентированный анализ и проектирование»

### Лекция 1. Основы программной инженерии

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.

*Программное обеспечение* – это система, включающая в себя: компьютерные программы; документацию; данные, необходимые для корректной работы программ.

*Проектирование ПО* – это процесс создания спецификаций ПО на основе исходных требований к нему.

*Проект ПО* – совокупность спецификаций ПО (включающих модели и проектную документацию), обеспечивающих создание ПО в конкретной программно-технической среде.

ПО можно разбить на два класса: «малое» и «большое».

«Малое» программное обеспечение имеет следующие характеристики:

- решает одну несложную, четко поставленную задачу;
- размер исходного кода не превышает нескольких сотен строк;
- скорость работы программного обеспечения и необходимые ему ресурсы не играют большой роли;
- ущерб от неправильной работы не имеет большого значения;
- модернизация программного обеспечения, дополнение его возможностей требуется редко;
- как правило, разрабатывается одним программистом или небольшой группой (5 или менее человек);
- подробная документация не требуется, ее может заменить исходный код, который доступен.

«Большое» программное обеспечение имеет 2-3 или более характеристик из следующего перечня:

- решает совокупность взаимосвязанных задач;
- использование приносит значимую выгоду;
- удобство его использования играет важную роль;
- обязательно наличие полной и понятной документации;
- низкая скорость работы приводит к потерям;
- сбои, неправильная работа, наносит ощутимый ущерб;
- программы в составе ПО во время работы взаимодействуют с другими программами и программно-аппаратными комплексами;
- работает на разных платформах;
- требуется развитие, исправление ошибок, добавление новых возможностей;
- группа разработчиков состоит из более 5 человек.

Далее рассматривается проектирование «большого» ПО, поскольку создание «малого» не вызывает больших трудностей, не требует специальной технологии и инструментов.

Классификация программных проектов по размеру группы разработчиков и длительности проекта:

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

С конца 60-х годов прошлого века до сегодняшних дней продолжается так называемый «кризис ПО». Выражается он в том, что большие проекты выполняются с превышением сметы расходов и/или сроков отведенных на разработку, а разработанное ПО не обладает требуемыми функциональными возможностями, имеет низкую производительность и качество. По результатам исследований американской индустрии разработки ПО, выполненных в 1995 году Standish Group ([www.standishgroup.com](http://www.standishgroup.com)), только 16% проектов завершились в срок, не превысили запланированный бюджет и реализовали все требуемые функции и возможности. 53% проектов завершились с опозданием, расходы превысили запланированный бюджет, требуемые функции не были реализованы в полном объеме. 31% проектов были аннулированы до завершения. Для двух последних категорий проектов бюджет среднего проекта оказался превышенным на 89%, а срок выполнения – на 122%. В последние годы процентное соотношение трех перечисленных категорий проектов незначительно изменяется в лучшую сторону.

1995	1	1	2	2	2	2009				
	996	998	000	004	006					
31% отменены	0%	4	8%	2	3%	8%	9%	1	1	24%
53% ущербны	3%	3	6%	4	9%	4	3%	5	4	44%
16% успешны	7%	2	6%	2	8%	2	9%	2	3	32%

Причины неудач:

- нечеткая и неполная формулировка требований;
- недостаточное вовлечение пользователей в работу над проектом;
- отсутствие необходимых ресурсов;
- неудовлетворительное планирование и отсутствие грамотного управления проектом;
- частое изменение требований и спецификаций;
- новизна и несовершенство используемой технологии;
- недостаточная поддержка со стороны высшего руководства;
- недостаточно высокая квалификация разработчиков, отсутствие необходимого опыта.

При планировании проектов зачастую по тем или иным причинам устанавливаются невыполнимые сроки, закладываются недостаточные ресурсы. Таким образом, возникают *безнадежные проекты* (death march projects)<sup>1</sup>. Признаки безнадежного проекта:

- план проекта сжат более чем наполовину по сравнению с нормальным расчетным планом;
- количество разработчиков уменьшено более чем наполовину по сравнению с действительно необходимым для проекта данного размера и масштаба;
- бюджет и связанные с ним ресурсы урезаны наполовину;
- требования к функциям, производительности и другим характеристикам вдвое превышают

<sup>1</sup> Понятие безнадежного проекта введено Эдвардом Йорданом. См. *Эдвард Йордон. Путь камикадзе*. 2-е изд. – М.: Лори, 2004

значения, которые они могли бы иметь в нормальных условиях. Другой причиной неверного планирования является заблуждение относительно производительности проектировщиков. В большом проекте общая производительность группы разработчиков не равна сумме производительностей отдельных членов группы (посчитанной как если бы они работали в одиночку). Об этом в книге «Мифический человеко-месяц»<sup>2</sup> пишет Фредерик Брукс. Выводы Брукса:

- самая частая причина провала – нехватка времени;
- иногда работы нельзя ускорить, не испортив результат;
- человеко-месяц – опасное заблуждение, поскольку предполагается, что количество людей и месяцы можно поменять местами;
- разделение задачи между несколькими людьми вызывает накладные затраты;
- если проект не укладывается в срок, то добавление людей задержит его еще больше;
- «серебряной пули» нет!

Последнее положение касается технологии разработки. Брукс утверждает, что технологии, позволяющей на порядок повысить производительность разработчиков, не существует. То есть, нельзя полагать, что какая-либо новейшая технология разработки позволит осуществить проект в 10 раз быстрее.

Особенности современных проектов ПО:

- 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.

Разработка ПО имеет следующие специфические особенности:

- 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;
- «неощутимость», «воздушность» ПО, что подталкивает к безответственному переделыванию, поскольку легко стереть и переписать, чего не сделаешь при проектировании зданий и аппаратуры.

Путем выхода из кризиса ПО стало создание программной инженерии (software engineering).

*Инженерия ПО* (software engineering) – совокупность инженерных методов и средств создания ПО. 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.

Этапы становления и развития SE:

---

<sup>2</sup> Брукс Ф. Мифический человеко-месяц или как создаются программные системы. – СПб.: Символ-Плюс, 1999

- 70-е и 80-е годы – систематизация и стандартизация процессов создания ПО (структурный подход)
- 90-е годы – начало перехода к сборочному, индустриальному способу создания ПО (объектно-ориентированный подход)

Программная инженерия применяется для удовлетворения требований заказчика ПО. Основные цели программной инженерии:

- Системы должны создаваться в короткие сроки и соответствовать требованиям заказчика на момент внедрения.
- Качество ПО должно быть высоким.
- Разработка ПО должна быть осуществлена в рамках выделенного бюджета.
- Системы должны работать на оборудовании заказчика, а также взаимодействовать с имеющимся ПО.
- Системы должны быть легко сопровождаемыми и масштабируемыми.

Основным понятием программной инженерии является понятие *жизненного цикла ПО*.

*Жизненный цикл ПО (software lifecycle)* – это период времени, который начинается с момента принятия решения о необходимости создания ПО и заканчивается в момент его полного изъятия из эксплуатации.



Основной нормативный документ, регламентирующий ЖЦ ПО – стандарт ISO/IEC 12207: 1995 “Information Technology – Software Life Cycle Processes” (ГОСТ Р ИСО/МЭК 12207-99). В рамках технологий создания ПО понятие ЖЦ уточняется, но указанные стандарты не нарушаются.

С точки зрения статической структуры ЖЦ является совокупностью процессов ЖЦ.

*Процесс ЖЦ* – набор взаимосвязанных действий, преобразующих некоторые входные данные и ресурсы в выходные.

Каждый процесс характеризуется задачами, методами их решения, действующими лицами, результатами. Процессы ЖЦ протекают параллельно. Каждый процесс разделен на набор действий, каждое действие – на набор задач. Каждый процесс, действие или задача инициируется и выполняется по мере необходимости, причем не существует заранее определенных последовательностей выполнения.

- основные (приобретение, поставка, разработка, эксплуатация, сопровождение);
- вспомогательные (документирование, управление конфигурацией, обеспечение качества, верификация, аттестация, совместная оценка, аудит, разрешение проблем);
- организационные (управление, создание инфраструктуры, усовершенствование, обучение).

Для ознакомления приведем содержание процессов ЖЦ.

*Процесс приобретения* включает следующие действия: инициирование приобретения; подготовку заявочных предложений; подготовку и корректировку договора; надзор за деятельностью поставщика; приемку и завершение работ. Действующие лица: заказчик, поставщик. Задачи приобретения: определение заказчиком своих потребностей в ПО; анализ требований к ПО; принятие решения о приобретении ПО; выработка плана приобретения и

заявочных предложений; выбор поставщика; подготовка и заключение договора с поставщиком; контроль за соблюдением условий договора; корректировка договора при необходимости.

*Процесс поставки* включает в себя следующие действия: инициирование поставки; подготовку ответа на заявочные предложения; подготовку договора; планирование и выполнение поставки; контроль поставки; проверку и оценку. Действующие лица: заказчик, поставщик. Задачи поставки: оценка поставщиком заявочных предложений; подготовки и заключение договора с заказчиком, контроль со стороны поставщика за соблюдением условий договора, принятие решения о привлечении субподрядчика или выполнении работ своими силами, выработка плана управления проектом и др.

*Процесс разработки* включает в себя следующие действия: подготовительную работу; анализ требований к ПО; проектирование архитектуры ПО; детальное проектирование ПО; кодирование ПО; тестирование ПО; интеграцию ПО; установку ПО; приемку ПО. Действующие лица: разработчик, заказчик. Задачи разработки: выбор модели ЖЦ ПО и согласование с заказчиком; определение требований к ПО (функциональных и нефункциональных); определение состава компонентов ПО и создание документации по каждому компоненту; моделирование и спецификация компонент ПО; планирование интеграции компонент; создание исходных текстов компонент; поиск и исправление ошибок в исходных текстах и документации; сборка ПО; развертывание ПО; оценка результатов.

*Процесс эксплуатации* включает в себя следующие действия: подготовительную работу; эксплуатационное тестирование; эксплуатацию; поддержку пользователей. Действующие лица: оператор (организация, эксплуатирующая ПО), пользователи. Задачи эксплуатации: выработка плана эксплуатации и эксплуатационных стандартов; составление процедур локализации и разрешения проблем эксплуатации; поиск ошибок в ПО перед вводом в эксплуатацию его новых версий; оказание помощи пользователям и консультирование.

*Процесс сопровождения* включает в себя следующие действия: подготовительную работу; анализ проблем и запросов на модификацию ПО; проверку и приемку; перенос ПО в другую среду; снятие ПО с эксплуатации. Действующие лица: служба сопровождения, пользователи. Задачи сопровождения: выработка плана сопровождения; составление процедур локализации и разрешения проблем сопровождения; оценка целесообразности внесения модификаций в ПО; принятие решения о модификации; поиск ошибок в ПО после его модификации; проверка целостности ПО; архивирование при снятии с эксплуатации; обучение пользователей.

*Процесс документирования* включает в себя следующие действия: подготовительную работу; проектирование и разработку документации; выпуск документации; сопровождение.

*Процесс управления конфигурацией* в себя следующие действия: подготовительную работу; создание базы знаний о ПО (конфигурации); контроль за конфигурацией; учет состояния конфигурации; оценку конфигурации; управление выпуском и поставку ПО. *Конфигурация ПО* – это совокупность сведений о его функциональных и физических характеристиках на всех стадиях ЖЦ ПО. Основная задача управления конфигурацией: организация, систематический учет и контроль внесения изменений в ПО.

*Процесс обеспечения качества* включает в себя следующие действия: подготовительную работу; обеспечение качества продукта; обеспечение качества процесса; обеспечение других показателей качества ПО. Задачи обеспечения качества: гарантированное соответствие ПО требованиям

заказчика, зафиксированным в договоре; гарантированное соответствие процессов ЖЦ ПО, методов разработки, квалификации персонала установленным стандартам.

*Процесс верификации* включает в себя следующие действия: подготовительную работу; верификацию. Основная задача верификации – проверка соответствия разработанных программ в составе ПО их спецификациям.

*Процесс аттестации* состоит в определении полноты соответствия разработанного ПО требованиям заказчика. Основная задача аттестации – оценка достоверности тестирования ПО. Как правило, для аттестации привлекают независимых экспертов.

*Процесс совместной оценки* включает в себя следующие действия: подготовительную работу; оценку управления проектом; техническую оценку. Основная задача совместной оценки – контроль планирования и управления ресурсами, персоналом, инфраструктурой проекта.

*Процесс аудита* состоит в определении полноты соответствия проекта условиям договора.

*Процесс разрешения проблем* предусматривает анализ и разрешение проблем, возникающих в течение ЖЦ ПО.

*Процесс управления* включает в себя следующие действия: подготовительную работу; планирование; выполнение и контроль; проверку и оценку; завершение. Задачи управления: проверка достаточности имеющихся ресурсов; составление графиков работ; оценка затрат; выделение ресурсов; распределение ответственности; оценка рисков.

*Процесс создания инфраструктуры* состоит в выборе и поддержке технологии разработки ПО, стандартов и инструментальных средств; выборе и установке аппаратных и программных средств, необходимых для разработки, эксплуатации и сопровождения ПО.

*Процесс усовершенствования* предусматривает оценку, измерение, контроль и усовершенствование процессов ЖЦ ПО. Основная задача усовершенствования – повышение производительности труда.

*Процесс обучения* включает в себя следующие действия: подготовительную работу; разработку учебных планов, курсов, материалов; реализацию планов обучения. Задачи обучения: первоначальное обучение персонала; повышение квалификации персонала.

Процессы ЖЦ ПО взаимосвязаны.

Динамику, т. е. развитие ЖЦ во времени определяет *модель жизненного цикла*.

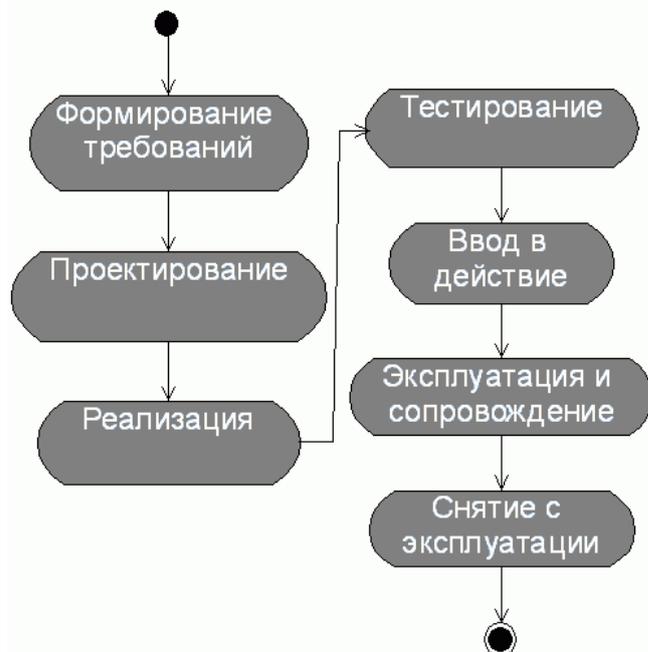
*Модель ЖЦ ПО* – это структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач на протяжении всего ЖЦ.

В любой модели ЖЦ рассматривается как совокупность стадий ЖЦ.

*Стадия ЖЦ* – это часть ЖЦ ограниченная временными рамками, по завершении которой достигается определенный важный результат в соответствии с требованиями для данной стадии ЖЦ.

## Модели ЖЦ:

- каскадная (водопадная);
- эволюционная;
- основанная на формальных преобразованиях;
- итерационные (пошаговая и спиральная).



### Схема каскадной модели ЖЦ:

Принципы *каскадной модели*: фиксация требований к системе в начале проекта; переход со стадии на стадию только после полного завершения работ на текущей стадии; недопустимость возврата на пройденные стадии; жесткая привязка процессов ЖЦ к стадиям ЖЦ.

Стадия *формирования требований* включает процессы, приводящие к созданию документа, описывающего поведение ПО с точки зрения внешнего по отношению к нему наблюдателя с фиксацией требований относительно его качества.

*Проектирование* охватывает процессы: разработку архитектуры ПО, разработку

структур программ в его составе и их детальную спецификацию.

*Реализация* или кодирование включает процессы создания текстов программ на языках программирования.

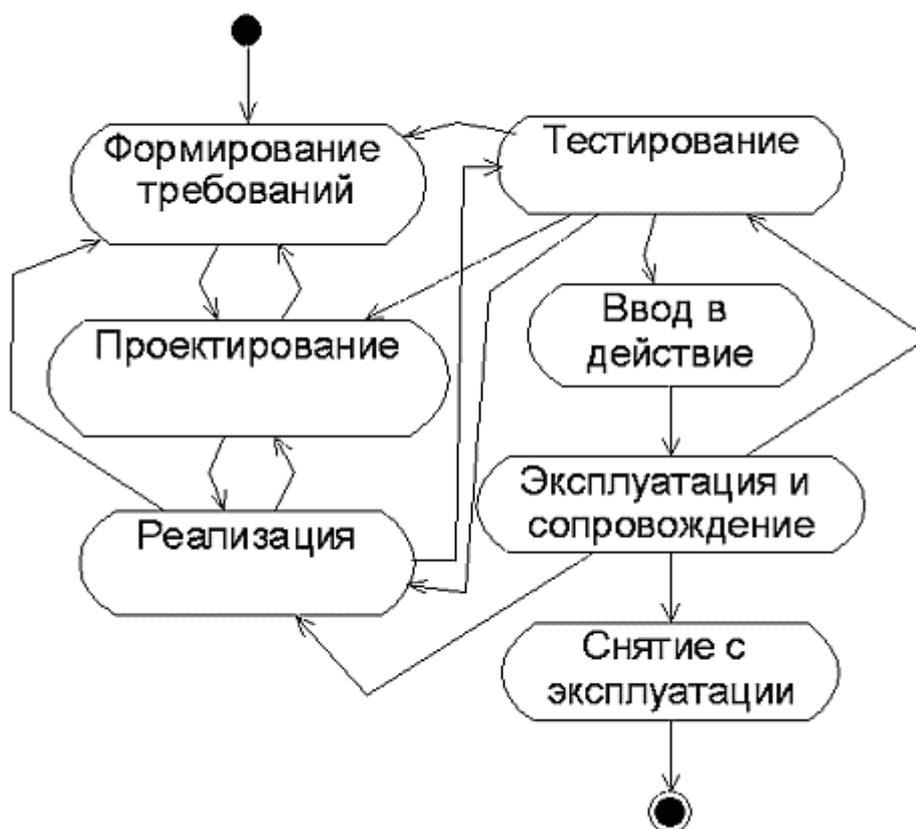
На этапе *тестирования* производится собственно тестирование, а также отладка и оценка качества ПО.

*Ввод в действие* – это развертывание ПО на целевой вычислительной системе, обучение пользователей и т.п.

*Эксплуатация ПО* – это использование ПО для решения практических задач на компьютере путем выполнения ее программ.

*Сопровождение ПО* – это процесс сбора информации о качестве ПО в эксплуатации, устранения обнаруженных в нем ошибок, его доработки и модификации, а также извещения пользователей о внесенных в него изменениях.

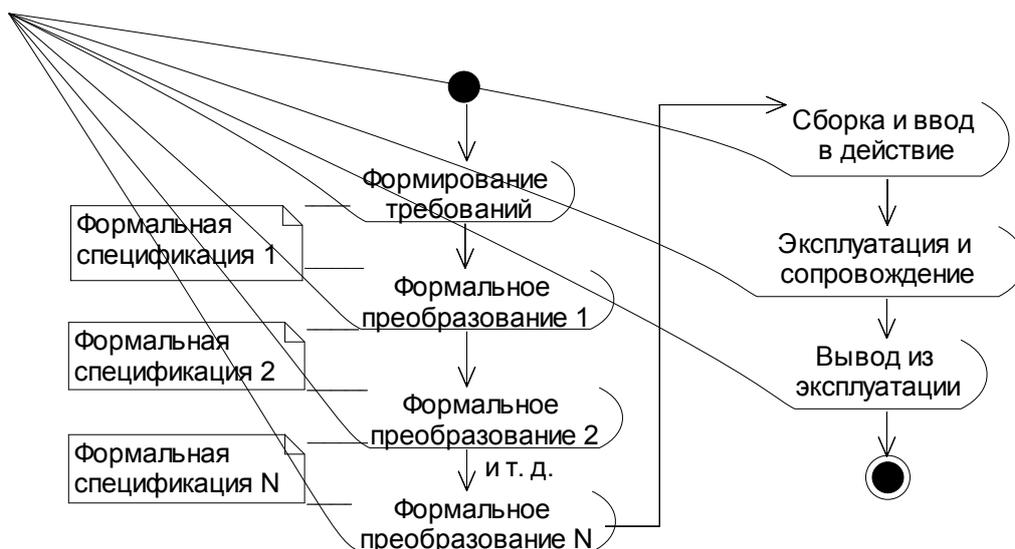
**Достоинства:** на каждой стадии формируется законченный набор проектной документации, отвечающий критериям полноты и согласованности; выполняемые в логичной последовательности стадии работ облегчают планирование сроков завершения всех работ и соответствующих затрат. **Недостатки:** позднее обнаружение проблем; выход из календарного графика, запаздывание с получением результатов; высокий риск создания системы, не удовлетворяющей изменившимся потребностям пользователей; избыточность документации; неравномерная нагрузка членов группы, работающей над проектом в ходе ЖЦ.



#### Неизбежные возвраты на предыдущие стадии в каскадной модели

На самом деле невозможно двигаться строго поступательно, необходимо возвращаться, чтобы исправлять ошибки, сделанные на ранних стадиях, устранять недоделки, учитывать меняющиеся в ходе проекта требования. В этом кроется причина недостатков водопадной модели.

Особенности модели ЖЦ, основанной на формальных преобразованиях: использование специальных нотаций для формального описания требований; кодирование и тестирование заменяется процессом преобразования формальной спецификации в исполняемую программу. Достоинства: формальные методы гарантируют соответствие ПО спецификациям требований к ПО, т. о. вопросы надежности и безопасности решаются сами собой. Недостатки: большие системы сложно описать формальными спецификациями; требуются специально подготовленные и высоко квалифицированные разработчики; есть зависимость от средств разработки и нотации спецификаций.



### Схема модели ЖЦ, основанной на формальных преобразованиях

Особенности *итерационных моделей*:

- разработка делится на несколько итераций, в рамках каждой из которых выполняются действия по созданию части системы (на разных итерациях части разные);
- действия не привязаны намертво к определенным стадиям ЖЦ, а выполняются по мере необходимости, повторяются, до тех пор, пока не будет получен нужный результат;
- с каждой пройденной итерацией ПО наращивается, в него интегрируются новые разработанные компоненты.

В итерационной пошаговой модели ЖЦ проект разбивается на несколько частей. Каждая часть создается в рамках отдельной итерации. Работы в рамках итерации ведутся по каскадной схеме. По окончании каждой итерации, начиная со второй, производится интеграция результатов работы на данной итерации с тем, что было сделано на предыдущих. Так преодолеваются недостатки каскадной модели.



### Схема пошаговой итерационной модели ЖЦ

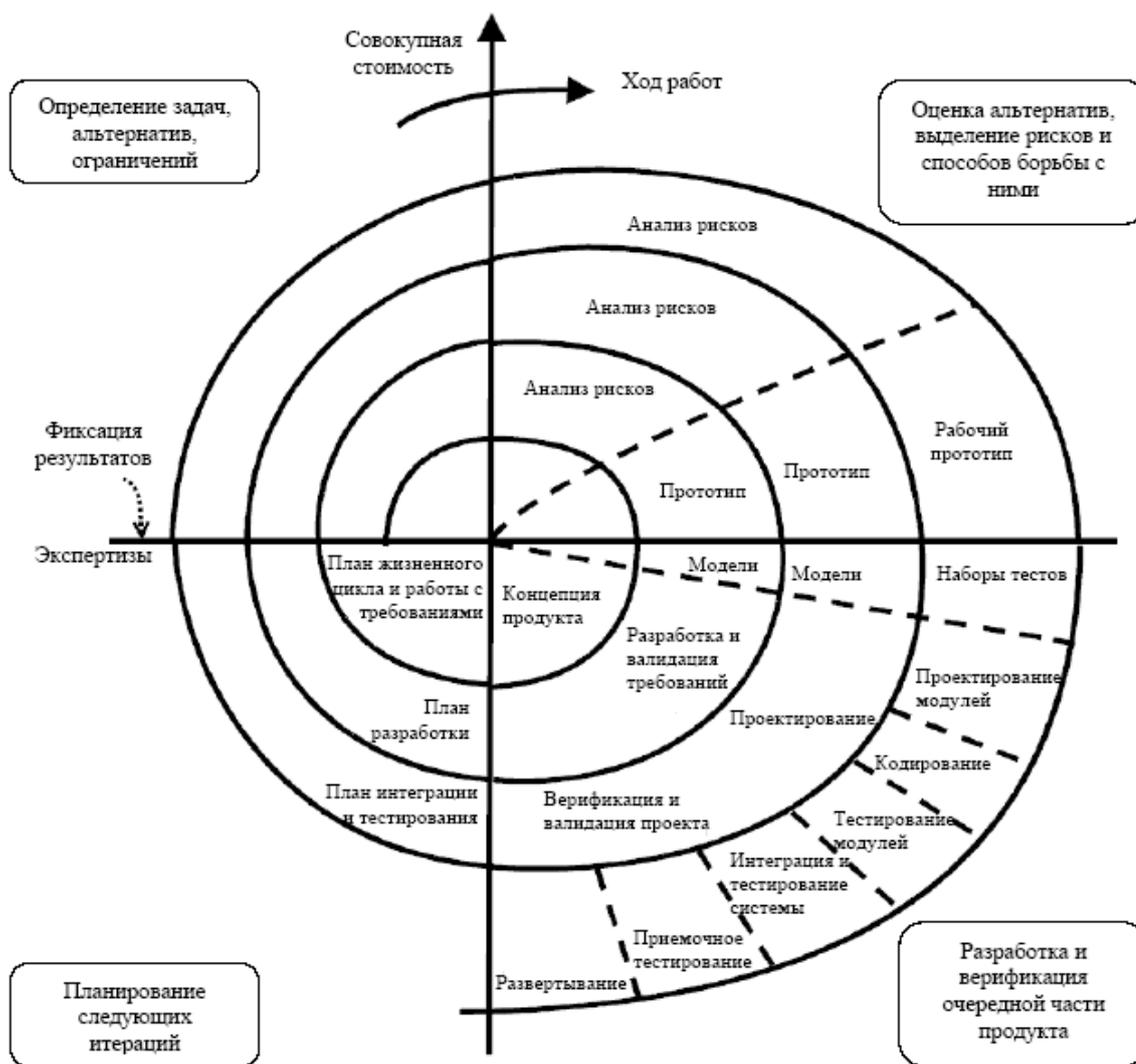


Схема спиральной модели ЖЦ

Особенности спиральной модели:

- общая структура действий на каждой итерации – планирование, определение задач, ограничений и вариантов решений, оценка предложенных решений и рисков, выполнение основных работ итерации и оценка их результатов;
- решение о начале новой итерации принимается на основе результатов предыдущей;
- досрочное прекращение проекта в случае обнаружения его нецелесообразности;
- в конце «миникасад», завершающийся выпуском финальной версии ПС.

Достоинства итерационных моделей:

- полный учет требований заказчика, большее его участие в проекте;
- равномерная нагрузка на группу разработчиков;
- раннее обнаружение проблем и их разрешение по мере возникновения;
- уменьшение рисков на каждой итерации.

Недостатки итерационных моделей: сложность планирования; плохая документированность создаваемого ПО.

Проблемой современной программной инженерии являются «тяжелые» процессы.

Характеристики «тяжелого» процесса:

- 1) необходимость документировать каждое действие разработчиков;

- 2) множество рабочих продуктов (в первую очередь - документов), создаваемых в бюрократической атмосфере;
- 3) отсутствие гибкости;
- 4) детерминированность (долгосрочное детальное планирование и предсказуемость всех видов деятельности, распределение человеческих ресурсов на длительный срок, охватывающий большую часть проекта).

Противоположностью «тяжелого» процесса является *«легковесный» процесс* – основа быстрой или гибкой разработки ПО (agile software development). Гибкая разработка ориентируется на эффективную коммуникацию между разработчиками, высокую квалификацию разработчиков и другие факторы, позволяющие сократить расходы на «бюрократию».

Манифест гибкой разработки:

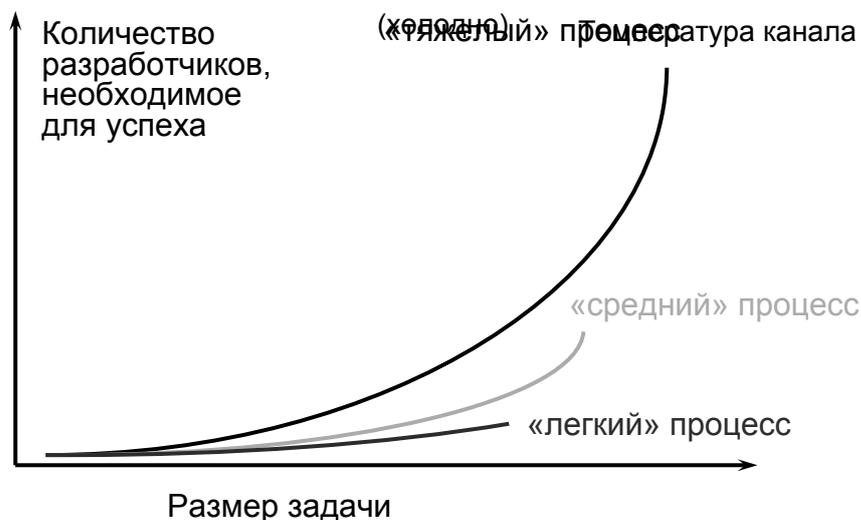
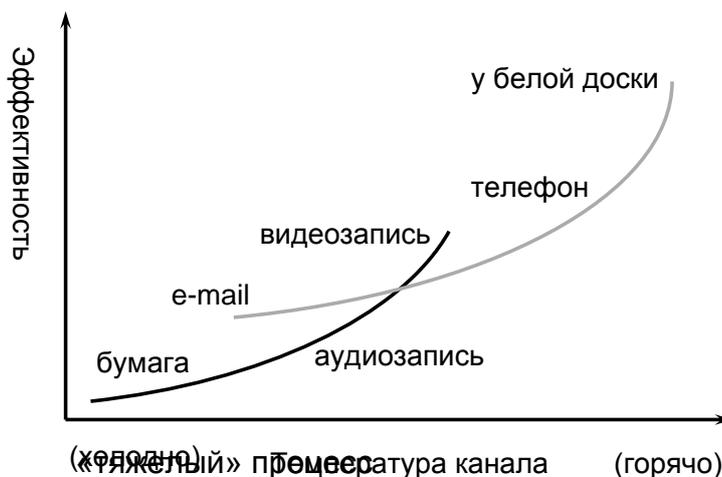
- Индивидуумы и взаимодействия ценятся выше процессов и инструментов.
- Работающее ПО ценится выше всеобъемлющей документации.
- Сотрудничество с заказчиками ценится выше согласования условий договора.
- Реагирование на изменения ценится выше соблюдения плана.

Важно, что стоящее в левой части не отменяет стоящего в правой (документация, процессы, инструменты планы важны, но в меньшей степени, чем индивидуумы, взаимодействия, работающий «софт» и реагирование на изменения).

Принципы быстрой разработки:

1. Диалог «лицом к лицу» – самый эффективный способ обмена информацией.
2. Избыточная «тяжесть» технологии (дополнительные рабочие продукты, планы, диаграммы, документы) стоит дорого.
3. Более многочисленные команды требуют более «тяжелых» технологий.

Добавим также, что возможности



«легких» процессов не безграничны. Для решения крупной задачи подойдет только «тяжелый» процесс.

4. Большая «тяжесть» процесса подходит для проектов с большей критичностью. Под критичностью понимаются масштабы последствий отказа разрабатываемого ПО. Уровни критичности:

- I) потеря удобства;
- II) потеря важных данных и/или рабочего времени;
- III) потеря невозместимых средств, дорогостоящего оборудования;
- IV) потеря человеческой жизни.

Примеры: I) данные выданы не в виде диаграммы, а в виде таблицы; II) «синий экран смерти»; III) взрыв ракеты «Ариан-5» 4 июня 1996 года (500 000 000\$) и массовое отключение электричества в Северной Америке 14 августа 2003 г. (4-10 Гига\$); IV) неправильная работа медицинской рентгеновской установки Therac-25 (3 смерти).

5. Возрастание обратной связи и коммуникации сокращает потребность в промежуточных продуктах.

Благодаря обратной связи с заказчиком и общению в команде можно оперативно получать сведения об изменениях требований, актуальных рисках, возникших проблемах, что дает возможность пересматривать план выпусков промежуточных релизов.

6. Дисциплина, умение и понимание противостоят процессу, формальности и документированию.

Процесс не заменит дисциплины разработчика, так как без нее работа будет вестись неэффективно. Известно понятие «забастовка по-итальянски», когда люди следуют букве инструкций, руководств, но фактически ничего не делают. Умение ценится выше формальности, так как отсутствующие навыки не могут быть заменены никаким справочным руководством по процессу. Наличие самой полной документации не равноценно пониманию. Знание в голове ценится выше, чем знания на бумаге.

7. Потеря эффективности в некритических видах деятельности вполне допустима.

В ходе разработки может возникнуть «бутылочное горлышко» – работа, скорость выполнения которой определяет скорость хода всего проекта. Сколько бы потов не сходило с разработчиков, не занятых на критическом участке, проект не ускорится. Поэтому разумно добиваться максимальной эффективности только там, где это действительно необходимо.

## Литература к лекции 1

1. Брукс Ф. Мифический человеко-месяц или как создаются программные системы. – СПб.: Символ-Плюс, 1999
2. Вендров А.М. Проектирование программного обеспечения экономических информационных систем: Учебник. – М.: Финансы и статистика, 2005
3. Коберн А. Быстрая разработка программного обеспечения. – М.: ЛОРИ, 2002
4. Соммервилл И. Инженерия программного обеспечения. 6-е издание.: Пер. с англ.: – М.: Вильямс, 2002
5. Жоголев Е. А. Технология программирования – М.: Научный мир, 2004
6. Кулямин В. В. Технологии программирования. Компонентный подход. – М.: Бином. Лаборатория знаний, 2007
7. Амблер С. Гибкие технологии: экстремальное программирование и унифицированный процесс разработки – СПб.: Питер, 2005

## **Лекция 2. МОДЕЛИ И ИХ РОЛЬ В СОЗДАНИИ СИСТЕМ. ОБЪЕКТНАЯ МОДЕЛЬ**

Одна из основных проблем при создании больших и сложных систем, в том числе ПО, – это проблема сложности. Виды сложности: техническая сложность и сложность управления.

Техническая сложность может быть вызвана:

- структурной сложностью (большим количеством элементов и сложными взаимосвязями между ними);
- отсутствием полных аналогов, ограничивающим возможность использования типовых проектных решений и прикладных систем;
- необходимостью интеграции существующих и вновь разрабатываемых приложений;
- функционированием в неоднородной среде на нескольких аппаратных платформах;
- высокими требованиями к надежности и производительности.

Сложность управления порождается следующими причинами:

- сильное воздействие внешней среды (политика, экономическая ситуация, контракты, много заинтересованных лиц, противоречивые требования);
- большой коллектив разработчиков, много различных проектов и продуктов;
- разобщенность и разнородность отдельных групп разработчиков по уровню квалификации и традициям использования инструментальных средств;
- значительная временная протяженность проекта.

Подход к решению этой проблемы основан на принципе «разделяй и властвуй» (divide et impera).

Сложная программная система должна быть разделена на небольшие подсистемы, каждую из которых можно разрабатывать независимо (в какой-то степени) от других. Декомпозиция является главным способом преодоления сложности разработки ПО. Принципы декомпозиции:

- количество связей между подсистемами должно быть минимальным («низкая связанность» или «слабое зацепление» – Low Coupling);
- степень взаимодействия внутри каждой подсистемы должна быть максимальной («сильная связанность» или «высокая прочность» – High Cohesion).

При разбиении системы на подсистемы необходимо выполнить следующие требования:

- каждая подсистема должна инкапсулировать свое содержимое (скрывать его от других подсистем);
- каждая подсистема должна иметь четко определенный интерфейс с другими подсистемами, устанавливающий стандартные ограничения на взаимодействие.

Следование этим правилам увеличивает понятность и модифицируемость создаваемого ПО.

Два основных подхода к декомпозиции систем: *функционально-модульный*, основанный на функциональной декомпозиции, при которой структура системы описывается в терминах иерархии ее функций и иерархии структур данных; *объектно-ориентированный*, использующий объектную декомпозицию, при которой структура системы описывается в терминах объектов и связей между ними, а поведение системы описывается в терминах обмена сообщениями между объектами. Подходы имеют много общего. Достоинством второго подхода является то, что есть единая иерархия, и нет необходимости отслеживать соответствие между двумя иерархиями функционально-модульного подхода.

В рамках обоих подходов используется понятие архитектуры ПО. *Архитектура ПО* – набор ключевых правил, определяющих организацию системы:

- совокупность структурных элементов системы и связей между ними;

- поведение элементов системы в процессе их взаимодействия;
- иерархия подсистем, объединяющих структурные элементы;
- архитектурный стиль (типовой способ организации системы).

Архитектура ПО многомерна, поскольку различные специалисты работают с её различными аспектами. Различные представления архитектуры служат различным целям (модель «4+1»):

- представление функциональных возможностей ПО (представление вариантов использования, содержащее сценарии взаимодействия системы с внешней средой и роли, которые играют пользователи ПО и внешние системы);
- представление логической организации ПО (логическое представление, элементами которого являются пакеты, подсистемы, классы, связи между ними);
- представление физической структуры программных компонент, входящих в состав ПО (представление реализации, элементами которого являются компоненты, связи между ними);
- представление структуры потоков управления и аспектов параллельной работы ПО (представление процессов, элементы которого: потоки управления, нити, параллелизм, синхронизация);
- описание физического размещения компонент ПО по узлам вычислительной системы (представление размещения, элементы которого: узлы вычислительной системы, устройства, линии связи, задачи).



Среди 5-ти представлений особое место занимает представление вариантов использования, поскольку оно используется при управлении разработкой, служит своего рода скелетом проекта.

Каждое *архитектурное представление* – это модель системы с определенной точки зрения, в которой отражены лишь существенные аспекты и опущено все, что несущественно при данном взгляде на систему.

*Модель ПО* – это формализованное описание системы ПО на определенном уровне абстракции. Каждая модель описывает конкретный аспект системы, использует набор диаграмм или формальных описаний и документов заданного формата, а также отражает точку зрения и является объектом деятельности различных людей с конкретными интересами, ролями или задачами. Модели служат полезным инструментом анализа проблем, обмена информацией

между всеми заинтересованными сторонами, проектирования ПО. Моделирование способствует более полному усвоению требований, улучшению качества системы и повышению степени ее управляемости.

Гради Буч:

*«Моделирование является центральным звеном всей деятельности по созданию качественного ПО. Модели строятся для того, чтобы понять и осмыслить структуру и поведение будущей системы, облегчить управление процессом ее создания и уменьшить возможный риск, а также документировать принимаемые проектные решения.»*

*Архитектурно значимый элемент* – это элемент, значительно влияющий на структуру системы, её функциональность, производительность, надежность, защищенность, возможность развития. Подсистемы, их интерфейсы, процессы и потоки управления являются архитектурно значимыми элементами.

Существуют различные графические модели, используемые при разработке ПО: блок-схемы, конечные автоматы, синтаксические диаграммы, семантические сети. Общее их достоинство графических моделей – наглядность.

Визуальное (графическое) моделирование – позволяет описывать проблемы с помощью зримых абстракций, воспроизводящих понятия и объекты реального мира. Моделирование осуществляется при помощи языка моделирования, который включает в себя: элементы модели; нотацию (систему обозначений); руководство по использованию.

Моделирование не является целью разработки ПО. Диаграммы – это лишь наглядные изображения. Причины, побуждающие прибегать к их использованию:

1. Графические модели помогают получить общее представление о системе, сказать о том, какого рода абстракции существуют в системе и какие ее части нуждаются в дальнейшем уточнении.
2. Графические модели образуют внешнее представление системы и объясняют, что эта система будет делать, тем самым облегчают общение с заказчиком.
3. Графические модели облегчают изучение методов проектирования, в частности объектно-ориентированных методов.

В процессе создания ПО используются следующие виды моделей:

- модели деятельности организации (или модели бизнес-процессов):
  - модели "AS-IS" ("как есть"), отражающие существующее положение;
  - модели "AS-TO-BE" ("как должно быть"), отражающие представление о новых процессах и технологиях работы организации;
- модели проектируемого ПО, которые строятся на основе модели "AS-TO-BE", уточняются и детализируются до необходимого уровня.

Состав моделей, используемых в каждом конкретном проекте, и степень их детальности в общем случае зависят от следующих факторов: сложности проектируемой системы; необходимой полноты ее описания; знаний и навыков участников проекта; времени, отведенного на проектирование.

Для облегчения труда разработчиков и автоматизированного выполнения некоторых рутинных действий используются *CASE-средства* (Computer Aided Software Engineering). В настоящее время CASE-средства обеспечивают поддержку большинства процессов жизненного цикла ПО, что

позволяет говорить о CASE-технологиях разработки ПО. *CASE-технология* – это совокупность методов проектирования ПО и инструментальных средств для моделирования предметной области, анализа моделей на всех стадиях ЖЦ ПО и разработки ПО.

Объектная модель является концептуальной базой объектно-ориентированного подхода (ООП).

Проблемы, стимулировавшие развитие ООП:

- Необходимость повышения производительности разработки за счет многократного (повторного) использования ПО.
- Необходимость упрощения сопровождения и модификации разработанных систем (локализация вносимых изменений).
- Облегчение проектирования систем (за счет сокращения семантического разрыва между структурой решаемых задач и структурой ПО).

Забегая вперед, скажем, какие решения данных проблем дает ООП. При ООП изменения локализуются внутри класса (компоненты или пакета, если изменяются несколько классов). Семантический разрыв ликвидируется, поскольку сущности предметной области представляются объектами, следовательно, разработчик и заказчик (пользователь) оперируют схожими понятиями. Повторное использование достигается за счет построения систем с использованием библиотек готовых компонент – модулей (заимствовано из структурного или функционального подхода).

Краткая история ООП:

- 1967: язык Simula – 1ый среди объектно ориентированных;
- 1970-е: Smalltalk – получил довольно широкое распространение;
- 1980-е: Теоретические основы, C++, Objective-C;
- 1990-е: Методы ООА и ООД (Booch, OMT, ....), появился язык Java;
- 1997: Принят стандарт OMG UML 1.1.

В основе объектно-ориентированного подхода лежит объектная декомпозиция, при этом статическая структура ПО описывается в терминах объектов и связей между ними, а динамический аспект ПО описывается в терминах обмена сообщениями между объектами. Каждый объект системы обладает своим собственным поведением, моделирующим поведение объекта реального мира.

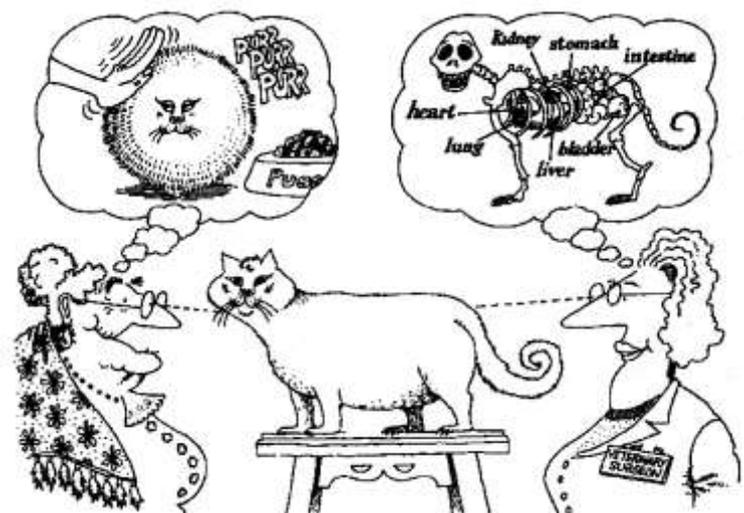
Объектная модель является естественным способом представления реального мира. Она является концептуальной основой ООП.

Основными принципами ее построения являются:

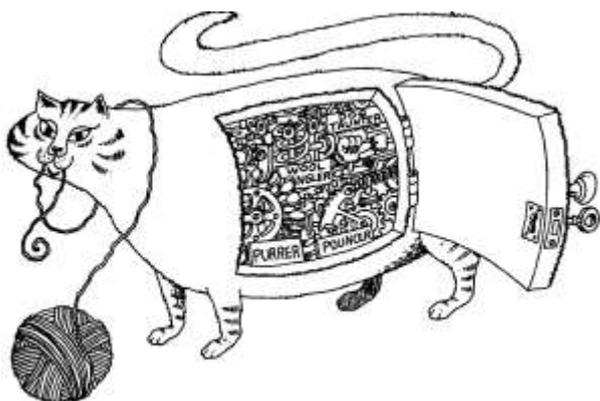
- абстрагирование;
- инкапсуляция;
- модульность;
- иерархия.

Дополнительные принципы:

- типизация;
- параллелизм;
- устойчивость (persistence).



*Абстрагирование* – это выделение наиболее существенных характеристик некоторого объекта, отличающих его от всех других видов объектов, важных с точки зрения дальнейшего рассмотрения и анализа, и игнорирование менее важных или незначительных деталей. Абстракцией является любая модель, включающая наиболее важные, существенные или отличительные характеристики некоторого объекта, и игнорирующая менее важные или незначительные детали. Абстрагирование позволяет управлять сложностью системы, концентрируясь на существенных свойствах объекта. Абстракция зависит от предметной области и точки зрения – то, что важно в одном контексте, может быть не важно в другом. Выбор правильного набора абстракций для заданной предметной области представляет собой главную задачу объектно-ориентированного проектирования. Объекты и классы – основные абстракции предметной области.



что внутренние ресурсы объекта, скрыты от внешней среды. Абстрагирование и инкапсуляция являются взаимодополняющими принципами.

*Модульность* – это свойство системы, связанное с возможностью ее декомпозиции на ряд внутренне сильно сцепленных, но слабо связанных между собой подсистем (частей).

Модульность снижает сложность системы, позволяя выполнять независимую разработку ее отдельных частей.

*Иерархия* – ранжированная или упорядоченная система абстракций, расположение их по уровням в виде древовидной структуры. Элементы, находящиеся на одном уровне иерархии, должны также находиться на одном уровне абстракции. Основными видами иерархических структур сложных систем являются структура классов и структура объектов. Иерархия классов строится по наследованию, а иерархия объектов – по агрегации.

*Тип* – точная характеристика некоторой совокупности однородных объектов, включающая структуру и поведение.

*Типизация* – способ защититься от использования

*Инкапсуляция* – локализация свойств и поведения в рамках единственной абстракции (рассматриваемой как «черный ящик»), скрывающей реализацию за общедоступным интерфейсом. При инкапсуляции отделяется внутреннее устройство объекта от его внешнего поведения. Объектный подход предполагает,

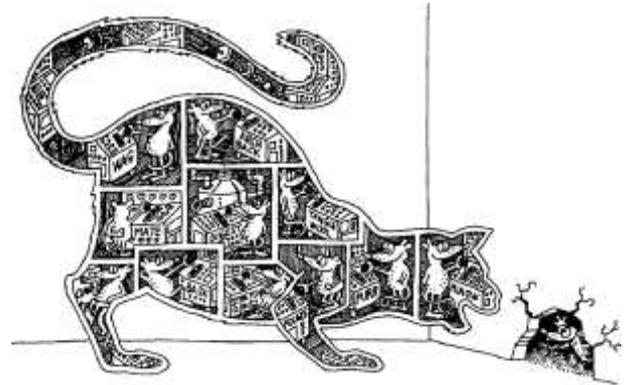


объектов одного класса вместо другого, или, по крайней мере, управлять таким использованием.

При строгой типизации (например, в языке Оберон) запрещается использование объектов неверного типа, требуется явное преобразование к нужному типу. При менее строгой типизации такого рода запреты ослаблены. В частности, допускается полиморфизм – многозначность имен. Одно из проявлений полиморфизма, использование объект подтипа (наследника) в роли объекта супертипа (предка).

Параллелизм – наличие в системе нескольких потоков управления одновременно. Объект может быть активен, т. е. может порождать отдельный поток управления. Различные объекты могут быть активны одновременно.

Устойчивость – способность объекта сохранять свое существование во времени и/или пространстве (адресном, в частности при перемещении между узлами вычислительной системы). В частности, устойчивость объектов может быть обеспечена за счет их хранения в базе данных.



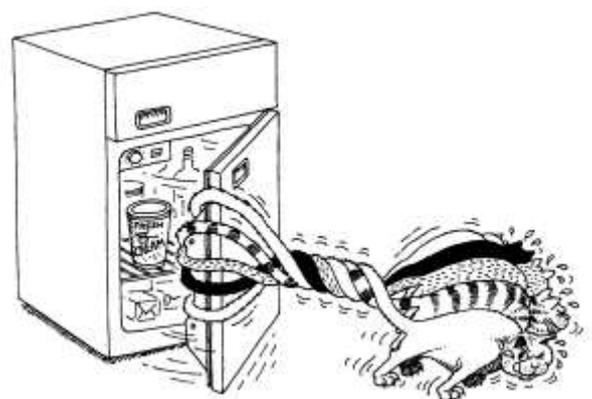
Переходим к основным понятиям объектно-ориентированного подхода (*элементам объектной модели*). К ним относятся: объект; класс; атрибут; операция; полиморфизм; наследование; компонент; пакет; подсистема; связь.

*Объект* – осязаемая сущность (tangible entity) – предмет или явление (процесс), имеющие четко выраженные границы, индивидуальность и поведение. Любой объект обладает состоянием, поведением и индивидуальностью. *Состояние объекта* определяется значениями его свойств (атрибутов) и связями с другими объектами, оно может меняться со временем. *Поведение* определяет действия объекта и его реакцию на запросы от других объектов. Поведение представляется с помощью набора сообщений, воспринимаемых объектом (операций, которые может выполнять объект). *Индивидуальность* – это свойства объекта, отличающие его от всех других объектов.

Структура и поведение схожих объектов определяют общий для них класс. *Класс* – это множество объектов, связанных общностью свойств, поведения, связей и семантики. Любой объект является экземпляром класса. Определение классов и объектов – одна из самых сложных задач объектно-ориентированного проектирования.

*Атрибут* – поименованное свойство класса, определяющее диапазон допустимых значений, которые могут принимать экземпляры данного свойства. Атрибуты могут быть скрыты от других классов, это определяет видимость атрибута: public (общий, открытый); private (закрытый, секретный); protected (защищенный).

Требуемое поведение системы реализуется через взаимодействие объектов. Взаимодействие объектов обеспечивается механизмом пересылки сообщений. Определенное воздействие одного объекта на



другой с целью вызвать соответствующую реакцию называется операцией или посылкой сообщения. Сообщение может быть послано только вдоль *соединения* между объектами. В терминах программирования *соединение* между объектами существует, если один объект имеет ссылку на другой.

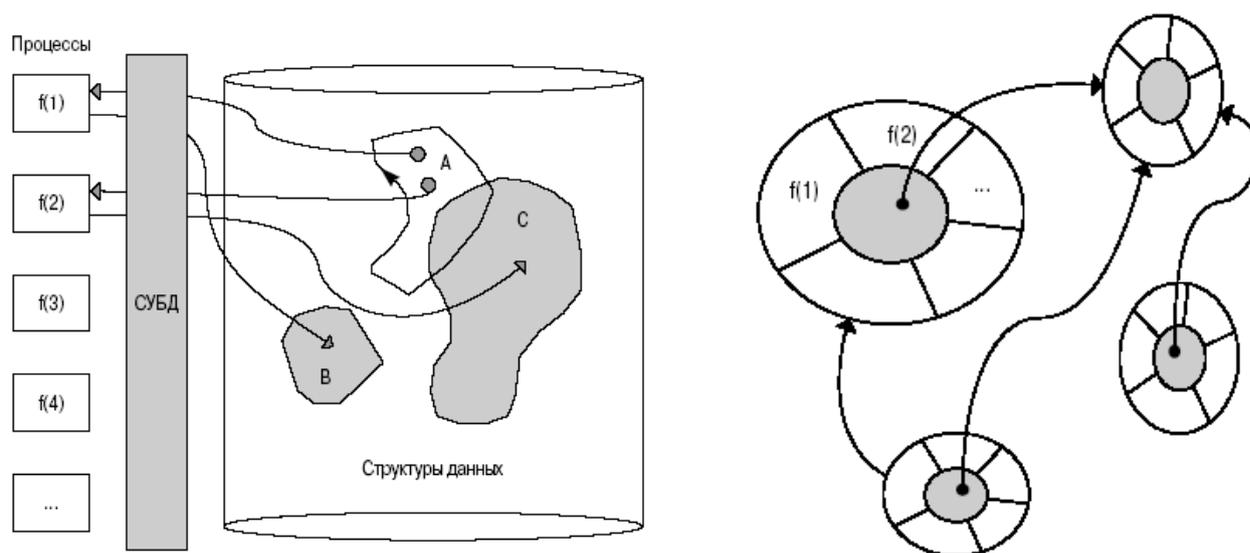
*Операция* – это услуга, которую можно запросить у любого объекта данного класса. Операции реализуют поведение экземпляров класса. Описание операции включает четыре части: имя; список параметров; тип возвращаемого значения; видимость. Реализация операции называется методом.

Результат операции зависит от текущего состояния объекта. Виды операций:

- Операции реализации (implementor operations) – реализуют требуемую функциональность.
- Операции управления (manager operations) управляют созданием и уничтожением объектов (конструкторы и деструкторы).
- Операции доступа (access operations) – так называемые, get-теры, set-теры – дают доступ к закрытым атрибутам.
- Вспомогательные операции (helper operations) – непубличные операции, служат для реализации операций других видов.

Объект может быть абстракцией некоторой сущности предметной области (объект реального мира) или программной системы (архитектурный объект).

Сравнение архитектур традиционной и ОО-системы:



В ОО-системе алгоритмы (поведение) и структуры данных (внутреннее устройство) объединены в объекты, за счет уменьшается сложность системы, локализуются изменения.

Понятие полиморфизма может быть интерпретировано, как способность объекта принадлежать более чем одному типу. *Полиморфизм* – способность скрывать множество различных реализаций под единственным общим именем или интерфейсом. Интерфейс – это совокупность операций, определяющих набор услуг класса или компонента. Интерфейс не определяет внутреннюю структуру, все его операции открыты. Пример, одна и та же операция *рассчитатьЗарплату* может иметь три различные реализации в трех различных классах: *СлужащийСПочасовойОплатой*, *СлужащийНаОкладе*, *ВременныйСлужащий*.

*Компонент* – это относительно независимая и замещаемая часть системы, выполняющая четко определенную функцию в контексте заданной архитектуры.

Компонент представляет собой физическую реализацию проектной абстракции и может быть: компонентом исходного кода (src-шник); компонентом времени выполнения (dll, ActiveX и т. п.); исполняемый компонентом (exe-шник). Компонент обеспечивает физическую реализацию набора интерфейсов. Компонентная разработка (component-based development) представляет собой создание программных систем, состоящих из компонентов (не путать с объектно-ориентированным программированием (ООП)).

ООП – способ создания программных компонентов, базирующихся на объектах.

Компонентная разработка – технология, позволяющая объединять объектные компоненты в систему.

*Пакет* – это общий механизм для организации элементов в группы. Это элемент модели, который может включать другие элементы. Каждый элемент модели может входить только в один пакет.

Пакет является:

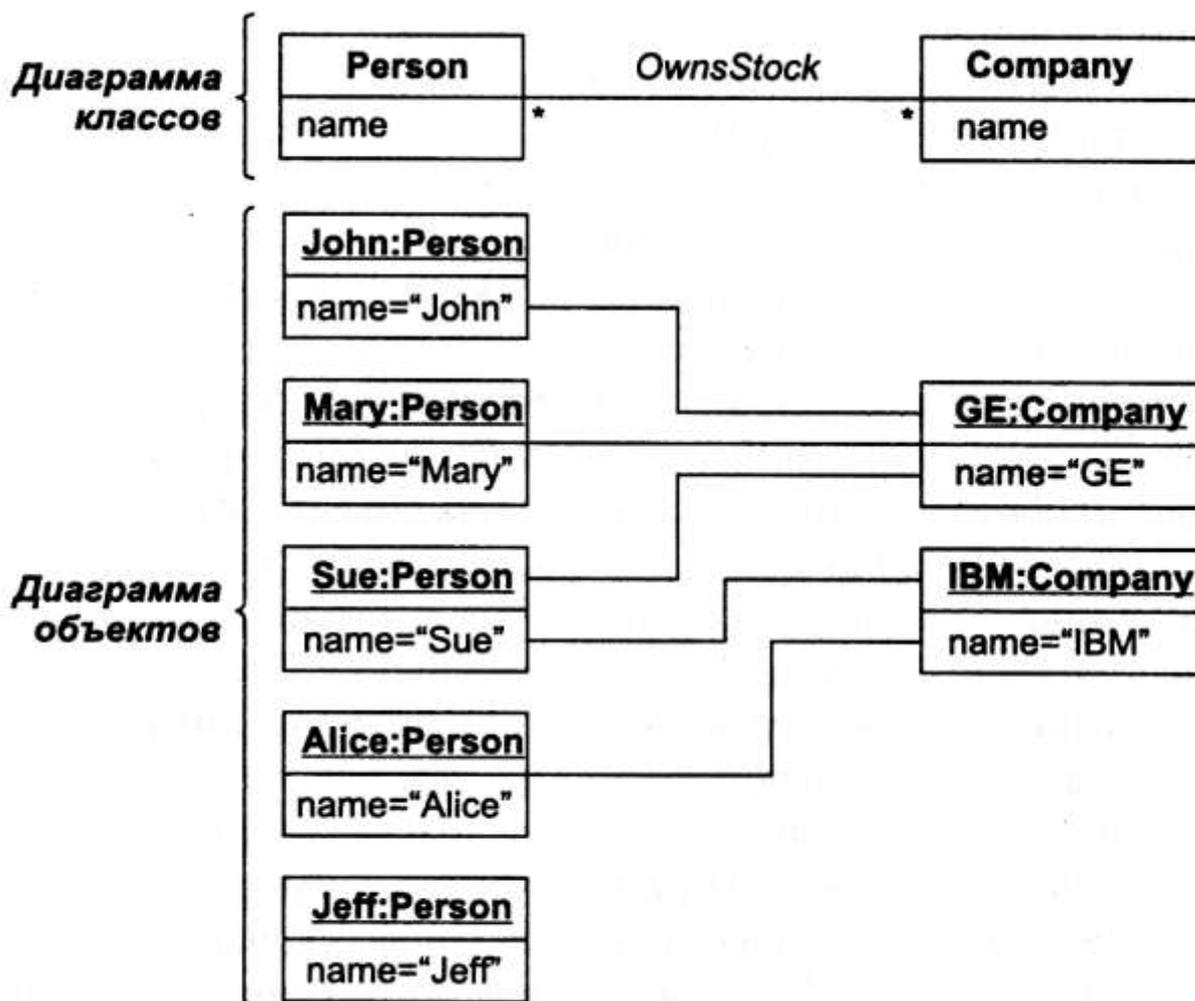
- средством организации модели в процессе разработки, повышения ее управляемости и читаемости;
- единицей управления конфигурацией.

*Подсистема* – это комбинация пакета (может включать другие элементы модели) и класса (обладает поведением). Подсистема реализует один или более интерфейсов, определяющих ее поведение. Она используется для представления компонента в процессе проектирования.

Между элементами объектной модели существуют различные виды *связей*.

*Соединение (link)* – физическая или концептуальная связь между *объектами*, позволяющая им взаимодействовать.

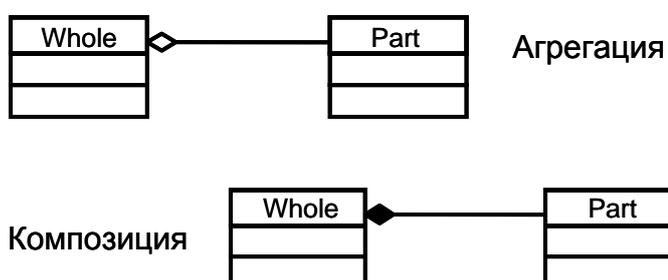
*Ассоциация* – связь между классами, описывающая группу однородных по структуре и семантике соединений между экземплярами классов. Соединения являются экземплярами ассоциации точно так же, как соединенные объекты являются экземплярами классов, связанных ассоциацией.



ассоциацией.

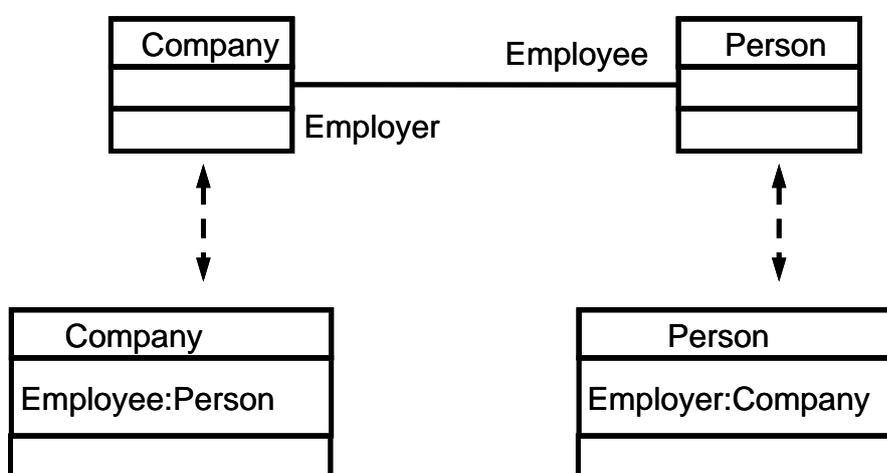
Пример показывает, что ассоциация ВладеетАкциями между классами Персона и Компания может иметь несколько экземпляров – соединений. Обратите внимание, что два соединения, являющиеся экземплярами одной и той же ассоциации, не могут связывать одни и те же объекты дважды.

*Агрегация* – более сильный тип ассоциативной связи между целым и его частями (пример: автомобиль и мотор). *Композиция* – усиленная агрегация, когда часть не может существовать без целого (пример: университет, факультет, кафедра). Композиция и агрегация транзитивны, в том смысле, что если В является частью А, и С является частью В, то С также является частью А (но на диаграмме связи, возникающие за счет транзитивности, явно не изображаются).



Соединения, являющиеся экземплярами композиций или агрегаций также изображаются с ромбами на полюсах.

Ассоциации (включая агрегации и композиции) характеризуются: направлением, именем, ролевыми именами участников связи, мощностями. *Направление* указывает ход сообщений. По умолчанию ассоциации двунаправлены, т. е. сообщения могут исходить из любого конца ассоциации. Если введено ограничение по направлению, то добавляется стрелка на конце связи. Ассоциации может быть дано имя, полюсам (концам) ассоциации могут быть назначены роли. Например, у ассоциации между классом Компания и классом Персона полюсу класса Компания может быть назначена роль Работодатель, а другому полюсу – роль Служащий. Понятие ассоциации связано с понятием *атрибута*. При наличии ассоциации между классами их экземпляры соединены ссылками, то есть имеют атрибуты, значениями которых являются ссылки на экземпляры связанного класса (см. рис.). Как правило соглашения моделирования предписывают изображать атрибуты простых типов (числа, символы, строки, логические

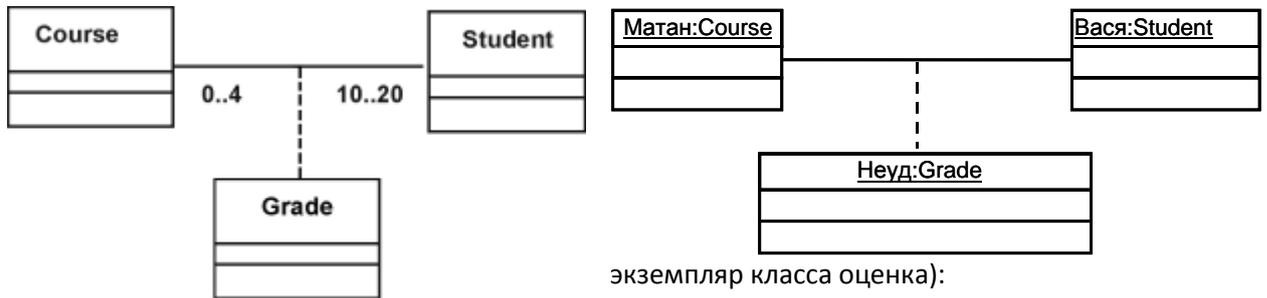


переменные, время, даты). Атрибуты сложных типов изображаются как ассоциации.

*Мощность* (multiplicity) показывает, как много объектов может участвовать в соединениях – экземплярах ассоциации. Мощность – это количество объектов одного класса (с той стороны связи, где приписана мощность), которые соединены с *одним* объектом другого класса (на другом конце связи). Для каждой ассоциации существуют два указателя мощности – по одному на каждом конце связи. Для соединений мощность не указывают, так как на любом конце соединения находится ровно один объект. Обозначения мощностей в UML:

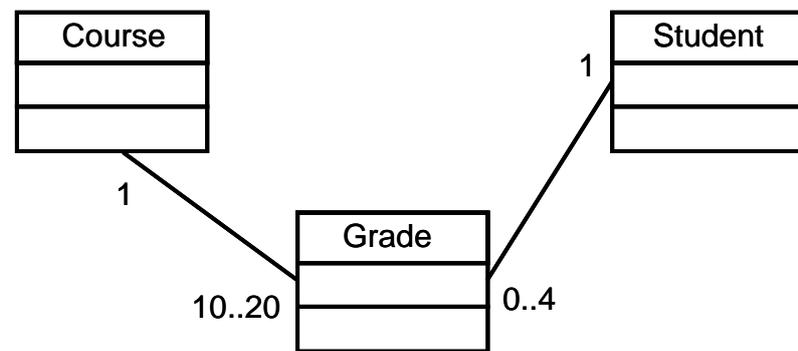
Мощность	Значение
1	Ровно один
0..* или *	Ноль или больше
1..*	Один или больше
0..1	Ноль или один
2..4	Заданный диапазон
2, 4..6	Несколько диапазонов

Частный случай ассоциации – класс ассоциации, при помощи которого атрибуты и операции можно привязать непосредственно к соединению. Т. е. при наличии класса ассоциации с каждым соединением связан его экземпляр (в примере для каждой связанной пары курс – студент есть



экземпляр класса оценка):

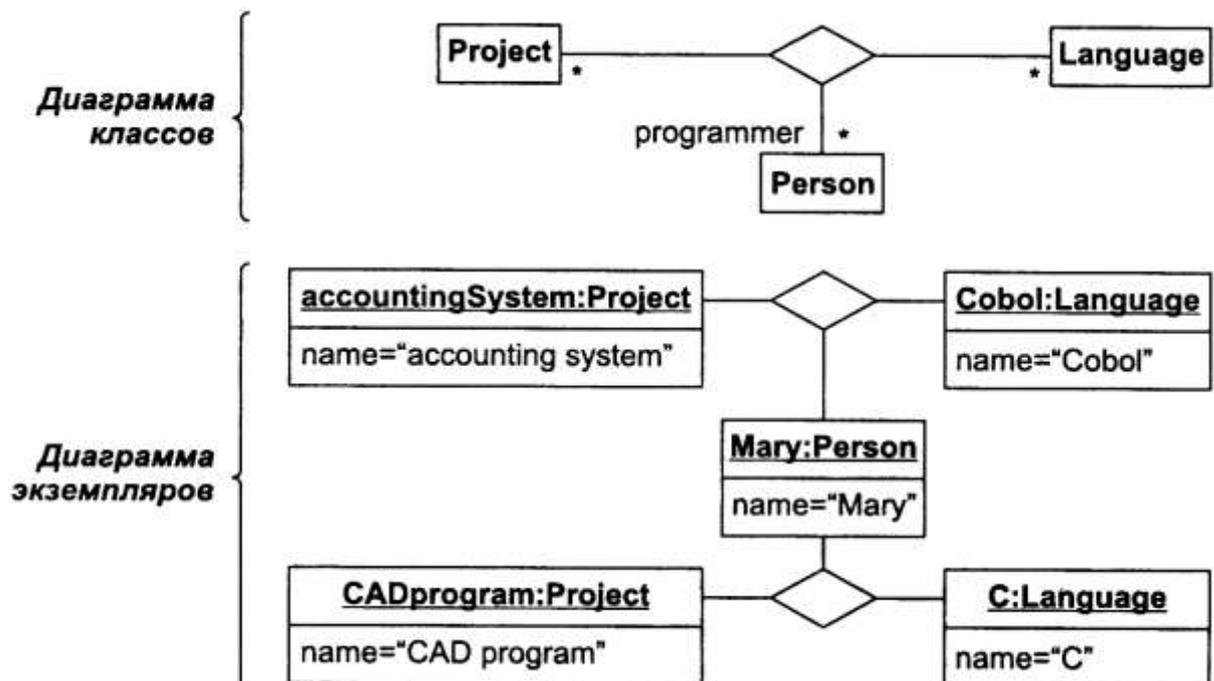
Следует заметить, что классы ассоциаций являются артефактами моделирования, то есть ими оперируют аналитики, архитекторы. Языки программирования не имеют средств, поддерживающих эти конструкции. В ходе реализации программистам приходится преобразовывать модели так, чтобы можно было создать код. При этом теряются ограничения целостности. Так модель курсы-оценки-студенты будет преобразована к следующему виду, допускающему в отличие от исходной модели более одной



оценки по одному курсу у одного студента:

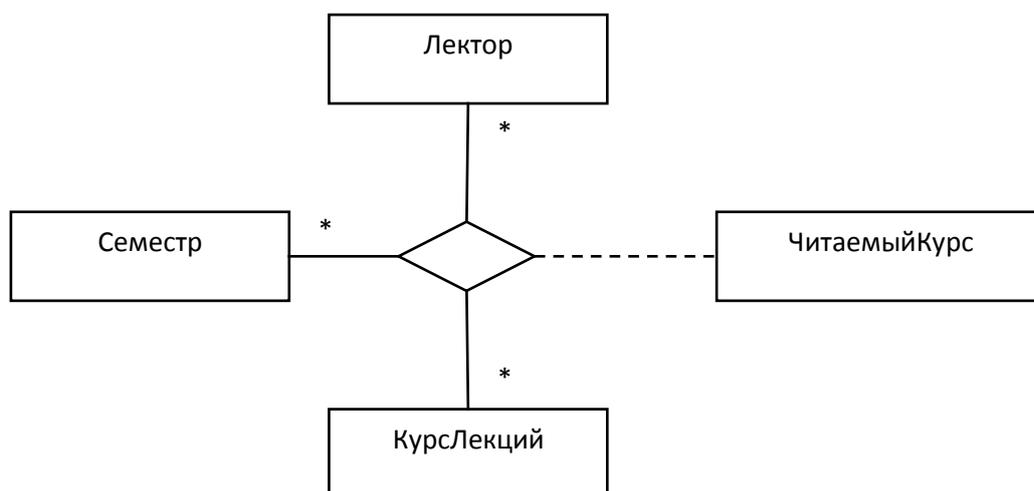
Обратите внимание как переместились мощности связей.

В UML 2.0 к объектной модели добавлено понятие N-арной ассоциации (для каждой комбинации объектов не более одной связи):

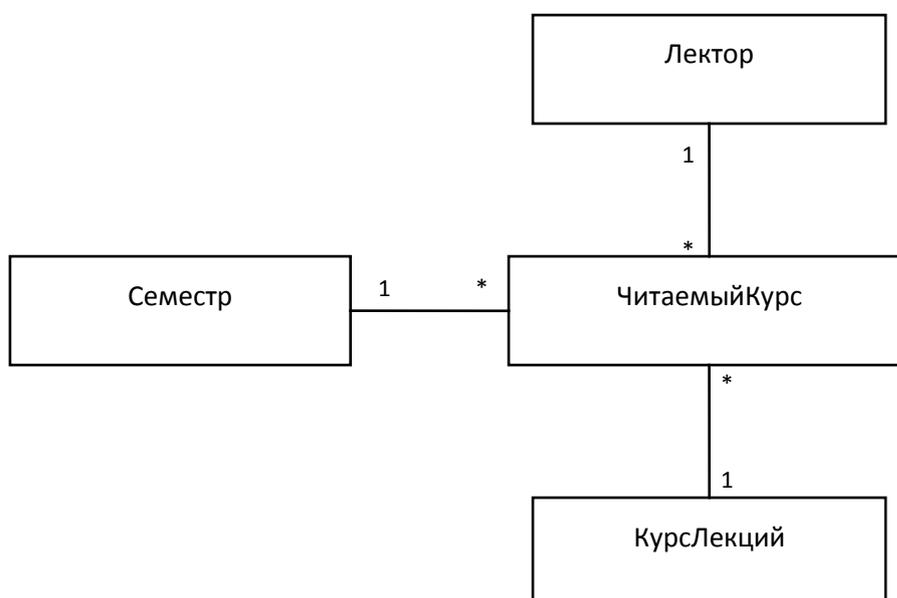


На рисунке представлена тернарная ассоциация и класс-ассоциация, связывающая проект, программистов, занятых в проекте, и языки, на которых они программируют в проекте. Некоторым аналогом тернарной ассоциации является реляционное отношение над тремя доменами (таблица со столбцами *проект, программист, язык*). Экземплярами N-арных ассоциаций являются N-арные соединения. Так, программистка Мэри в одном проекте пишет на Коболе, а в другом на Си. Заметим, что мощности на концах тернарной ассоциации не воспрепятствуют Мэри писать в одном и том же проекте на разных языках (хотя на диаграмме объектов такое не показано). Единственное ограничение, наложенное в данном случае, – два разных тернарных соединения не могут связывать одну и ту же тройку объектов – например: Мэри, проект accountingSystem и Кобол.

К N-арным ассоциациям могут быть присоединены классы ассоциаций. Классы Лектор, Семестр и КурсЛекций связаны тернарной ассоциацией (означающей, что некоторый лектор читает курс лекций в определенном семестре). Класс ассоциаций ЧитаемыйКурс может хранить дополнительные сведения о связи, например, количество слушателей и т. п.

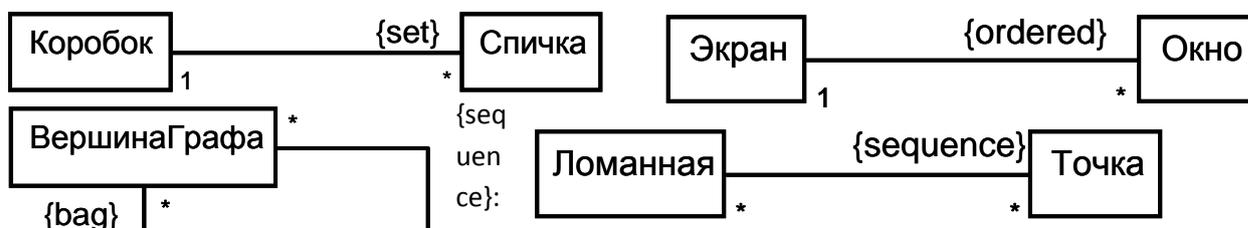


В объектно-ориентированных языках N-арные ассоциации не поддерживаются стандартными средствами. Их можно промоделировать с помощью обычных (бинарных) ассоциаций, но при этом снимается ограничение на единственность соединения, связывающего N-ку объектов:



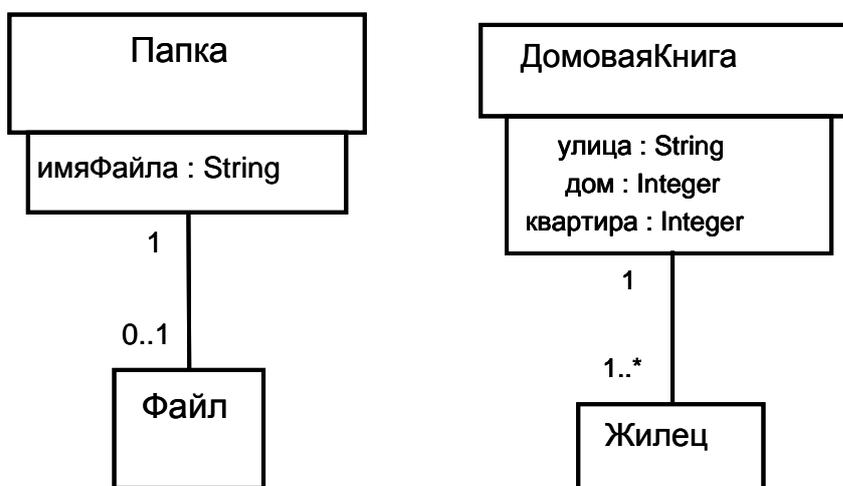
Во второй модели тройка объектов лекторПетров, семестрСедьмой и курсЛекцийМатан могут быть соединены более чем единожды посредством разных экземпляров класса ЧитаемыйКурс.

Полюса ассоциаций с мощностью «много» имеют еще две характеристики: упорядоченность связываемых объектов и повторяемость (т. е. образуют ли связываемые объекты множество или мультимножество). Различные сочетания этих характеристик образуют четыре типа полюсов: множества {set} (тип полюса по умолчанию), упорядоченные множества {ordered}, мультимножества {bag} и последовательности



Спички в коробке образуют множество (неупорядоченное). Окна на экране – упорядоченное (по глубине) множество. В мультиграфе между парой вершин может быть несколько ребер, значит каждая вершина может быть связана с мультимножеством вершин. Вершины ломанной линии, если допускаются наложения, образуют последовательность, в которой одна и та же вершина может встречаться несколько раз.

Ассоциациям могут быть приписаны квалификаторы. Квалификатор – атрибут или набор атрибутов ассоциации, значение которых позволяет выбрать для конкретного объекта квалифицированного класса множество целевых объектов на противоположном конце соединения. Например, если в папке может находиться не более одного файла с заданным именем, то имя файла – квалификатор ассоциации папка -> файл. Квалификатор не обязательно состоит из одного атрибута (также как и потенциальный ключ записей в таблице). Например, жильцы из домовой книги проиндексированы адресами, состоящими из названия улицы, номера дома и номера квартиры.

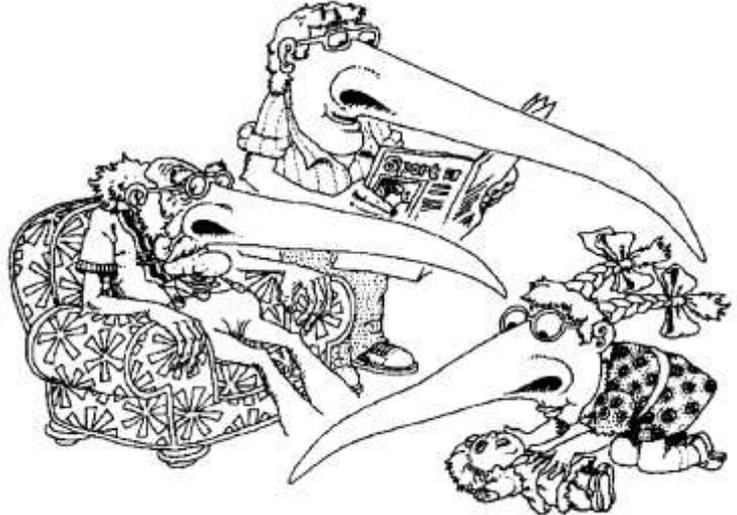


*Зависимость* – связь между двумя элементами модели, при которой изменения в спецификации одного элемента могут повлечь за собой изменения в другом элементе. Например, пакет, который импортирует классы другого пакета, является зависимым от него.

Зависимость изображается как пунктирная стрелка с обычным наконечником. Зависимость между классами возникает в следующих случаях:

- в сигнатуре операции одного класса есть аргумент – объект другого класса;
- в методе одного класса есть локальный объект другого класса;
- результатом операции одного класса является экземпляр другого класса.

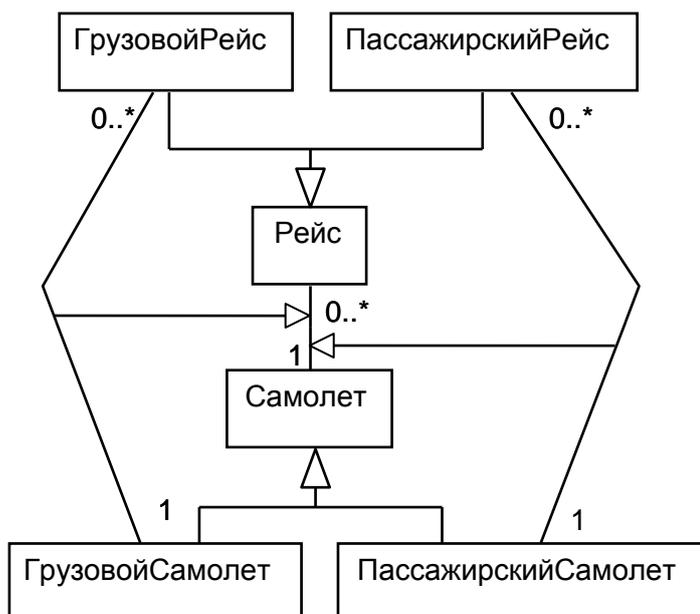
*Обобщение* – это связь «тип – подтип». Оно реализует механизм наследования (inheritance), поддерживает полиморфизм. *Наследование* – это построение новых классов на основе существующих с возможностью добавления или переопределения свойств (атрибутов) и поведения (операций). Изображается как стрелка с треугольным наконечником, исходящая из наследника и указывающая на родителя. Еще одним обозначением является выделение курсивом имен абстрактных классов (не имеющих собственных экземпляров).



Общие атрибуты, операции и/или отношения отображаются на верхнем уровне иерархии. Заметим, что ассоциации класса-предка наследуются классами потомками, т. е. экземпляры потомков могут иметь соединения того же рода, что и экземпляры родительского класса. Действует принцип подстановки Лисковской (Liskov substitution principle LSP), по которому любое утверждение, справедливое для экземпляров класса, сохраняет справедливость для экземпляров всех его подклассов. Например, из LSP следует, что экземпляр подкласса подкласса (класса «внука») может рассматриваться как экземпляр подкласса или экземпляр исходного класса (своего «деда»).

В объектной модели наследование может быть множественным. На связи могут накладываться ограничения. Например, если необходимо, множественное наследование в некоторой иерархии классов может быть запрещено (над связью указывается ключевое слово: **{disjoint}**).

Обобщение рассматривается не только для классов, но и для ассоциаций. В этом случае стрелка обобщения соединяет две ассоциации. Заметим, что в моделях такое встречается редко. Ниже приведен пример, когда связь между рейсом и самолетом переопределяется в классах наследниках связью-наследницей.



*Реализация* – связь между контрактом (интерфейсом, вариантом использования) и его исполнением (классом, подсистемой, компонентой и т. п.). Изображается пунктирной стрелкой с треугольным наконечником, исходящей из исполнения (класса, подсистемы) и указывающей на контракт (интерфейс).

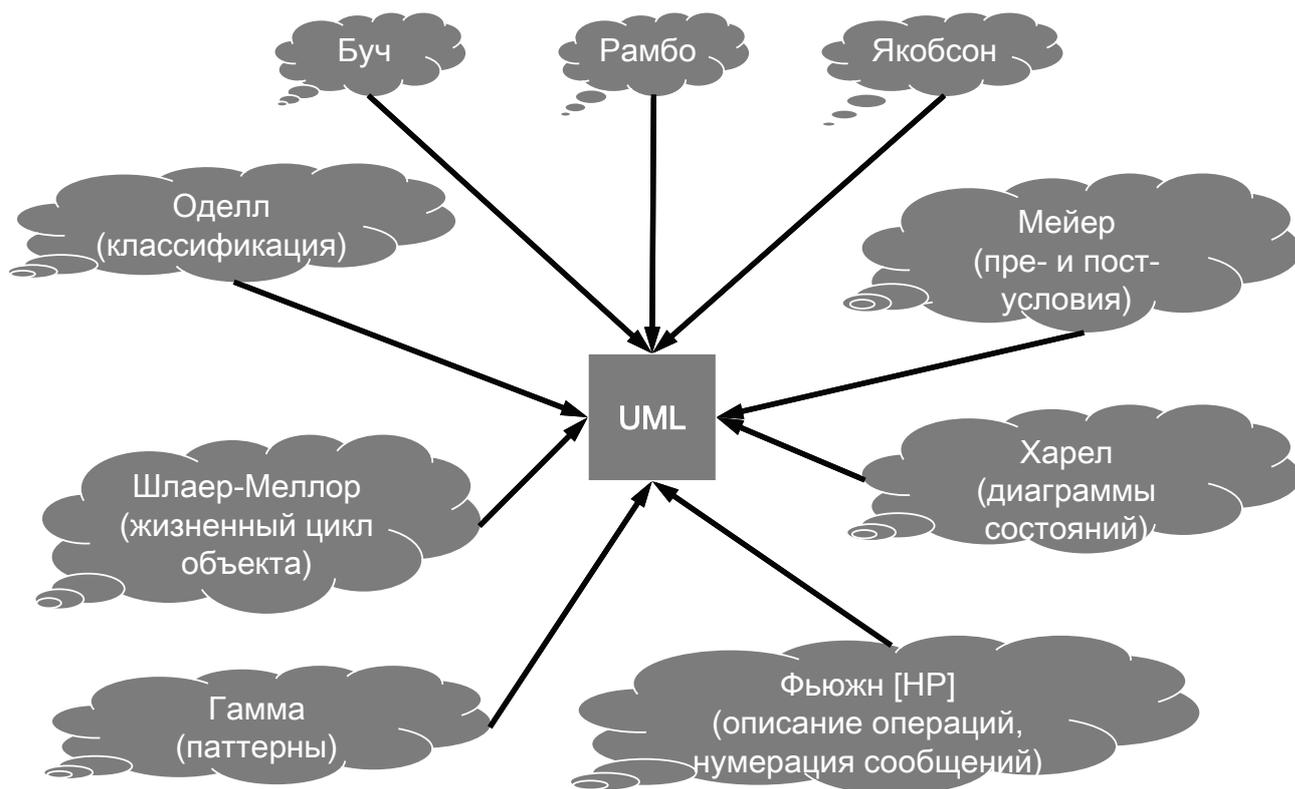
## Литература к лекции 2

1. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений. 3-е изд. – М.: Вильямс, 2008. – Часть I.
2. Вендров А. М. Проектирование программного обеспечения экономических информационных систем. 2-е изд. – М.: Финансы и статистика, 2005. – Глава 2.
3. Рамбо Дж., Блаха М. UML 2.0. Объектно-ориентированное моделирование и разработка. 2-е изд. – СПб.: Питер, 2006. – Главы 2-3.
4. Грэхем И. Объектно-ориентированные методы. Принципы и практика. 3-е изд.: Пер. с англ. – М.: Вильямс, 2004. – Глава 1.

## Лекция 3. Унифицированный язык моделирования (Unified Modelling Language)

Унифицированный язык моделирования UML (Unified Modeling Language) – это язык для определения, представления, проектирования и документирования программных систем, организационно-экономических систем, технических систем и других систем различной природы. UML содержит стандартный набор диаграмм и нотаций самых разнообразных видов.

UML является наследником методов объектно-ориентированного анализа и проектирования, появившихся в конце 1980-х и начале 1990-х годов. Создание UML началось в конце 1994 г., с объединения методов Booch и OMT (Object Modeling Technique) под эгидой компании Rational Software. К концу 1995 г. Гради Буч и Джеймс Рамбо создали первую спецификацию Unified Method, версия 0.8. Тогда же в 1995 г. к ним присоединился создатель метода OOSE (Object-Oriented Software Engineering) Ивар Якобсон. UML является унификацией методов Буча, Рамбо и Якобсона а также суммой передового опыта по разработке ПО:



Разработка UML преследовала следующие цели:

- 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.

UML широко используется в индустрии ПО. Практически все мировые производители CASE-средств поддерживают UML в своих продуктах. В 1997 году Object Management Group (OMG) приняла стандарт UML 1.1. В 2004 году был пройден следующий важный этап – принят стандарт UML версии 2.0. В настоящее время UML проходит процесс стандартизации ISO. Текущая версия UML – 2.1.2.

Основные «строительные блоки» UML:

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

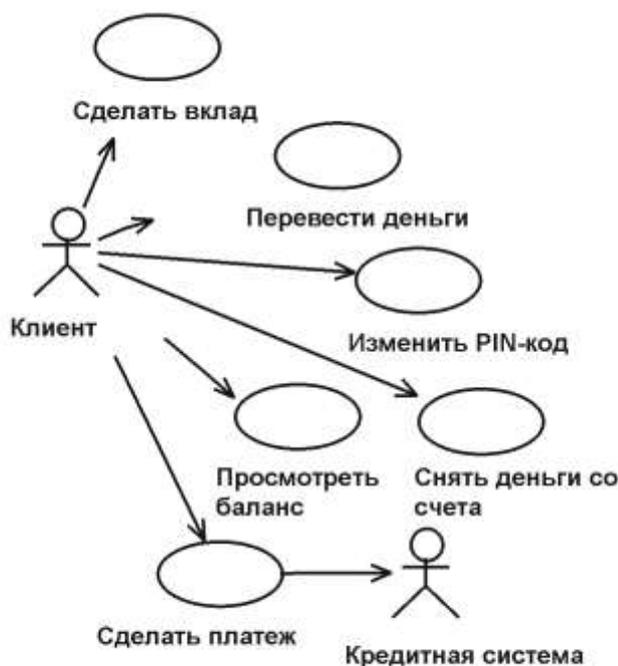
Состав диаграмм UML 1.x:

- структурные:
  - диаграммы классов, моделирующие статическую структуру классов системы и связи между классами;
  - диаграммы компонентов, моделирующие иерархии компонентов ПО;
  - диаграммы размещения, моделирующие физическую архитектуру системы;

- поведенческие:
  - диаграммы вариантов использования, моделирующие бизнес-процессы и требования к ПО;
  - диаграммы взаимодействия (диаграммы последовательности и коммуникационные диаграммы), моделирующие обмен сообщениями между объектами;
  - диаграммы состояний, моделирующие поведение объектов;
  - диаграммы деятельности, моделирующие поведение системы в целом и потоки управления.

В UML 2.0 введены новые типы диаграмм, которых ранее не было: диаграммы обзора взаимодействия, временные диаграммы и диаграммы составных структур.

Рассмотрим диаграммы вариантов использования. Вариант использования – это ответные действия ПО, являющиеся реакцией на событие, инициируемое извне. Вариант использования описывает типичное взаимодействие между пользователем и ПО. Он отражает представление о поведении системы с точки зрения пользователя. На диаграммах варианты использования



представляются в виде овалов.

Действующее лицо – это роль, которую пользователь играет по отношению к системе. На диаграммах вариантов использования они изображаются в виде стилизованных человеческих фигурок. Действующим лицом может быть пользователь-человек, внешняя программная система или время, если запуск каких-либо событий в системе зависит от времени.

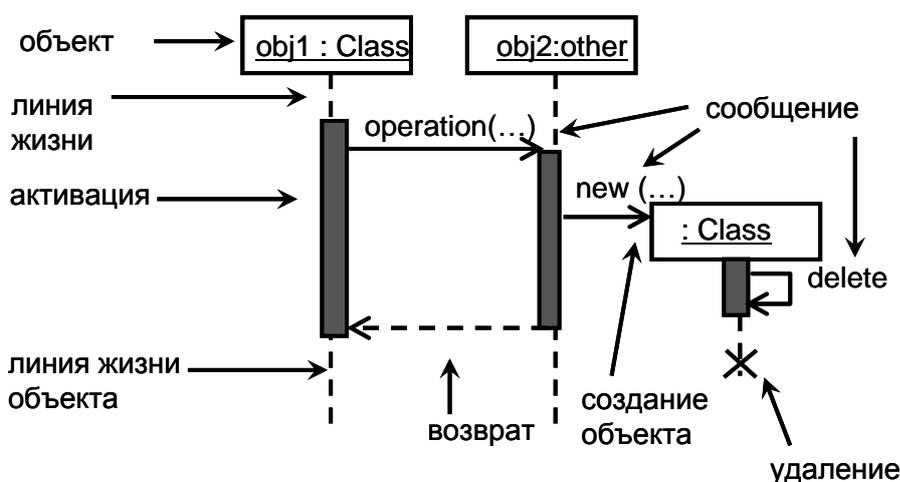
Диаграмма вариантов использования является самым общим представлением функциональных требований к системе. Детально функциональные требования описываются в документе, называемом «сценарий варианта использования» или «поток событий». Он подробно документирует процесс взаимодействия действующего лица с системой, реализуемого в рамках варианта использования.

В диаграммах вариантов использования может присутствовать несколько типов связей:

- связь коммуникации (линия со стрелкой, обозначающая связь между вариантом использования и действующим лицом, по сути такая связь – путь для передачи запросов или данных);
- связь включения (пунктирная линия со стрелкой, обозначающая включение многократно используемой функциональности, представленной в виде отдельного варианта использования);
- связи расширения (пунктирная линия со стрелкой, соединяющая вариант – особый случай – с базовым вариантом использования);
- связь обобщения (сплошная линия с треугольным концом, используемая в иерархиях наследования действующих лиц или вариантов использования).

Связи коммуникации от действующих лиц к вариантам использования показывают, какие действующие лица инициируют варианты использования. Коммуникации, направленные от вариантов использования к действующим лицам указывают, какие действующие лица получают данные в ходе выполнения варианта использования (или обрабатывают запросы от разрабатываемого ПО). Коммуникации между вариантами использования не допускаются.

*Диаграммы взаимодействия* описывают поведение взаимодействующих групп объектов в рамках потока событий. На диаграмме отображается ряд объектов и сообщения, которыми они обмениваются между собой. Сообщение – это средство, с помощью которого объект-отправитель запрашивает у объекта-получателя выполнение одной из его операций. Существует два вида диаграмм взаимодействия: диаграммы последовательности и коммуникационные диаграммы (ранее называемые кооперативными).

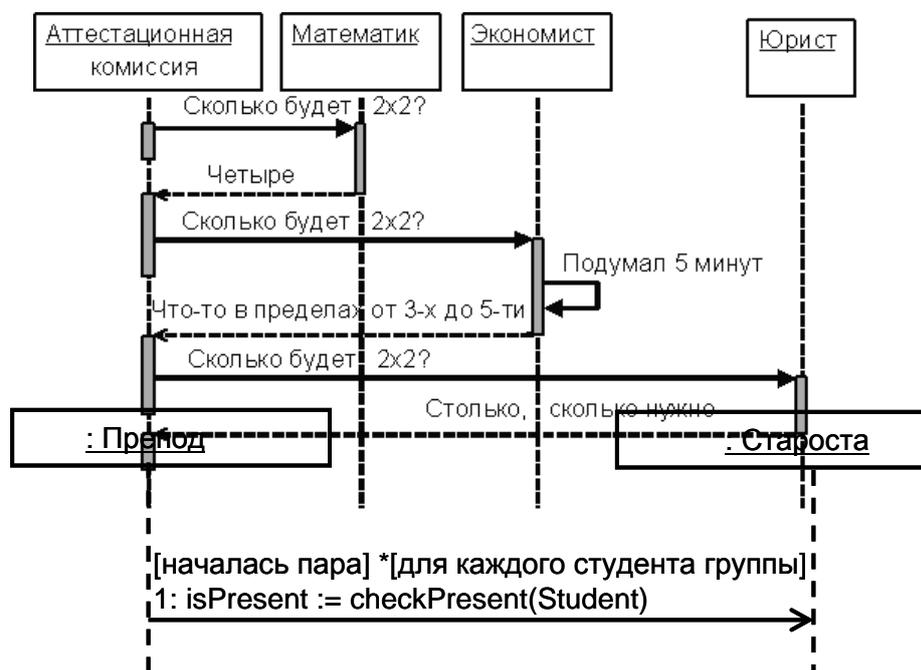


*Диаграммы последовательности* отражают временную последовательность событий, происходящих в рамках варианта использования. Экземпляры действующих лиц и объекты (экземпляры классов) системы изображаются в верхней части диаграммы. От каждого из них вниз проведена пунктирная вертикальная черта – «линия жизни». Стрелки, соответствующие сообщениям, которые передаются между экземпляром действующего лица и объектом или между объектами, соединяют линии жизни отправителя и получателя сообщения. Порядок отправки сообщений соответствует их размещению на диаграмме сверху вниз. Вдоль линии жизни прямоугольниками отмечены *активации* – периоды, когда объекты активны, т. е. выполняются связанные с ними процессы.

Каждое сообщение может быть описано в таком формате:

**[сторожевое условие] \*[повторение] номер : переменная := операция (аргументы)**

**[сторожевое условие]** – сообщение посылается только при выполненном условии, при невыполненном не посылается;



**\*[повторение]** – итерация, которая указывает, что сообщение посылается столько раз, сколько указано внутри [ ];

**номер :** – порядковый номер сообщения;

**переменная :=** – указание, где будет сохранен результат;

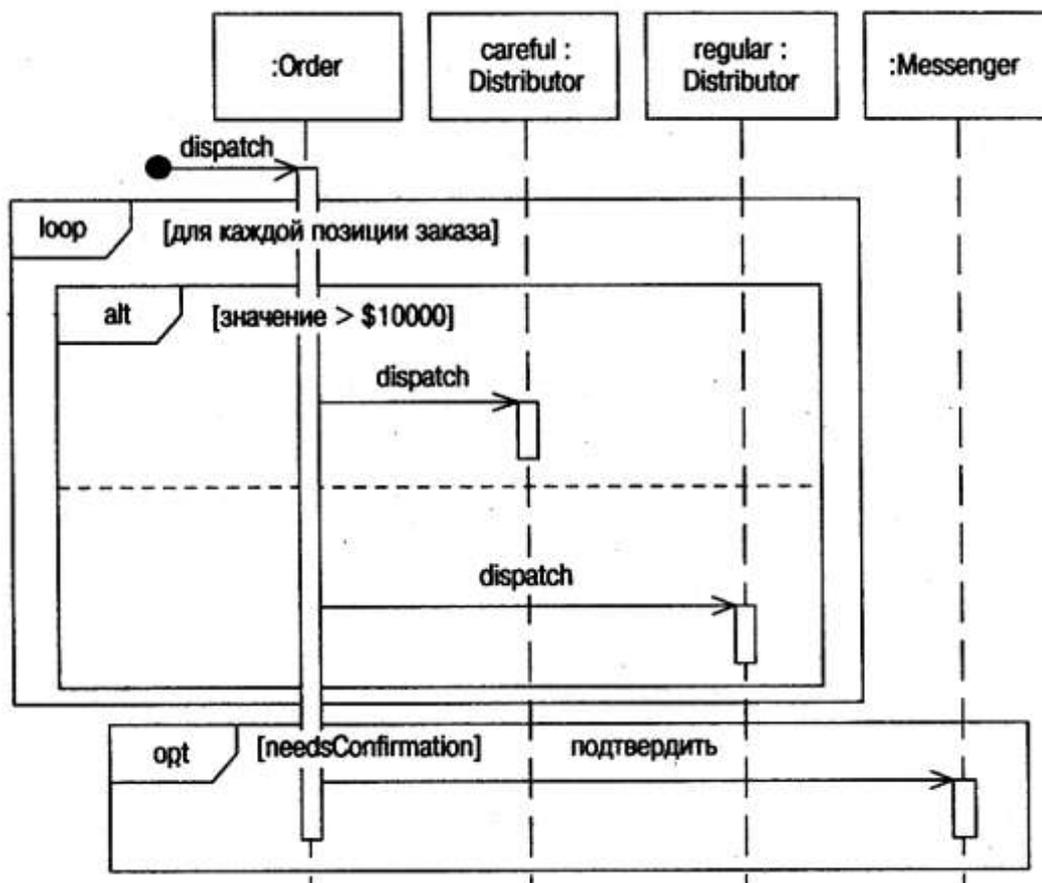
**операция (аргументы)** – какая операция с какими аргументами будет вызвана.

Любая из частей описания может отсутствовать.

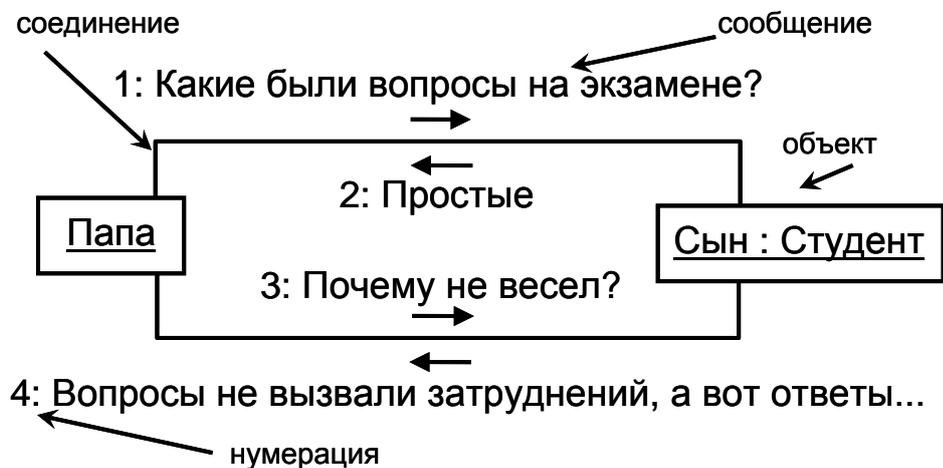
В примере показано взаимодействие при проверке посещаемости. В начале занятия по каждому студенту из списка преподаватель ждет ответа от старосты, присутствует ли студент.

В UML 2.0 диаграммы последовательности могут содержать блоки разных типов:

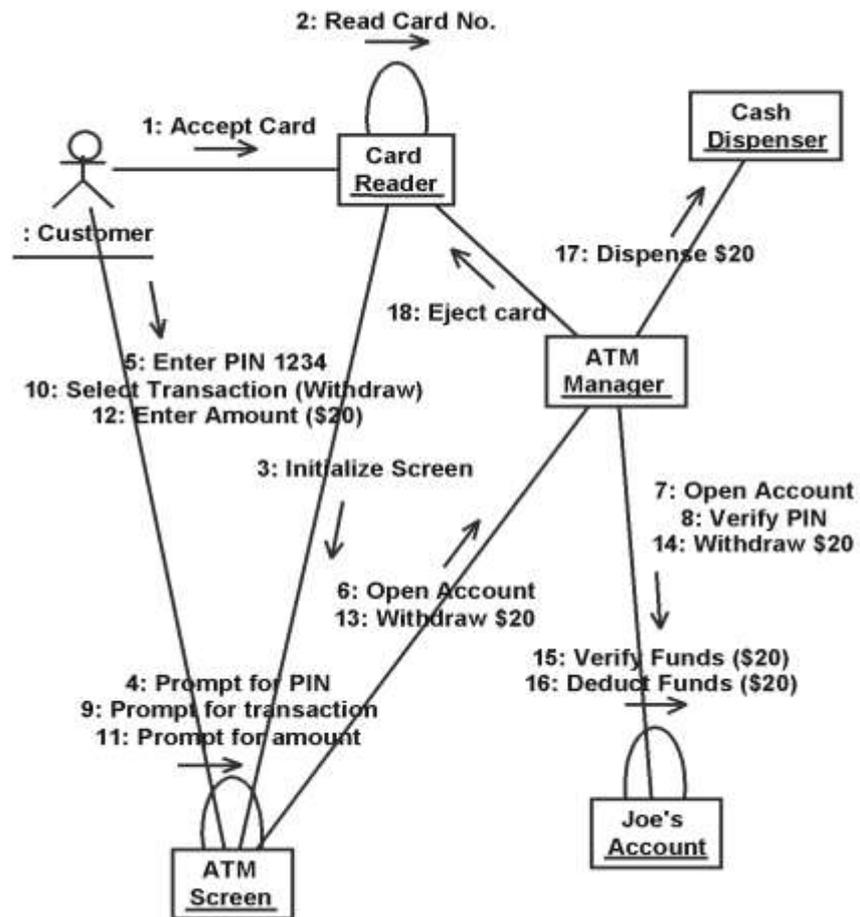
- alt – несколько альтернатив (каждая альтернатива – часть блока помеченная сторожевым условием);
- opt – необязательный блок (взаимодействие выполняемое при истинности сторожевого условия);
- par – блок из параллельно выполняемых разделов;
- loop – цикл (пока истинно условие);
- region – критический участок;
- neg – неверное взаимодействие;
- ref – ссылка на другую диаграмму;
- sd – блок, включающий диаграмму последовательности целиком.



Вторым видом диаграмм взаимодействия являются *коммуникационные диаграммы* (в UML 1 их называли *кооперативными*). Как и диаграммы последовательности, они отображают поток событий варианта использования. На коммуникационных диаграммах внимание сконцентрировано на соединениях между объектами. Из них легче понять связи между объектами, однако, труднее уяснить последовательность событий. Объекты и/или действующие лица, обменивающиеся сообщениями, связываются линиями (соединениями), над которыми в виде стрелок обозначаются сообщения. Нумерация сообщений указывает их последовательность во времени.



Рефлексивные сообщения (который объект посылает сам себе) изображаются над



псевдосоединением – дугой над объектом.

Если необходимо указать итерации во взаимодействии, на коммуникационных диаграммах используют примечания. Например:

Примечание, заключенное в фигурные скобки, предписывает посылать последовательности из четырех сообщений для каждого работника в организации.

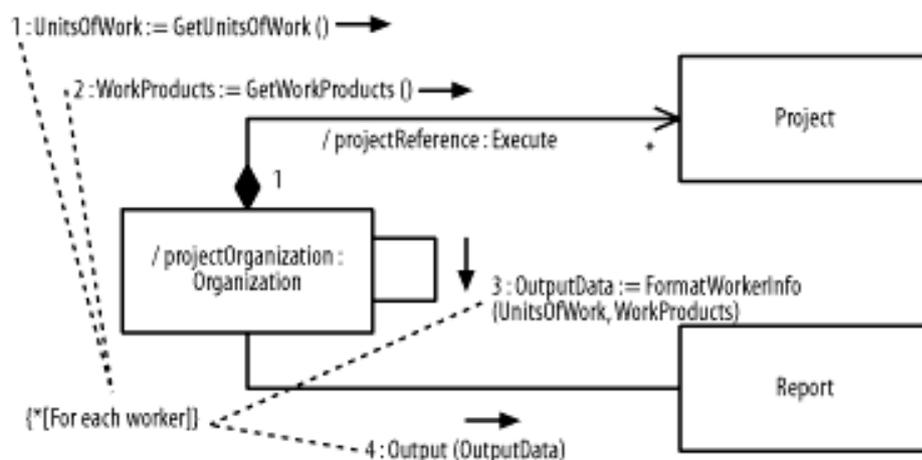
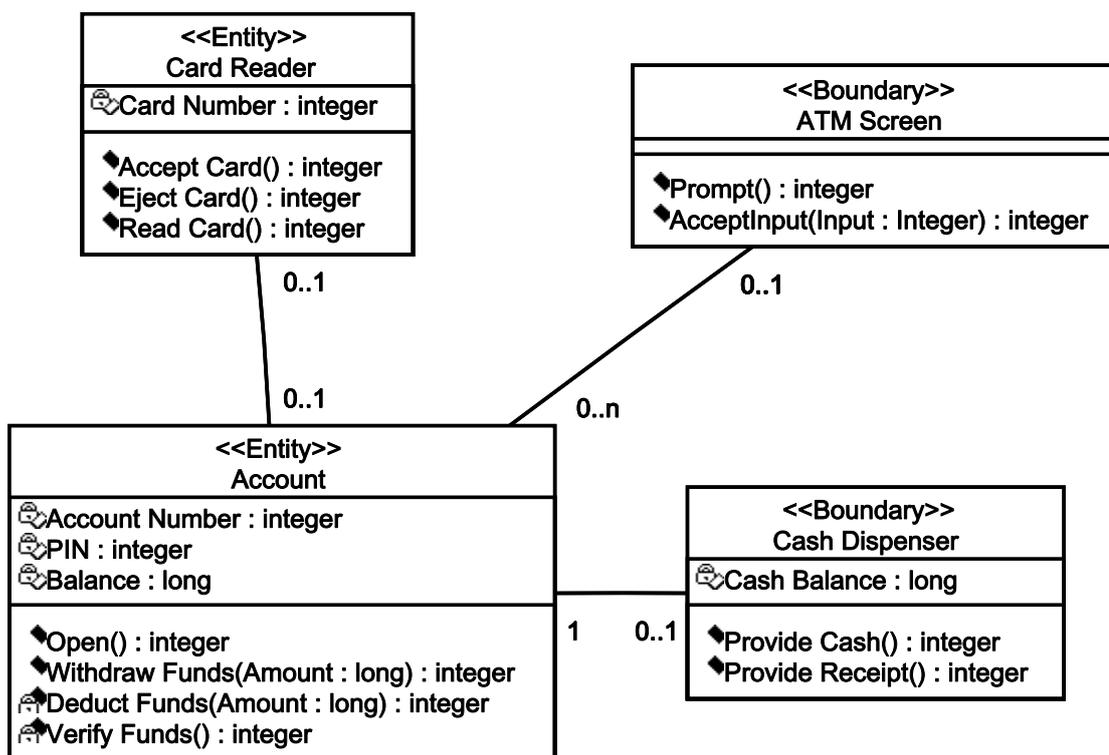


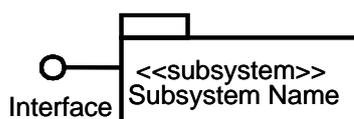
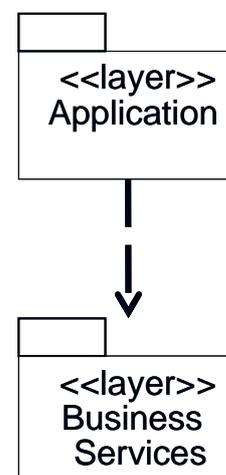
Диаграмма классов определяет классы системы и различного рода связи, которые существуют между ними (ассоциации, агрегации, композиции, зависимости, обобщения, реализации). На диаграммах классов изображаются также атрибуты классов, операции классов и ограничения, которые накладываются на связи между классами. Классы изображаются в виде прямоугольников, ассоциации – в виде сплошных линий, направления ассоциаций указываются стрелками, агрегации и композиции – в виде сплошных линий с ромбом на конце, связь обобщения – в виде сплошных линий с треугольником на конце, зависимость – в виде пунктирной линии со стрелкой. Роль, возложенная на класс изображается на диаграммах с помощью стереотипов. Класс может быть помечен как граничный (boundary), если он отвечает за взаимодействие с пользователем или внешней системой. Класс-контроллер реализует бизнес-логику приложения. Класс-сущность отвечает за представление данных. Активные классы (процессы или потоки) на диаграмме выделяют с помощью более толстых, чем у обычных классов



границ.

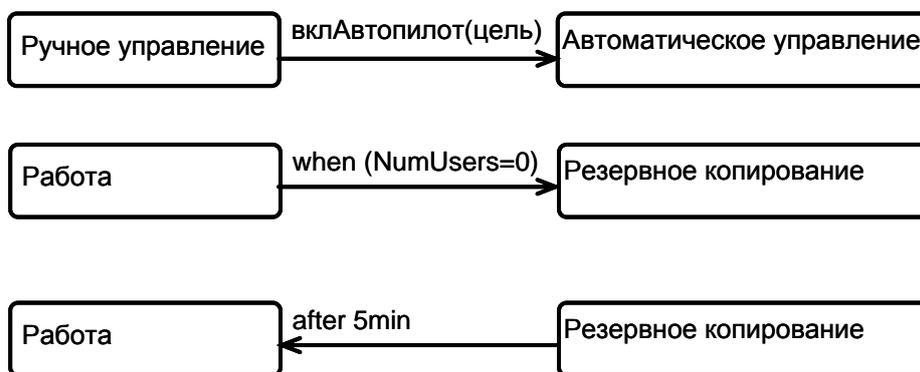
Для группировки классов, обладающих некоторой общностью, применяются пакеты. Пакет – общий механизм для организации элементов модели в группы. Каждый пакет – это группа элементов модели, иногда сопровождаемая диаграммами, поясняющими структуру группы. Каждый элемент модели может входить только в один пакет. Диаграммы пакетов отображают зависимости между пакетами, возникающие, если элемент одного пакета зависит от элемента другого.

Пакеты также используются для представления подсистем. Подсистема – это комбинация пакета (поскольку она включает некоторое множество классов) и класса (поскольку она обладает поведением, т.е. реализует набор операций, которые определены в ее интерфейсах). Связь между подсистемой и интерфейсом называется связью реализации.

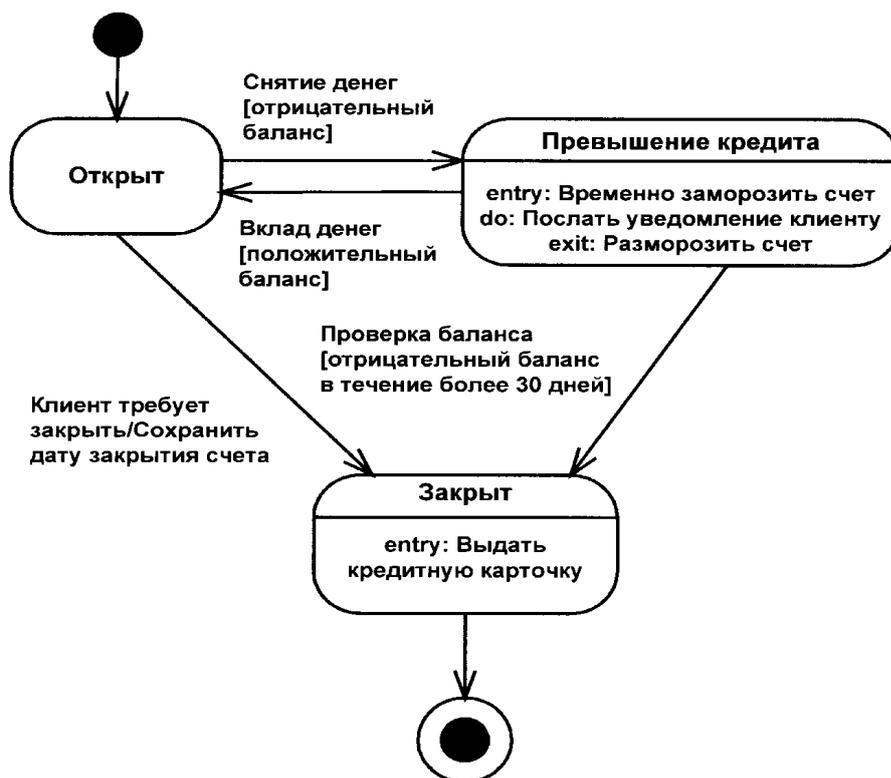


Диаграммы состояний определяют все возможные состояния, в которых может находиться экземпляр некоторого класса, а также процесс смены состояний объекта в результате наступления некоторых событий. Математической базой диаграмм состояний является конечный автомат. Автомат состоит из состояний между которыми есть переходы. Одно (или несколько, при наличии параллелизма) состояний могут быть текущим (текущими). Смена текущего состояния происходит в результате перехода, который вызывается событием.

События бывают разных видов: событие вызова (наступает, когда вызывается операция объекта, например, `вклАвтопилот`); событие изменения (наступает, когда становится истинным условие, например, `when(NumUses=0)`, всегда начинается со слова `when`); событие времени (наступает, когда наступает заданный момент времени, или истекает заданный промежуток времени, например `after 5min`). События, приписанные переходам, вызывают смену состояний. Если событие не отмечено на диаграмме, объект на него не реагирует. Если переходу не приписано событие, то это переход по завершению, который может осуществиться по окончании



деятельности внутри события.



На каждой диаграмме состояний имеется одно начальное состояние и произвольное количество финальных. Начальное состояние выделено черной точкой, оно соответствует состоянию объекта, когда он только что был создан. Финальное состояние обозначается черной точкой в белом кружке, оно указывает либо непосредственное уничтожение объекта, либо то, что вплоть до уничтожения объекта с ним ничего происходить не будет. Из финального состояния не бывает переходов, также как нет переходов в начальное.

Когда объект находится в каком-то конкретном состоянии, могут выполняться различные процессы. Они, называются деятельностями состояния и указываются на диаграмме (см. состояние Превышение кредита, деятельность помечена **do**:). Деятельность состояния – 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 состояния изображают внутри самого состояния, ей должно предшествовать слово *do* и двоеточие. Для состояния могут быть указаны входное и выходное действия. Входное действие выполняется всякий раз, когда объект переходит в данное состояние. В отличие от деятельности, входное действие рассматривается как непрерываемое. Входное действие также показывают внутри состояния, ему предшествует слово *entry* и двоеточие. Выходное действие осуществляется как составная часть любого выхода из состояния. Оно также является непрерываемым. Его изображают внутри состояния, ему предшествует слово *exit* и двоеточие. Действия, выполняемые в состоянии по наступлению события, помечаются словом именем события, после которого через двоеточие указывается действие. Это так называемые внутренние переходы. При выполнении внутреннего перехода действия по выходу и по входу не выполняются. Если вместо внутреннего перехода создать переход из состояния в само себя, то входное и выходное действия выполняются.

Переходом (transition) называется смена одного состояния объекта на другое. На диаграмме все переходы изображают в виде линий со стрелками. Объект может перейти в то же состояние, в котором он в настоящий момент находится. С переходом можно связать событие, сторожевое

условие и действие. Событие вызывает переход из одного состояния в другое. Событие размещают на диаграмме вдоль линии перехода. Сторожевые условия определяют, когда переход может произойти, а когда нет. Их изображают на диаграмме вдоль линии перехода после имени события, заключая в квадратные скобки [ ]. Действие, являющееся частью перехода, изображают вдоль линии перехода после имени события и сторожевого условия, ему предшествует косая черта.



Пример перехода по событию вызова:

Когда объект находится в состоянии State1, и из очереди событий (а все они считаются упорядоченными и не совпадающими по времени) извлекается событие вызова операции Ev с аргументами Arg, то диаграмма предписывает сделать следующее:

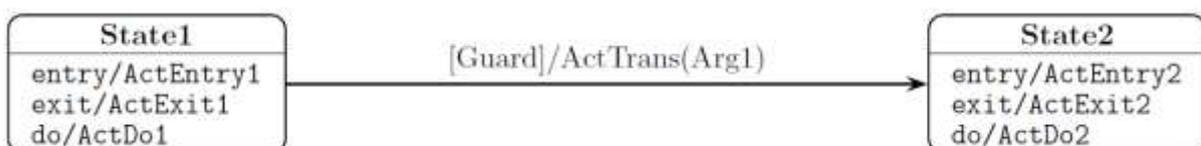
1. Убрать событие Ev(Arg) из очереди.
2. Проверить сторожевое условие Guard, если ложно – отменить переход (не делать пункты 3-7).
3. Прервать деятельность ActDo1 состояния State1, если она не закончилась.
4. Выполнить действие по выходу ActExit1.
5. Выполнить действие перехода ActTrans(Arg, Arg1).
6. Выполнить действие по входу ActEntry2 и назначить текущим состояние State2.
7. Запустить деятельность состояния ActDo2.



Пример перехода по событию изменения:

Когда объект находится в состоянии State1, все время проверяется условие Guard. Заметьте, что это не сторожевое условие, это условие, являющееся частью события изменения. Если оно стало истинным, то диаграмма предписывает сделать следующее:

1. Прервать деятельность ActDo1 состояния State1, если она не закончилась.
2. Выполнить действие по выходу ActExit1.
3. Выполнить действие перехода ActTrans(Arg1).
4. Выполнить действие по входу ActEntry2 и назначить текущим состояние State2.
5. Запустить деятельность состояния ActDo2.

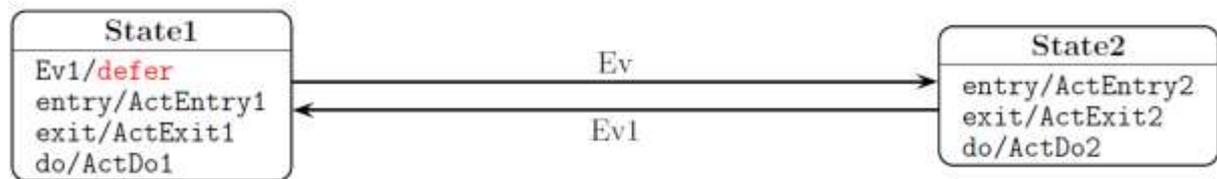


Пример перехода по завершению:

Когда объект находится в состоянии State1, и завершается выполнение деятельности ActDo1, диаграмма предписывает сделать следующее:

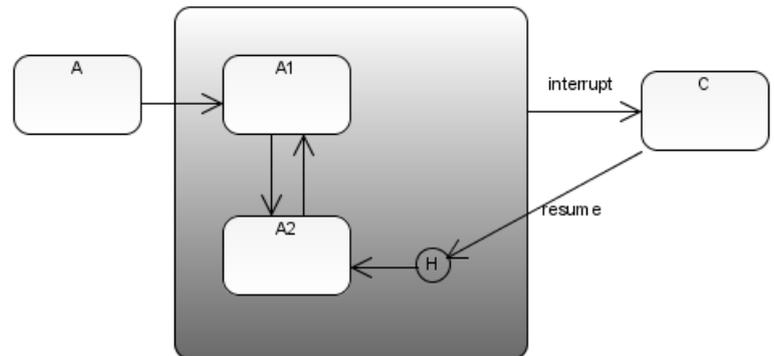
1. Проверить сторожевое условие Guard, если ложно – отменить переход (не делать пункты 2-5, заметим, что если потом условие станет истинным, переход уже не произойдет, т. е. условие проверяется единожды, сразу после окончания ActDo1).
2. Выполнить действие по выходу ActExit1.
3. Выполнить действие перехода ActTrans(Arg1).
4. Выполнить действие по входу ActEntry2 и назначить текущим состояние State2.
5. Запустить деятельность состояния ActDo2.

Внутри состояния можно отложить событие, для этого используется действие **defer**, например:



Пусть текущее состояние State1, очередь событий начинается с Ev1, Ev. Событие Ev1 откладывается. По событию Ev срабатывает переход в State2. Событие Ev1 в состоянии State2 не является отложенным, поэтому оно реактивируется (если отложено несколько событий, они активизируются в произвольном порядке). По событию Ev1 срабатывает переход в State1. Без defer событие Ev1 было бы извлечено из очереди и проигнорировано. Заметим, что если бы событие Ev было также отложенным в состоянии State1, то переход по Ev все равно произошел бы, без откладывания. Т. е. откладывать следует лишь те события, которые не приписаны переходам из данного состояния.

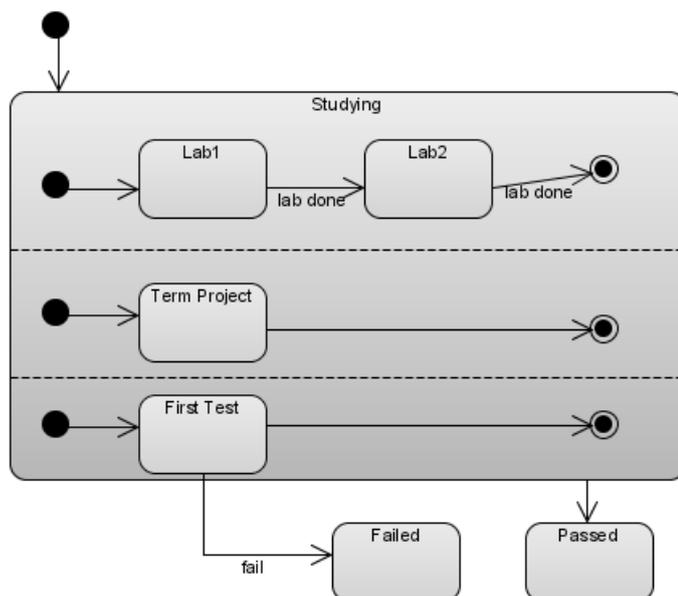
Для некоторых состояний (называемых суперсостояниями или композитными состояниями) указывают вложенные подсостояния. Внутри каждого такого состояния могут быть указано начальное и финальные состояния. Также в них может быть указано так называемое историческое состояние, переход в



которое означает возврат к предыдущему активному подсостоянию, которое запоминается всякий раз при выходе из суперсостояния. Выход из композитного состояния относится ко всем подсостояниям (в C можно перейти по событию interrupt из A1 и A2).

Если извне композитного состояния делается переход в подсостояние, то должно быть выполнено сначала действие по входу в композитное состояние, а затем действие по входу в подсостояние. Аналогично переход из подсостояния за границу композитного состояния вызывает сначала выполнение выходных действий подсостояния, а затем выходных действий суперсостояния.

В композитных состояниях могут использоваться параллельные области (см. рис. ниже). В таких случаях для объекта, находящегося в композитном состоянии текущими могут быть несколько

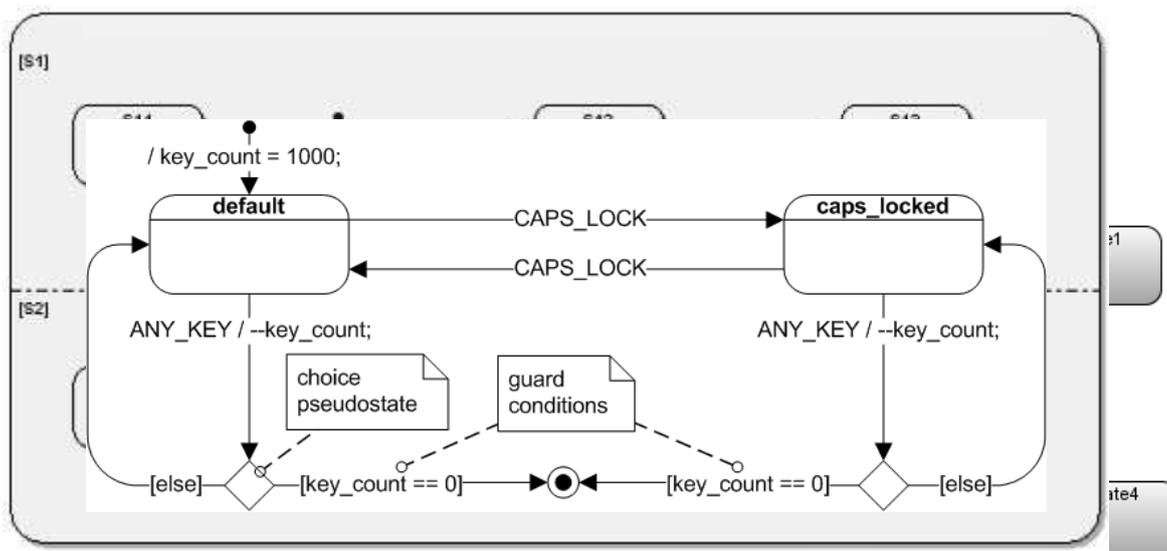


подсостояний (из разных параллельных областей).

Переходы могут разделяться (см. ниже переход из S11) и сливаться (см. переходы в S23). Переход из S11 в S12 осуществляется как обычно, когда активно S11. Переход в S23 осуществляется, когда одновременно активны S11 и S22.

Для сокращения количества переходов иногда применяют переходные псевдосостояния (черный кружок). На исходящих из него стрелках не могут быть отмечены события. В примере справа заданы 6 переходов. Каждый обозначается 2мя стрелками. Сторожевые условия получаются логическим умножением. То есть, имеем 3 перехода из State0 по событию e2: в State2 со сторожевым условием  $[b < 0 \text{ and } a < 0]$ , в State3 со сторожевым условием  $[b < 0 \text{ and } a = 5]$ , в State4 со сторожевым условием  $[b < 0 \text{ and } a > 7]$ . И 3 перехода из State1 по событию e1: в State2 со сторожевым условием  $[b < 0 \text{ and } a < 0]$ , в State3 со сторожевым условием  $[b < 0 \text{ and } a = 5]$ , в State4 со сторожевым условием  $[b < 0 \text{ and } a > 7]$ . Переход не может остановиться в переходном псевдосостоянии. Действия, приписанные частям сложного перехода, выполняются последовательно.

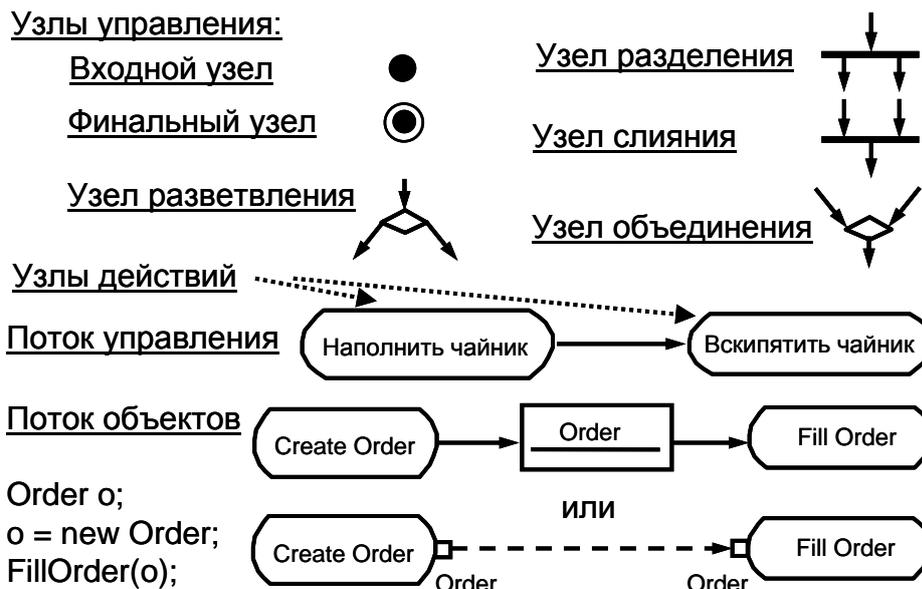
На диаграммах состояний могут применяться псевдосостояния выбора (choice pseudostate). Разница между переходным псевдосостоянием и псевдосостоянием выбора в том, что сторожевые условия на стрелках выходящих из выбора вычисляются после выполнения действий



на входящих стрелках (в случае с переходным псевдосостоянием все действия совершаются после вычисления условий). В примере диаграмма состояний описывает автомат ожидающий 1000 нажатий на клавиши без учета нажатий на CAPS LOCK.

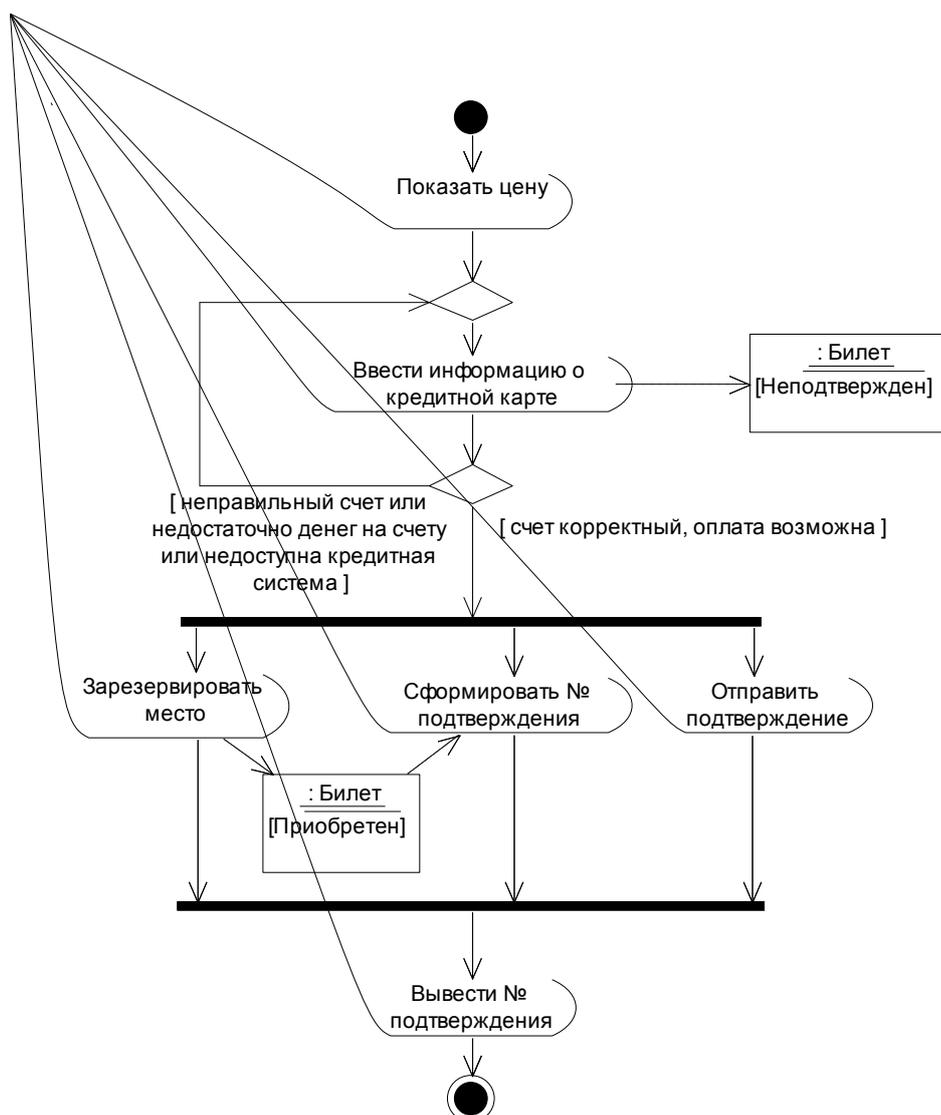
Диаграммы состояний создают лишь для классов, экземпляры которых имеют сложное поведение.

Диаграммы деятельности полезны в описании поведения, включающего большое количество параллельных процессов. Также их можно применять для представления потоков событий



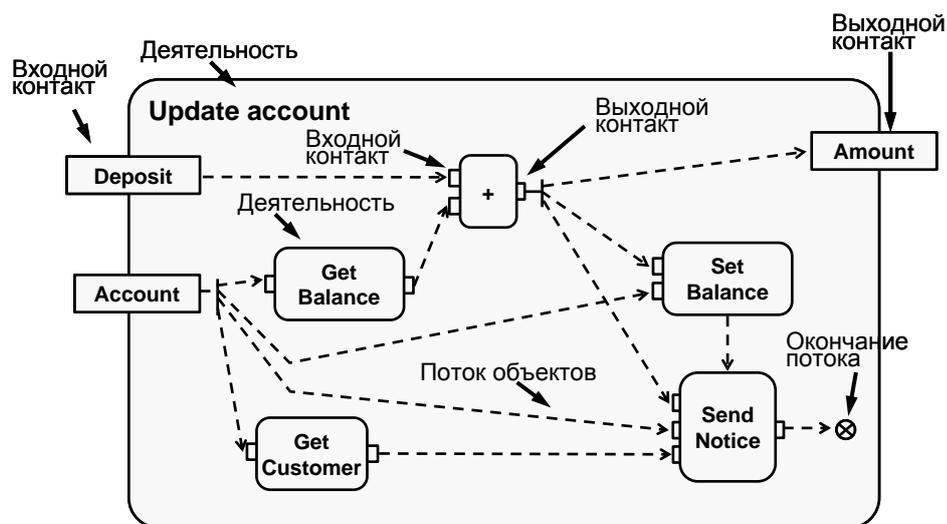
вариантов использования в наглядной графической форме. Элементы диаграмм деятельности:

В UML 2 проведена граница между диаграммами состояний, базирующимися на формализме конечных автоматов, и диаграммами деятельности, основанными на сетях Петри. Основным элементом диаграмм деятельности является узел действия. Каждый такой узел представляет собой элементарную единицу работы (это может быть решение некоторой задачи, которую необходимо выполнить вручную или автоматизированным способом, или выполнение метода класса). Узел действия изображается в виде закругленного прямоугольника с текстовым описанием. Диаграмма деятельности может иметь *входной узел*, вообще говоря, не один, (в UML 1 *должен* быть *ровно один*), определяющий начало потока управления. *Финальный узел* необязателен. На диаграмме может быть несколько *финальных узлов*. На диаграмме могут присутствовать объекты и потоки объектов, в тех случаях, если объект используется или изменяется в одном из узлов действий. Поток объектов отмечается сплошной или пунктирной стрелкой от узла действия к объектному узлу или от объектного узла к узлу действия, использующего объект. Ребра (сплошные стрелки) между узлами действий показывают потоки управления или потоки объектов. *Узел разветвления (узел принятия решения)*, а также *узел объединения* потоков изображается ромбом. Если необходимо показать, что две или более ветвей потока выполняются параллельно, используются «линейки синхронизации» – *узлы разделения* и *узлы слияния* (на рисунке – жирные горизонтальные линии).

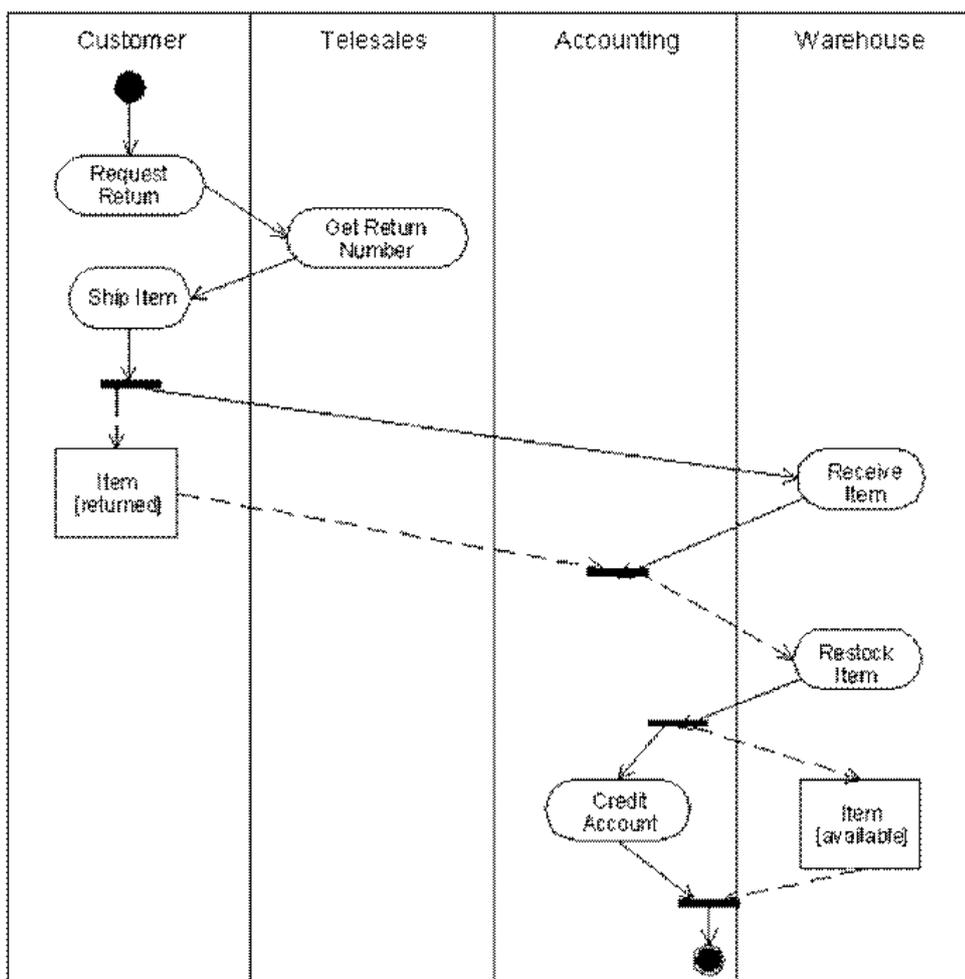


Как уже говорилось, диаграммы деятельности интерпретируются в соответствии с формализмом сетей Петри. Начальная точка порождает один курсор управления (или маркер) для каждого исходящего перехода. Если для ребра определено событие, то курсор достигает конца ребра только после наступления такого события. Ограничивающее (сторожевое) условие также определяет, возможно ли перемещение курсора по ребру. При попадании курсора в узел действия происходит ожидание курсоров на всех входящих ребрах и лишь потом запускается единица работы. По завершении выполнения работы генерируются курсоры на всех исходящих ребрах. При попадании в узел разветвления (он имеет один вход и несколько выходов) курсор проходит дальше лишь по тому ребру, для которого выполнено сторожевое условие (вообще говоря, лучше когда оно одно, если несколько – произвольно выбирается единственное для передачи курсора). При попадании в узел объединения курсор из любого входа копируется на выходе (единственном). При попадании в узел разделения курсор дублируется на все выходы одновременно (происходит распараллеливание). Синхронизация обеспечивается узлом слияния, где происходит ожидание курсоров на всех входах и лишь затем выдается курсор на выходе. При попадании курсора в любой финальный узел вся деятельность, описываемая диаграммой, прекращается. Примерно по тем же правилам перемещаются курсоры объектов. Узлы действия могут иметь входные и выходные контакты для приема/выдачи курсоров объектов,

изображаемые в виде квадратиков на границе узла. Если входных контактов несколько, действие в узле выполняется лишь тогда, когда на всех их пришли курсоры объектов. Пример:



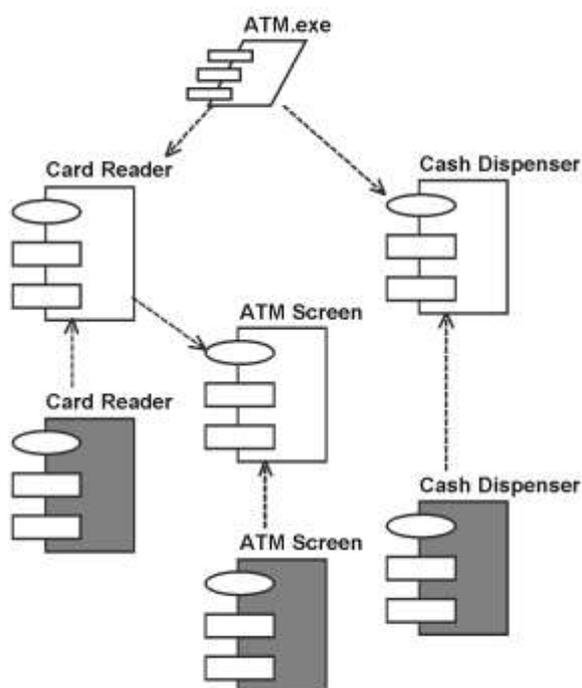
Узлы разделения и объединения могут синхронизировать потоки управления с потоками объектов. Например:



Здесь прием возвращаемой позиции заказа на склад синхронизируется с изменением состояния объекта Item, который должен перейти в состояние returned (возвращен). Аналогично, возврат денег по возвращенному заказу синхронизируется со сменой состояния объекта Item на *available* (доступен). Обратите внимание, диаграмма с помощью вертикальных линий – «плавательных дорожек» разделяется на зоны ответственности (заказчик, продажи, бухгалтерия, склад).

Диаграммы деятельности следует использовать в следующих ситуациях:

- при анализе потоков событий в конкретном варианте использования;
- при анализе потоков событий во взаимодействующих вариантах использования.



Диаграммы компонентов моделируют физический уровень системы. На них изображаются компоненты ПО и связи между ними. На такой диаграмме обычно выделяют два типа компонентов: исполняемые компоненты и библиотеки кода. Каждый класс модели (или подсистема) преобразуется в компонент исходного кода. Между отдельными компонентами изображают зависимости, соответствующие зависимостям на этапе компиляции или выполнения программы. В модели системы может быть несколько диаграмм компонентов, в зависимости от числа подсистем или исполняемых файлов. Каждая подсистема является пакетом компонентов. Диаграммы компонентов применяются теми участниками проекта, кто отвечает за компиляцию и сборку системы.

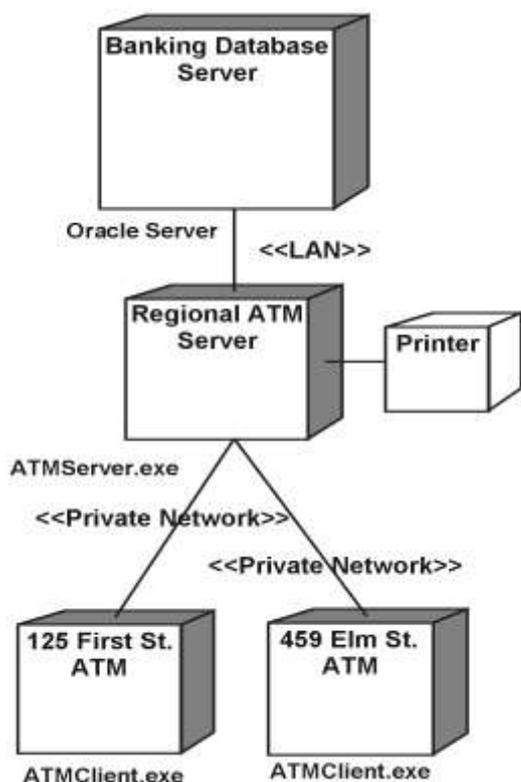


Диаграмма размещения отражает физические взаимосвязи между программными и аппаратными компонентами системы. Она является хорошим средством для того, чтобы показать размещение объектов и компонентов в распределенной системе. Ее основные элементы:

- узел (node) – вычислительный ресурс – процессор (изображаемый с закрашенными боковыми гранями) или устройство (дисковая память, контроллеры различных устройств и т.д.);
- соединение (connection) – канал взаимодействия узлов.

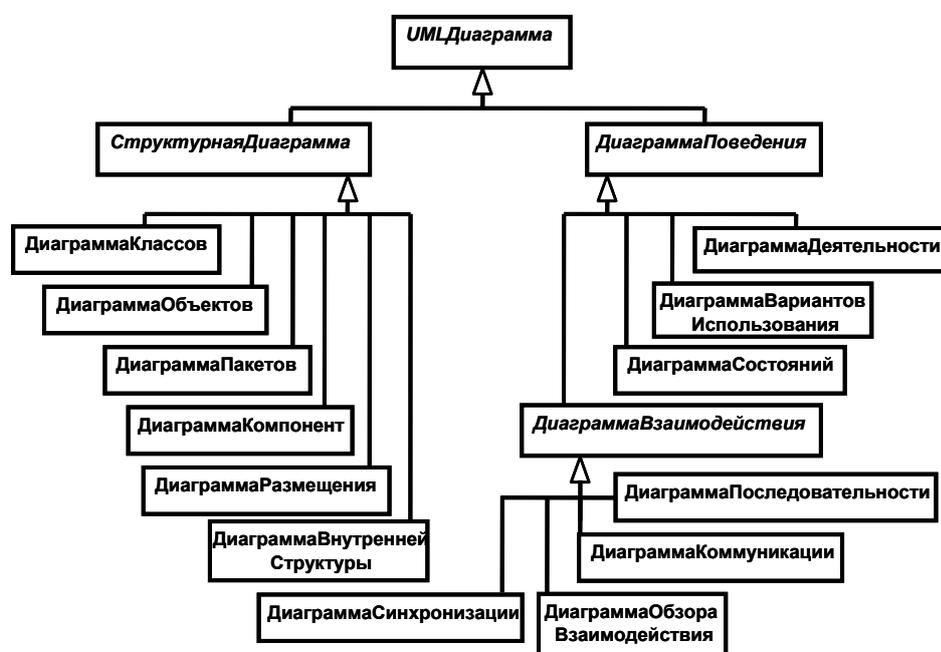
Для узла можно задать выполняющиеся на нем процессы.

Диаграмма размещения используется менеджером проекта, пользователями, архитектором системы и эксплуатационным персоналом (системными инженерами), чтобы понять физическое

размещение системы и распределение ее отдельных подсистем по вычислительным узлам.

При моделировании встроенных систем диаграмма размещения отображает связи микропроцессоров и устройств в составе системы.

Традиционные средства описания «линейных» языков (т. е. языков состоящих из цепочек символов) – БНФ, регулярные выражения, грамматики – плохо подходят для описания «плоского» языка, каковым является UML. Поэтому для описания UML применяется UML, такое описание называется метамоделью. Ниже приведен фрагмент метамодели. Подробнее о ней можно узнать



из стандарта.

Механизмы расширения UML предназначены для того, чтобы разработчики могли адаптировать язык моделирования к своим конкретным нуждам, не меняя при этом его основу (метамодель). Наличие механизмов расширения принципиально отличает UML от других средств моделирования и позволяет расширять его область применения. К механизмам расширения UML относятся: стереотипы; теги (именованные значения); примечания; ограничения.

*Стереотип* – это новый тип элемента модели, который определяется на основе уже существующего элемента. Стереотипы расширяют нотацию модели, могут применяться к любым элементам модели и представляются в виде текстовой метки или пиктограммы.

Стереотипы классов – это механизм, позволяющий разделять классы на категории. Например, основными стереотипами, используемыми в процессе анализа системы, являются: Boundary (граничный класс), Entity (класс-сущность) и Control (управляющий класс).

Помимо упомянутых стереотипов, разработчики ПО могут создавать свои собственные наборы стереотипов, формируя тем самым специализированные подмножества UML (например, для описания бизнес-процессов, Web-приложений, баз данных и т.д.). Такие подмножества (наборы стереотипов) в стандарте языка UML носят название профилей языка.

*Теги (именованные значения)* – специальные термины, используемые спецификации ограничений и свойств, такие как disjoint, complete, incomplete и др., могут сопровождаться указанием значения свойства, например, author=Вася или location=server.

*Примечание* – элемент диаграммы для комментария или другой текстовой информации. Примечание может содержать дополнительные сведения об элементах модели (с ними его соединяет пунктирная линия).

*Ограничение* – это семантическое ограничение, имеющее вид текстового выражения на естественном или формальном языке (OCL – Object Constraint Language), которое невозможно выразить с помощью нотации UML. Средства OCL не предназначены для описания процессов вычисления выражений, а только лишь фиксируют необходимость выполнения тех или иных условий применительно к отдельным компонентам моделей. Он может быть использован для решения следующих задач:

- описание инвариантов классов и типов в модели классов;
- описание пред- и постусловий в операциях и методах;
- описание ограничивающих условий элементов модели;
- навигация по структуре модели;
- спецификация ограничений на операции.

### Литература к лекции 3

1. Г. Буч, И. Якобсон, Дж. Рамбо «UML. Классика CS». 2-е изд. – СПб.: Питер, 2006.
2. Г. Буч, Дж. Рамбо, И. Якобсон «UML. Руководство пользователя». 2-е изд. – М.: ДМК Пресс, 2007
3. Рамбо Дж., Блаха М. UML 2.0. Объектно-ориентированное моделирование и разработка. 2-е изд.: Пер. с англ. – СПб.: Питер, 2006. – Главы 1, 2.
4. Арлоу Дж., Нейштадт А. UML 2 и Унифицированный процесс. Практический объектно-ориентированный анализ и проектирование – СПб. Символ-Плюс, 2007.
5. Фаулер М. UML. Основы. 3-е издание. Краткое руководство по стандартному языку объектного моделирования.: Пер. с англ. – СПб: Символ-Плюс, 2005.
6. Леоненков А. Самоучитель UML 2 – СПб.: БХВ-Петербург, 2007.
7. Шмюллер Дж. Освой самостоятельно UML за 24 часа – М.: Вильямс, 2005.

### Ссылки:

<http://www.uml.org>

<http://www.uml2.ru>

### Лекция 4. Объектный язык ограничений (OCL)

*Ограничение (constraint)* – это условие, накладываемое на значения одного или нескольких элементов модели. Ограничение не является инструкцией или командой, которую следует выполнить, оно формулируется как утверждение, которое должно быть истинным. Под элементом модели здесь имеется в виду объект, или класс, или пакет, или подсистема, или атрибут, или операция, или связь.

Рассмотрим пример:

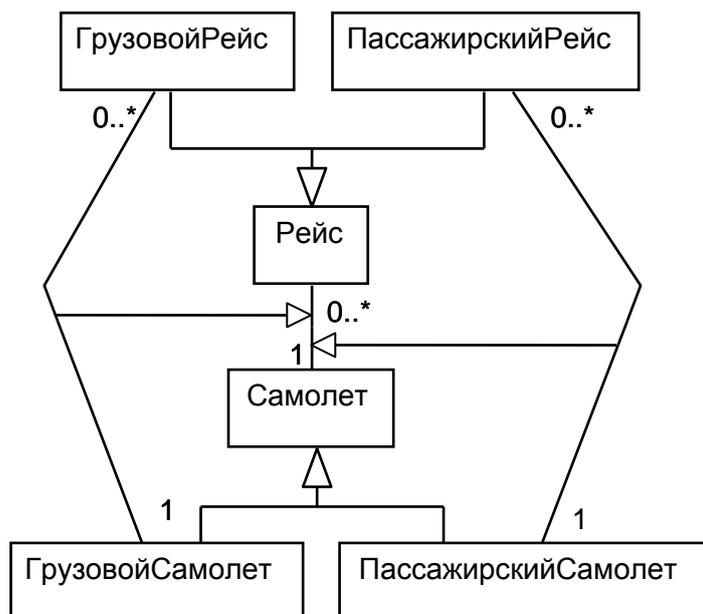


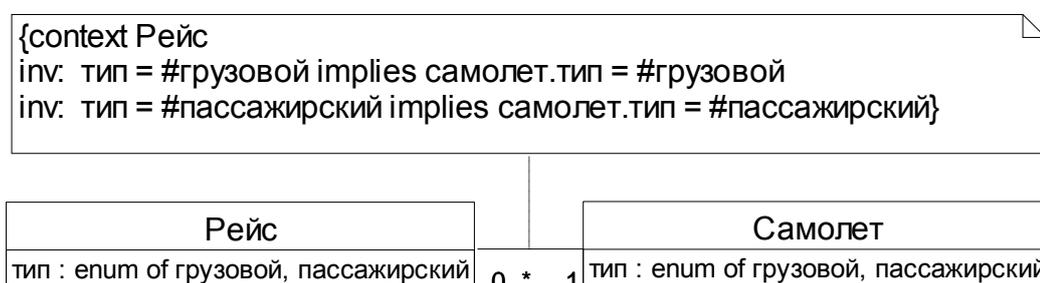
Диаграмма содержит большое количество связей (ассоциаций и обобщений) необходимых для указания, что тип самолета должен соответствовать типу рейса, т. е. что пассажиров нельзя перевозить грузовым самолетом. Это ограничение можно зафиксировать иначе, упростить



диаграмму, сделать ее более наглядной:

Ограничение, записанное на естественном языке, неформально, его можно неправильно трактовать (например, что чартерные рейсы должны выполняться старыми самолетами, а регулярные – новыми). Поэтому имеет смысл использовать для записи формальный язык, который не допускает произвольных толкований и имеет стандартный синтаксис и семантику. Таковым является объектный язык ограничений OCL (Object Constraint Language), являющийся одним из расширений UML. С использованием OCL диаграмма будет выглядеть так:

Слово *implies* означает логическую операцию импликации ( $a \rightarrow b$ , читается так: из  $a$  следует  $b$ , это выражение ложно лишь при  $a$  – истина и  $b$  – ложь, в остальных случаях оно истинно). Вообще говоря, для записи ограничений можно использовать и другие формальные языки, например, языки программирования. Основное неудобство при этом – ограничение, записанное на языке программирования, похоже на часть программы, что придает ограничению посторонний смысл



(может сложиться впечатление, что происходят какие-либо манипуляции над элементами модели, а это противоречит определению понятия ограничения).

Классификация ограничений:

- Инвариант класса – условие, которое всегда справедливо для всех экземпляров класса (ключевое слово **inv**:).
- Предусловие операции – условие, которое должно быть истинно перед выполнением операции (ключевое слово **pre**:).
- Постусловие операции – условие, истинное всегда после выполнения операции (ключевое слово **post**:).
- Тело запроса – описание результата операции-запроса, не модифицирующей объекты (ключевое слово **body**:)
- Начальное значение атрибута или соединения (ключевое слово **init**:)
- Правило вывода, описывающее производные атрибуты, связи или классы (ключевое слово **derive**:).
- Определение (ключевое слово **def**:).

В примере мы описали инварианты класса Рейс.

Характеристики OCL:

- текстовый (невизуальный) язык описания ограничений;
- формальный язык, часть стандарта UML;
- язык со строгой типизацией;
- декларативный язык (для ограничений не определяется конкретная процедура их проверки);
- платформи-независимый.

Никакое ограничение OCL не меняет состояния элементов модели (у него нет побочных эффектов), но с его помощью можно добавлять производные атрибуты и операции (используя **def**:).

Синтаксис OCL-выражения

<OCL-выражение> ::=

<указание контекста>

[[**inv** | **pre** | **post** | **body** | **init** | **derive** | **def**] : <тело выражения>]

В записи использованы символы языка БНФ: <> выделяют нетерминалы, ( | ) – вхождение одной из указанных альтернатив, [] вхождение 1 или более раз, {} – вхождение 0 или более раз.

Терминалы записаны жирным шрифтом.

*Контекст.* В любом OCL-выражении указывается определенный контекст. Как правило, контекстом является элемент модели (пакет, класс, атрибут, операция), с которым связано ограничение.

<указание контекста> ::= **context** <имя элемента модели>

В примере контекстом является класс Рейс.

Для того чтобы сослаться на контекст в теле выражения используется слово **self**. Чтобы много раз не писать **self**, оно часто опускается. По смыслу **self** аналогично **this** в C++. В примере **тип** – сокращенная запись **self.тип**.

В теле выражения используются

- выражения простых типов (boolean, integer, string, real);
- элементы модели, для которой составлено ограничение;
- коллекции.

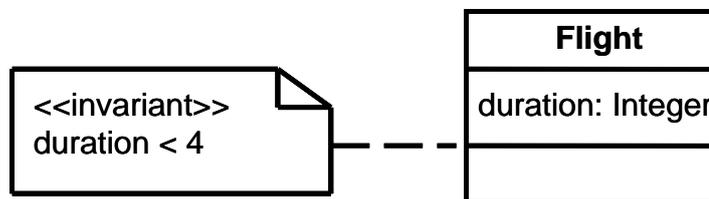
Логический тип OCL почти таков как в языках программирования. Есть дополнительные операции **xor** и **implies**. Приоритет логических операций (кроме **not**) меньше арифметических и операций сравнения – так проще записывать сложные логические выражения. Целый и вещественный типы также стандартны. Имеются дополнительные операции **a.max(b)** и **a.min(b)**, возвращающие максимум и минимум из двух чисел. Строки также похожи на строки языков программирования, только их нельзя сравнивать в лексикографическом порядке.

Примеры использования простых типов:

**context Airline inv: name.toLower = 'klm'** – здесь **'klm'** – строка, **toLower** – стандартная операция над строкой, дающая как результат строку в нижнем регистре. Смысл ограничения: у любого экземпляра класса *Airline* значение атрибута *name*, записанное строчными буквами совпадает со строкой **'klm'**.

**context Passenger inv: age >= ((9.6 - 3.5) \* 3.1).floor implies mature = true** – здесь **floor** – «округление вниз». Смысл ограничения: у любого экземпляра класса *Passenger* значения атрибутов *age* и *mature* таковы, что если *age* > 18, то *mature* = true.

Ограничения могут быть указаны на диаграмме в примечании, якорь примечания в таком случае играет роль указателя на контекст.. Например, ограничение **context Flight inv: self.duration < 4**



может изображаться на диаграмме так:

В OCL употребляются условные выражения:

<условное выражение> ::=

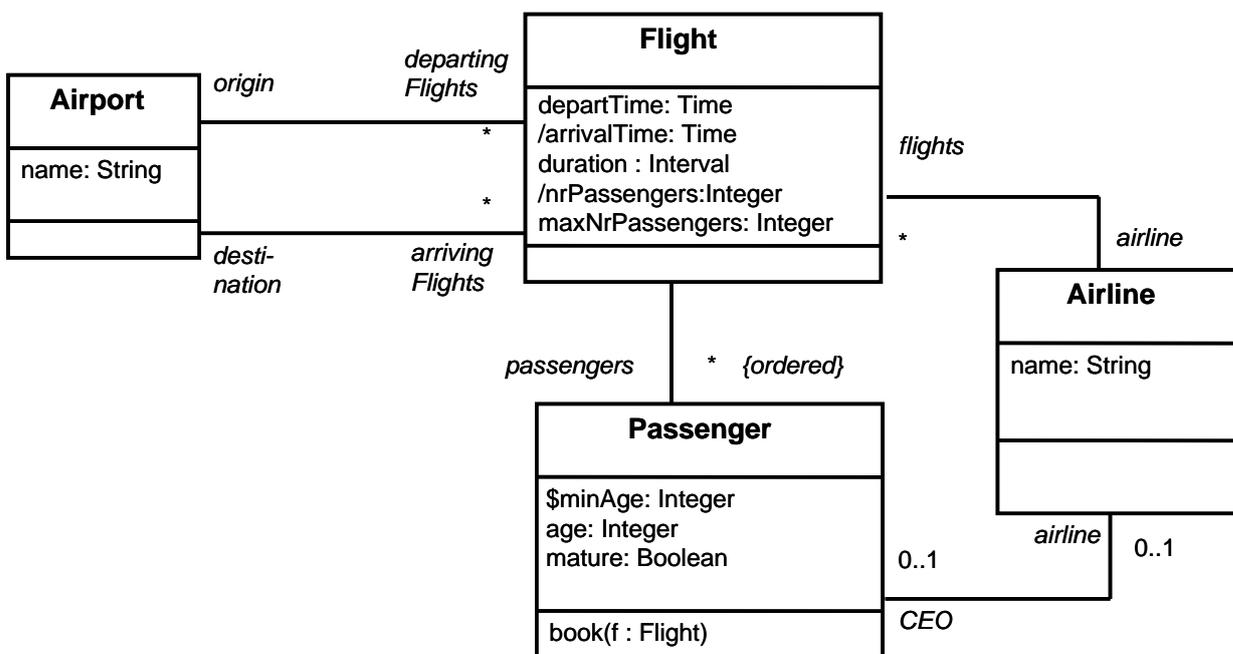
**if** <логическое выражение> **then** <выражение>

**else** <выражение>

**endif**

Пример: **if (x >= 0) then x else -x endif** возвращает модуль *x* или *x.abs()*.

В телах OCL-выражений используются типы и имена (классов, атрибутов, операций) из модели. Далее в примерах мы будем рассматривать модель авиаперевозок:



Обратите внимание на производные атрибуты в классе Flight (arrivalTime, nrPassengers) и статический атрибут minAge класса Passenger.

Для указания атрибута или операции используется выражение с точкой:

<выражение>.<имя>

**context Flight inv: self.maxNrPassengers <= 1000** – в любом рейсе максимальное количество пассажиров не превышает 1000 (использован атрибут объекта – экземпляра класса **Flight**).

**context Flight:maxNrPassengers:Integer init: 1000** – в любом рейсе максимальное количество пассажиров по умолчанию = 1000.

**context Passenger inv: self.age >= Passenger::minAge** – у любого пассажира возраст больше минимального (использован атрибут **age** экземпляра класса **Passenger** и атрибут того же класса **minAge**).

Примеры с производными атрибутами:

**context Flight def: arrivalTime:Time = departTime.plus(duration)**

или

**context Flight:arrivalTime derive: departTime.plus(duration)**

По смыслу ограничения совпадают, но в первом определяется производный атрибут, которого не было в модели, во втором определяется правило вывода для атрибута в составе модели. В примерах использован класс Time.

Time
\$midnight: Time
month : String
day : Integer
year : Integer
hour : Integer
minute : Integer
difference(t:Time):Interval
before(t: Time): Boolean
plus(d : Interval) : Time

Пример определения операции:

**context Interval::equals(i:Interval):Boolean body:**

**(self.nrOfDays\*60+self.nrOfHours)\*60+self.nrOfMins =**

Interval
nrOfDays : Integer
nrOfHours : Integer
nrOfMins : Integer
equals(i:Interval):Boolean
\$Interval(d, h, m : Integer) : Interval

$(i.nrOfDays*60+i.nrOfHours)*60+i.nrOfMins$

Заметим, что данное ограничение не описывает, вообще говоря, как именно проверяются интервалы на совпадение, допускается любой способ, который дает результат, совпадающий со значением из ограничения.

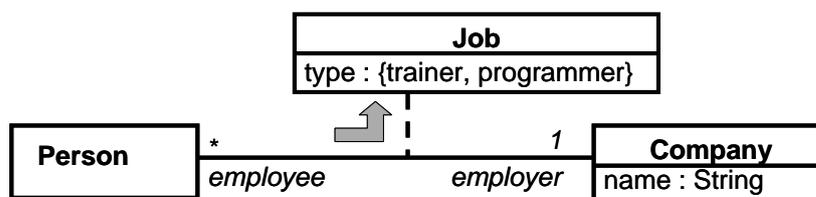
Поскольку ограничения часто накладываются не только на объекты классов, но и на связанные с ними объекты других классов, каждая ассоциация рассматривается как путь навигации. Контекст выражения является стартовой точкой. Имя роли определяет, по какой ассоциации осуществляется навигация (если их несколько), если ассоциация одна, то используется имя класса на другом конце ассоциации. Пример:

**context Flight**

**inv: self.origin <> self.destination**

**inv: self.origin.name = 'Amsterdam'** – у любого рейса аэропорт назначения и аэропорт вылета не совпадают, а также аэропорт вылета называется 'Amsterdam'.

Если есть класс ассоциаций, то используется его имя. Если ассоциация квалифицированная, то после имени роли в квадратных скобках указывают значения квалификаторов.



**context Person inv:**

**if employer.name = 'ОАО МММ' then**

**Job.type = #trainer**

**else**

**Job.type = #programmer**

**endif**

Смысл ограничения: у любой персоны, занятой в 'ОАО МММ', тип работы trainer, у остальных тип работы programmer. То же самое по смыслу ограничение можно записать, используя в качестве контекста класс ассоциаций:

**context Job inv:**

**if employer.name = 'ОАО МММ' then**

**self.type = #trainer**

**else**

**self.type = #programmer**

**endif**

При навигации по связям, если на другом конце указана мощность связи \*, от одного объекта мы приходим к нескольким связанным с ним (например, один аэропорт является аэропортом вылета для нескольких рейсов), поэтому в OCL введено понятие коллекции. Вообще говоря, коллекции могут состоять либо из объектов, либо из элементов простых типов, либо из элементов типов, определенных в модели, либо из коллекций. Виды коллекций:

- **Set** (множество– неупорядоченный набор без повторов) – для экземпляра класса **Airport** прибывающие рейсы составляют множество объектов **Flight**.
- **Bag** (неупорядоченный набор с повторами) – для экземпляра класса **Airport** длительности всех прибывающих рейсов составляют bag вещественных значений.
- **OrderedSet** (упорядоченный набор без повторов) – для экземпляра класса **Flight** все его пассажиры составляют orderedSet объектов **Passenger**.
- **Sequence** (упорядоченный набор с повторами) – для экземпляра класса **Flight** возрасты всех его пассажиров составляют sequence целых значений.

В OCL имеется большое количество predefined операций над коллекциями (isEmpty, size, includes, union ...). Синтаксис: <коллекция> -> <операция>

Операция **collect** возвращает коллекцию значений, полученных при вычислениях выражения для всех элементов коллекции. Запись: <коллекция>->**collect**(<выражение>) – здесь и далее круглые скобки и | – символы OCL, а не языка БНФ. Сокращенная запись: <коллекция>.<выражение>

Пример: **context Airport inv: self.arrivingFlights -> collect(airLine) -> nonEmpty** – для любого аэропорта множество авиакомпаний, выполняющих прибывающие рейсы, не пусто.

Операция **select** возвращает совокупность тех элементов коллекции, для которых <выражение> истинно. Запись: <коллекция>->**select**(<выражение>)

Пример: **context Airport inv: self.departingFlights->select(duration<4)->notEmpty** – для любого аэропорта есть хоть один отправляющийся рейс длительностью менее 4 часов.

Операция **forall** возвращает **true** если <выражение> истинно для всех элементов коллекции. Запись: <коллекция>->**forall**(<выражение>)

Пример: **context Airport inv: self.departingFlights->forall(maxNrPassengers < 1000)** – для любого аэропорта справедливо, что у любого отправляющегося рейса максимальное количество пассажиров < 1000.

Операция **exists** возвращает **true** если хотя бы для одного элемента коллекции <выражение> истинно. Запись: <коллекция>->**exists**(<выражение>)

Другие операции над коллекциями:

- **isEmpty**: истина, если коллекция пуста;
- **notEmpty**: истина, если коллекция непуста;
- **size**: количество элементов коллекции;
- **sum**: сумма элементов коллекции чисел;
- **count**(<элемент>): количество вхождений элемента;
- **includes**(<элемент>): истина, если <элемент> входит в коллекцию;
- **excludes**(<элемент>): истина, если <элемент> отсутствует в коллекции;
- **includesAll**(<коллекция>): истина, если все элементы параметра <коллекция> входят в

коллекцию.

Коллекции можно сравнивать на = и <>. Преобразование коллекций к другому виду осуществляется при помощи операций asSet(), asBag(), asOrderedSet(), asSequence(). Преобразование типов осуществляется с помощью операцииoclAsType(type). Коллекцию коллекций можно сделать «плоской» с помощью операции flatten(). Примеры ее работы:

Set { Set { 1, 2 }, Set { 2, 3 }, Set { 4, 5, 6 } } → flatten → Set { 1, 2, 3, 4, 5, 6 }

Bag { Set { 1, 2 }, Set { 1, 2 }, Set { 4, 5, 6 } } → flatten → Bag { 1, 1, 2, 2, 4, 5, 6 }

Sequence { Set { 1, 2 }, Set { 2, 3 }, Set { 4, 5, 6 } } → flatten → Sequence { 2, 1, 2, 3, 5, 6, 4 }

Дополнительные возможности OCL:

- result и @pre в постусловиях;
- локальные переменные;
- итераторы;
- наследование;
- указание состояний объектов.

С помощью **result** в постусловии можно указать, что операция возвращает в качестве результата:

```
context Airline::servedAirports() : Set(Airport)
```

```
pre : --none
```

```
post: result = flights.destination->asSet
```

**@pre** в постусловии дает возможность использовать значения атрибутов, какими они были в начале выполнения операции

```
context Passenger::Book(f:Flight)
```

```
pre : Flight.nrPassengers < Flight.maxNrPassengers
```

```
post: Flight.nrPassengers = Flight.nrPassengers@pre + 1
```

Конструкция **let** определяет локальную переменную. Запись:

```
let <name> : <тип> = <expression1> in <expression2>
```

Пример: **context Airport inv: let supportedAirlines : Set(Airline) = self.arrivingFlights ->collect(airLine) in (supportedAirlines ->notEmpty) and (supportedAirlines ->size < 500)** – здесь для упрощения логического выражения определена локальная переменная **supportedAirlines**, представляющая собой коллекцию авиалиний, обслуживающих, прибывающие в аэропорт рейсы. Указано, что для любого аэропорта таких авиалиний должно быть больше нуля, но меньше 500.

Конструкция **iterate** позволяет описывать нестандартные операции над коллекциями. Запись: <коллекция>->

```
iterate(<переменная1> : <тип>; <переменная2> : <тип> [= <нач. значение>] | <тело>)
```

где <переменная1> – параметр итератора, <переменная2> – результат итератора, <тело> – OCL-выражение с <переменная1> и <переменная2>.

Например, ограничение:

```
context Airline inv: self.flights->select(maxNrPassengers > 150)->notEmpty
```

идентично:

```
context Airline inv: self.flights->iterate (f : Flight; answer : Set(Flight) = Set{ } |
```

```
if f.maxNrPassengers > 150 then answer->including(f) else answer endif )->notEmpty
```

Поясним второе OCL-выражение. Для авиакомпании будет собрана коллекция всех ее рейсов и на этой коллекции будет запущен итератор. В начале работы результат итератора инициализируется пустым множеством. Затем для каждого рейса  $f$  из коллекции, одного за другим, будет вычислено условное выражение, которое добавит в результат лишь те рейсы,  $\text{maxNrPassengers}$  которых больше 150.

В наследовании ограничений работает принцип подстановки Лисковской (Liskov's Substitution Principle): «Где может находиться экземпляр суперкласса, туда всегда может быть подставлен экземпляр его любого подкласса.» Это означает, что:

- Инвариант суперкласса наследуется любым подклассом.
- Подклассы могут усиливать инвариант.
- Предусловие может быть ослаблено в подклассе.
- Постусловие может быть усилено в подклассе.

Если в ограничении требуется проверить, является ли экземпляр суперкласса также экземпляром конкретного подкласса, то используют стандартную операцию **oclIsTypeOf**. Вспоминая пример, приведенный в начале лекции, мы можем описать следующие ограничения:

```
context ГрузовойСамолет inv:
```

```
Рейс->forAll(r | r.oclIsTypeOf(ГрузовойРейс))
```

```
context ПассажирскийСамолет inv:
```

```
Рейс->forAll(r | r.oclIsTypeOf(ПассажирскийРейс))
```

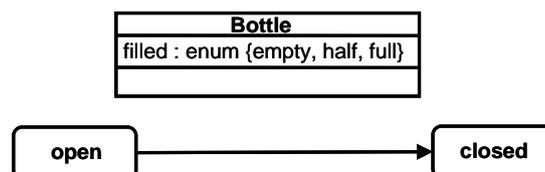
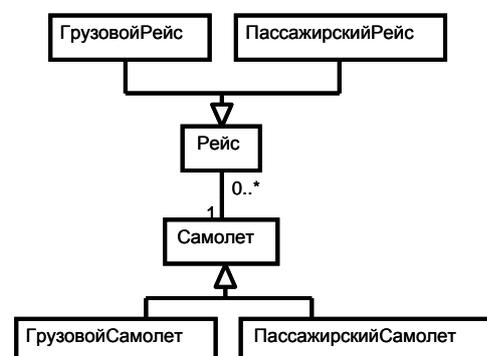
Операция **oclInState** возвращает истину, если объект находится в определенном состоянии. Пример:

```
context Bottle inv:
```

```
self.oclInState(closed) implies filled = #full
```

Мы рассмотрели OCL применительно к моделям (диаграммам) классов. Он также может использоваться на других диаграммах. На

диаграммах взаимодействия OCL применяют для записи сторожевых условий (от выполнения которых зависит, будет ли послано сообщение). На диаграммах деятельности OCL применяют для описания действий, узлов принятия решения, сторожевых условий на потоках. На диаграммах состояний OCL используется для описания состояний и сторожевых условий на переходах между состояниями.



Подведем итоги.

- OCL позволяет уточнять модель, формулировать запросы к модели, сохранять при этом свободу реализации.
- Пре- и постусловия OCL позволяют точнее описывать интерфейсы и компоненты.
- Рекомендуется при использовании OCL писать простые ограничения, совместно использовать OCL и естественный язык, применять CASE-средства, поддерживающие OCL (например, из состава Eclipse Model Development Tools или Dresden OCL Toolkit).

#### **Литература к лекции 4**

1. J. Warmer, A. Kleppe. Object Constraint Language, The: Getting Your Models Ready for MDA, Second Edition
2. OCL portal <http://www-st.inf.tu-dresden.de/ocl/>
3. Рамбо Дж., Блаха М. UML 2.0. Объектно-ориентированное моделирование и разработка. 2-е изд.: Пер. с англ. – СПб.: Питер, 2007. – Глава 3.
4. Дж. Арлоу, Ай. Нейштадт. UML 2 и унифицированный процесс, 2-е изд. – СПб.: Символ-Плюс, 2007. – Глава 25.

#### **Лекция 5. Моделирование бизнес-процессов**

Моделирование бизнес-процессов является важной составной частью крупномасштабных проектов по созданию ПО. Отсутствие таких моделей является одной из главных причин неудач многих проектов.

*Бизнес-процесс (производственный процесс)* определяется как логически завершённый набор взаимосвязанных и взаимодействующих видов деятельности, поддерживающий деятельность организации и реализующий ее политику, направленную на достижение поставленных целей. Бизнес-процесс использует определенные ресурсы (финансовые, материальные, человеческие, информационные). Выделяют следующие классы процессов:

- основные процессы (производство товаров и услуг, приносят доход, составляют основную деятельность компании);
- обеспечивающие процессы (обеспечение основных процессов финансами, кадрами, комплектующими, тех. обслуживанием, администрирование и юридическое обеспечение);
- процессы управления (планирование и контроль бизнес-процессов других видов).

Бизнес-модель – это формализованное описание бизнес-процессов предприятия, фиксирующее существующее положение дел (модель AS-IS «как есть») или устанавливающее новые усовершенствованные способы осуществления деятельности (модель AS-TO-BE «как будет»).

Цели бизнес-моделирования:

- 1) обеспечить понимание структуры организации и происходящих в ней процессов;
- 2) обеспечить понимание текущих проблем организации и возможностей их решения;
- 3) убедиться, что заказчики, пользователи и разработчики одинаково понимают цели и задачи организации;
- 4) создать базу для формирования требований к будущему ПО организации.



Рис. Аналитик бизнес-процессов, его деятельность и рабочие документы.

Бизнес-модель должна давать ответы на вопросы:

1. Какие процедуры (функции, работы) необходимо выполнить для получения заданного конечного результата?
2. В какой последовательности выполняются эти процедуры?
3. Какие механизмы контроля и управления существуют в рамках рассматриваемого бизнес-процесса?
4. Кто выполняет процедуры процесса?
5. Какие входящие документы/информацию использует каждая процедура процесса?
6. Какие исходящие документы/информацию генерирует процедура процесса?
7. Какие ресурсы необходимы для выполнения каждой процедуры процесса?
8. Какая документация/условия регламентирует выполнение процедуры?

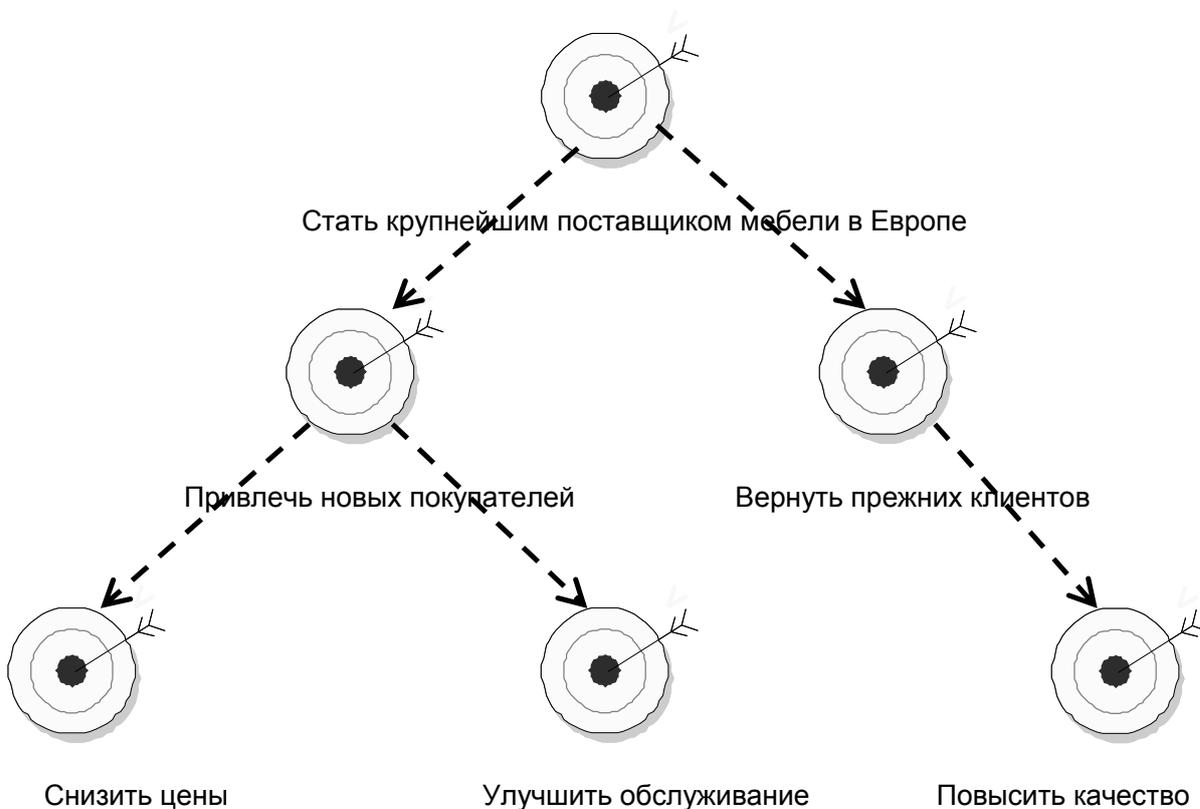
Рассмотрим методику моделирования деловых процессов, являющуюся составной частью технологии Rational Unified Process.

Аналитик бизнес-процессов возглавляет и координирует бизнес-моделирование. Он отвечает за:

- видение бизнеса – документ, где определены цели бизнес-моделирования;
- оценку организации – документ, описывающий текущее состояние дел в организации;
- бизнес-правила – условия, соблюдение которых необходимо;
- глоссарий деятельности – словарь основных терминов организации;
- дополнительную спецификацию – документ со сведениями, не вошедшими в другие документы;
- модель бизнес-процессов (Business Use Case Model), моделирующую взгляд на предприятие извне, как на «черный ящик»;
- модель бизнес-анализа (Business Analysis Model), моделирующую взгляд на предприятие

изнутри, как на «белый ящик».

Модель бизнес-целей представляет собой древовидную структуру, описывающую зависимости вида цель-подцель (см. рисунок). Связи в дереве таковы, что достижение подцелей приводит или приближает к достижению родительской бизнес-цели. Дерево целей изображается на диаграмме



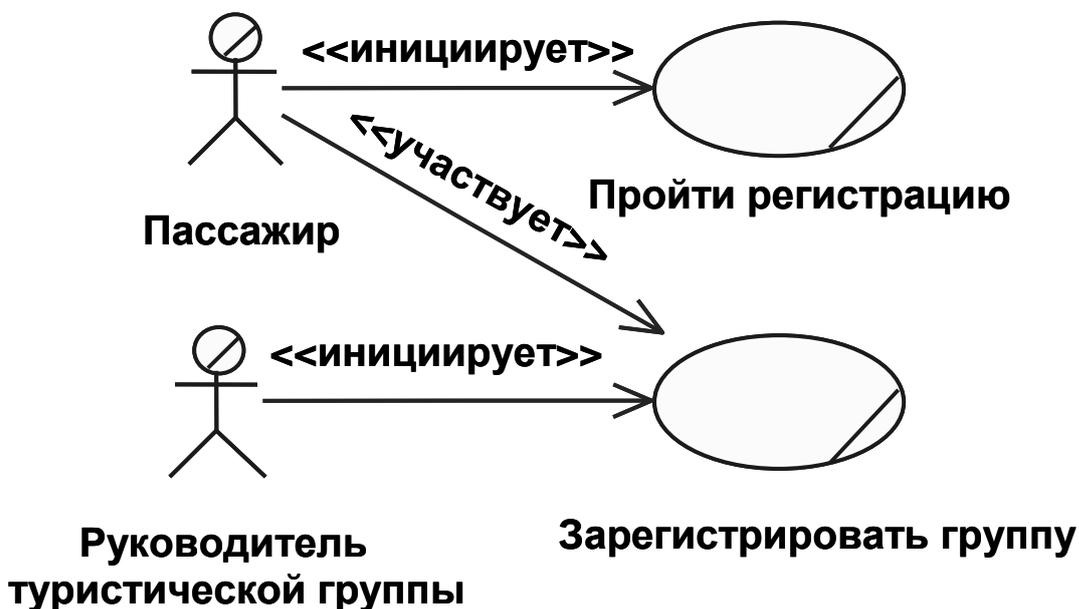
классов.

Модель бизнес-процессов (Business Use Case Model) – модель, описывающая бизнес-процессы организации в терминах ролей и их потребностей. Она представляет собой расширение модели вариантов использования UML за счет введения набора стереотипов Business Actor (деловое действующее лицо – стереотип действующего лица) и Business Use Case (бизнес-процесс – стереотип варианта использования). Из этой модели видно в каком контексте работает предприятие, но не видно как именно протекает его работа (это описывает модель бизнес-анализа).

Деловое действующее лицо (business actor) – некоторая роль, выполняемая по отношению к бизнес-процессам организации. Кандидатами на эту роль являются: акционеры, заказчики, поставщики, партнеры, потенциальные клиенты, местные органы власти, коллеги из подразделений, не охваченных моделью, внешние бизнес-системы (предприятия или подразделения). Деловыми действующими лицами, как правило, не являются должностные лица, работающие на предприятии. Обнаружить действующих лиц бизнес-процессов можно, найдя ответы на вопросы:

- Кто извлекает пользу из существования организации?
- Кто помогает организации осуществлять свою деятельность?
- Кому организация передает информацию и от кого получает?

Бизнес процесс (Business use-case) описывает последовательность действий в рамках экономической деятельности предприятия, приносящую ощутимый результат конкретному деловому действующему лицу.



Пример модели бизнес-процессов (регистрация пассажиров на рейс в аэропорту):

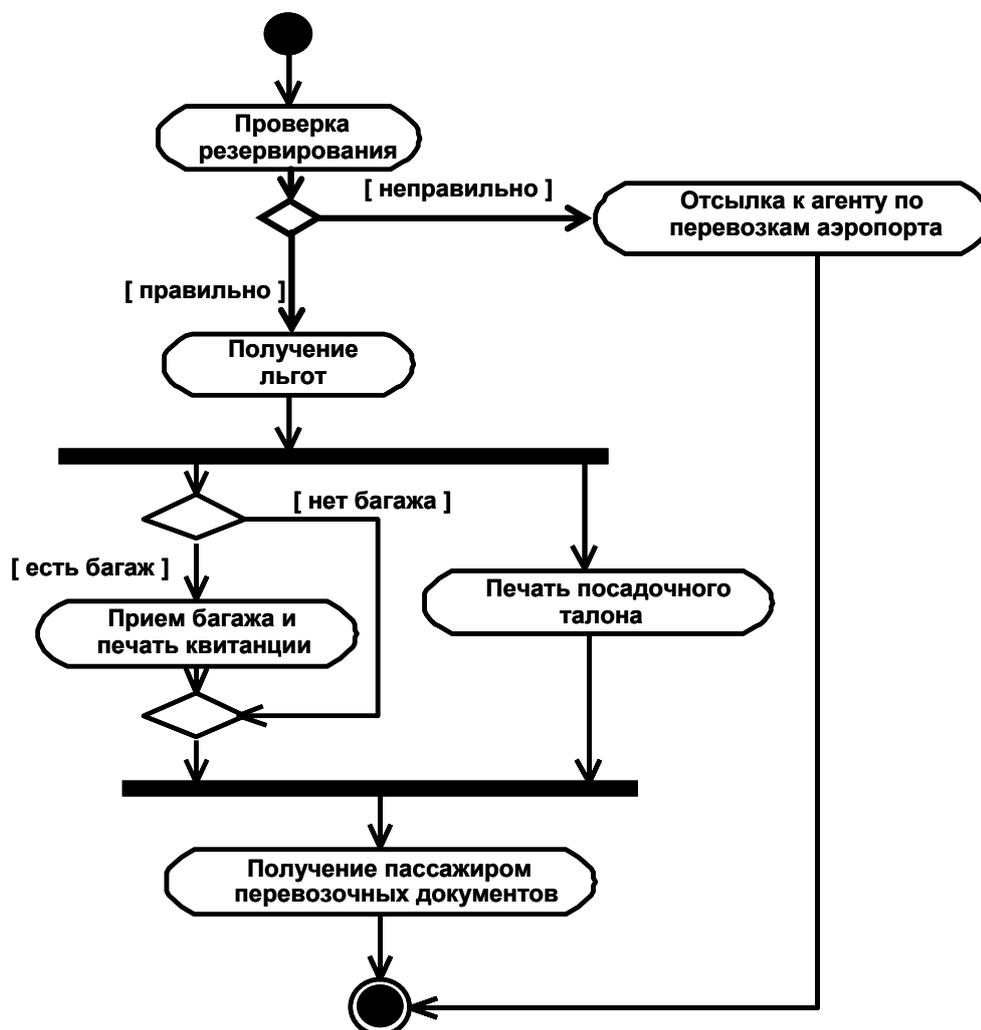
Каждый бизнес-процесс сопровождается спецификацией, в которой содержится:

- наименование;
- краткое описание бизнес-процесса;
- цели и результаты;
- описание сценариев (основного и альтернативных);
- специальные требования (время и стоимость);
- расширения (исключительные ситуации);
- связи;
- диаграммы деятельности (моделирующие сценарии бизнес-процесса).

Пример:

- Наименование – Пройти регистрацию.
- Краткое описание – Процесс регистрации пассажира на рейс.
- Цели – Получить посадочный талон и сдать багаж.
- Основной сценарий:
  1. Пассажир встает в очередь к стойке регистратора.
  2. Пассажир предъявляет билет регистратору.
  3. Регистратор подтверждает правильность билета.
  4. Регистратор оформляет багаж.
  5. Регистратор резервирует место для пассажира.
  6. Регистратор печатает посадочный талон.
  7. Регистратор выдает пассажиру посадочный талон и квитанцию на багаж.
  8. Пассажир уходит от стойки регистратора.
- Альтернативные сценарии:
  - 1а. Билет неправильно оформлен – регистратор отправляет пассажира к агенту по перевозкам.
  - 2а. Багаж превышает установленный вес – регистратор оформляет доплату.

- Специальные требования – Время регистрации не должно превышать 1 минуты.
- Модель бизнес-процессов может быть структурирована: при необходимости вводятся связи обобщения между действующими лицами и связи включения и расширения между бизнес-вариантами использования. Для моделирования сценариев бизнес-варианта по отдельности или в



совокупности используются диаграммы деятельности:

Модель бизнес-анализа (модель бизнес-объектов) создается другим исполнителем в рамках RUP – бизнес-разработчиком, но руководит ее созданием бизнес-аналитик.

Бизнес-разработчик выполняет следующие деятельности:

- работает над бизнес-системой (отделом или подразделением организации);
- уточняет спецификацию бизнес-процессов (business use case);
- моделирует реализацию бизнес-процессов в виде модели бизнес-анализа (business analysis).

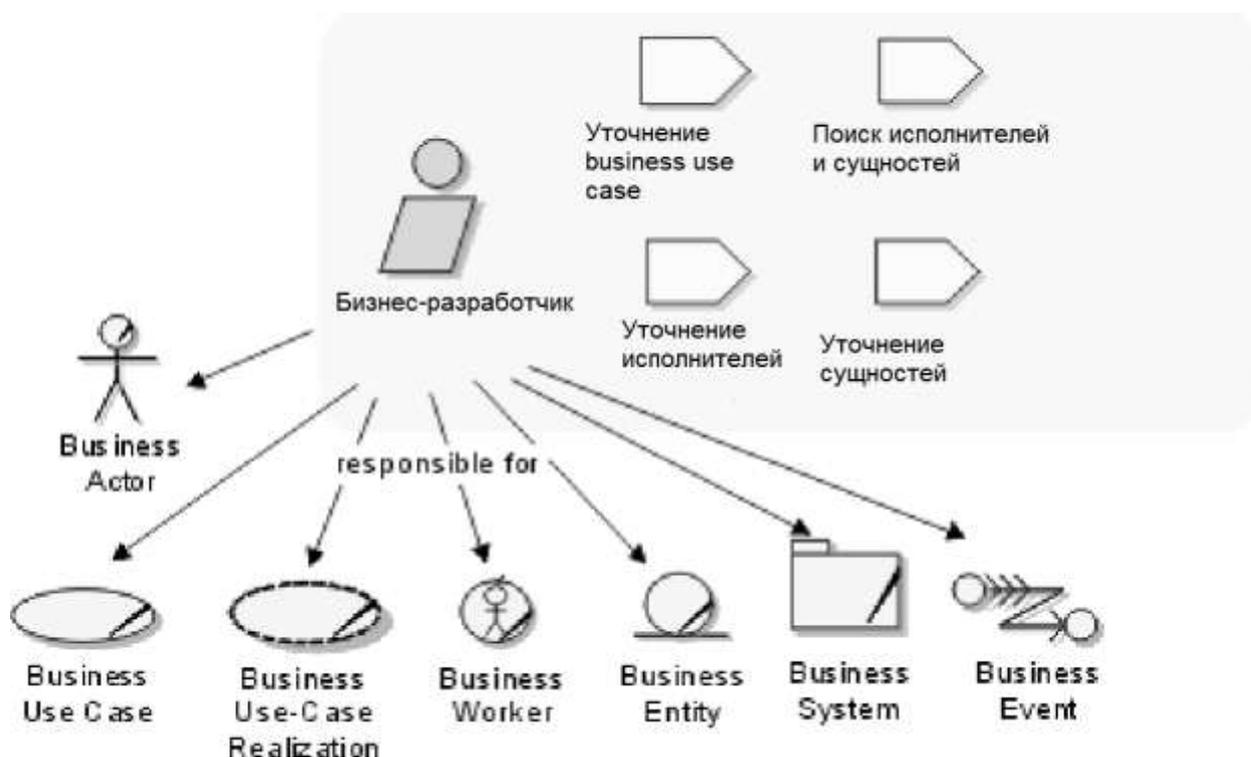


Рис. Деятельности, выполняемые бизнес-разработчиком и рабочие документы, создаваемые им.

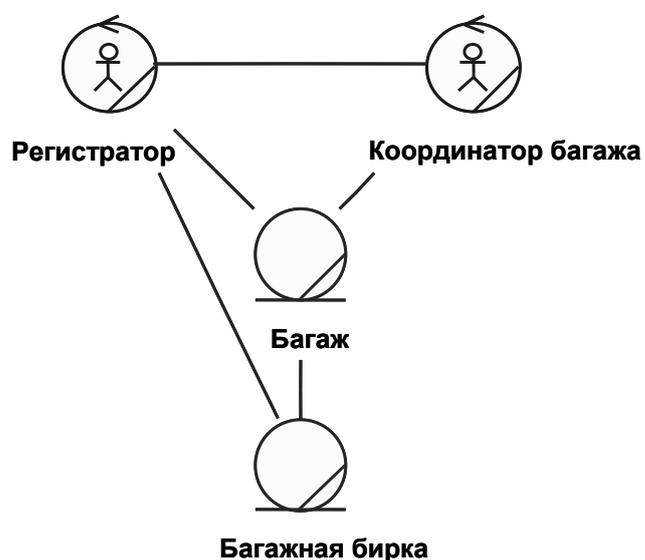
Модель бизнес-анализа – это объектная модель, элементами которой являются исполнитель (business worker) и бизнес-сущность (business entity). Эта модель описывает внутреннее устройство бизнес-процессов с точки зрения структуры и поведения. Но из этой модели нельзя понять деловое окружение предприятия (что описано моделью бизнес-процессов).

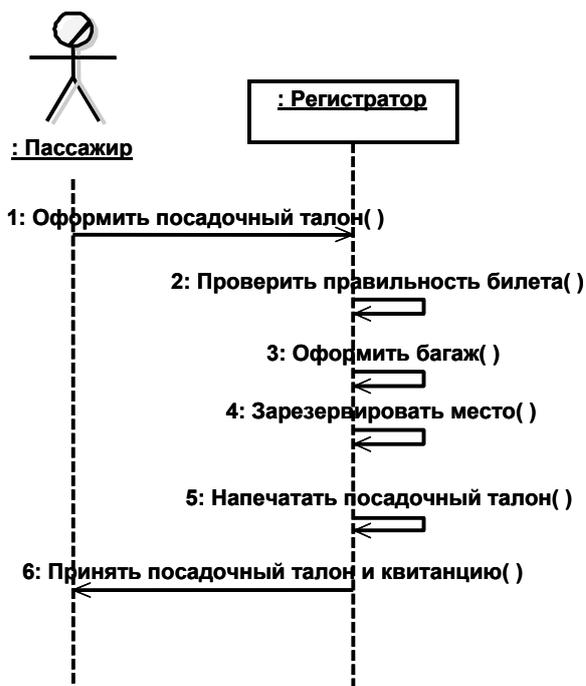
Business worker – исполнитель, действующий в рамках бизнес-системы. В отличие от делового действующего лица исполнитель работает на предприятии. Он имеет связи взаимодействия с другими исполнителями и манипулирует бизнес-сущностями, участвуя в реализациях бизнес-процессов. Представляется на диаграммах как класс со стереотипом «business worker».

Деловая сущность (Business entity) – это ресурс (информационный, материальный, финансовый и т. д.), не инициирующий никаких взаимодействий, он может участвовать во многих реализациях различных бизнес-процессов и является предметом различных манипуляций со стороны исполнителей. На диаграммах представлен классом со стереотипом «business entity».

Модель бизнес-анализа включает в себя диаграммы разных видов:

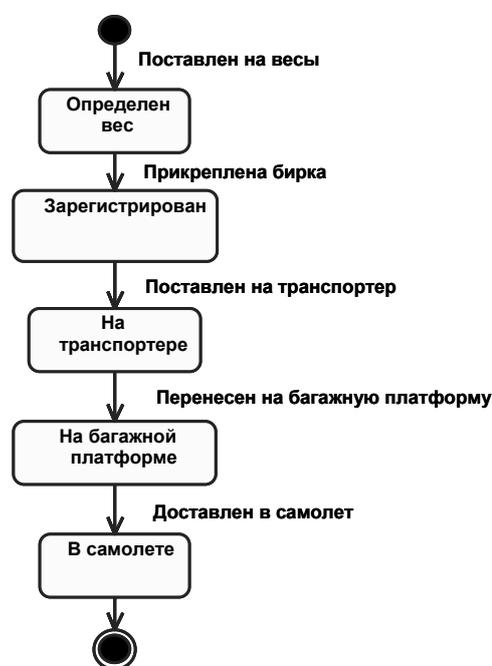
- диаграммы классов, отражающих структурные соединения, из которых следует как взаимосвязаны исполнители и деловые сущности (из примера следует, что исполнители двух разных классов могут взаимодействовать между собой, что исполнитель первого класса имеет доступ к деловым сущностям багаж и Багажная бирка, а исполнитель второго класса – только к Багажу, что сущности связаны между собой.);





- диаграммы взаимодействия (последовательности, кооперативные), описывающие реализацию одного из сценариев бизнес-процесса, и служащие для моделирования распределения обязанностей между исполнителями (например, диаграмма последовательности может изображать реализацию основного потока событий бизнес-процесса Пройти регистрацию, и на ней видно какими сообщениями обмениваются экземпляр делового действующего лица и экземпляр исполнителя, а также видно, обработка каких сообщений входит в обязанности исполнителя);

- диаграммы состояний для моделирования жизненного цикла экземпляров того или иного исполнителя или экономического ресурса, т. е. деловой сущности, (в примере видно, что экземпляр деловой сущности Багаж меняет свои

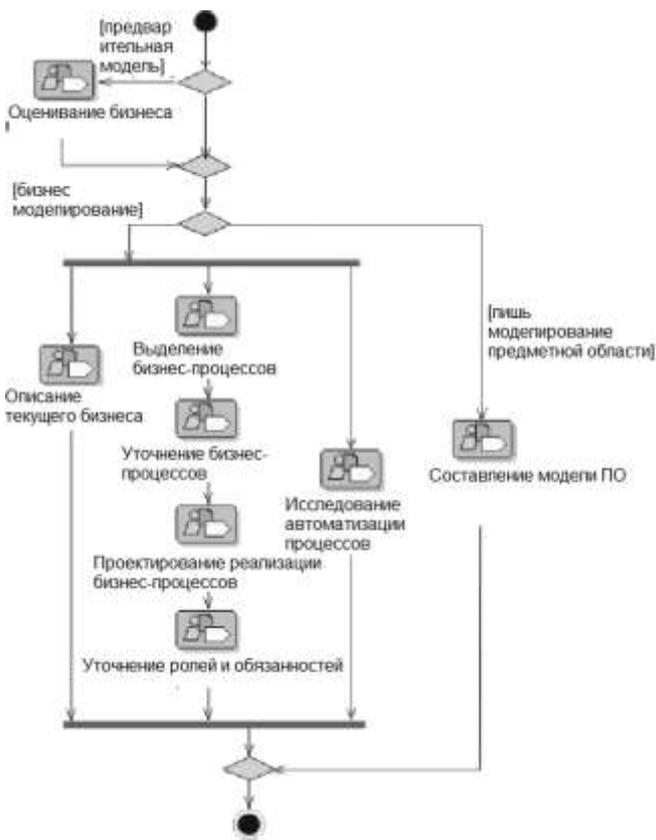


внутренние состояния, принимая сообщения от исполнителей, изображенные как события «Поставлен на весы», «Прикреплена бирка» и т. д.);

- диаграммы деятельности, моделирующие выполнение исполнителями своих обязанностей (например, как именно регистратор проверяет правильность билета). Ход бизнес-моделирования в целом отображает следующая диаграмма деятельности:

При оценивании бизнеса создаются следующие документы:

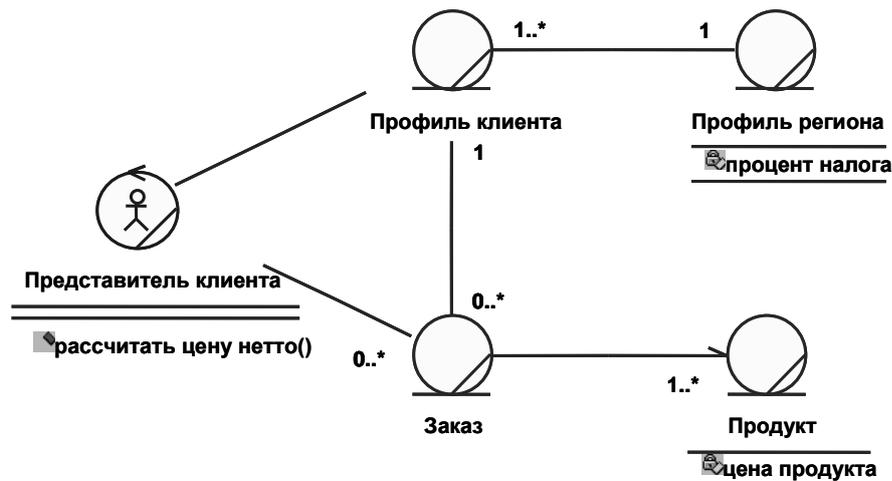
- видение бизнеса;



- оценка организации.

На основании этих документов принимается решение: либо моделировать только предметную область, либо осуществляется полное деловое моделирование. Исследование автоматизации процессов предпринимается, если создаваемое программное обеспечение должно автоматизировать бизнес, ранее ведущийся по старинке.

Бизнес правила представляют собой ограничения, которые должны обязательно выполняться в ходе деловых процессов. Формулировки бизнес-правил составляют специальный документ – Описание бизнес-правил. Каждое бизнес-правило должно так или иначе прослеживаться на диаграммах бизнес-модели. Например, бизнес-правило: *Цена нетто = цена продукта \* (1 + процент налога / 100)* задает условие на структурные связи в модели бизнес-анализа, а именно: исполнитель, ответственный за расчет цены нетто должен иметь возможность получить все



подставляемые в формулу значения. Соответствующая диаграмма классов из модели бизнес-анализа должна быть проверена, и при необходимости на нее должны быть добавлены дополнительные связи. На представленной диаграмме видны «маршруты» получения

- цены продукта (Заказ ->Продукт.цена продукта);
- процента налога (Профиль клиента -> Профиль региона. процент налога).

В связи с большим количеством типов возможных бизнес-правил вводят их классификацию:

- правила-ограничения:
  - управляющие воздействия и реакции на воздействия (например, бизнес-правило: «При отмене заказа, если он еще не доставлен, то его следует отметить как закрытый», здесь отмена заказа – управляющее воздействие, а закрытие отмененного заказа – реакция);
  - операционные ограничения или предусловия и постусловия (например, бизнес-правило: «Доставить заказ клиенту только при наличии адреса доставки» устанавливает предусловие для операции доставки заказа);
  - структурные ограничения (например, бизнес-правило «Заказ включает в себя по крайней мере одну позицию», устанавливает мощность связи между классами деловых сущностей Заказ и Позиция заказа, такое бизнес-правило дополнительно отображается на диаграмме классов из модели бизнес-анализа в виде показателей мощности на полюсах ассоциации, соединяющей упомянутые сущности);
- правила вывода и вычислительные правила (например, рассмотренная выше формула расчета цены).

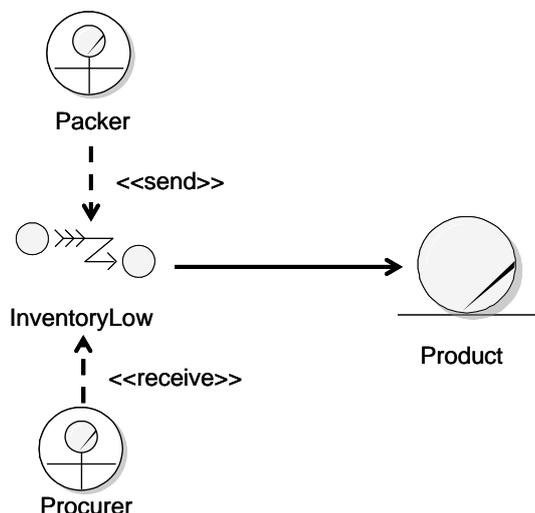
Бизнес-разработчик должен учитывать все бизнес-правила и отслеживать их выполнение в модели бизнес-анализа.

Модель бизнес-анализа может быть достаточно большой, что вызывает необходимость ее структурировать. Это осуществляется при помощи таких элементов как реализация бизнес-процесса и бизнес-система.

Реализация бизнес-процесса – кооперация со стереотипом «business use case realization»). описывает структуру бизнес-классов (исполнителей и деловых сущностей) и взаимодействие их экземпляров (бизнес-объектов) при реализации конкретного бизнес-процесса. Другими словами, диаграммы классов, диаграммы взаимодействия, относящиеся к одному бизнес-процессу объединяются в одну реализацию бизнес-процесса.

Бизнес-система – пакет со стереотипом «business system» – объединяет относящиеся к одному подразделению организации исполнителей и экономические ресурсы (деловые сущности), относящиеся к ведению подразделения, а также связанные с ними диаграммы состояний. Если какая-либо реализация бизнес-процесса осуществляется целиком в рамках подразделения, в соответствующую бизнес-систему помещается реализация этого бизнес-процесса (кооперация). Большая бизнес-система может быть разделена на части – бизнес-системы подчиненных отделов подразделения.

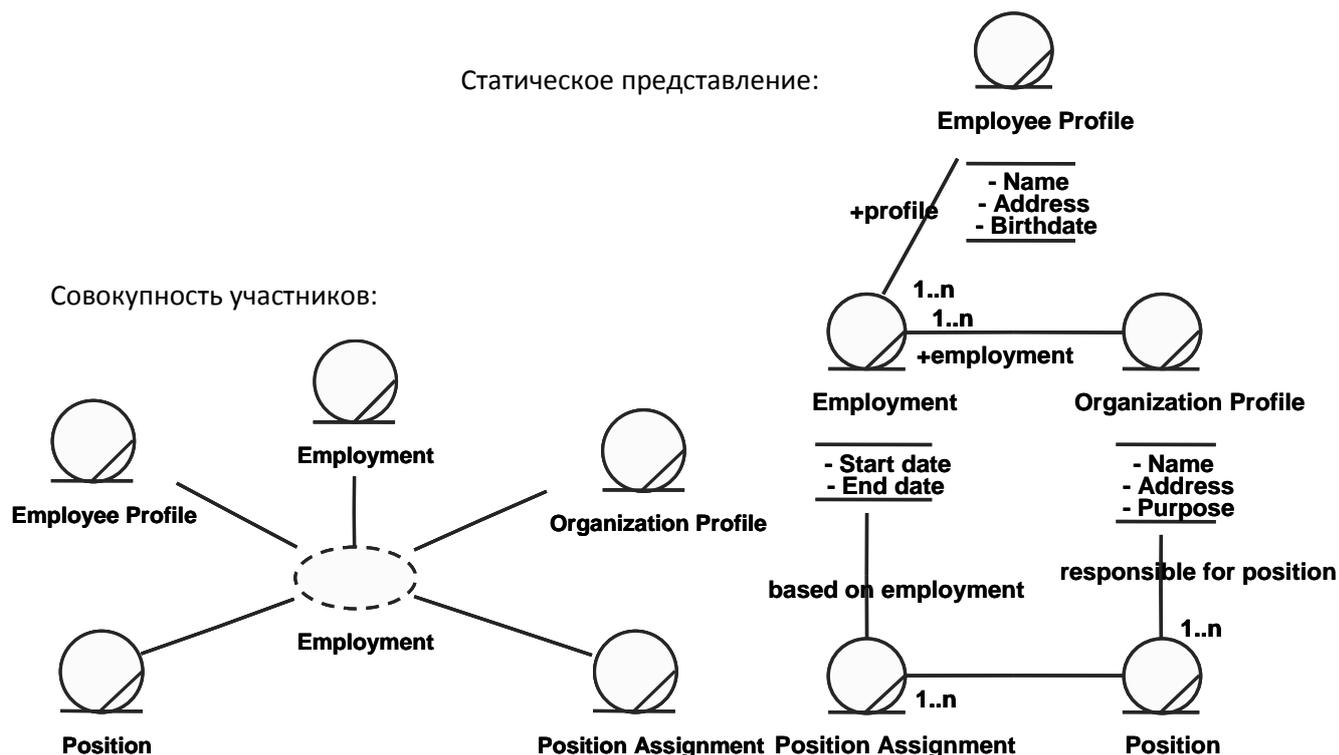
Бизнес-событие (Business Event) описывает значимое явление в пространстве и времени, важное для бизнеса. Бизнес-события обычно связаны ассоциациями с бизнес-сущностями (например, событие «малый остаток на складе» может быть связано с деловой сущностью Товар в бизнес-системе склада). При моделировании бизнес-событий определяются условия для возникновения событий, существенная информация о событии, перечень исполнителей, которые могут обнаружить возникновение события и которые должны быть уведомлены о возникновении события, ожидаемая реакция этих участников.



Типовые решения в области бизнес-моделирования оформляются в виде бизнес-образцов (паттернов). Описание образца содержит имя, описание решаемой проблемы и ее контекста, решение (модель и ее описание), результаты (следствия применения образца). Образцы бизнес-моделирования представляются в виде коопераций со стереотипом «business pattern» и включают:

- диаграммы классов, описывающие совокупность классов-участников для решения проблемы (эти классы играют роль гнезд или ячеек, куда при применении образца подставляются конкретные классы из бизнес-модели); – статическое представление;
- динамическое представление – диаграммы деятельности или взаимодействия.

Пример. Бизнес-образец «Занятость».



*Проблема:* описание различных форм занятости сотрудников внутри организации.

*Решение:* занятость моделируется как контракт между личностью и организацией, указывающий выполняемые обязанности, контрактные условия, даты начала и конца работы. Личность характеризуется набором атрибутов (имя, адрес, дата рождения), может занимать более чем одну должность в организации.

Это структурный бизнес-образец, в нем отсутствует динамическое представление. Классы-участники: Employee Profile – данные о служащем; Organization Profile – данные об организации; Employment – наем служащего (период занятости); Position – должность; Position Assignment – занимаемая служащим должность.

## Литература к лекции 5

1. Вендров А. М. Проектирование программного обеспечения экономических информационных систем. 2-е изд. – М.: Финансы и статистика, 2005. – Глава 3.
2. Джонстон С. UML-профиль для моделирования бизнес-систем. 2004. – <http://www.ibm.com/developerworks/ru/library/5167/> (обратите внимание, что некоторые пиктограммы перепутаны).

## Лекция 6. ОПРЕДЕЛЕНИЕ ТРЕБОВАНИЙ К ПРОГРАММНОМУ ОБЕСПЕЧЕНИЮ

Требование – это условие, которому должно удовлетворять программное обеспечение, или свойство, которым оно должно обладать, чтобы:

- удовлетворить потребность пользователя в решении некоторой задачи;
- удовлетворить требования контракта, спецификации или стандарта.

Спецификация требований к ПО является основным документом, определяющим план разработки ПО. Все требования, указанные в спецификации, делятся на функциональные и

нефункциональные. Функциональные требования определяют действия, которые должна выполнять система, без учета ограничений, связанных с ее реализацией. Тем самым функциональные требования определяют поведение системы в процессе обработки информации. Нефункциональные требования не определяют поведение системы, но описывают атрибуты системы или атрибуты системного окружения. Можно выделить следующие типы нефункциональных требований:

- требования к применению, которые определяют качество пользовательского интерфейса, документации и учебных курсов;
- требования к производительности, которые накладывают ограничения на функциональные требования, задавая необходимую эффективность использования ресурсов, пропускную способность и время реакции;
- требования к реализации, которые предписывают использовать определенные стандарты, языки программирования, операционную среду и др.;
- требования к надежности, которые определяют допустимую частоту и воздействие сбоев, а также возможности восстановления;
- требования к интерфейсу, которые определяют внешние сущности, с которыми может взаимодействовать система, и регламент этого взаимодействия.

Методы выявления требований:

- собеседование (интервьюирование);
- анкетирование;
- проведение совещаний;
- сессии по выявлению требований (мозговой штурм);
- раскадровка (storyboard);
- создание и демонстрация работающих прототипов;
- ролевые игры.

В ходе проекта работа с требованиями делится на следующие этапы:

- Определение типов требований и групп участников проекта, работающих с ними.
- Первичный сбор требований, их классификация и занесение в базу данных требований.
- Использование базы данных требований для управления проектом.

Любое требование в базе проекта имеет следующие атрибуты:

- Приоритет (высокий, средний, низкий).
- Статус (предложено, одобрено, реализовано, верифицировано).
- Стоимость реализации (высокая, средняя, низкая – или числовое значение).
- Сложность реализации (высокая, средняя, низкая).
- Стабильность (высокая, средняя, низкая).
- Ответственный исполнитель.

Хороший набор требований удовлетворяет следующим показателям качества (IEEE 830-1998 «Recommended Practice for Software Requirements Specifications»):

- Корректность или адекватность (соответствие реальным потребностям).
- Недвусмысленность (однозначность понимания).
- Полнота (отражение всех выделенных потребностей и всех возможных ситуаций, в которых придется работать системе).
- Непротиворечивость (согласованность между различными элементами).
- Упорядоченность по приоритету и стабильности.
- Проверимость (выполнение каждого требования нужно должно проверяться достаточно эффективным способом – непроверяемые требования должны быть удалены из рассмотрения или сведены к проверяемым вариантам).

- Модифицируемость (оформление в удобных для внесения изменений структуре и стилях).
- Прослеживаемость в ходе разработки (возможность увязать требование с подсистемами, модулями и операциями, ответственными за его выполнение, и с тестами, проверяющими его выполнение).

Выявленные требования к ПО оформляются в виде ряда документов и моделей. К основным документам, регламентируемым технологией Rational Unified Process, относятся:

- Концепция – определяет глобальные цели проекта и основные особенности разрабатываемой системы. Существенной частью концепции является постановка задачи разработки, определяющая требования к выполняемым системой функциям.
- Словарь предметной области (глоссарий) – определяет общую терминологию для всех моделей и описаний требований к системе. Глоссарий предназначен для описания терминологии предметной области и может быть использован как словарь данных системы.
- Дополнительные спецификации (технические требования) – содержит описание нефункциональных требований к системе, таких, как надежность, удобство использования, производительность, сопровождаемость и др.

Функциональные требования к системе моделируются и документируются с помощью вариантов использования (use case). *Вариант использования* (use case) – связный элемент функциональности, предоставляемый системой при взаимодействии с действующими лицами. *Действующее лицо* (actor) – роль, обобщение элементов внешнего окружения системы, ведущих себя по отношению к системе одинаковым образом.

Каждый вариант использования фиксирует соглашение между участниками проекта относительно поведения системы. Он описывает поведение системы при различных условиях, как реакцию системы на запрос одного из участников, называемого основным действующим лицом. Основное действующее лицо инициирует взаимодействие с системой, чтобы добиться некоторой цели. Система отвечает, соблюдая интересы всех участников.

Варианты использования – это вид документации, применяемый, когда требуется сконцентрировать усилия на обсуждении принципиальных функциональных требований к разрабатываемой системе, а не на подробном их описании. Стиль их написания зависит от масштаба, количества участников и критичности проекта. Формат описания варианта использования (по Коберну):

- Имя – цель в виде краткой активной глагольной фразы
- Контекст использования – более длинное описание цели
- Область действия
- Основное действующее лицо
- Участники и интересы
- Предусловие (определяет, выполнение какого условия гарантирует система перед тем, как разрешить запуск варианта использования)
- Минимальные гарантии (наименьшие обещания системы участникам, в частности, когда цель основного действующего лица не может быть достигнута)
- Гарантии успеха (или постусловие – postcondition – устанавливает, что интересы участников удовлетворяются по успешном завершении варианта использования в конце основного сценария)
- Триггер (событие, которое запускает вариант использования)
- Основной сценарий (простой для понимания типичный сценарий, в котором достигается цель основного действующего лица и удовлетворяются интересы всех участников)
- Расширения (запускаются при возникновении определенного условия, содержат последовательность шагов, описывающих, что происходит при этом условии, и заканчивается

достижением цели или отказом от неё)

- Список изменений в технологии и данных
- Вспомогательная информация

Существуют четыре уровня точности описания вариантов использования, расположенные по степени повышения точности:

- Действующие лица и цели (перечисляются действующие лица и все их цели, которые будет обеспечивать система). На этом уровне определяется границы системы, контекст в котором она работает. Действующее лицо может: быть активным, посылая запросы в систему; быть пассивным, получая данные от системы. Его роль может исполнять человек – пользователь, устройство, внешняя программная система, время (в случае если функциональность запускается по расписанию), температура или другое свойство состояния окружающей среды, играющее роль триггера.
- Краткое изложение варианта использования (в один абзац) или основной поток событий (без анализа возможных ошибок).
- Условия отказа (анализ мест возникновения возможных ошибок в основном потоке событий).
- Обработка отказа (написание всех альтернативных потоков событий).

Введение перечисленных уровней преследует своей целью грамотное планирование и экономию времени разработки. В итерационном цикле создания системы не следует пытаться за один прием подробно описать все требования, их нужно постепенно уточнять, повышая уровень точности. Выбор первоочередных вариантов использования для уточнения определяется их приоритетами. Факторами ранжирования вариантов использования (и вообще всех требований) по приоритетам являются:

- существенное влияние на архитектуру системы;
- рискованные, сложные для реализации или срочные функции;
- применение новой, неапробированной технологии;
- значимость в экономических процессах.

Правила написания сценариев:

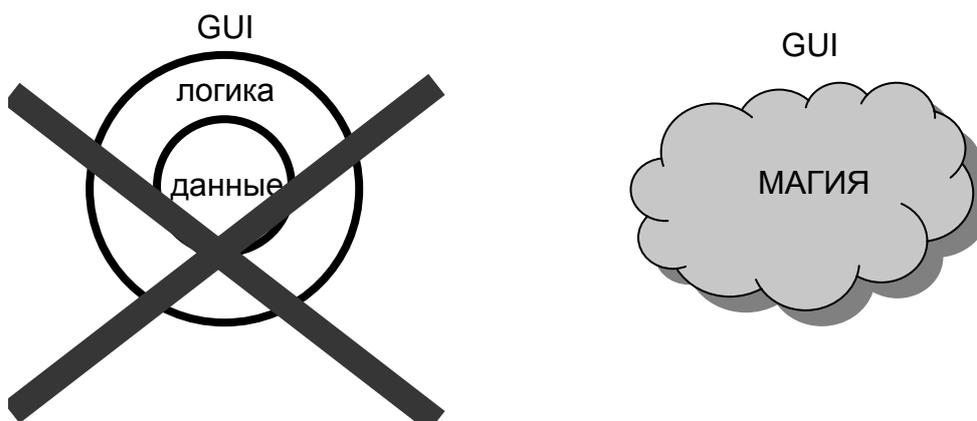
- 1) Используйте простые предложения: Подлежащее...сказуемое...прямое дополнение...предложный оборот (Система...удерживает...сумму...из остатка на счёте).
- 2) Ясно укажите, кто «владеет мячом». На каждом шаге одно из действующих лиц "владеет мячом" – сообщением и данными, которые одно действующее лицо передаёт другому.
- 3) Пишите, глядя на вариант использования с точки зрения пользователя, а не системы.
- 4) Не показывайте слишком незначительные, мелкие действия.

Виды альтернативных сценариев:

- Некорректное действие действующего лица (ввод неверного пароля).
- Бездействие основного действующего лица (истечение времени ожидания пароля).
- Предложение "система подтверждает" связано с обработкой неподтверждения (неверный учётный номер).
- Несоответствующая реакция второстепенного действующего лица или её отсутствие (истечение времени ожидания ответа).
- Внутренняя ошибка в разрабатываемой системе, которая должна быть обнаружена и обработана в обычном порядке (заблокирован автомат для выдачи наличных).
- Неожиданная и необычная ошибка, которую необходимо обработать (обнаружено повреждение журнала транзакций).
- Критически важные недостатки в производительности системы (время реакции не

укладывается в 5 секунд).

Писать варианты использования следует с позиции пользователя, т. е.:



Типичные ошибки в сценариях:

- Отсутствует система.
- Отсутствует основное действующее лицо.
- Слишком много деталей пользовательского интерфейса.
- Слишком низкий (подробный) уровень описания.

Связи между вариантами использования и действующими лицами отображаются на диаграмме



вариантов использования:

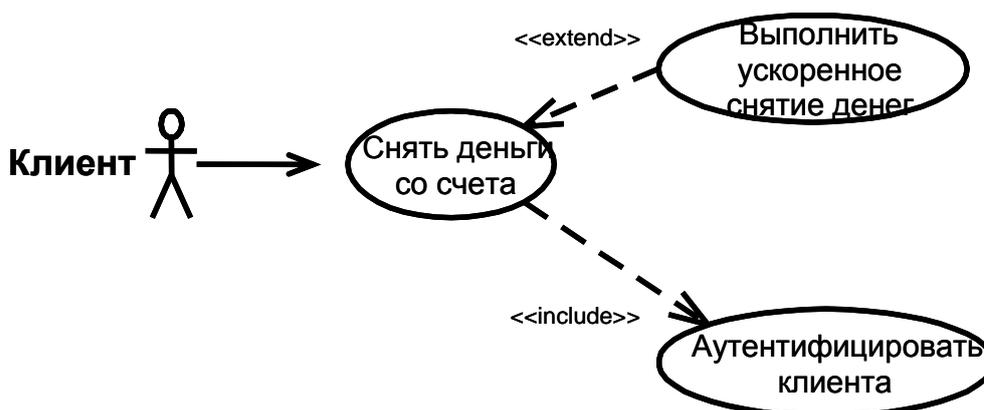
Правила составления этих диаграмм:

- 1) Каждый вариант использования должен быть инициирован действующим лицом.
- 2) Не моделируйте связи коммуникации между действующими лицами.
- 3) Не соединяйте стрелкой (связью коммуникации) два варианта использования непосредственно. Диаграммы данного типа описывают только, какие варианты использования доступны системе, а не порядок их выполнения. На диаграммах варианты использования могут быть связаны либо зависимостями (связями включения или расширения) или

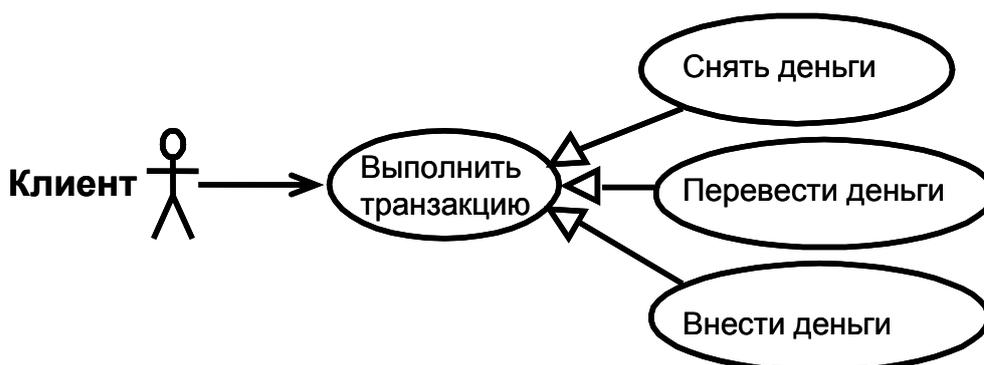
обобщением (наследованием). Для отображения порядка выполнения вариантов использования применяют диаграммы деятельности.

- 4) Избегайте многочисленных и запутанных связей между действующими лицами и вариантами использования.

Для снижения сложности начальная модель вариантов использования может быть подвергнута структуризации, в ходе которой выделяются вспомогательные варианты использования (включаемые или расширяющие). Сложность снижается за счет вынесения части описаний из основного варианта использования во включаемый (который может быть включен в несколько ВИ, что еще больше упрощает модель) или расширяющий.



Инструментом структуризации также является обобщение вариантов использования.



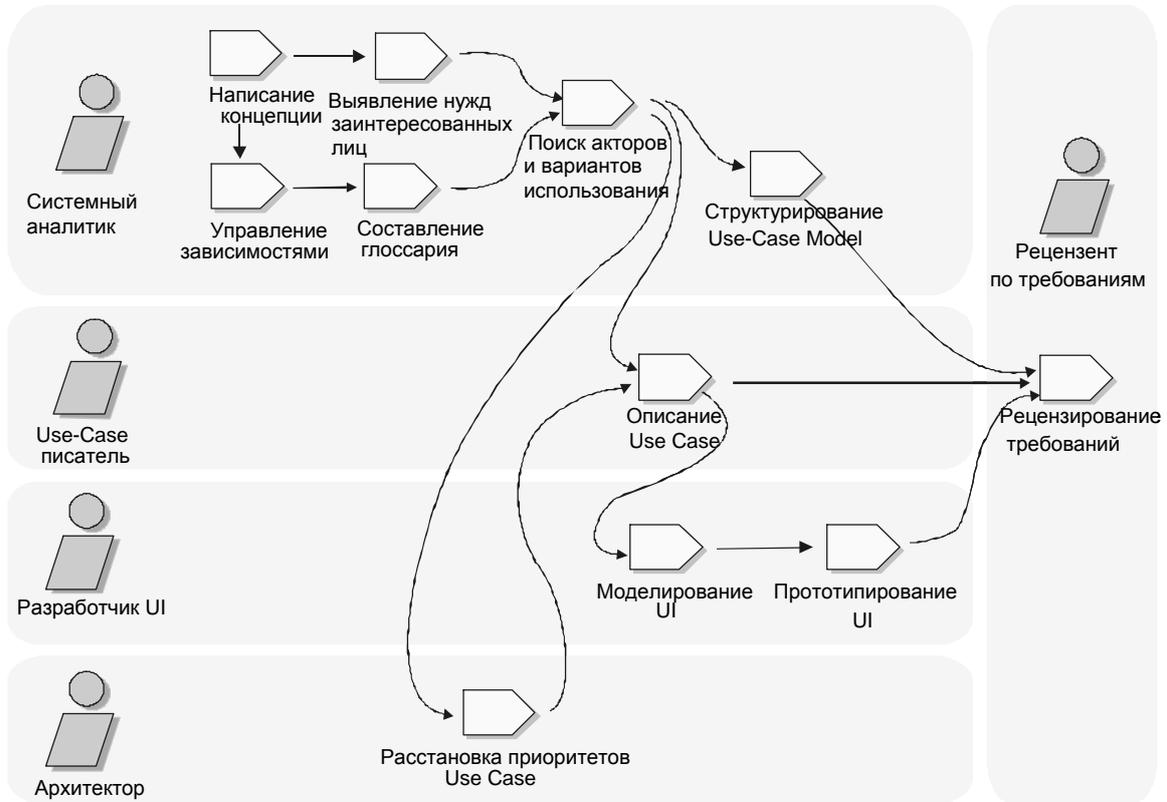
При обобщении в базовый ВИ выносится описание общее для всех наследников, дочерние ВИ специализируют описание базового, они участвуют во всех связях расширения и/или включения базового. Обобщение может быть использовано и для действующих лиц. В таком случае дочерние действующие лица наследуют связи коммуникации родительского действующего лица.

Методика моделирования вариантов использования в технологии Rational Unified Process предусматривает специальное соглашение, связанное с группировкой структурных элементов и диаграмм модели. Это соглашение включает следующие правила:

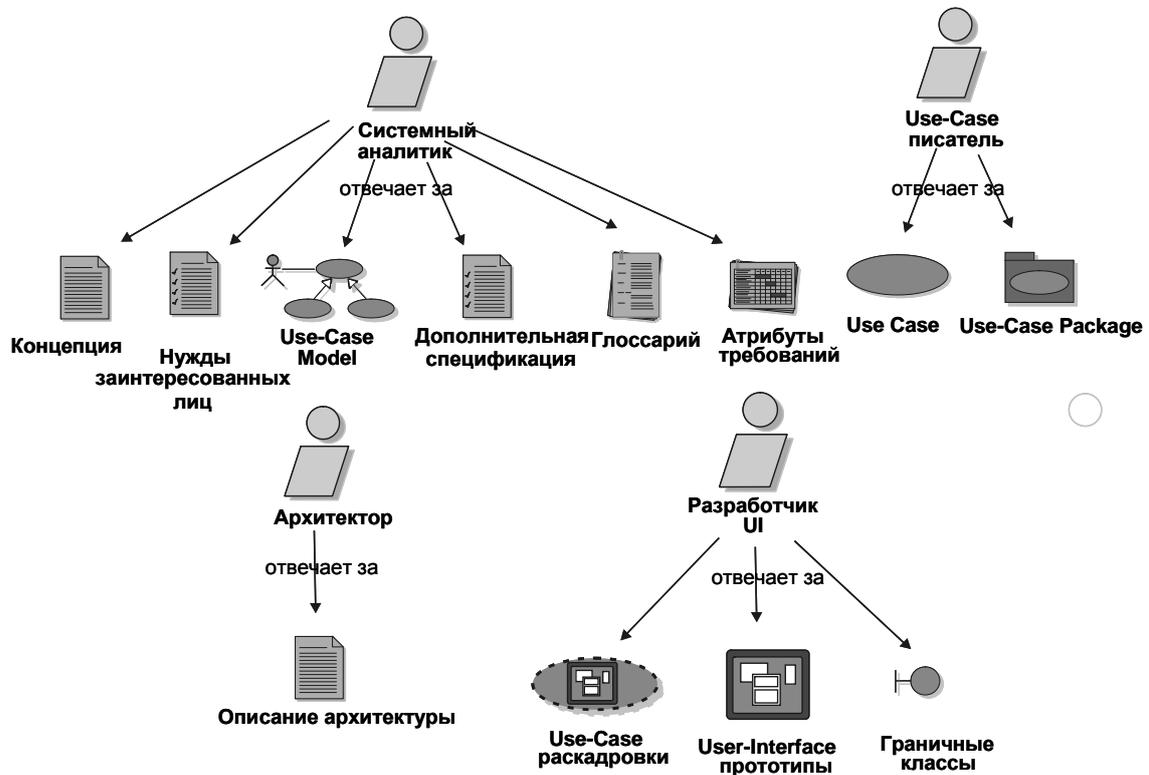
- Все действующие лица, варианты использования и диаграммы вариантов использования помещаются в пакет с именем Use Case Model.
- Если моделируется сложная многофункциональная система, то совокупность всех действующих лиц и вариантов использования может разделяться на пакеты. В качестве принципов разделения могут использоваться:
  - структуризации модели в соответствии с типами пользователей (действующих лиц);
  - функциональная декомпозиция;

- разделение модели на пакеты между группами разработчиков (в качестве объектов управления конфигурацией).

Дисциплина определения требований в рамках RUP описывается следующим набором ролей и деятельностью:

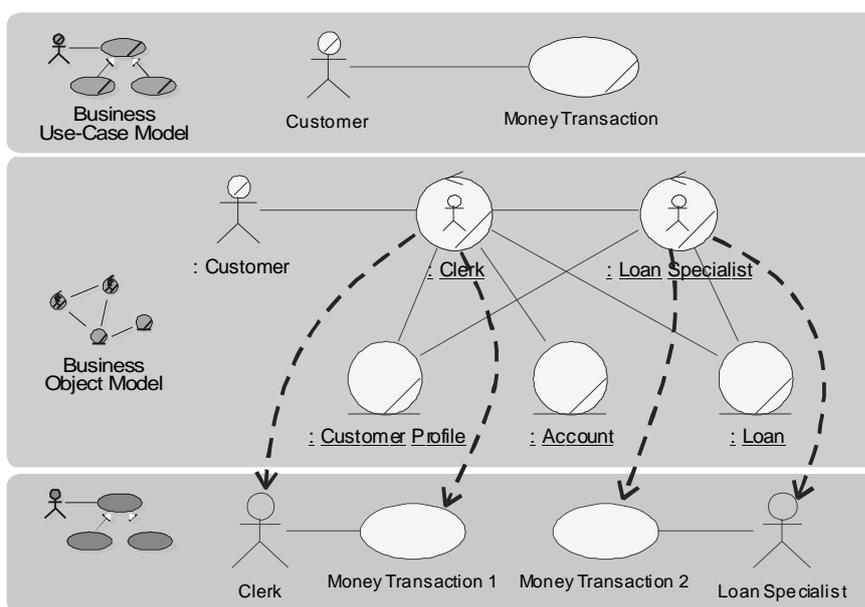


Наборы рабочих продуктов определения требований:



Спецификация требований в технологии Rational Unified Process не требует обязательного моделирования бизнес-процессов организации, для которых создается ПО, однако, наличие бизнес-моделей существенно упрощает построение системной модели вариантов использования. При переходе от бизнес-модели к начальной версии модели вариантов использования применяются следующие правила:

- Для каждого исполнителя в модели бизнес-анализа, который в перспективе станет пользователем новой системы, в модели вариантов использования создается действующее лицо с таким же наименованием. В состав действующих лиц включаются также внешние системы, играющие в бизнес-процессах пассивную роль источников информации.
- Варианты использования для данного действующего лица создаются на основе анализа обязанностей соответствующего исполнителя (в простейшем случае для каждой операции



исполнителя создается вариант использования, реализующий данную операцию в системе).

Такая начальная версия модели описывает минимальный вариант системы, пользователями которой являются только исполнители бизнес-процессов. Если в дальнейшем, в процессе развития системы, ее непосредственными пользователями будут становиться действующие лица бизнес-процессов, то модель вариантов использования будет соответствующим образом модифицироваться.

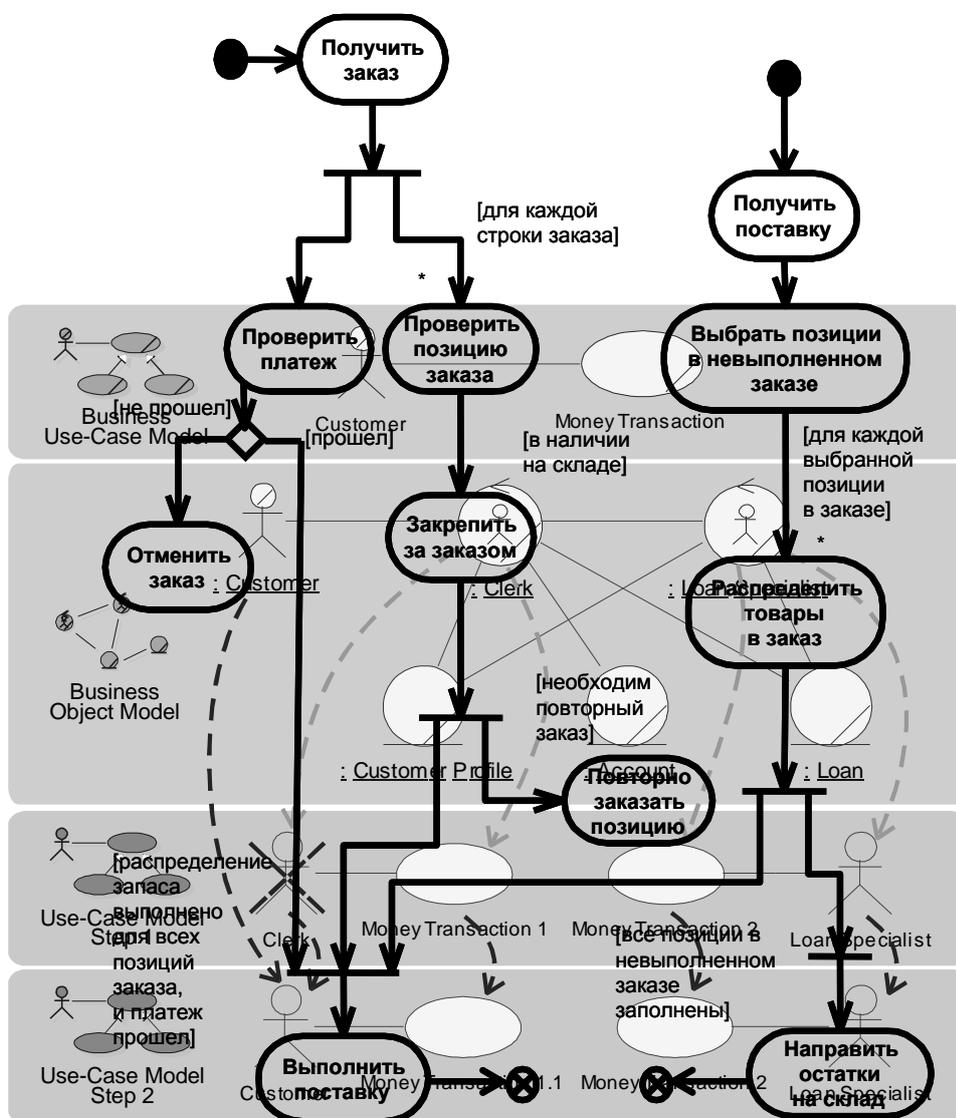
Для описания функциональных требований используются диаграммы деятельности:

- для описания поведения, включающего большое количество параллельных процессов
- для анализа варианта использования (описывают последовательность действий и их взаимосвязь)
- для анализа потоков работ (workflow) в различных вариантах использования. Когда варианты использования взаимодействуют друг с другом, диаграммы деятельности являются средством представления и анализа их поведения.

Пример диаграммы деятельности для двух синхронных потоков:

Модель вариантов использования можно считать завершенной, если есть утвердительный ответ на следующие вопросы:

- Можно ли на основании модели сформировать четкое представление о функциях системы и их взаимосвязях?
- Присутствует ли каждое функциональное требование хотя бы в одном варианте использования? Если требование не нашло отражение в варианте использования, оно не будет реализовано.



- Учили ли вы, как с системой будет работать каждое заинтересованное лицо?
- Какую информацию каждое заинтересованное лицо будет передавать системе?
- Учили ли вы все внешние системы, с которыми будет взаимодействовать данная?

### **Литература к лекции 6**

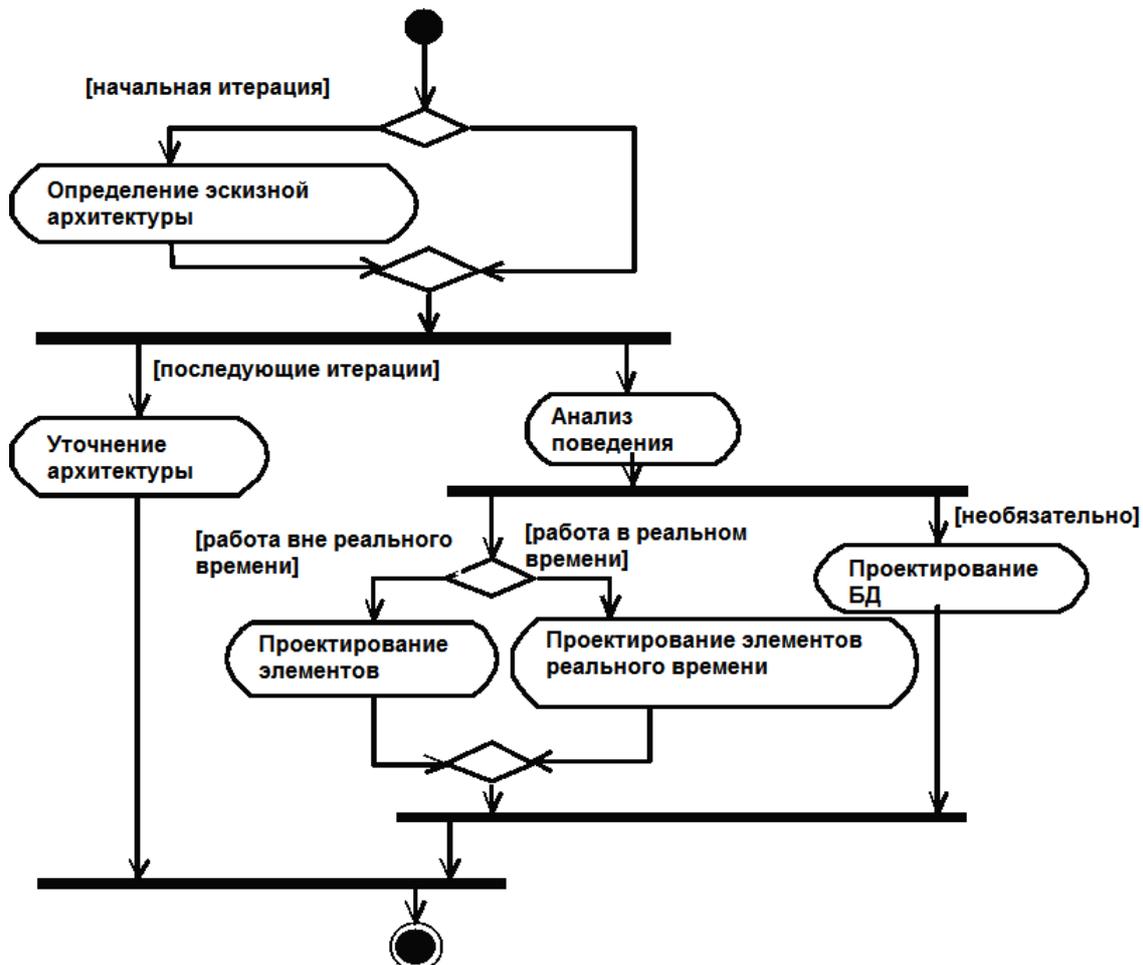
1. Коберн А. Современные методы описания функциональных требований к системам.: Пер. с англ. – М. Лори, 2002.
2. Кулямин В. В. Технологии программирования. Компонентный подход. – М.: Бином. Лаборатория знаний. 2007. Лекция 4.  
<http://panda.ispras.ru/~kuliamin/lectures-sdt/sdt-book-2006.pdf>
3. Вендров А. М. Проектирование программного обеспечения экономических информационных систем. 2-е изд. – М.: Финансы и статистика, 2005. Параграфы 2.4-2.5.
4. Соммервил И. Инженерия программного обеспечения. 6-е изд.: Пер. с англ. – М.: Вильямс, 2002. – Главы 5, 6.
5. Якобсон А., Буч Г., Рамбо Дж. Унифицированный процесс разработки программного обеспечения.: Пер. с англ. – СПб.: Питер, 2002. – Главы 6, 7.

### ***Лекция 7. Анализ и проектирование программного обеспечения. Анализ ПО***

Сначала охарактеризуем в целом технологический процесс анализа и проектирования (один из процессов в рамках RUP). Его цели:

- преобразование требований в системный проект;
- создание стабильной (т. е. не подлежащей существенным изменениям) архитектуры системы;
- адаптация системного проекта к среде реализации.

На входе анализа и проектирования находятся модель вариантов использования, глоссарий и дополнительная спецификация, на выходе – проектная модель системы, модель базы данных, описание архитектуры.



Процесс анализа и проектирования является итерационным, т. е. разбит на несколько итераций в ходе которых выполняется часть работ по анализу и проектированию части системы. Результаты каждой итерации интегрируются в общую модель. Поток работ можно представить диаграммой деятельности:

Эскизная архитектура включает:

- Набор ключевых абстракций.
- Набор классов анализа.
- Механизмы анализа.
- Иерархию уровней.
- Структуру системы.
- Реализации вариантов использования.

Уточнение архитектуры состоит в переходе от классов анализа к проектным классам.

Определяются: проектные классы; механизмы проектирования; представление размещения.

Анализ поведения включает:

- анализ вариантов использования;
- определение элементов проекта;
- рецензирование проекта.

Проектирование элементов включает:

- выявление пакетов и подсистем;
- проектирование классов;
- проектирование подсистем.

Проектирование БД включает:

- выделение устойчивых классов;
- разработку структуры БД;
- определение механизмов хранения (таких как ODBC или OODBMS).

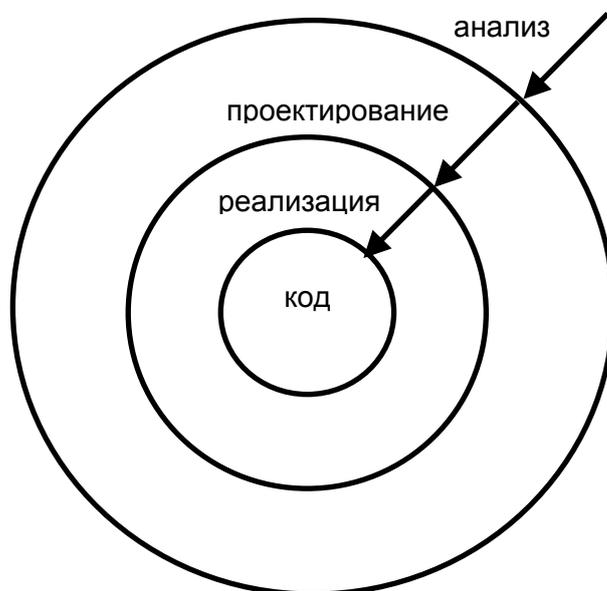
Анализ и проектирование отличаются подходом к создаваемой системе. Анализ характеризуется тем, что:

- в центре внимания – проблема;
- не придается значения деталям;
- описывается структура и поведение системы;
- реализуются функциональные требования в архитектуре системы;
- модель анализа имеет относительно небольшой размер.

Характеристики проектирования:

- в центре внимания – решение;
- придается значение деталям – операциям и атрибутам;
- учитываются аспекты производительности;
- модель приближена к реальному коду;
- реализуются нефункциональные требования;
- проектная модель значительно больше модели анализа.

В целях борьбы со сложностью система создается на различных уровнях детализации (абстракции): уровне анализа, уровне проектирования, уровне реализации и самом подробном – уровне кода.



Целью объектно-ориентированного анализа является трансформация функциональных требований к ПО в предварительный системный проект и создание стабильной основы архитектуры системы. В дальнейшем предварительный проект уточняется с учетом нефункциональных требований и выбранных средств реализации проекта – это происходит при проектировании.

Исполнителями процесса анализа являются архитектор, разработчик (или проектировщик), разработчик БД, разработчик пользовательского интерфейса, рецензент. Обязанности архитектора:

- координация и руководство процессом анализа и проектирования;
- определение структуры каждого архитектурного представления;
- осуществление архитектурного анализа.

Обязанности разработчика:

- анализ вариантов использования;
- определение обязанностей, поведения, свойств классов и связей между классами;

- анализ одного или нескольких пакетов или подсистем.  
Разработчик БД отвечает за модель данных (схему БД).

Разработчик пользовательского интерфейса (UI) создает экранные формы, навигационные карты, осуществляет прототипирование UI.

Рецензент оценивает решения, принятые в ходе процесса и созданные рабочие продукты (документы).

Объектно-ориентированный анализ включает два вида деятельности выполняемые друг за другом:

- 1) архитектурный анализ;
- 2) анализ вариантов использования.

*Архитектурный анализ* выполняется архитектором системы и включает в себя следующие работы:

- 1) утверждение общих стандартов (соглашений) моделирования и документирования системы;
- 2) предварительное выявление архитектурных механизмов (механизмов анализа);
- 3) формирование набора основных абстракций предметной области (классов анализа);
- 4) формирование начального представления архитектурных уровней.

*Соглашения моделирования* определяют:

- используемые диаграммы и элементы модели;
- правила их применения;
- соглашения по именованию элементов модели;
- организацию модели (пакеты).

Соглашения фиксируются в документе «Руководящие указания по проектированию» (Design Guidelines). Пример соглашений:

- 1) Имена вариантов использования должны быть короткими глагольными фразами.
- 2) Для каждого варианта использования должен быть создан пакет Use-Case Realization, включающий:
  - a) по крайней мере одну реализацию варианта использования;
  - b) диаграмму “View Of Participating Classes” (VOPC).
- 3) Имена классов должны быть существительными, соответствующими, по возможности, понятиям предметной области.
- 4) Имена классов должны начинаться с заглавной буквы.
- 5) Имена атрибутов и операций должны начинаться со строчной буквы.
- 6) Составные имена должны быть сплошными, без подчеркиваний, каждое отдельное слово должно начинаться с заглавной буквы.

*Архитектурные механизмы* отражают нефункциональные требования к системе (надежность, безопасность, хранение данных в конкретной среде, интерфейсы с внешними системами и т.д.) и их реализацию в архитектуре системы. Архитектурные механизмы представляют собой набор типовых решений, или образцов, принятых в качестве стандарта данного проекта. Категории архитектурных механизмов:

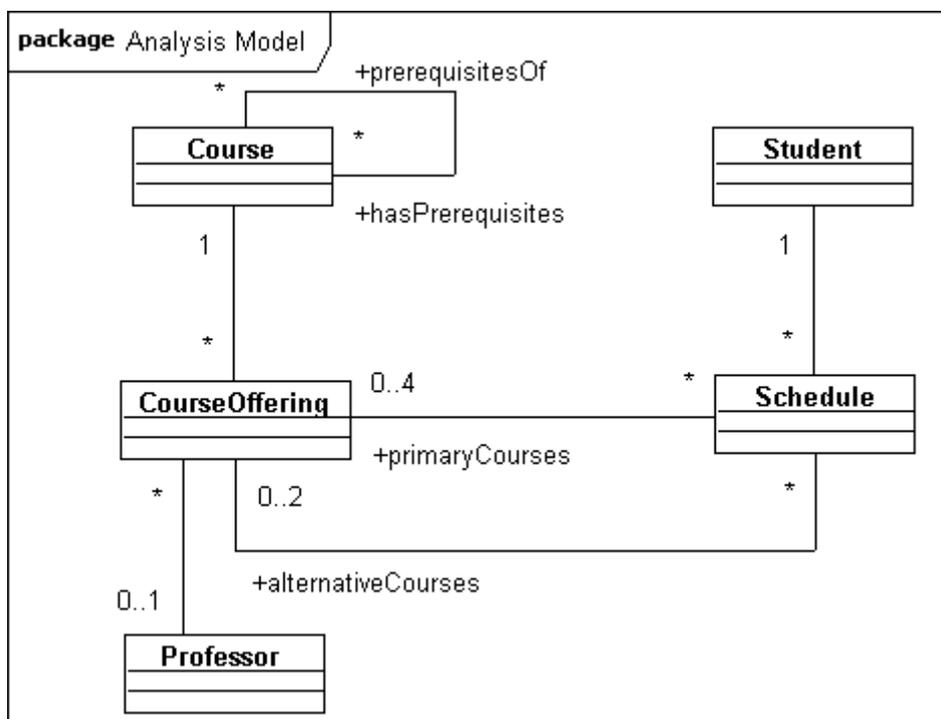
- Механизмы анализа: обеспечивают взаимодействие классов и компонентов предметной области, без деталей реализации.
- Проектные механизмы: учитывают некоторые детали среды реализации, без привязки к конкретной среде (например, выбор между РСУБД и ООСУБД).
- Механизмы реализации: зависят от конкретной технологии, языка программирования,

поставщика (Oracle, Sun или Microsoft) и т.д.

Примеры механизмов анализа:

- Устойчивость (persistency): элементы модели, которые должны сохранять свое состояние в течение длительного времени должны быть определены как устойчивые (для каждого устойчивого элемента определяются его размер и количество хранимых объектов, сроки хранения, механизмы и частотные характеристики доступа).
- Интерфейс с унаследованными системами (legacy interface) – к этому механизму относят все элементы модели, ответственные за интерфейс с унаследованной системой.
- Безопасность (уровни, правила, привилегии) – элементы, обеспечивающие контроль доступа к системе.
- Распределение – элементы, которые должны быть распределены по узлам сети.

*Идентификация основных абстракций* заключается в определении набора классов системы (классов анализа), представляющих основные понятия предметной области. Набор ключевых абстракций создается на основе описания предметной области и спецификации требований к системе (в частности, глоссария проекта). Рисуется диаграмма классов Key Abstractions, на которую помещаются все ключевые абстракции. Пример (регистрация на курсы):



Архитектурные уровни образуют иерархию уровней представления любой крупной системы.

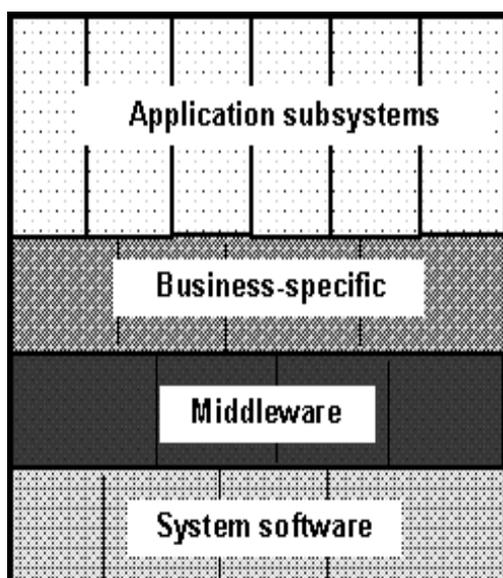
Почти в любой системе можно выделить следующие уровни:

- прикладной, содержащий набор компонентов, реализующих основную функциональность системы;
- бизнес-уровень, включающий набор компонентов, специфичных для конкретной предметной области;
- промежуточный (middleware), куда входят платформно-независимые сервисы;
- системный, содержащий компоненты вычислительной и сетевой инфраструктуры.

При формировании архитектурных уровней архитектор определяет начальная структура модели (набор пакетов и их зависимостей, распределение пакетов по уровням), рассматриваются только верхние уровни (прикладной и бизнес-логика), используются архитектурные образцы (patterns) и каркасы (frameworks). Образец представляет собой типичное решение некоторой проблемы в

заданном контексте. Каркас является архитектурным образцом, определяющим шаблон для приложений в конкретной области. Примеры архитектурных образцов:

- Уровни (Layers) – способ декомпозиции приложения на набор слоев, соответствующих различным уровням абстракции.
- Модель-представление-управление (Model-view-controller, M-V-C) – разделение приложения на три части: данные и бизнес-правила; пользовательское представление; обработку данных.
- Каналы и фильтры (Pipes and filters) – шаблон архитектуры системы для потоковой обработки данных.



Архитектурный образец «Layers»:

Прикладной уровень (Application subsystems) – реализация функциональности вариантов использования.

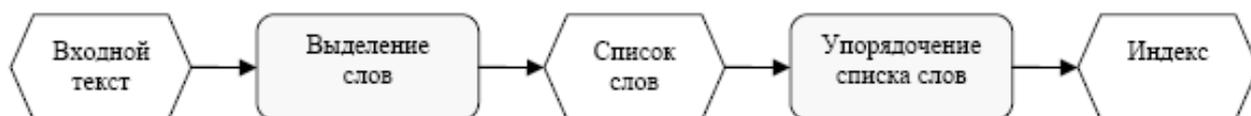
Бизнес-уровень (Business-specific) – набор компонентов, специфичных для конкретной предметной области.

Уровень промежуточного ПО (Middleware) – платформо-независимые сервисы (GUI, ORB, ...)

Уровень базового ПО (System software) – обеспечение вычислительной и сетевой инфраструктуры (ОС, сетевые протоколы и др.).

Благодаря использованию этого образца система получается модифицируемой (т. е. внесение в нее изменений сравнительно не трудоемко) и мобильной (т. е. может переноситься на другие платформы)

Образец «Каналы и фильтры» разбивает большую сложную задачу по обработке данных на упорядоченную совокупность небольших независимых этапов обработки (каждый такой этап – «фильтр»), соединенных «каналами» – потоками данных. При этом для некоторых задач удастся

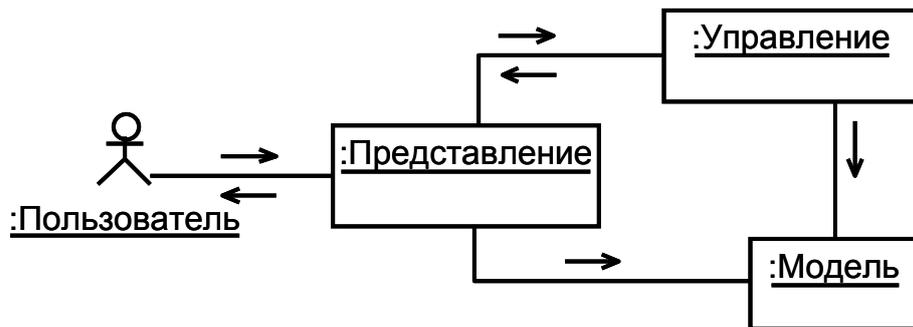
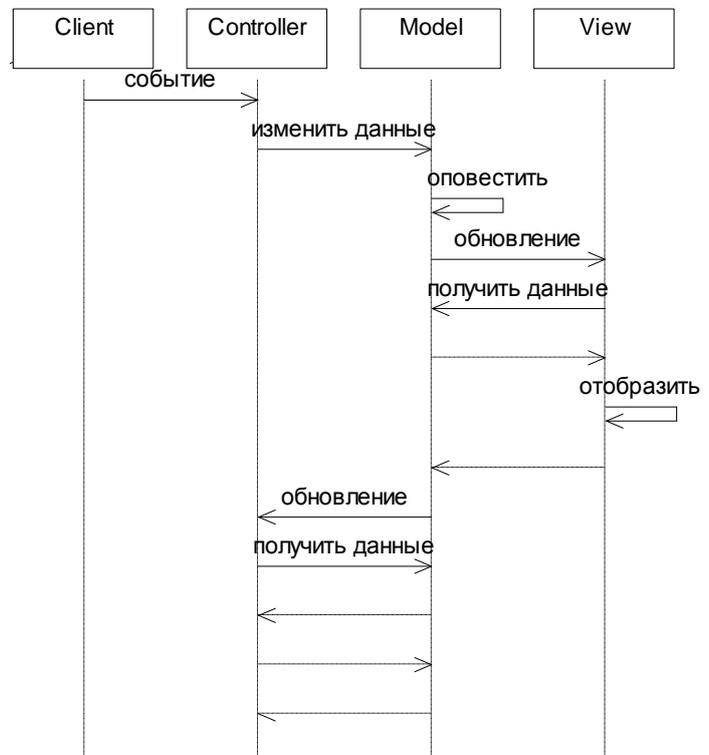


добиться эффективного решения за счет конвейера.

Образец «Model-view-controller» родом из языка Smalltalk. Проблема: *Изменения во внешнем представлении достаточно вероятны, одна и та же информация представляется по-разному в нескольких местах, система должна быстро реагировать на изменения данных.* Решение: выделить набор компонентов 3-х типов: компонентов хранения данных, компонентов представления для пользователей, и компонентов обработки (воспринимающих команды, преобразующих данные и обновляющих представления).

Рис. Один из вариантов организации взаимодействия, предписываемый образцом MVC.

Надо заметить, что при указанном взаимодействии объекты модели отвечают не только за хранение данных как таковое, но и за оповещение всех подписчиков об изменениях данных. Такая дополнительная нагрузка неоднозначно оценивается разными экспертами, например, Мартин Фаулер считает это решение неудачным и предлагает другое, в котором оповещение возлагается на объекты



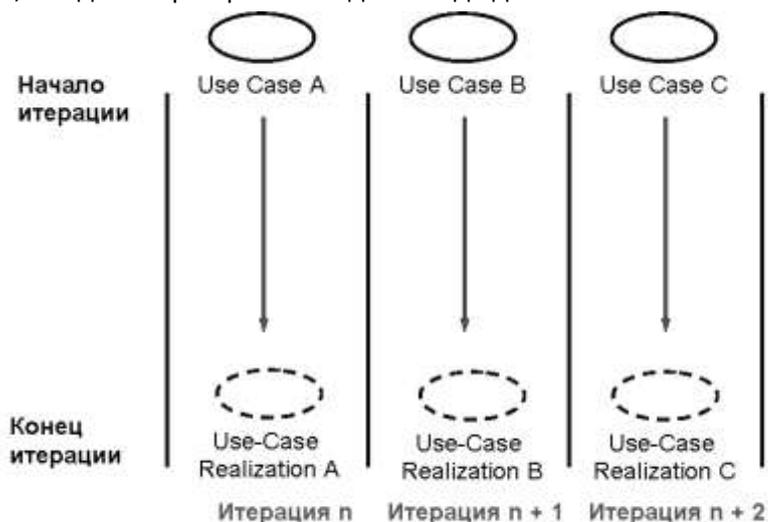
управления:

В такой схеме достигается независимость модели от представления, обеспечивающая больше возможностей для повторного использования.

*Анализ вариантов использования* выполняется разработчиками и включает в себя:

- идентификацию классов, участвующих в реализации потоков событий, (так называемых, классов анализа);
- определение обязанностей классов, их атрибутов и ассоциаций;
- унификацию классов анализа;
- квалификацию механизмов анализа.

Анализ вариантов использования является итерационным процессом – делится на несколько итераций, в ходе которых работа ведется над одним или несколькими (но не всеми



сразу) вариантами использования. Как правило, распределение вариантов использования по итерациям осуществляется на основе их приоритета (высокоприоритетные раньше, низкоприоритетные позже).

Шаги анализа вариантов использования:

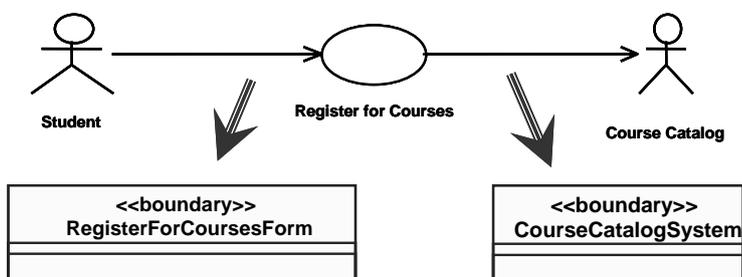
1. Уточнение и дополнение описаний вариантов использования.
2. Для каждой реализации варианта использования:
  - a. Выявление классов, участвующих в реализации потоков событий варианта использования.
  - b. Распределение поведения, реализуемого вариантом использования, между классами (обязанности классов).
3. Для каждого выявленного класса анализа:
  - a. Определение обязанностей класса.
  - b. Определение атрибутов и ассоциаций.
  - c. Квалификация механизмов анализа.
4. Унификация классов анализа.

Классы анализа отражают функциональные требования к системе и моделируют объекты предметной области. Совокупность классов анализа представляет собой начальную концептуальную модель системы. Эта модель проста и позволяет сосредоточиться на реализации функциональных требований, не отвлекаясь на детали реализации, обеспечение эффективности и надежности. Для решения этих вопросов готовая модель анализа трансформируется в проектную модель в ходе проектирования. В потоках событий варианта использования выявляются классы трех типов:

- граничные классы, являющиеся посредниками при взаимодействии с внешними объектами;
- классы-сущности, представляющие собой основные абстракции (понятия) разрабатываемой системы;
- управляющие классы, обеспечивающие координацию поведения объектов в системе.

Правило выделения граничных

классов: для каждой связи между действующим лицом и вариантом использования создается граничный

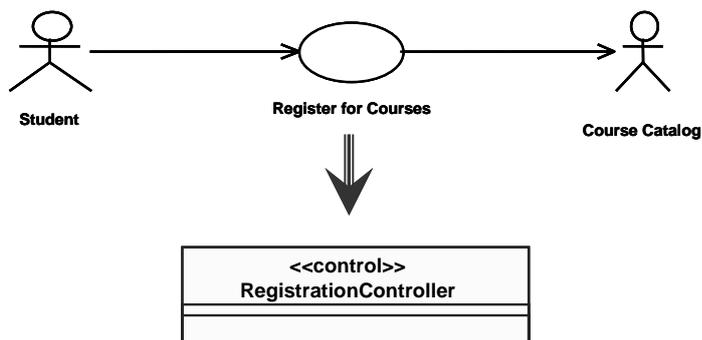


класс, отвечающий за данное взаимодействие. Типы граничных классов:

- пользовательский интерфейс (обмен информацией с пользователем, без деталей UI – кнопок, списков, окон);
- системный интерфейс и аппаратный интерфейс (используемые протоколы, без деталей их реализации).

Способы выделения классов-сущностей:

- Перевод бизнес-сущностей из бизнес-модели в классы-сущности.
- Анализ глоссария (некоторые термины являются именами искомым классов-сущностей).
- Анализ описаний вариантов использования.



Правило выделения управляющих классов: для каждого варианта использования создается ответственный за его реализацию класс управления.

Все созданные при анализе данного варианта использования классы анализа помещаются на диаграмму VOPC (View Of Participating Classes).

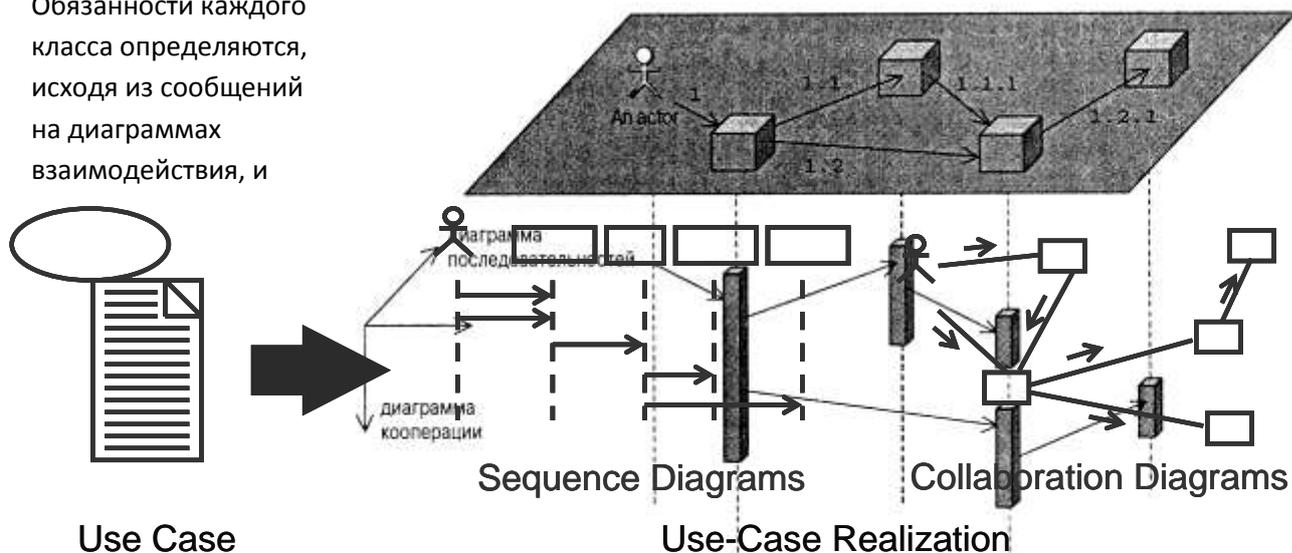
Распределение поведения, реализуемого вариантом использования, между классами реализуется с помощью диаграмм взаимодействия (диаграмм последовательности и кооперативных диаграмм). Диаграммы последовательности и кооперативные диаграммы являются как бы двумя ортогональными проекциями, описывающими взаимодействие (проекция на вертикальную плоскость – диаграмма последовательности, на горизонтальную – диаграмма кооперации):

Виды обязанностей классов:

- Знание:
  - наличие информации о данных или вычисляемых величинах;
  - наличие информации о связанных объектах.
- Действие:
  - выполнение некоторых действий самим объектом (прием и обработка входящих сообщений);
  - инициация действий других объектов (отправка исходящих сообщений);
  - координация действий других объектов (посредничество).

В процессе анализа конкретного варианта использования в первую очередь строится диаграмма последовательности, описывающая основной поток событий и его подчиненные потоки. Для каждого альтернативного потока событий строится отдельная диаграмма последовательности.

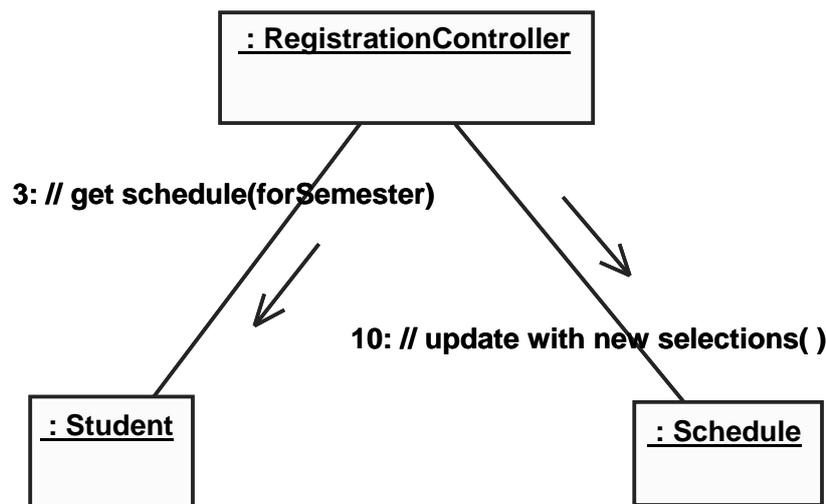
Обязанности каждого класса определяются, исходя из сообщений на диаграммах взаимодействия, и



документируются в классах в виде «операций анализа». В процессе проектирования каждая «операция анализа» будет преобразована в одну или более операций класса, которые в дальнейшем будут реализованы в коде системы.

При построении диаграмм взаимодействия возникают проблемы правильного распределения обязанностей между классами. Для их решения существует ряд образцов: Information Expert, Creator, Low Coupling, High Cohesion.

*Образец "Information Expert"*. Проблема: Нужно определить наиболее общий принцип распределения обязанностей между классами. Решение: Следует назначить обязанность информационному эксперту – классу, у которого имеется информация, требуемая для



выполнения обязанности. Пример:

При выполнении подчиненного потока событий "Обновить график" варианта использования "Зарегистрироваться на курсы" студент-пользователь должен получить доступ к своему графику прежде, чем изменить его. Согласно образцу "Information Expert", нужно определить, объект какого класса содержит информацию, необходимую для доступа к графику. На эту роль информационного эксперта, очевидно, претендует объект класса-сущности Student, поскольку график принадлежит именно ему. Поэтому сообщение 3 "get schedule(forSemester)" должно быть направлено от контроллера объекту класса Student. После того, как студент получит график и внесет в него необходимые изменения, они должны быть зафиксированы в объекте Schedule. В данном случае уже сам объекте Schedule будет играть роль информационного эксперта, поскольку он непосредственно доступен контроллеру, и сообщение 10 "update with new selections" будет направлено именно ему.

Следствия:

При распределении обязанностей образец Information Expert используется гораздо чаще любого другого образца. Большинство сообщений на диаграммах взаимодействия соответствуют данному образцу. Образец Information Expert не содержит неясных или запутанных идей и отражает обычный интуитивно понятный подход. Он заключается в том, что объекты осуществляют действия, связанные с имеющейся у них информацией. Если информация распределена между

различными объектами, то при выполнении общей задачи они должны взаимодействовать с помощью сообщений.

В некоторых ситуациях применение образца Information Expert нежелательно.

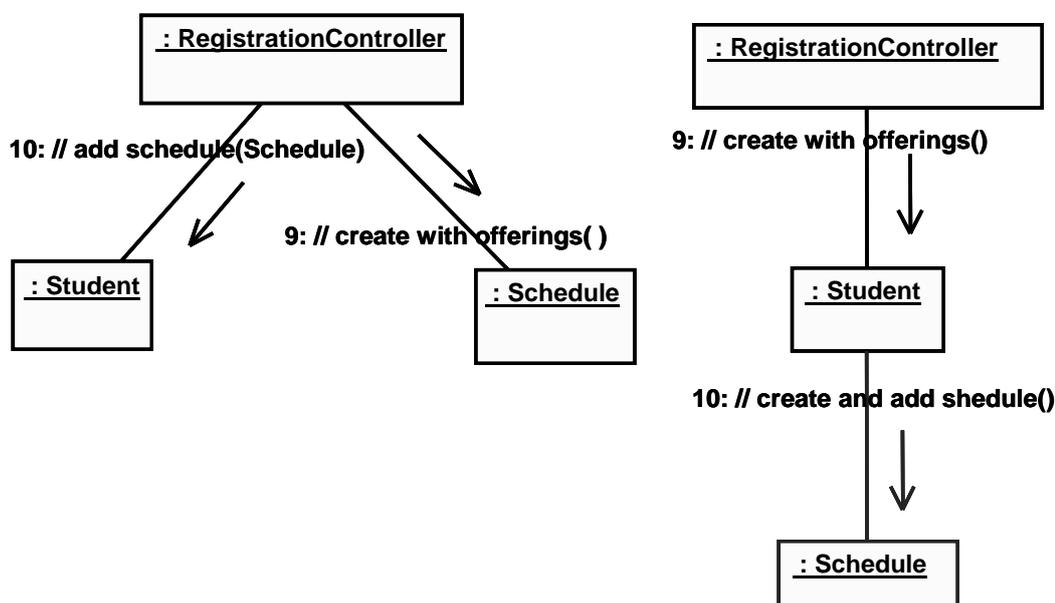
*Образец "Creator". Проблема:* Нужно определить, кто должен отвечать за создание нового экземпляра некоторого класса. Создание новых объектов в объектно-ориентированной системе является одним из стандартных видов деятельности. Следовательно, при назначении обязанностей, связанных с созданием объектов, полезно руководствоваться некоторым основным принципом. *Решение:* Следует назначить классу В обязанность создавать экземпляры класса А, если выполняется одно из следующих условий:

- класс В агрегирует, содержит или активно использует объекты класса А;
- класс В обладает данными инициализации, которые будут передаваться объектам класса А при их создании (т.е. класс В является информационным экспертом).

Класс В при этом определяется как создатель (creator) объектов класса А.

Если несколько классов удовлетворяют этим условиям, то предпочтительнее использовать в качестве создателя класс, агрегирующий или содержащий класс А.

*Пример:* При выполнении подчиненного потока событий "Создать график" варианта использования "Зарегистрироваться на курсы" необходимо решить, кто должен отвечать за создание нового графика в системе. На рис. показаны два возможных варианта решения этой



задачи.

Согласно образцу "Creator", оба решения подходят, так как с одной стороны объект-контроллер обладает данными, необходимыми для инициализации, с другой стороны объект Student агрегирует объекты Schedule.

*Следствия:* Образец "Creator" определяет способ распределения обязанностей, связанный с процессом создания объектов. В объектно-ориентированных системах эта задача является наиболее распространенной. Основным назначением образца Creator является выявление

объекта-создателя, который при возникновении любого события должен быть связан со всеми созданными им объектами. При таком подходе обеспечивается низкая степень связанности объектов.

В некоторых случаях в качестве создателя выбирается класс, который содержит данные инициализации, передаваемые объекту во время его создания. На самом деле это пример использования образца Information Expert.

*Образец "Low Coupling" (низкая связанность). Проблема:* Нужно распределить обязанности между классами таким образом, чтобы снизить взаимное влияние изменений в них и повысить возможность повторного использования. *Решение:* Следует распределить обязанности таким образом, чтобы обеспечить низкую связанность. Связанность (coupling) – это мера, определяющая, насколько жестко один элемент связан с другими элементами, или каким количеством данных о других элементах он обладает. Элемент с низкой связанностью зависит от небольшого числа других элементов. Класс с высокой связанностью зависит множества других классов. Наличие таких классов нежелательно, поскольку оно приводит к возникновению следующих проблем:

- Изменения в связанных классах приводят к локальным изменениям в данном классе.
- Затрудняется понимание каждого класса в отдельности.
- Усложняется повторное использование, поскольку для этого требуется дополнительный анализ классов, с которыми связан данный класс.

*Пример:* Рассмотрим подчиненный поток событий "Создать график" варианта использования "Зарегистрироваться на курсы" (предыдущий рисунок). Согласно образцу "Low Coupling", наилучшим решением является вариант справа, поскольку при этом у класса RegistrationController будет на одну связь меньше (т.е., будет обеспечена более низкая связанность).

*Следствия:* Образец Low Coupling поддерживает независимость классов, что, в свою очередь, повышает возможности повторного использования. Его нельзя рассматривать изолированно от других образцов, таких как Information Expert и High Cohesion. Он также обеспечивает выполнение одного из основных принципов проектирования, применяемых при распределении обязанностей.

*Образец "High Cohesion" (высокая функциональная прочность или сильное зацепление обязанностей). Проблема:* Нужно распределить обязанности между классами таким образом, чтобы каждый класс не выполнял много разнородных функций или несвязанных между собой обязанностей. Такие классы создавать нежелательно, поскольку они приводят к возникновению таких же проблем, как у классов с сильной связанностью. *Решение:* Следует распределить обязанности таким образом, чтобы обеспечить высокую функциональную прочность. В терминах объектно-ориентированного проектирования функциональная прочность (cohesion) – это мера взаимодействия и непротиворечивости обязанностей класса. Считается, что элемент обладает высокой прочностью, если его обязанности тесно связаны между собой, и он не выполняет излишнего объема работы. В роли таких элементов могут выступать классы, подсистемы, модули и т.д.

Классы с низкой прочностью, как правило, выполняют обязанности, которые можно легко распределить между несколькими классами. Грубо говоря, можно поделить класс два или более, отмерив каждой части подмножества атрибутов и операций, так чтобы получившиеся части были функционально прочны.

*Пример:* Используем тот же пример, что и для предыдущего образца. Согласно образцу "High Cohesion", наилучшим решением также является вариант справа, поскольку при этом класс RegistrationController делегирует обязанность создания нового объекта класса Shedule классу Student, и у самого класса RegistrationController будет на одну обязанность меньше (т.е., его прочность будет выше). Обязанность отданная Student относится к тому же роду, что и другие его обязанности, так как касается порождения объектов – частей.

*Следствия:* Как правило, класс с высокой прочностью содержит сравнительно небольшое число методов, которые функционально тесно связаны между собой, и не выполняет слишком много функций. Он взаимодействует с другими классами для выполнения более сложных задач. Высокая степень однотипной функциональности в сочетании с небольшим числом операций упрощают поддержку и модификацию класса, а также возможность его повторного использования.

Следует обратить внимание, что обязанностью экземпляров классов является не только прием сообщений, но и их отправка другим объектам. То есть объект *обязан* знать о других объектах, которым он *должен* будет посылать сообщения. Распределить обязанности такого рода позволяют образцы *Сценарий транзакции* и *Модель предметной области*. Они введены Мартином Фаулером в книге «Архитектура корпоративных программных приложений».

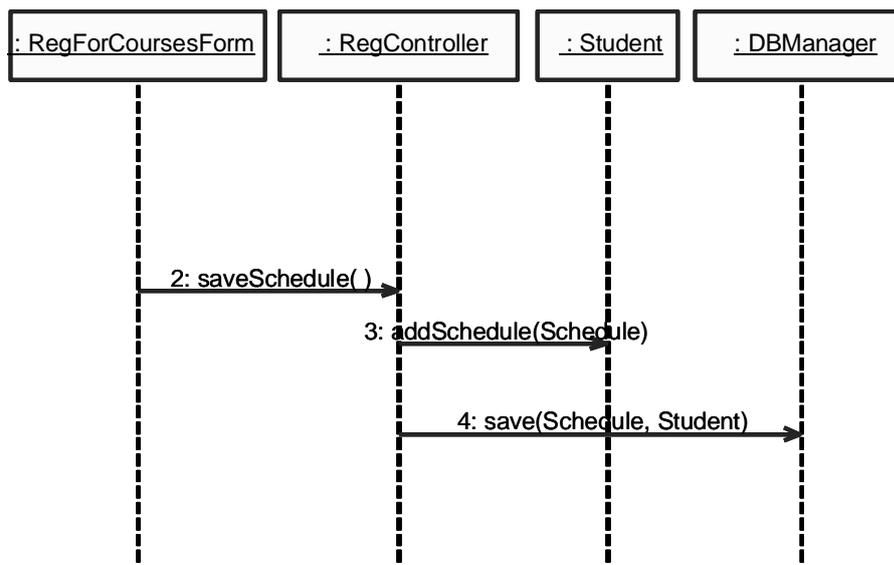
*Образец «Сценарий транзакции»*

**Аннотация:** В управляющем классе заводится по одной операции на каждый запрос. Экземпляр управляющего класса обеспечивает правильную последовательность шагов сценария обработки каждого запроса, рассылая сообщения объектам сущностям, которые сами по себе не реализуют сложного поведения. Типовая последовательность шагов такова: 1) прием входного запроса; 2) получение данных из базы; 3) обработка данных; 4) выдача результата. Все сложное поведение реализовано в контроллерах.



**Эскиз:**

**Назначение:** Главное достоинство: простота, естественность, производительность. Подходит для небольших приложений. Недостаток: при усложнении бизнес-логики дублируется большое количество кода – снижается гибкость и сопровождаемость. В таких случаях нужно применять образец «Модель предметной области».

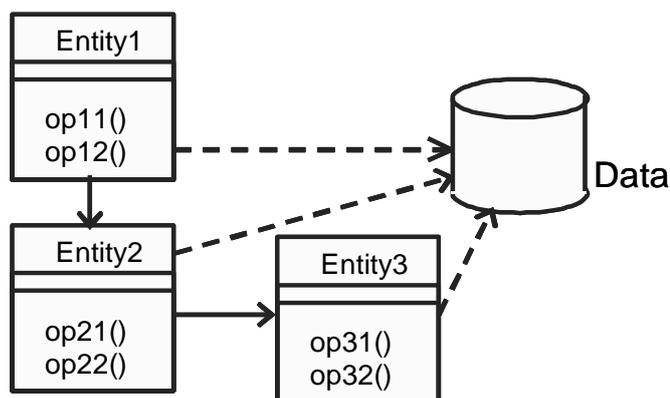


Пример взаимодействия согласно образцу «Сценарий транзакции»:

Объект-контроллер ведет транзакцию (сохранение расписания) по сценарию от начала до конца.

*Образец «Модель предметной области»*

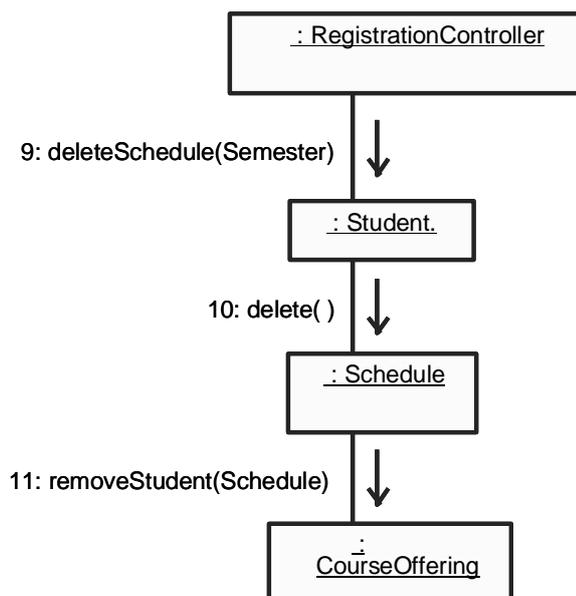
**Аннотация:** Обязанности по обработке запросов распределены по сети объектов-сущностей. В приложении создается слой объектов, описывающих структурные и поведенческие аспекты предметной области. Поведение сочетается с данными и реализуется в сущностях. Управляющие объекты довольствуются ролью посредников между граничными объектами и сущностями.



**Эскиз:**

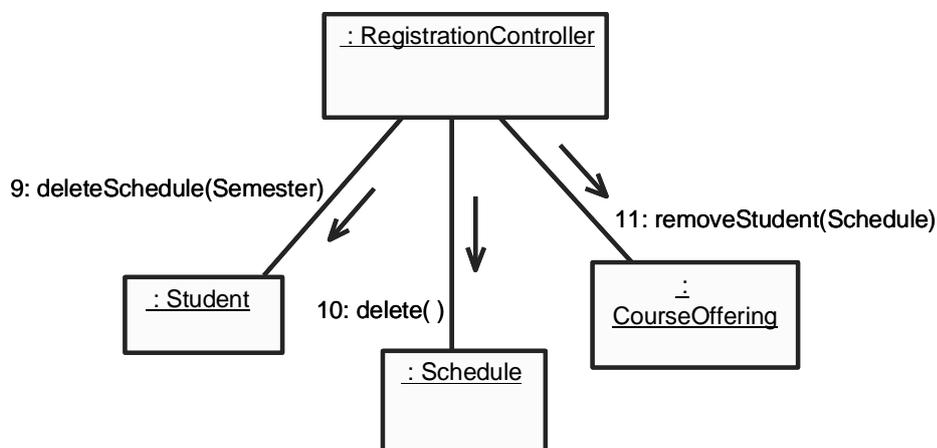
**Назначение:** Подходит для приложений с запутанной бизнес-логикой. Недостаток: создание модели предметной области более трудоемко, чем реализация сценария транзакции. В простых случаях нужно применять сценарий транзакции.

**Пример:** в системе регистрации на курсы бизнес-логика распределена между классами-



сущностями

Отдельные этапы удаления расписания реализуются отдельными объектами. Объект студент обнуляет в себе ссылку на расписание и передает работу дальше объекту-расписанию. Объект-расписание отсылает запрос на удаление ссылки на себя из объекта-курса и освобождает занимаемую собой память.



Если бы применялся *сценарий транзакции*, взаимодействие выглядело бы так:

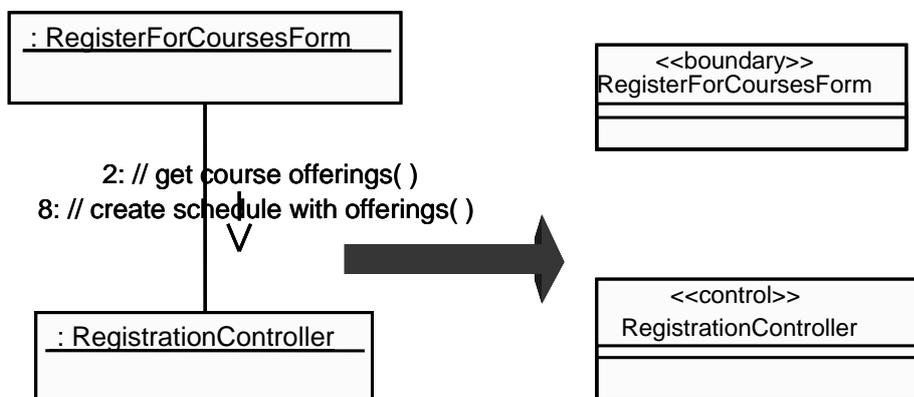
В этом варианте всю транзакцию ведет объект-контроллер.

Набор обязанностей классов, полученный в результате их распределения, должен быть проанализирован на предмет выявления и устранения следующих проблем:

- дублирования одинаковых обязанностей в различных классах;
- противоречивых обязанностей в рамках класса;
- классов с одной обязанностью или вообще без обязанностей;
- классов, взаимодействующих с большим количеством других классов.

Атрибуты классов анализа определяются, исходя из знаний о предметной области, требований к системе, глоссария и бизнес-модели. В процессе анализа атрибуты определяются только для классов-сущностей. Атрибуты должны быть простыми. Сложные атрибуты должны моделироваться как ассоциации между классами.

Связи между классами определяются в два этапа. Начальный набор связей определяется на основе анализа кооперативных диаграмм. Если два объекта обмениваются сообщениями, между



ними должна быть ассоциация.

На втором этапе, исходя из знаний о предметной области, создаются связи между классами-сущностями (ассоциации, агрегации, обобщения). Композиции выделяются на этапе проектирования, во время анализа используются только агрегации.

Квалификация механизмов анализа состоит в том, что:

- Составляется список всех механизмов анализа.
- Классы анализа ставятся в соответствие механизмам анализа.
- Определяются характеристики механизмов анализа.

Механизмы анализа играют следующие роли:

- Отражают нефункциональные требования к системе (надежность, безопасность и т.д.) и их реализацию в архитектуре системы.
- Представляют собой набор типовых решений, или образцов (patterns), принятых в качестве стандарта данного проекта.
- Позволяют сосредоточиться на преобразовании функциональных требований в программные абстракции, отвлекаясь от особенностей реализации.

Так, имея дело с устойчивыми классами, достаточно указать для них механизм «persistence», не задумываясь как именно будет реализовано сохранение их экземпляров в базе данных. Пример:

Класс анализа	Механизм(ы) анализа
Student	Persistence, Security
Schedule	Persistence, Security
CourseOffering	Persistence, Legacy Interface
Course	Persistence, Legacy Interface
RegistrationController	Distribution

В описания классов, сопоставленных с архитектурными механизмами, добавляются дополнительные характеристики, например:

Характеристики класса Schedule:

- Объем: до 2000 графиков.
- Частота доступа:
  - Создание: 500 в день.
  - Чтение: 2,000 обращений в час.
  - Обновление: 1,000 в день.
  - Удаление: 50 в день.

Унификация классов анализа заключается в изменении модели анализа таким образом, чтобы соблюдалось выполнение следующих условий:

- имя и описание каждого класса анализа должно отражать сущность его роли в системе;
- классы с одинаковым поведением, или представляющие одно и то же явление, должны объединяться;
- классы-сущности с одинаковыми атрибутами должны объединяться (даже если их поведение отличается).

По результатам унификации классов должны быть модифицированы описания вариантов использования.

По окончании модель анализа должна быть подвергнута проверке:

1. Все ли классы обоснованы надлежащим образом?
2. Отражает ли имя каждого класса его роль?
3. Представляет ли класс единственную, четко определенную абстракцию?
4. Являются ли все атрибуты и обязанности класса функционально связанными?
5. Отражают ли классы всю функциональность вариантов использованию, заключенную в основных, подчиненных и альтернативных потоках событий?
6. Однозначно ли распределено поведение по классам?

Упомянутые в лекции образцы подробно описаны в книгах [4] и [5].

## Литература к лекции 7

5. Вендров А. М. Проектирование программного обеспечения экономических информационных систем. 2-е изд. – М.: Финансы и статистика, 2005. – Глава 3.
6. Рамбо Дж., Блаха М. UML 2.0. Объектно-ориентированное моделирование и разработка. 2-е изд.: Пер. с англ. – СПб.: Питер, 2007. – Главы 12-13.
7. Кулямин В. В. Технологии программирования. Компонентный подход. – М.: Бином. Лаборатория знаний. 2007. Лекция 4.
8. Ларман Крэг. Применение UML 2.0 и шаблонов проектирования. 3-е изд. Пер. с англ. – М.: Вильямс, 2007.
9. Фаулер М. Архитектура корпоративных программных приложений. – М.: Вильямс, 2007.