Московский государственный университет

имени М.В.Ломоносова

Факультет вычислительной математики и кибернетики

Кафедра системного программирования

Курс

«Объектно-ориентированный анализ и проектирование»

Лектор: доц. кафедры СП, канд. физ.-мат. наук Малышко В. В.

Авторы программы: канд. техн. наук Вендров А. М., канд. физ.-мат. наук Малышко В. В.

Москва
2009

Оглавление

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 взаимодействует 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.

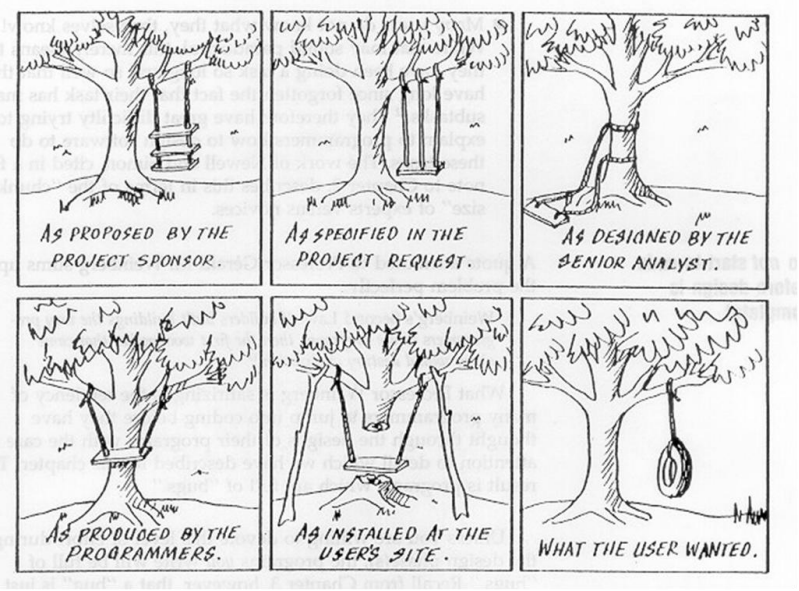
С конца 60-х годов прошлого века до сегодняшних дней продолжается так называемый *«кризис ПО»*. Выражается он в том, что большие проекты выполняются с превышением сметы расходов и/или сроков отведенных на разработку, а разработанное ПО не обладает требуемыми функциональными возможностями, имеет низкую производительность и качество. По результатам исследований американской индустрии разработки ПО, выполненных в 1995 году Standish Group (www.standishgroup.com), 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	1998	2004	2009
31% аннулируются до завершения	28%	18%	24%
53% не укладываются в поставленные сроки, превышают запланированные расходы и не реализуют в полном объеме требуемые функции	46%	53%	44%
16% завершаются в срок	26%	29%	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.

Проблемы, возникающие при создании программной системы, иллюстрирует карикатура «Качели» появившаяся в программистском фольклоре в 1970-х.



Первый рисунок (верхний слева) показывает, что заказчик не может толком сформулировать требования, завышает их. На втором рисунке (верхнем в середине) демонстрируется, что ответное предложение поставщика не вполне соответствует заявке заказчика, детали поняты превратно. Аналитик предлагает ошибочную эскизную

архитектуру (вверху справа). Программисты создают код с ошибками (внизу слева). После отладки система введена в действие (внизу в середине). То, что на самом деле требовалось, указано на последнем рисунке (внизу справа).

При планировании проектов зачастую по тем или иным причинам устанавливаются невыполнимые сроки, закладываются недостаточные ресурсы. Таким образом, возникают *безнадежные проекты* (death march projects)[1]. 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 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»[2] 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.

Бруксу принадлежит метафора проекта в виде крупного зверя попавшего в смоляную яму. Такое представление наглядно изображает необходимость компромиссов при разработке ПО. Как зверь не может освободить из смолы одну лапу, не утопив глубже другие, так и руководитель проекта может решать одну проблему (например, повышает качество ПО) только за счет усугубления других (отладка, шлифовка требуют времени, денег, людей).

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;

---

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

[2] Брукс Ф. Мифический человеко-месяц или как создаются программные системы. – СПб.: Символ-Плюс, 1999

- 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.

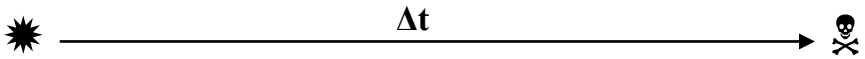
Этапы становления и развития SE:

- 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; гарантированнее 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.

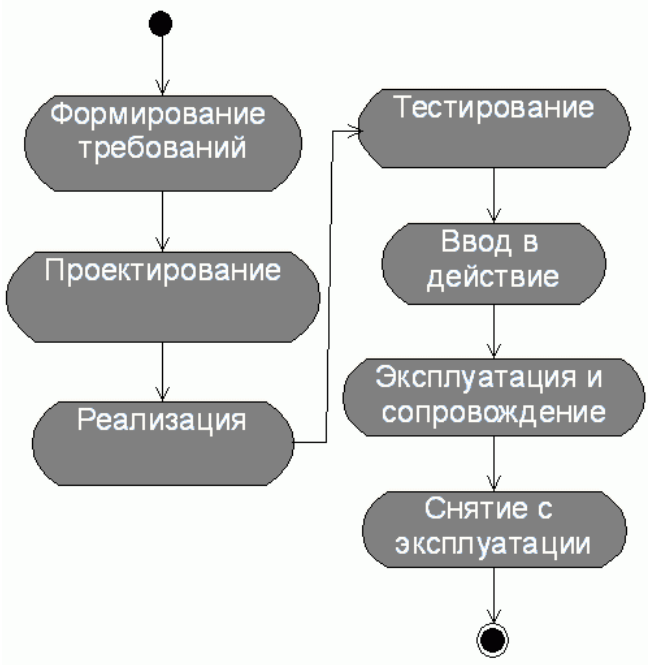
Процесс разрешения проблем предусматривает анализ и разрешение проблем, возникающих в течение ЖЦ ПО.

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);
- 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 предобразования 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.

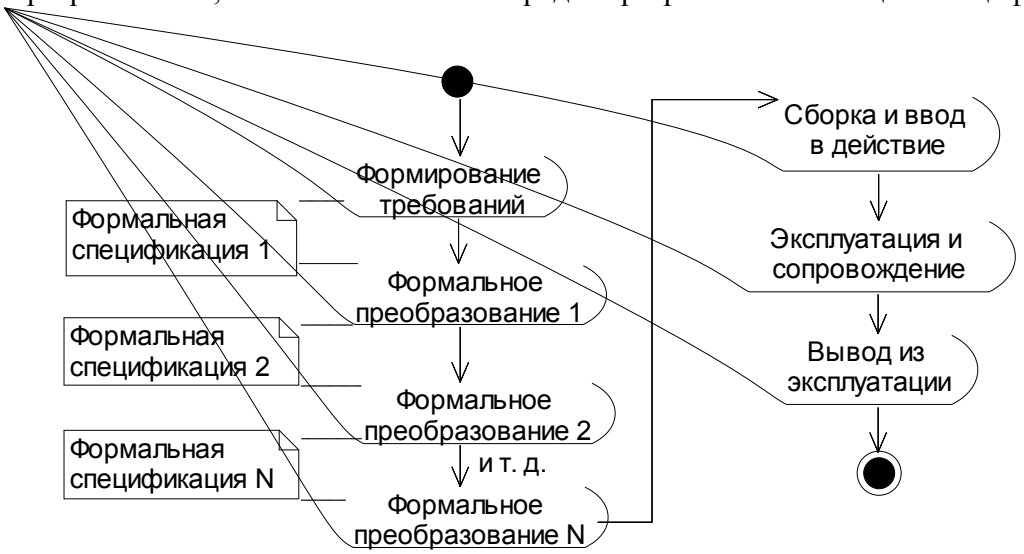


Схема модели ЖЦ, основанной на формальных преобразованиях

Особенности *итерационных моделей*:

- процесс разработки разбивается на последовательность шагов, выполняемых циклически;
- разные виды деятельности не привязаны намертво к определенным этапам разработки, а выполняются по мере необходимости, иногда 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 каскада.

Схема пошаговой итерационной модели ЖЦ

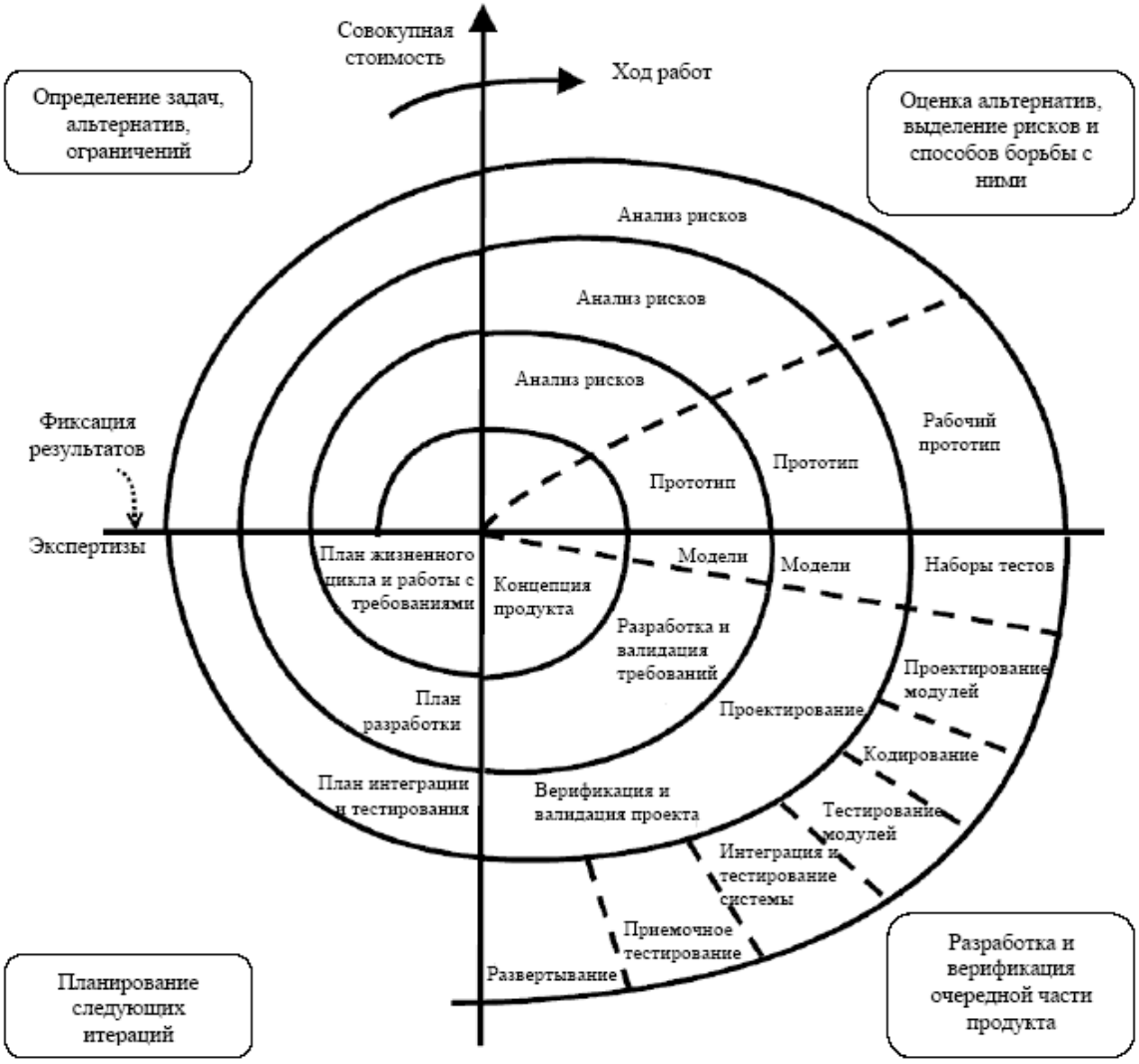


Схема спиральной модели ЖЦ

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 «миникаскад», 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».

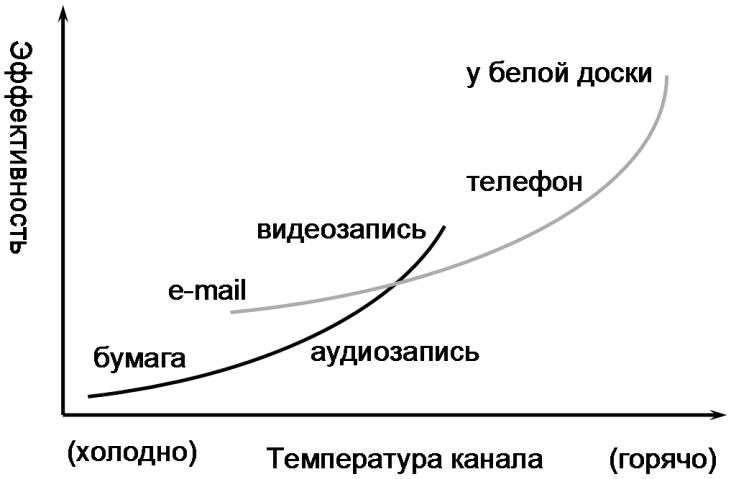
Манифест гибкой разработки:

- 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.

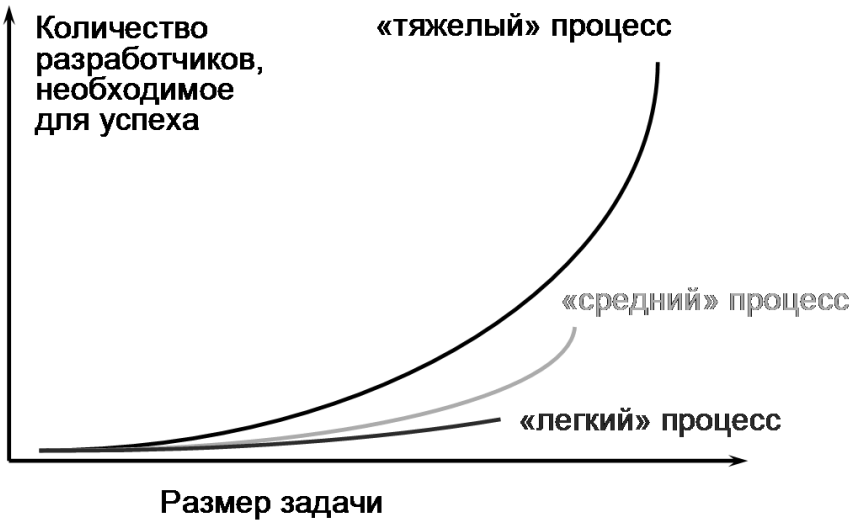
Важно, что стоящее в левой части не отменяет стоящего в правой (документация, процессы, инструменты планы важны, но в меньшей степени, чем индивидуумы, взаимодействия, работающий «софт» и реагирование на изменения).

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. Легкий процесс не позволит большой команде, необходимой для реализации крупного проекта, взаимодействовать достаточно эффективно.



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.
6. Дисциплина, умение и понимание противостоят процессу, формальности и документированию.
7. Потеря эффективности в некритических видах деятельности вполне допустима.

Основные направления развития современной программной инженерии:

1) Управление требованиями
2) Управление конфигурацией и изменениями
3) Управление качеством ПО
4) Итерационная разработка ПО
5) Использование компонентной архитектуры (объектно-ориентированный подход)
6) Визуальное моделирование ПО

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, ограничивающее 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 связность» 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 OOA и OOD (Booch, OMT, ...), появился язык Java;
- 1997: Принят стандарт OMG UML 1.1.

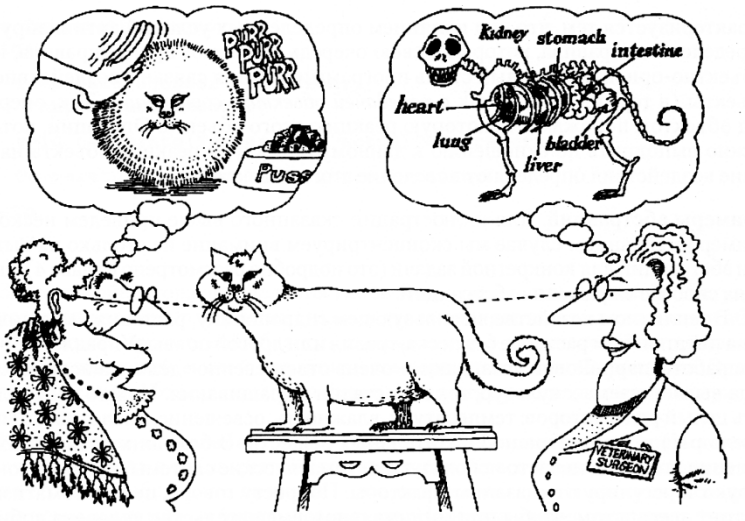
В основе объектно-ориентированного подхода лежит объектная декомпозиция, при этом статическая структура ПО описывается в терминах объектов и связей между ними, а динамический аспект ПО описывается в терминах обмена сообщениями между объектами. Каждый объект системы обладает своим собственным поведением,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

Объектная модель является естественным способом представления реального мира. Она является концептуальной основой ООП. *Основными принципами* ее построения являются:

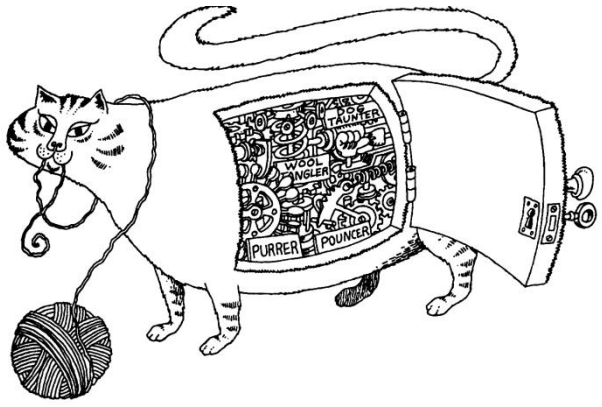
- абстрагирование;
- инкапсуляция;
- модульность;
- иерархия.

Дополнительные принципы:

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



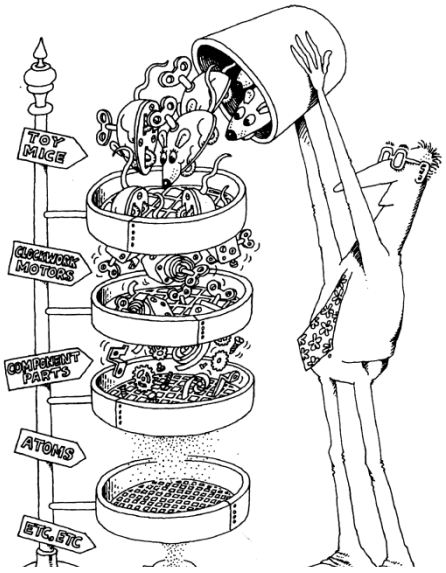
Абстрагирование – это выделение наиболее существенных характеристик некоторого объекта, отличающих его от всех других видов объектов, важных с точки зрения дальнейшего рассмотрения и анализа, и игнорирование менее важных или незначительных деталей. Абстракцией является любая модель, включающая наиболее 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.



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

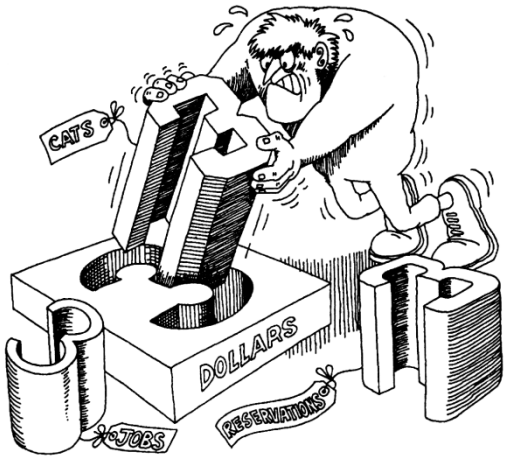
Модульность – это свойство системы, связанное с возможностью ее декомпозиции на ряд внутренне сильно сцепленных, но слабо связанных между собой подсистем (частей). Модульность снижает сложность системы, позволяя выполнять независимую разработку ее отдельных частей.



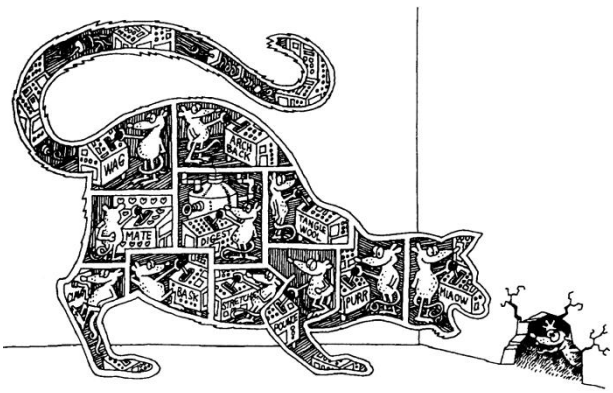
Иерархия – ранжированная или упорядоченная система абстракций, расположение их по уровням в виде древовидной структуры. Элементы, находящиеся на одном уровне иерархии, должны также находиться на одном уровне абстракции. Основными видами иерархических структур сложных систем являются структура классов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

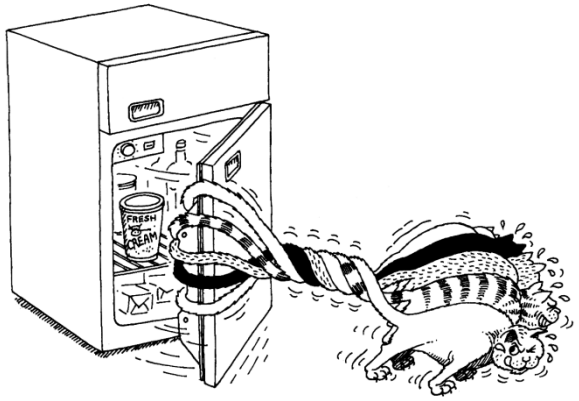
Переходим к основным понятиям объектно-ориентированного подхода (*элементам объектной модели*). К ним относятся: объект; класс; атрибут; операция; полиморфизм;

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.

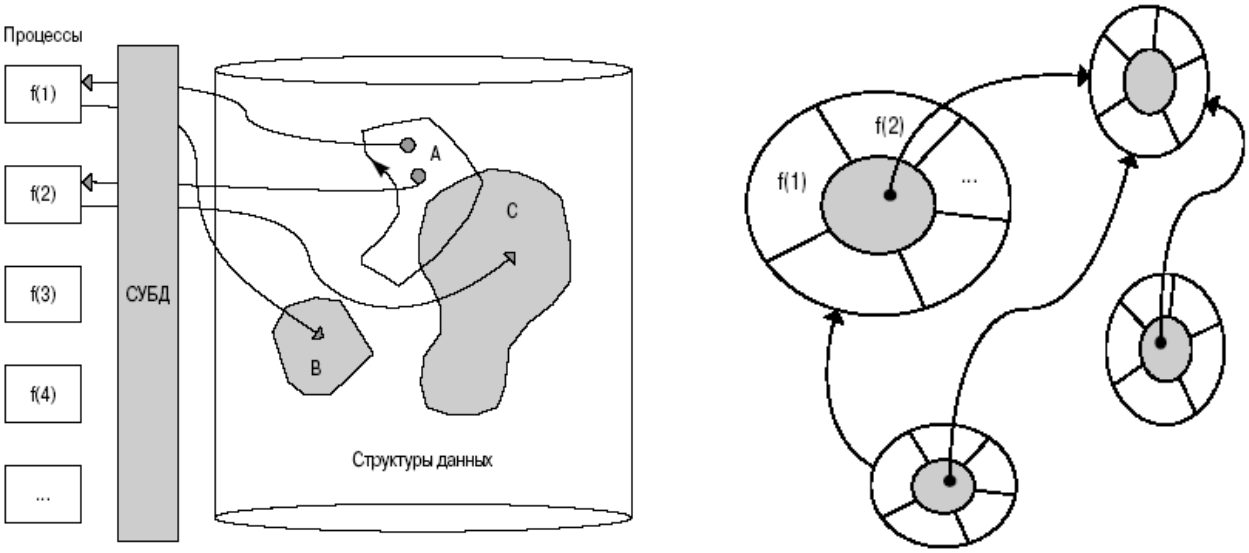
Операция – это реализация услуги, которую можно запросить у любого объекта данного класса. Операции реализуют связанное с классом поведение, его обязанности. Описание операции включает четыре части: имя; список параметров; тип возвращаемого значения; видимость.

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 (cpp-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:

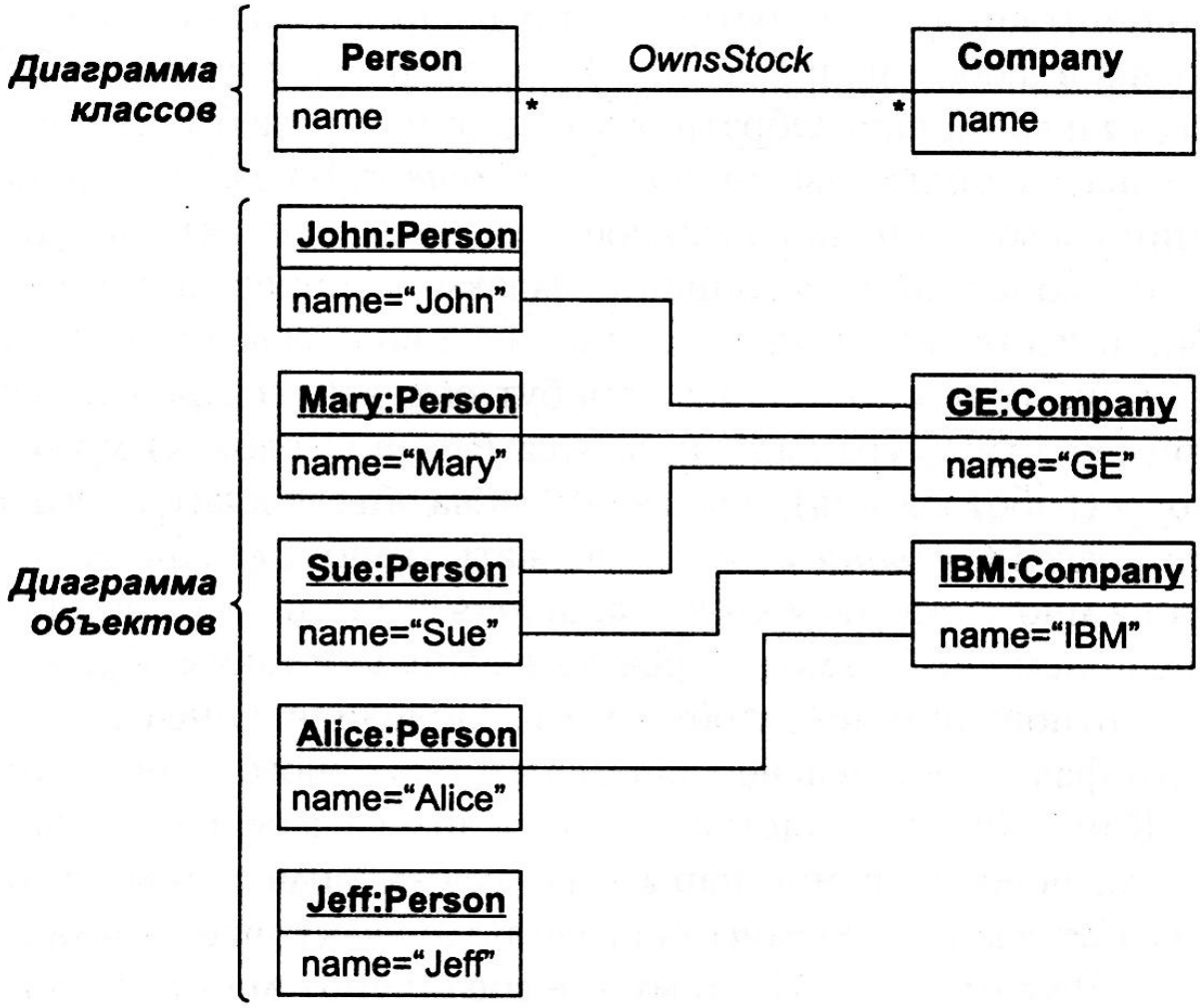
- средством организации модели в процессе разработки, повышения ее управляемости и читаемости;
- единицей управления конфигурацией.

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*.

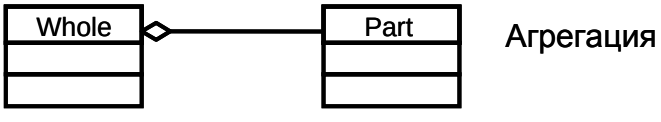
Соединение (link) – физическая или концептуальная связь между *объектами*, позволяющая им взаимодействовать.

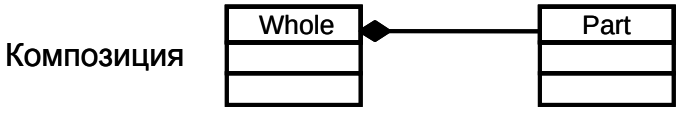
Ассоциация – связь между классами, описывающая группу однородных по структуре и семантике соединений между экземплярами классов. Соединения являются экземплярами ассоциации точно так же, как соединенные объекты являются экземплярами классов, связанных ассоциацией.



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.

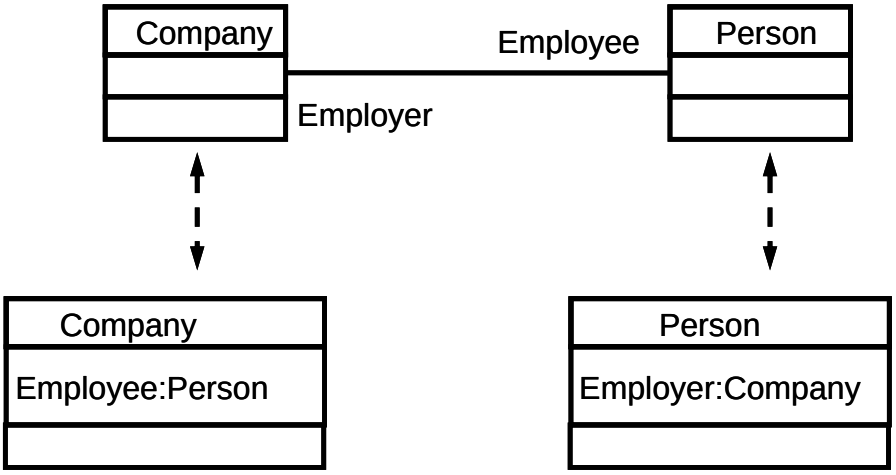
Агрегация – более сильный тип ассоциативной связи между целым и его частями (пример: автомобиль и мотор). *Композиция* – усиленная агрегация, когда часть не может существовать без целого (пример: университет, факультет, кафедра). Композиция и агрегация транзитивны, в том смысле, что если B является частью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





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

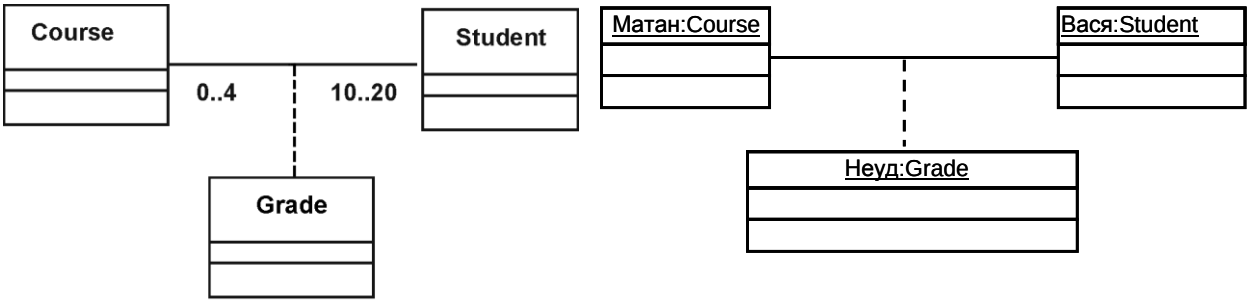
Ассоциации (включая агрегации и композиции) характеризуются: направлением, именем, ролевыми именами участников связи, мощностями. *Направление* указывает ход сообщений. По умолчанию ассоциации двунаправлены, т. е. сообщения могут исходить из любого конца ассоциации. Если введено ограничение по направлению, 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.



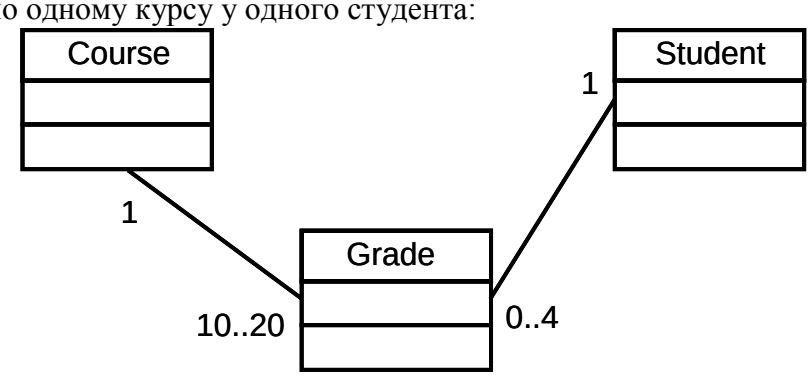
Мощность (multiplicity) показывает, как много объектов может участвовать в соединениях – экземплярах ассоциации. Мощность – это количество объектов одного класса (с той стороны связи, где приписана мощность), которые соединены с *одним* объектом другого класса (на другом конце связи). Для каждой ассоциации 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):

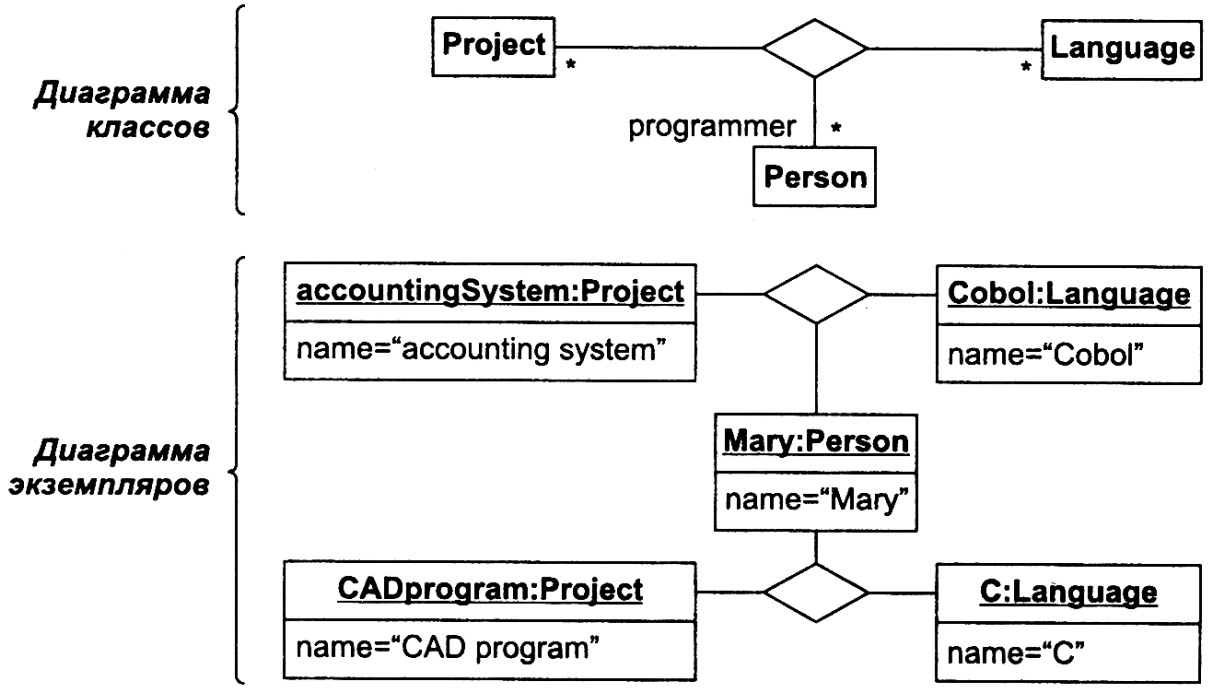


Следует заметить, что классы ассоциаций являются артефактами моделирования, то есть ими оперируют аналитики, архитекторы. Языки программирования не имеют средств, поддерживающих эти конструкции. В ходе реализации программистам приходится преобразовывать модели так, чтобы можно было создать код. При этом 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.

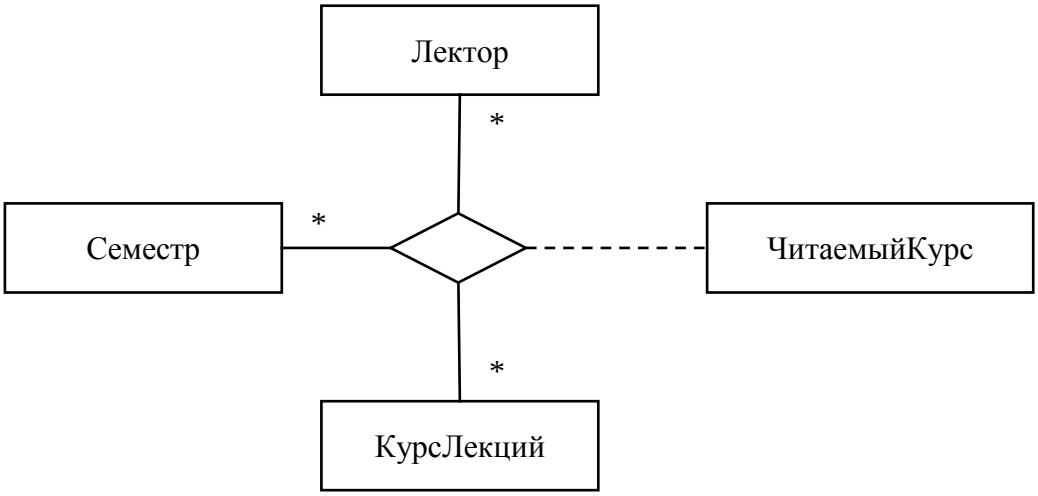
В UML 2.0 к объектной модели добавлено понятие N-арной ассоциации (для каждой комбинации объектов не более одной связи):



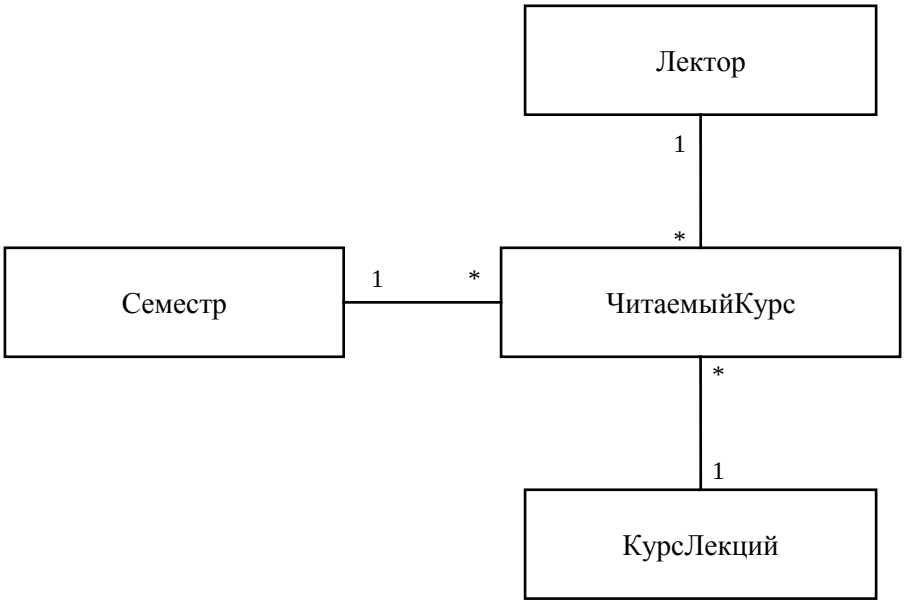
На рисунке представлена тернарная ассоциация и класс-ассоциация, связывающая проект, программистов, занятых в проекте, и языки, на которых они программируют в проекте. Некоторым аналогом тернарной ассоциации является реляционное отношение над тремя доменами (таблица со столбцами *проект, программист, 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 воспрещают 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.

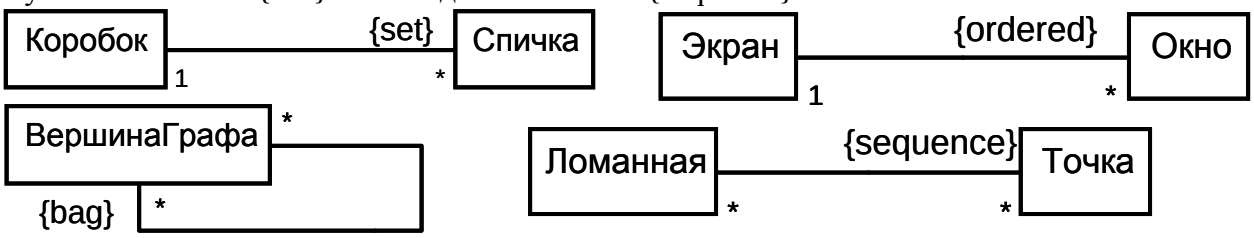


В объектно-ориентированных языках N-арные ассоциации не поддерживаются стандартными средствами. Их можно промоделировать с помощью обычных (бинарных) ассоциаций, но при этом снимается ограничение на единственность соединения, связывающего N-ку объектов:



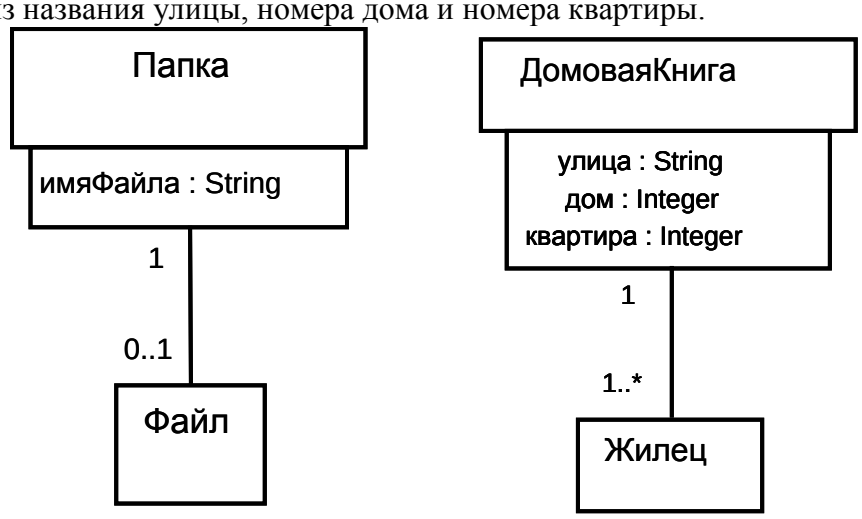
Во второй модели тройка объектов лекторПетров, семестрСедьмой и курсЛекцийМатан могут быть соединены более чем единожды посредством разных экземпляров класса ЧитаемыйКурс.

Ассоциации «1 ко многим» или «многие ко многим» имеют еще две характеристики: упорядоченность связываемых объектов; повторяемость (т. е. образуют ли связываемые объекты множество или мультимножество). Различные сочетания этих характеристик образуют четыре типа ассоциаций: множества {set}, упорядоченные множества {ordered}, мультимножества {bar} и последовательности {sequence}:



Спички в коробке образуют множество (неупорядоченное). Окна на экране – упорядоченное (по глубине) множество. В мультиграфе между парой вершин может быть несколько ребер. 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 неболее 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.

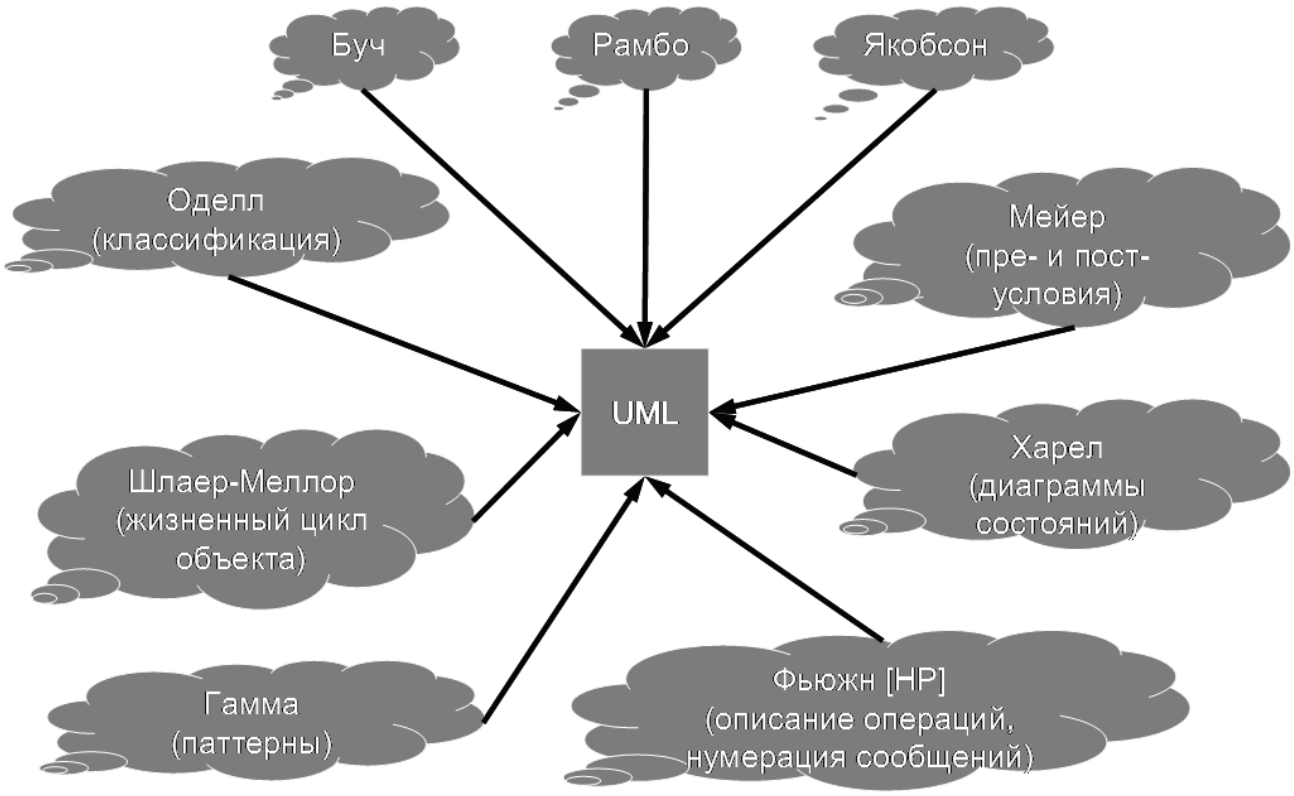
Общие атрибуты, операции и/или отношения отображаются на верхнем уровне иерархии. В объектной модели наследование может быть множественным. На связи могут накладываться ограничения. Например, если необходимо, множественное наследование в некоторой иерархии классов может быть запрещено (над связью указывается ключевое слово: **{disjoint}**).

Реализация – связь между контрактом (интерфейсом, вариантом использования) и его исполнением (классом, подсистемой, компонентой и т. п.). Изображается пунктирной стрелкой с треугольным наконечником, исходящая из исполнения и указывающая на контракт.

Лекция 3. Унифицированный язык моделирования (UML)

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.);

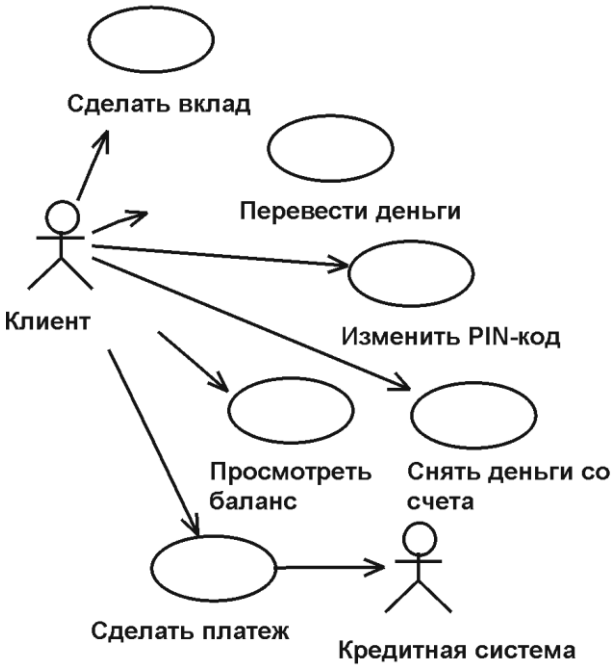
- связи (ассоциации, обобщения, зависимости и др.);
- механизмы расширения (стереотипы, ограничения, примечания, именованные значения);
- диаграммы.

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.

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.

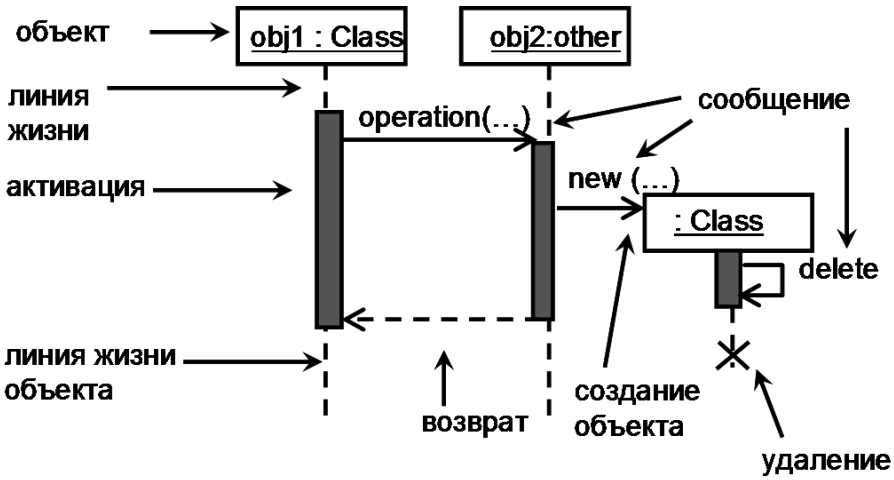
Диаграммы вариантов использования показывают, какие действующие лица инициируют варианты использования (от них идет стрелка к варианту использования). Из диаграмм понятно, какие действующие лица получают данные в ходе выполнения варианта использования (к ним идет стрелка от варианта использования).

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 процесса взаимодействия действующего лица с системой, реализуемого в рамках варианта использования.

В диаграммах вариантов использования может присутствовать несколько типов связей:

- связи коммуникации (линия со стрелкой, обозначающая связь между вариантом использования и действующим лицом);
- связи включения (пунктирная линия со стрелкой, обозначающая включение многократно используемой функциональности, представленной в виде абстрактного варианта использования);
- связи расширения (пунктирная линия со стрелкой, указывающая на особый случай, описанный в абстрактном варианте использования);
- связи обобщения (линия с треугольным концом, показывающая, что у нескольких действующих лиц имеются общие черты и различия).

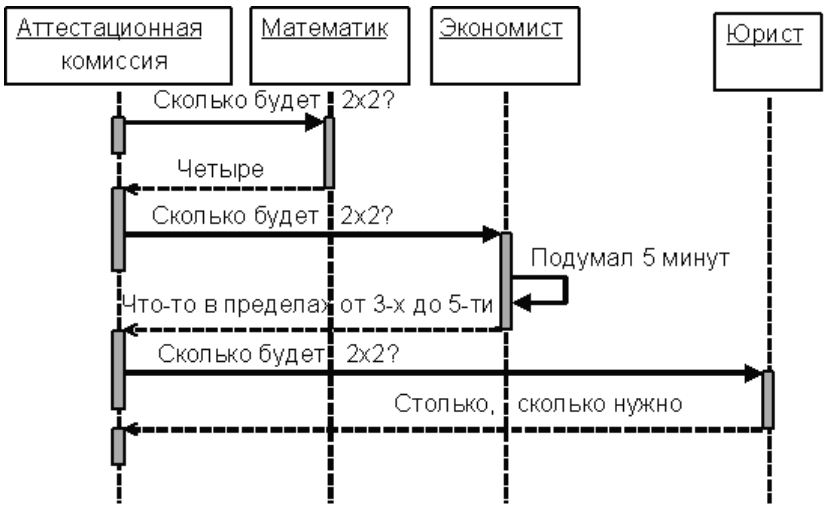
Диаграммы взаимодействия описывают поведение взаимодействующих групп объектов в рамках потока событий. На диаграмме отображается ряд объектов и сообщения, которыми они обмениваются между собой. Сообщение – это средство, с помощью которого объект-отправитель запрашивает у объекта-получателя выполнение 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.

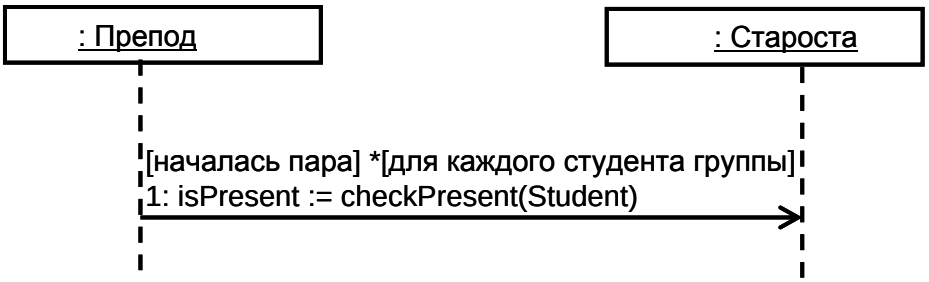
Каждое сообщение может быть описано в таком формате:

[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.

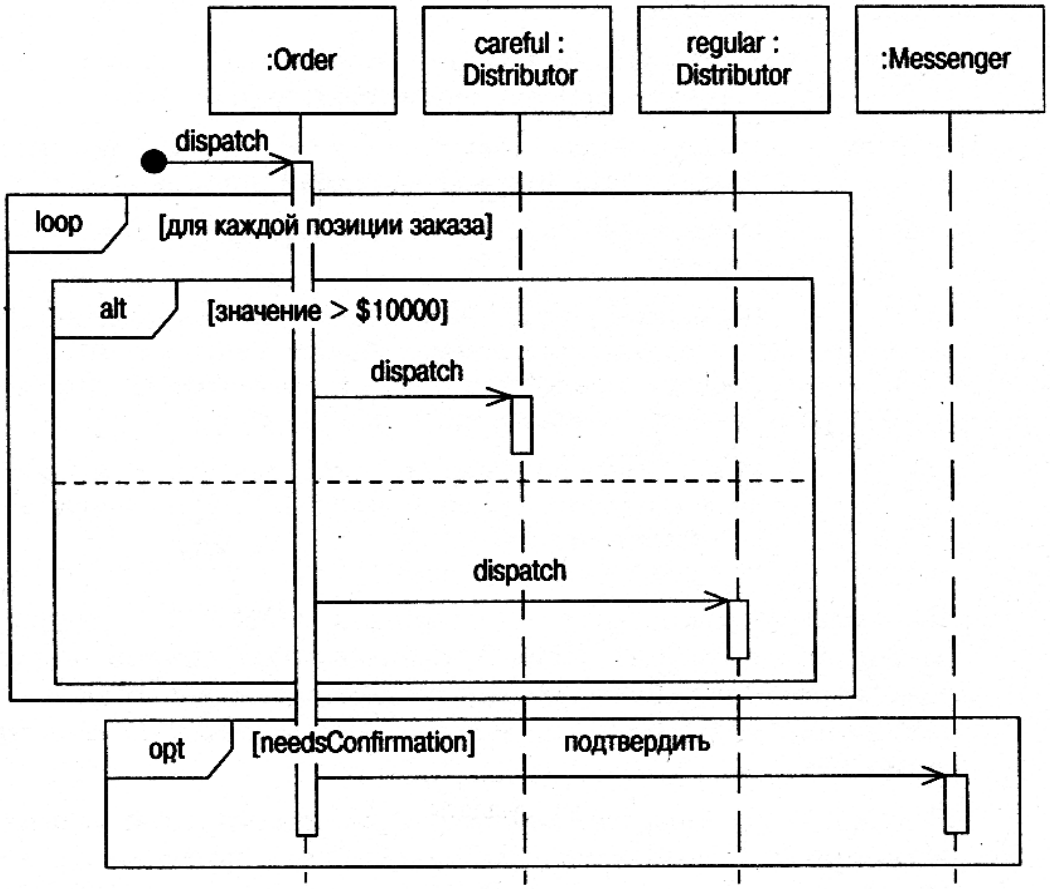
Любая из частей описания может отсутствовать.



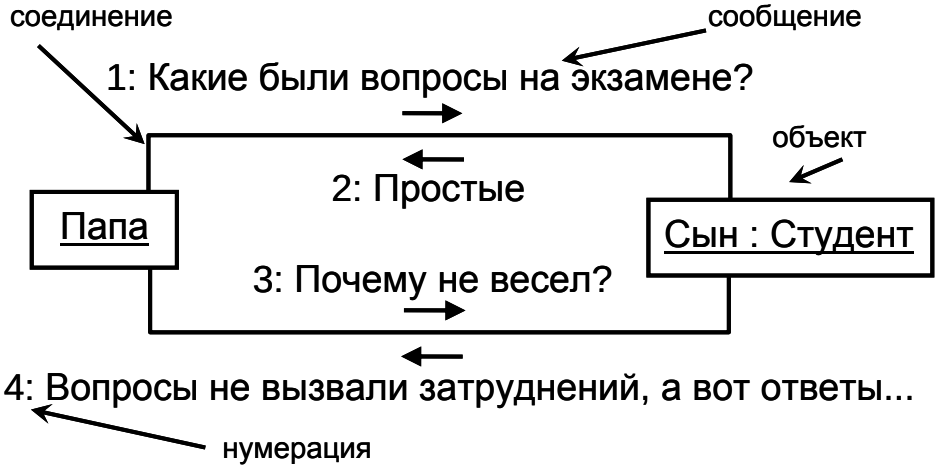
В примере показано взаимодействие при проверке посещаемости. В начале занятия по каждому студенту из списка преподаватель ждет ответа от старосты, присутствует ли студент.

В UML 2.0 диаграммы последовательности могут содержать блоки разных типов:

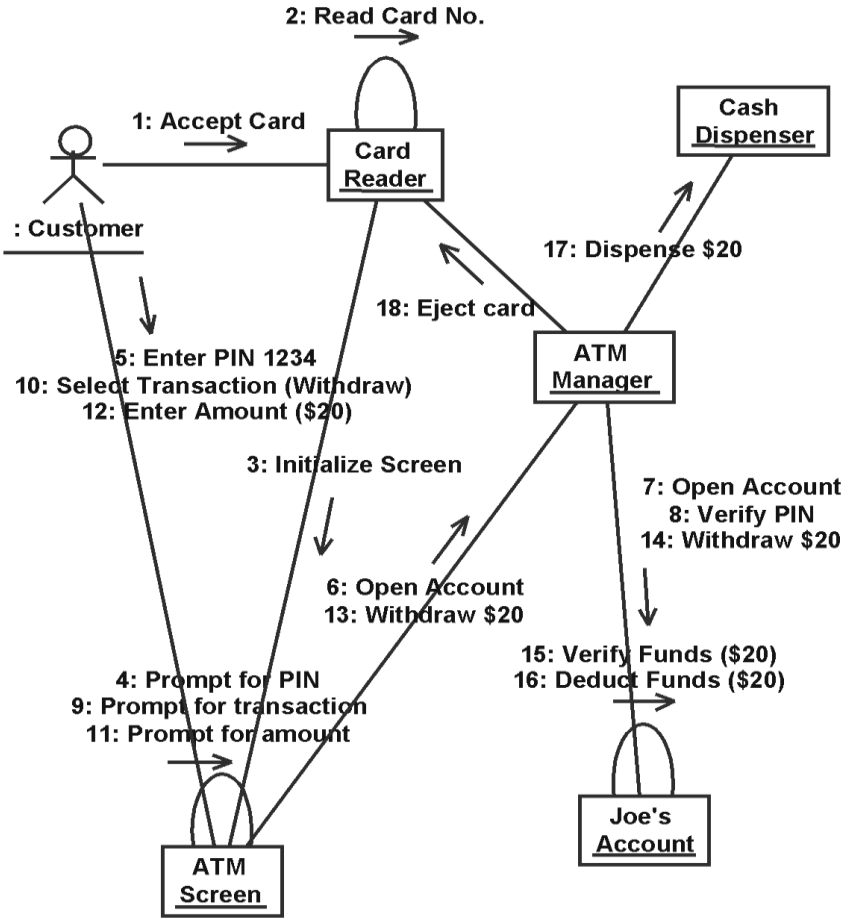
- alt – несколько альтернатив (каждая альтернатива – часть блока помеченная сторожевым условием);
- opt – необязательный блок (взаимодействие выполняемое при истинности сторожевого условия);
- par – параллельно выполняемые блоки;
- loop – цикл (пока истинно условие);
- region – критический участок;
-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



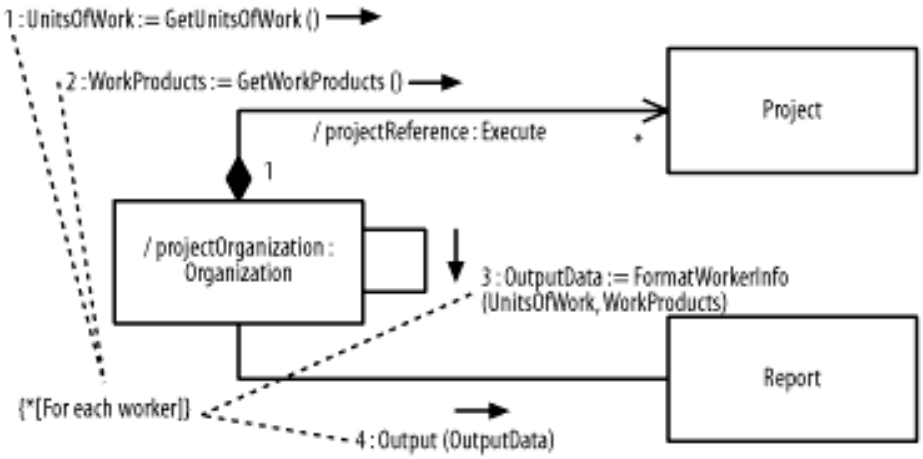
Вторым видом диаграмм взаимодействия являются *коммуникационные диаграммы* (в UML 1 их называли *кооперативными*). Как и диаграммы последовательности, они отображают поток событий варианта использования. На коммуникационных диаграммам внимание сконцентрировано на связях между объектами. Из них легче понять связи между объектами, однако, труднее уяснить последовательность событий. Объекты и/или действующие лица, обменивающиеся сообщениями, соединяются линиями, над которым 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.

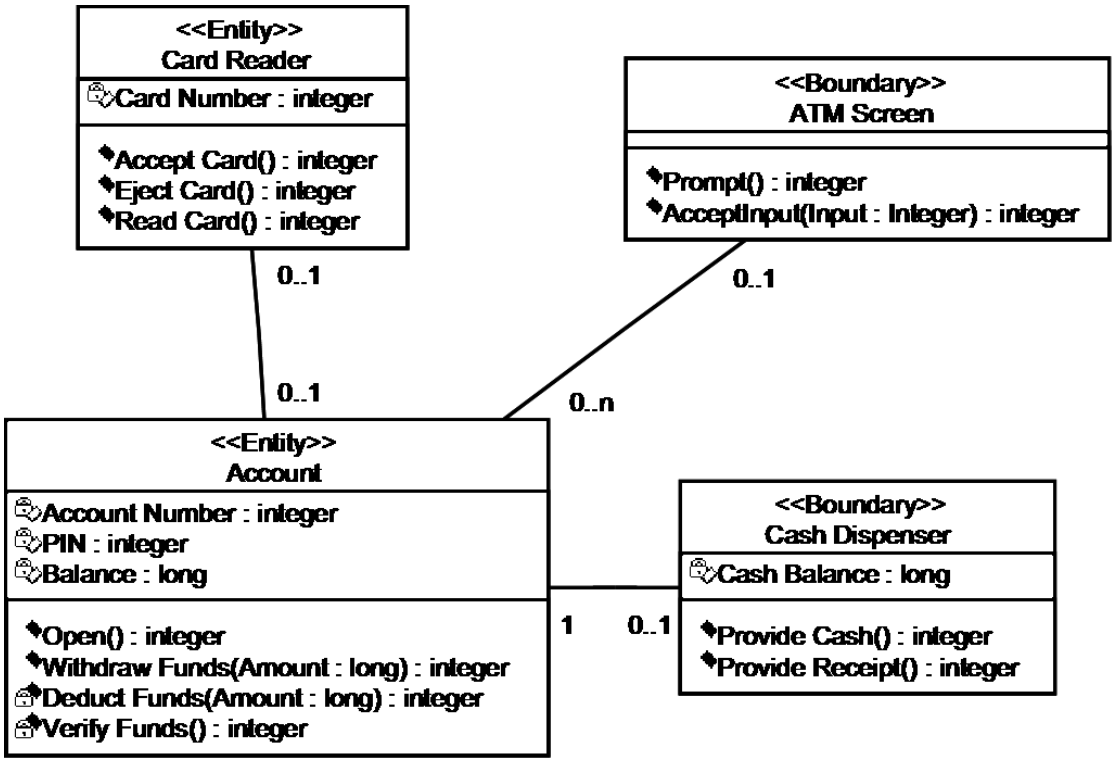


Если необходимо указать итерации во взаимодействии, на коммуникационных диаграммах используют примечания. Например:

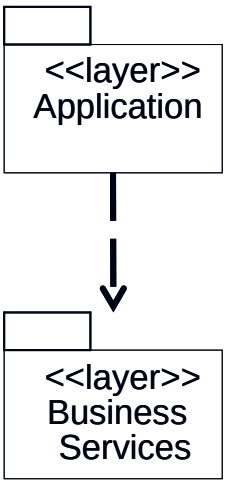


Примечание, заключенное в фигурные скобки, предписывает посылать последовательности из четырех сообщений для каждого работника в организации.

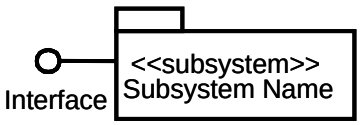
Диаграмма классов определяет классы системы и различного рода связи, которые существуют между ними (ассоциации, агрегации, композиции, зависимости, обобщения, реализации). На диаграммах классов изображаются также атрибуты классов, операции классов и ограничения, которые накладываются на связи между классами. Классы изображаются в виде прямоугольников, ассоциации – в виде линий со стрелками, агрегации и композиции – в виде линий с ромбом на конце, связь обобщения – в виде линии с треугольником на конце, зависимость – в виде пунктирной линии со стрелкой. Ответственность классов 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.



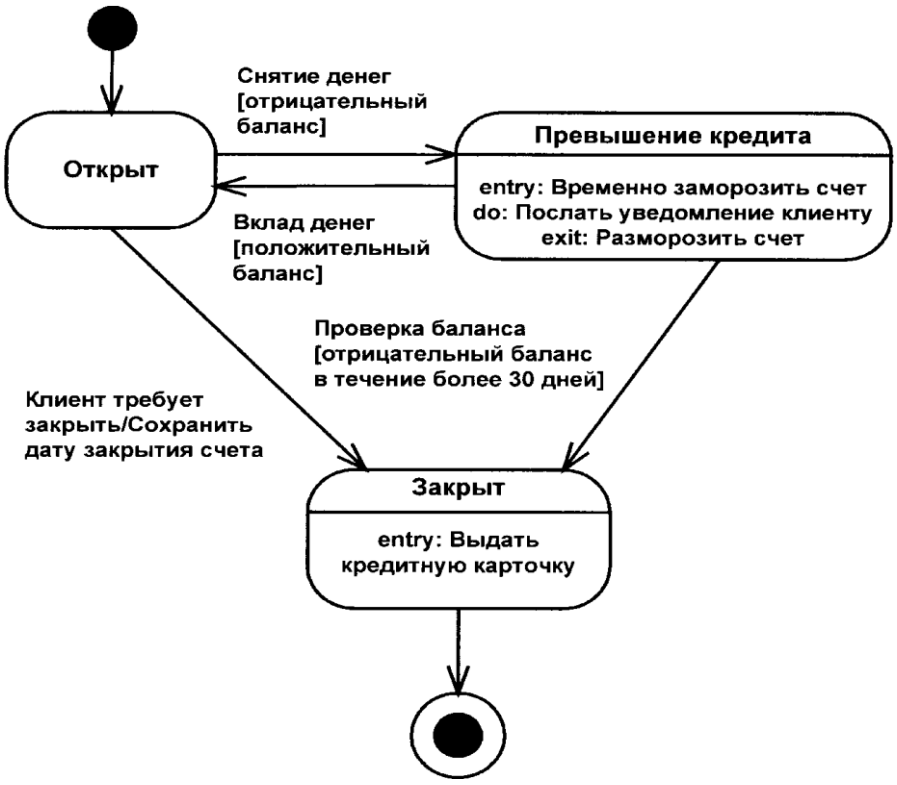
Для группировки классов, обладающих некоторой общностью, применяются пакеты. Пакет – общий механизм для организации элементов модели в группы. Каждый пакет – это группа элементов модели, иногда сопровождаемая диаграммами, поясняющими структуру группы. Каждый элемент модели может входить только в один 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 автомат. 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, что вполоть до уничтожения объекта с ним ничего происходить не будет. Из финального состояния не бывает



переходов, также как нет переходов в начальное.

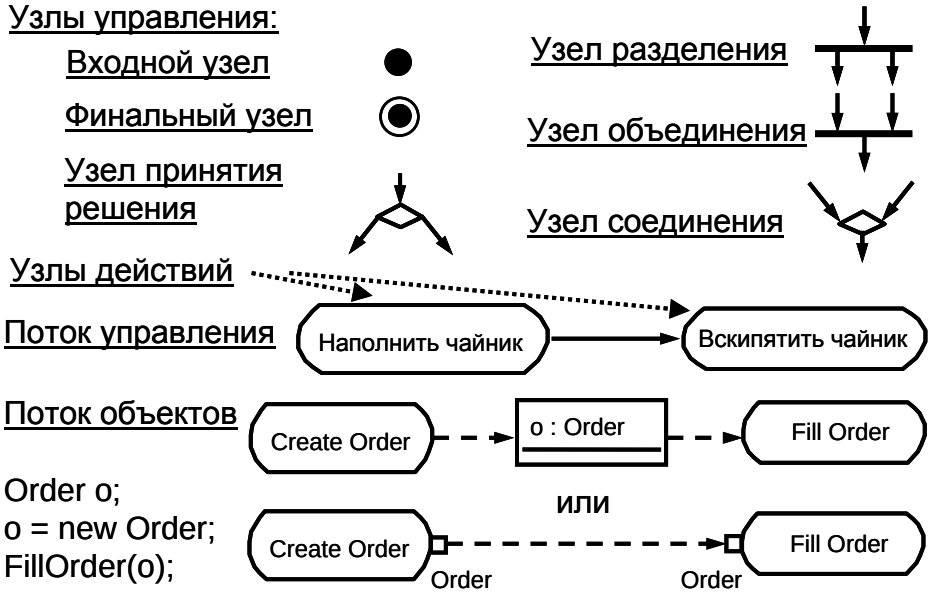
Когда объект находится в каком-то конкретном состоянии, могут выполняться различные процессы. Они, называются деятельностями состояния и указываются на диаграмме. Деятельность состояния – это прерываемое поведение. Оно может выполняться до своего завершения, пока объект находится в данном состоянии, 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, когда объект переходит в данное состояние, как часть этого перехода. 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 выхода из данного состояния. Оно также является непрерываемым. Его изображают внутри состояния, ему предшествует слово *exit* и двоеточие. Действия, выполняемые в состоянии по наступлению события, помечаются словом *event*, после которого через двоеточие указывается событие, а затем действие.

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

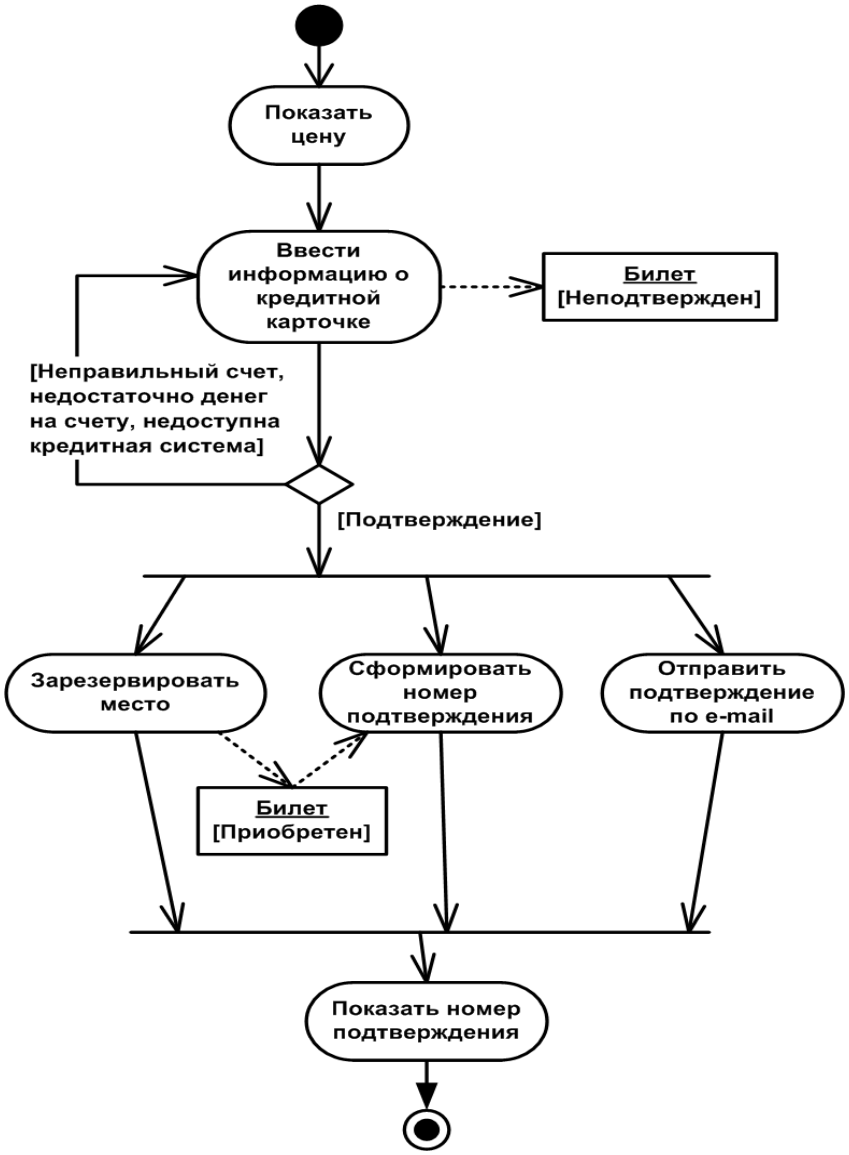
изображают на диаграмме вдоль линии перехода после имени события, заключая в квадратные скобки [ ]. Действие, являющееся частью перехода, изображают вдоль линии перехода после имени события, ему предшествует косая черта. Для некоторых состояний (называемых суперсостояниями) указывают вложенные подсостояния. 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.

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

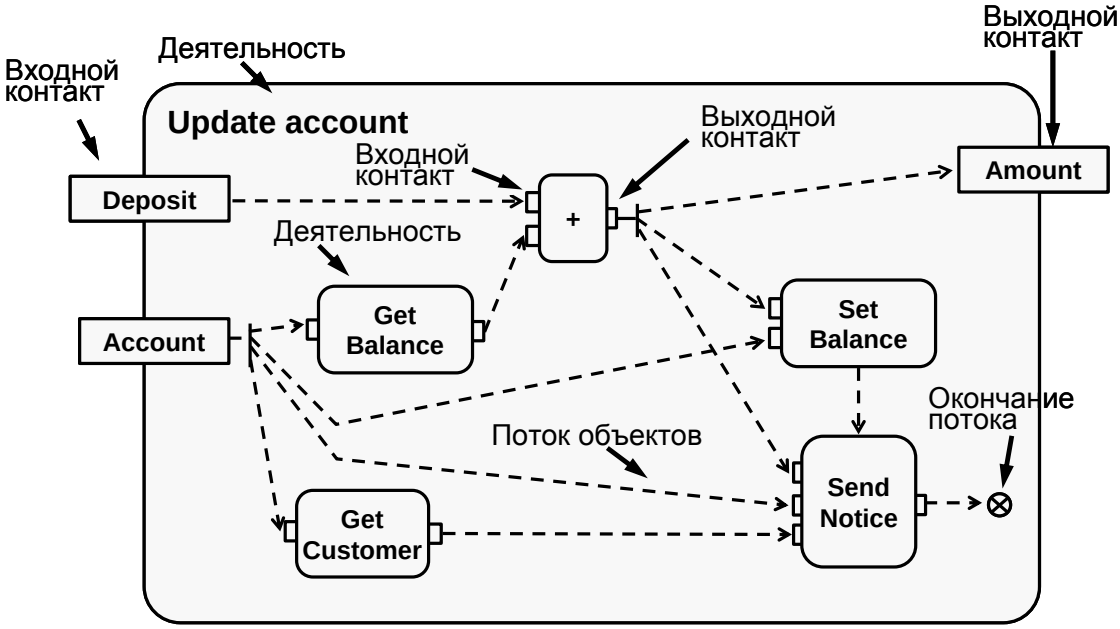
Диаграммы деятельности полезны в описании поведения, включающего большое количество параллельных процессов. Также их можно применять для представления потоков событий вариантов использования в наглядной графической форме. Элементы диаграмм деятельности:



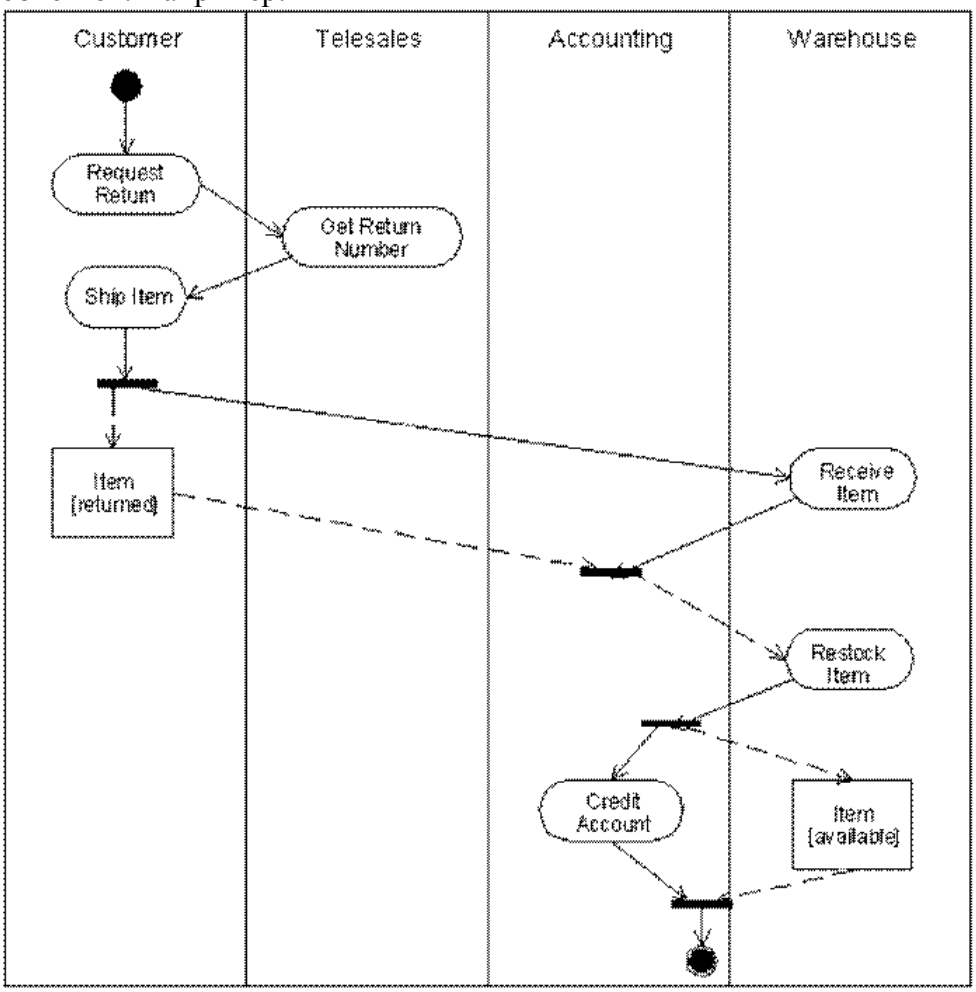
В UML 2 проведена граница между диаграммами состояний, базирующимися на формализме конечных автоматов, и диаграммами деятельности, основанными на сетях Петри. Основным элементом диаграмм деятельности является узел действия. Каждый такой узел представляет собой элементарную единицу работы (это может быть 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 пунктирной стрелкой от деятельности к изменяемому объекту или от объекта к деятельности, использующей объект. Ребра (сплошные стрелки) между узлами действий показывают потоки управления. Условный переход (*узел принятия решения*), а также *узел слияния (или узел соединения)* потоков изображается ромбом. Если необходимо показать, что две или более ветвей потока выполняются параллельно, используются «линейки синхронизации» – *узлы ветвления* и *узлы объединения* (жирные 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 генериру-ются курсоры на всех исходящих ребрах. При попадании в узел принятия решения (он имеет один вход и несколько выходов) курсор проходит дальше лишь по тому ребру, для которого выполнено сторожевое условие (вообще говоря, оно должно быть одно). При попадании в узел слияния курсор из любого входа копируется на выходе (единственном). При попадании в узел ветвления курсор дублируется на все выходы одновременно (происходит распараллеливание). Синхронизация обеспечивается узлом объединения, где происходит ожидание курсоров на всех входах и лишь затем выдается курсор на выходе. При попадании курсора в любую конечную точку 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:



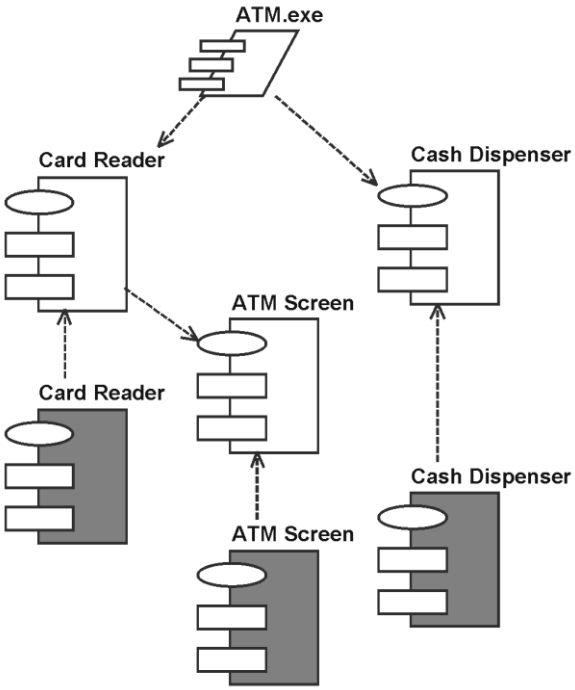
Узлы разделения и объединения могут синхронизировать потоки управления с потоками объектов. Например:



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.

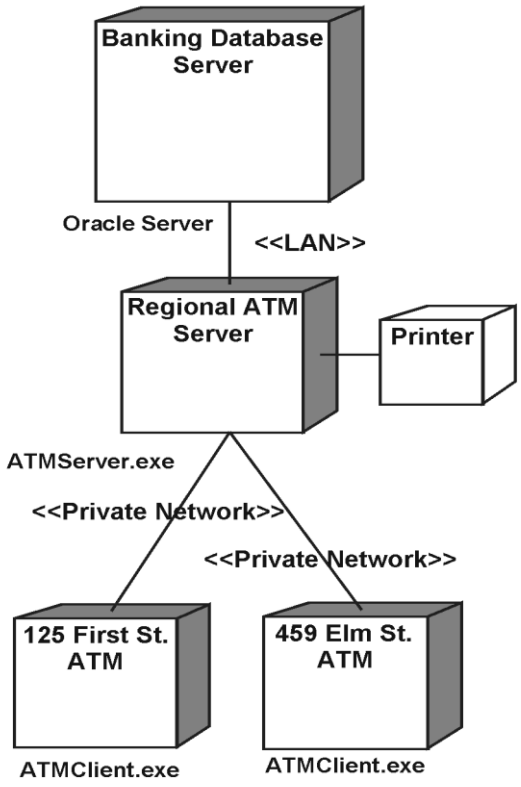


Диаграмма размещения отражает физические взаимосвязи между программными и аппаратными компонентами системы. Она является хорошим средством для того, чтобы показать размещение объектов и компонентов в распределенной системе. Ее основные элементы:

• 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, 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.

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.

Лекция 4. 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 определяется как логически завершенный 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.

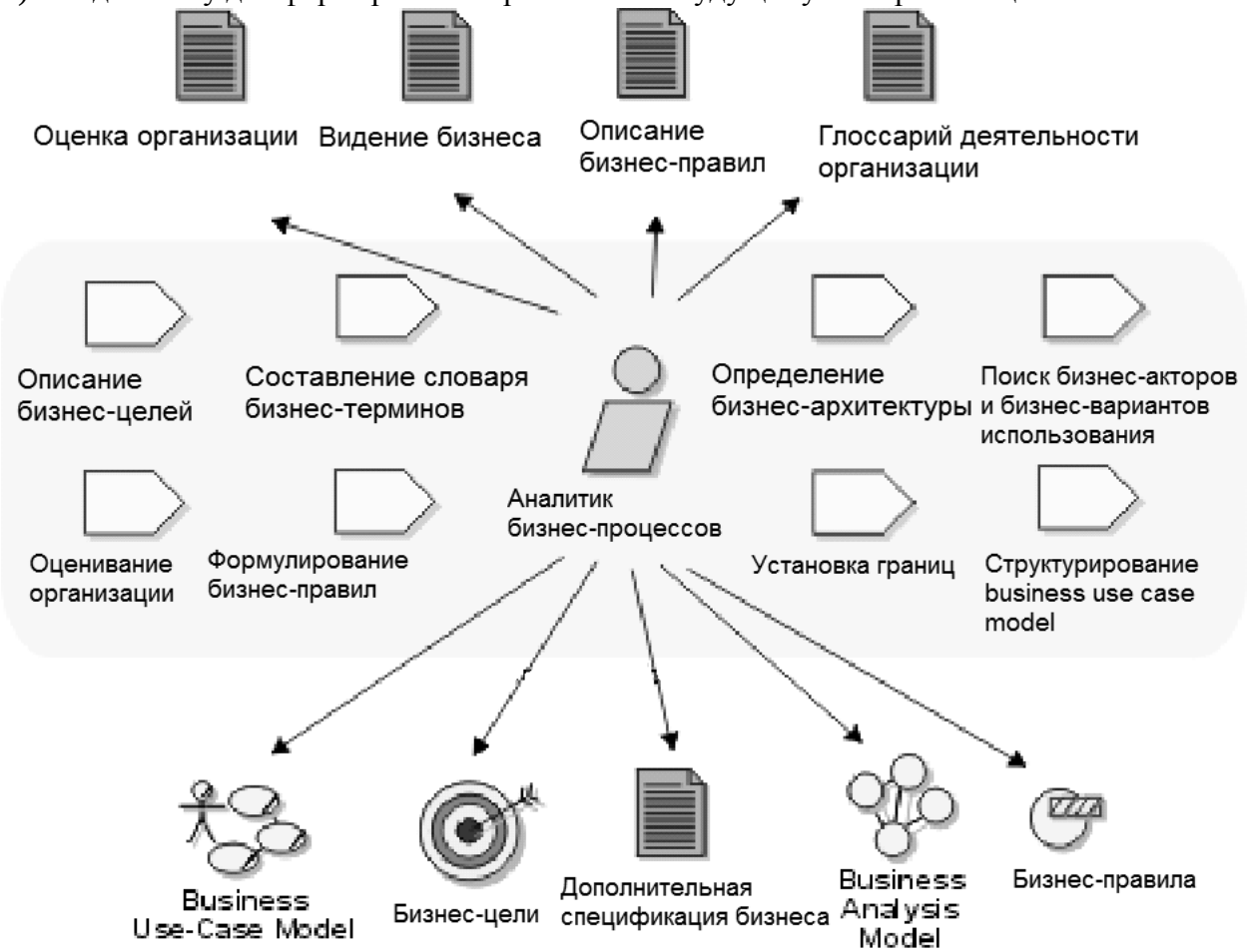


Рис. Аналитик бизнес-процессов, его деятельность и рабочие документы.

Бизнес-модель должна давать ответы на вопросы:

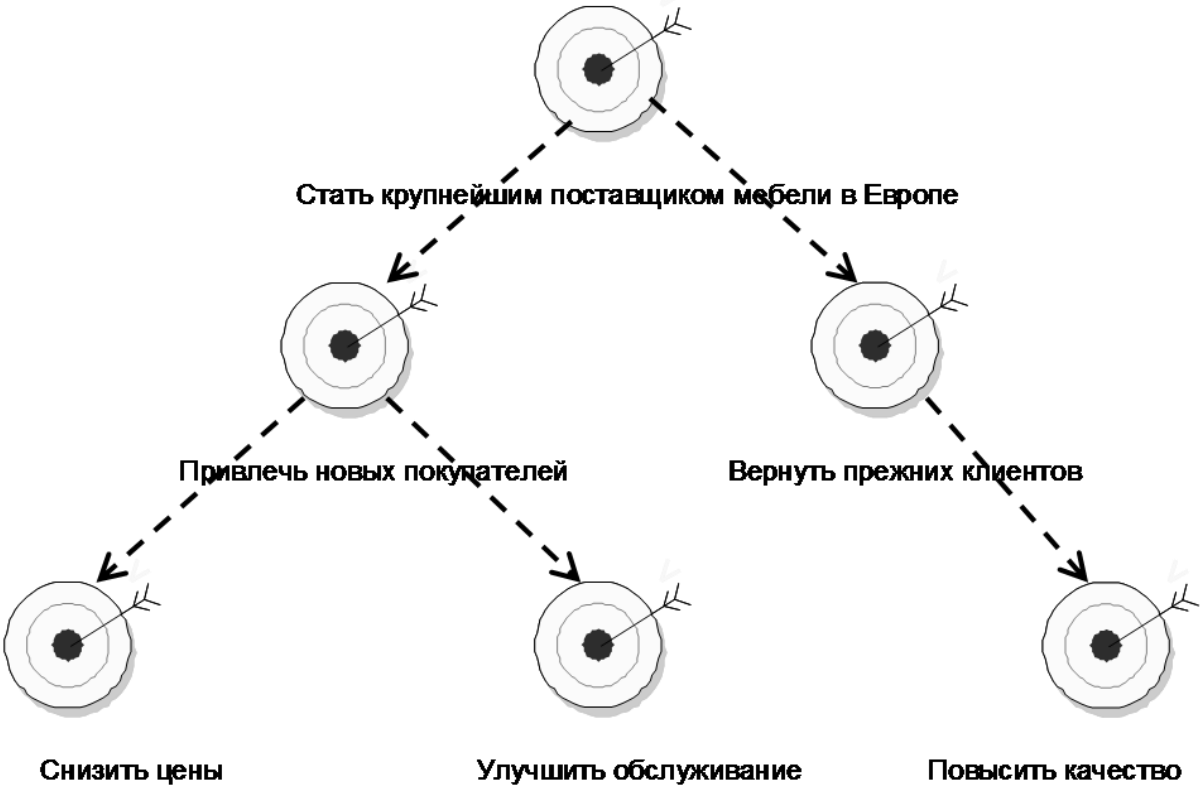
1. Какие процедуры (функции, работы) необходимо выполнить для получения заданного конечного результата?
2. В какой последовательности выполняются эти процедуры?
3. Какие механизмы контроля и управления существуют в рамках рассматриваемого бизнес-процесса?
4. Кто выполняет процедуры процесса?
5. Какие 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. Создает:

- 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.



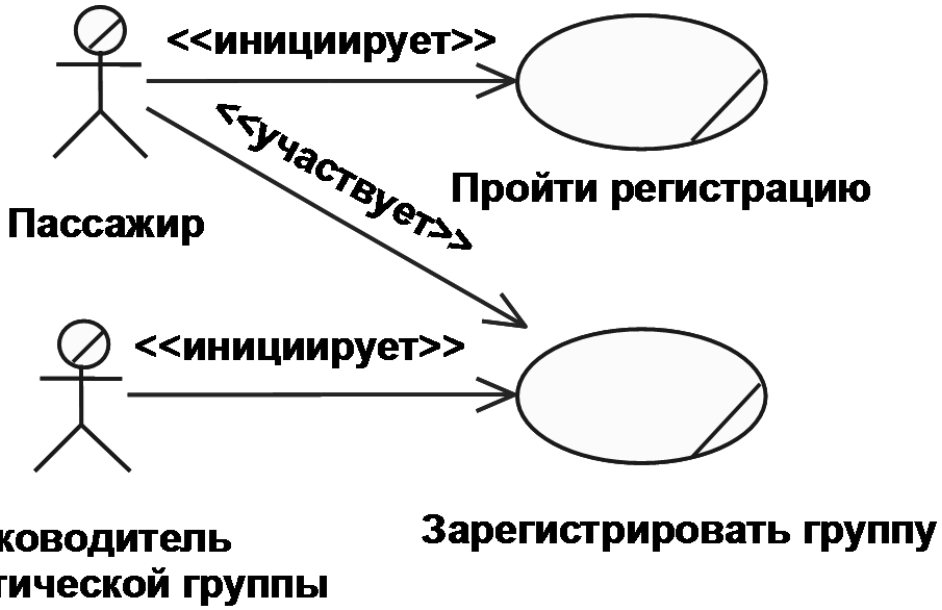
Модель бизнес-процессов (Business Use Case Model) – модель, описывающая бизнес-

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

Деловое 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). 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.

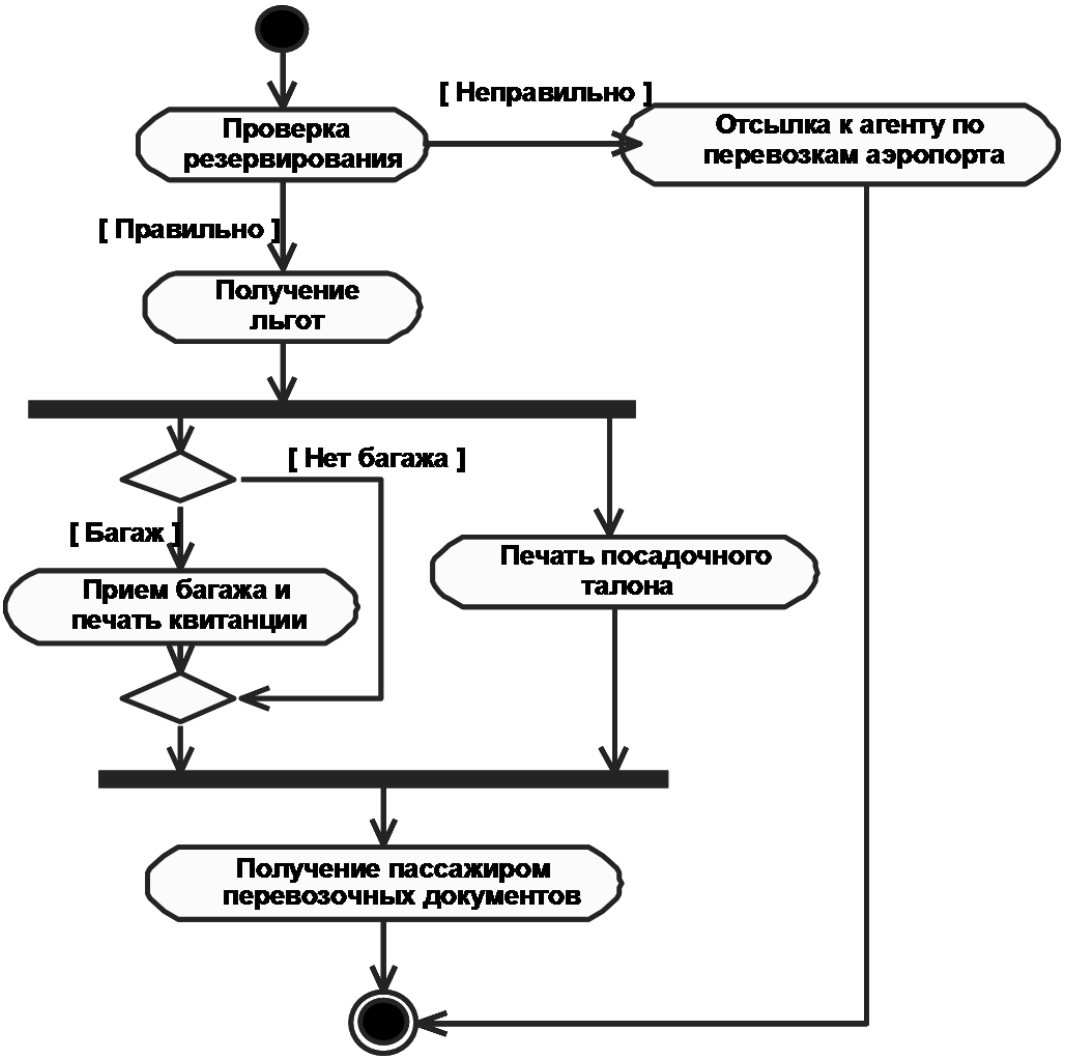
3. Регистратор подтверждает правильность билета.
4. Регистратор оформляет багаж.
5. Регистратор резервирует место для пассажира.
6. Регистратор печатает посадочный талон.
7. Регистратор выдает пассажиру посадочный талон и квитанцию на багаж.
8. Пассажир уходит от стойки регистратора.

- Альтернативные сценарии:

  1а. Билет неправильно оформлен – регистратор отсылает 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.

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).

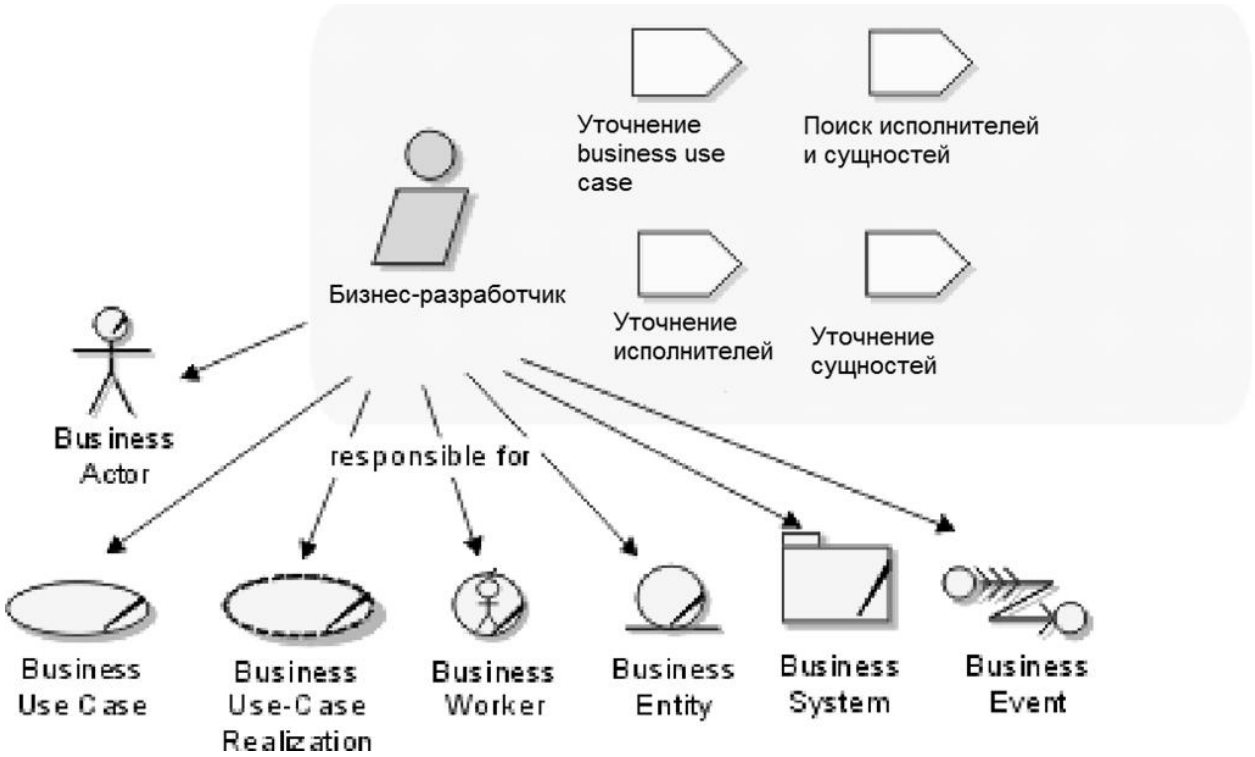
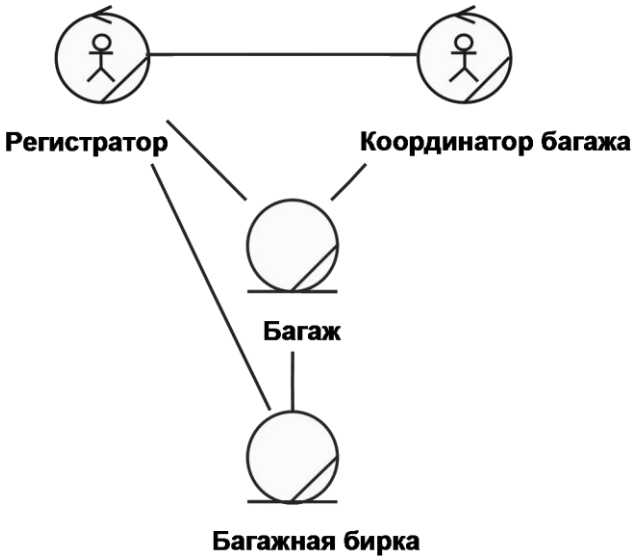


Рис. Деятельности, выполняемые бизнес-разработчиком и рабочие документы, создаваемые им.

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).

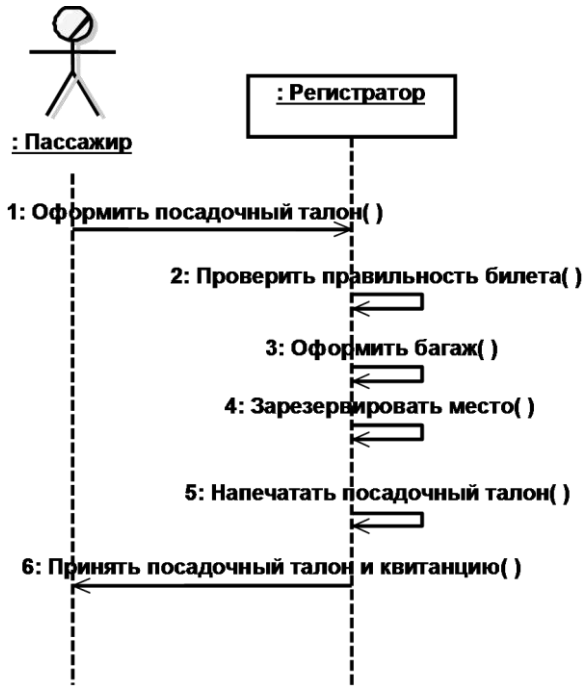
Business worker –исполнитель, действующий в рамках бизнес-системы. В отличие от делового действующего лица исполнитель работает на предприятии. Он имеет связи взаимодействия с другими исполнителями и манипулирует бизнес-сущностями, участвуя в реализациях бизнес-процессов. Представляется на диаграммах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

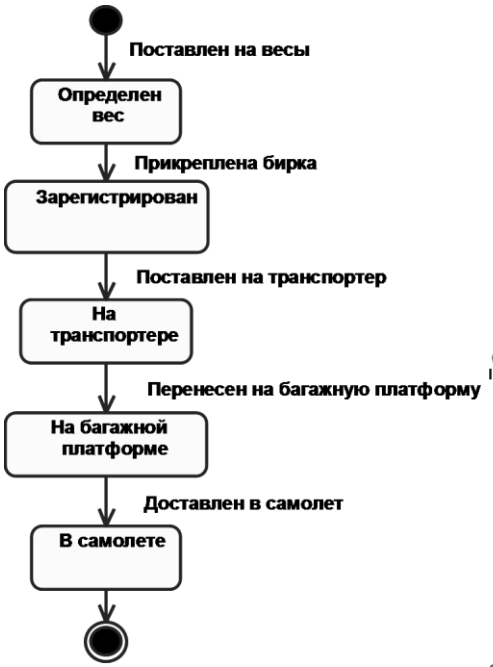
Деловая сущность (Business entity) – это ресурс (информационный, материальный, финансовый и т. д.), не инициирующий никаких взаимодействий, он может участвовать во многих реализациях различных бизнес-процессов и является предметом различных манипуляций со стороны исполнителей. На диаграммах представлен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





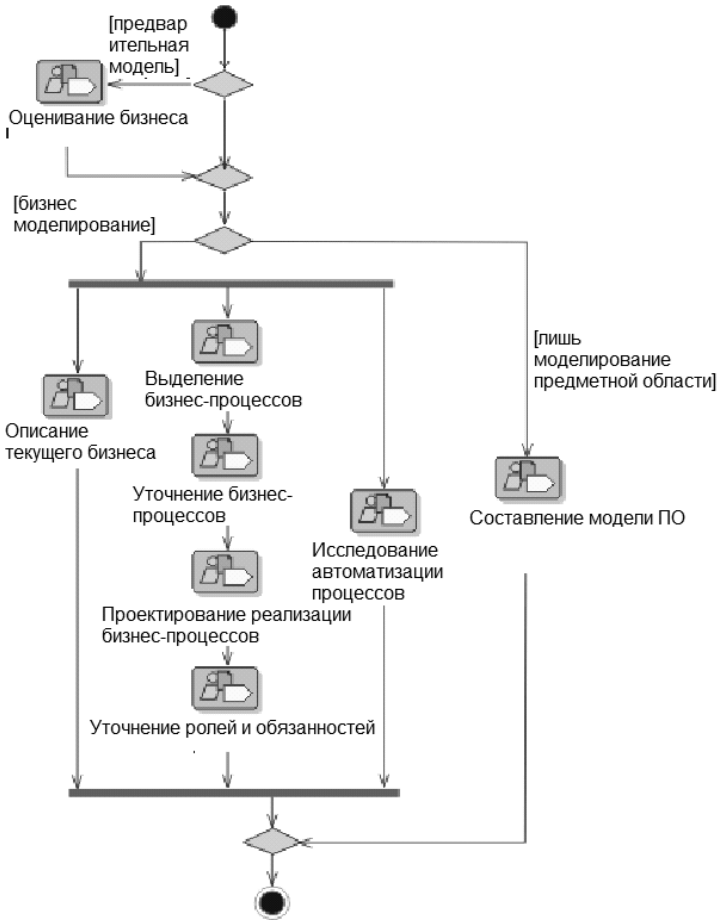
При оценивании бизнеса создаются следующие документы:

- видение бизнеса;
- оценка организации.

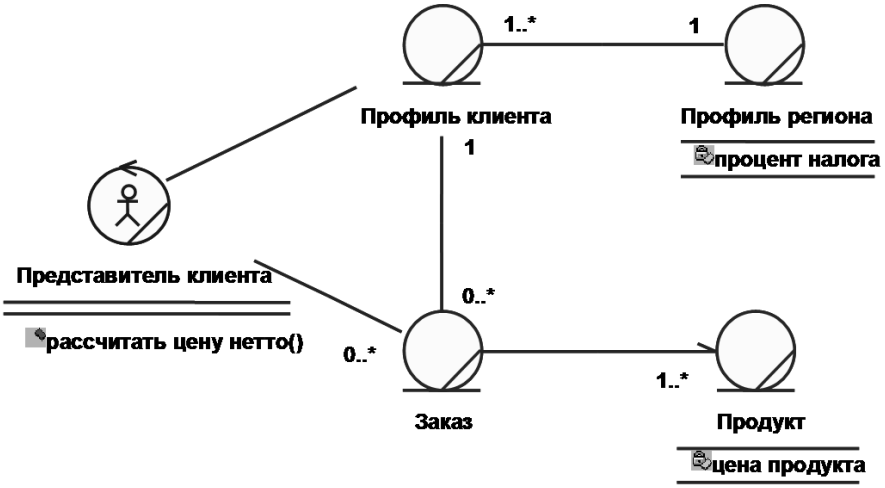
На основании этих документов принимается решение: либо моделировать только предметную область, либо осуществляется полное деловое моделирование. Исследование автоматизации процессов предпринимается, если создаваемое программное обеспечение должно автоматизировать бизнес, ранее ведущийся по старинке.

- 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 ответственность 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.).

Ход бизнес-моделирования в целом отображает следующая диаграмма деятельности:



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 (кооперация со стереотипом «business use case realization») и бизнес-система (пакет со стереотипом «business system»). Реализация бизнес-процесса описывает реализацию конкретного бизнес-процесса в рамках модели бизнес-анализа в терминах взаимодействия объектов (экземпляров исполнителей и деловых сущностей) и в терминах структурных связей между исполнителями и деловыми сущностями. Другими словами, диаграммы классов, диаграммы взаимодействия относящиеся к одному бизнес-

процессу объединяются в одну реализацию бизнес-процесса. Бизнес-система 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. Большая бизнес-система может быть разделена на части – бизнес-системы подчиненных отделов подразделения.

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 кооперaций и включают диаграммы, 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 – диаграмму классов;
- динамическое представление – диаграммы деятельности.

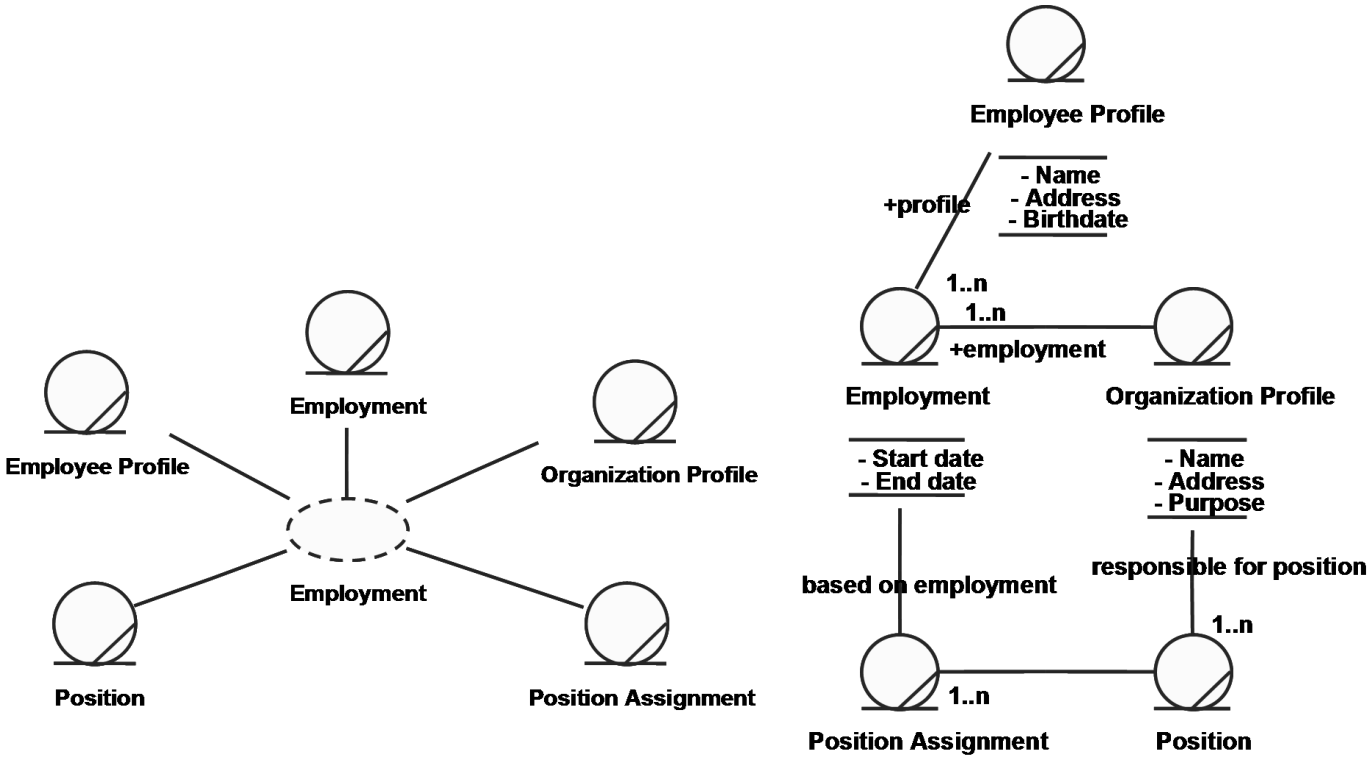
Пример. Бизнес-образец «Занятость».

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.

Совокупность участников — Статическое представление (структурные связи)

Это структурный бизнес-образец, в нем отсутствует динамическое представление.



Лекция 5. Спецификация 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.

Определение требований к ПО является составной частью процесса управления требованиями. Спецификация требований к ПО является основным документом, определяющим план разработки ПО. Все требования, определенные 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 ее атрибуты или атрибуты системного окружения. Можно выделить следующие типы нефункциональных требований:

- требования к применению определяют качество пользовательского интерфейса, документации и учебных курсов;
- требования к производительности накладывают ограничения на функциональные требования, задавая необходимую эффективность использования ресурсов, пропускную способность и время реакции;
- требования к реализации предписывают использовать определенные стандарты, языки программирования, операционную среду и др.;
- требования к надежности определяют допустимую частоту и воздействие сбоев, а также возможности восстановления;
- требования к интерфейсу 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);
- описание и анализ бизнес-процессов;
- ролевые игры;
- создание и демонстрация работающих прототипов.

Этапы работы с требованиями:

- 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.
- Проверяемость (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.

В контексте процесса управления требованиями варианты использования трактуются следующим образом:

- вариант использования фиксирует соглашение между участниками проекта относительно поведения системы;
- вариант использования описывает поведение системы при различных условиях, когда система отвечает 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 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:

- 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 (при описании 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 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.

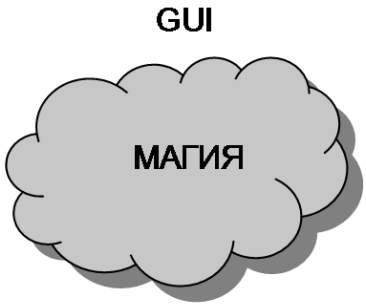
Виды альтернативных сценариев:

- 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).

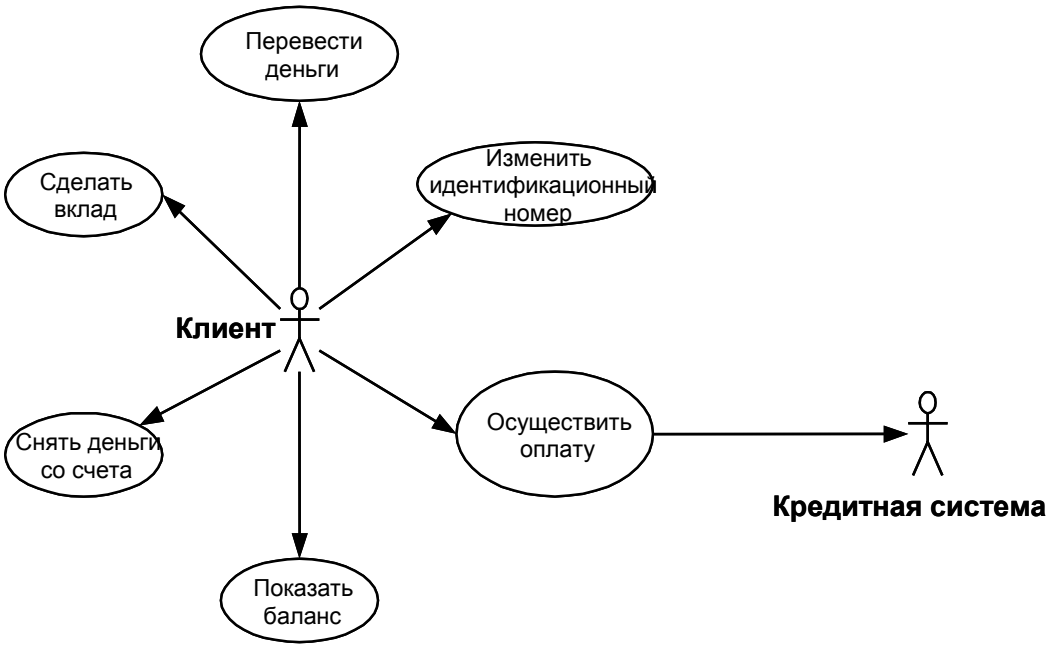
Писать варианты использования следует с позиции пользователя, т. е.:



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:

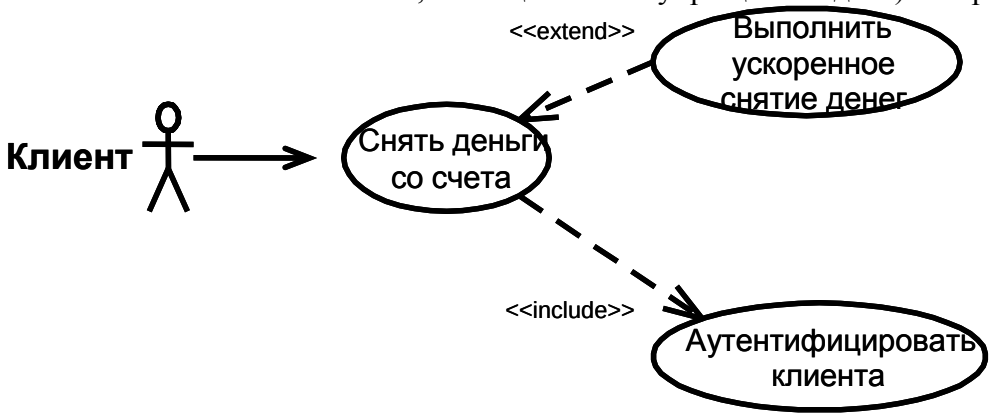


Правила составления этих диаграмм:

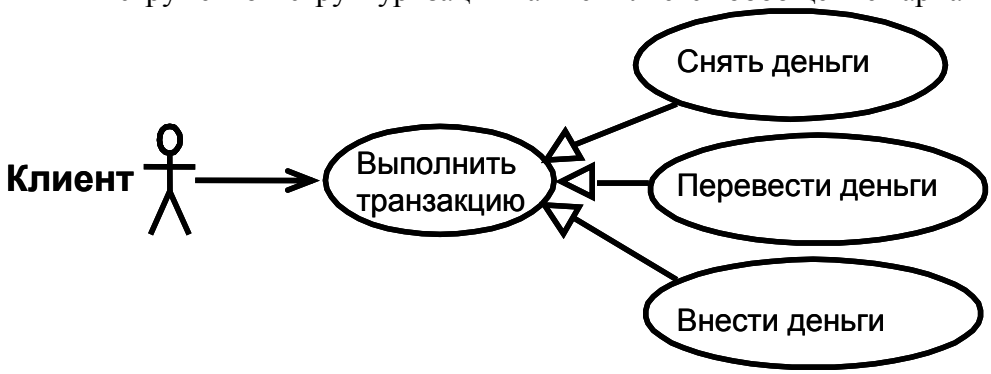
1) Каждый вариант использования должен быть инициирован действующим лицом.
2) Не моделируйте связи между действующими лицами.

3) Не соединяйте стрелкой 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 лика наследуют связи коммуникации родительского действующего лица.

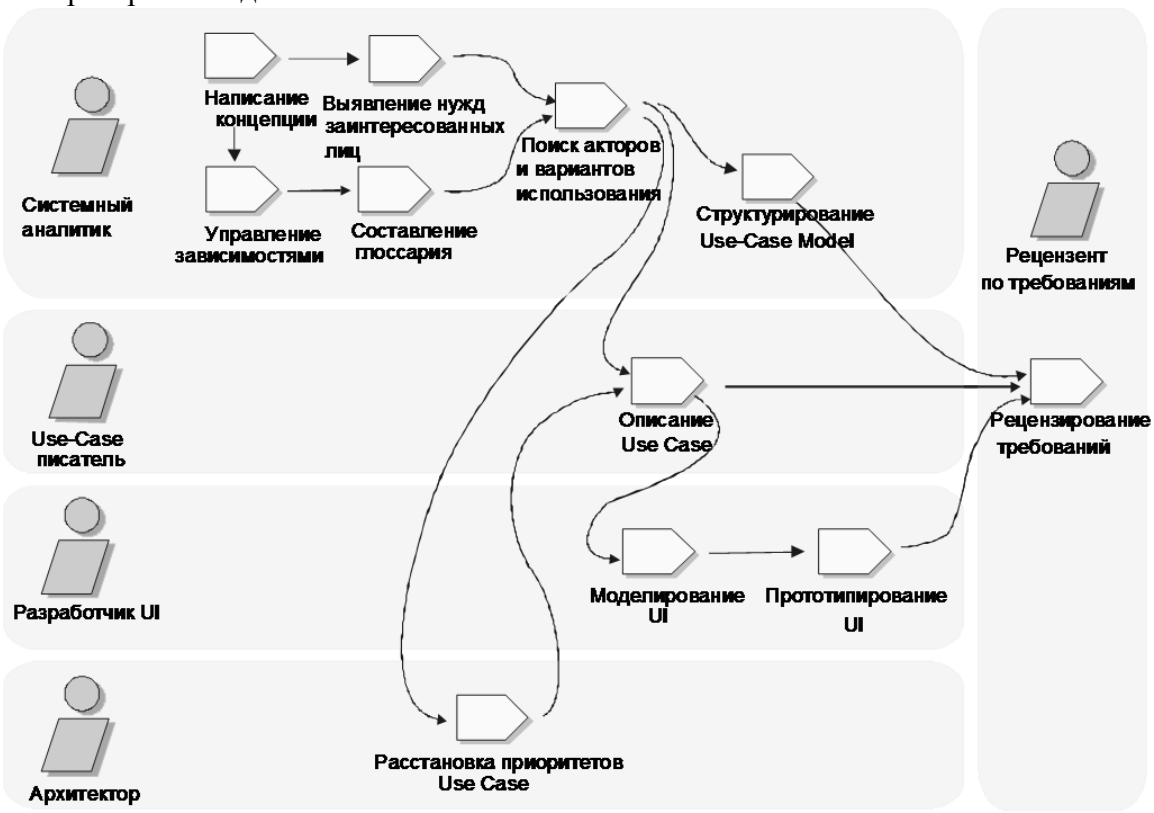
Методика моделирования вариантов использования в технологии Rational Unified Process предусматривает специальное соглашение, связанное с группировкой структурных элементов и диаграмм модели. Это соглашение включает следующие правила:

• Все действующие лица, варианты использования и диаграммы вариантов использования помещаются в пакет с именем Use Case Model.

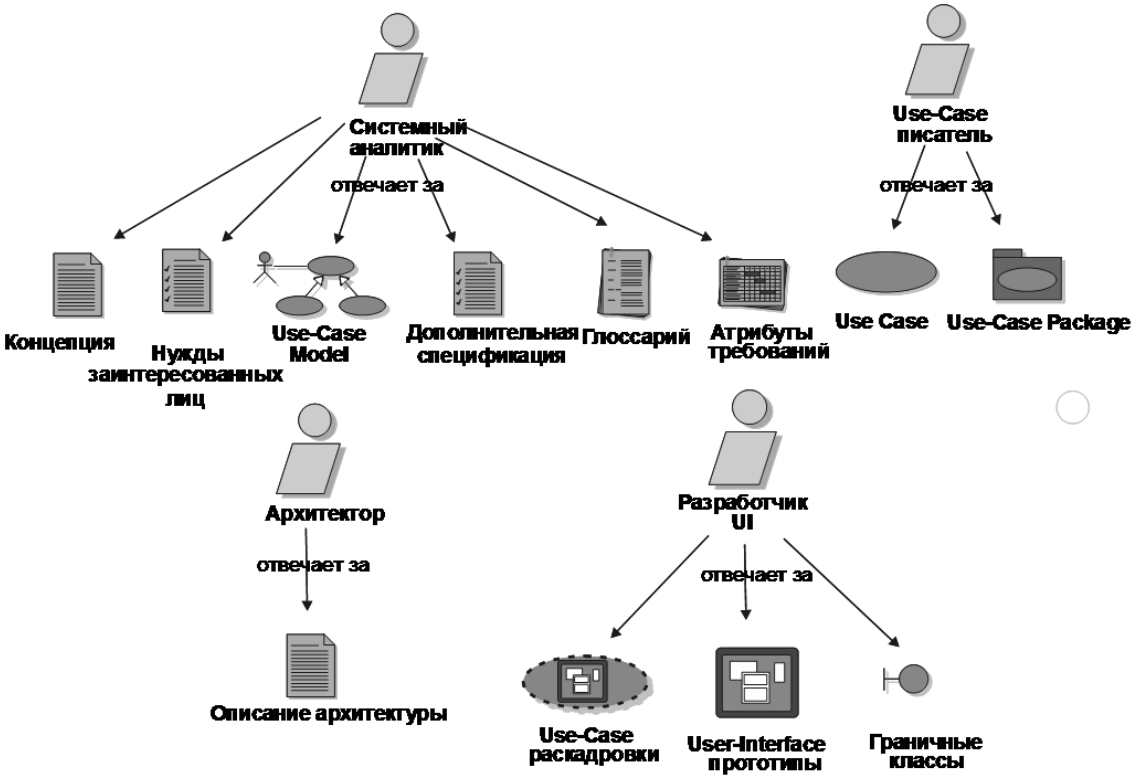
• Если моделируется сложная многофункциональная система, то совокупность всех действующих лиц и вариантов использования может разделяться на пакеты. В качестве принципов разделения могут использоваться:

- 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 деятельностей:



Наборы рабочих продуктов определения требований:

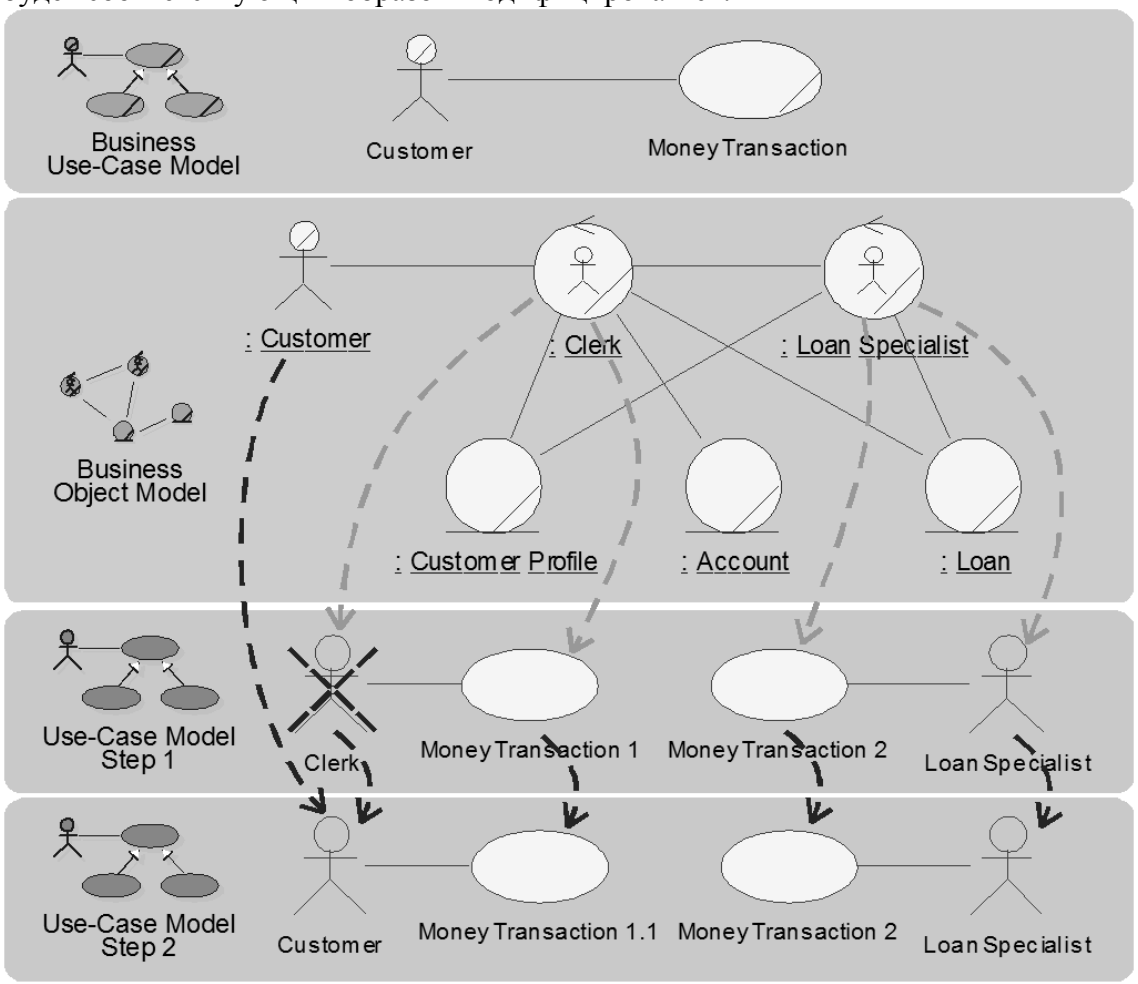


Спецификация требований в технологии Rational Unified Process не требует обязательного моделирования бизнес-процессов организации, для которых создается ПО, однако, наличие бизнес-моделей существенно упрощает построение системной модели вариантов использования. При переходе от бизнес-модели к начальной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

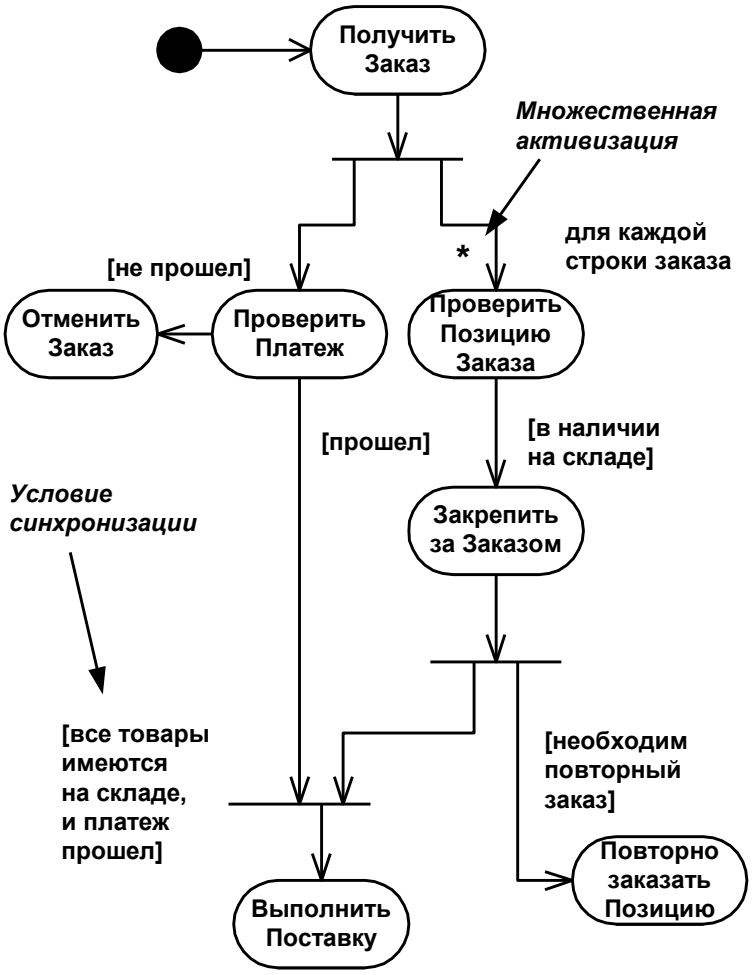
Такая начальная версия модели описывает минимальный вариант системы, пользователями которой являются только исполнители бизнес-процессов. Если в дальнейшем, в процессе развития системы, ее непосредственными пользователями будут становиться действующие лица бизнес-процессов, то модель вариантов использования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



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

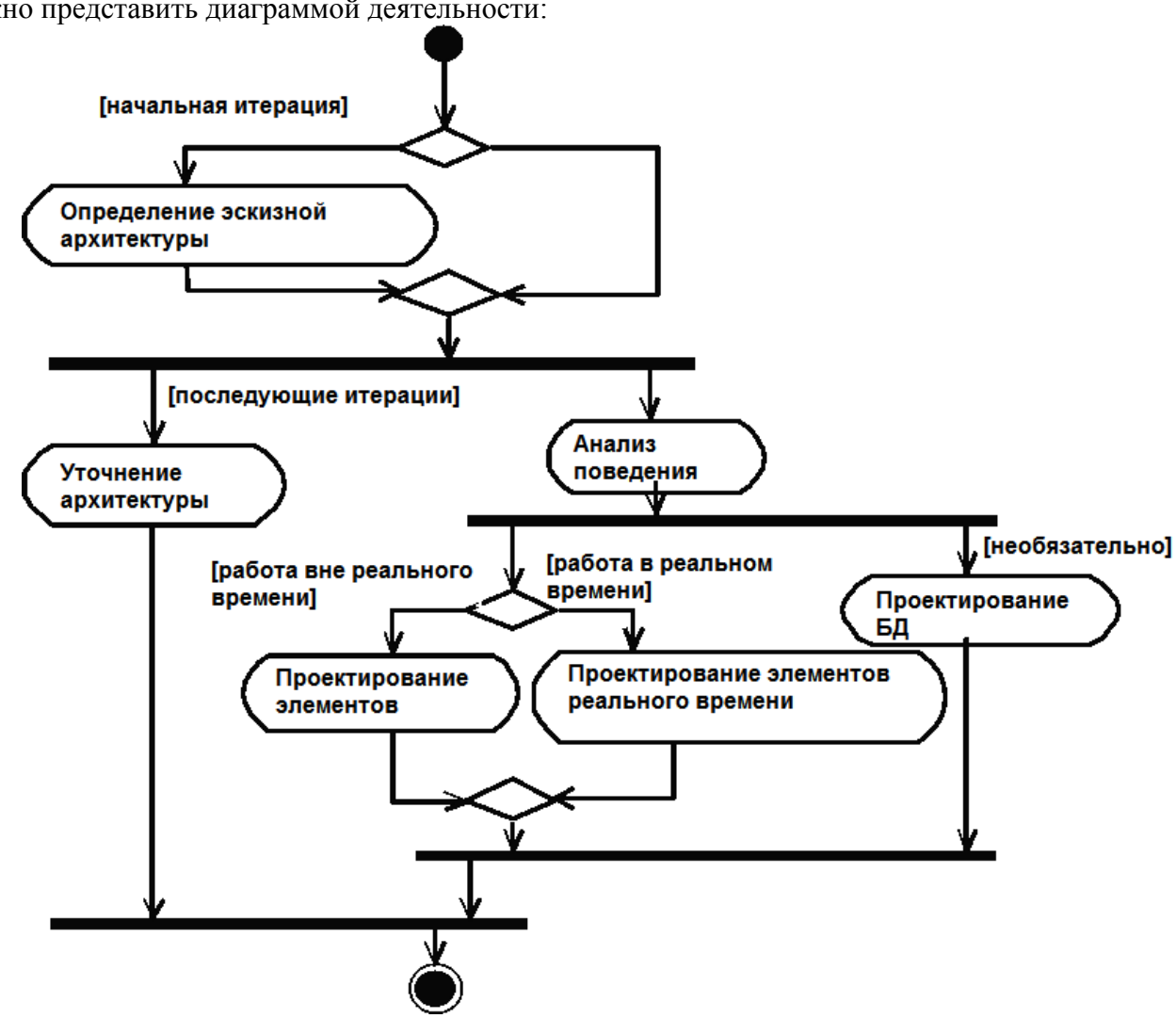
Лекция 6. Анализ и проектирование программного обеспечения. Анализ ПО

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

- трансформация требований в системный проект;
- создание стабильной 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).

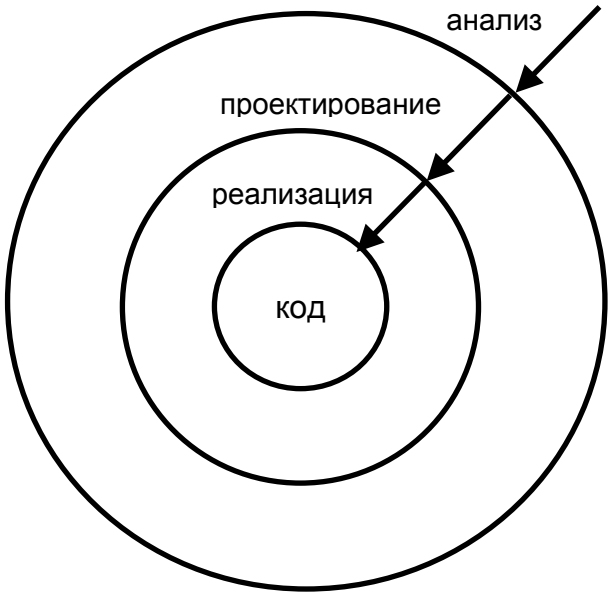
Анализ и проектирование отличаются подходом к создаваемой системе. Анализ характеризуется тем, что:

- 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 низком – 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, 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:

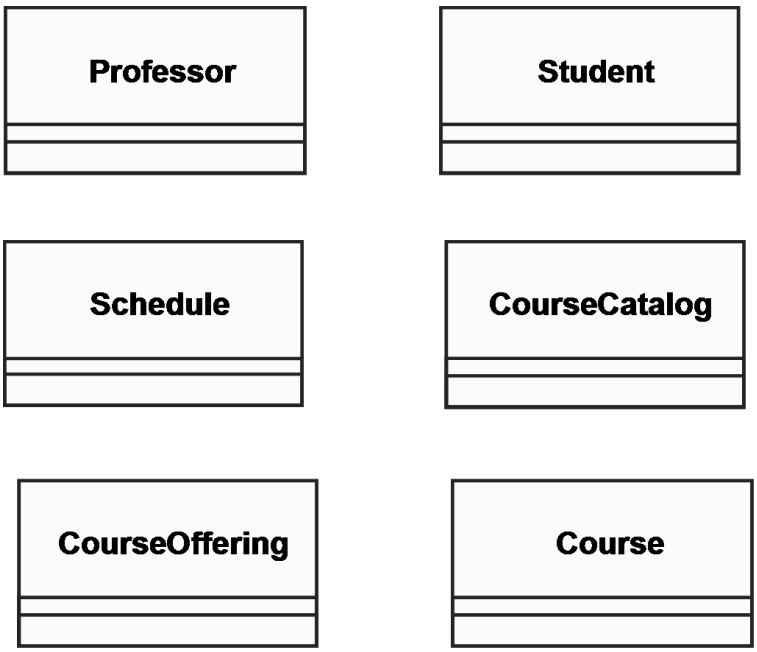
- Механизмы анализа: обеспечивают взаимодействие классов и компонентов предметной области, без деталей реализации.

- 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) на основе описания предметной области и спецификации требований к системе (в частности, глоссария проекта). Как правило создается диаграмма классов, на которую помещаются все ключевые абстракции. Пример (регистрация на курсы):



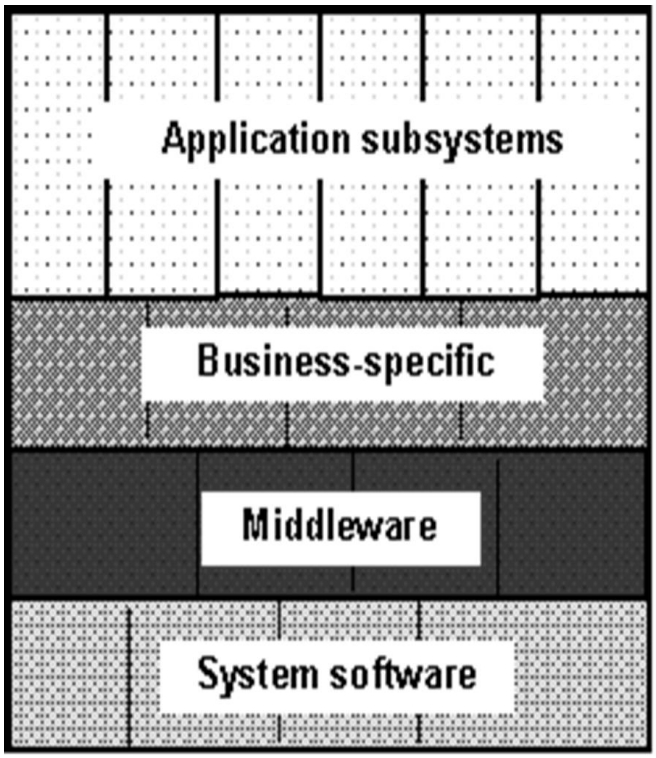
Архитектурные уровни образуют иерархию уровней представления любой крупной системы. Почти в любой системе можно выделить следующие уровни:

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

При формировании архитектурных уровней определяется 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.

Layers:
Прикладной уровень – реализация функциональности вариантов использования.
Бизнес-уровень – набор компонентов, специфичных для конкретной предметной области.
Middleware – платформо-независимые сервисы (GUI, ORB, …)
System software – ПО для вычислительной и сетевой инфраструктуры (ОС, сетевые протоколы и др.)

Образец «Каналы и фильтры» решает следующую проблему: *Нужно обеспечить преобразование непрерывных потоков данных. При этом преобразования инкрементны и следующее может быть начато до окончания предыдущего. Имеется, возможно, несколько входов и несколько выходов. В дальнейшем возможно добавление дополнительных преобразований.*

Предлагается представить систему как последовательность частей (фильтров), реализующих отдельные этапы обработки данных, и каналов, связывающих входы и выходы соседних фильтров и обеспечивающих непрерывное поступление данных:

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):*

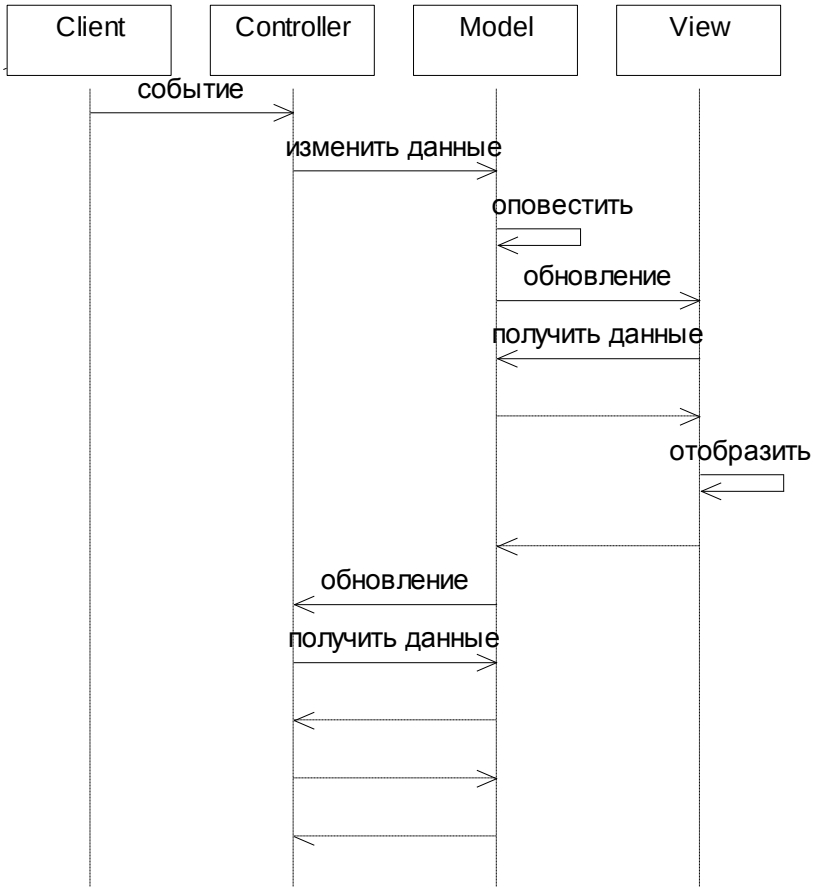
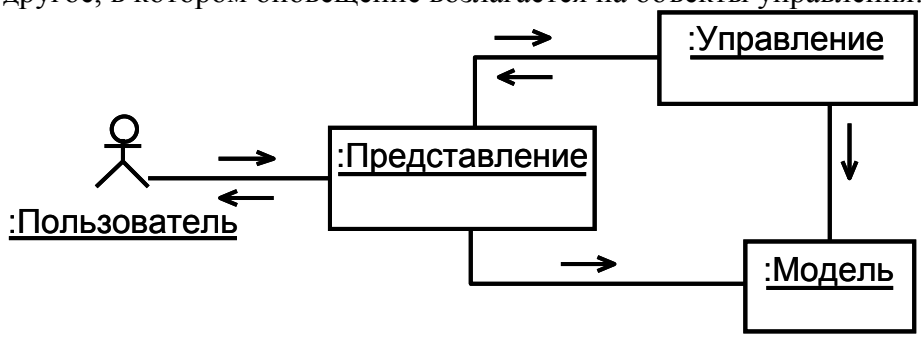


Рис. Один из вариантов организации взаимодействия, предписываемый образцом MVC.

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:

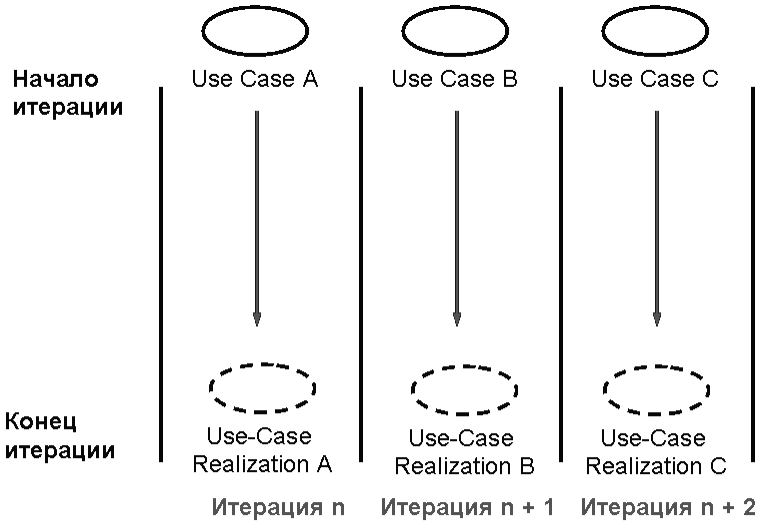


В такой схеме достигается независимость модели от представления, обеспечивающая больше возможностей для повторного использования.

Анализ вариантов использования выполняется проектировщиками 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 одни 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.

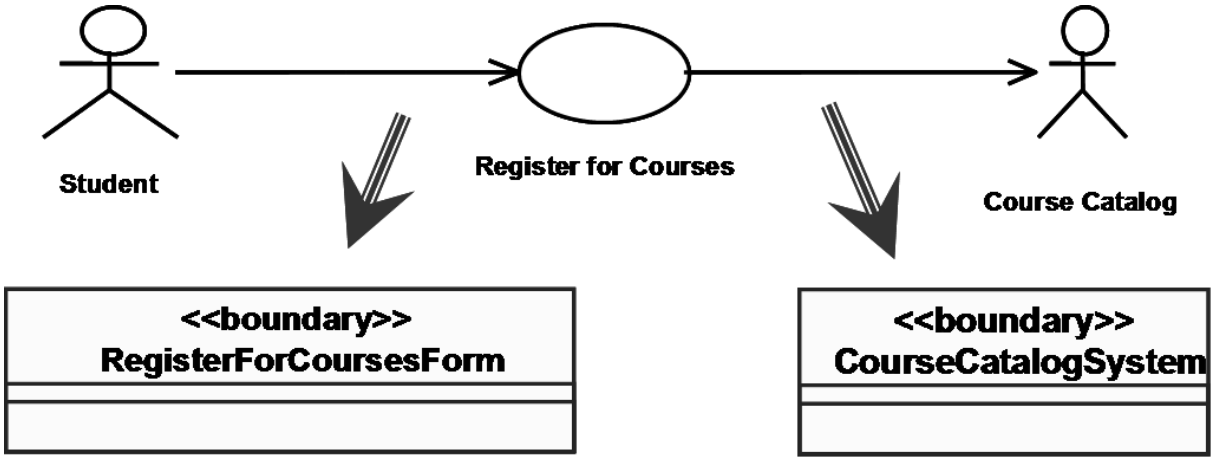
Классы анализа отражают функциональные требования к системе и моделируют объекты предметной области. Совокупность классов анализа представляет собой начальную концептуальную модель системы. Эта модель проста и позволяет сосредоточиться на реализации функциональных требований, не отвлекаясь на детали 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 искомых классов-сущностей).
- Анализ описаний вариантов использования.

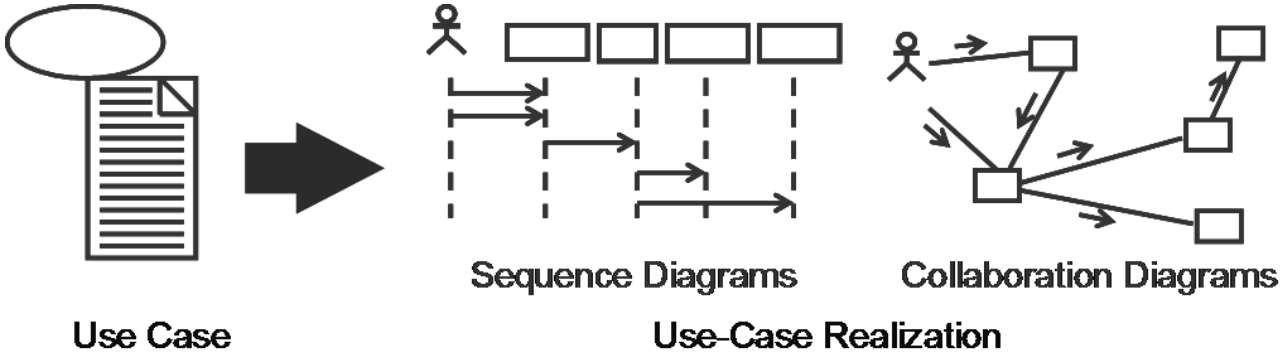
Правило *выделения управляющих классов*: для каждого варианта использования создается ответственный за его реализацию класс управления.

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). Виды обязанностей классов:

- Знание:
  - наличие информации о данных или вычисляемых величинах;
  - наличие информации о связанных объектах.
- Действие:
  - выполнение некоторых действий самим объектом;
  - инициация действий других объектов;

- координация действий других объектов.

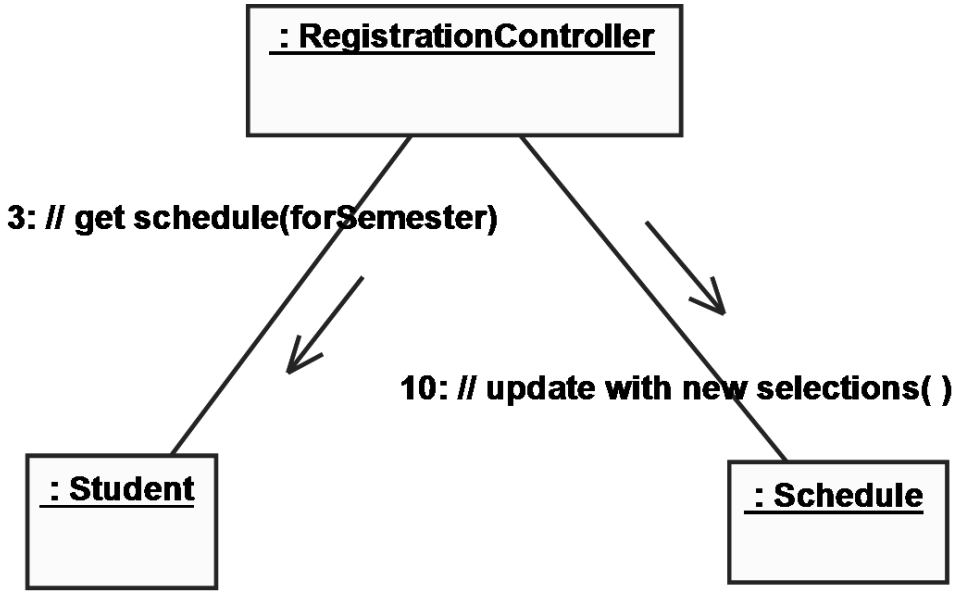


В процессе анализа конкретного варианта использования в первую очередь строится диаграмма последовательности, описывающая основной поток событий и его подчиненные потоки. Для каждого альтернативного потока событий строится отдельная диаграмма последовательности.

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 B обязанность создавать экземпляры класса A, если выполняется одно из следующих условий:

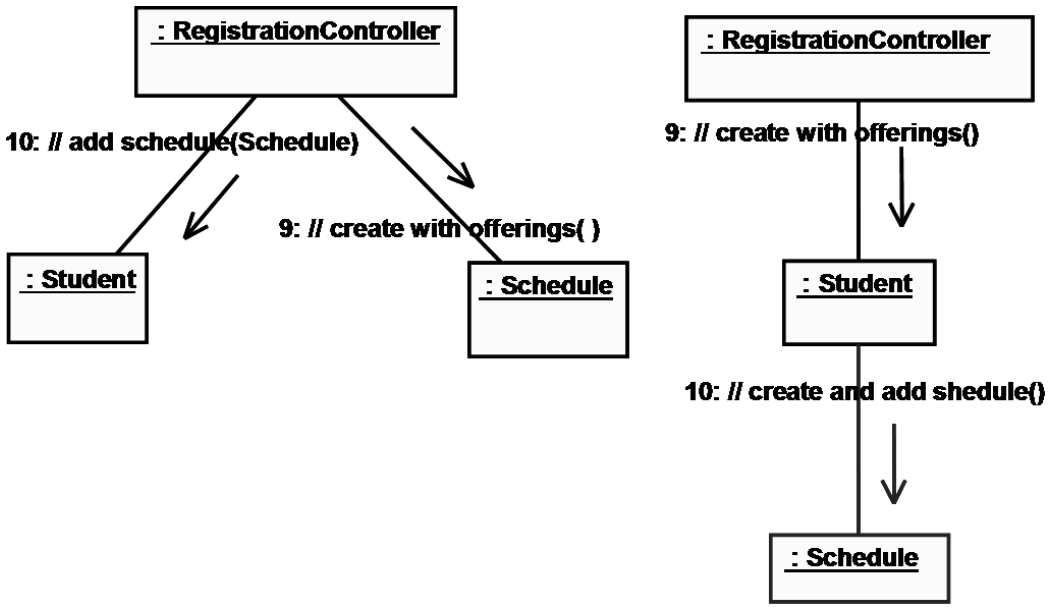
• класс B агрегирует, содержит или активно использует объекты класса A;

• класс B обладает данными инициализации, которые будут передаваться объектам класса A при их создании (т.е. класс B является информационным экспертом).

Класс B при этом определяется как создатель (creator) объектов класса A.

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

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", наилучшим решением является вариант справа (новый объект класса Schedule создается классом Student, а не RegistrationController, поскольку именно Student удовлетворяет первому из перечисленных выше условий).

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" (слабое зацепление или 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 слабое зацепление. Зацепление (coupling) – это мера, определяющая насколько жестко один элемент связан с другими элементами, или каким количеством данных о других элементах он обладает. Элемент со низким зацеплением зависит от небольшого числа других элементов. Класс с высоким зацеплением 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 обеспечено более низкое зацепление).

Следствия: Образец Low Coupling поддерживает независимость классов, что, в свою очередь, повышает возможности повторного использования и обеспечивает более высокую эффективность приложения. 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 прочность или сильная связность). 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 прочность. В терминах объектно-ориентированного проектирования 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 другими классами.

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).

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*.

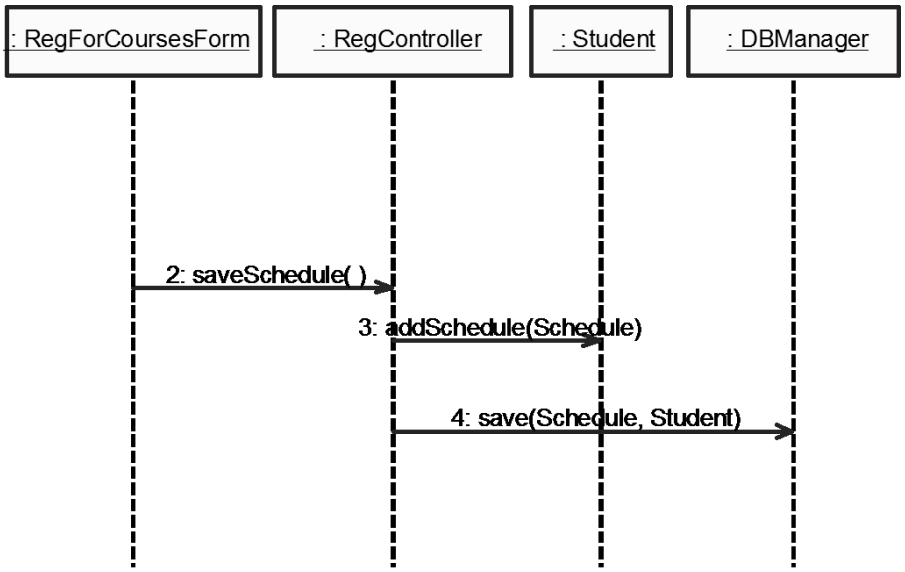
Образец «Сценарий транзакции».

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».

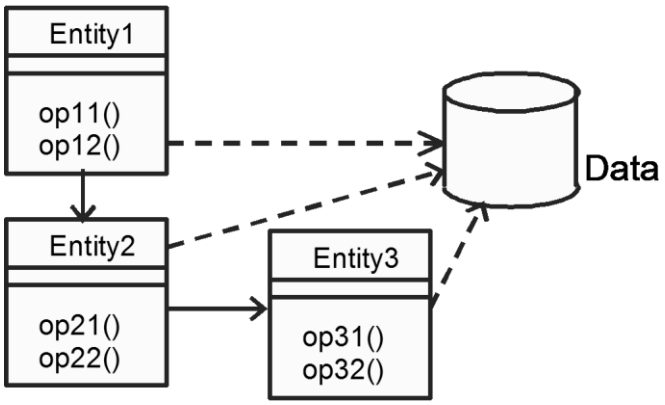
Пример:



Образец «Модель предметной области»

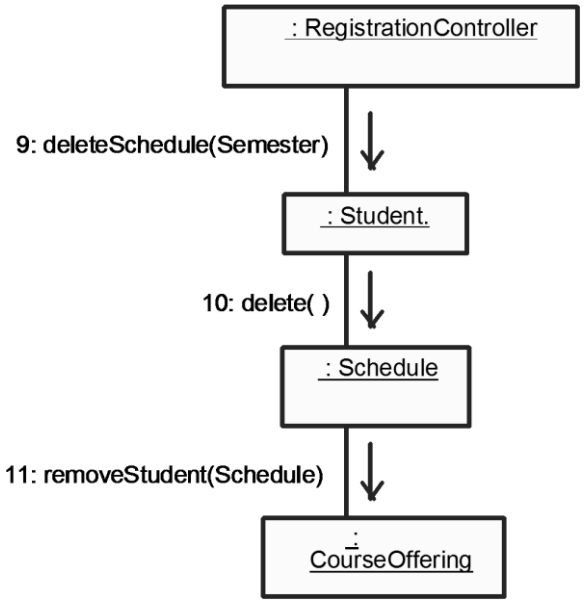
Аннотация: Обязанности по обработке запросов распределены по сети объектов-сущностей. В приложении создается слой объектов, описывающих структурные и поведенческие аспекты предметной области. Поведение сочетается с данными и реализуется в сущностях. Управляющие объекты довольствуются ролью посредников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

Эскиз:

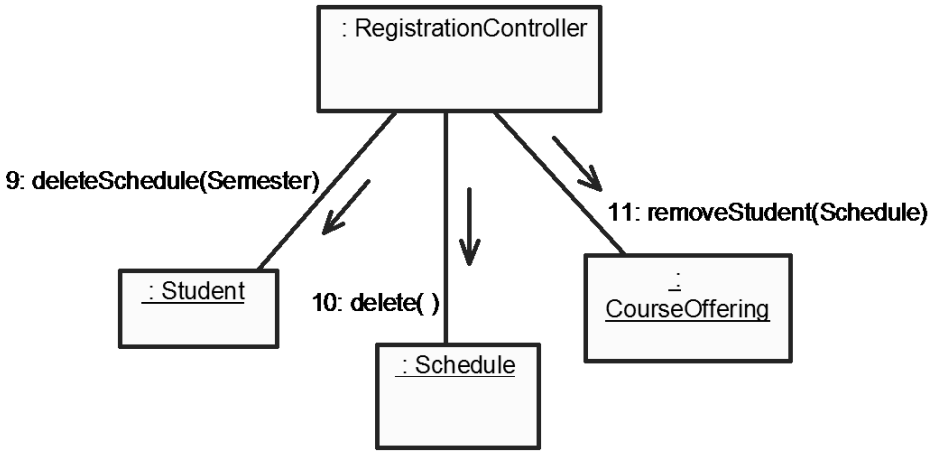


Назначение: Подходит для приложений с запутанной бизнес-логикой. Недостаток: создание модели предметной области более трудоемко, чем реализация сценария транзакции. В простых случаях нужно применять сценарий транзакции.

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



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



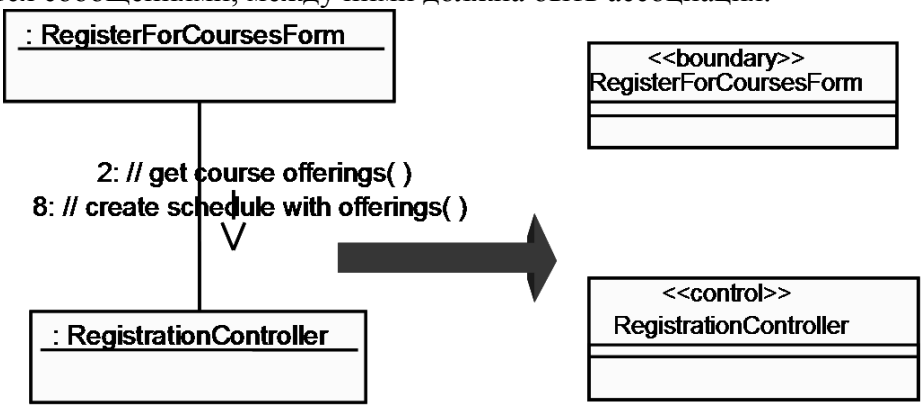
Набор обязанностей классов, полученный в результате их распределения, должен быть проанализирован на предмет выявления и устранения следующих проблем:

• 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 «persistency», не задумываясь как именно будет реализовано сохранение их экземпляров в базе данных. Пример:

Класс анализа	Механизм(ы) анализа
Student	Persistency, Security
Schedule	Persistency, Security
CourseOffering	Persistency, Legacy Interface
Course	Persistency, 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?

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

Лекция 7. Проектирование программного обеспечения

Проектирование системы является поиском ответа на вопрос *как* следует сделать то, что следует сделать.

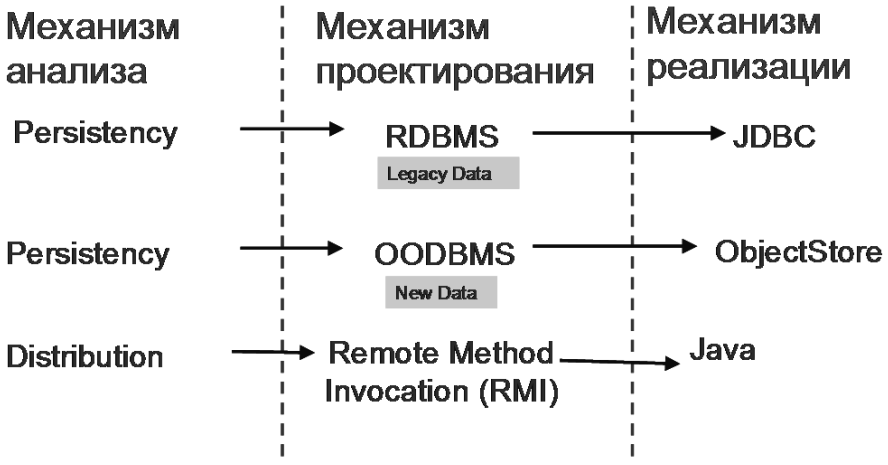
Этапы проектирования:

1. Проектирование архитектуры системы.
   1.1. Идентификация архитектурных решений и механизмов проектирования;
   1.2. Анализ взаимодействий между классами анализа, выявление проектных классов, подсистем и интерфейсов;
   1.3. Формирование архитектурных уровней;
   1.4. Проектирование структуры потоков управления;
   1.5. Проектирование конфигурации системы.
2. Проектирование элементов системы.
   2.1. Уточнение реализаций вариантов использования.
   2.2. Проектирование подсистем.
   2.3. Проектирование классов.
   2.4. Проектирование баз данных.

Проектирование архитектуры системы выполняется архитектором системы.

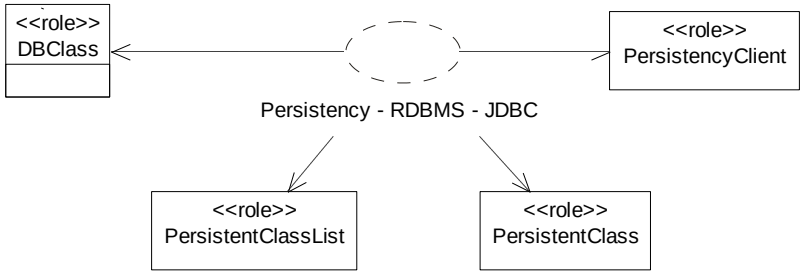
В процессе проектирования определяются детали реализации архитектурных механизмов, обозначенных в процессе анализа. Например, уточняется способ хранения данных, реализация дублирования для повышения надежности системы и т. п. Поскольку практически все механизмы – это некоторые типовые решения (образцы), они документируются в проекте (модели) с помощью кооперации со стереотипом <<mechanism>>, при этом структурная часть механизма описывается с помощью диаграмм классов, а поведение – с помощью диаграмм взаимодействия. Ранее мы встречались с использованием кооперации для моделирования реализаций вариантов использования. В них также объединялись структурные модели (диаграммы классов) с моделями поведения (диаграммами взаимодействия).

Проектные механизмы являются переходным этапом от механизмов анализа к механизмам реализации. Механизм реализации – решение, имеющее конкретного поставщика, проектный механизм – каркас, максимально приближенный к реализации, имеющий конкретное наполнение, чем отличается от механизма анализа, являющегося лишь своеобразной меткой.



В качестве примера рассмотрим механизм Persistency – хранение экземпляров устойчивых классов в БД. Предположим, что в проекте системы регистрации в качестве языка программирования используется Java. Поскольку существующая система каталога курсов функционирует на основе реляционной СУБД, механизмом проектирования, обеспечивающим доступ к этой внешней базе данных, будет RDBMS (Relational Database Management System), реализовать который можно решением от Sun – JDBC (Java Database Connectivity).

Рис. Диаграмма классов, отображающая механизм JDBC и роли в нем



Стереотип <<role>> используется для элементов модели, являющихся метками-заполнителями (placeholders), – своего рода гнезд, в которые при проектировании будут подставлены реальные элементы, созданные разработчиком системы. Роли являются своего рода параметрами механизма, при подстановке на их место конкретных классов определяется экземпляр механизма, используемый при проектировании системы.

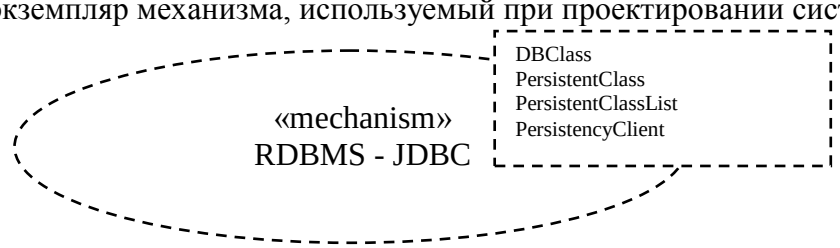


Рис. Изображение механизма JDBC в виде параметризованной кооперации.

Классы-участники механизма JDBC:

- DBClass – роль, которая отвечает за чтение и запись данных, реализует все услуги по хранению устойчивых объектов в реляционной БД.
- DriverManager, Connection, Statement, ResultSet – библиотечные классы, которые отвечают за реализацию запроса к БД (выполнение оператора SQL) – пакет java.sql.
- PersistentClass – роль, представляющая любой устойчивый класс.
- PersistentClassList – роль, представляющая список объектов, которые являются результатом запроса к БД –DBClass.read().
- PersistencyClient – роль, представляющая любой клиентский класс.

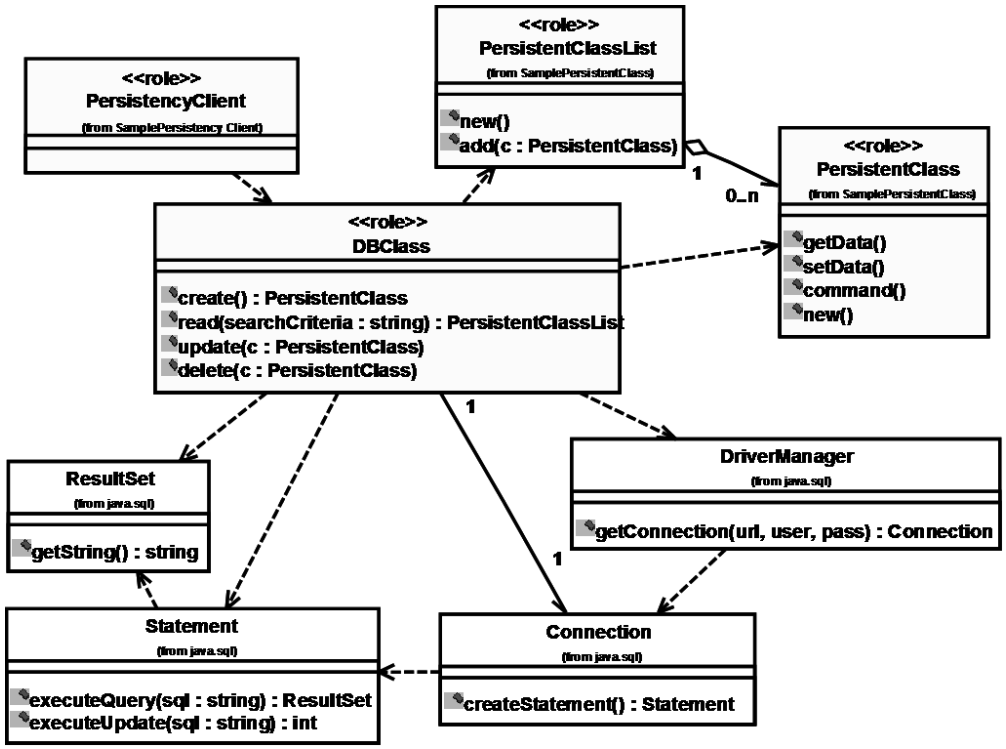
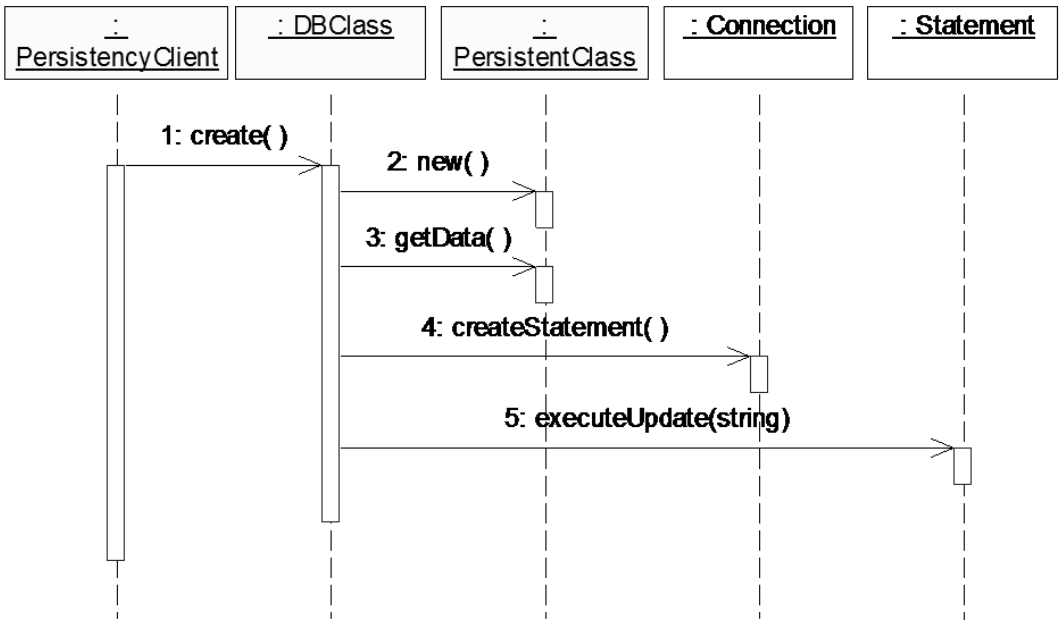


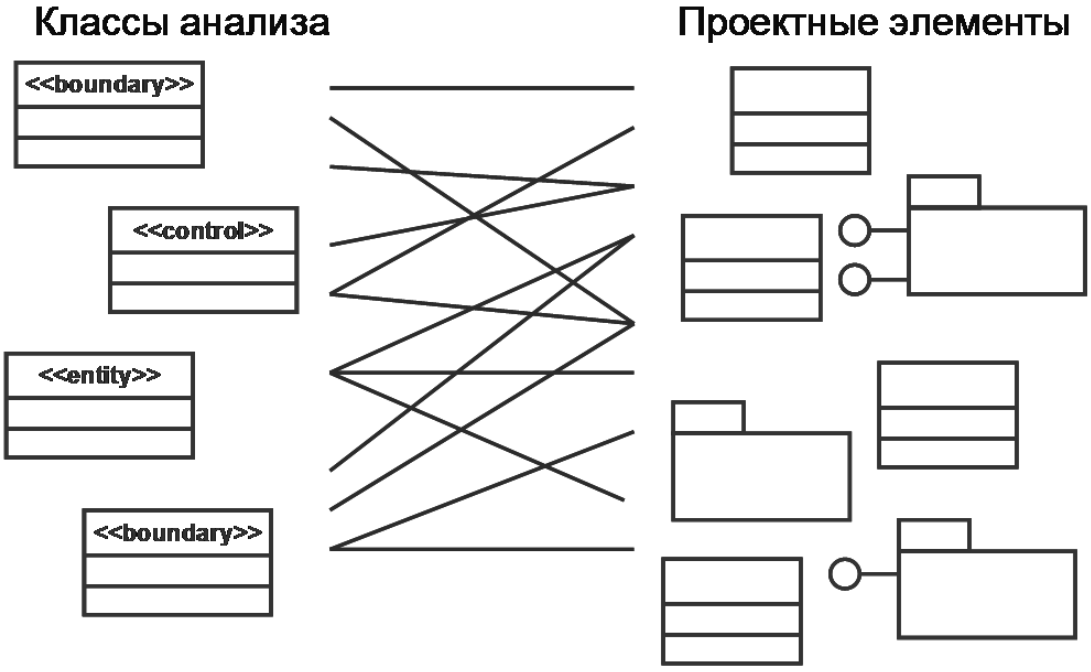
Рис. Диаграмма классов, отображающая структурные связи классов-участников JDBC

Взаимодействие экземпляров классов, в рамках механизма описывается диаграммами взаимодействия, например:



Из диаграммы видно, что для создания новых данных (нового экземпляра устойчивого класса) экземпляр клиентского класса PersistencyClient запрашивает DBClass. DBClass создает новый экземпляр PersistentClass, запрашивает начальные значения его атрибутов (записанные конструктором). Затем DBClass создает новый оператор SQL, используя операцию createStatement() класса Connection. Этот запрос должен добавить новую запись в таблицу. В результате выполнения этого оператора SQL данные нового экземпляра устойчивого класса помещаются в БД.

Следующим этапом является выявление подсистем и интерфейсов. Декомпозиция системы на подсистемы реализует принцип модульности. Первым действием архитектора при выявлении подсистем является преобразование классов анализа в проектные классы.



По каждому классу анализа принимается одно из двух решений:

- класс анализа отображается в проектный класс, если он простой или представляет единственную логическую абстракцию;
- сложный класс анализа может быть разбит на несколько классов, преобразован в пакет или в подсистему.

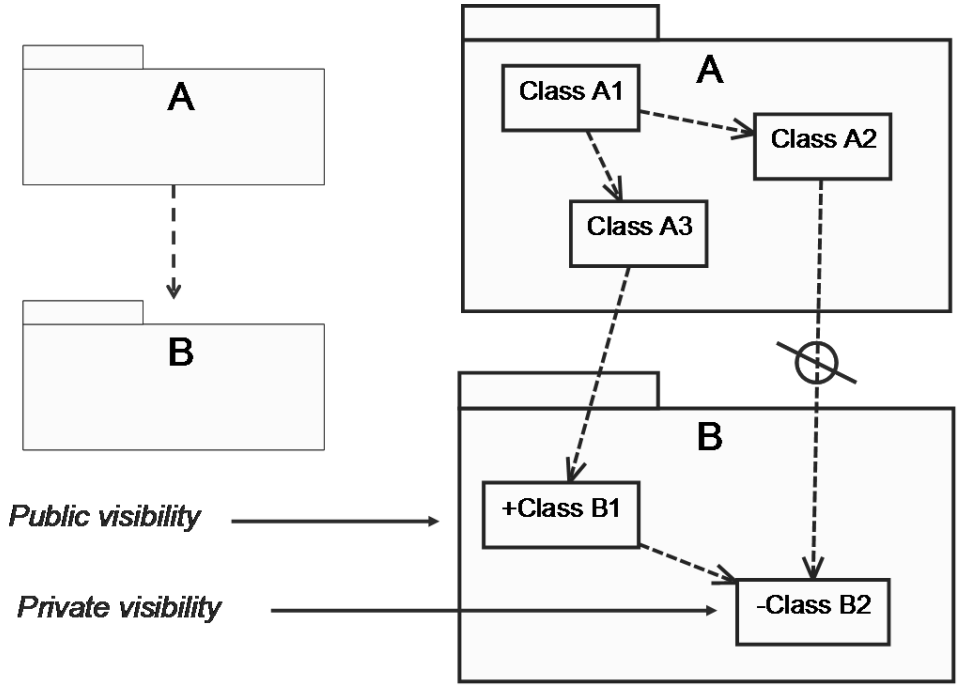
Совокупности классов объединяются в пакеты и подсистемы. При объединении классов в пакеты учитывается, что:

- Пакеты – это единицы управления конфигурацией, поэтому члены пакета должны быть одинаково стабильны.
- Пакеты – средство распределения ресурсов между командами разработчиков.
- Разные пакеты могут соответствовать разным типам пользователей.
- Как пакет можно оформить повторно используемый код, встроенный в систему.

Например, если пользовательский интерфейс нестабилен, имеет смысл объединить все классы, его реализующие, в отдельный пакет. Если UI не будет подвергаться существенным изменениям, можно объединить в отдельные пакеты классы, взаимодействующие при реализации разных вариантов использования.

Распределяя классы по пакетам желательно добиться ситуации, когда через границы пакетов проходит значительно меньшее количество связей, чем находится внутри пакетов.

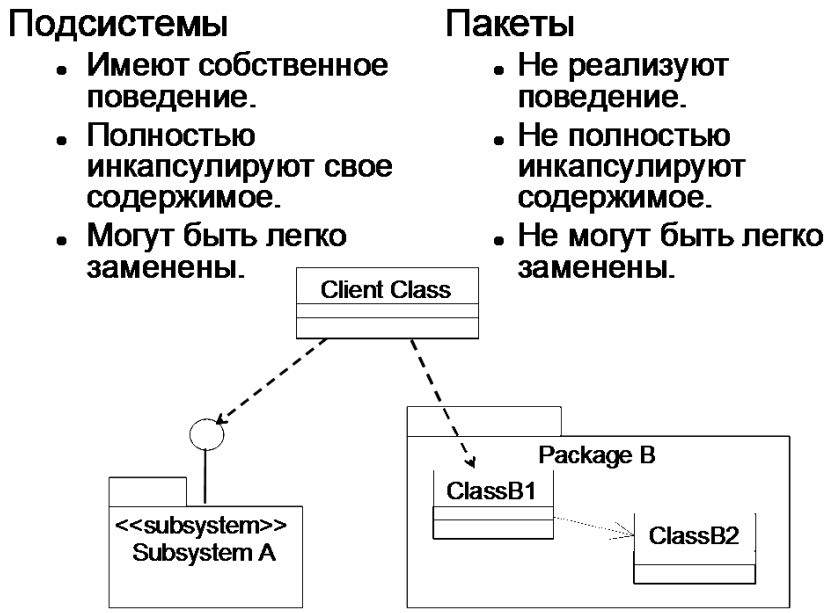
После выделения пакетов устанавливаются зависимости между ними и видимость членов пакета. К закрытым членам пакета доступ извне запрещен. Это позволяет скрыть внутреннее устройство пакета.



Несколько классов могут быть объединены в подсистему если:

- классы имеют функциональную связь (участвуют в реализации варианта использования и взаимодействуют только друг с другом);
- совокупность классов реализует функциональность, которая может быть удалена из системы или заменена на альтернативную;
- классы сильно связаны;

- классы размещены на одном узле вычислительной сети.

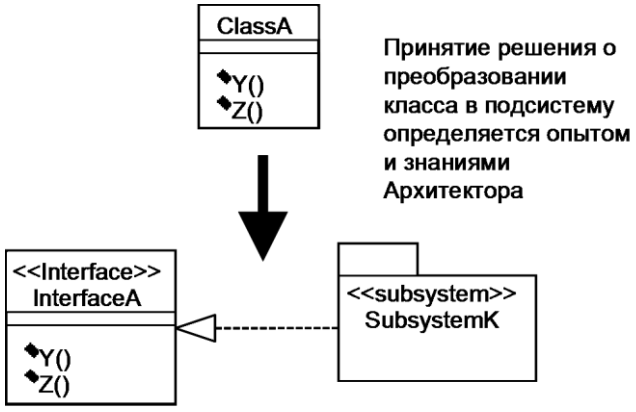


Заметим, что связь клиентского класса с подсистемой более гибкая, чем с пакетом, в том смысле, что изменения внутри подсистемы не затрагивают клиентский класс. Во втором случае изменения из пакета могут распространиться на клиента вдоль зависимости. Обратной стороной гибкости является такой неприятный факт – интерфейс должен быть стабильным. Любое изменение в интерфейсе затронет реализующую его подсистему (вдоль связи реализации), а также клиентский класс, которому требуется данный интерфейс (вдоль зависимости). Очень желательно сразу правильно описать интерфейсы!

Примеры возможных подсистем: подсистема безопасности, защиты данных, архивирования; подсистема пользовательского интерфейса или интерфейса с внешними системами; коммуникационное ПО, доступ к базам данных.

При создании подсистем в модели выполняются следующие преобразования:

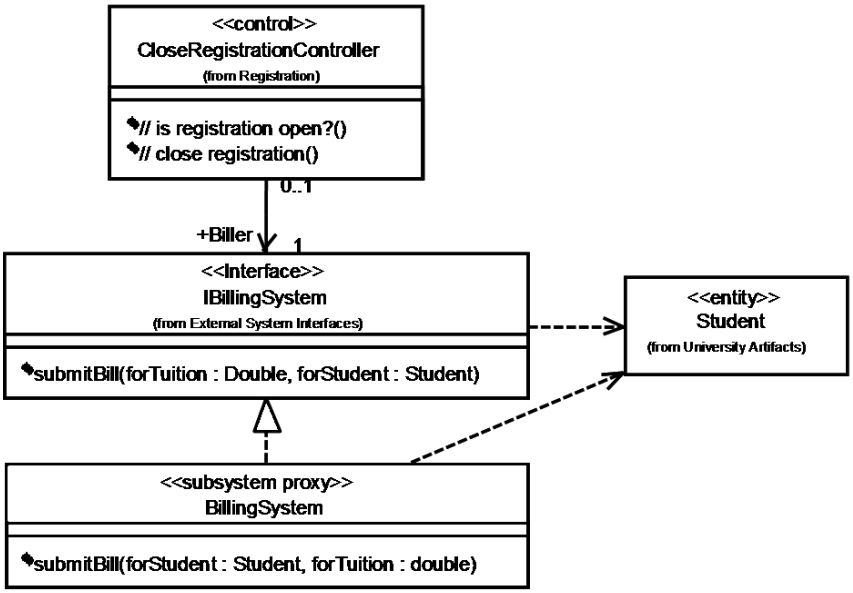
- объединяемые классы помещаются в специальный пакет с именем подсистемы и стереотипом <<subsystem>>;
- спецификации операций классов, образующих подсистему, выносятся в интерфейс подсистемы – класс со стереотипом <<interface>>;
- характер использования операций интерфейса и порядок их выполнения документируется с помощью диаграмм взаимодействия, которые вместе с диаграммой классов подсистемы объединяются в кооперацию с именем интерфейса и стереотипом <<interface realization>>;
- в подсистеме создается класс-посредник со стереотипом <<subsystem proxy>>, управляющий реализацией операций интерфейса.



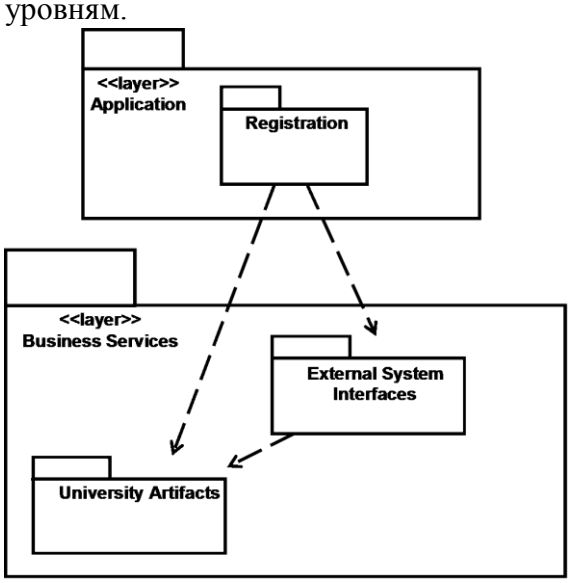
Все интерфейсы подсистем должны быть полностью определены в процессе проектирования архитектуры, поскольку они будут служить в качестве точек синхронизации при параллельной разработке системы. Описание интерфейса включает:

- Имя интерфейса: короткое (одно-два слова), отражающее его роль в системе.
- Описание интерфейса: должно отражать его ответственности (размер – небольшой абзац).
- Описание операций: имя, отражающее результат операции, ключевые алгоритмы, возвращаемое значение, параметры с типами.
- Документирование интерфейса: характер использования операций и порядок их выполнения (показывается с помощью диаграмм последовательности), тестовые планы и сценарии и т.д. Вся эта информация объединяется в специальный пакет.

В качестве примера (для системы регистрации) приведем подсистему BillingSystem, которая создана вместо граничного класса BillingSystem. Взаимодействие с ней осуществляется через объект-посредник класса BillingSystem, реализующего интерфейс iBillingSystem. На диаграмме показаны внешние связи подсистемы.



Следующим этапом является *формирование архитектурных уровней*. В ходе анализа было принято предварительное решение о выделении архитектурных уровней. В ходе проектирования все проектные элементы системы должны быть распределены по данным уровням. С точки зрения модели это означает распределение проектных классов, пакетов и подсистем по пакетам со стереотипом «layer», соответствующим архитектурным уровням.



Пример формирования архитектурных уровней (система регистрации на курсы):

- выделены 2 архитектурных уровня (пакета со стереотипом <<layer>>);
- на уровень Application (Приложение) помещен пакет Registration, куда включены граничные и управляющие классы;
- граничные классы BillingSystem и CourseCatalogSystem преобразо-ваны в подсистемы уровня бизнес-логики Business Services;
- в слой Business Services, помимо подсистем, включены еще два пакета: пакет External System Interfaces включает интерфейсы

подсистем (классы со стереотипом <<Interface>>), а пакет University Artifacts – все классы-сущности.

Следующий этап – *проектирование структуры потоков управления*. Оно выполняется при наличии в системе параллельных процессов (параллелизма). Цель проектирования – выявление существующих в системе процессов, характера их взаимодействия, создания, уничтожения и отображения в среду реализации. Требование параллелизма возникает в следующих случаях:

• необходимо распределение обработки между различными процессорами или узлами;
• система управляется потоком событий (event-driven system);
• вычисления в системе обладают высокой интенсивностью;
• в системе одновременно работает много пользователей.

Например, система регистрации курсов обладает свойством параллелизма, поскольку она должна допускать одновременную работу многих пользователей (студентов, регистраторов и профессоров), каждый из которых порождает в системе отдельный процесс.

Процесс – это ресурсоемкий поток управления, который может выполняться параллельно с другими потоками. Он выполняется в независимом адресном пространстве и в случае высокой сложности может разделяться на два или более потоков.

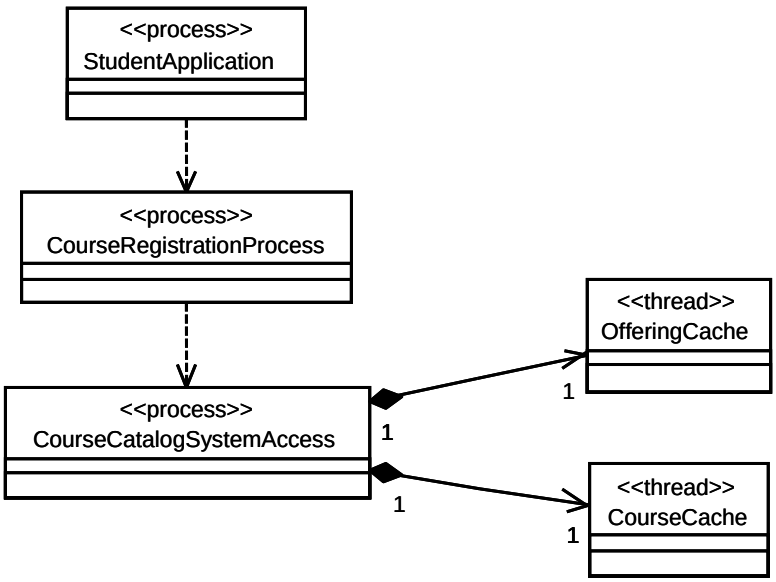
Поток (нить) – это облегченный поток управления, который может выполняться параллельно с другими потоками в рамках одного и того же процесса в общем адресном пространстве.

Необходимость создания потоков в системе регистрации курсов диктуется следующими требованиями:

• если курс окажется заполненным в то время, когда студент формирует свой учебный график, включающий данный курс, то он должен быть извещен об этом (необходим независимый процесс, управляющий доступом к информации конкретных курсов);
• существующая база данных каталога курсов не обеспечивает требуемую производительность (необходим процесс промежуточной обработки – подкачки данных).

Реализация процессов и потоков обеспечивается средствами операционной системы.

Для моделирования структуры потоков управления используются так называемые активные классы – классы со стереотипами <<process>> и <<thread>>. Активный класс владеет собственным процессом или потоком и может инициировать управляющие воздействия. Связи между процессами моделируются как зависимости. Потоки могут существовать только внутри процессов, поэтому связи между процессами и потоками моделируются как композиции. Модель потоков управления помещается в пакет Process View (представление процессов – одно из архитектурных представлений в модели «4+1»).



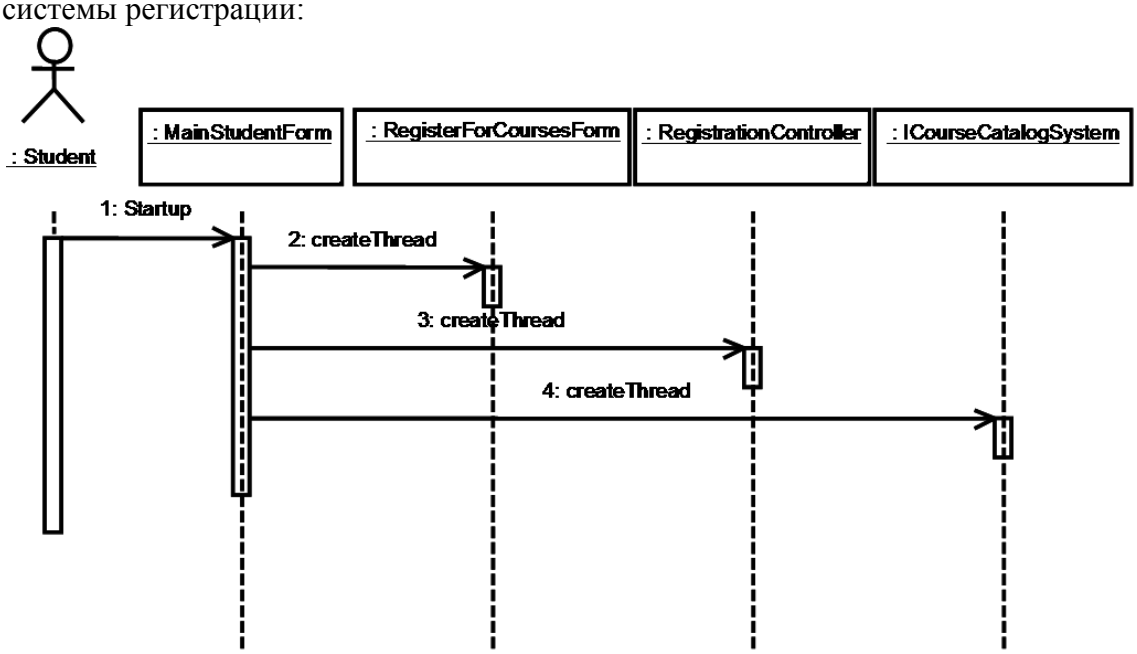
В качестве примера приведена диаграмма классов, описывающая структуру процесса регистрации студента на курсы. Обратите внимание, что все классы на ней являются активными (выделены жирными границами).

Активные классы, показанные на этих диаграммах, выполняют следующее назначение:

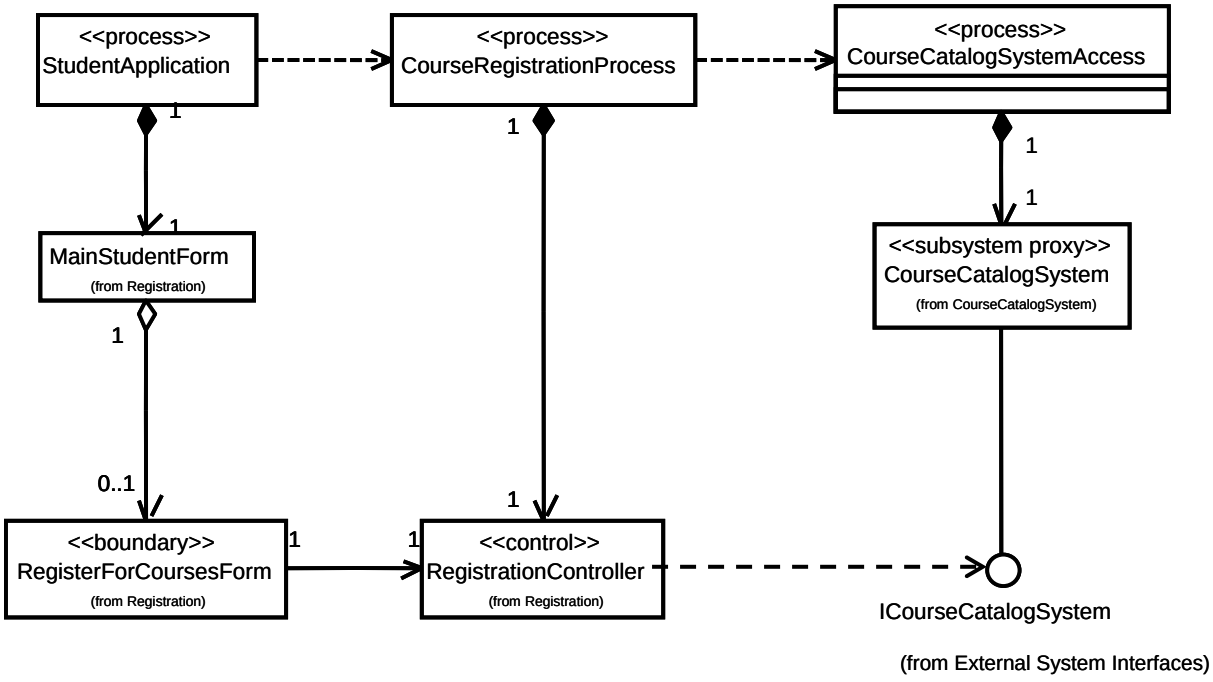
• StudentApplication – процесс, управляющий всеми функциями студента-пользователя в системе. Для каждого студента,

начинающего регистрироваться на курсы, создается один объект данного класса.
• CourseRegistrationProcess – процесс, управляющий непосредственно регистрацией студента. Для каждого студента, начинающего регистрироваться на курсы, также создается один объект данного класса.
• CourseCatalogSystemAccess – управляет доступом к системе каталога курсов. Один и тот же объект данного класса используется всеми пользователями при доступе к каталогу курсов.
• CourseCache и OfferingCache используются для асинхронного доступа к данным в БД с целью повышения производительности системы. Они представляют собой кэш для промежуточного хранения данных о курсах, извлеченных из БД.

Создание потоков во время инициализации приложения моделируется с помощью диаграмм взаимодействия. Объекты любого проектного класса или подсистемы должны существовать внутри по крайней мере одного процесса. Связи между процессами и проектными классами моделируются на диаграммах классов. Ниже приведены примеры для системы регистрации:



Обратите внимание, что на следующей диаграмме зависимости между процессами соответствуют ассоциациям и зависимостям между классами, экземпляры которых содержатся в процессах.



Следующий этап – проектирование конфигурации. Если создаваемая система является распределенной, то необходимо спроектировать ее конфигурацию в вычислительной среде, т. е., описать вычислительные ресурсы, коммуникации между ними и использование вычислительных ресурсов различными системными процессами.

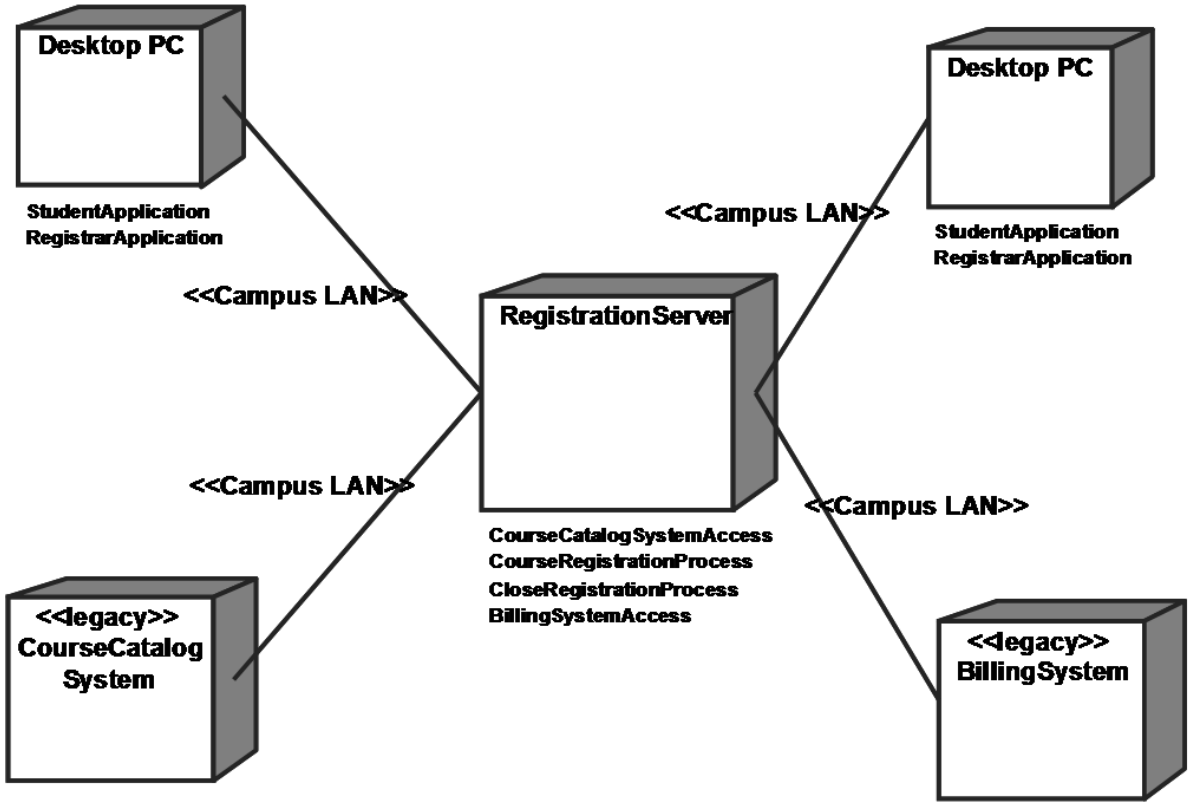
Распределенная сетевая конфигурация системы моделируется с помощью диаграммы размещения. Диаграмма размещения – это единственная диаграмма, входящая в состав представления размещения – одного из архитектурных представлений, входящих в модель «4+1» Основные элементы диаграммы размещения:

- узел (node) – вычислительный ресурс (процессор или другое устройство: дисковая память, контроллеры различных устройств и т.д.). Для узла можно задать выполняющиеся на нем процессы.
- соединение (connection) – канал взаимодействия узлов (сеть).

Распределение процессов, составляющих структуру потоков управления, по узлам сети производится с учетом следующих факторов:

- используемые образцы распределения (трехзвенная архитектура клиент-сервер, «толстый клиент», «тонкий клиент», «точка-точка» и т. д.);
- время отклика;
- минимизация сетевого трафика;
- мощность узла;
- надежность оборудования и коммуникаций.

Пример – диаграмма размещения для системы регистрации на курсы:



Следующий этап – *уточнение реализаций вариантов использования*. Уточнение заключается в модификации диаграмм взаимодействия и диаграмм классов с учетом вновь появившихся на шаге проектирования классов и подсистем, а также проектных механизмов. Вносятся изменения в описания потоков событий вариантов использования (с помощью скриптов и примечаний). Производится унификация классов и подсистем по следующим правилам:

- Имена элементов модели должны отражать их функции. Следует избегать подобных имен и синонимов.
- Элементы модели, реализующие сходное поведение или представляющие одно и то же

явление, должны объединяться.

- Классы, представляющие одно и то же понятие или имеющие одинаковые атрибуты, должны объединяться, даже если их поведение различно.
- Для абстрактных элементов модели должно использоваться наследование, повышающее гибкость модели.
- При обновлении элемента модели должны обновляться соответствующие реализации вариантов использования.

Следующий этап – *проектирование подсистем*, осуществляемое разработчиками. Оно включает в себя следующие действия:

- Распределение поведения подсистемы по ее элементам
  - документирование взаимодействия элементов подсистемы в виде кооперации («реализаций интерфейса»);
  - построение одной или более диаграмм взаимодействия для каждой операции интерфейса подсистемы/
- Документирование элементов подсистемы (в виде диаграмм классов).
- Описание зависимостей между подсистемами.

Пример:

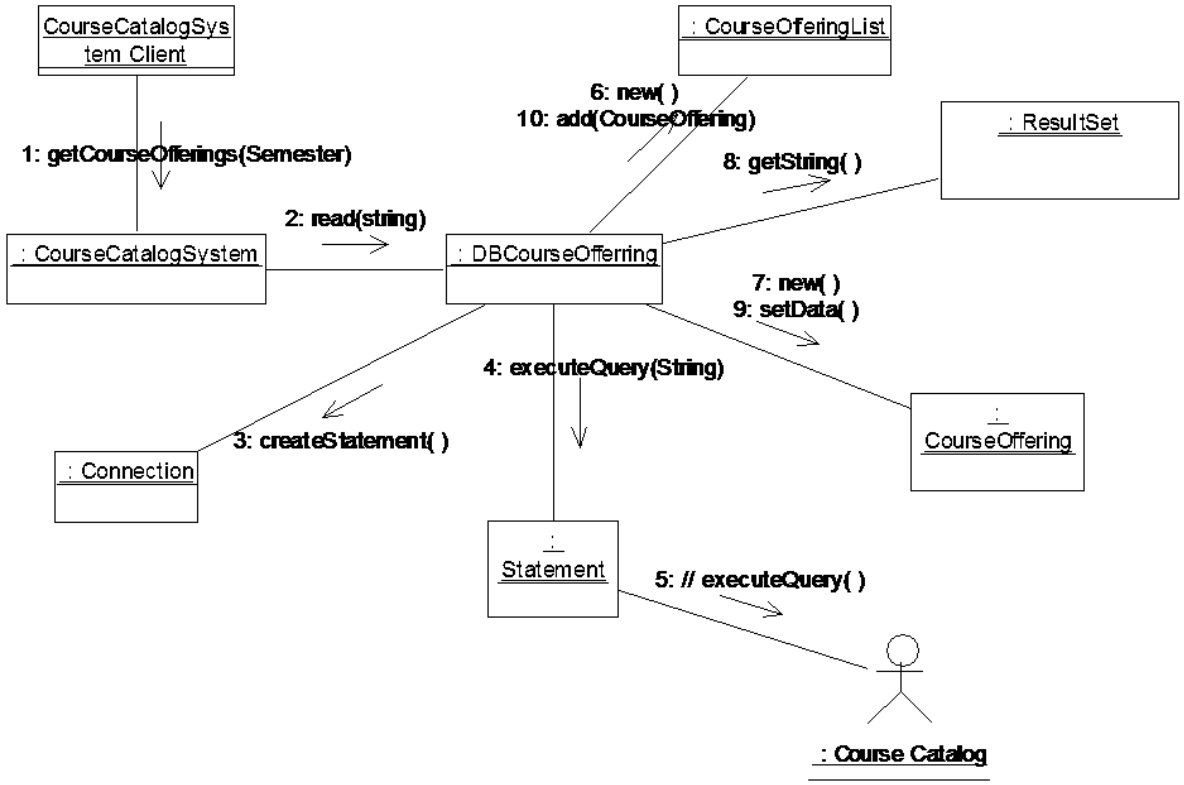


Рис.Диаграмма кооперации, описывающая реализацию интерфейсной операции getCourseOfferings() в подсистеме CourseCatalogSystem.

Заметим, что реализация подсистемы CourseCatalogSystem выполнена созданием экземпляра механизма JDBC, т. е. подстановкой класса DBCourseOffering вместо роли DBClass, CourseOffering – вместо PersistentClass, CourseOfferingList – вместо PersistentClassList, CourseCatalogSystem вместо PersistencyClient. То есть, берется шаблонная диаграмма взаимодействия из модели механизма и уточняется подстановкой конкретных классов. Аналогично будет получена диаграмма классов, описывающая структурные связи подсистемы.

Еще одним важным моментом является использованное на диаграмме соглашение моделирования: Вызов операции getCourseOfferings(), реализацию которой мы описываем,

производится клиентским объектом (названным CourseCatalogSystemClient) не имеющим указания класса. В самом деле, подсистема ничего не знает о своих клиентских классах, так что указать класс невозможно. По смыслу на месте объекта CourseCatalogSystemClient может быть либо экземпляр RegistrationController, либо экземпляр CloseRegistrationController.

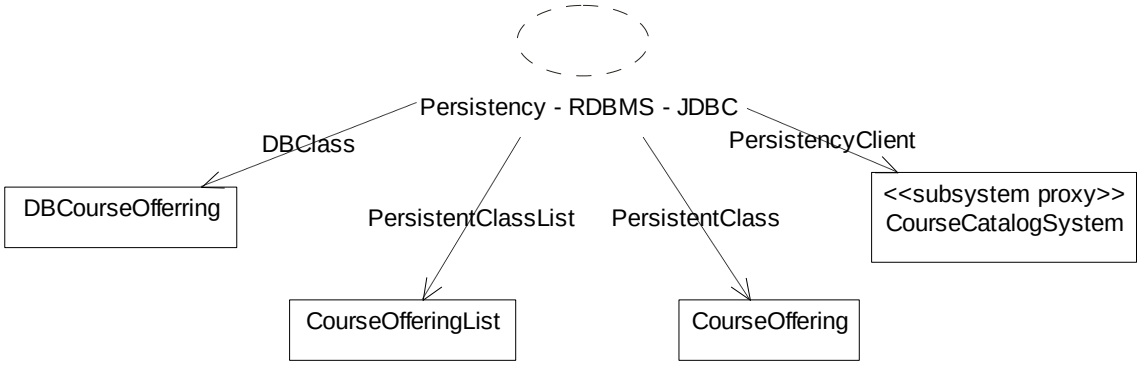
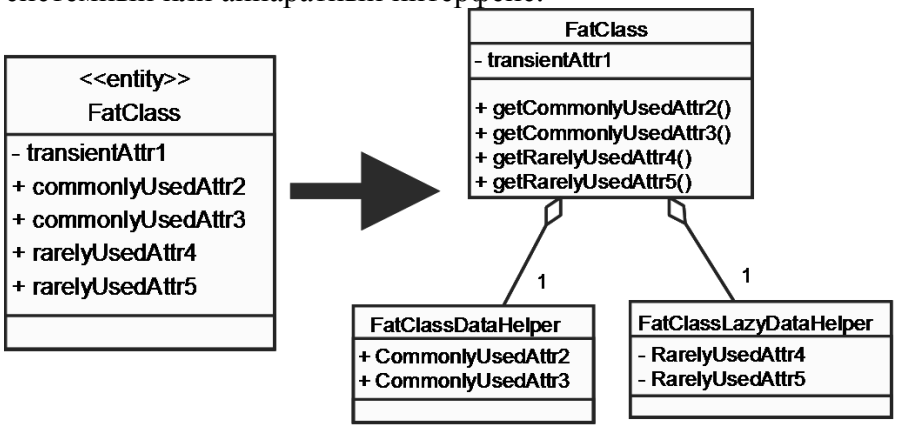


Рис. Связывание образца JDBC с конкретными классами системы

После проектирования подсистем производится *проектирование классов*, которое включает следующие действия:

1) детализация проектных классов;
2) уточнение операций и атрибутов;
3) моделирование состояний для классов;
4) уточнение связей между классами.

Каждый граничный класс преобразуется в некоторый набор классов, в зависимости от своего назначения. Это может быть набор элементов пользовательского интерфейса, зависящий от возможностей среды разработки, или набор классов, реализующий системный или аппаратный интерфейс.



Классы-сущности с учетом соображений производительности и защиты данных могут разбиваться на ряд классов. Основанием для разбиения является наличие в классе атрибутов с различной частотой использования или видимостью. Такие атрибуты, как правило, выделяются в отдельные классы.

Что касается управляющих классов, то классы, реализующие простую передачу информации от граничных классов к сущностям, могут быть удалены. Сохраняются классы, выполняющие существенную работу по управлению потоками событий (управление транзакциями, распределенная обработка и т.д.).

Полученные в результате уточнения классы подлежат непосредственной реализации в коде системы.

Обязанности классов, определенные в процессе анализа и документированные в виде «операций анализа», преобразуются в операции, которые будут реализованы в коде. При этом:

- каждой операции присваивается краткое имя, характеризующее ее результат;
- определяется полная сигнатура операции;
- создается краткое описание операции, включая смысл всех ее параметров;

- определяется видимость операции: public, private или protected;
- определяется область действия операции: операция объекта или операция класса.

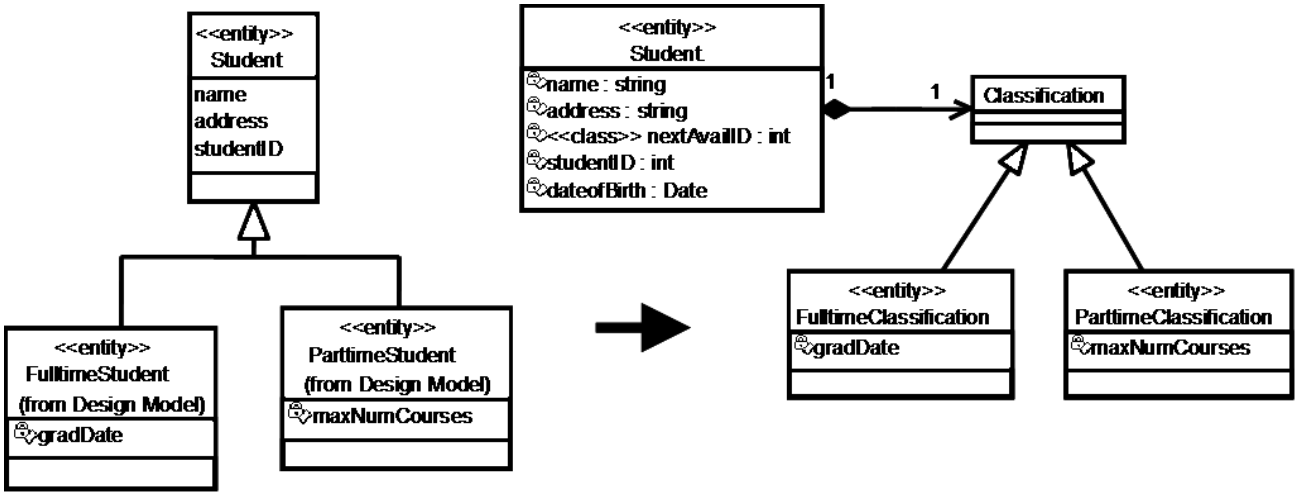
Уточнение атрибутов классов заключается в следующем:

- задается его тип атрибута и значение по умолчанию (необязательно);
- задается видимость атрибутов: public, private или protected;
- при необходимости определяются производные (вычисляемые) атрибуты.

Если в системе присутствуют объекты со сложным поведением, то строят диаграммы состояний. Построение диаграмм состояний может оказать следующее воздействие на описание классов:

- события могут отображаться в операции класса;
- особенности конкретных состояний могут повлиять на детали выполнения операций;
- описание состояний и переходов может помочь при определении атрибутов класса.

В процессе проектирования связи между классами подлежат уточнению.



Некоторые ассоциации преобразуются в зависимости (в случаях, когда соединения экземпляров классов не стабильны, т. е. временны, например, если объект является параметром или результатом операции или ее локальной переменной). Оставшиеся ассоциации преобразуются в агрегации или композиции. Композиции бывают 2-х видов:

- безраздельно обладает (зависимость по существованию, транзитивность, асимметричность, стационарность);
- обладает (зависимость по существованию, транзитивность, асимметричность).

Примеры: университет -> факультет -> кафедра; здание -> этаж здания.

Виды агрегаций:

- включает (зависимость по существованию, транзитивность);
- участник (нет ограничений).

Примеры: автомобиль -> колесо; предприятие -> сотрудник.

Определяются направления связей, при этом учитываются взаимодействия объектов, а также ожидаемое количество экземпляров классов. Классы ассоциаций являются артефактами моделирования и не поддерживаются языками программирования, поэтому они должны быть преобразованы. Структурные связи с множественными полюсами уточняются. Им приписываются 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. Соответствующие атрибуты у целевых классов должны быть удалены. Квалификатор не обязательно состоит из одного атрибута (также как и потенциальный ключ записей в таблице). Указываются типы множественных связей: множество, упорядоченное множество, мультимножество, упорядоченное мультимножество. Используются классы-контейнеры (список, хэш-

таблица и проч.). Классам с необязательными связями добавляются операции проверки, существования соединения между их экземплярами.

Связи обобщения могут преобразовываться в ситуациях с так называемой метаморфозой подтипов, когда есть необходимость менять тип объектов (например, преобразовывать студента-заочника в студента дневного отделения или наоборот).

В модели добавляются ограничения. Для их записи используется язык OCL, рассматриваемый на лекции 8.

Проектирование баз данных производится, если используется реляционная БД, при этом классы-сущности объектной модели отображаются в таблицы реляционной БД. Подробное рассмотрение вопросов проектирования БД содержится в лекции 9.

Лекция 8. 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:

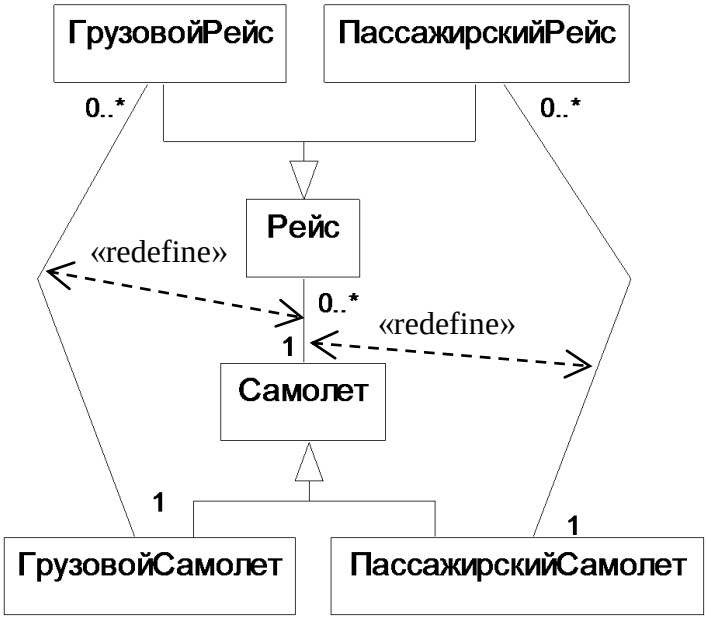
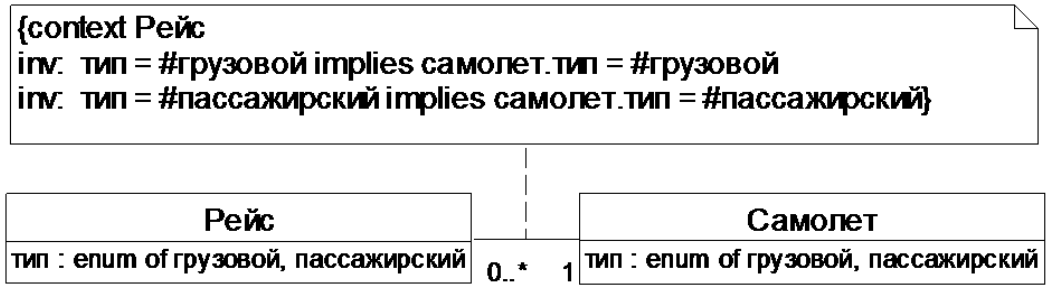


Диаграмма содержит большое количество связей (ассоциаций и обобщений) необходимых для указания, что тип самолета должен соответствовать типу рейса, т. е. что пассажиров нельзя перевозить грузовым самолетом. Это ограничение можно зафиксировать иначе, упростить диаграмму, сделать ее более наглядной:

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:**).

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);
- платформо-независимый.

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

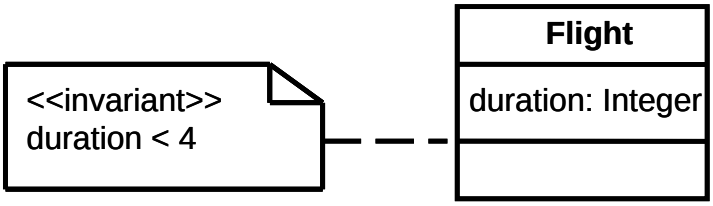
дополнительные операции **xor** и **implies**. Приоритет логических операций (кроме **not**) меньше арифметических и операций сравнения – так проще записывать сложные логические выражения. Целый и вещественный типы также стандартны. Имеются дополнительные операции **a.max(b)** и **a.min(b)**, возвращающие 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 строчным 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> ::=

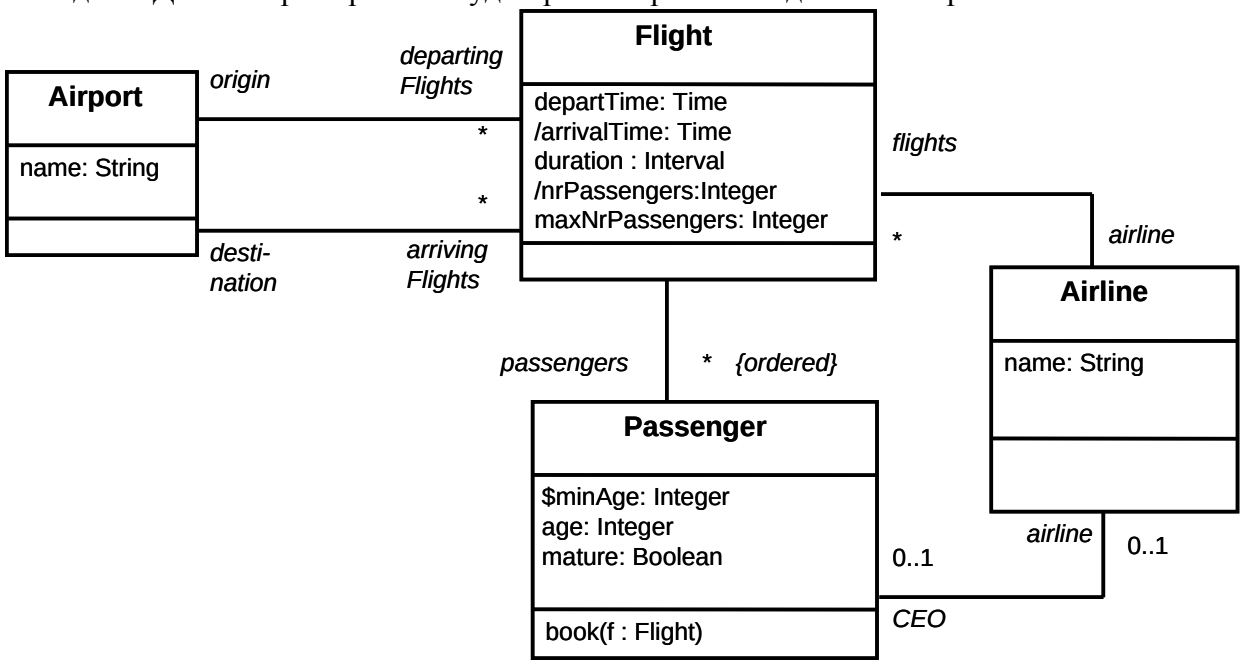
if <логическое выражение> **then** <выражение>

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.

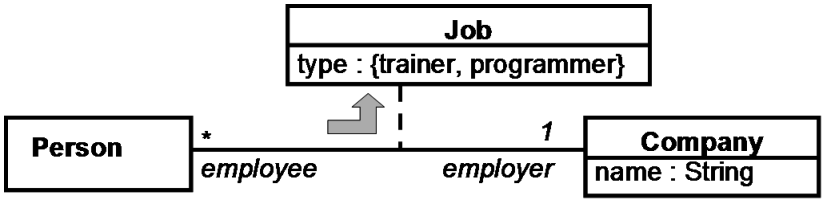
Time
$midnight: Time month : String day : Integer year : Integer hour : Integer minute : Integer
difference(t:Time):Interval before(t: Time): Boolean plus(d : Interval) : Time

Пример определения операции:
context Interval::equals(i:Interval):Boolean body:
(self.nrOfDays*60+self.nrOfHours)*60+self.nrOfMins =
(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.

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

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 = 'OAO MMM' then
Job.type = #trainer
else
Job.type = #programmer
endif

Смысл ограничения: у любой персоны, занятой в 'OAO MMM', 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 = 'OAO MMM' 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 предопределенных 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 операции oclAsType(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> **:** <type> **=** <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 авиалинии 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, **maxNrPassengers** которых больше 150.

В наследовании ограничений работает принцип подстановки Лисковой (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.

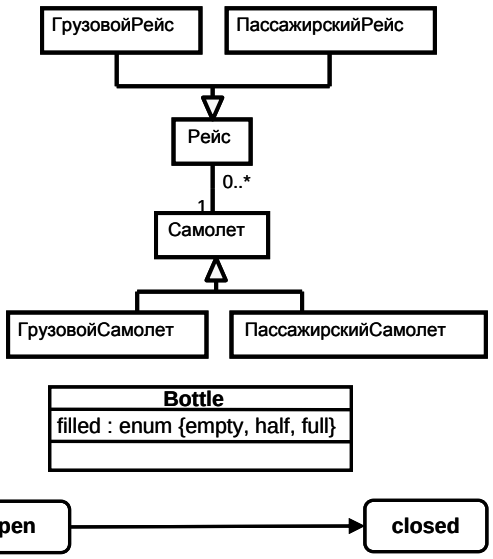
Если в ограничении требуется проверить, является ли экземпляр суперкласса также экземпляром конкретного подкласса, то используют стандартную операцию **ocllsTypeOf.** Вспоминая пример, приведенный в начале лекции, мы можем описать следующие

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 деятельностей, 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 Develepment Tools или Dresden OCL Toolkit).

Лекция 9. Проектирование баз данных

Проектирование баз данных производится, если используется реляционная БД, при этом устойчивые классы объектной модели отображаются в таблицы реляционной БД.

Основные понятия реляционной модели:

База данных – долговременное самодокументированное хранилище данных. Самодокументация = схема данных, хранящаяся в БД.

Система управления базами данных (СУБД) – ПО доступа к данным. Обеспечивает:

- защиту данных;
- эффективность;
- многопользовательский режим;
- разделение данных между несколькими приложениями;
- распределенность данных;
- безопасность.

Структура реляционных данных – совокупность таблиц. В любой таблице фиксированное количество столбцов и произвольное – строк. Совокупность значений ячеек одной строки – запись.

Оператор SQL – предложение SQL для манипуляции данными (выборка, изменение, добавление, удаление).

Ограничения – условия, являющиеся частью схемы БД. Если какая-либо добавляемая запись нарушает какое-то ограничение, то она не будет добавлена.

Виды ограничений:

Возможный ключ (потенциальный ключ) – сочетание столбцов, уникально идентифицирующих каждую запись в таблице, такое что, все столбцы необходимы для уникальной идентификации и ни в одном нет пустых значений.

Основной (первичный) ключ – возможный ключ, который предпочтительнее использовать для работы с таблицей. Есть у каждой таблицы.

Внешний ключ – ссылка из другой таблицы на возможный ключ.

Пример:

persID	Name	SurName	addr	*employer*
1	Вася	Пупкин	Лондон	*1000*

compID	compName	compAddr
1000	ОАО «МММ»	Москва

Первичные ключи: persID в 1-ой таблице, compID – во 2-ой. Внешний ключ – столбец employer 1-ой таблицы.

Для связанных таблиц актуальна поддержка ссылочной целостности. *Ссылочная целостность* – это зависимость значений во внешнем ключе от значений в первичном ключе связанной таблицы (например, при удалении записи о компании необходимо: либо проверить отсутствие связанных записей о сотрудниках и отменить удаление в случае обнаружения таковых, либо каскадировать удаление, удаляя связанные записи о сотрудниках, либо обнулить значения во внешнем ключе у связанных записей).

Для обеспечения ссылочной целостности применяют *триггеры* – процедуры, описанные на языке SQL, которые автоматически запускаются при модификации таблиц, с которыми они связаны. Триггеры не только обеспечивают целостность, с их помощью удобно реализовывать и более сложные манипуляцию с данными (бизнес-логику).

Триггеры являются частным случаем хранимых процедур. *Хранимая процедура* – это процедура, работающая с таблицами, которая скомпилирована и хранится в виде кода в

БД и может быть вызвана из клиентской программы. Хранимые процедуры выгоднее SQL-запросов, но они доступны всем приложениям, работающим с БД, что иногда нежелательно.

Реляционная схема данных и объектная модель оперируют разными понятиями, в связи с чем необходима специальная работа по объектно-реляционному отображению. Отображение возможно в обе стороны: в прямую (от классов к таблицам) и в обратную (от таблиц к классам). В лекции мы будем говорить о прямом отображении, но обратное отображение подразумевается.

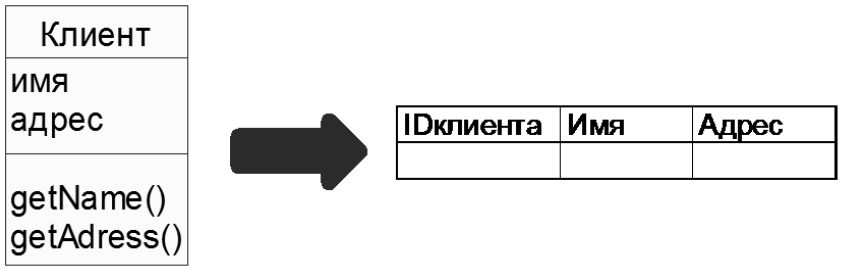
Еще одно предваряющее замечание. Схема БД зависит не только от совокупности устойчивых классов и связей между ними, но и от практических соображений. Например, может оказаться, что решение хранить все устойчивые объекты в одной «толстой» ненормализованной таблице, вполне себя оправдывает тем, что делает приемлемой скорость большинства запросов к БД. Описывая объектно-реляционное отображение, мы будем рассматривать решения, тяготеющие к получению нормализованной БД.

Переводить модель классов в схему БД предлагается в 3 этапа: отобразить классы в таблицы, отобразить ассоциации и отобразить связи обобщения.

Отображение классов

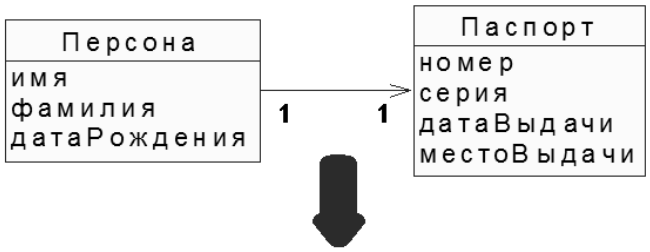
1. Каждый класс переводится в отдельную таблицу, атрибуты становятся столбцами таблицы. Операции на структуру таблицы не влияют. Они могут переводиться в хранимые процедуры, если мы ходим переложить часть работы по манипулированию данными на СУБД.
2. Уникальный идентификатор устойчивого класса превращается в первичный ключ таблицы. Если имеется несколько альтернативных уникальных идентификаторов, выбирается наиболее используемый. Если в модели для устойчивого класса явно не указан идентификатор, то в таблицу добавляется столбец ID – первичный ключ.

Пример:



При отображении ассоциаций пытаются сэкономить и не создавать дополнительные таблицы для хранения соединений между устойчивыми объектами за счет объединения нескольких таблиц в одну или добавления дополнительных столбцов в таблицы, порожденные классами, если семантика ассоциации позволяет.

Отображение бинарных ассоциаций:

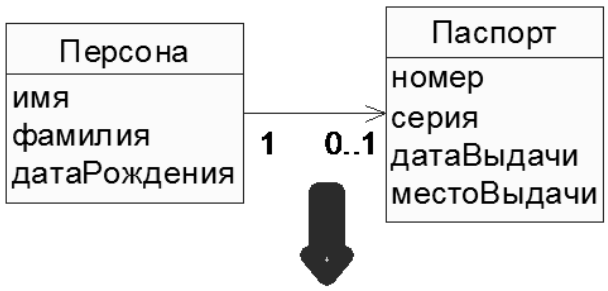


IDперсоны	Имя	Фамилия	ДатаРождения	НомерПаспорта	СерияПаспорта	ДатаВыдачи	МестоВыдачи

- «1 к 1му» – возможны различные решения, лучше создать общую таблицу для 2х

классов. Столбцы – совокупность атрибутов. Первичный ключ – любой ID (первого или второго класса). Достигается максимально возможная экономия: одна таблица представляет оба класса и ассоциацию между ними.

- «1 к 0..1» – внешний ключ добавляется к таблице необязательного класса.

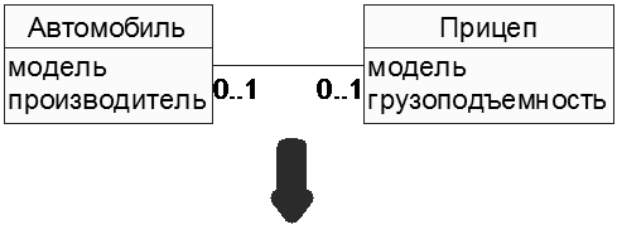


IDперсоны	Имя	Фамилия	ДатаРождения
1	Иван	Иванов	1.01.1990

НомерПаспорта	СерияПаспорта	ДатаВыдачи	МестоВыдачи	ПерсонаID
123456	77	20.02.2008	Москва	1

Вообще говоря, можно было бы использовать тот же прием, что и в «1 к 1», но получающаяся таблица будет «разреженной» – в некоторых записях будут пустые поля. Обратите внимание, что внешний ключ добавляется в таблицу, представляющую необязательный класс, поскольку записей в ней будет меньше, чем в другой.

- «0..1 к 0..1» – рекомендуется отдельная таблица для связи. Ее столбцы – внешние ключи для таблиц классов, связанных ассоциацией. Основной ключ – комбинация этих столбцов.



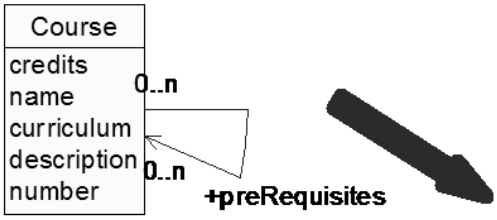
IDАвто	Модель	Производитель
1	600	Мерседес

АвтомобильID	ПрицепID
1	2

IDПрицепа	Модель	Грузоподъемность
1	ИЖ	1000
2	КАМАЗ	1500

В этом случае можно было добавить внешний ключ к какой-либо из таблиц, но не всегда допускаются пустые значения во внешнем ключе.

- «* к *» – для ассоциации заводится отдельная таблица. Ее столбцы – внешние ключи для таблиц классов, связанных ассоциацией. Основной ключ – комбинация этих столбцов.



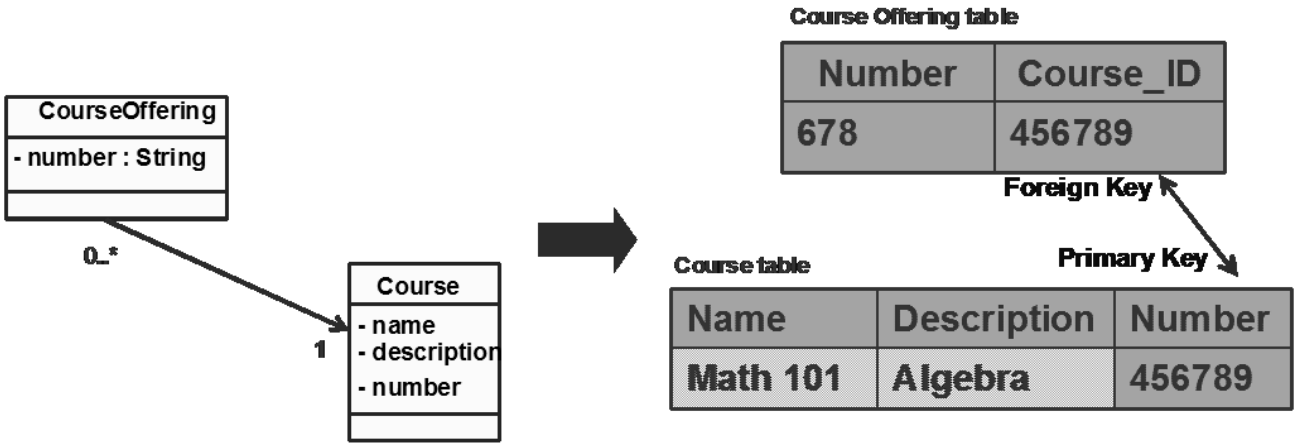
IDCourse	Credits	Name	...
1	100	МАТАН1	...
2	150	МАТАН2	...
3	150	ФУНКАН	...

IDCourse1	IDCourse2
3	1
3	2

В примере показано, как можно представить в таблицах связи между курсами (чтобы записаться на курс функционального анализа, необходимо предварительно прослушать два курса математического анализа).

- «1 к 1..*» – к таблице класса у полюса «1..*» добавляется столбец – внешний ключ для таблицы класса у полюса «один».
- «1 к 0..*» – как «1 к 1..*».

Пример:



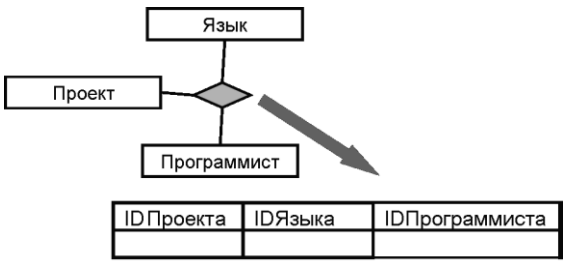
- «0..1 к *» – применяются те же решения, что и в «0..1 к 0..1».

Всякий раз, когда при реализации ассоциаций возникают связанные таблицы, возникают и ограничения целостности (в разных случаях разные), т. е. фактически добавляется не только внешний ключ и/или таблица, но и триггеры.

Отображение классов связанных N-арной ассоциацией:

Требуется таблица для хранения связи. Например, в случае тернарной (N=3) связи формируются четыре таблицы, по одной для каждого класса и одна для связи. Таблица связи будет иметь среди своих атрибутов ключи каждой из 3х других таблиц.

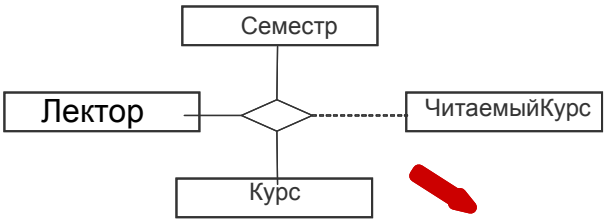
Пример:



Отображение классов-ассоциаций:

Атрибуты класса-ассоциации добавляются либо в создаваемую для связи таблицу, либо (если дополнительная таблица не требуется) в ту таблицу, куда добавляется внешний ключ.

Пример:



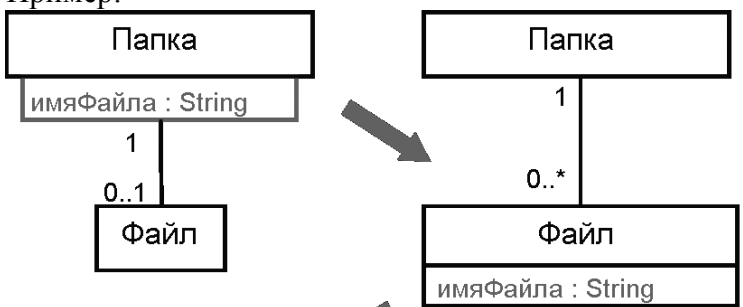
IDЛектор	IDСеместр	IDКурс	АтрибутЧитКурс

Отображение квалифицированных ассоциаций

Решение:

- Определить, какие мощности были бы у полюсов ассоциации без квалификатора.
- Применить решение для ассоциаций с полученными мощностями.
- Добавить квалификатор как столбец (или столбцы) в ту же таблицу, куда добавляются внешние ключи.
- Как правило, квалификатор становится частью возможного ключа таблицы, в которую он добавлен.

Пример:



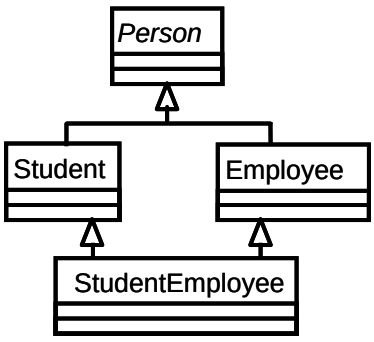
IDПапки	...
1	

IDФайла	IDПапки	имяФайла
2	1	readme

Отображение обобщения (наследования):

Стратегии:

- для каждого класса своя таблица;
- для всей иерархии наследования одна таблица;
- таблицы только для конкретных (не абстрактных) классов;
- таблицы только для различных конкретных классов.



Какой именно способ выбрать диктуют соображения эффективности (скорость в обмен на объем памяти). Рассмотрим на примере. Пусть класс Person – абстрактный.

При использовании 1-го подхода будут созданы 4 таблицы. У таблиц Person, Student, Employee будет дополнительный столбец – тип, в котором будет храниться реальный тип объекта. Для каждого экземпляра класса Student будут две записи, одна в таблице студентов, вторая – в таблице персон. Для экземпляра StudentEmployee – четыре записи, по одной в каждой таблице. При чем у всех их будет

одинаковое значение первичного ключа. Дублирование записей позволит быстро находить всех персон, всех студентов и т. п.

При втором подходе используется общая разреженная таблица. В ней также для каждой записи хранится ее тип. Неудобство состоит в том, что для любой записи о персоне есть риск «залезть» в поля, принадлежащие подклассу.

Третий подход позволяет сэкономить одну таблицу – Person – для поиска всех персон в БД будет создан View, объединяющий записи 3-х созданных таблиц.

Четвертый путь дает две таблицы (студентов и служащих) и два View (один для персон, второй для студентов-служащих).

Лекция 10. Образцы проектирования

Образец (паттерн) – это типовое проектное решение конкретной задачи проектирования, описанное специальным образом, чтобы облегчить его повторное применение.

Фактически, каждый паттерн является формализованным опытом лучших разработчиков в индустрии создания ПО. Исторически понятие образца возникло в архитектуре, введено архитектором Кристофером Александром – проектировщиком зданий и городов в конце 1970-х.

«Любой паттерн описывает задачу, которая снова и снова возникает в нашей работе, а также принцип ее решения, причем таким образом, что это решение можно потом использовать миллион раз, ничего не изобретая заново.» Кристофер Александр.

Основные составляющие части образца:

Имя. Идентифицирует образец, Хорошее имя характеризует решаемую проблему и способ ее решения.

Задача. Описание ситуации, в которой следует применять образец. Это описание включает в себя: постановку проблемы, контекст проблемы, перечень условий, при выполнении которых имеет смысл применять образец.

Решение. Описание элементов архитектуры, связей между ними, функций каждого элемента. Включает в себя UML-диаграммы.

Результаты. Следствия применения паттерна и компромиссы. Преимущества и недостатки образца. Влияние использования образца на гибкость, расширяемость и переносимость системы.

Полное описание образца включает:

- Название.
- Назначение (краткое описание функций образца и решаемой задачи).
- Другие известные названия.
- Мотивация (иллюстрация решаемой задачи проектирования).
- Применимость (описание ситуаций, в которых применим паттерн. Примеры проектирования, которые можно улучшить с помощью образца).
- Структура (диаграмма классов).
- Участники (слоты или роли, задействованные в образце, на место которых следует подставлять конкретные проектные классы).
- Отношения (диаграмма взаимодействия экземпляров классов-участников).
- Результаты.
- Реализация (сложности, подводные камни при реализации образца, рекомендации, зависимость от языка программирования).
- Пример кода.
- Известные применения (2 или более применений в различных областях).
- Родственные паттерны (связь с другими образцами, различия, использование образца в сочетании с другими).

Каталог образцов[3] содержит 23 образца. Существуют другие каталоги – например, каталог Фаулера[4].

Классификация образцов:

- Порождающие образцы (способы создания экземпляров классов).
- Структурные образцы (способы задания статических связей между проектными классами).
- Образцы поведения (способы организации взаимодействий).

[3] Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – СПб.: Питер, 2007.

[4] Фаулер М. Архитектура корпоративных приложений. – М.: Вильямс, 2007.

Рассмотрим несколько образцов.

Абстрактная фабрика (Abstract Factory)

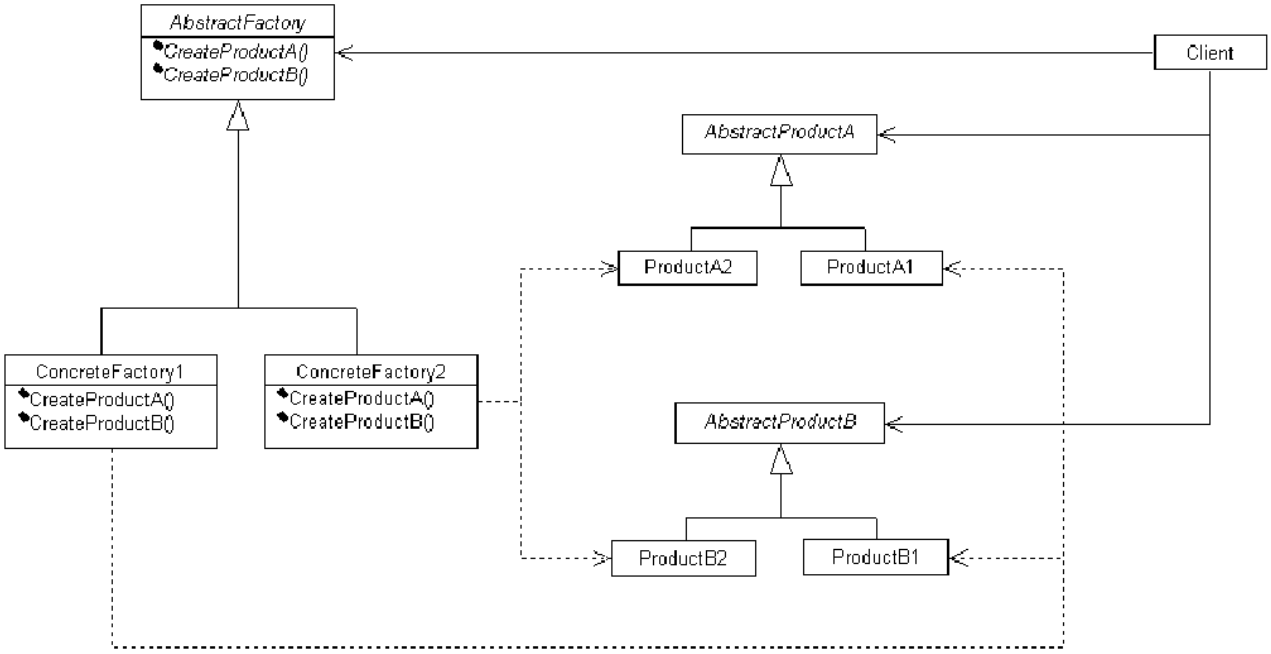
Классификация: образец порождения объектов.

Назначение: предоставляет интерфейс для создания взаимосвязанных и взаимозависимых объектов, не определяя их конкретных классов.

Мотивация: часто встает задача проектирования программной системы независимой от конкретной реализации GUI.

Ситуации применимости:

- Система не должна зависеть от того как создаются, компонуются и представляются входящие в нее объекты;
- Входящие в семейство объекты должны использоваться вместе и необходимо обеспечить выполнение этого ограничения;
- Система должна конфигурироваться одним из семейств составляющих ее объектов;
- Предоставляется библиотека классов, реализация которых скрыта за интерфейсом.



Участники:

- AbstractFactory – интерфейс с операциями для порождения экземпляров абстрактных классов-продуктов.
- ConcreteFactory – реализация порождения экземпляров конкретных классов.
- AbstractProduct – интерфейс с операциями класса-продукта.
- ConcreteProduct – реализация абстрактного продукта, объекты которой порождаются одной из конкретных фабрик.
- Client – класс пользующийся интерфейсами AbstractFactory и AbstractProduct.

Отношения:

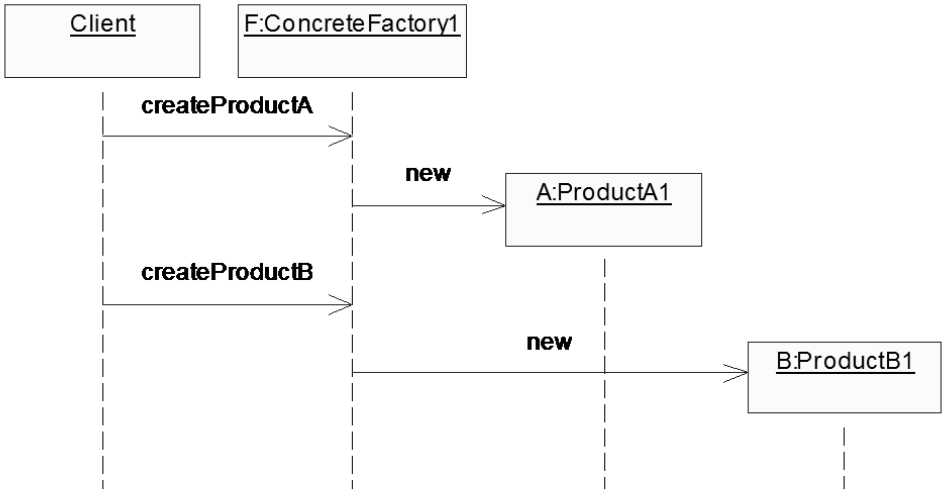
Обычно, во время выполнения создается один экземпляр ConcreteFactory, который создает экземпляры конкретных продуктов одного из семейств. Для использования объектов другого семейства нужно породить другую конкретную фабрику.

Результаты:

- изоляция клиента от деталей реализации классов-продуктов (их имена известны только конкретной фабрике);
- упрощение замены семейств продуктов;
- набор продуктов фиксирован, добавлять новые трудно.

Представим, что нужно добавить третий класс продуктов. Потребуется добавить

иерархию из 3-х классов и дополнительный метод в каждую фабрику.



Пример: взаимодействие при создании новых объектов по образцу Абстрактная фабрика.

Фабричный метод (Factory method)

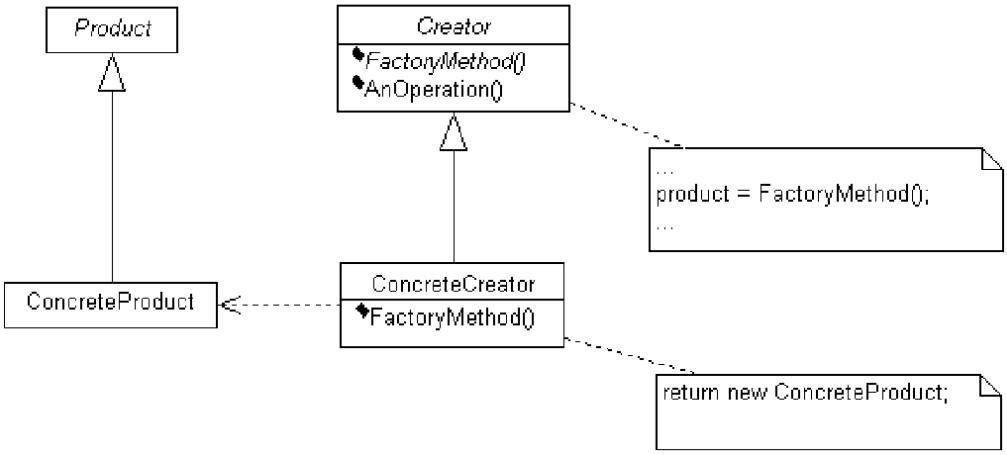
Классификация: образец порождения объектов.

Назначение: определяет интерфейс для создания объектов, но оставляет реализациям решение о том, какой объект какого именно класса создавать.

Мотивация: нужно создать экземпляр одной из реализаций интерфейса, но заранее неизвестно какой. Например, в текстовом редакторе работающем с разными форматами при создании нового документа не известен его тип.

Ситуации применимости:

- Класс заранее не знает, объекты каких классов нужно создавать;
- Известно лишь, что это будет экземпляр одного из подклассов (реализации) какого-то абстрактного класса (интерфейса);
- Класс делегирует свои обязанности одному из вспомогательных подклассов и планируется локализовать знание о том, какой именно из подклассов берет эти обязанности на себя.



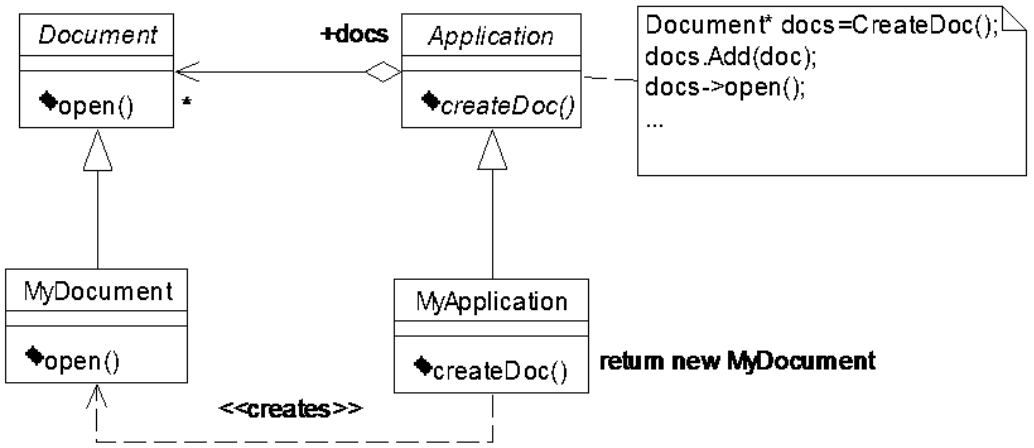
Участники:

- Product – интерфейс порождаемых объектов;
- ConcreteProduct – конкретные реализации продукта;
- Creator – интерфейс порождения экземпляров продукта, в котором определен фабричный метод (factoryMethod), может определять реализацию фабричного метода

по умолчанию;

- ConcreteCreator – реализация создателя для порождения конкретного продукта.

Пример (редактор разноформатных документов):



Отношения:

Creator полагается на свои подклассы в определении factoryMethod'a, возвращающего экземпляр конкретного продукта.

Результаты:

- нет необходимости встраивать в код зависящие от приложения классы;
- код связан лишь с интерфейсом класса Product, поэтому может работать с любым конкретным продуктом;
- накладные расходы: при создании даже одного экземпляра конкретного продукта понадобится создать экземпляр ConcreteCreator.

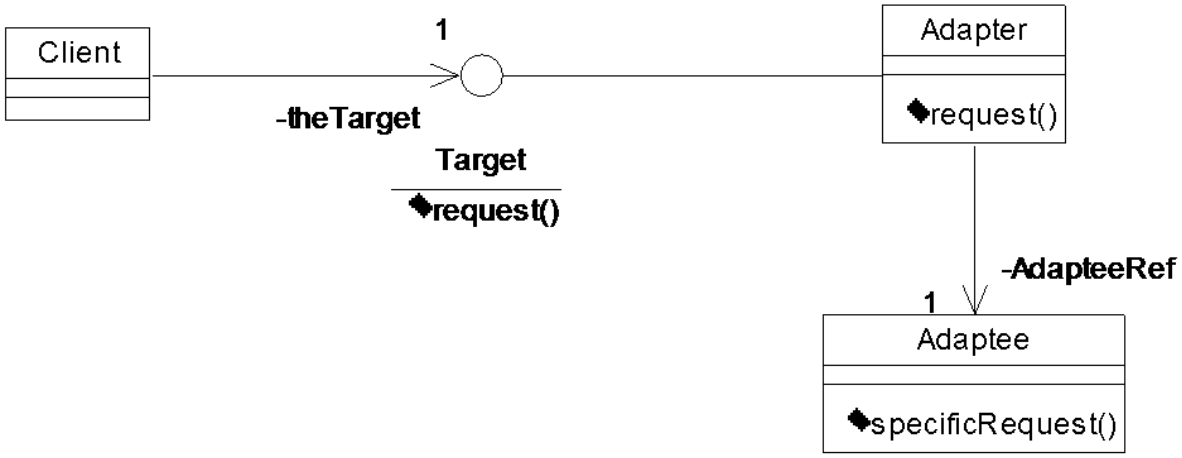
Адаптер (Adapter)

Классификация: структурный образец.

Назначение: преобразует один интерфейс в другой, обеспечивая совместимость.

Мотивация: есть библиотечный класс, который можно было бы использовать, но он не поддерживает требуемый интерфейс.

Ситуация применимости: обеспечение совместимости существующего класса при его повторном использовании.



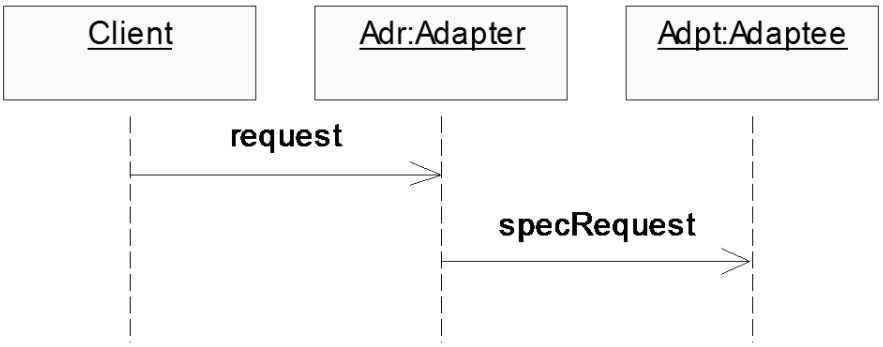
Участники:

- Target – требуемый клиентом интерфейс;
- Client – клиент;
- Adaptee – адаптируемый класс или интерфейс;

- Adapter – класс-адаптер, в методе request вызывается specificRequest из Adaaptee.

Отношения:

Адаптер получает запросы клиентов и преобразует их в вызовы операций адаптируемого класса или интерфейса.



Результаты:

- паттерн не работает, если надо адаптировать не только класс, но и его подклассы;
- вводится только один дополнительный объект.

Composite (Компоновщик)

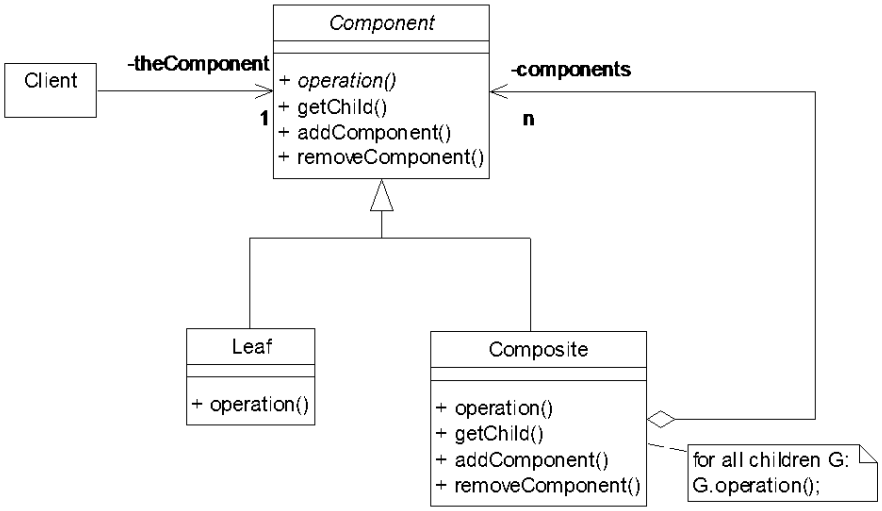
Классификация: структурный образец.

Назначение: организует объекты в древовидные структуры, позволяет клиентам единообразно работать с составными и элементарными объектами.

Мотивация: есть совокупность объектов-контейнеров, состоящих из элементарных объектов и других контейнеров.

Ситуации применимости:

- нужно представить иерархию объектов «часть-целое»;
- нужно дать одинаковый интерфейс к составным и простым объектам.



Участники:

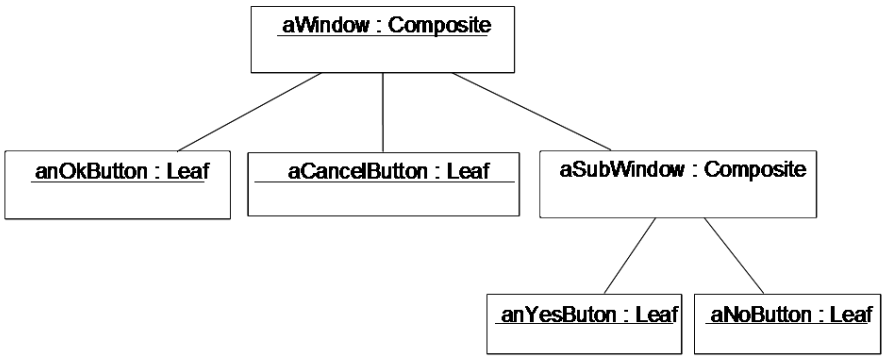
- Component – компонент, определяющий единый интерфейс, содержащий реализации общих операций по умолчанию;
- Leaf – простые (элементарные) объекты;
- Composite – составной объект;
- Client – работает с объектами через интерфейс, объявленный в компоненте.

Отношения:

Клиенты вызывают операции Component. Если получатель запроса – лист, то он и обрабатывает запрос. Если – составной объект, то он дополнительно рассылает запросы своим частям.

Результаты:

- упрощается клиент, т. к. он единообразно работает с целым и частями;
- облегчается добавление новых классов – они являются подклассами Leaf или Composite;
- трудно ограничить состав видов объектов в составе композиции того или иного вида.



Рассмотрим пример (диаграмму объектов):

Если требуется запретить появление кнопок определенного вида в окнах определенного вида, то эта задача целиком ложится на программиста, образец Компоновщик нисколько не помогает её решить.

Мост (Bridge)

Классификация: структурный образец.

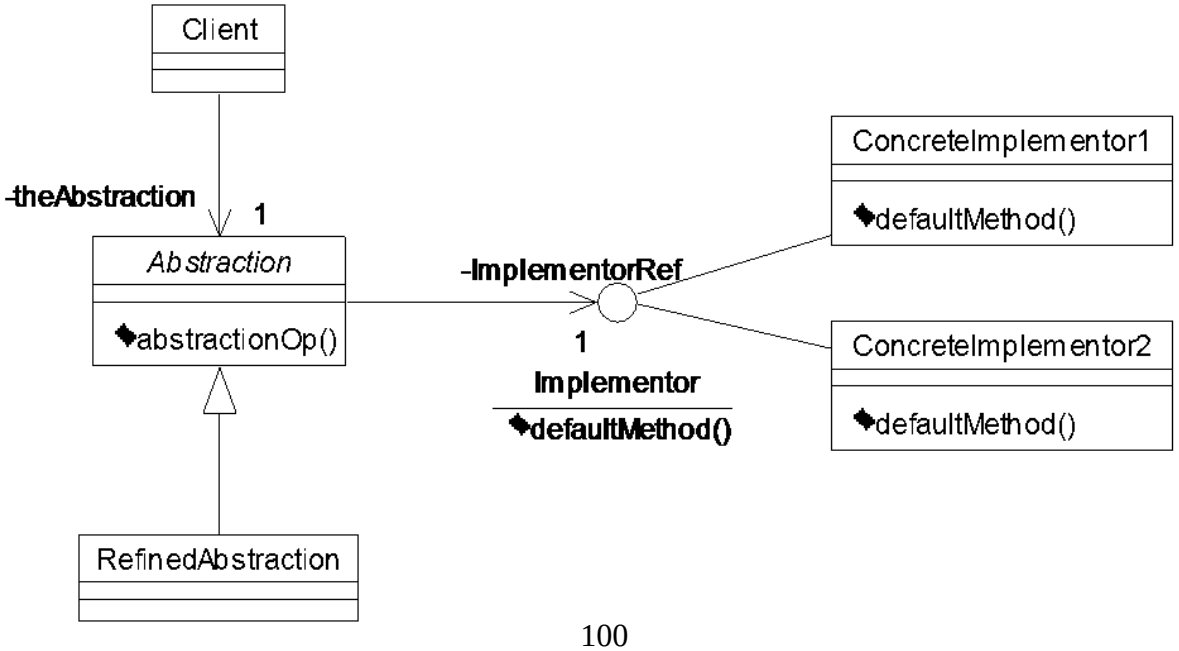
Назначение: отделить абстракцию от реализации.

Мотивация: наследование жестко привязывает реализацию к абстракции, поэтому лучше иметь иерархию наследования для интерфейсов и отдельно их реализации.

Ситуации применимости:

- обеспечение независимости абстракции и реализации;
- необходимо расширять подклассами как интерфейсы, так и их реализации;
- изменения в реализации не должны влиять на клиента;
- необходимо разделить большую иерархию наследования на части.

Участники:



- Abstraction – абстракция, в которой определен интерфейс требуемый клиенту;
- RefinedAbstraction – уточненная абстракция с расширенным интерфейсом;
- Implementor – интерфейс для классов-реализаций;
- ConcreteImplementor – конкретный реализатор.

Отношения:

Абстракция перенаправляет запросы клиента одной из реализаций Implementora.

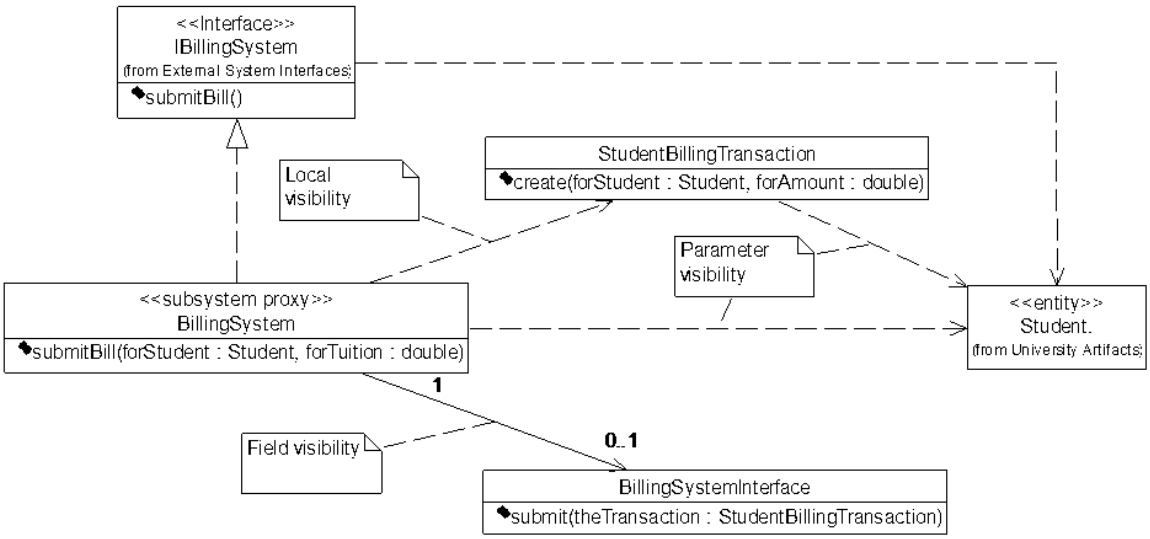
Результаты:

- реализация отделяется от интерфейса;
- чтобы заменить реализацию нет необходимости перекомпилировать абстакцию и ее клиента;
- система становится более легко модифицируемой.

Фасад (Facade)

Структурный паттерн, идея которого в том, чтобы предоставить унифицированный интерфейс к подсистеме (или пакету) в виде прокси-класса SubsystemProxy (в случае пакета – в виде фасада пакета).

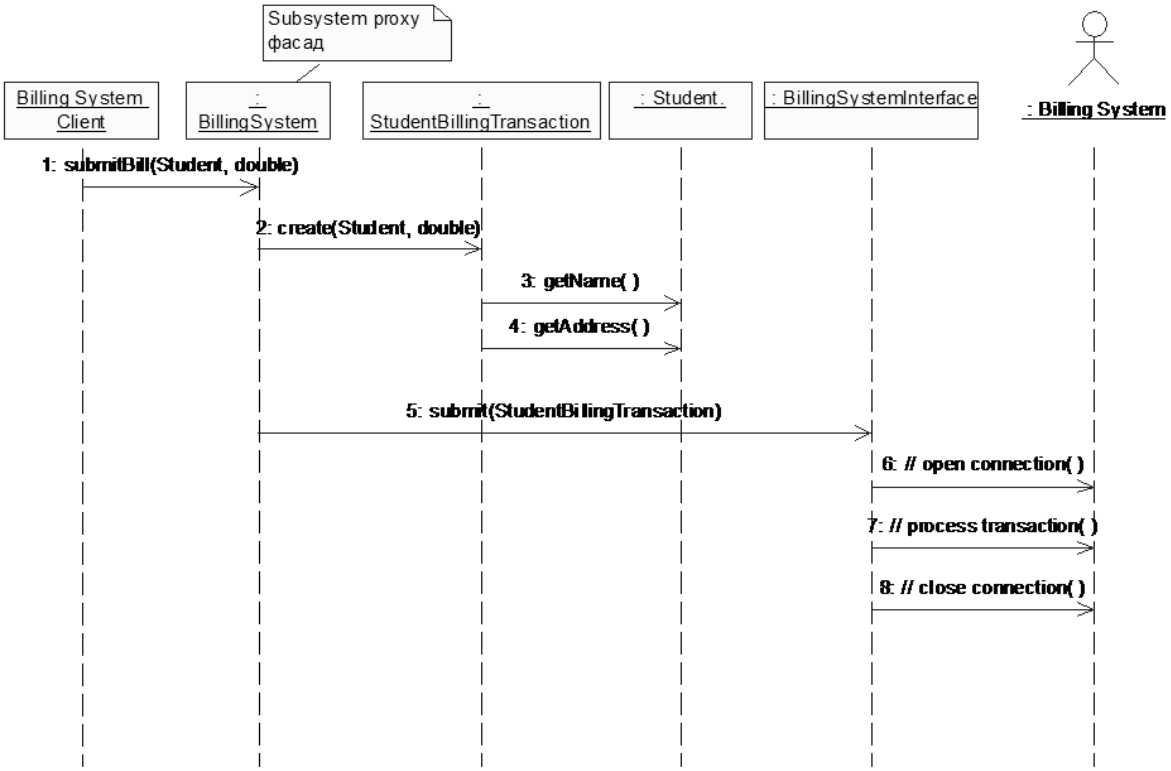
Пример:



Используется для предоставления простого интерфейса к сложной подсистеме, явного определения точки входа в подсистему, сокращения связей клиентов подсистемы с ее внутренними классами.

Упрощает работу с подсистемой, ослабляет связность, скрывает внутреннее устройство подсистемы.

В системе регистрации ВУЗа для подсистемы BillingSystem создается фасад BillingSystem, реализующий единственную операцию submitBill интерфейса подсистемы IBillingSystem.



Реализация операции заключается в формировании параметра для вызова метода BillingSystemInterface::submit(theTransaction) и собственно вызова этой операции. По сути, фасад здесь является также Адаптером (вызов submitBill преобразуется в вызов submit граничного класса). Ситуация, когда совместно используются несколько образцов, часто встречается на практике.

Proxy (Заместитель)

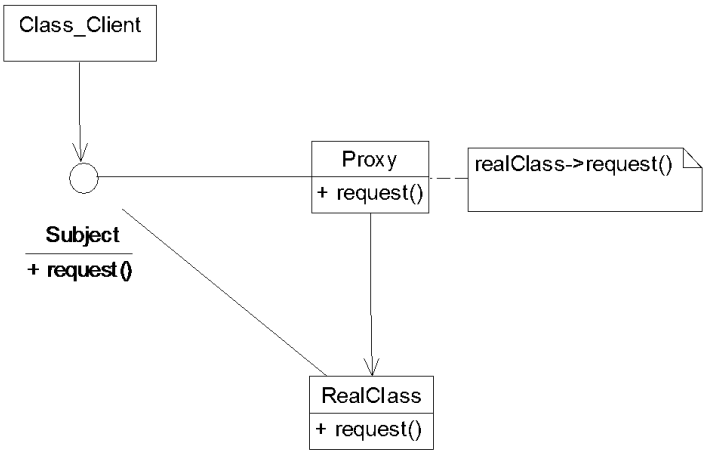
Классификация: структурный образец.

Назначение: для объекта создается суррогат, контролирующий доступ.

Мотивация: есть «тяжелый» класс, объекты которого разумно создавать по требованию – эта обязанность возлагается на легкие суррогаты.

Ситуации применимости:

- удаленный заместитель (тяжелый объект невыгодно перемещать с узла на узел, поэтому на другом узле его представляет заместитель);
- виртуальный заместитель;
- защищающий заместитель (проверяет права доступа).

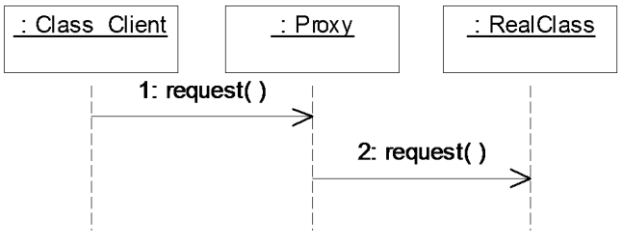


Участники:

- Proxy – заместитель, хранящий ссылку на реальный объект;
- Class_Client – клиентский класс;
- Subject –общий интерфейс для класса и его заместителя;
- RealClass – класс, для которого создается заместитель.

Отношения:

Заместитель получает запрос клиента и переадресует его реальному классу. Детали зависят от вида заместителя.



Результаты (зависят от вида заместителя):

- удаленный заместитель скрывает тот факт, что реальный объект находится на другом узле;
- виртуальный заместитель оптимизирует приложение («тяжелые» объекты создаются по требованию, пока в их создании нет необходимости, их представляют «легкие» объекты-заместители);
- защищающий заместитель обеспечивает нужный режим доступа к объекту.

Цепочка обязанностей (Chain of Responsibility)

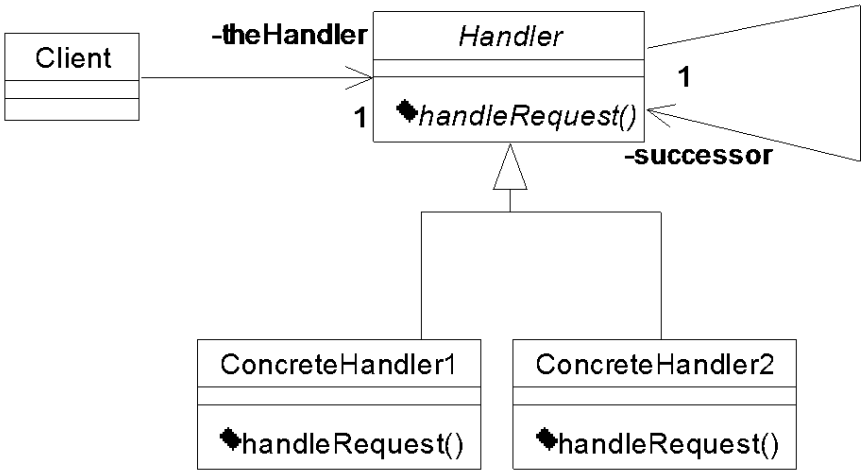
Классификация: образец поведения.

Назначение: избежать привязки отправителя запроса к получателю, давая возможность передать запрос через нескольких посредников.

Ситуации применимости:

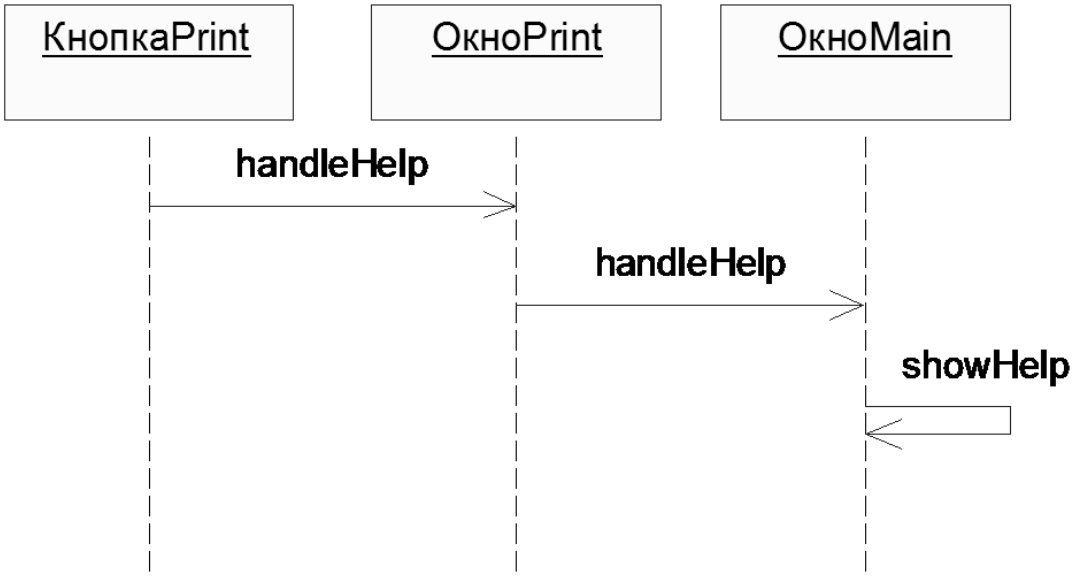
- есть более одного объекта, способного обработать запрос, настоящий обработчик неизвестен и должен быть найден автоматически (передадим по цепочке, пока кто-нибудь не обработает);
- адресат запроса не указан, но один из группы;

Участники:



- Handler – обобщенный обработчик, все специальные обработчики в цепочке являются его подклассами;

- ConcreteHandler – конкретный обработчик:
  - обрабатывает запрос;
  - знает следующего по цепочке;
  - если может обработать – обрабатывает иначе передает дальше.
- Client – отправляет запрос началу цепочки.



Например, при вызове справки о кнопке Print кнопка не знает, кто должен показывать справку, поэтому она передает запрос по цепочке окну Print, оно, в свою очередь, главному окну, которое может обработать запрос.

Результаты:

- ослабляется связность (клиент связан лишь с началом цепочки);
- гибкость в распределении обязанностей;
- нет гарантий, что запрос будет обработан, может дойти до конца цепочки и пропасть.

Iterator (Итератор)

Классификация: образец поведения.

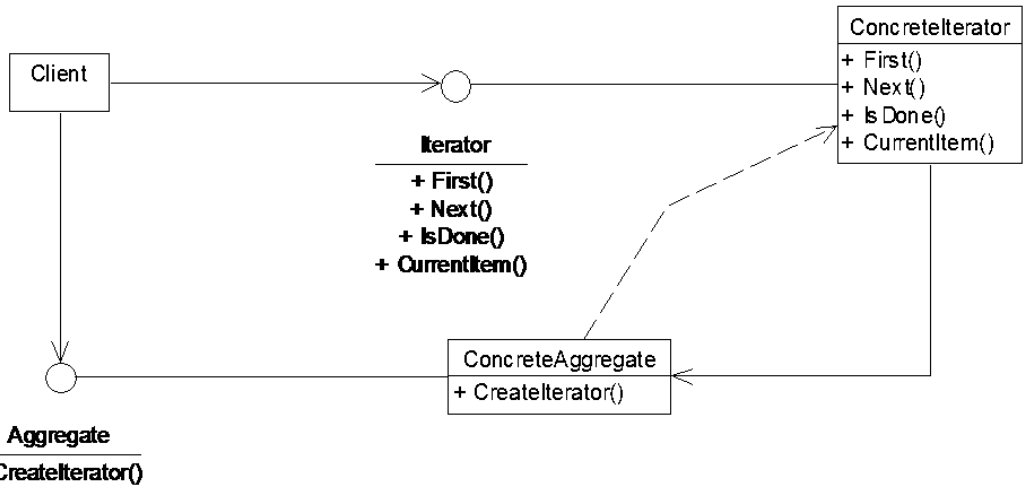
Назначение: дать последовательный доступ к набору однородных объектов, не раскрывая его внутреннего представления.

Ситуация применимости: есть структура данных, для которой нужно реализовать один или несколько способов обхода – каждый осуществляется отдельным итератором.

Участники:

- Iterator – общий интерфейс для доступа и обхода структуры данных;
- ConcreteIterator –
  - конкретный способ обхода;
  - помнит позицию курсора (текущий элемент);
- Aggregate – интерфейс для создания итератора;

- ConcreteAggregate – реализация интерфейса Aggregate.



Результаты:

- поддерживаются разные способы обхода;
- упрощается интерфейс Aggregate;
- есть возможность иметь одновременно несколько активных обходов (столько, сколько объектов-итераторов).

Strategy (Стратегия)

Классификация: образец поведения.

Назначение: Определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми.

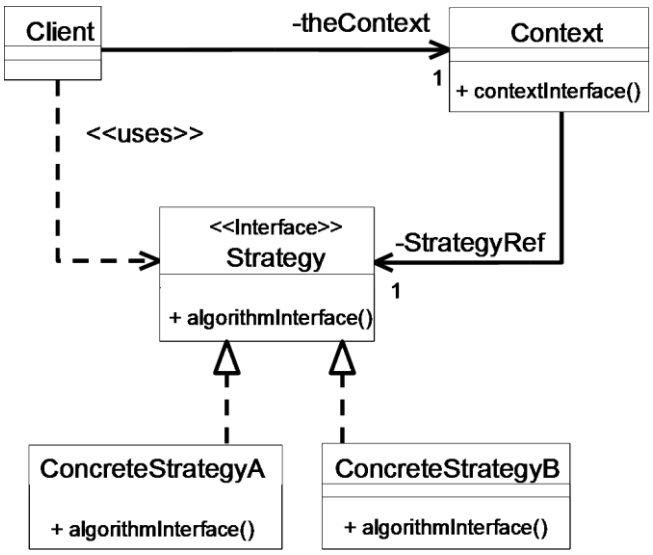
Мотивация: есть несколько алгоритмов решения одной задачи, которые нежелательно «зашивать» в клиентский класс.

Ситуации применимости:

- Имеется много родственных классов, отличающихся только поведением.
- Необходимо иметь несколько разных реализаций одной операции.
- Нужно скрыть от клиента сложные, специфичные для алгоритма структуры данных.
- Упрощение кода метода, представляющего собой длинное ветвление или switch.

Участники:

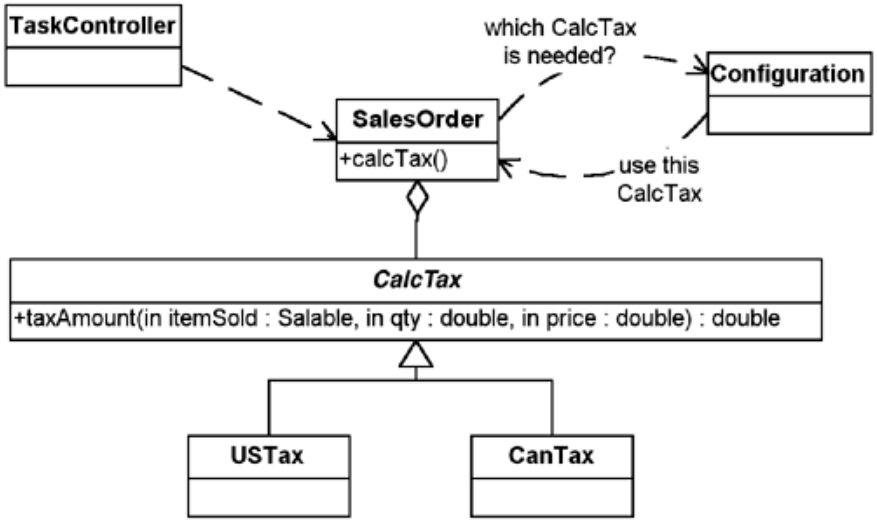
- Strategy – интерфейс общий для семейства алгоритмов;
- ConcreteStrategy – конкретная стратегия, реализующая интерфейс;
- Context – контекст, направляющий запросы клиента стратегиям;
- Client – клиентский класс.



Результаты:

- Иерархия классов стратегий определяет семейство алгоритмов или поведений, которые можно повторно использовать.
- Инкапсуляция алгоритма в отдельный класс позволяет изменять его независимо от контекста.
- Избавляемся от if и switch (улучшаем читаемость кода).
- Интерфейс класса Strategy общий для всех подклассов ConcreteStrategy – неважно, сложна или тривиальна их реализация. Поэтому вполне вероятно, что некоторые стратегии не будут пользоваться всей передаваемой им информацией, особенно простые.

Приведем пример использования образца для реализации разных стратегий расчета налогов:



Decorator (Декоратор)

Классификация: структурный образец.

Назначение: добавление объекту новых обязанностей в динамике. Альтернатива подклассам.

Мотивация: Например, хотим, чтобы библиотека GUI могла добавлять свойства (рамку) или новое поведение (прокрутку) к любому элементу GUI. Для этого «оборачиваем» элемент GUI в объект-декоратор. Декоратор имеет тот же интерфейс. Он переадресует запросы элементу, который в него «завернут».

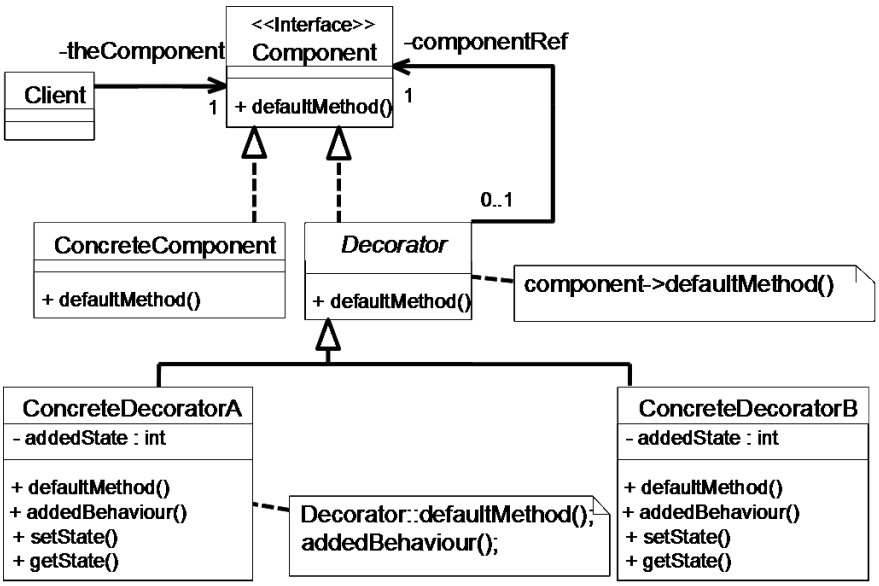
Ситуации применимости:

- для динамического, прозрачного для клиентов добавления обязанностей объектам;
- для реализации обязанностей, которые могут быть сняты с объекта;
- когда расширение путем порождения подклассов по каким-то причинам неудобно или невозможно.

Участники:

- Component – интерфейс для декорируемых объектов;
- ConcreteComponent – класс декорируемых объектов;
- Decorator – декоратор, ссылающийся на декорируемый объект и определяющий интерфейс для декораторов, соответствующий интерфейсу Component;
- ConcreteDecorator – реализует какое-либо дополнительное поведение;
- Client – клиентский класс.

Результаты:



- Позволяет гибко добавлять объекту новые обязанности. Обязанности могут быть добавлены или удалены во время выполнения программы. Применение нескольких декораторов обеспечивает сочетание добавляемых обязанностей.
- Хотя декорированный объект и обычный имеют общий интерфейс, они различны.
- Получаемая система состоит из множества мелких объектов, которые похожи друг на друга. Проектировщик, разбирающийся в устройстве системы, может легко настроить ее, но постороннему изучать и отлаживать ее тяжело.

Рассмотрим пример, в котором для класса Ticket применены две обертки для печати с верхним и нижним колонтитулом. Диаграмма классов:

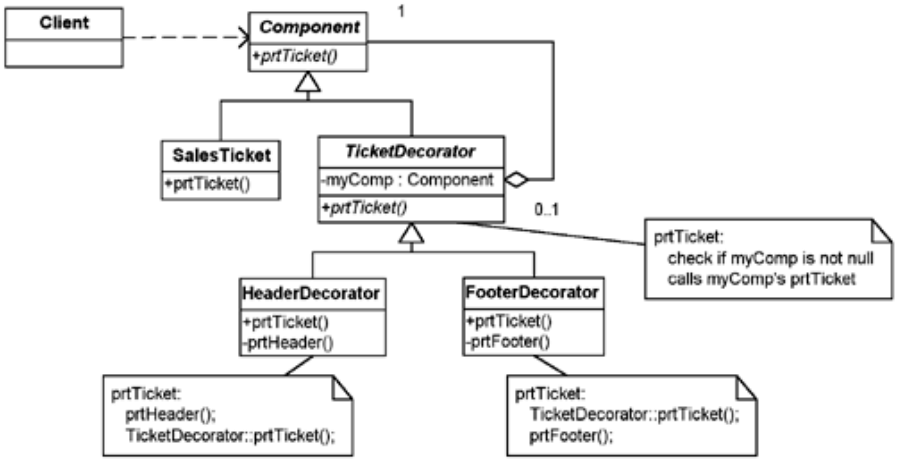
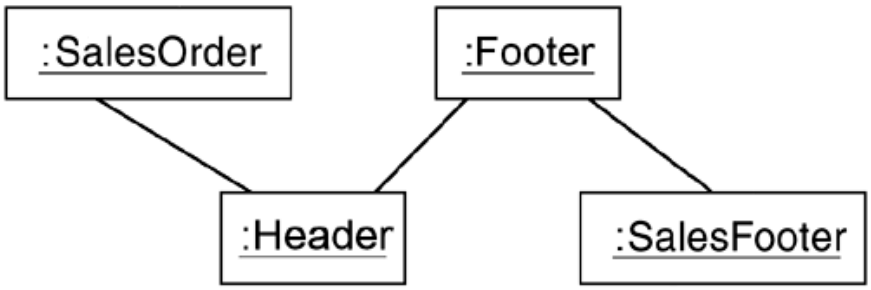


Диаграмма кооперации, демонстрирующая цепочку объектов, задействованных при печати:



Лекция 11. Технология создания программного обеспечения. RUP

Основные определения:

Технология создания ПО (ТС ПО) – это упорядоченная совокупность взаимосвязанных технологических процессов в рамках ЖЦ ПО.

Технологический процесс – это совокупность взаимосвязанных технологических операций.

Технологическая операция – это основная единица работы, выполняемая определенной ролью, которая:

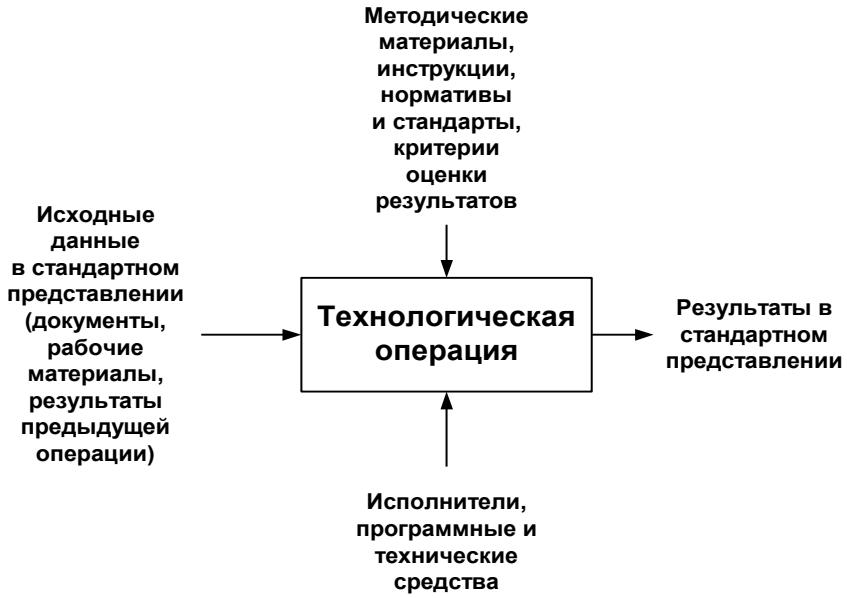
- подразумевает четко определенную ответственность роли;
- дает четко определенный результат (набор рабочих продуктов), базирующийся на определенных исходных данных (другом наборе рабочих продуктов);
- представляет собой единицу работы с жестко определенными границами, которые устанавливаются при планировании проекта.

Рабочий продукт – информационная или материальная сущность, которая создается, модифицируется или используется в некоторой технологической операции (модель, документ, код, тест и т.п.).

Роль – определение поведения и обязанностей отдельного лица или группы лиц в среде организации-разработчика ПО, осуществляющих деятельность в рамках некоторого технологического процесса и ответственных за определенные рабочие продукты.

Руководство – практическое руководство по выполнению одной или совокупности технологических операций. Руководства включают методические материалы, инструкции, нормативы, стандарты и критерии оценки качества рабочих продуктов.

Инструментальное средство (CASE-средство) – программное средство, обеспечивающее автоматизированную поддержку деятельности, выполняемой в рамках технологических операций.



Основным требованием, предъявляемым к современным ТС ПО, является их соответствие стандартам жизненного цикла (ЖЦ) ПО и оценкой технологической зрелости организаций-разработчиков (ISO 12207, ISO 9000, CMM и др.). Согласно этим нормативам, ТС ПО должна поддерживать полный набор процессов ЖЦ, к которым относятся:

- управление требованиями;
- анализ и проектирование ПО;
- разработка ПО;
- эксплуатация;

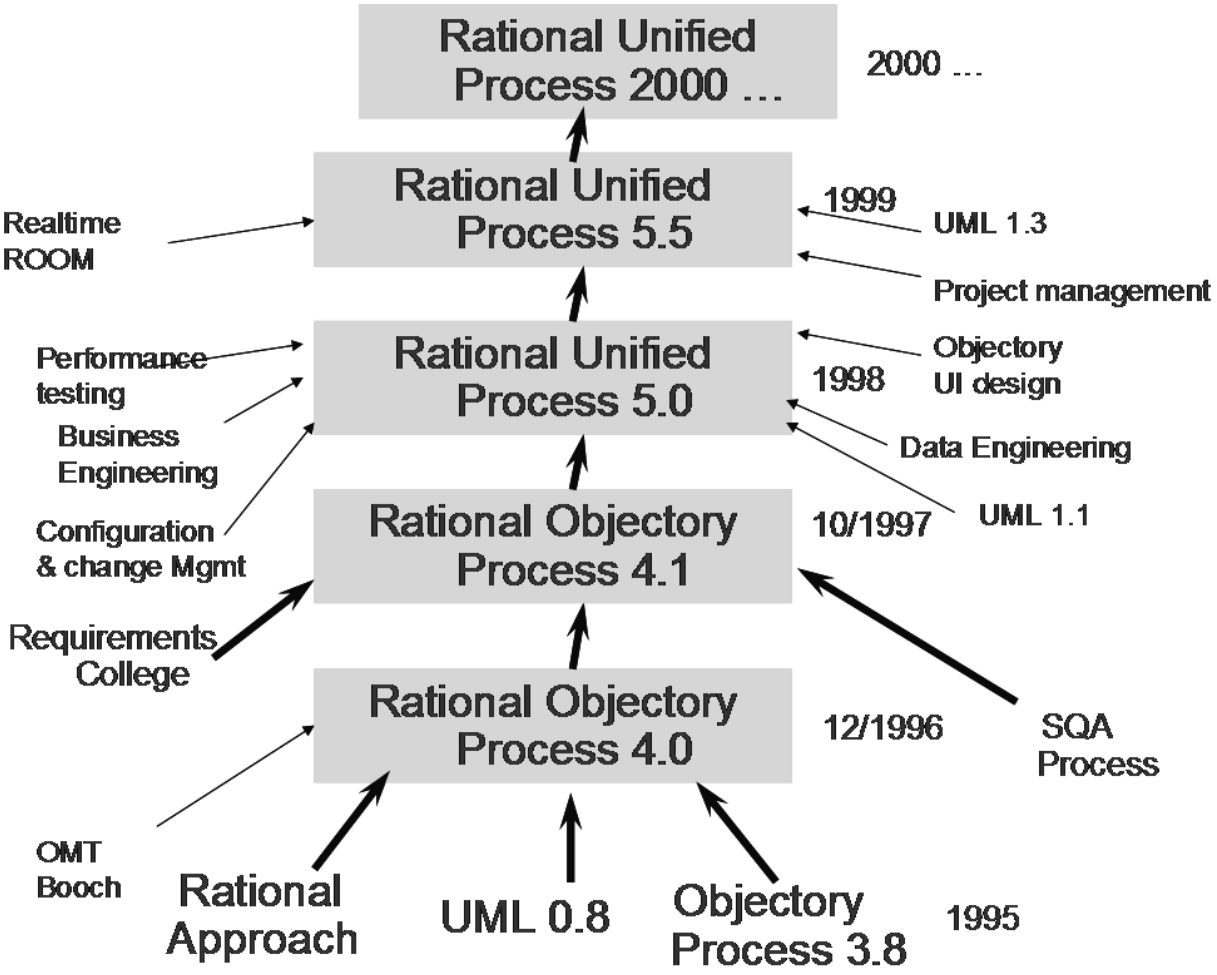
- сопровождение;
- документирование;
- управление конфигурацией и изменениями;
- тестирование;
- управление проектом.

Полнота поддержки процессов ЖЦ ПО должна поддерживаться комплексом инструментальных средств (CASE-средств). Также есть ряд других требований:

- адаптируемость к условиям применения;
- поддержка поставщика;
- простота использования;
- удовлетворительные стоимостные характеристики.

В качестве примера ТС ПО рассмотрим Rational Unified Process (RUP).

RUP является развитием процесса разработки, принятого в компании Ericsson в 70-х–80-х годах XX века. Эта модель была создана Джекобсоном (Ivar Jacobson), впоследствии, в 1987, основавшим собственную компанию Objectory AB именно для развития технологического процесса разработки ПО как отдельного продукта, который можно было бы переносить в другие организации. После вливания Objectory в Rational в 1995 разработки Джекобсона были интегрированы с работами Ройса (Walker Royce), Крачтена (Philippe Kruchten) и Буча (Grady Booch), а также с развивавшимся параллельно универсальным языком моделирования (Unified Modeling Language, UML).



RUP в значительной степени соответствует указанным выше требованиям. Ее основными принципами являются:

- *Раннее определение рисков и выработка ответных мер.* В начале каждой стадии ЖЦ составляется список рисков (примеры: неосвоенная СП, интеграция с унаследованным

кодом, орг. проблемы). По каждому риску из списка составляется перечень ответных мер. RUP учитывает изменчивость рисков – на разных этапах проекта списки рисков разные.

- *Выполнение требований заказчиков.* Требования описываются вариантами использования. Варианты использования – это отправная точка создания для модели анализа и проектной модели. Планирование и управление проектом ведется на основе вариантов использования. Варианты использования являются «сырьем» для тестов и пользовательской документации.
- *Концентрация на работающем коде.* Прогресс проекта оценивается по тому, какая часть системы готова и работает. Практика – лучший критерий проверки правильности того или иного решения. Все рабочие продукты проекта за исключением работающего кода являются вспомогательными, поэтому не следует слепо их создавать лишь из-за того, что они указаны в руководствах по RUP.
- *Готовность к изменениям с самого начала проекта и управление ими.* Система слишком сложна, чтобы с самого начала получить верное и окончательное проектное решение. Изменения позволяют улучшать принятые решения. Итеративный процесс позволяет исправлять дефекты, допущенные на ранних итерациях. Стоимость изменений в ходе проекта увеличивается. Управление изменениями позволяет уложиться в бюджет и сроки.
- *Сборка системы из компонентов.* Компонентная архитектура позволяет воспользоваться преимуществами инкапсуляции: повышает модифицируемость системы и возможности повторного использования ее частей. Стоимость разработки системы может быть снижена за счет использования компонентных платформ (J2EE, .NET).
- *Визуальное моделирование.* Графические модели более наглядны, удобны чем тексты на естественных и формальных языках. UML – стандартный язык, так что модели понятны большой аудитории (состоящей не только из людей, но и из CASE-средств), по той же причине имеется больше возможностей для повторного использования моделей.
- *Обеспечение качества в течение всего ЖЦ.* Используется упреждающее тестирование, лозунг которого: «Тесты раньше программ!» Регрессионное тестирование и первоочередная реализация приоритетных вариантов использования обеспечивают надежность ключевой функциональности системы. Качество создается в течение всего жизненного цикла за счет того, что все рабочие продукты критически оцениваются при рецензировании.

На рисунке показано общее представление RUP в двух измерениях:

- горизонтальное измерение представляет время, отражает динамические аспекты процессов и оперирует такими понятиями, как стадии, итерации и контрольные точки;
- вертикальное измерение отражает статические аспекты процессов и оперирует такими понятиями, как виды деятельности, рабочие продукты, исполнители и дисциплины.

Динамический аспект

Согласно технологии RUP, ЖЦ ПО разбивается на отдельные циклы, в каждом из которых создается новое поколение продукта. Каждый цикл, в свою очередь, разбивается на четыре последовательные стадии:

начальная стадия (inception);
стадия разработки (elaboration);
стадия конструирования (construction);
стадия ввода в действие (transition).

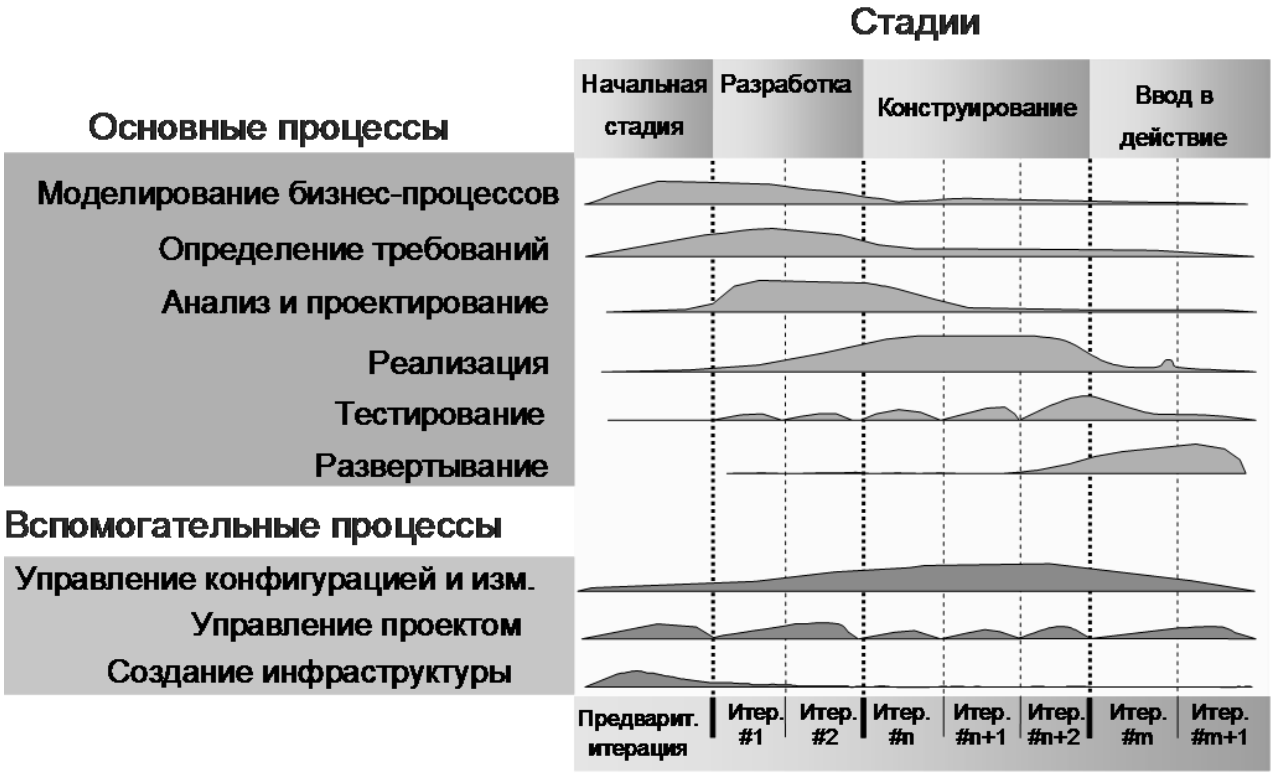


Рис. Общее представление RUP

Каждая стадия завершается в четко определенной контрольной точке (milestone). В этот момент времени должны достигаться важные результаты и приниматься критически важные решения о дальнейшей разработке.

Первыми двумя стадиями являются начальная стадия проекта и разработка. Во время начальной стадии вырабатывается бизнес-план проекта, определяется, сколько приблизительно он будет стоить и какой доход принесет. Определяются также границы проекта, и выполняется некоторый начальный анализ для оценки размеров проекта.

Результатами начальной стадии являются:

- общее описание системы: основные требования к проекту, его характеристики и ограничения;
- начальная модель вариантов использования (степень готовности - 10-20%);
- начальный проектный глоссарий (словарь терминов);
- начальный бизнес-план;
- план проекта, отражающий стадии и итерации;
- один или несколько прототипов.

На стадии разработки выявляются более детальные требования к системе, выполняется высокоуровневый анализ предметной области и проектирование для построения базовой архитектуры системы, создается план конструирования и устраняются наиболее рискованные элементы проекта.

Результатами стадии разработки являются:

- модель вариантов использования (завершенная по крайней мере на 80%), определяющая функциональные требования к системе;
- перечень дополнительных требований, включая требования нефункционального характера и требования, не связанные с конкретными вариантами использования;
- описание базовой архитектуры будущей системы;
- работающий прототип;
- уточненный бизнес-план;
- план разработки всего проекта, отражающий итерации и критерии оценки для каждой

итерации.

Стадия разработки занимает около пятой части общей продолжительности проекта.

RUP представляет собой итерационный и пошаговый процесс разработки, в котором программное обеспечение разрабатывается и реализуется по частям. На стадии конструирования построение системы выполняется путем серии итераций. Каждая итерация является своего рода мини-проектом. На каждой итерации для конкретных вариантов использования выполняются анализ, проектирование, кодирование, тестирование и интеграция («мини-каскад»). Итерация завершается демонстрацией результатов пользователям и выполнением системных тестов для контроля корректности реализации вариантов использования.

При итеративной разработке на каждой итерации выполняются все процессы, что позволяет оперативно справляться со всеми возникающими проблемами. Итерации на стадии конструирования являются одновременно инкрементными и повторяющимися. В конце каждой итерации должна выполняться полная интеграция. С другой стороны, интеграция может и должна выполняться еще чаще. Приложения следует интегрировать после выполнения каждой сколько-нибудь значительной части работы. Во время каждой интеграции должен выполняться полный набор тестов.

Главная особенность итерационной разработки заключается в том, что она жестко ограничена временными рамками, и сдвигать сроки недопустимо. Смысл таких ограничений – поддерживать строгую дисциплину разработки и не допускать переноса сроков.

Результатом стадии конструирования является продукт, готовый к передаче конечным пользователям. Как минимум, он содержит следующее:

- ПО, интегрированное на требуемых платформах;
- руководства пользователя;
- описание текущей реализации.

Назначением стадии ввода в действие является передача готового продукта в распоряжение пользователей. Данная стадия включает:

- бета-тестирование, позволяющее убедиться, что новая система соответствует ожиданиям пользователей;
- параллельное функционирование с существующей (legacy) системой, которая подлежит постепенной замене;
- конвертирование баз данных;
- оптимизацию производительности;
- обучение пользователей и специалистов службы сопровождения.

На стадии ввода в действие продукт не дополняется никакой новой функциональностью (кроме самого необходимого минимума). Во время нее только вылавливаются ошибки и шлифуется качество. Хорошим примером для стадии ввода в действие может служить период времени между выпуском бета-версии и окончательной версии продукта.

Статический аспект

Статический аспект RUP представлен четырьмя основными элементами:

- роли;
- виды работ;
- рабочие продукты;
- дисциплины (процессы).

Понятие «роль» (role) определяет поведение и ответственность личности или группы личностей, составляющих проектную команду. Одна личность может играть в проекте много различных ролей.

Видом работы конкретного исполнителя понимается элементарная работа – технологическая опреация, выполняемая им.

Дисциплина (discipline) соответствует понятию технологического процесса и

представляет собой последовательность работ, приводящую к получению значимого результата.

В рамках RUP определены шесть основных дисциплин:

- построение бизнес-моделей;
- определение требований;
- анализ и проектирование;
- реализация;
- тестирование;
- развертывание;

и три вспомогательных:

- управление конфигурацией и изменениями;
- управление проектом;
- создание инфраструктуры.

По большей части, содержание основных дисциплин мы рассмотрели на предыдущих лекциях. Повторим кратко.

Задачи построения бизнес-моделей – понять предметную область или бизнес-контекст, в которых должна будет работать система, и убедиться, что все заинтересованные лица понимают его одинаково, осознать имеющиеся проблемы, оценить их возможные решения и их последствия для бизнеса организации, в которой будет работать система. В рамках этой дисциплины создаются следующие рабочие продукты:

- видение бизнеса;
- глоссарий деятельности предприятия;
- модель бизнес-процессов (описывающая контекст бизнеса и бизнес-процессы предприятия на диаграмме вариантов использования);
- модель бизнес-анализа (описывающая протекание бизнес-процессов, т. е. их реализацию, на диаграммах взаимодействия, участниками которых являются исполнители и бизнес-сущности, а также на диаграммах деятельности);
- описания бизнес-целей, бизнес-правил и бизнес-событий;
- дополнительная спецификация бизнеса.

Полученные модели служат основой для моделей требований и анализа. Подробно дисциплина рассматривалась на лекции «Моделирование бизнес-процессов».

Задачи определения требований – понять, что должна делать система, и убедиться во взаимопонимании по этому поводу между заинтересованными лицами, определить границы системы и основу для планирования проекта и оценок затрат ресурсов в нем.

Требования принято фиксировать в виде модели вариантов использования и различных документов: описаний вариантов использования, концепции системы, глоссария системы, дополнительной спецификации, отражающей нефункциональные требования. Подробно дисциплина рассматривалась на лекции «Требования к программному обеспечению».

Задачи анализа и проектирования – выработать архитектуру системы на основе требований, убедиться, что данная архитектура может быть основой работающей системы в контексте ее будущего использования. В результате должна появиться модель проектирования, включающая в себя диаграммы классов системы, диаграммы ее компонентов, диаграммы взаимодействия между объектами в ходе реализации вариантов использования, диаграммы состояний для отдельных объектов и диаграммы развертывания, а также модель развертывания и документ – описание архитектуры. Подробное рассмотрение дисциплины см. в лекциях 6, 7 и 9.

Дисциплина реализации решает следующие задачи – определить структуру исходного кода системы, разработать код ее компонентов и протестировать их, интегрировать систему в работающее целое. Она включает в себя:

- Реализацию архитектуры (переход от проектной модели к модели реализации,

представленной в виде диаграмм компонент и диаграмм пакетов).

- Выработку плана сборки для каждой итерации.
- Распределение компонентов системы по узлам вычислительной среды.
- Реализацию кода классов и подсистем.
- Покомпонентное тестирование.

Реализацию архитектуры осуществляет архитектор. Заключается она в трассировке проектных классов, пакетов и подсистем в компоненты и установлении связей (зависимостей) между компонентами.

План сборки описывает функциональность, которая должна быть реализована в билде (сборке) и те компоненты, которые входят в билд. Планы составляет системный интегратор.

За реализацию кода отвечает инженер по компонентам.

Покомпонентное тестирование – это раздельное тестирование компонент системы. Осуществляет его инженер по компонентам путем тестирования спецификации («черный ящик») и тестирования структуры («белый ящик»).

Дисциплина тестирования решает следующие задачи – найти и описать дефекты системы (проявления недостатков ее качества), оценить ее качество в целом, оценить выполнены или нет гипотезы, лежащие в основе проектирования, оценить степень соответствия системы требованиям. Она включает в себя:

- Планирование тестов на каждой итерации.
- Составление тестовых вариантов (test-case) и тестовых сценариев (test scripts).
- Тестирование с целью обнаружения дефектов.

Тестовый вариант включает входные данные, условия выполнения отдельных шагов и корректные ответы системы для всякого шага, на котором ответ системы можно наблюдать. С вариантом тестирования связан один или более тестовых сценариев.

Тестовый сценарий – способ выполнения одного или нескольких тестовых наборов в рамках тестового варианта. Выполняется вручную или автоматически.

Тестовые варианты и сценарии разрабатываются инженерами по тестированию. Некоторые тестовые варианты предназначены для интеграционного тестирования, они проверяют целостность сборки, т. е. правильность взаимодействия компонент, вошедших в сборку. За тестирование целостности отвечает тестировщик целостности. Другие тестовые варианты используются при системном тестировании – проверке правильности работы системы в целом. За него отвечает системный тестировщик. Помимо этого применяются другие виды тестирования:

- регрессионное (при котором новые сборки проверяются на тестах для предыдущих сборок, поскольку при интеграции новых компонент может быть нарушено правильное функционирование старых и системы в целом);
- инсталляционное (проверка возможности установки системы на вычислительной среде и правильность работы инсталлятора);
- конфигурационное (проверка работы системы в разных конфигурациях);
- отрицательное (проверка устойчивости системы на заведомо неверных данных, при неверных действиях пользователя – «защита от дурака» – при недостаточных ресурсах);
- нагрузочное (проверка нефункциональных свойств, например, производительности, пропускной способности).

Дисциплина развертывания решает следующие задачи – установить систему в ее рабочем окружении и оценить ее работоспособность на том месте, где она должна будет работать.

Управление конфигурациями и изменениями решает следующие задачи – определение элементов, подлежащих хранению в репозитории проекта и правил построения из них согласованных конфигураций, поддержание целостности текущего состояния системы, проверка согласованности вносимых изменений.

Управление проектом решает следующие задачи – планирование, управление персоналом, обеспечение взаимодействия на благо проекта между всеми заинтересованными лицами, управление рисками, отслеживание текущего состояния проекта.

Создание инфраструктуры решает следующие задачи – подстройка процесса под конкретный проект, выбор и замена технологических инструментов, используемых в проекте.

RUP опирается на интегрированный комплекс инструментальных средств Rational Suite, обеспечивающий поддержку полного жизненного цикла ПО.