Классификации методологий проектирования программных систем

В последние годы наблюдается широкомасштабное применением вычислительной техники в различных сферах деятельности человека. Поэтому наиболее остро встала проблема разработки программных систем, автоматизирующих его деятельность с целью повышения ее эффективности.

Создание современных программных систем невозможно без использования методологий и технологий их проектирования, поскольку их разработка представляет длительный, трудоемкий и наукоемкий процесс, требующий больших материальных и финансовых затрат. Думается, здесь уместно вспомнить толкование слова «Технология»: «Совокупность производственных методов и процессов в определенной отрасли производства, а также научное описание способов производства». Таким образом, в области создания программного обеспечения под термином «Технология» понимается совокупность методов и процессов создания программных систем. Из-за большой сложности процесса разработки программных систем методы их проектирования выделяются от технологий их создания, образуя методологии разработки программных систем. Поэтому в дальнейшем изложении они будут рассматриваться отдельно.

Создано несколько классификаций методологий и технологий разработки программных систем. В соответствии с одной из них они разделяются в зависимости от научно-практических направлений, в рамках которых возникли. Как было отмечено во введении, к настоящему времени сформировались следующие научно-практические направления, занимающиеся вопросами создания методологий и технологий проектирования программных систем:

- ***информационная инженерия (Information Engineering);***
- ***искусственный интеллект (Artificial Intelligence);***
- ***обратное перепроектирование (Re-engineering);***
- ***реинжиниринг бизнес-процессов (Business Process Reengineering);***
- ***многоагентные системы (Multi-Agent Systems);***
- ***управление знаниями (Knowledge Management);***
- ***промышленная инженерия (Industrial Engineering);***
- ***управление качеством (Total Quality Management).***

- Приведем их краткую характеристику. В середине 90-х годов в методологиях и технологиях разработки программных систем произошла смена одной из парадигм: программная инженерия (Software Engineering) сменилась на информационную инженерию (Information Engineering). Информационная инженерия представляет собой совокупность методологий и программных инструментальных средств, поддерживающих создание программных систем, автоматизирующих деятельность человека. Главной отличительной особенностью информационной инженерии от программной инженерии является наличие методов и программных инструментальных средств, поддерживающих этап стратегического планирования жизненного цикла программного обеспечения. На этапе стратегического планирования осуществляется обследование деятельности организации с целью повышения эффективности труда сотрудников. Для этого анализ организации проводится на трех уровнях: макроуровне, микроуровне и уровне организации. На макроуровне осуществляется анализ политической обстановки, экономического положения и технической политики, которые влияют на выбор сфер деятельности организации. На микроуровне осуществляется анализ рыночных отношений между организацией, потребителями и конкурентами. На уровне организации осуществляется анализ внутреннего состояния организации: организационная структура, производство продукции, финансовое положение, профессионализм кадров и т.п. Стратегический анализ деятельности организации представляет трудоемкий длительный процесс, цена ошибок на котором очень высока, поскольку они могут привести к краху организации.

- Возникшие в рамках программной инженерии CASE-технологии (Computer-Aided Software Engineering) позволяют значительно сократить время проектирования программных систем, повысить их качество, а также значительно уменьшить затраты на их создание. Научно-практическое направление «CASE-технологии» занимается вопросами создания методологий проектирования программных систем и программных инструментальных средств их поддержки (CASE-средств). Наиболее широко применяемыми методологиями являются представители двух классов методологий: структурных и объектно-ориентированных. Структурные методы проектирования программных систем появились в конце 70-х годов. Их создание связано с именами Йордона, Джэксена, Якобсона и многими другими. Следует отметить, что Yordon является разработчиком первого CASE-средства CASE\Analyst\Design , созданного в 1979 г., в котором на основе построенных диаграмм генерируется программный код на языке Ада. К настоящему времени создан многочисленный ряд CASE-средств, поддерживающих структурные методы. К нему относятся CASE-средства: BPWin, Idef, Silverrun и многие другие.
- В последние годы наиболее перспективной методологией для создания программных систем признана объектно-ориентированная методология, одним из основоположников которой является Г. Буч. CASE-средства, поддерживающие эту методологию, широко применяются в фирмах в США, Японии и в других ведущих странах мира. В России широко используемыми CASE-средствами, поддерживающими объектно-ориентированные методы, являются Rational Rose Enterprise Edition 2000/2002 (фирмы Rational Software Corporation), Oracle Developer Suite 2000 (фирмы Oracle) и др. Следует заметить, что Г. Буч и Дж. Рамбоух являются создателями языка UML (Unified Modeling Language), появившегося в свет в 1994 г., который положен в основу CASE-средства Rational Rose Enterprise Edition, сотрудником фирмы, его разработавшей, в настоящее время является Г. Буч.

- Перспективным подходом к созданию CASE-средств является применение в них интеллектуальных методов. Необходимость их применения для создания программных систем обусловлена следующими факторами. На этапах создания программных систем разработчикам приходится работать с неполной, нечеткой, неточной, противоречивой информацией. Созданные CASE-средства с использованием интеллектуальных методов позволяют преодолеть трудности, возникающие у разработчиков программных систем, более быстрыми темпами и с меньшими трудозатратами. Бурный рост работ, касающихся вопросов использования интеллектуальных методов в CASE-средствах, стал основой для появления научно-практического направления в рамках информационной инженерии, получившего название «Knowledge Based Software Engineering (KBSE)».
- Одним из основных результатов научно-практического направления, занимающего вопросами создания интеллектуальных систем (ИС), являются интеллектуальные методологии и технологии их проектирования. Если в период становления этого направления считалось, что такие методологии и технологии базируются на моделях представления знаний и механизмах вывода в них, то в последние годы наблюдается тенденция к их интеграции. Отметим, что наиболее широко используемыми методологиями являются методологии проектирования ИС, основанные на моделях представления знаний, мягкие вычисления, объединившие нечеткую логику, нейротехнологии и генетические алгоритмы, а также, следует упомянуть о набирающем силу синергетическом подходе к разработке ИС.
- Методы обратного перепроектирования и программные инструментальные средства их поддержки направлены на оптимизацию характеристик созданных программных систем и обеспечение их «стыковки» с существующими в организации программными системами.

- Одновременно с информационной инженерией появляется направление менеджмента, получившего название – реинжиниринг бизнес-процессов. Ключевые моменты этого направления будут рассмотрены при приведении 2-й классификации методологий и технологий проектирования программных систем, под влиянием которого она и сформировалась.
- Многоагентные методологии и технологии разработки программных систем, завоевавшие популярность в конце 90-х годов, признаны, как одни из наиболее перспективных.
- Еще одни из наиболее перспективных методологий и технологий создания программных систем являются методологии и технологии их построения как систем управления знаниями (СУЗ), где под управлением знаниями понимается методология, включающая в себя комплекс формальных методов, охватывающих:

- ***поиск и извлечение знаний из живых и неживых объектов (носителей знаний);***
- ***структурирование и систематизацию знаний (для обеспечения их удобного хранения и поиска);***
- ***анализ знаний (выявление зависимостей и аналогий);***
- ***обновление (актуализацию) знаний;***
- ***распространение знаний;***
- ***генерирование новых знаний.***

- Промышленная инженерия, возникшая в середине XX века, занимается управлением и организацией производства. В ней наиболее широко применяемыми методологиями и технологиями являются JIT (Just-in-time – точно вовремя), OPT (Optimised Production Technology – оптимизационная технология производства), CIM (Computer Integrated Manufacturing – интегрированные производства на основе вычислительной техники), CALS (Continuous Acquisition and Life Circle Support – поддержка непрерывного жизненного цикла продукции), ERP, MRP (Material Requirements Planning – планирование потребностей в материалах), MRP II (Manufacturing Recourse Planning – планирование ресурсов производства), CAD/CAM/CAE и т.д. Разработан и внедрен ряд ERP-систем.
- Технология разработки систем качества (Total Quality Management) базируются на концепции управления качеством, документированной в стандартах ISO 9000. Следует отметить, что стандарт ISO 9000 представляет собой серию стандартов 9000, 9001, 9002, 9003, 9004, в которой ISO 9001 является наиболее полным стандартом, специфицирующим модель обеспечения качества на всех этапах жизненного цикла товара/услуги.

- Следует отметить, что в последние годы стираются границы между выделенными классами методологий, поскольку осуществляется проникновение методов проектирования программных систем одних методологий в другие. Так, например, интеллектуальные методы применяются в информационной инженерии (разработка CASE-систем, основанных на знаниях), в реинжиниринге бизнес-процессов (использование моделей рассуждений, основанной на прецедентах (Case-based reasoning)), в многоагентных технологиях (проектирование интеллектуальных агентов). Без их использования невозможно создание систем управления знаниями.
- Вторая классификация методологий и технологий создания программных систем, как уже упоминалось, сложилась под влиянием реинжиниринга бизнес-процессов (БПР).
- В настоящее время достаточно большое число организаций, включая университеты, имеют организационные формы и процедуры функционирования, не обеспечивающие конкурентноспособности организации и не способствующие ее развитию. На основе методов БПР они должны быть кардинальным образом перестроены. Хаммер и Чампли, изобретатели термина БПР, определили БПР как «фундаментальное переосмысление и радикальное перепроектирование деловых процессов для достижения резких/скачкообразных улучшений в решающих, современных показателях деятельности компании, таких как стоимость, качество, сервис и темпы». БПР – есть совокупность методов и программных инструментальных средств, предназначенных для кардинального улучшения основных показателей деятельности компании, путем моделирования, анализа и перепроектирования существующих бизнес-процессов.

- Перепроектирование процессов становится возможным, как правило, благодаря использованию программных инструментальных средств, поддерживающих методологии реинжиниринга бизнес-процессов , а также применению CASE-технологий.
- В соответствии со второй классификацией методологии и технологии проектирования программных систем делятся на следующие группы:

❖ поддерживающие;
❖ развивающие;
❖ принципиально новые.

- На основе технологий, относящихся к 1-й и 2-й группам, создаются бизнес-поддерживающие программные системы, которые поддерживают и развивают бизнес-процессы, существующие в организации, используя методологии реинжиниринга бизнес-процессов. На основе технологий, относящихся к 3-й группе, создаются бизнес-образующие программные системы и проекты, поддерживающие новые бизнес-процессы и новый бизнес, используя методологии инжиниринга бизнес-процессов.
- Выделение целей организации и проверка их на конкурентноспособность, выполняемые на этапе обследования организации, являются одинаковыми при создании программных систем на основе методов информационной инженерии и реинжиниринга бизнес-процессов. Следует отметить, что от правильного выполнения этих работ зависит не только успех создания программных систем, но и эффективность функционирования организации.
- Остановимся на рассмотрении одних из перспективных методологий и технологий создания программных систем – интеллектуальных методологий и технологий, методы проектирования которых, как отмечалось выше, проникают во все методологии проектирования таких систем, увеличивая конкурентноспособность систем, построенных на их основе.
- Также свидетельствует о необходимости создания интеллектуальных методологий и технологий резкое увеличение сложности задач, для решений которых создаются ИС, помогающие человеку в его мыслительной деятельности.

- Одним из важных классов ИС, применяемых в различных сферах деятельности человека, являются интеллектуальные системы поддержки принятия решений (ИСППР), помогающие человеку при управлении им сложными системами и процессами. Перед разработчиками ИСППР на начальных этапах их создания встает задача моделирования поведения сложных объектов управления (ОУ). При решении этой задачи разрабатывается модель функционирования ОУ и осуществляется извлечение знаний из экспертов и лиц, принимающих решение, (ЛПР), и формализация нормативных документов по его управлению, которые представляются в моделях принятия решений. Построение таких моделей представляет собой трудоемкий и наукоемкий процесс, который не всегда заканчивается успешно, поскольку их создание происходит обычно в условиях невозможности разработки математической модели ОУ из-за неполной, противоречивой, нечеткой информации, выделяемой при описании реальных ОУ.
- Следует отметить, что сбор статистических данных, описывающих ОУ, представляет длительный процесс или иногда невозможен из-за сложности проведения измерений характеристик ОУ. Поэтому одним из перспективных подходов к решению задачи моделирования поведения сложных объектов является использование интеллектуальных методов. В последние годы наблюдается резкий рост количества работ, касающихся логического подхода к решению этой задачи. Большой вклад в это научное направление внесли и вносят зарубежные логики, среди которых можно выделить Дж. Маккарти, первые результаты которого была получены в 1963 г.. Большой вклад в это научное направление вносит Р. Рейтер, который разработал логический фундамент для моделирования динамических систем, создал язык логического программирования GOLOG (Algol in Logic), реализовал различные версии этого языка: последовательный GOLOG, параллельный GOLOG, временной GOLOG, реактивный GOLOG. Семантика языка GOLOG описывается исчислением предикатов второго порядка. Синтаксис языка напоминает синтаксис языка Prolog. Также здесь можно упомянуть Р. Миллера и М. Шэхэнэн, создавшие исчисление событий, и многих других зарубежных логиков.
- Российские ученые также вносят большой вклад в создание логических методов моделирования сложных объектов, вершиной работ в котором стал подход семиотического моделирования, предложенный Д.А.Поспеловым и предоставляющий математический базис для построения ИС качественно нового уровня. Так, на смену формальной системы и ее частичных модификаций приходит семиотическая система, являющаяся фундаментом для создания семиотических моделей, позволяющих адекватно описывать современные сложные проблемные среды.

Причины возникновения ошибок при разработке программных средств

- Первым шагом в разработке программных средств (ПС) является определение того, что необходимо сделать. Заказчик , используя накопленные знания, сообщает аналитику первичный набор требований к ПС. Аналитик оценивает требования, и руководитель проекта принимает решение о его разработке. Встает проблема: почему разработанные ПС не соответствуют требованием заказчика. Существуют четыре важные причины, возникновения этой проблемы.
- 1. Заказчик не может сформулировать требования.
- 2. Заказчик не может корректно или точно сформулировать требования.
- 3. Аналитик не совсем правильно понимает сообщаемые требования.
- 4. Требования постоянно изменяются.

Причины возникновения ошибок при разработке программных средств

- Поэтому процесс выработки требования из-за его огромной сложности не может быть автоматизирован.
- Существуют следующие каналы передачи требований заказчиком аналитику:
- речь;
- текст;
- диаграммы и модели.
- Существует четыре источника «шума», возникающие из-за плохого пересечения знаний заказчика и аналитика.
- «Я не знаю, какой это способ».
- «Это означает одно из или оба».
- Это не является верным во всех случаях, поэтому я не понимаю».
- «Я слышу, что вы говорите, то я совсем не понимаю».

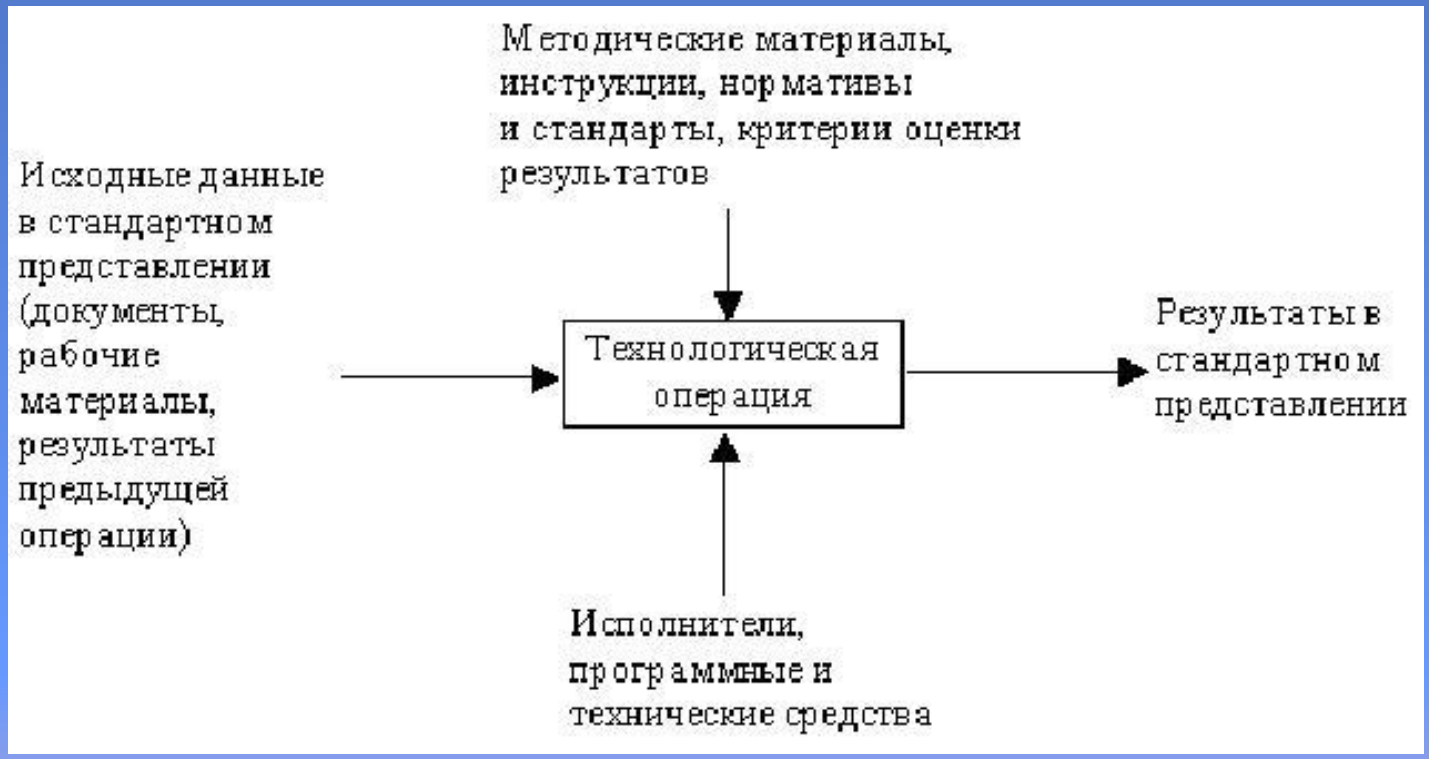
Причины возникновения ошибок при разработке программных средств

- Ошибки проектирования и кодирования
- Эти ошибки классифицируются в четыре группы:
- 1) синтаксические (неправильное написание конструкций диаграмм или языка программирования);
- 2) семантические, возникающие вследствие неправильного понимания того что нужно сделать;
- 3) логические, приводящие к логически некорректному поведению ПС, например, неправильная запись логических условий;
- 4) арифметические, связанные с арифметическими операциями, например деление на нуль.

Общие требования к методологии и технологии

- Методологии, технологии и инструментальные средства проектирования (CASE-средства) составляют основу проекта любой ИС. Методология реализуется через конкретные технологии и поддерживающие их стандарты, методики и инструментальные средства, которые обеспечивают выполнение процессов ЖЦ.
- Технология проектирования определяется как совокупность трех составляющих:
- пошаговой процедуры, определяющей последовательность технологических операций проектирования (рис. 4);
- критериев и правил, используемых для оценки результатов выполнения технологических операций;
- нотаций (графических и текстовых средств), используемых для описания проектируемой системы.

Общие требования к методологии и технологии



- **Рис.4. Представление технологической операции проектирования**

Общие требования к методологии и технологии

- Технологические инструкции, составляющие основное содержание технологии, должны состоять из описания последовательности технологических операций, условий, в зависимости от которых выполняется та или иная операция, и описаний самих операций.
- Технология проектирования, разработки и сопровождения ИС должна удовлетворять следующим общим требованиям:
- технология должна поддерживать полный ЖЦ ПО;
- технология должна обеспечивать гарантированное достижение целей разработки ИС с заданным качеством и в установленное время;

Общие требования к методологии и технологии

- технология должна обеспечивать возможность выполнения крупных проектов в виде подсистем (т.е. возможность декомпозиции проекта на составные части, разрабатываемые группами исполнителей ограниченной численности с последующей интеграцией составных частей). Опыт разработки крупных ИС показывает, что для повышения эффективности работ необходимо разбить проект на отдельные слабо связанные по данным и функциям подсистемы. Реализация подсистем должна выполняться отдельными группами специалистов. При этом необходимо обеспечить координацию ведения общего проекта и исключить дублирование результатов работ каждой проектной группы, которое может возникнуть в силу наличия общих данных и функций;
- технология должна обеспечивать возможность ведения работ по проектированию отдельных подсистем небольшими группами (3-7 человек). Это обусловлено принципами управляемости коллектива и повышения производительности за счет минимизации числа внешних связей;

Общие требования к методологии и технологии

- технология должна обеспечивать минимальное время получения работоспособной ИС. Речь идет не о сроках готовности всей ИС, а о сроках реализации отдельных подсистем. Реализация ИС в целом в короткие сроки может потребовать привлечения большого числа разработчиков, при этом эффект может оказаться ниже, чем при реализации в более короткие сроки отдельных подсистем меньшим числом разработчиков. Практика показывает, что даже при наличии полностью завершенного проекта, внедрение идет последовательно по отдельным подсистемам;
- технология должна предусматривать возможность управления конфигурацией проекта, ведения версий проекта и его составляющих, возможность автоматического выпуска проектной документации и синхронизацию ее версий с версиями проекта;

Общие требования к методологии и технологии

- технология должна обеспечивать независимость выполняемых проектных решений от средств реализации ИС (систем управления базами данных (СУБД), операционных систем, языков и систем программирования);
- технология должна быть поддержана комплексом согласованных CASE-средств, обеспечивающих автоматизацию процессов, выполняемых на всех стадиях ЖЦ.

Общие требования к методологии и технологии

- Реальное применение любой технологии проектирования, разработки и сопровождения ИС в конкретной организации и конкретном проекте невозможно без выработки ряда стандартов (правил, соглашений), которые должны соблюдаться всеми участниками проекта. К таким стандартам относятся следующие:
- стандарт проектирования;
- стандарт оформления проектной документации;
- стандарт пользовательского интерфейса.

Общие требования к методологии и технологии

- Стандарт проектирования должен устанавливать:
- набор необходимых моделей (диаграмм) на каждой стадии проектирования и степень их детализации;
- правила фиксации проектных решений на диаграммах, в том числе: правила именования объектов (включая соглашения по терминологии), набор атрибутов для всех объектов и правила их заполнения на каждой стадии, правила оформления диаграмм, включая требования к форме и размерам объектов, и т. д.;
- требования к конфигурации рабочих мест разработчиков, включая настройки операционной системы, настройки CASE-средств, общие настройки проекта и т. д.;

Общие требования к методологии и технологии

- механизм обеспечения совместной работы над проектом, в том числе: правила интеграции подсистем проекта, правила поддержания проекта в одинаковом для всех разработчиков состоянии (регламент обмена проектной информацией, механизм фиксации общих объектов и т.д.), правила проверки проектных решений на непротиворечивость и т. д.

Общие требования к методологии и технологии

- Стандарт оформления проектной документации должен устанавливать:
- комплектность, состав и структуру документации на каждой стадии проектирования;
- требования к ее оформлению (включая требования к содержанию разделов, подразделов, пунктов, таблиц и т.д.),
- правила подготовки, рассмотрения, согласования и утверждения документации с указанием предельных сроков для каждой стадии;

Общие требования к методологии и технологии

- требования к настройке издательской системы, используемой в качестве встроенного средства подготовки документации;
- требования к настройке CASE-средств для обеспечения подготовки документации в соответствии с установленными требованиями.

Общие требования к методологии и технологии

- Стандарт интерфейса пользователя должен устанавливать:
- правила оформления экранов (шрифты и цветовая палитра), состав и расположение окон и элементов управления;
- правила использования клавиатуры и мыши;
- правила оформления текстов помощи;
- перечень стандартных сообщений;
- правила обработки реакции пользователя.

CASE-технология

- CASE-технология представляет собой методологию проектирования ИС, а также набор инструментальных средств, позволяющих в наглядной форме моделировать предметную область, анализировать эту модель не всех этапах разработки и сопровождения ИС и разрабатывать приложения в соответствии с информационными потребностями пользователей.

CASE-технология

- Широкое использование вычислительной техники в различных сферах деятельности человека привело к потребности создания соответствующего программного обеспечения (ПО). Однако трудоемкость и наукоемкость разработки программ настолько огромны, что в настоящее время ведутся работы по созданию новых технологий автоматизации проектирования программных средств. Это направление получило название CASE (Computer-Aided Software Engineering).
- Одна из важных особенностей CASE-средств состоит в автоматизации всех этапов жизненного цикла ПО и прежде всего начальнах, в отделении проектирования ПО от кодирования и последующих операций разработки. CASE есть комбинация программных средств и методологий разработки ПО.
- Если в середине 80-х годов, CASE системы включали только средства анализа и документирования, то в последние годы CASE включают функциональные средства, обеспечивающие поддержку всего жизненного цикла разработки ПО.

Классификация CASE-средств

- В настоящее время существует два популярных механизма классификации CASE-средств, которые представлены на рис.1.
- В соответствии с одной из них, так называемой "вертикальной" классификацией, CASE-средства делятся на верхние CASE, которые поддерживают этапы стратегического планирования, анализа и проектирования в жизненном цикле ПО; и нижние CASE, которые поддерживают этапы программирования и тестирования.
- В противоположность "вертикальной" классификации, в которой CASE-средства обеспечивают поддержку специальных стадий в жизненном цикле, выделяют "горизонтальные" CASE средства жизненного цикла, которые обеспечивают автоматизированную поддержку всего процесса разработки ПО и его сопровождение.
- Основой интеграции в CASE системах является репозиторий (информационная база проекта), в котором сосредоточена информация о создаваемом ПО на всех стадиях от технического задания до сопровождения. При этом репозиторий должен обеспечивать хранение не только самих проектируемых объектов, но и версий этих объектов или вносимых изменений. В зависимости от уровня автоматизации и степени охвата ею этапов разработки в репозитории хранятся тексты программ, спецификации требований, различные текстово-графические представления проекта, комментарий к нему, а также проектная и программная документация.

Классификация CASE-средств

CASE Workbench или Environment

- Таким образом, CASE-средства автоматизируют проектирование ПО как на его отдельных стадиях разработки, так и обеспечивают автоматизированную поддержку всего процесса разработки ПО и его сопровождения.
- Различают два поколения CASE-средств. К первому поколению относятся "верхние" и "нижние" CASE-средства (отдельные "инструменты"), которые предназначены для автоматизации создания ПО на его отдельных стадиях разработки. CASE-средства первого поколения называют CASE Toolkit (инструментарий).
- К второму поколению относятся "горизонтальные" (интегрированные) CASE-средства, предназначенные для автоматизации всего жизненного цикла ПО. Эти CASE-средства называют CASE Workbench или Environment (окружение).
- CASE Workbench или environment (окружение) есть набор интегрированных программных средств, представленных на рис.2

CASE Workbench

Отличительные черты CASE Workbench от CASE Toolkit

- Эти средства обеспечивают автоматизированную поддержку всех стадий жизненного цикла разработки ПО. Отличительными чертами CASE Workbench от CASE Toolkit являются следующие:
- ***единая методология;***
- ***общий репозиторий (информационная база проекта), содержащий всю техническую информацию и информацию управления проектом, необходимую для построения и сопровождения программной системы;***
- ***автоматическое прохождение информации о программной системе от одной стадии разработки к другой;***
- ***единый пользовательский интерфейс.***
- Рассмотрим компоненты интегрированных CASE средств более детально.

Диаграммные средства

- Диаграммные средства поддерживают стадию анализа в жизненном цикле разработки ПО. На этой стадии жизненного цикла разработки ПО используются различные типы диаграмм, такие как диаграммы “потоков-данных”, которые показывают течение данных среди процессов в разрабатываемой системе, т.е. для информационной системы: где данные определяются, куда передаются и т.д.; диаграммы “сущность-связь”, которые описывают структуру предметной области; диаграммы “состояние-переход”, используемые для создания систем реального времени, и другие. Диаграммеры CASE средств обеспечивают автоматическую поддержку создания этих диаграмм, структурных схем и других графиков.

Синтаксический верификатор

- Синтаксический верификатор выполняет автоматический синтаксический контроль за созданными диаграммами. Например, диаграммы “потоков данных” требуют, чтобы процессы имели как входы, так и выходы. Эти средства, на которые часто ссылаются как на анализаторы разработки, выполняют проверку на непротиворечивость, проверку уровня сбалансированности диаграмм “потоков данных” и другую обработку ошибок.

Центральный репозиторий

- Ядром CASE окружения является центральный репозиторий или информационный репозиторий. Центральный репозиторий есть больше чем словарь данных, т.к. в нем хранится разнообразная информация, связанная с разрабатываемой системой. Репозиторий становится связующим звеном между всеми разработчиками программной системы, а также между всеми компонентами CASE-окружения.

Средства прототипирования

- Средства прототипирования позволяют создать быстрый прототип разрабатываемой системы и его модифицировать. Они используют определения данных, хранящихся в центральном репозитории, чтобы определить входные/выходные файлы.

Генераторы кода

- Генераторы кода позволяют создать модульный код из спецификаций, заданных на языке высокого уровня. Генераторы кода могут использоваться как отдельные средства. Как часть CASE-окружения, они интегрированы с другими компонентами.

Управление проектом и средства поддержки методологии

- Средства управления проектом используются руководителями проекта, чтобы успешно выполнить разработку и управление ресурсами. Работа CASE-средств подчиняется стандартам, которые устанавливаются методологией.

Обратное перепроектирование (Re-engineering)

- Обратное перепроектирование (Re-engineering) - это анализ готового ПО с целью устранения ошибок и, главное, оптимизации его характеристик. Эти средства делятся на две категории: в одну из которых входят средства, изменяющие программный код готового ПО, а в другую - средства, создающие структурные схемы, словари данных и другую информацию для существующих систем. Таким образом, они обеспечивают возможность полной интеграции существующих систем с новыми системами.
- CASE технология обращена на продуктивную разработку информационных систем, систем реального времени и систем для научных приложений. В соответствии с этим все CASE-средства можно разбить на следующие три большие группы, в зависимости от класса задач для решения которого они предназначены:
- ***CASE-средства проектирования информационных систем;***
- ***CASE-средства проектирования научных приложений;***
- ***CASE-средства проектирования систем реального времени.***

Классификация CASE-средств

- В соответствие с еще одной классификацией CASE-средств все современные CASE-средства могут быть классифицированы по типам и категориям. Классификация по типам отражает функциональную ориентацию CASE-средств на те или иные процессы ЖЦ. Им соответствуют CASE Toolkit по 1-й классификации. Классификация по категориям определяет степень интегрированности по выполняемым функциям и включает отдельные локальные средства, решающие небольшие автономные задачи (tools), набор частично интегрированных средств (CASE Workbench по 1-й классификации), охватывающих большинство этапов жизненного цикла ИС (toolkit) и полностью интегрированные средства, поддерживающие весь ЖЦ ИС и связанные общим репозиторием. Помимо этого, CASE-средства можно классифицировать по следующим признакам:
- - применяемым методологиям и моделям систем и БД;
- - степени интегрированности с СУБД;
- - доступным платформам.

Классификация CASE-средств

- Классификация по типам в основном совпадает с компонентным составом CASE-средств и включает следующие основные типы:
- - средства анализа (Upper CASE), предназначенные для построения и анализа моделей предметной области (Design/IDEF (Meta Software), BPwin (Logic Works));
- - средства анализа и проектирования (Middle CASE), поддерживающие наиболее распространенные методологии проектирования и использующиеся для создания проектных спецификаций (Vantage Team Builder (Cayenne), Designer/2000 (ORACLE), Silverrun (CSA), PRO-IV (McDonnell Douglas), CASE.Аналитик (МакроПроджект)). Выходом таких средств являются спецификации компонентов и интерфейсов системы, архитектуры системы, алгоритмов и структур данных;
- - средства проектирования баз данных, обеспечивающие моделирование данных и генерацию схем баз данных (как правило, на языке SQL) для наиболее распространенных СУБД. К ним относятся ERwin (Logic Works), S-Designor (SDP) и DataBase Designer (ORACLE). Средства проектирования баз данных имеются также в составе CASE-средств Vantage Team Builder, Designer/2000, Silverrun и PRO-IV;

Классификация CASE-средств

- - средства разработки приложений. К ним относятся средства 4GL (Uniface (Compuware), JAM (JYACC), PowerBuilder (Sybase), Developer/2000 (ORACLE), New Era (Informix), SQL Windows (Gupta), Delphi (Borland) и др.) и генераторы кодов, входящие в состав Vantage Team Builder, PRO-IV и частично - в Silverrun;
- - средства реинжиниринга, обеспечивающие анализ программных кодов и схем баз данных и формирование на их основе различных моделей и проектных спецификаций. Средства анализа схем БД и формирования ERD входят в состав Vantage Team Builder, PRO-IV, Silverrun, Designer/2000, ERwin и S-Designor. В области анализа программных кодов наибольшее распространение получают объектно-ориентированные CASE-средства, обеспечивающие реинжиниринг программ на языке C++ (Rational Rose (Rational Software), Object Team (Cayenne)).

Классификация CASE-средств

- Вспомогательные типы включают:
- - средства планирования и управления проектом (SE Companion, Microsoft Project и др.);
- - средства конфигурационного управления (PVCS (Intersolv));
- - средства тестирования (Quality Works (Segue Software));
- - средства документирования (SoDA (Rational Software)).
- На сегодняшний день Российский рынок программного обеспечения располагает следующими наиболее развитыми CASE-средствами:
- Vantage Team Builder (Westmount I-CASE);
- Designer/2000;
- Silverrun;
- ERwin+BPwin;
- S-Designor;
- CASE.Аналитик.
- Кроме того, на рынке постоянно появляются как новые для отечественных пользователей системы (например, CASE /4/0, PRO-IV, System Architect, Visible Analyst Workbench, EasyCASE), так и новые версии и модификации перечисленных систем.

ЖИЗНЕННЫЙ ЦИКЛ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

- В основе деятельности по созданию и использованию программного обеспечения (ПО) лежит понятие его жизненного цикла (ЖЦ). ЖЦ является моделью создания и использования ПО, отражающей его различные состояния, начиная с момента возникновения необходимости в данном программном изделии и заканчивая моментом его полного выхода из употребления у всех пользователей.
- Традиционно выделяются следующие основные этапы ЖЦ ПО:
- ***• анализ требований,***
- ***• проектирование,***
- ***• кодирование (программирование),***
- ***• тестирование и отладка,***
- ***• эксплуатация и сопровождение.***

ЖИЗНЕННЫЙ ЦИКЛ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

- ЖЦ образуется в соответствии с принципом нисходящего проектирования и, как правило, носит итерационный характер: реализованные этапы, начиная с самых ранних, циклически повторяются в соответствии с изменениями требований и внешних условий, введением ограничений и т.п. На каждом этапе ЖЦ порождается определенный набор документов и технических решений, при этом для каждого этапа исходными являются документы и решения, полученные на предыдущем этапе. Каждый этап завершается верификацией порожденных документов и решений с целью проверки их соответствия исходным.
- Существующие модели ЖЦ определяют порядок исполнения этапов в ходе разработки, а также критерии перехода от этапа к этапу. В соответствии с этим наибольшее распространение получили три следующие модели ЖЦ.

ЖИЗНЕННЫЙ ЦИКЛ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

- ЖЦ ПО - это непрерывный процесс, который начинается с момента принятия решения о необходимости его создания и заканчивается в момент его полного изъятия из эксплуатации.
- Основным нормативным документом, регламентирующим ЖЦ ПО, является международный стандарт ISO/IEC 12207 (ISO - International Organization of Standardization - Международная организация по стандартизации, IEC - International Electrotechnical Commission - Международная комиссия по электротехнике). Он определяет структуру ЖЦ, содержащую процессы, действия и задачи, которые должны быть выполнены во время создания ПО.

ЖИЗНЕННЫЙ ЦИКЛ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

- Структура ЖЦ ПО по стандарту ISO/IEC 12207 базируется на трех группах процессов:
- основные процессы ЖЦ ПО (приобретение, поставка, разработка, эксплуатация, сопровождение);
- вспомогательные процессы, обеспечивающие выполнение основных процессов (документирование, управление конфигурацией, обеспечение качества, верификация, аттестация, оценка, аудит, решение проблем);
- организационные процессы (управление проектами, создание инфраструктуры проекта, определение, оценка и улучшение самого ЖЦ, обучение).

ЖИЗНЕННЫЙ ЦИКЛ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

- Разработка включает в себя все работы по созданию ПО и его компонент в соответствии с заданными требованиями, включая оформление проектной и эксплуатационной документации, подготовку материалов, необходимых для проверки работоспособности и соответствующего качества программных продуктов, материалов, необходимых для организации обучения персонала и т.д. Разработка ПО включает в себя, как правило, анализ, проектирование и реализацию (программирование).
- Эксплуатация включает в себя работы по внедрению компонентов ПО в эксплуатацию, в том числе конфигурирование базы данных и рабочих мест пользователей, обеспечение эксплуатационной документацией, проведение обучения персонала и т.д., и непосредственно эксплуатацию, в том числе локализацию проблем и устранение причин их возникновения, модификацию ПО в рамках установленного регламента, подготовку предложений по совершенствованию, развитию и модернизации системы.

ЖИЗНЕННЫЙ ЦИКЛ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

- Управление проектом связано с вопросами планирования и организации работ, создания коллективов разработчиков и контроля за сроками и качеством выполняемых работ. Техническое и организационное обеспечение проекта включает выбор методов и инструментальных средств для реализации проекта, определение методов описания промежуточных состояний разработки, разработку методов и средств испытаний ПО, обучение персонала и т.п. Обеспечение качества проекта связано с проблемами верификации, проверки и тестирования ПО. Верификация - это процесс определения того, отвечает ли текущее состояние разработки, достигнутое на данном этапе, требованиям этого этапа. Проверка позволяет оценить соответствие параметров разработки с исходными требованиями. Проверка частично совпадает с тестированием, которое связано с идентификацией различий между действительными и ожидаемыми результатами и оценкой соответствия характеристик ПО исходным требованиям. В процессе реализации проекта важное место занимают вопросы идентификации, описания и контроля конфигурации отдельных компонентов и всей системы в целом.

ЖИЗНЕННЫЙ ЦИКЛ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

- Управление конфигурацией является одним из вспомогательных процессов, поддерживающих основные процессы жизненного цикла ПО, прежде всего процессы разработки и сопровождения ПО. При создании проектов сложных ИС, состоящих из многих компонентов, каждый из которых может иметь разновидности или версии, возникает проблема учета их связей и функций, создания унифицированной структуры и обеспечения развития всей системы. Управление конфигурацией позволяет организовать, систематически учитывать и контролировать внесение изменений в ПО на всех стадиях ЖЦ. Общие принципы и рекомендации конфигурационного учета, планирования и управления конфигурациями ПО отражены в проекте стандарта ISO 12207-2.

ЖИЗНЕННЫЙ ЦИКЛ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

- Каждый процесс характеризуется определенными задачами и методами их решения, исходными данными, полученными на предыдущем этапе, и результатами. Результатами анализа, в частности, являются функциональные модели, информационные модели и соответствующие им диаграммы. ЖЦ ПО носит итерационный характер: результаты очередного этапа часто вызывают изменения в проектных решениях, выработанных на более ранних этапах.

ЖИЗНЕННЫЙ ЦИКЛ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

- Стандарт ISO/IEC 12207 не предлагает конкретную модель ЖЦ и методы разработки ПО (под моделью ЖЦ понимается структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач, выполняемых на протяжении ЖЦ. Модель ЖЦ зависит от специфики ИС и специфики условий, в которых последняя создается и функционирует). Его регламенты являются общими для любых моделей ЖЦ, методологий и технологий разработки. Стандарт ISO/IEC 12207 описывает структуру процессов ЖЦ ПО, но не конкретизирует в деталях, как реализовать или выполнить действия и задачи, включенные в эти процессы.

Модели жизненного цикла ПО

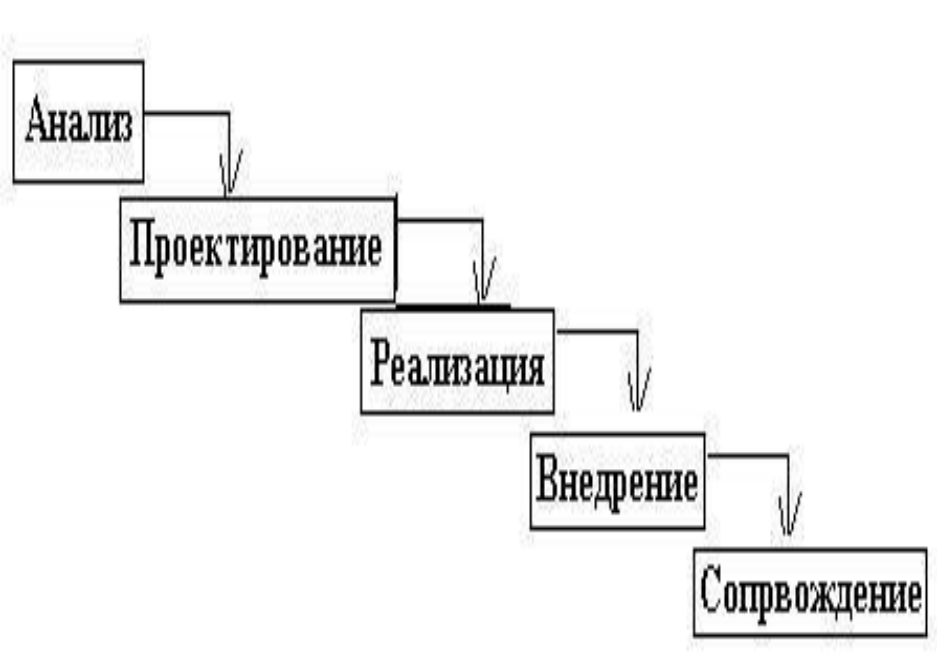
- Стандарт ISO/IEC 12207 не предлагает конкретную модель ЖЦ и методы разработки ПО (под моделью ЖЦ понимается структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач, выполняемых на протяжении ЖЦ. Модель ЖЦ зависит от специфики ИС и специфики условий, в которых последняя создается и функционирует). Его регламенты являются общими для любых моделей ЖЦ, методологий и технологий разработки. Стандарт ISO/IEC 12207 описывает структуру процессов ЖЦ ПО, но не конкретизирует в деталях, как реализовать или выполнить действия и задачи, включенные в эти процессы.
- К настоящему времени наибольшее распространение получили следующие две основные модели ЖЦ:
- каскадная модель;
- спиральная модель.

Модели жизненного цикла ПО

- В изначально существовавших однородных ИС каждое приложение представляло собой единое целое. Для разработки такого типа приложений применялся каскадный способ. Его основной характеристикой является разбиение всей разработки на этапы, причем переход с одного этапа на следующий происходит только после того, как будет полностью завершена работа на текущем (рис. 1). Каждый этап завершается выпуском полного комплекта документации, достаточной для того, чтобы разработка могла быть продолжена другой командой разработчиков.
- Положительные стороны применения каскадного подхода заключаются в следующем:
- на каждом этапе формируется законченный набор проектной документации, отвечающий критериям полноты и согласованности;
- выполняемые в логичной последовательности этапы работ позволяют планировать сроки завершения всех работ и соответствующие затраты.

Модели жизненного цикла ПО

- Рис.1. Каскадная схема разработки ПО

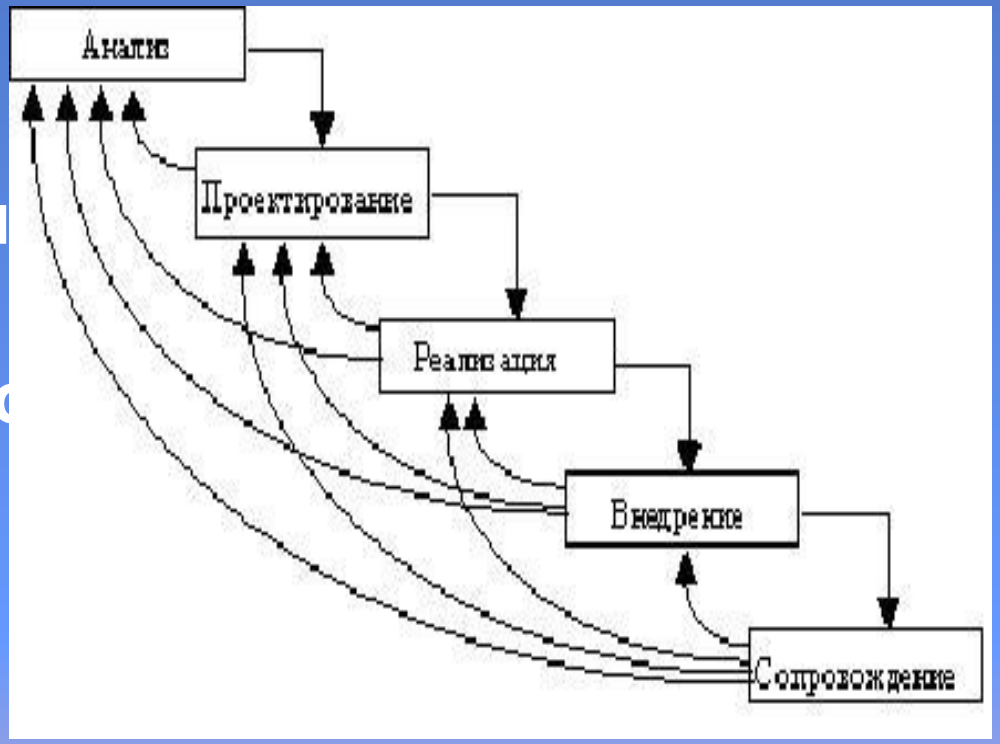


Модели жизненного цикла ПО

- Каскадный подход хорошо зарекомендовал себя при построении ИС, для которых в самом начале разработки можно достаточно точно и полно сформулировать все требования, с тем чтобы предоставить разработчикам свободу реализовать их как можно лучше с технической точки зрения. В эту категорию попадают сложные расчетные системы, системы реального времени и другие подобные задачи. Однако, в процессе использования этого подхода обнаружился ряд его недостатков, вызванных прежде всего тем, что реальный процесс создания ПО никогда полностью не укладывался в такую жесткую схему. В процессе создания ПО постоянно возникала потребность в возврате к предыдущим этапам и уточнении или пересмотре ранее принятых решений. В результате реальный процесс создания ПО принимал следующий вид (рис.2):

Модели жизненного цикла ПО

- Рис. 2. Реальный процесс разработки ПО по каскадной схеме



Модели жизненного цикла ПО

- Основным недостатком каскадного подхода является существенное запаздывание с получением результатов. Согласование результатов с пользователями производится только в точках, планируемых после завершения каждого этапа работ, требования к ИС "заморожены" в виде технического задания на все время ее создания. Таким образом, пользователи могут внести свои замечания только после того, как работа над системой будет полностью завершена. В случае неточного изложения требований или их изменения в течение длительного периода создания ПО, пользователи получают систему, не удовлетворяющую их потребностям. Модели (как функциональные, так и информационные) автоматизируемого объекта могут устареть одновременно с их утверждением.

Модели жизненного цикла ПО

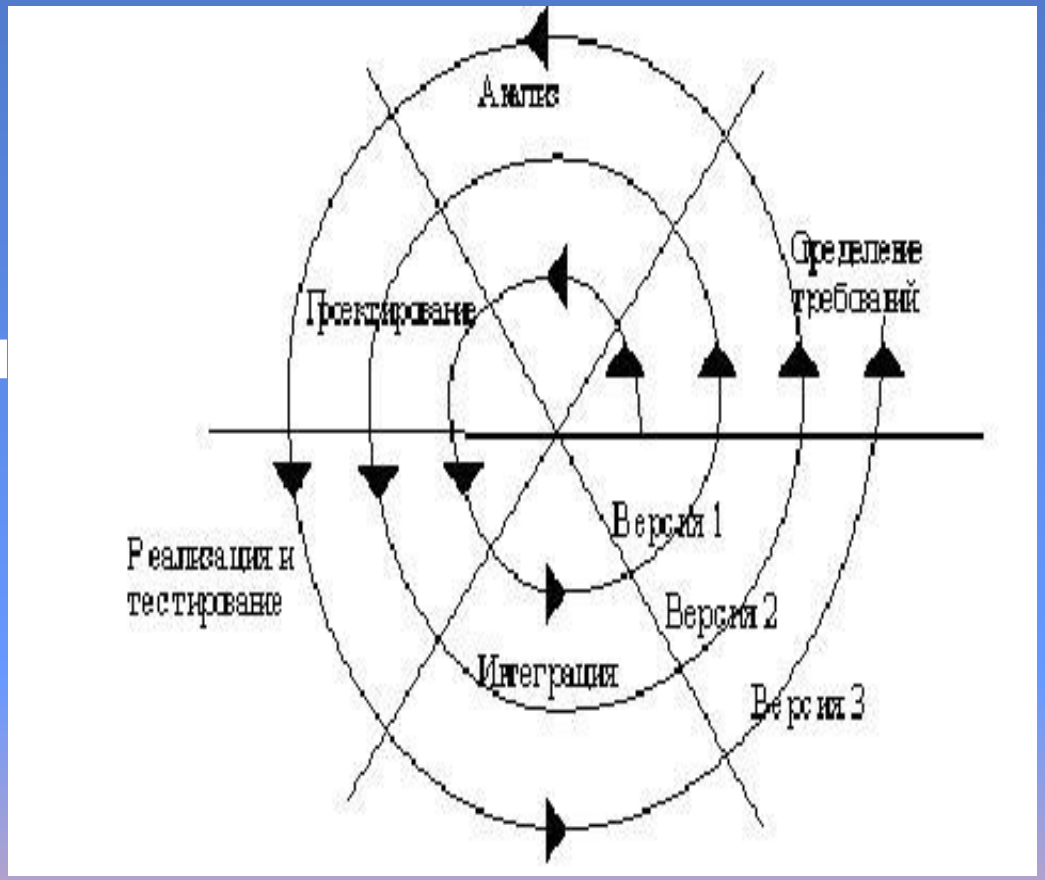
- Для преодоления перечисленных проблем была предложена спиральная модель ЖЦ (рис. 3), делающая упор на начальные этапы ЖЦ: анализ и проектирование. На этих этапах реализуемость технических решений проверяется путем создания прототипов. Каждый виток спирали соответствует созданию фрагмента или версии ПО, на нем уточняются цели и характеристики проекта, определяется его качество и планируются работы следующего витка спирали. Таким образом углубляются и последовательно конкретизируются детали проекта и в результате выбирается обоснованный вариант, который доводится до реализации.

Модели жизненного цикла ПО

- Разработка итерациями отражает объективно существующий спиральный цикл создания системы. Неполное завершение работ на каждом этапе позволяет переходить на следующий этап, не дожидаясь полного завершения работы на текущем. При итеративном способе разработки недостающую работу можно будет выполнить на следующей итерации. Главная же задача - как можно быстрее показать пользователям системы работоспособный продукт, тем самым активизируя процесс уточнения и дополнения требований.
- Основная проблема спирального цикла - определение момента перехода на следующий этап. Для ее решения необходимо ввести временные ограничения на каждый из этапов жизненного цикла. Переход осуществляется в соответствии с планом, даже если не вся запланированная работа закончена. План составляется на основе статистических данных, полученных в предыдущих проектах, и личного опыта разработчиков.

Модели жизненного цикла ПО

- Рис 3. Спиральная модель ЖЦ



Жизненный цикл разработки сложных программных средств

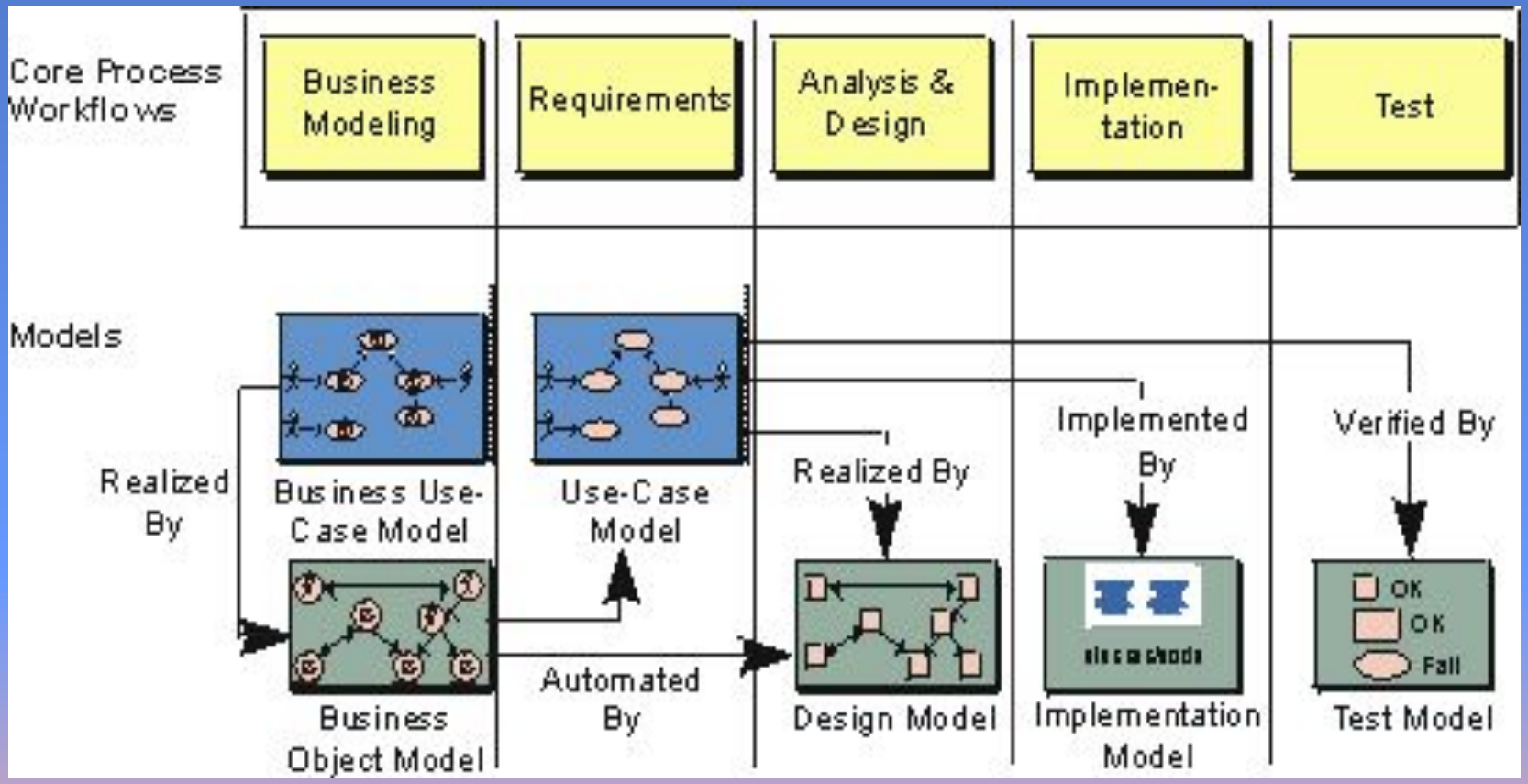
- Этап 1. Системный анализ проекта ПС.
- Этап 2. Предварительное (пилотное) проектирование ПС.
- Этап 3. Детальное проектирование ПС.
- Этап 4. Кодирование (программирование), отладка и разработка документации компонентов ПС.
- Этап 5. Интеграция (комплексирование) и комплексная отладка ПС.
- Этап 6. Испытание и документирование ПС.
- Этап 7. Внедрение и поддержка разработчиками процесса эксплуатации ПС пользователями.
- Этап 8. Сопровождение и развитие ПС.

ОБЪЕКТНО-ОРИЕНТИРОВАННАЯ МЕТОДОЛОГИЯ ПРОЕКТИРОВАНИЯ программных систем **Rational Unified Process** фирмы **Rational Software Corporation**

Rational Unified Process (RUP)

- Процесс создания программных систем (**ПС**) по методологии разработки программных систем **Rational Unified Process** фирмы **Rational Software Corporation** включает следующие шесть этапов.
- 1. Моделирование предметной области (**Business Modeling**).
- 2. Определение требований к системе (**Requirements**).
- 3. Анализ и проектирование (**Analysis & Design**).
- 4. Разработка (**Implementation**).
- 5. Тестирование (**Test**).
- 6. Внедрение (**Deployment**).

Этапы разработки ПС в RUP



Моделирование предметной области (**Business Modeling**)

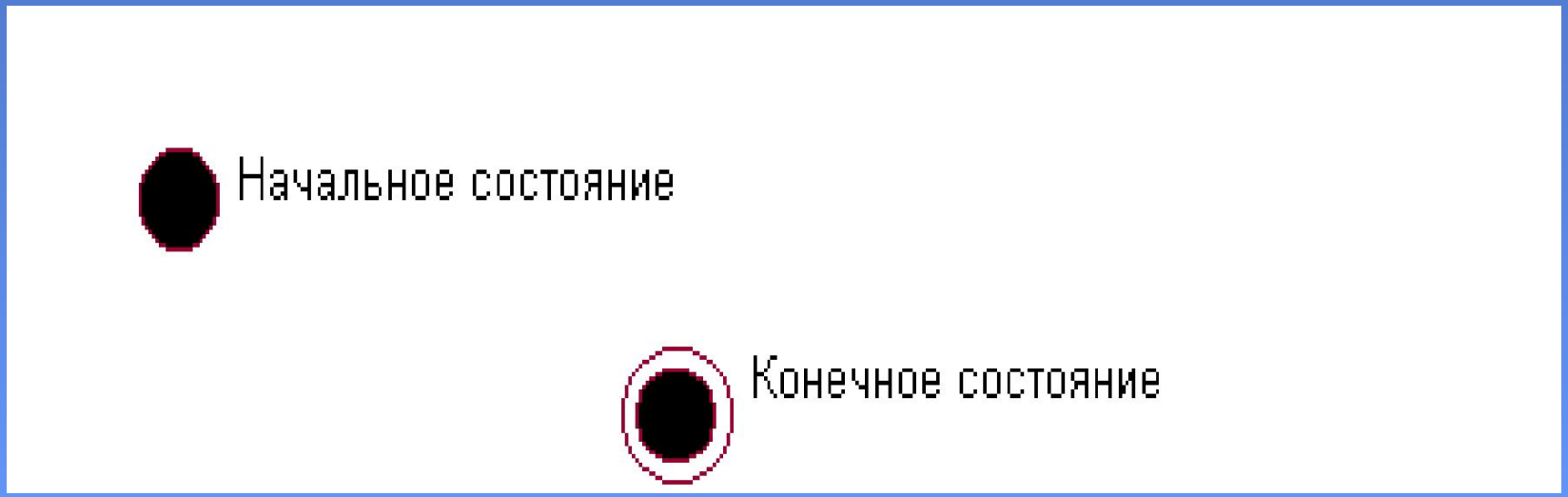
- На этапе моделирования предметной области разрабатываются диаграммы деятельности (activity diagram), которые используются для описания последовательности различных действий субъектов и объектов (действующих лиц производственного процесса), а также могут быть использованы и для описания их состояний.

Элементы диаграммы деятельности (activity diagram)

- Диаграммы деятельности включают следующие элементы.
- 1. Начальное состояние (**start state**), которое обозначается черным маленьким кружком, с которым может быть связано название “**начало**”.
- 2. Конечное состояние (**end state**), которое обозначается большим черным кружком внутри круга, с которым может быть связано название “**конец**”.

(Каждая диаграмма должна иметь только одно начальное состояние и может иметь несколько заключительных состояний).

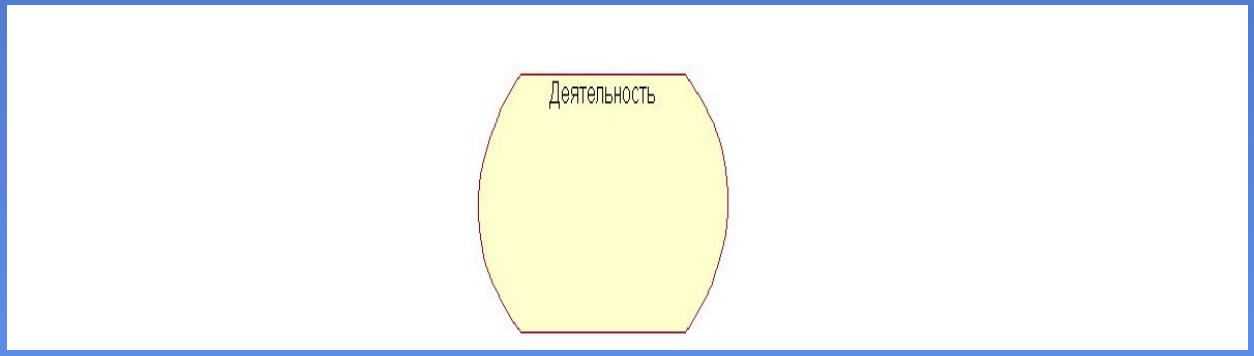
Пример начального (**start state**) и конечного состояния (**end state**)

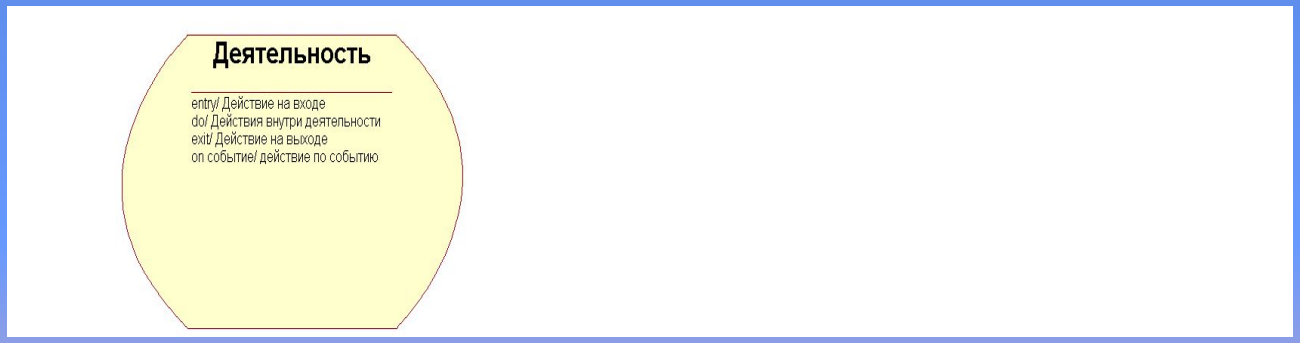


Элементы диаграммы деятельности (activity diagram)

- 3. Деятельность (**activity**), которая обозначается прямоугольником с закругленными сторонами. **Имя** должно отражать цель деятельности.
- 4. Состояние (**state**), которое обозначается прямоугольником с закругленными углами. Элемент состояние (**state**) используется для описания определенных состояний какого-либо субъекта или объекта. С этим элементом должно быть связано **имя**. Имя должно отражать состояние субъекта или объекта.
- 5. Переход **(state transition)**.

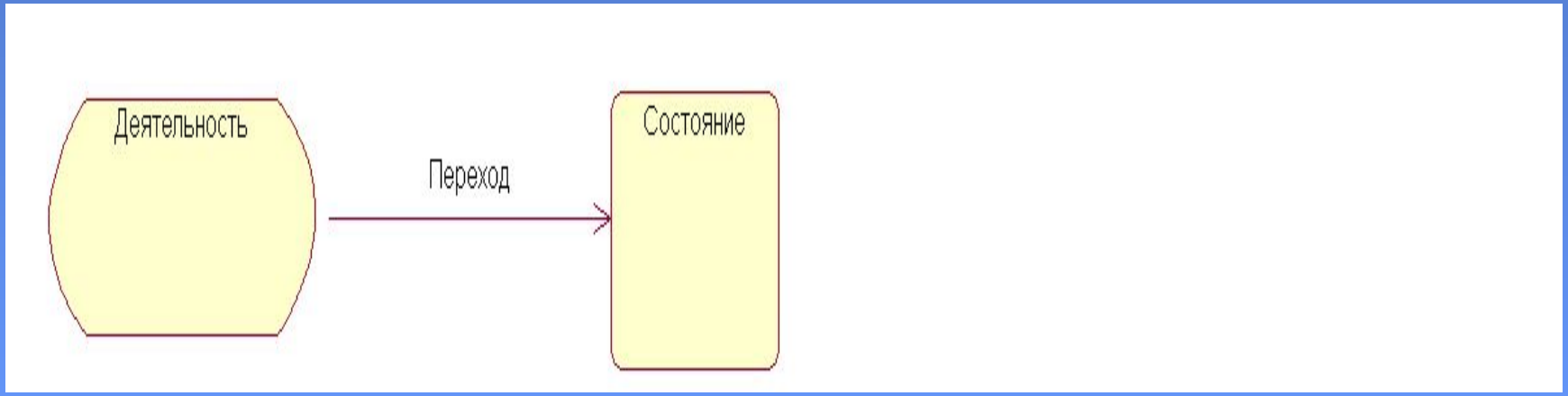
Пример элемента деятельность (**activity**)





Пример элемента состояния (**state**)

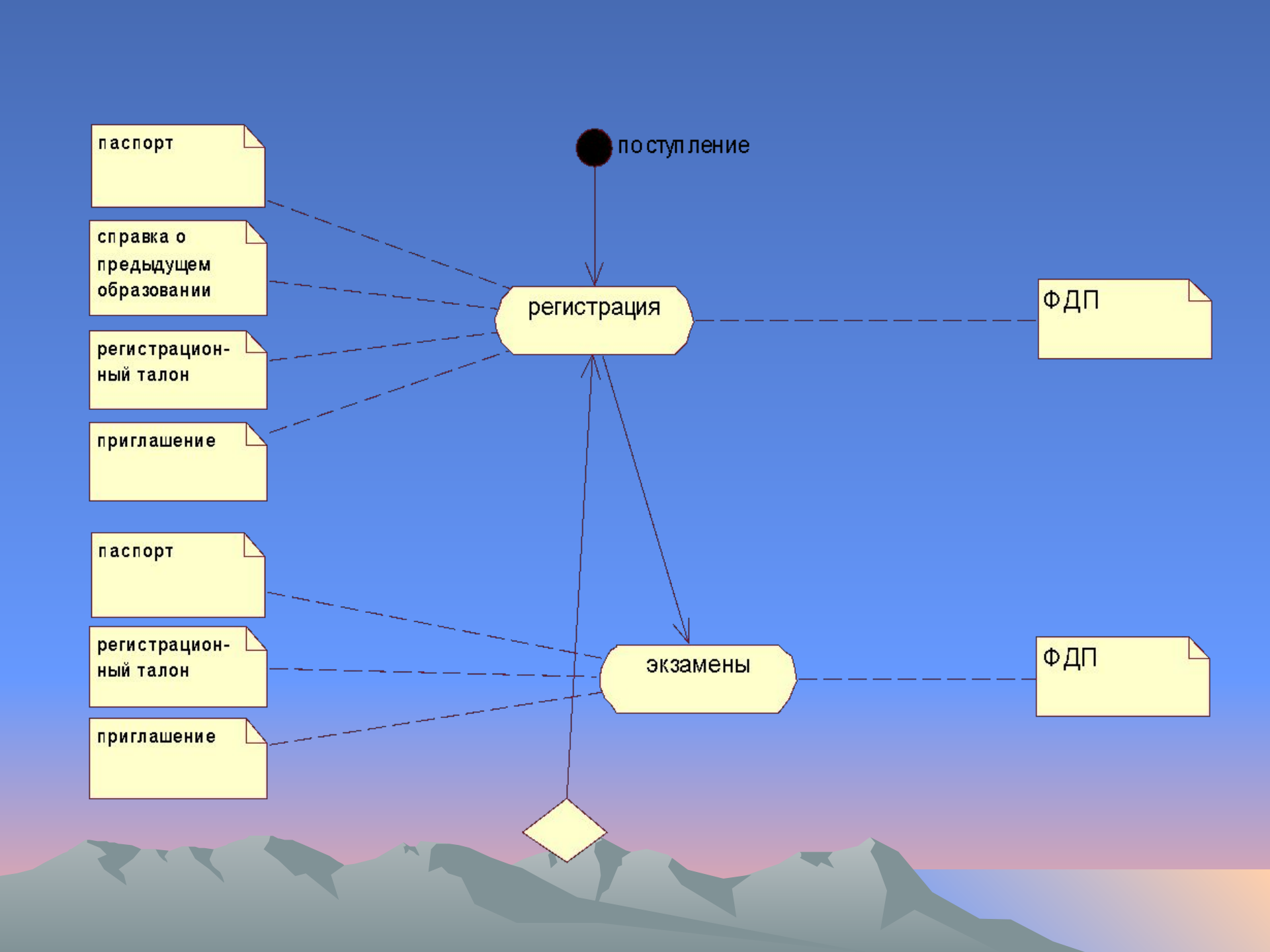
Пример элемента перехода (**state transition**)

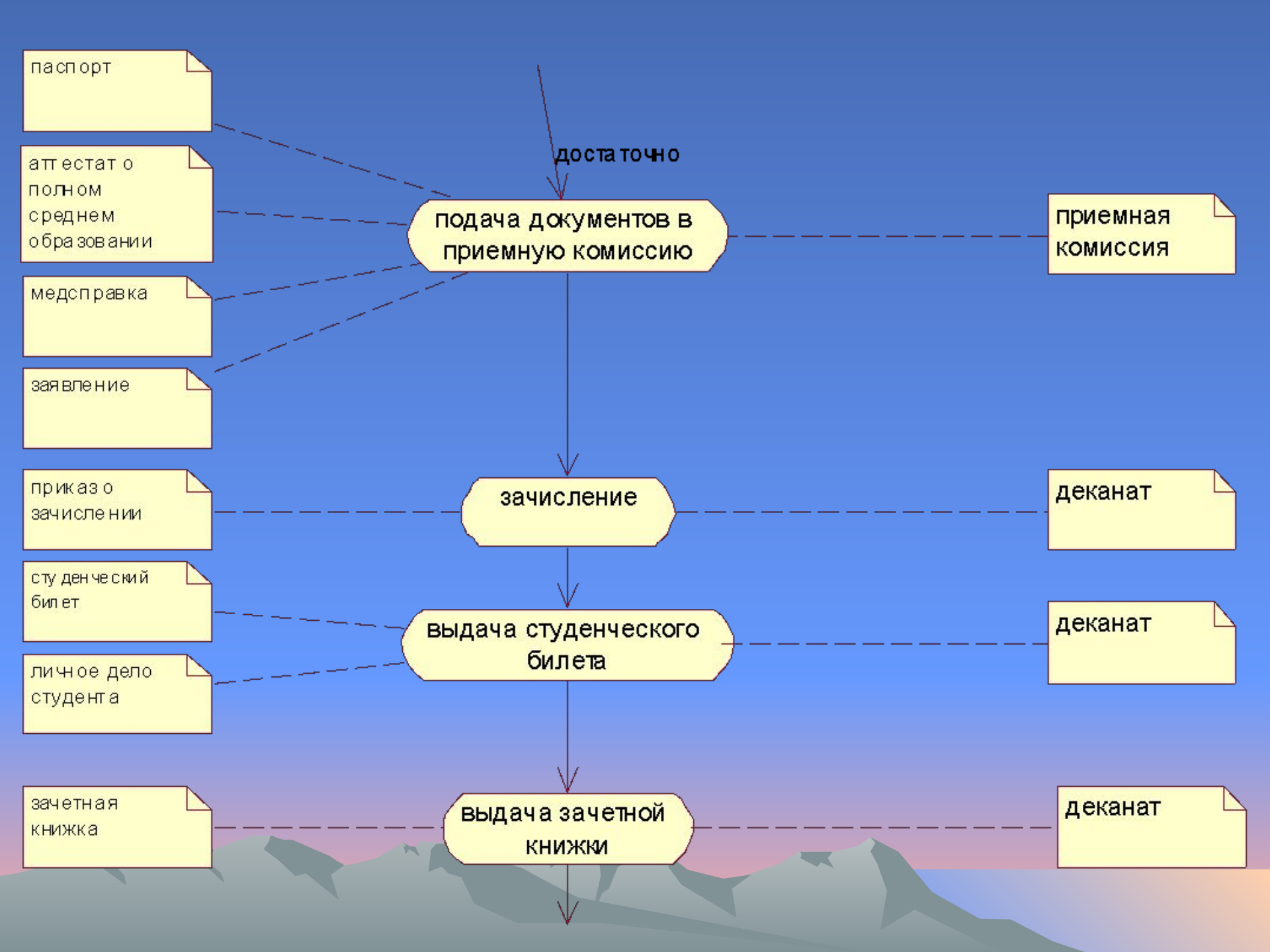


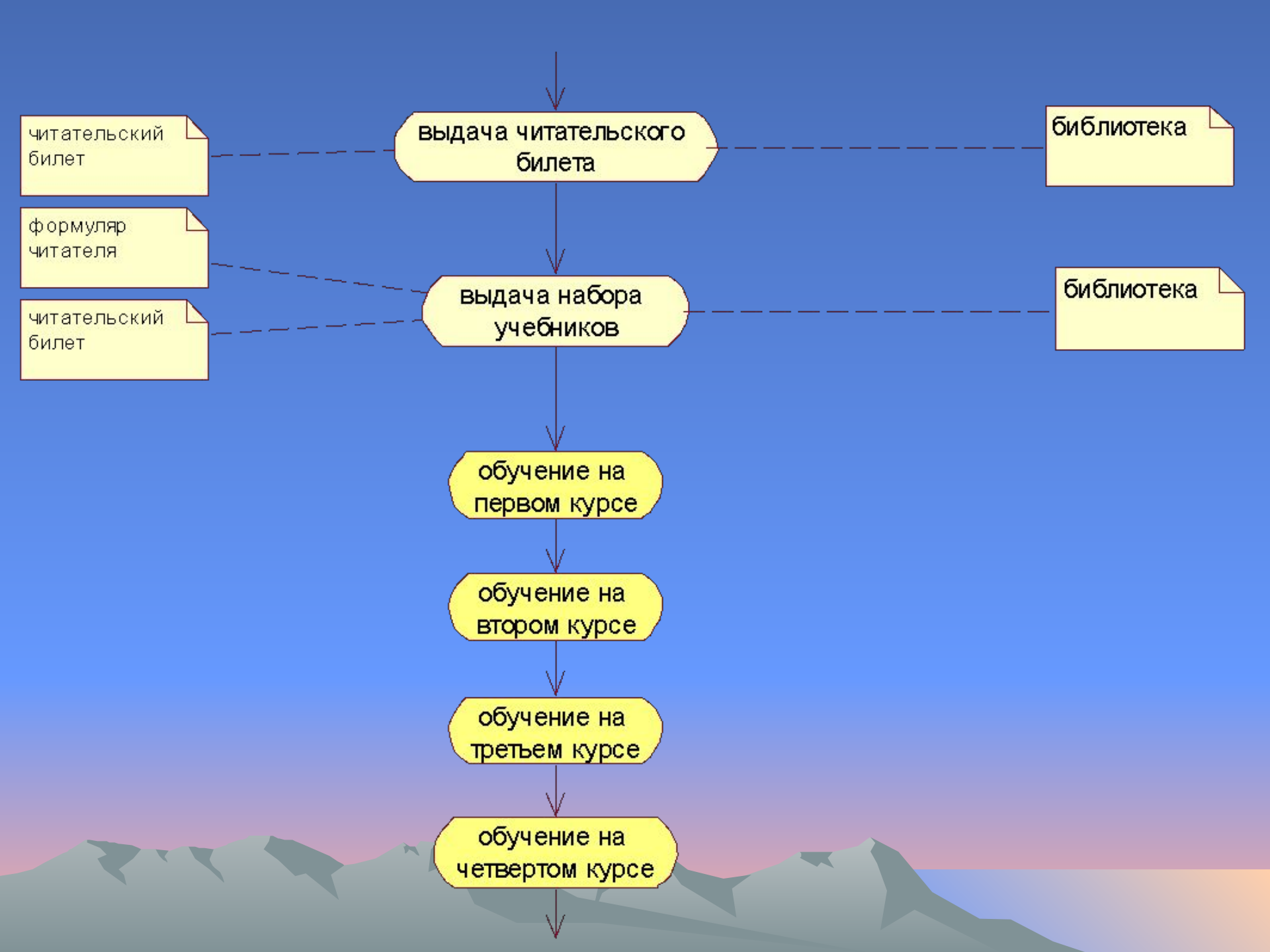
Переход (**state transition**) может иметь имя, связанное с **событием** его вызвавшим.

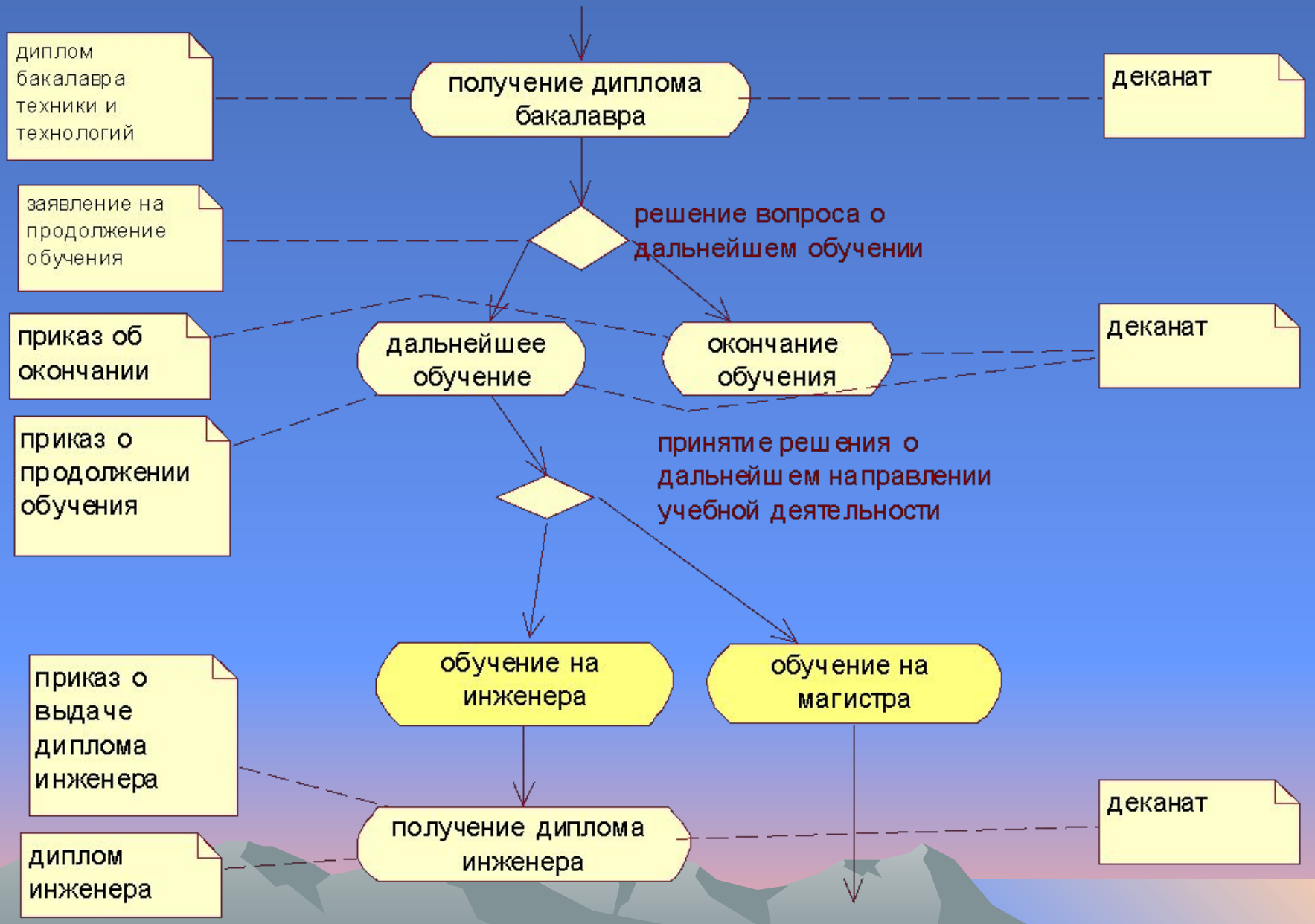
Элементы диаграммы деятельности (activity diagram)

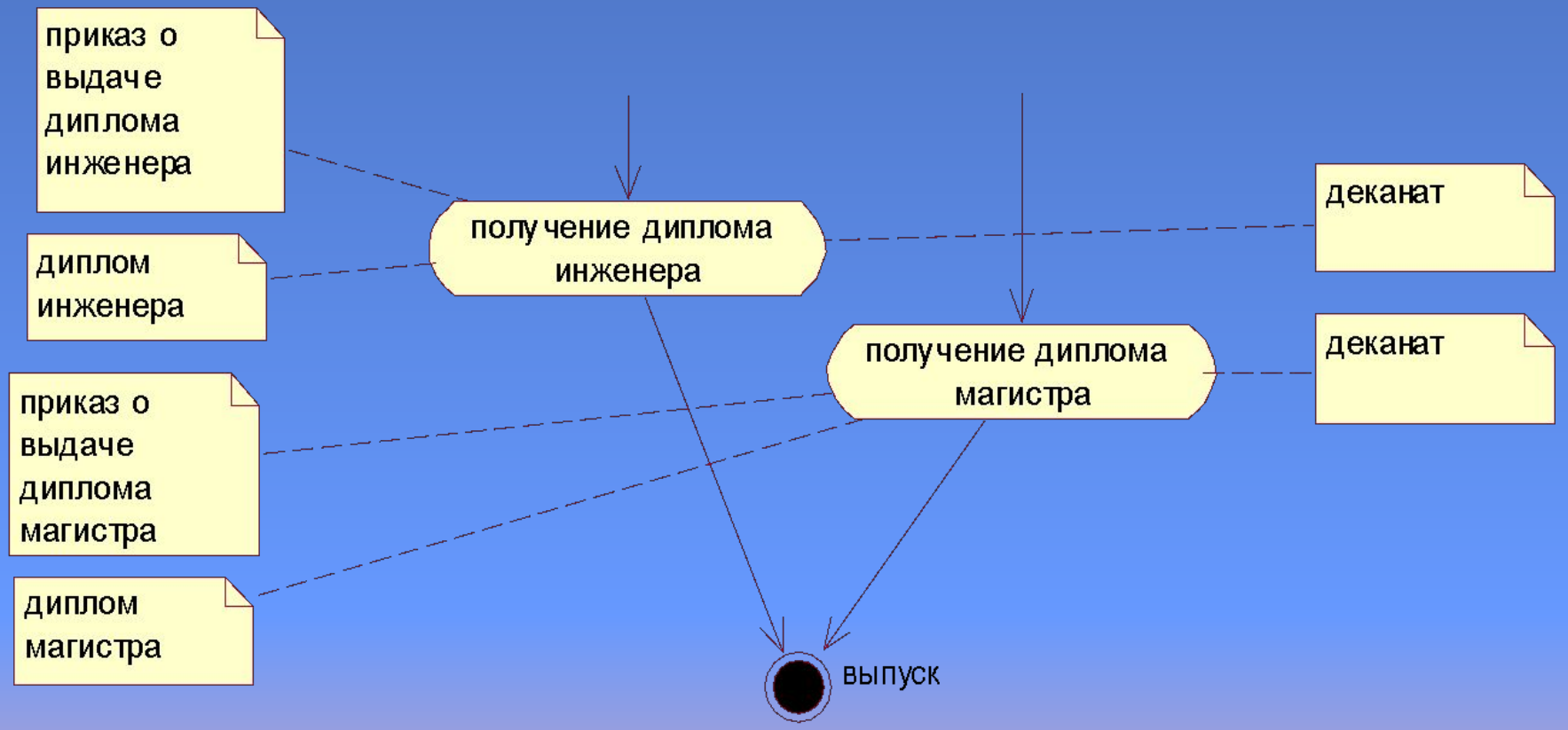
- 6. Решение (**decision**), используемый для отображений действий, выполняемых по условию.
- 7. Горизонтальные синхронизаторы (**horizontal synhronization**), которые используются для отражения выполнения параллельных деятельностей.
- 8. Разделительные линии (**swimline**), используются для разделения диаграммы на части, например, с целью отражения на диаграммах, ответственных за выполнение определенных действий .

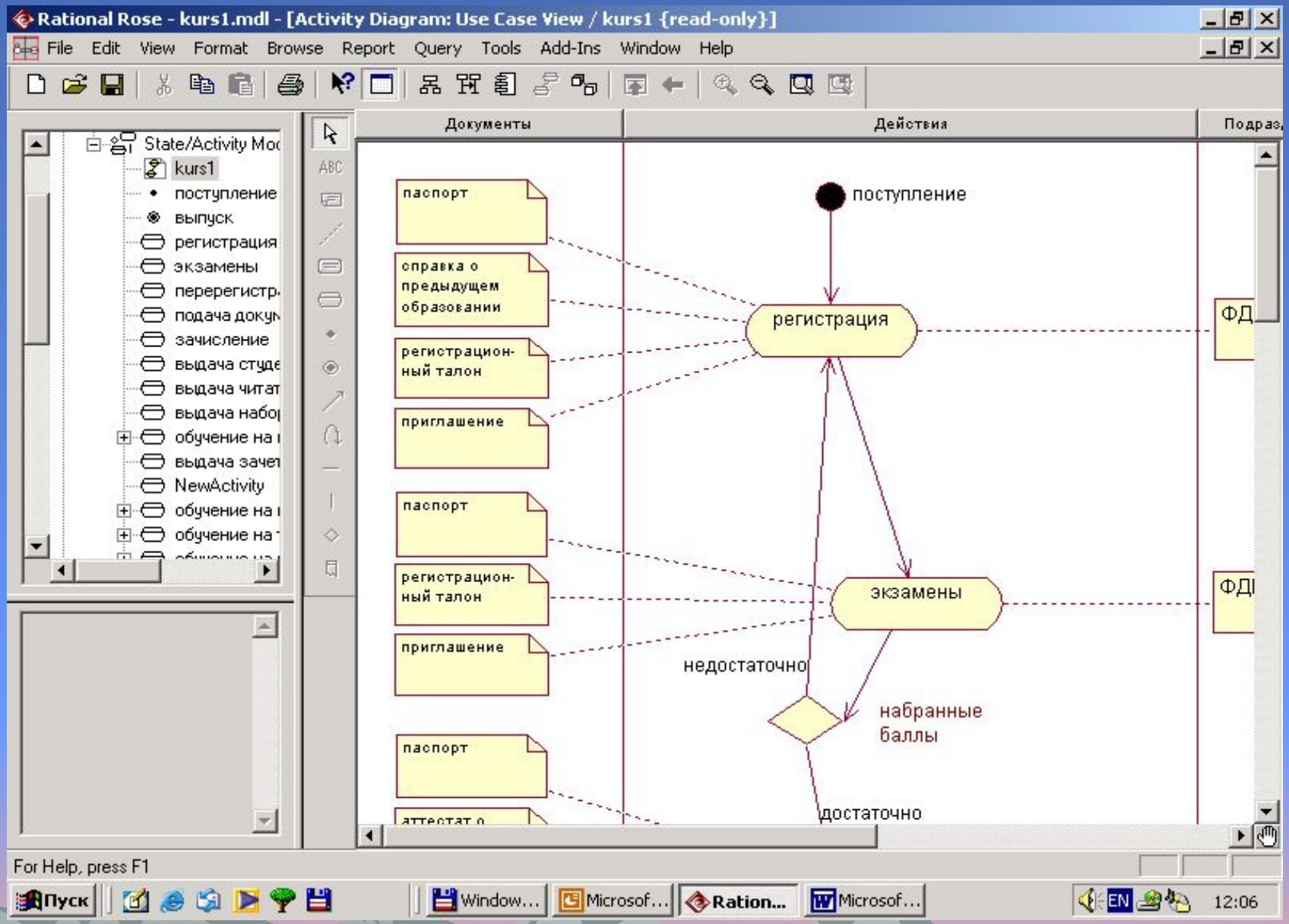

паспорт
справка о предыдущем образовании
регистрационный талон
приглашение
поступление
регистрация
ФДП
паспорт
регистрационный талон
приглашение
экзамены
ФДП


паспорт
аттестат о полном среднем образовании
медсправка
заявление
достаточно
подача документов в приемную комиссию
приемная комиссия
приказ о зачислении
зачисление
деканат
студенческий билет
личное дело студента
выдача студенческого билета
деканат
зачетная книжка
выдача зачетной книжки
деканат


читательский билет
выдача читательского билета
библиотека
формуляр читателя
читательский билет
выдача набора учебников
библиотека
обучение на первом курсе
обучение на втором курсе
обучение на третьем курсе
обучение на четвертом курсе


диплом бакалавра техники и технологий
получение диплома бакалавра
деканат
заявление на продолжение обучения
решение вопроса о дальнейшем обучении
приказ об окончании
дальнейшее обучение
окончание обучения
деканат
приказ о продолжении обучения
принятие решения о дальнейшем направлении учебной деятельности
обучение на инженера
обучение на магистра
приказ о выдаче диплома инженера
получение диплома инженера
деканат
диплом инженера


приказ о выдаче диплома инженера
диплом инженера
получение диплома инженера
деканат
приказ о выдаче диплома магистра
диплом магистра
получение диплома магистра
деканат
выпуск



Пример вертикальных линий (swimline)

Этап моделирования предметной области в методологии RUP

- На этапе моделирования предметной области вместе с диаграммами деятельности могут быть разработаны диаграммы состояний **(Statechart diagram).**

Диаграммы состояний (Statechart diagram)

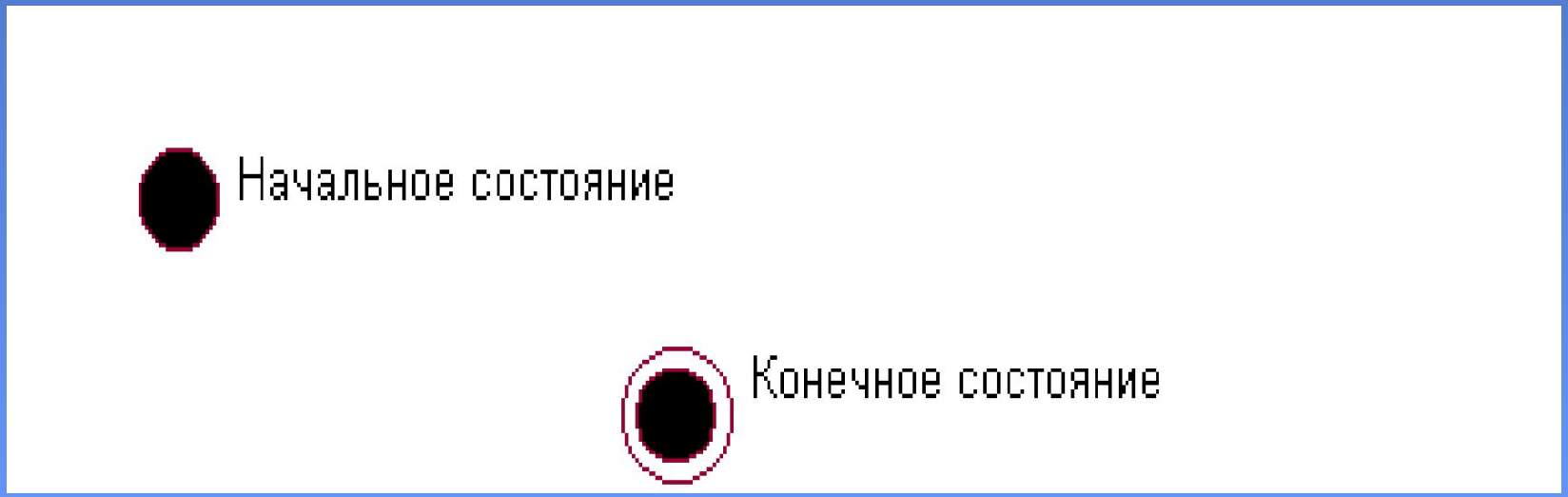
- **Диаграммы состояний (Statechart diagram)** используются для описания **динамики поведения** субъектов и объектов. Диаграмм состояний (**Statechart diagram**) показывают **состояния** объекта или субъекта, **события**, которые влекут **переход из одного состояния в другое**, **действия**, которые происходят при изменении состояния.

Элементы диаграммы состояний (**Statechart diagram**)

- Диаграммы состояний включают следующие элементы.
- 1. Начальное состояние (**start state**), которое обозначается черным маленьким кружком, с которым может быть связано название “**начало**”.
- 2. Конечное состояние (**end state**), которое обозначается большим черным кружком внутри круга, с которым может быть связано название “**конец**”.

(Каждая диаграмма должна иметь только одно начальное состояние и может иметь несколько заключительных состояний).

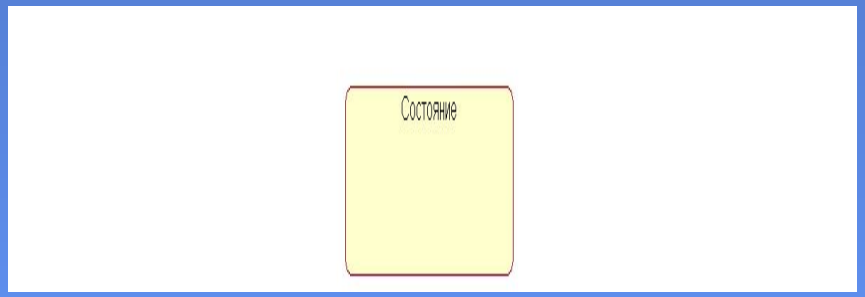
Пример начального (**start state**) и конечного состояния (**end state**)

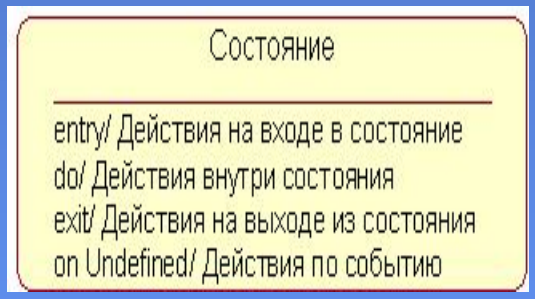


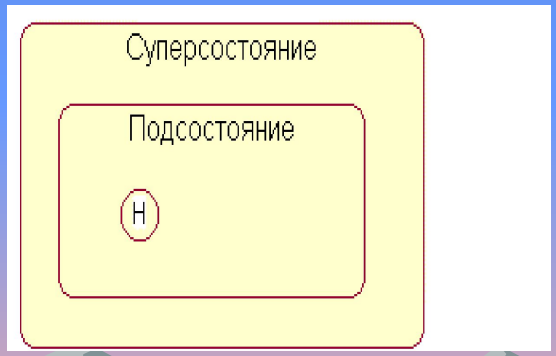
Элементы диаграммы состояний (**Statechart diagram**)

- 3. Состояния.
- 4. Переходы между состояниями.

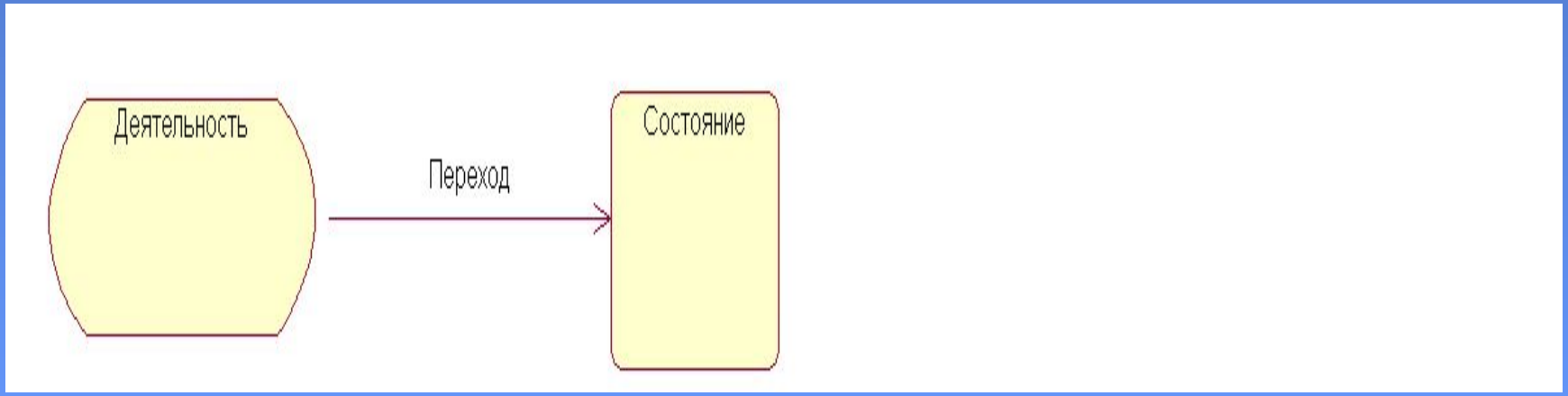
Пример элемента состояния (**state**)







Пример элемента перехода (**state transition**)



Переход (**state transition**) может иметь имя, связанное с **событием** его вызвавшим.

Этап моделирования предметной области в методологии RUP

На этом этапе осуществляется моделирование производственного процесса предметной области, выбранного для автоматизации

При моделировании производственного процесса разрабатывается **с использованием модель «business use case model».**

Цель построения **модели «business use case model».**

- 1.Понимание структуры автоматизируемой организации заказчиками, конечными пользователям, и разработчикам автоматизированных систем;
- 2. Определение требований к автоматизированной системе, поддерживающей работу организации.

Модель производственного процесса (**business use case model**) представляет собой **иерархию диаграмм** производственных функций.

Первый уровень иерархии должен включать одну или несколько организационных единиц (**organization unit**) – например, предприятие, подлежащее автоматизации.

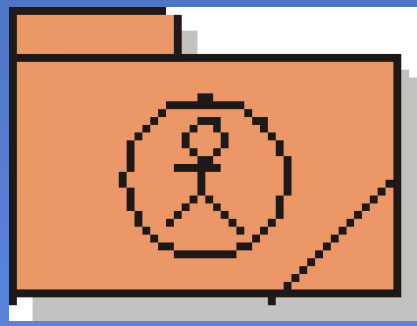
Последующие уровни иерархии могут включать также одну или несколько организационных единиц (**organization unit**), например, это могут быть подразделения автоматизируемого предприятия.

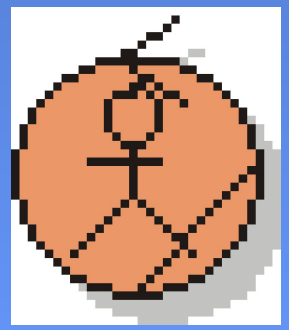
Отдельные производственные функции также могут быть декомпозированы другими диаграммами производственных функций, включающими исключительно действующих лиц производственного процесса, их функции, связи между действующими лицами и их функциями и между функциями.

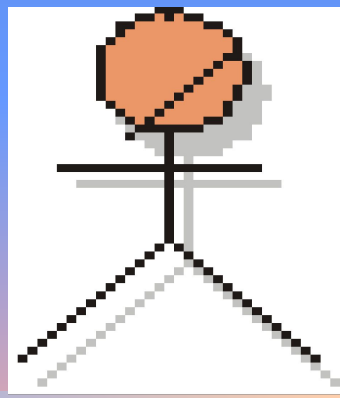
Модель производственных процессов (**business use case model**) строится с использованием диаграмм функций (**use case diagram**).

Элементы диаграмм функций (**use case diagram**)

- Организационные единицы (**organization unit**).
- **Субъект** производственного процесса (**business worker**).
- **Объект** производственного процесса.

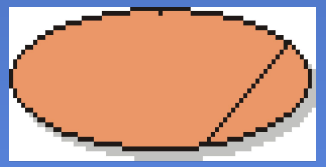


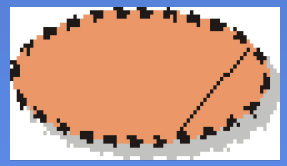




Элементы диаграмм функций (**use case diagram**)

- Функции производственного процесса.
- Декомпозированные функции производственного процесса.
- **Связи на диаграммах функций устанавливаются:**
- между организационными единицами;
- между действующим лицом и функцией;
- между функциями.





Типы связей на диаграмме функций

- Между организационными единицами может иметь место связь, которая является зависимостью. Связь обозначается **прерывистой линией со стрелкой.** Связь должна проводится **от зависимой** организационной единицы **к независимой**. Связь может быть двусторонней.

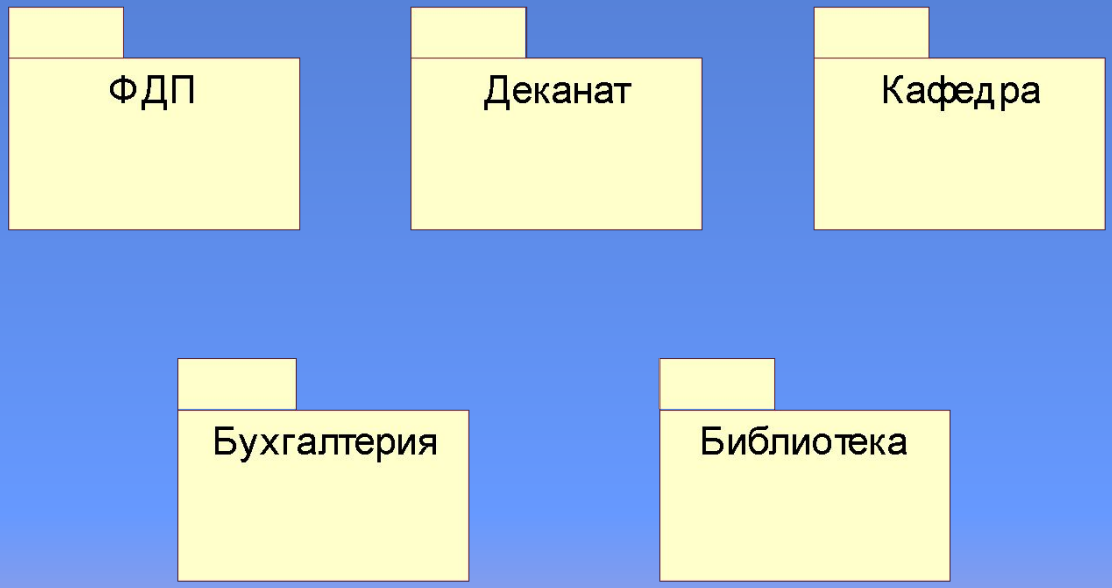
Типы связей на диаграмме функций

- Между действующим лицом производственного процесса (**business worker** или **business actor**) и функцией устанавливается **связь**, которая называется **ассоциацией**.
- Связь показывает **взаимодействие** между действующим лицом и функцией. Связь может быть двунаправленной. Связь обозначается сплошной линией со стрелкой или без нее.

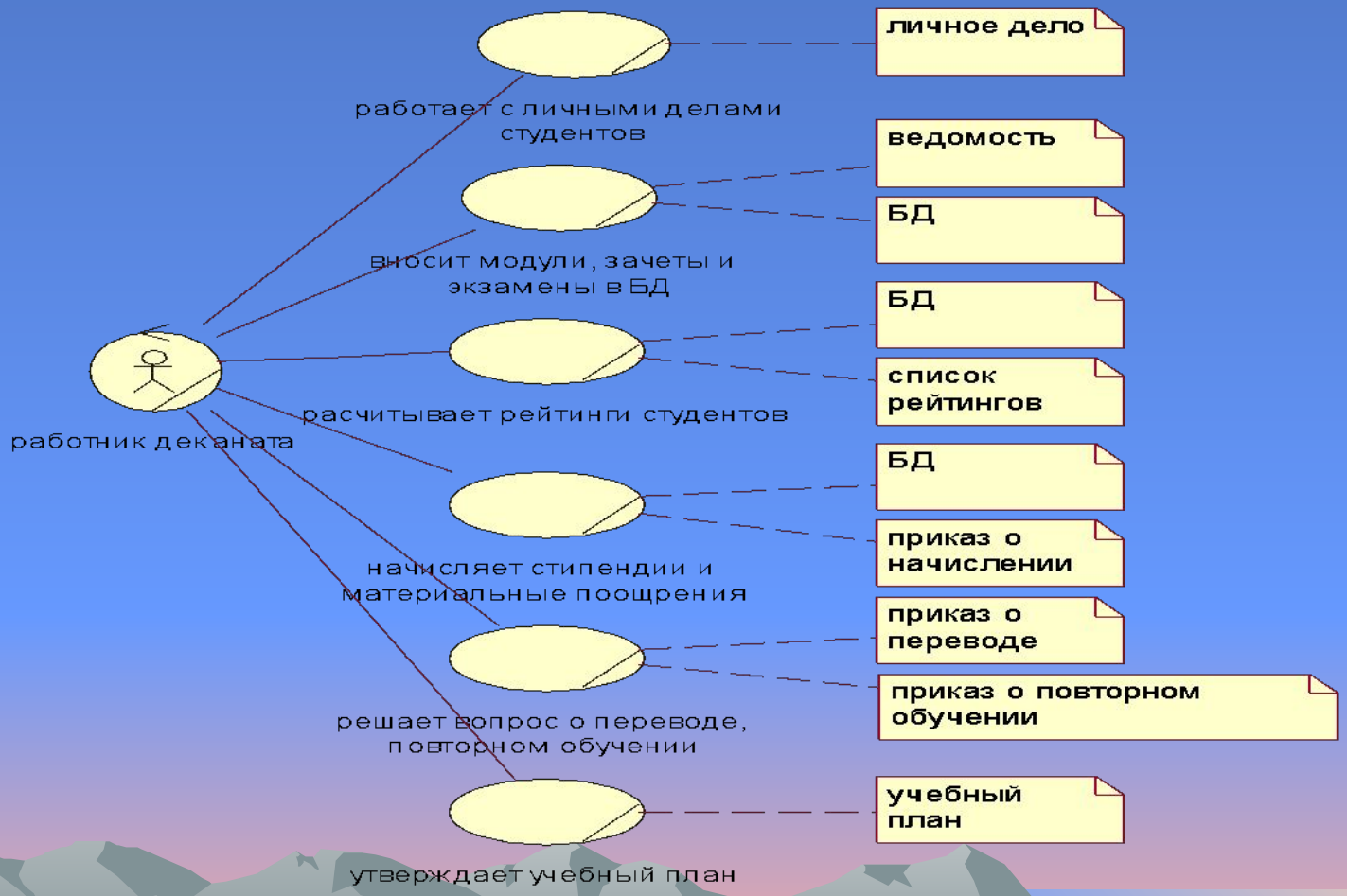
Типы связей на диаграмме функций

- На диаграммах производственных функций могут также используются и другие типы связей. Например, между функциями могут существовать связи типа <<**uses**>> (использует) и <<**extends**>> (расширяет).
- То есть, некоторые функции в системе могут **использовать** другие функции. Некоторые функции могут выполняться **при наступлении определенных условий** или **быть опциональными**. В первом случае используются связь <<**uses**>>, во втором случае - <<**extends**>>.
- Связи <<**uses**>> и <<**extends**>> обозначают **линией со стрелкой в виде не закрашенного треугольника**.
- Для связи <<**uses**>> стрелка направлена **к функции**, которую используют. Для связи <<**extends**>> стрелка направлена к функции, которая **включает функцию**, используемую опционально или по наступлении определенного условия.

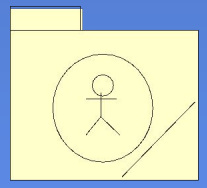
Пример модели «business use case model»



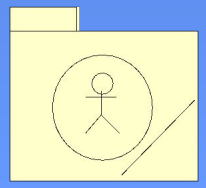
Пример модели «business use case model»



Пример модели «business use case model»

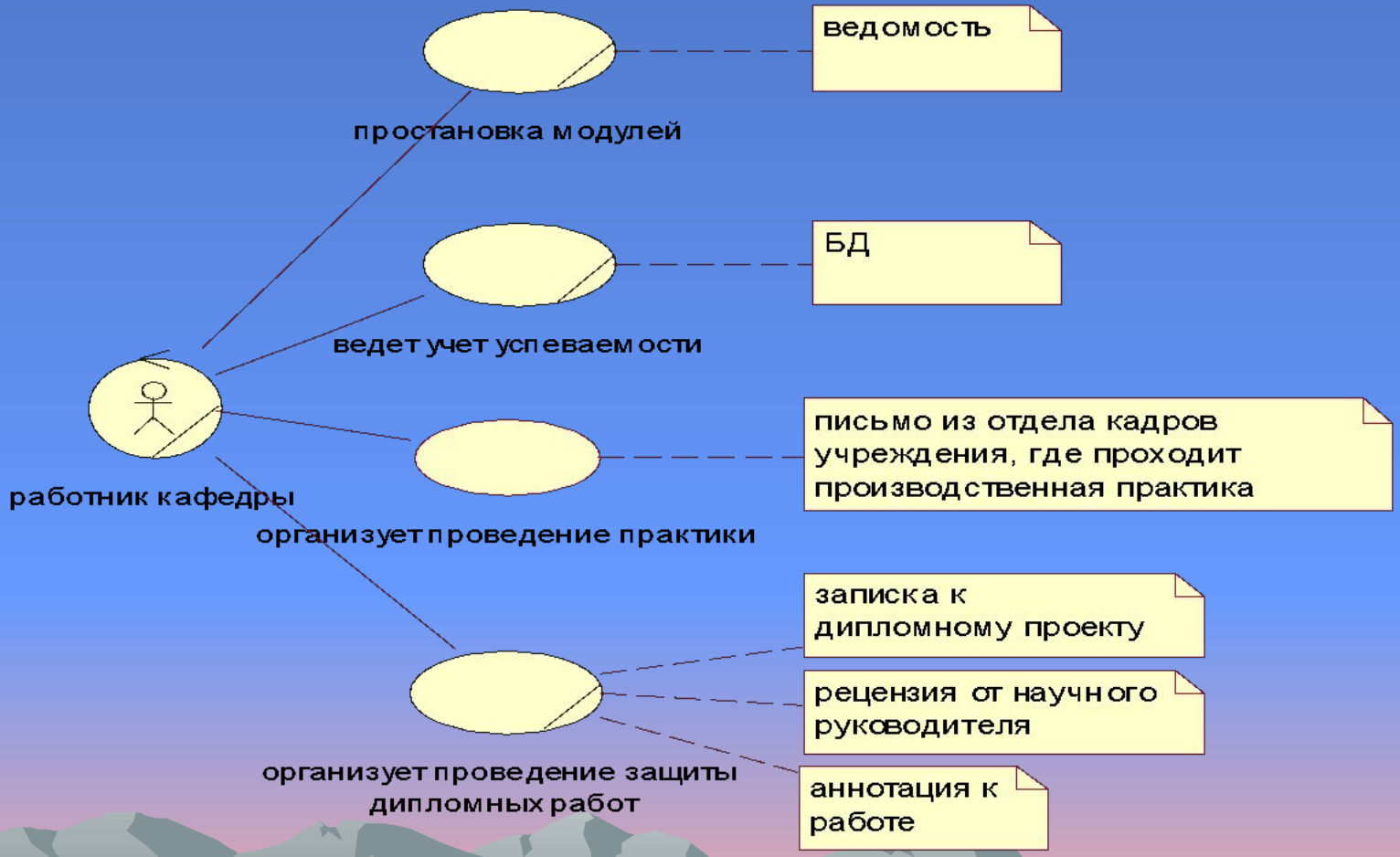


отдел методической работы

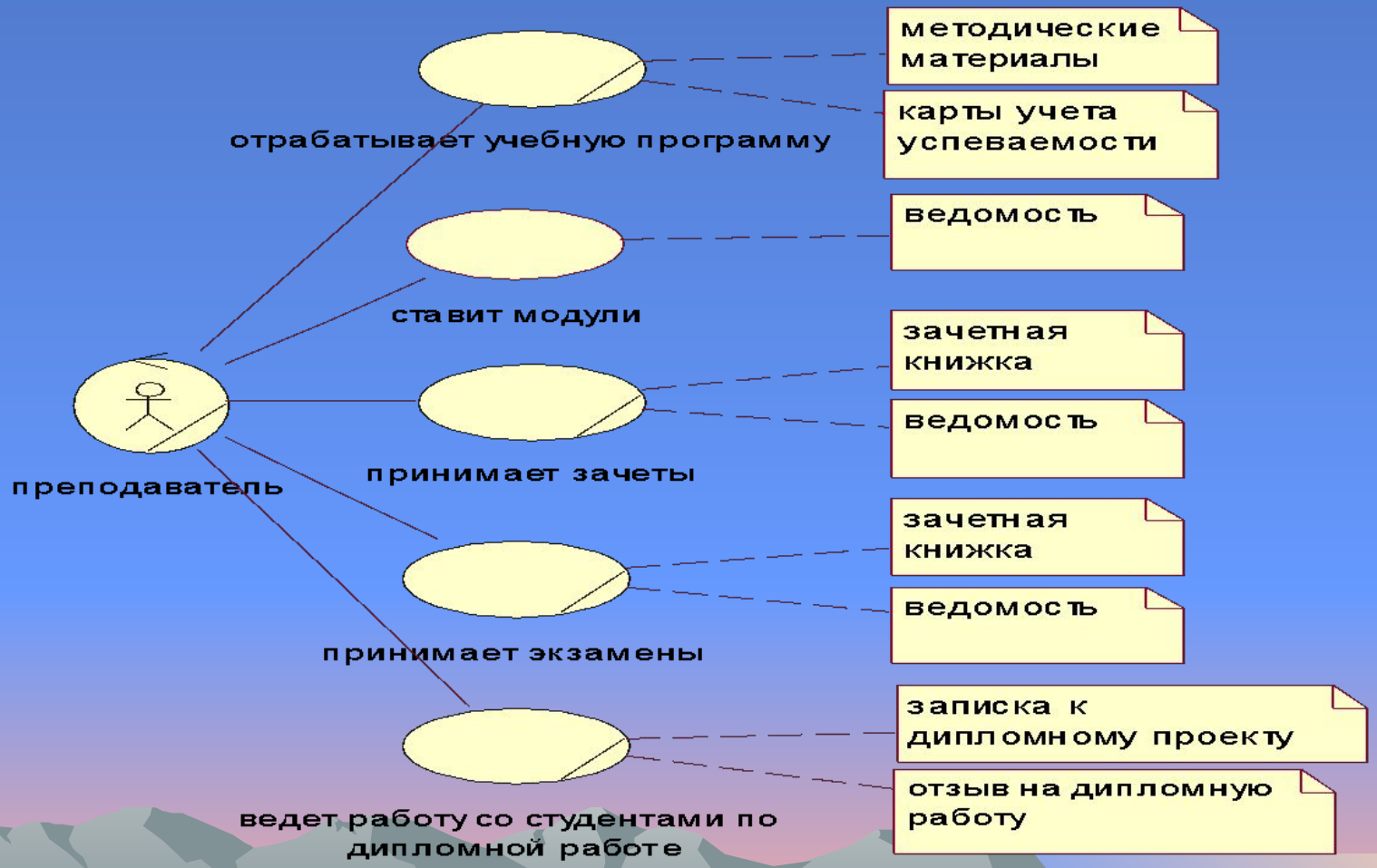


преподавательский состав

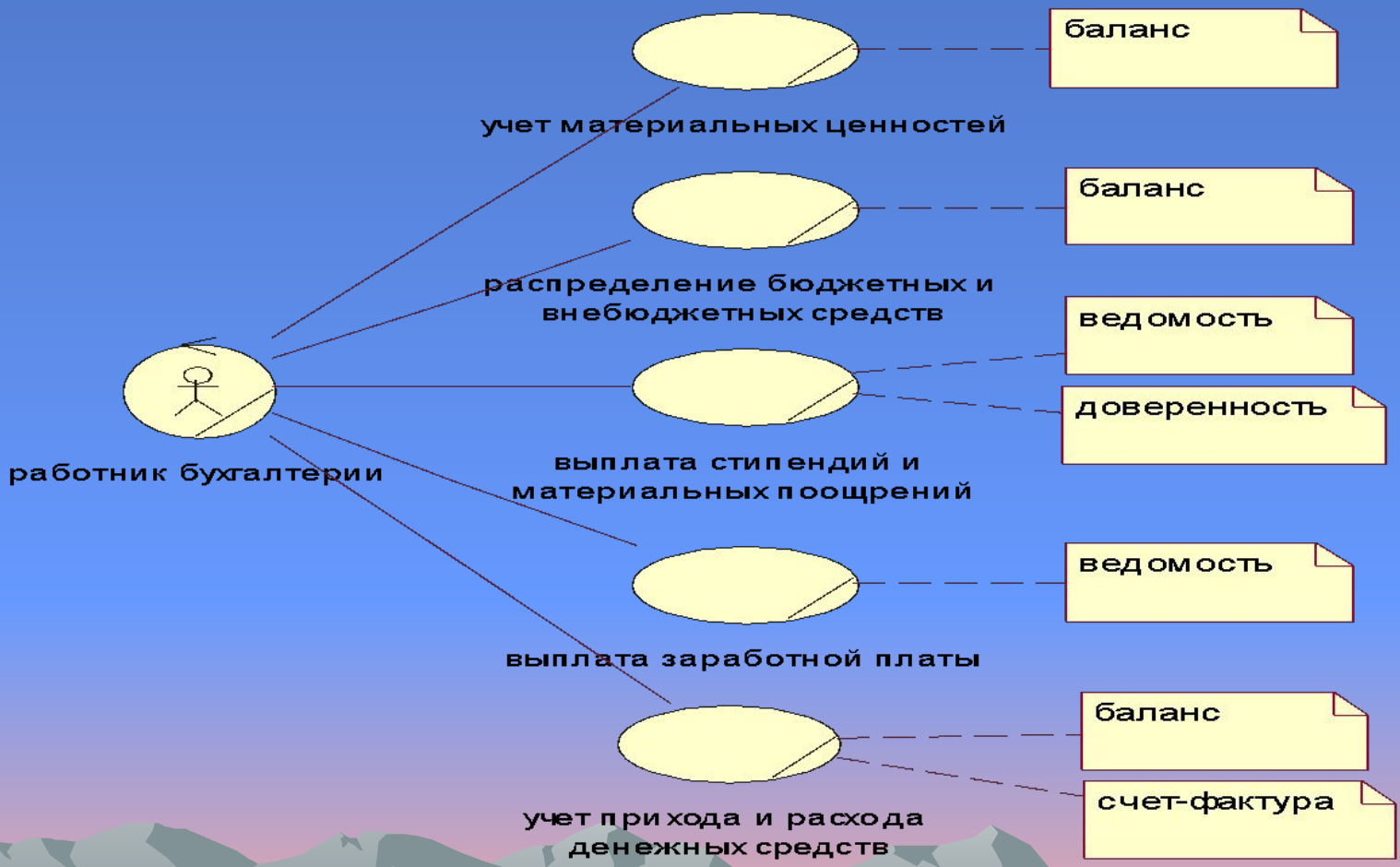
Пример модели «business use case model»



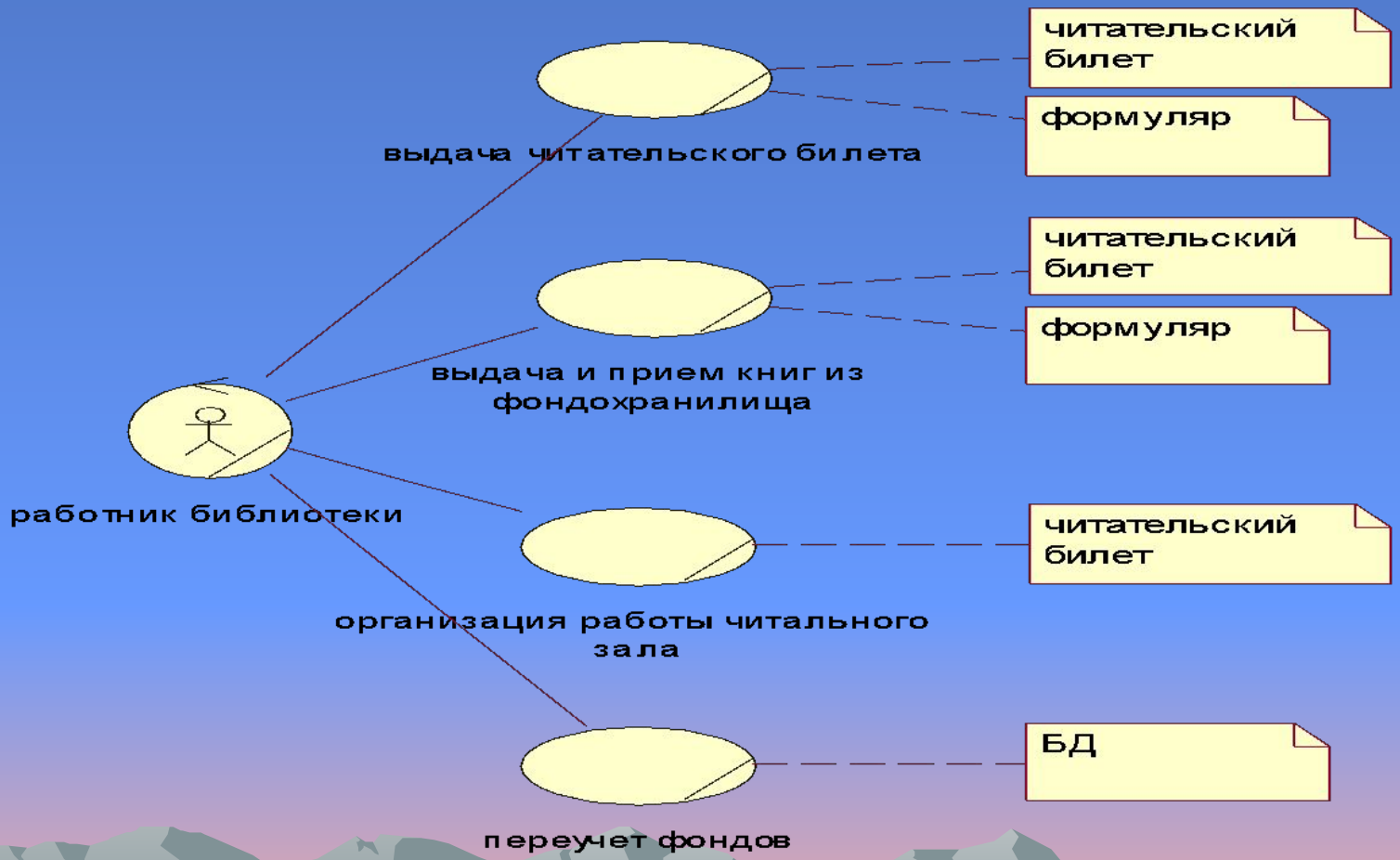
Пример модели «business use case model»



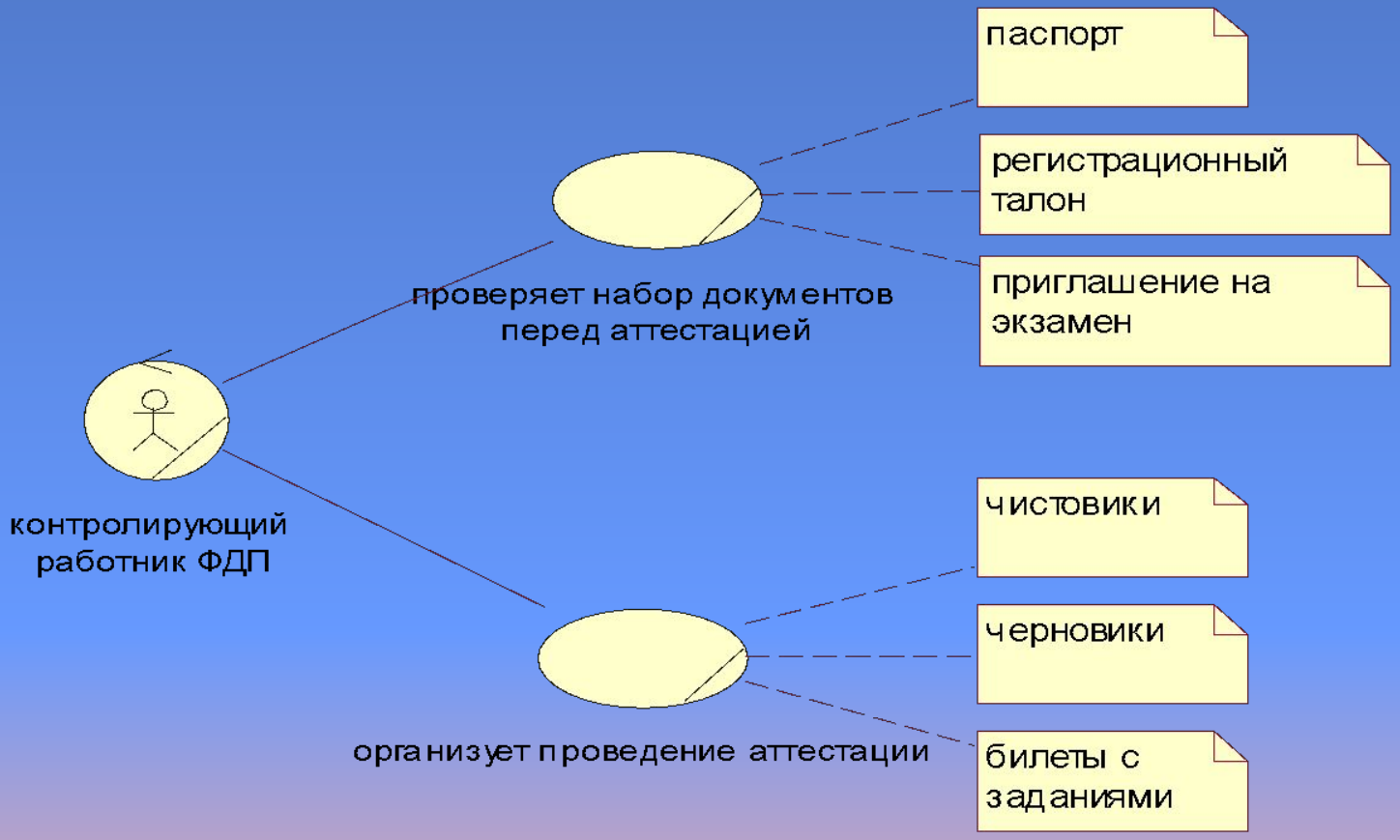
Пример модели «business use case model»



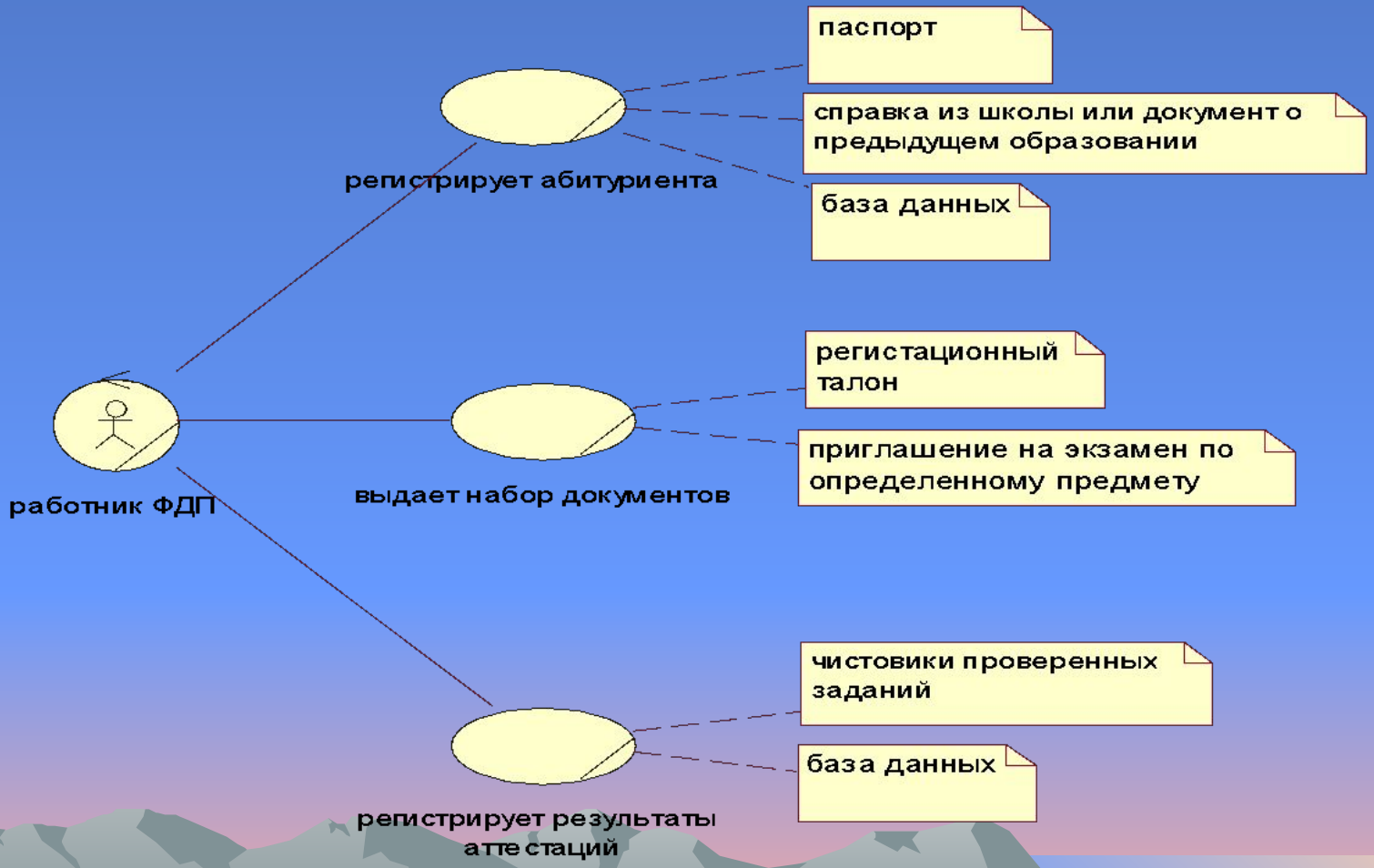
Пример модели «business use case model»



Пример модели «business use case model»



Пример модели «business use case model»



Модели взаимодействия субъектов и объектов (business object model)

Модели взаимодействия субъектов и объектов (business object model)

- Модели взаимодействия субъектов и объектов (business object model) используются для описания производственных функций предметной области, подлежащей автоматизации.
- Разрабатываются на этапе разработке программных средств (ПС).

Модели взаимодействия субъектов и объектов (business object model)

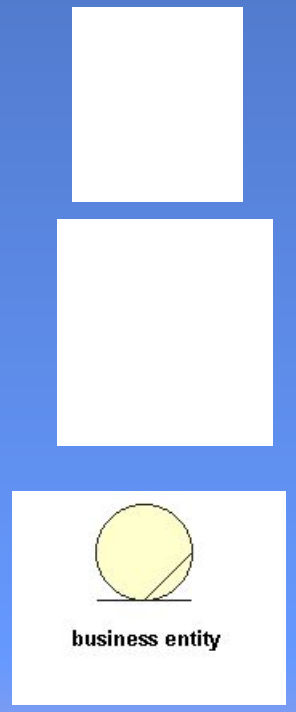
- **Модели взаимодействия субъектов и объектов (business object model)** используется для описание сценария выполнения производственных функций субъектами и объектами предметной области.

Модели взаимодействия субъектов и объектов

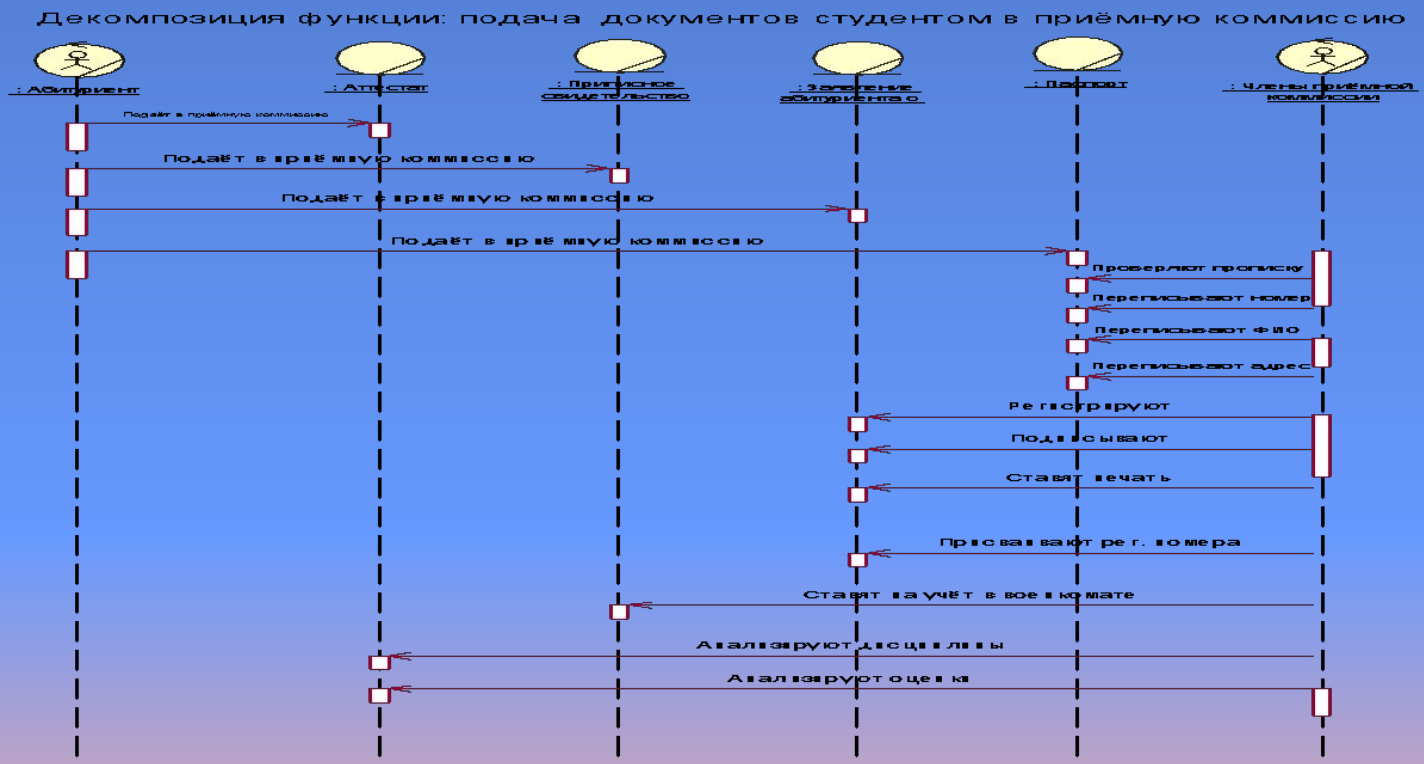
- **Для создания модели взаимодействия субъектов и объектов используются диаграммы последовательностей** (sequence diagram) **и/или взаимодействия** (collaboration diagram).

Элементы диаграмм

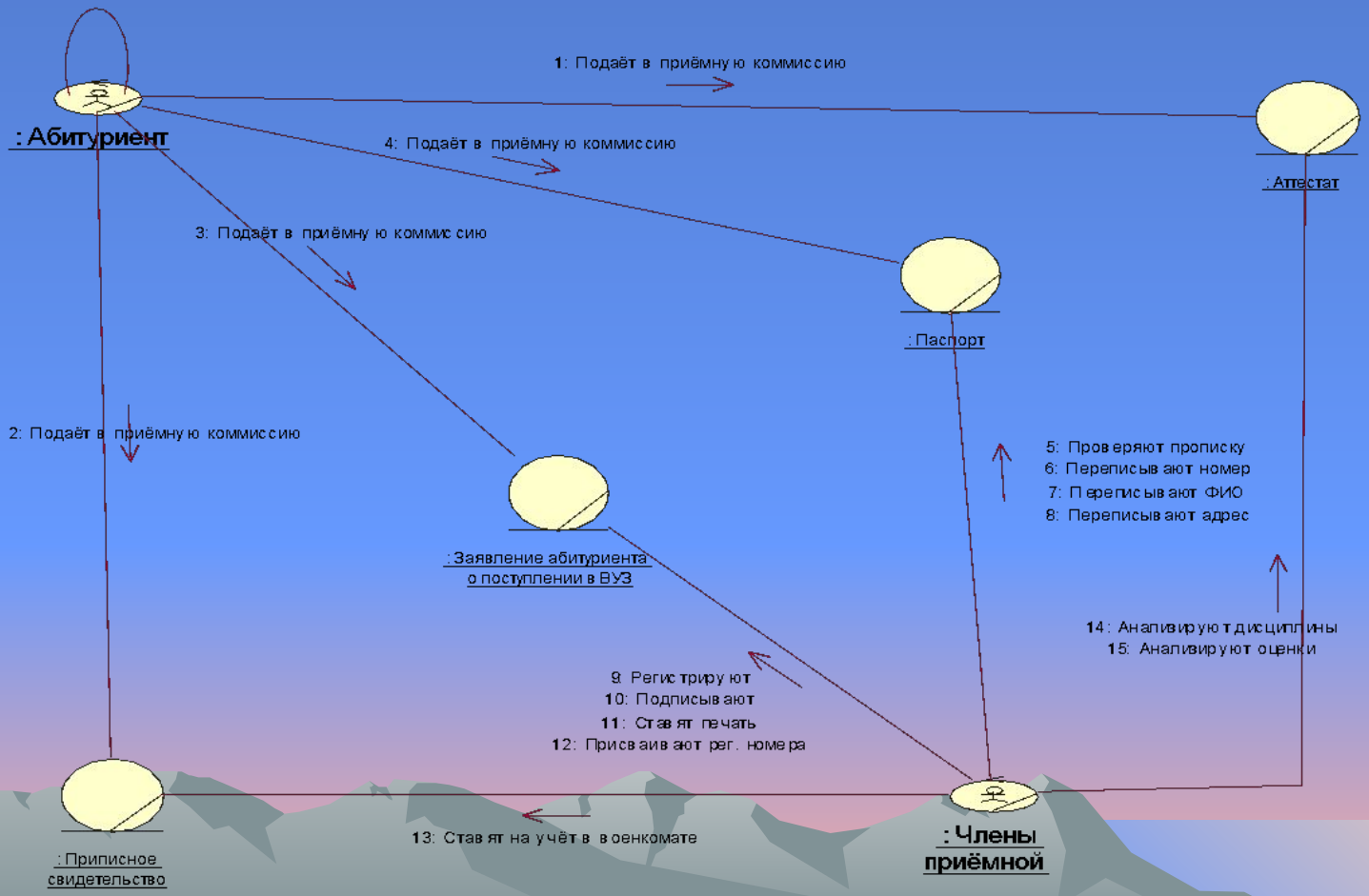
- Действующие лица производственного процесса (**business worker, business actor**).
- Сущности производственного процесса **(business entity).**
- Сообщения (**messages**).

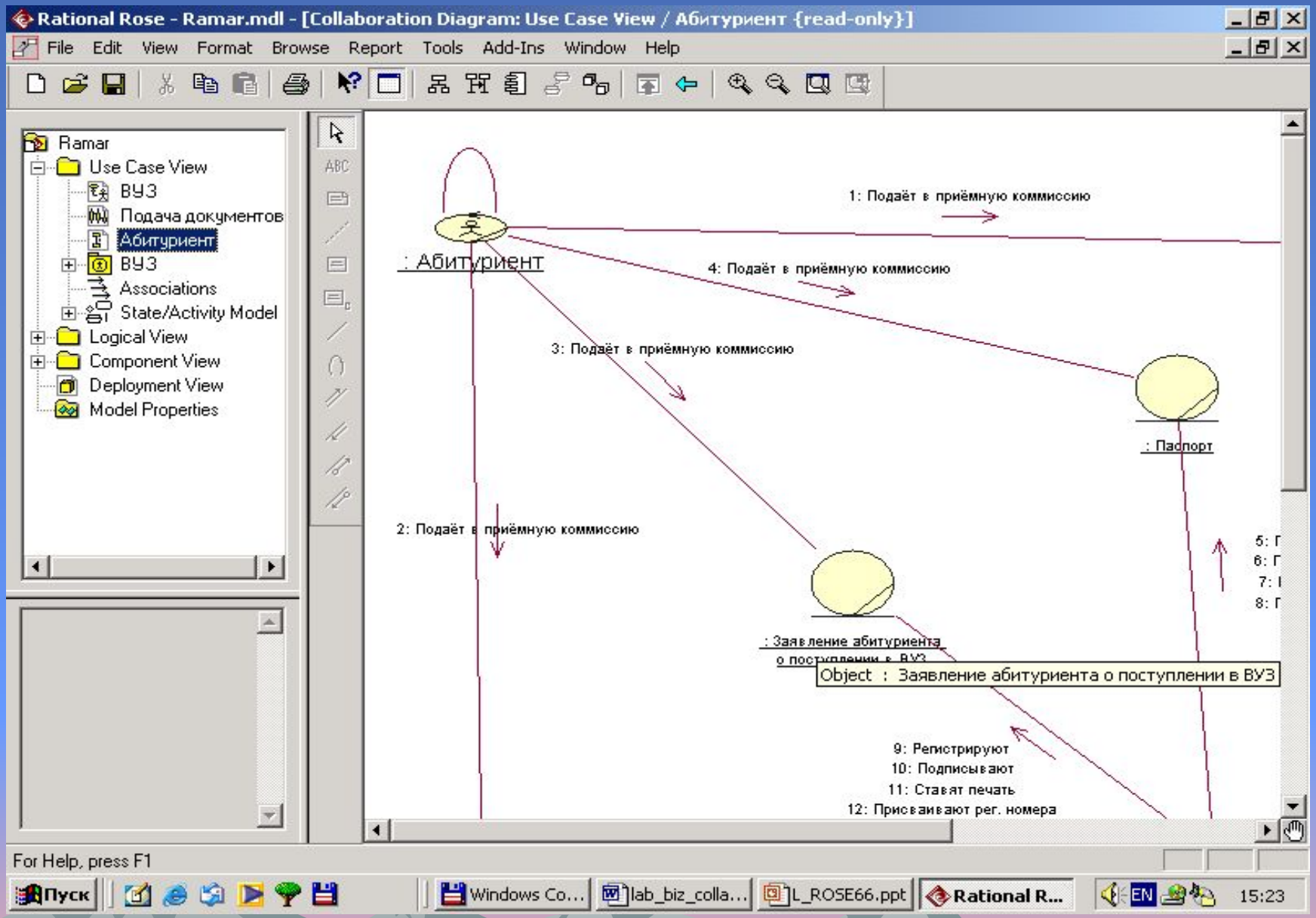


Пример диаграммы последовательностей (**sequence diagram**)



Пример диаграммы взаимодействия (**collaboration diagram**)




Rational Rose - Ramar.mdl - [Collaboration Diagram: Use Case View / Абитуриент {read-only}]
File Edit View Format Browse Report Tools Add-Ins Window Help
Ramar
Use Case View
ВУЗ
Подача документов
Абитуриент
ВУЗ
Associations
State/Activity Model
Logical View
Component View
Deployment View
Model Properties
: Абитуриент
1: Подаёт в приёмную коммиссию
4: Подаёт в приёмную коммиссию
3: Подаёт в приёмную коммиссию
2: Подаёт в приёмную коммиссию
: Паспорт
: Заявление абитуриента
Object : Заявление абитуриента о поступлении в ВУЗ
9: Регистрируют
10: Подписывают
11: Ставят печать
12: Присваивают рег. номера
For Help, press F1
Пуск
Windows Co...
lab_biz_colla...
L_ROSE66.ppt
Rational R...
EN
15:23

Классы и объекты. Диаграмма классов.

С точки зрения восприятия человеком
объектом может быть:
- Осязаемый или видимый предмет;
- Нечто, воспринимаемое мышлением;
- Нечто, на что направлена мысль или действие.

Объект моделирует часть окружающей действительности и таким образом существует во времени и пространстве.

Объект представляет собой конкретный опознаваемый предмет, единицу или сущность (реальную или абстрактную), имеющую **четко определенное функциональное** назначение в данной предметной области.

Существуют такие объекты, для которых определены **явные концептуальные границы**, но сами объекты представляют собой неосязаемые события и процессы.

Например, химический процесс на заводе
можно трактовать как объект.

Объекты могут быть осязаемы, но иметь размытые физические границы: реки, толпа.

Можно дать еще одно определение **объекта**:
Объект обладает **состоянием**, **поведением** и **идентичностью.**

Структура и поведение схожих объектов определяет общий для них класс.

Термины экземпляр класса и объект идентичны.

Состояние объекта определяется **набором свойств** или **атрибутами и связями**, которые может иметь объект с другими объектами.

Является типичным изменение состояния объекта во времени.

Поведение

Объекты не существуют изолированно, а подвергаются воздействию или сами воздействуют на другие объекты.

Определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию называется **операцией**.

В объектно-ориентированных языках **операции**, выполняемые над объектом, называются **методами**.

Поведение - это то,
как объект действует и реагирует.

Поведение выражается
в терминах **состояния** объекта
и **передачи сообщения**.

Идентичность

Идентичность это такое свойство объекта, которое отличает его от всех других объектов.

Понятия **объекта и класса** настолько тесно связаны, что невозможно говорить об объекте безотносительно к его классу.
Однако существует важное различие
этих двух понятий.

В то время как объект обозначает **конкретную сущность**, определенную во времени и пространстве, класс определяет **абстракцию существенного в объекте**.

Класс есть описание группы объектов, обладающих общими свойствами (**атрибутами**), общим **поведением**, общими **связями** с другими объектами и общей **семантикой**.

Класс является шаблоном для создания объекта.
Каждый объект является экземпляром некоторого класса.

Принято в языке **UML** обозначать класс **прямоугольником**, состоящим из трех частей, как представлено на рис. 1.

В верхней части прямоугольника указывается **название класса**, в средней - **атрибуты**, в нижней - **операции**.

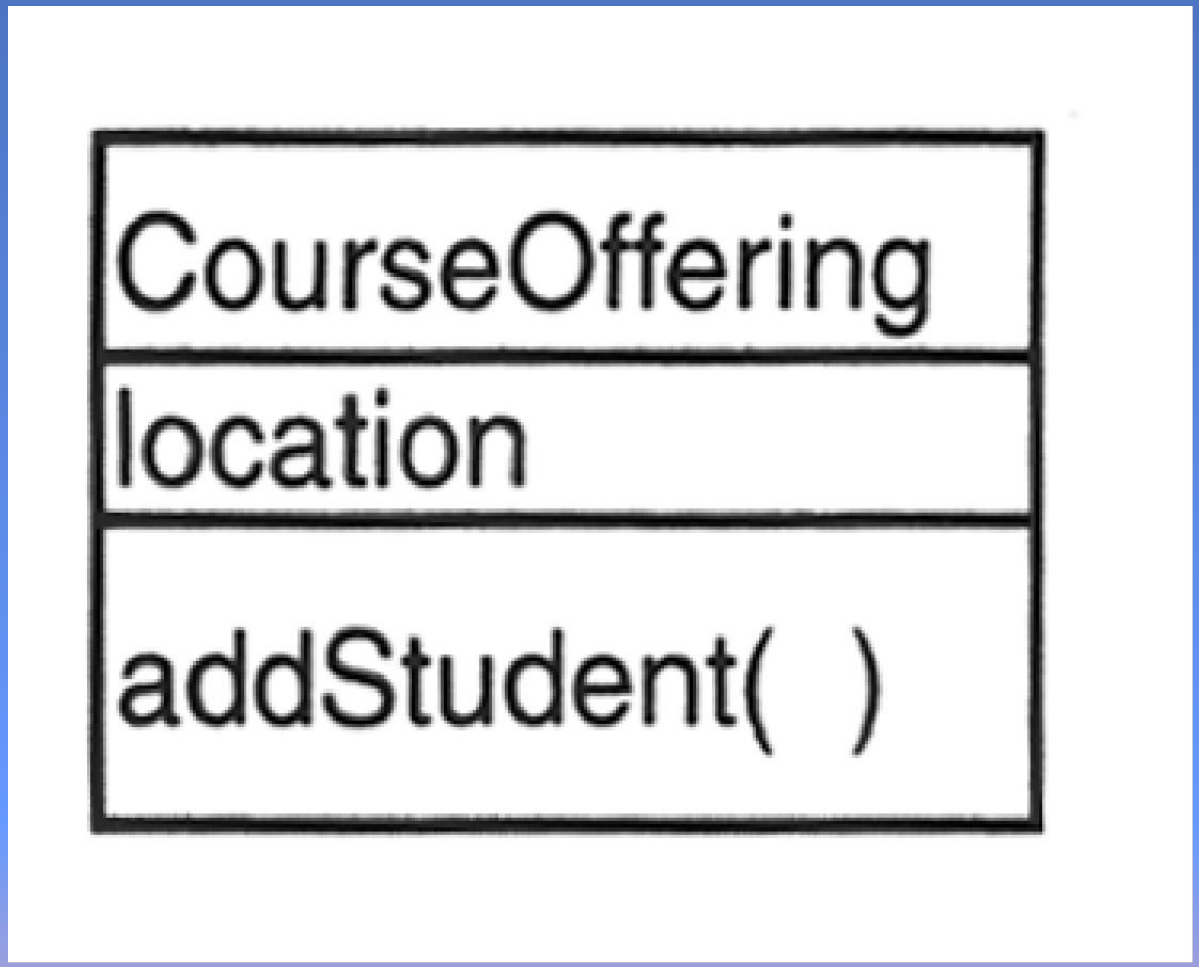


Рис. 1. Пример обозначения класса языке UML

Для описания классов в **Rational Rose**, как и для описания взаимодействия между функциями на диаграммах **Use Case** используются **стереотипы**.

Стереотипы позволяют конструировать новые виды классов.

Существуют следующие **стереотипы** классов:

- **entity** (сущности). Данный класс представляет абстракции сущностей реального мира;

- **boundary** (интерфейс). Класс используется для отражения взаимодействия системы и ее окружения или взаимодействия внутри системы;

- **control** (управление).
Класс используется для описания алгоритмов функционирования системы.
Хорошей практикой является создание для управляющих классов диаграмм состояний и переходов;

- **utility** (служебные классы, используемые для выполнения вспомогательных функций непосредственно, не относящихся к работе системы, например резервное копирование);

- **exception** (классы для обработки исключительных ситуаций, например ошибок).

В описании класса на языке **UML имя стереотипа** размещается над именем класса в двойных скобках <<>>, как представлено на рис. 2.

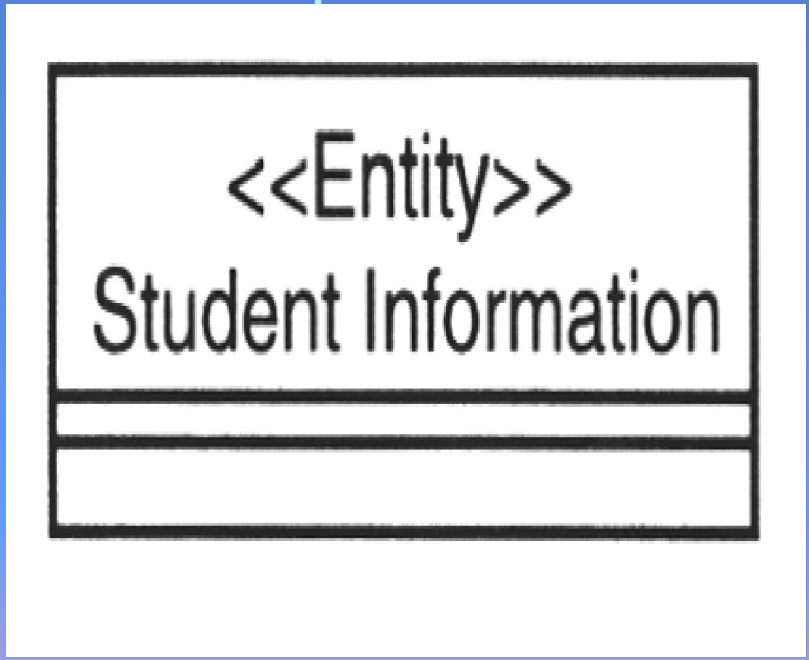


Рис. 2. Пример класса со стереотипом <<Entity>>

Классы и объекты не существуют изолированно, они могут взаимодействовать друг с другом.

Существуют следующие основные виды отношений или связей между классами:

- ассоциация;
- агрегация;
- наследование.

Ассоциация (Association Relationships).

Ассоциация есть **смысловая связь**. Связь является двунаправленной. Связь не объясняет, как классы общаются друг с другом, отмечается только **смысловая зависимость между классами**.

В **UML** эта связь изображается сплошной прямой линией как представлено на рис. 3.

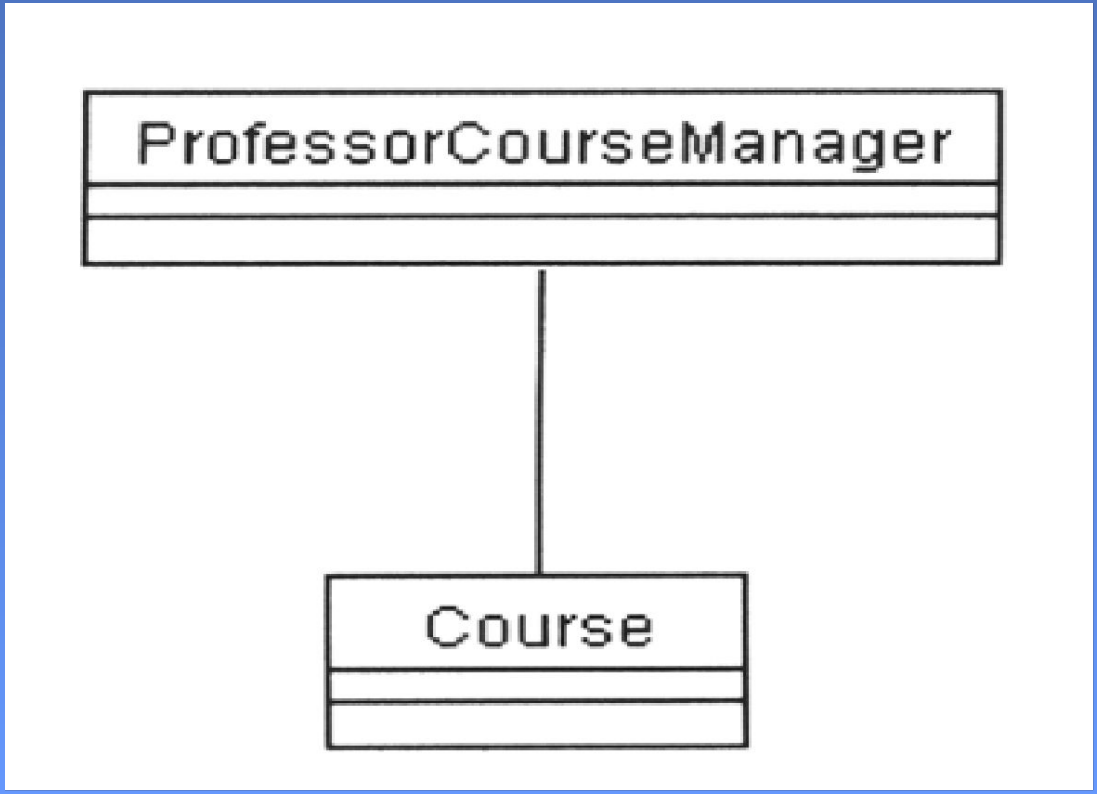


Рис. 3. Пример изображения ассоциативной связи в UML

Ассоциация может быть поименована.

Имя ассоциации обозначается глаголом, например, учит, управляет.

Рекомендуется указывать имя ассоциации так, чтобы оно читалось корректно
слева направо или сверху вниз, например как представлено на рис. 4.

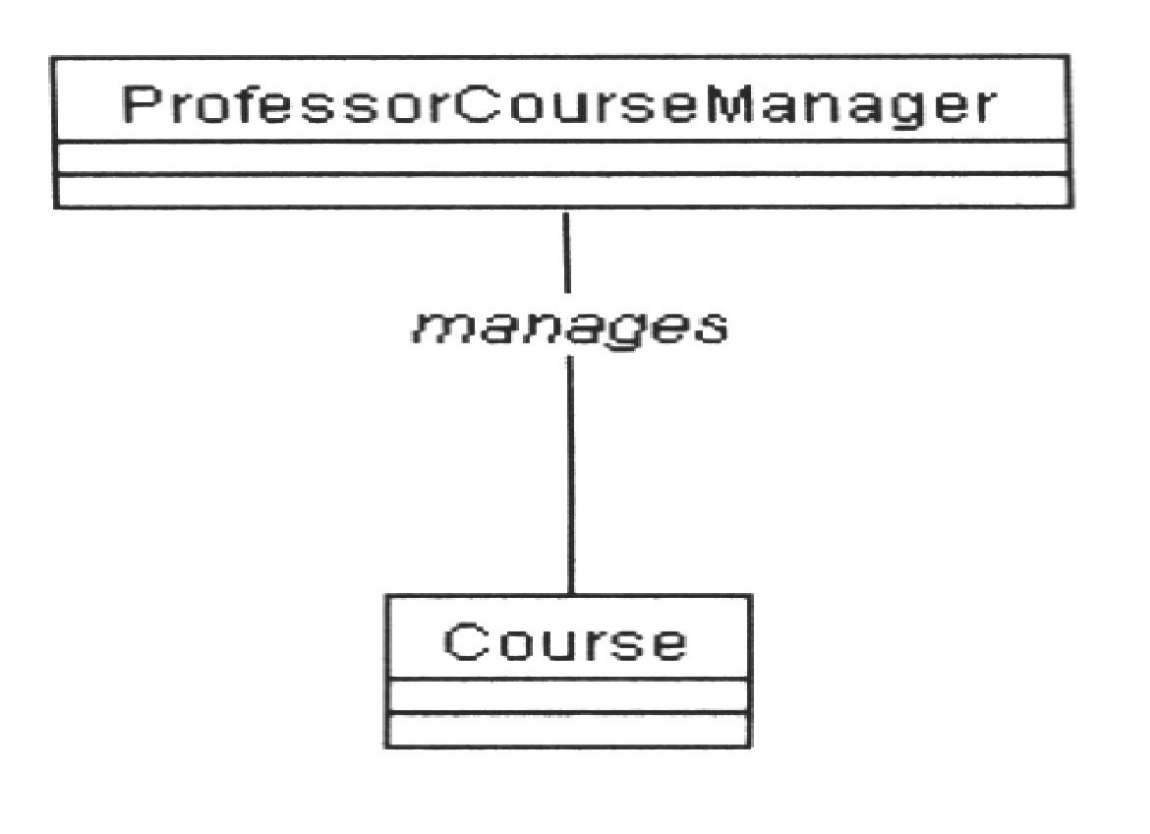


Рис. 4. Пример изображения поименованной ассоциативной связи в UML

Состояние и поведение класса определяет исполняемые им **роли**.

Роли могут размещаться как на одном конце связи, так и на обоих концах.

Имя роли помещается на связи рядом с классом.

Роль может быть использована вместо имени связи.

Если требуется, указывается и **наименование связи** и **роли**.

На рис. 5 представлен пример ассоциативной связи с ролью.

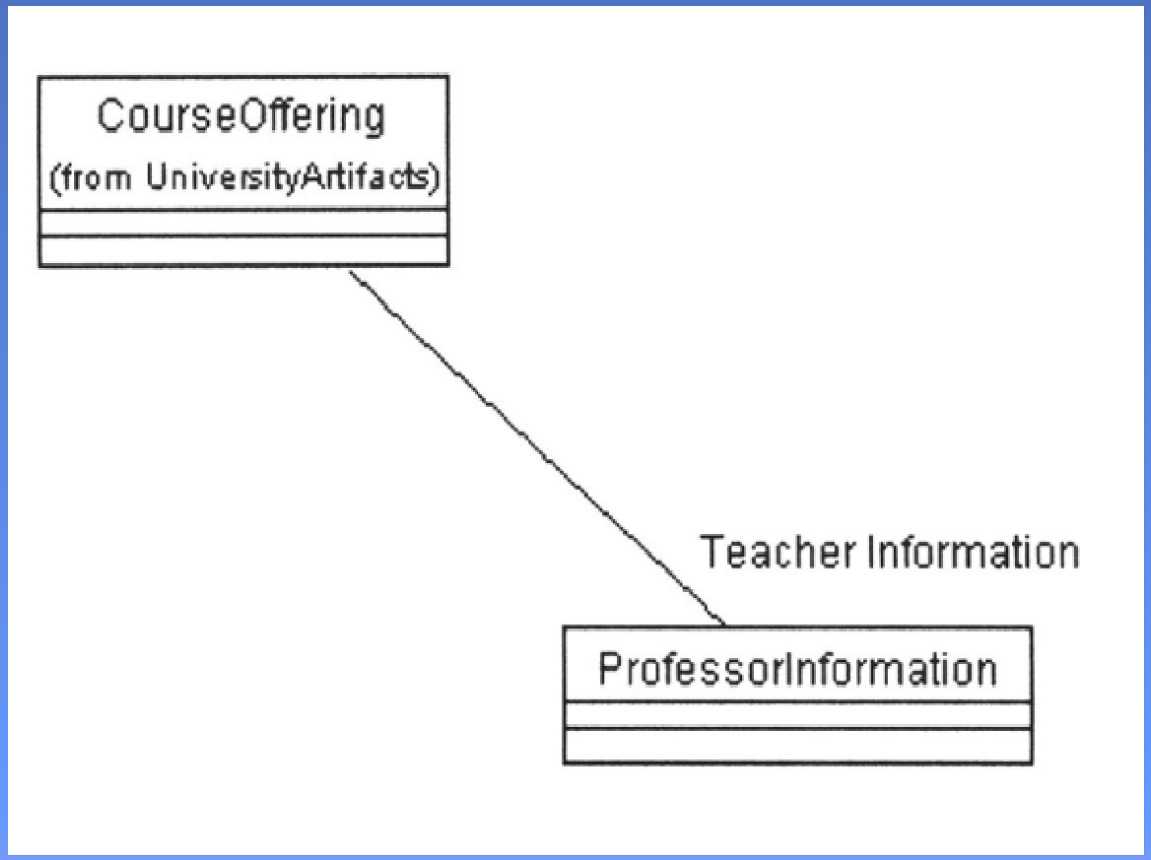


Рис. 5. Пример изображения ассоциативной связи с ролью в UML

Количество объектов класса, принимающих участие в связи называется **мощностью связи**.

Мощность указывается на каждом конце ассоциативной связи.

Мощность означает число связей между одним объектом в начале линии связи с объектами в конце линии связи.

Мощность может обозначаться следующим образом:

1 - точно один объект;

0...* - ноль или больше объектов;

1..*- один или больше объектов;

0..1 - ноль или один объект;

5..8 - специфический диапазон 5,6,7,8;

4..7,9 - комбинация 4,5,6,7, или 9 объектов.

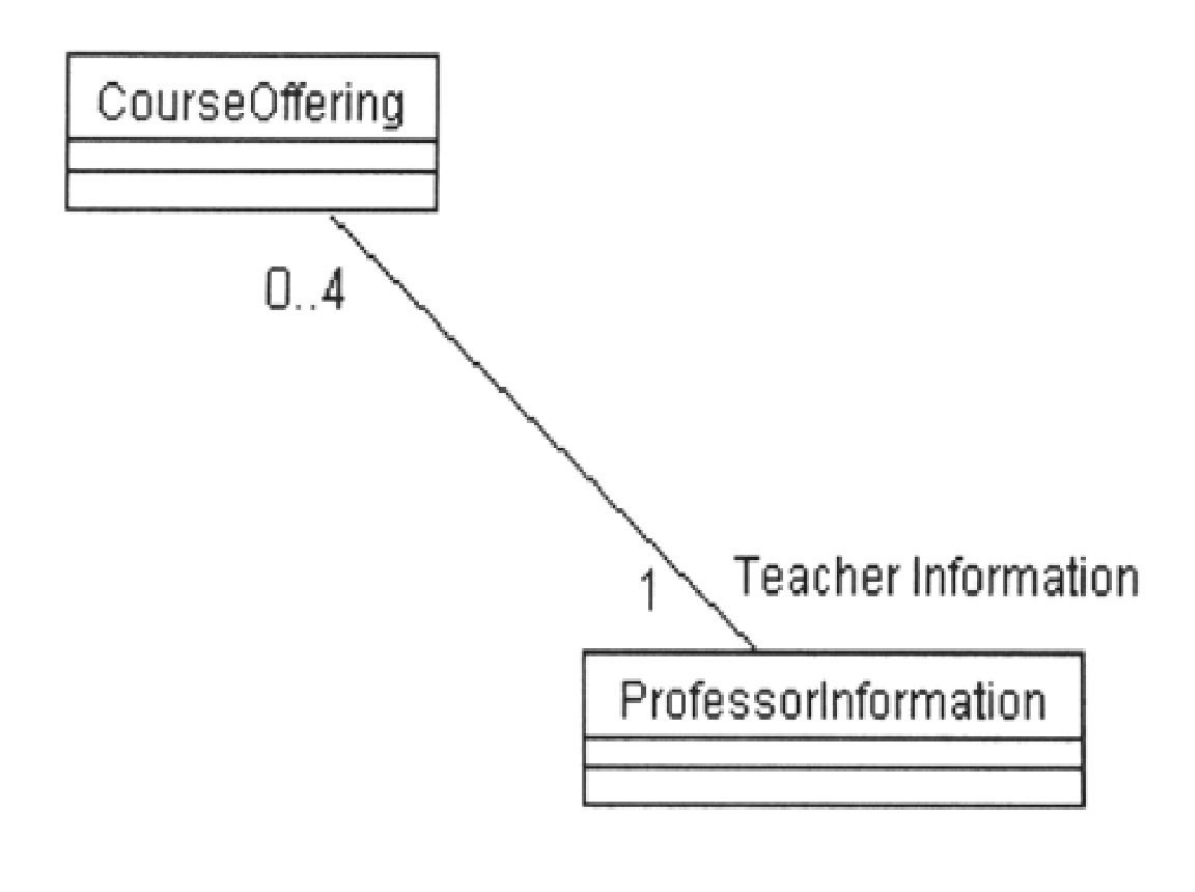


Рис. 6. Пример изображения ассоциативной связи с мощностью и ролью в UML

Представленную на рис. 6 связь классов следует интерпретировать как:

- один объект класса **CoursOffering** связан ровно с одним объектом класса **ProfessorInformation**. Например, курс "Математика 101, Часть 1" читается профессором Смитом.

- один объект класса **ProfessorInformation** связан от 0 до 4 объектов класса **CoursOffering**.

Например, профессором Смитом читается "Математика 101, Часть 1", "Алгебра 200, Часть 2".

Класс может иметь **ассоциативную связь с самим собой**.
Такая ассоциативная связь называется **рефлексивной**.

Пример ассоциативной рефлексивной связи представлен на рис. 7.

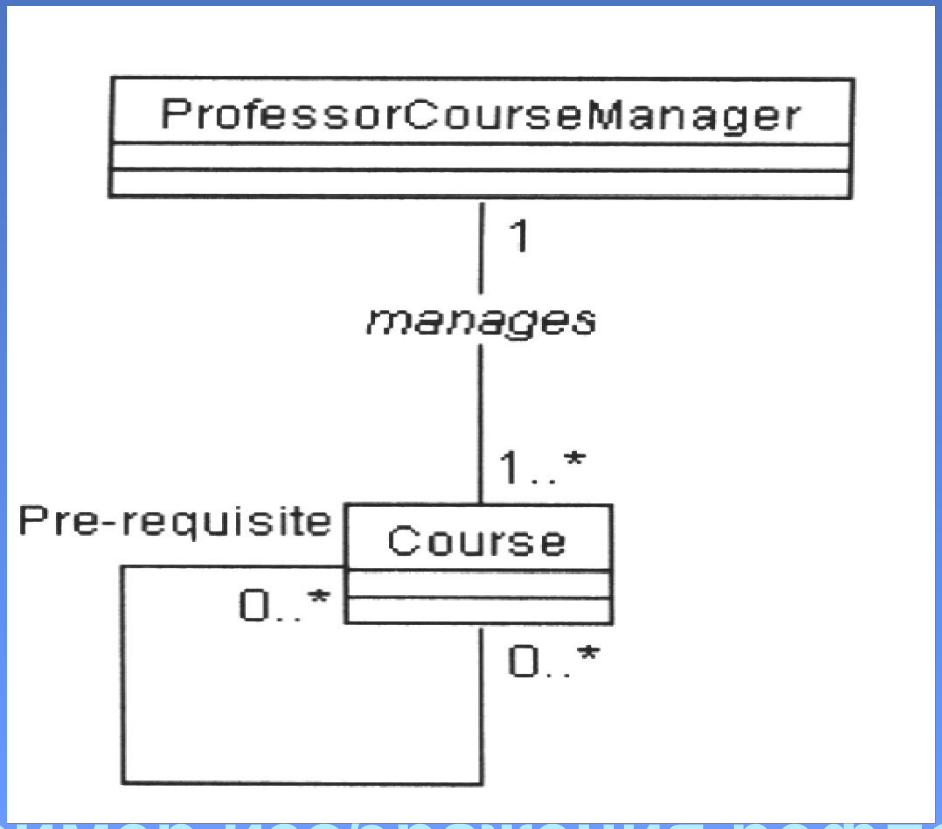


Рис. 7. Пример изображения рефлексивной ассоциативной связи с мощностью и ролью в UML

Представленную на рис. 7 связь классов следует интерпретировать как:

Один курс является предпосылкой для чтения от нуля и более курсов.

Для чтения одного курса предпосылками являются от 0 и более курсов.

В некоторых случаях **связь** может **иметь структуру и поведение**.

Это справедливо, когда информация относится одновременно к двум объектам, а не к одному.
Такая связь называется **ассоциативным классом**.
Пример ассоциативного класса представлен на рис. 8.

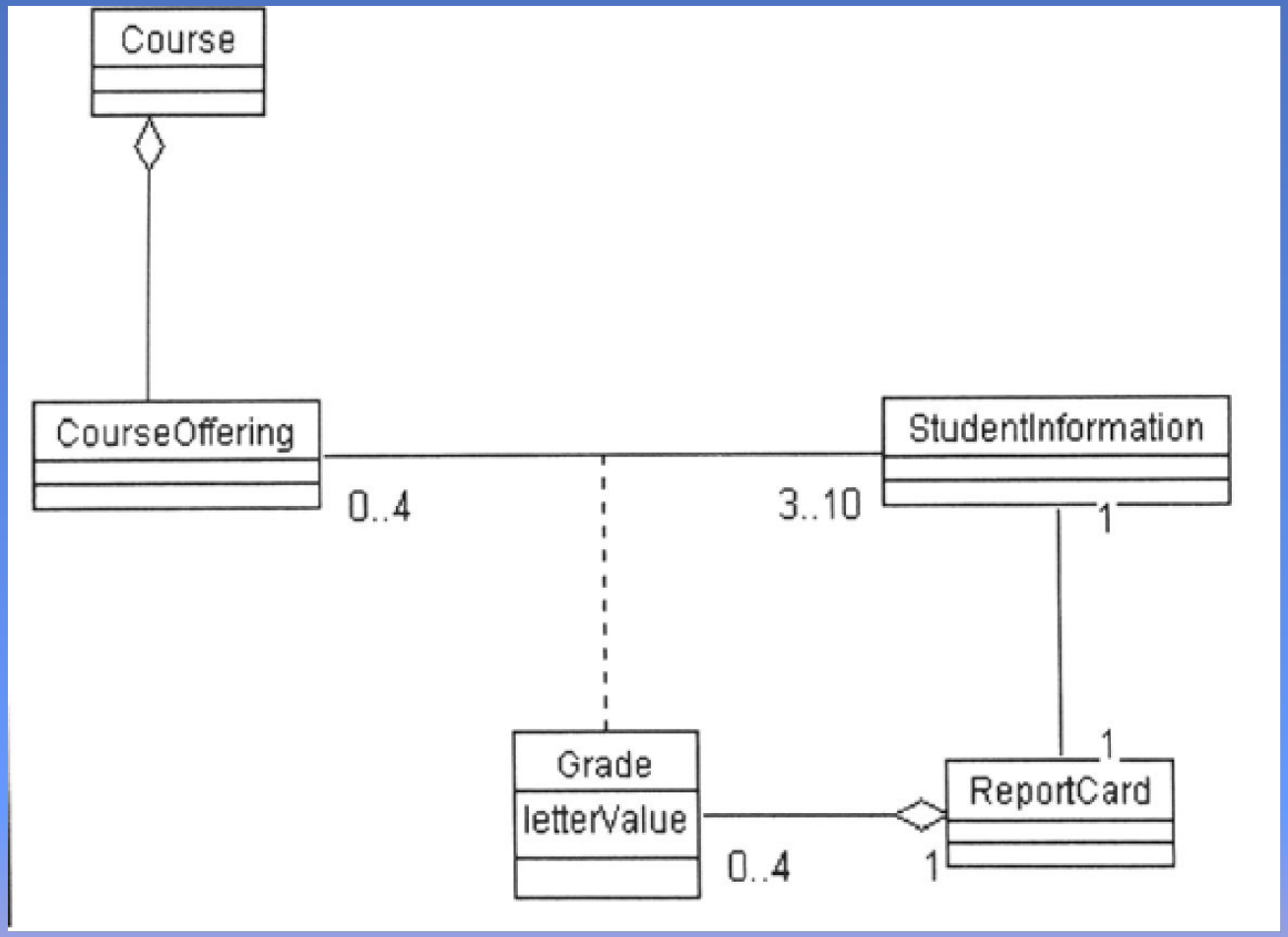


Рис. 8. Пример ассоциативного класса Grade

Например, один студент может изучать от 0 до 4 частей одного курса.
Одна часть курса может изучаться от 3 до 10 студентов.
Каждый студент должен получит оценку за каждую часть курса.
Где должны содержаться оценки.
Оценки должны принадлежать к классу, который присоединяется к связи между двумя классами студент и часть курса.

Агрегация (Aggregation Relationships)

Агрегация обозначает связь **часть целого** (**part of**).

Например, самолет состоит из крыльев, двигателей, шасси и прочих частей.

В **UML** эта связь изображается сплошной прямой линией с добавлением на конце ромба как представлено на рис. 9.

Ромб указывает на целое.

Рис. 9. Пример изображения агрегации в UML

Представленную на рис. 9 связь классов следует интерпретировать как:

Курс имеет части, например, курс "Математика 101" имеет части "Математика 101, Часть 1", "Математика 101, Часть 2".

Агрегация есть частный случай **ассоциации**.

Наследование (Generalize/Inherits Relationship)

Наследование это такое отношение между классами,
когда один класс повторяет структуру и поведение другого класса –
одиночное наследование
или других
множественное наследование классов.

Класс, структура и поведение которого наследуется, называется суперклассом.

Подкласс обычно расширяет или ограничивает существующую структуру и поведения класса. Наследование это связь **"is a"**.

Так как наследование не является связью между разными объектами,
она может **не именоваться**,
на ней не указываются роли и мощность.

Не существует ограничения в числе уровней наследования.

Однако имеется мнение ограничивать число уровней наследования **тремя - пятью**.

В **UML** наследование изображается стрелкой с не закрашенным треугольником, обращенным к суперклассу.

В случае, когда **много подклассов** наследуют свойства суперкласса для обозначения иерархии удобно использовать **дерево иерархии**.

В **UML** дерево иерархии изображается, как представлено на рис. 10, 11, 12.

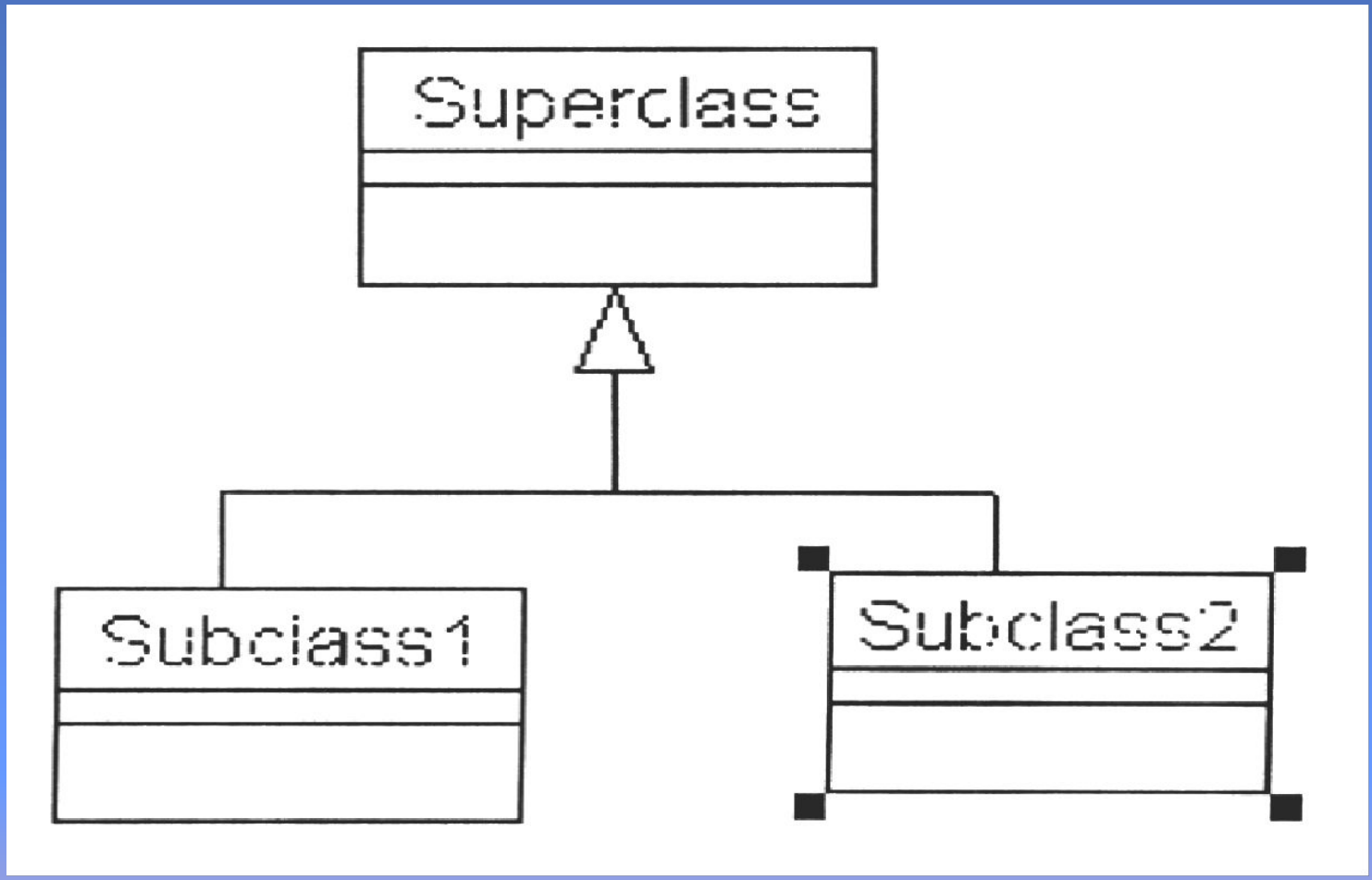


Рис. 10. Пример изображения наследования в UML с использованием дерева иерархии

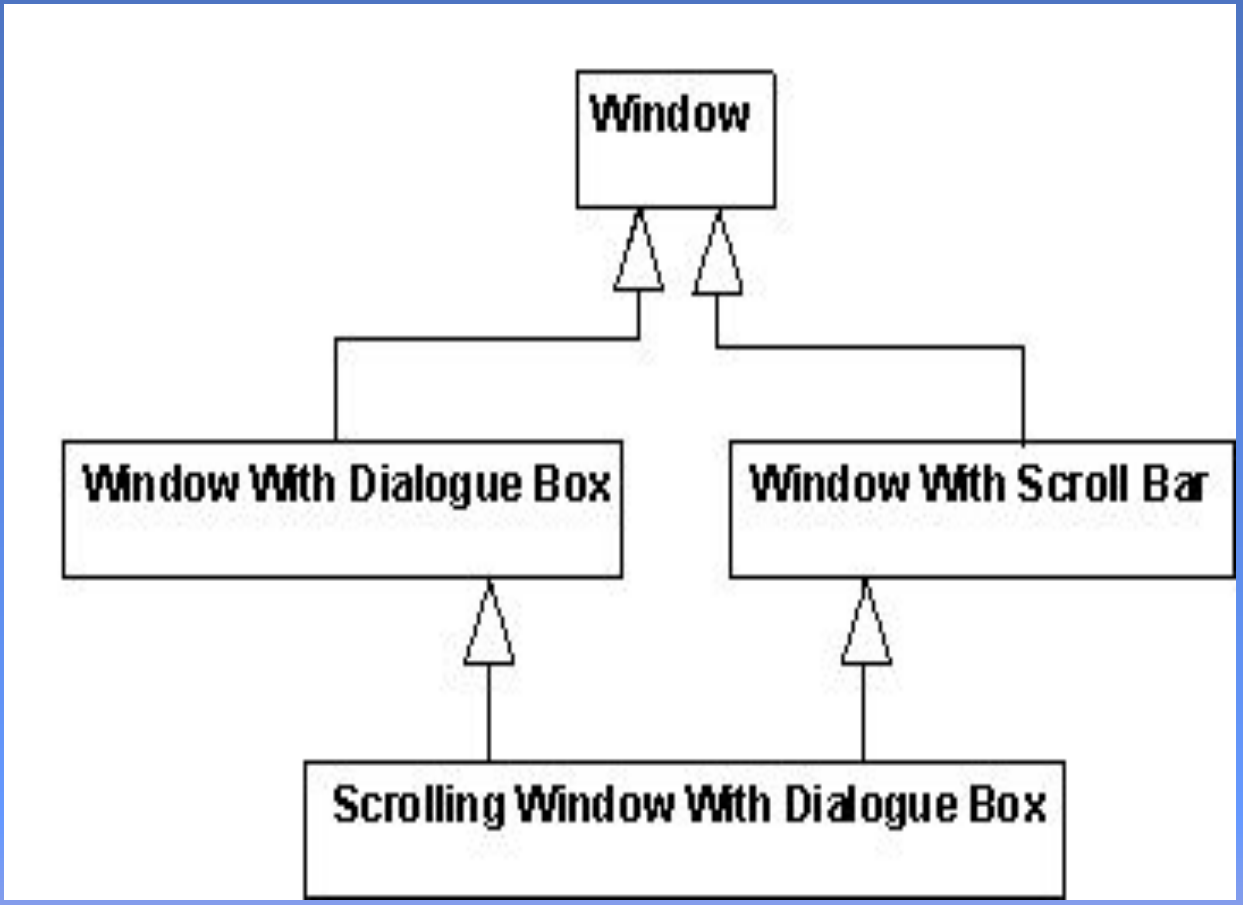


Рис. 11. Пример изображения множественного наследования в UML

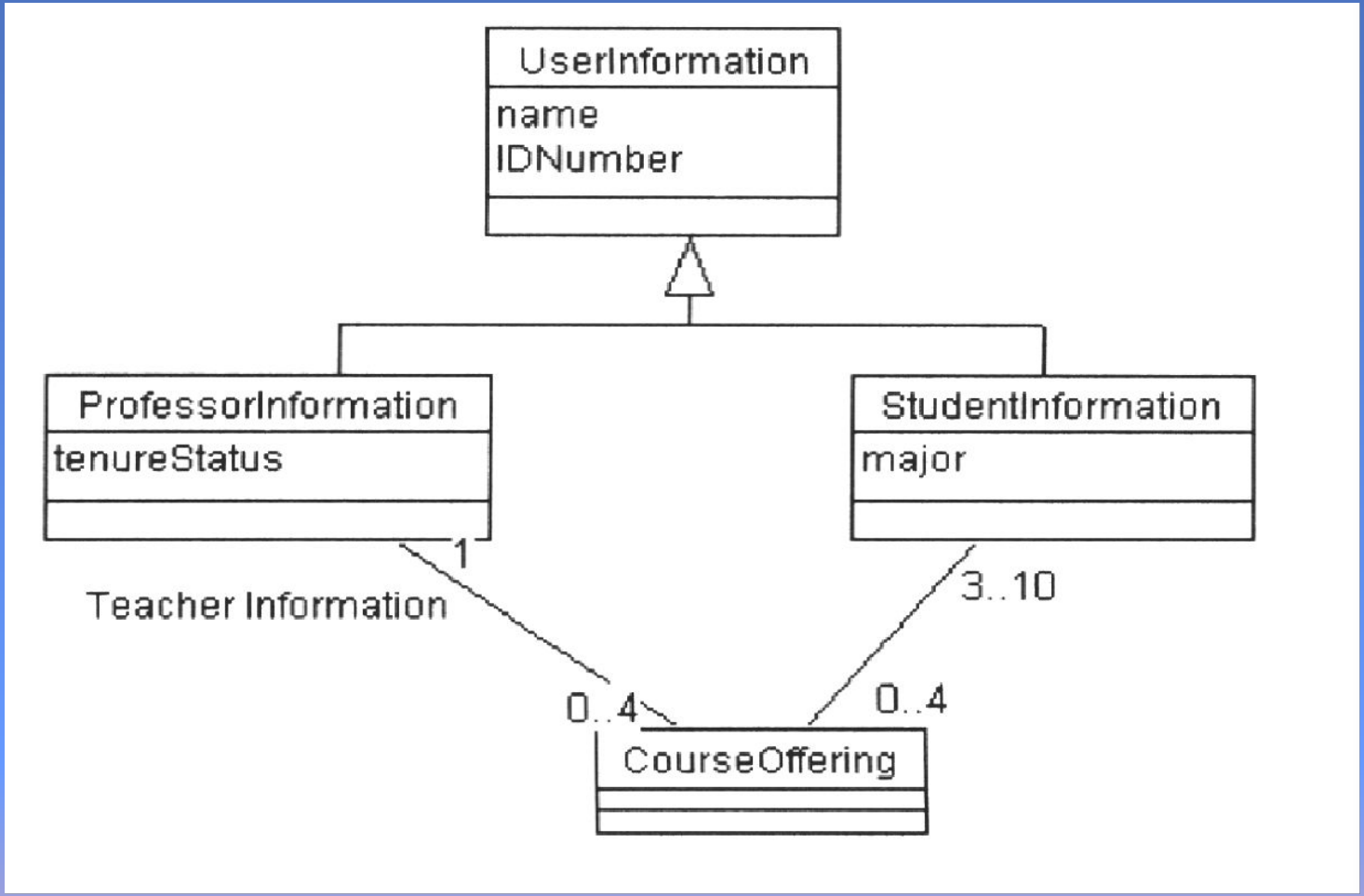


Рис. 12. Пример изображения наследования в UML

Диаграммы классов

Диаграмма классов показывает **классы** и их **отношения**.

Диаграмма классов создается для отражения **всех** или **некоторых** классов в модели.

Диаграмма классов используется на **стадии анализа** чтобы отображать понятия (роли и обязанности сущностей) изучаемой предметной области.

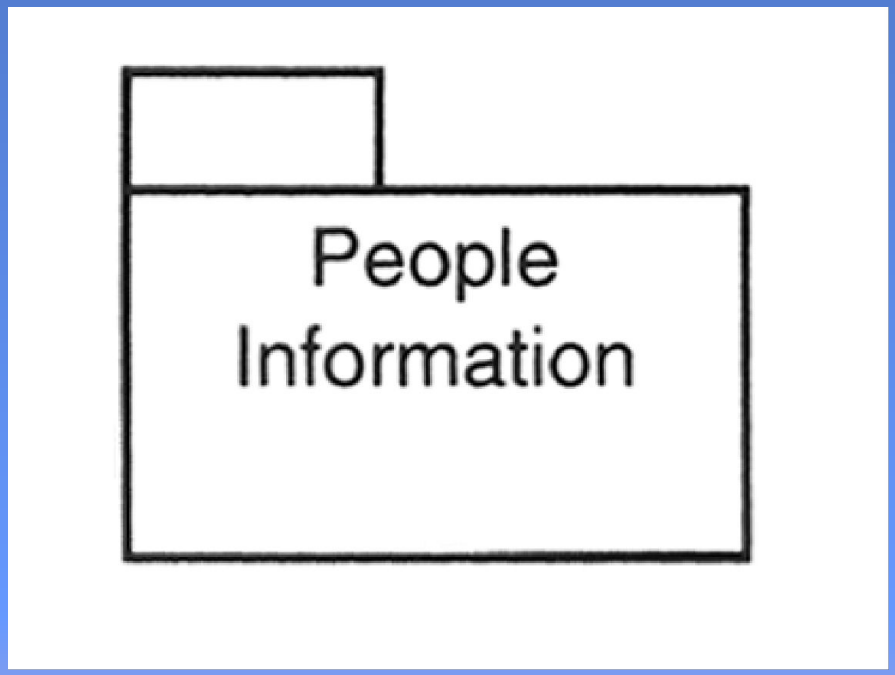
Диаграмма классов также может использоваться на **стадии проектирования** - чтобы передать **структуру классов, формирующих архитектуру системы**.

Главная диаграмма классов Main Class Diagram есть совокупность **пакетов** классов.

Каждый пакет имеет **свою главную диаграмму классов Package Main Class Diagram**, состоящую из пакетов и классов.

Создаются и другие диаграммы классов, например, детализирующие структуру и поведение одного или более классов в подсистеме, отражающие иерархию наследования.

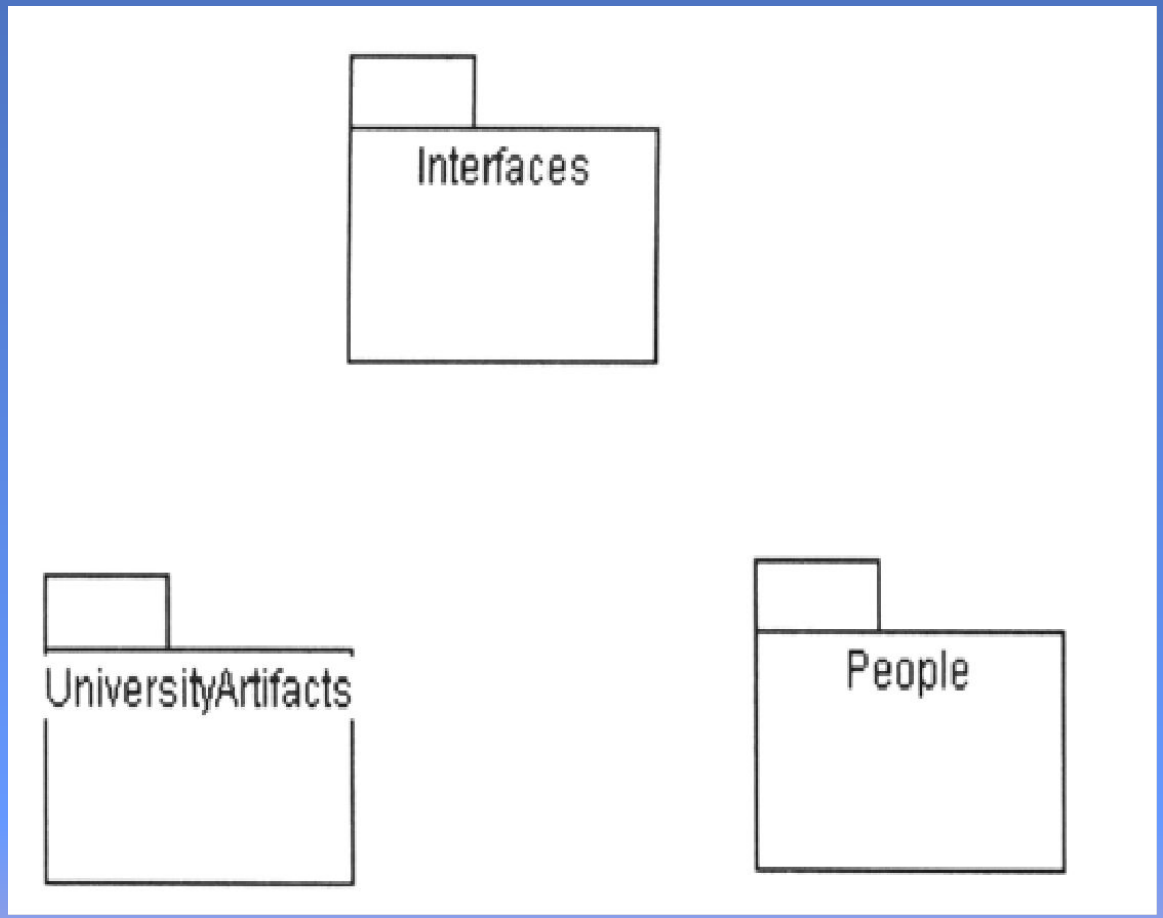
Изображение пакета классов в UML



Моделирование Главной диаграммы классов Main Class Diagram с использованием Rational Rose

Rational Rose а**втоматически** создает главную диаграмму классов и помещает ее в раздел логического проектирования модели.

Пример главной диаграммы классов



Диаграммы компонент

Диаграммы компонент

Диаграмма компонент отражает физическую структуру модели. **Диаграмма компонент** отражает организацию и связи среди компонент программного обеспечения, таких например, как **исходные тексты программ, объектные модули, исполняемые модули, библиотеки динамической компоновки.**

Диаграмма компонент включает:

1. Подсистемы компонент;
2. Собственно компоненты;
3. Интерфейс;
4. Связи между компонентами.

Подсистемы

Большие системы могут быть разложены на несколько сотен, даже **тысячи модулей**. Пытаться разобраться в физической структуре такой системы без ее **дополнительного структурирования** практически невозможно.

Подсистемы представляют собой совокупности логически **связанных модулей или компонент**.

Подсистемы модулей обозначаются в **UML** аналогично подсистемам классов и функций с использованием **пакета**.
В качестве **имени** подсистемы используется **имя директории**, в которой хранятся компоненты.

Подсистемы модулей могут иметь между собой **связи**.

Компоненты

Компонентами являются исходные тексты программ, объектные модули, исполняемые модули, библиотеки динамической компоновки.

На диаграммах компонента обозначается, как представлено на рис. 1.

В качестве имени компоненты используется **имя файла**, в котором храниться компонента.

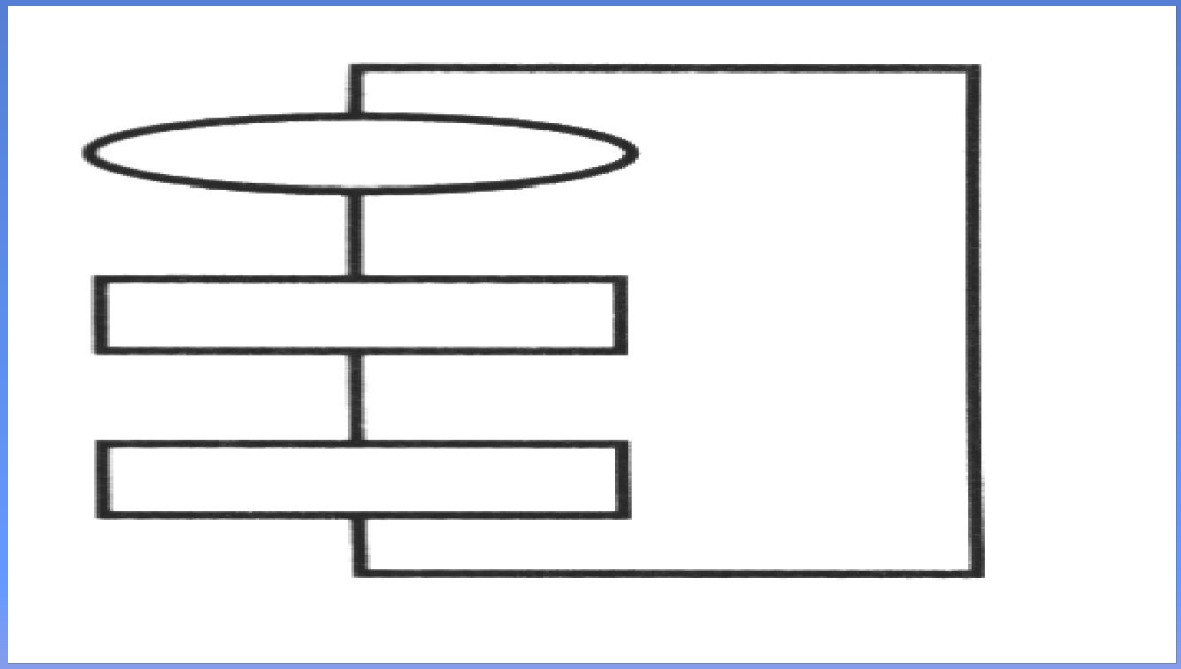


Рис. 1. Пример обозначения компоненты

Для указания различных назначений компонент используются **стереотипы**.

В настоящее время в **Rational Rose** поддерживаются следующие стереотипы компонент.

1. Подсистема;

2. Главная программа (файл, содержащий корневую программу);

3. Подпрограмма;

4. Задача (независимая по управлению подсистема или модуль автономной загрузки);

5. Исполняемый модуль;

6. Библиотека динамической компоновки.

На **диаграммах компонент** существуют и другие обозначения для компонент: **главная программа**, **подпрограмма**, **задача**.

Главная программа обозначается, как представлено на рис. 2.

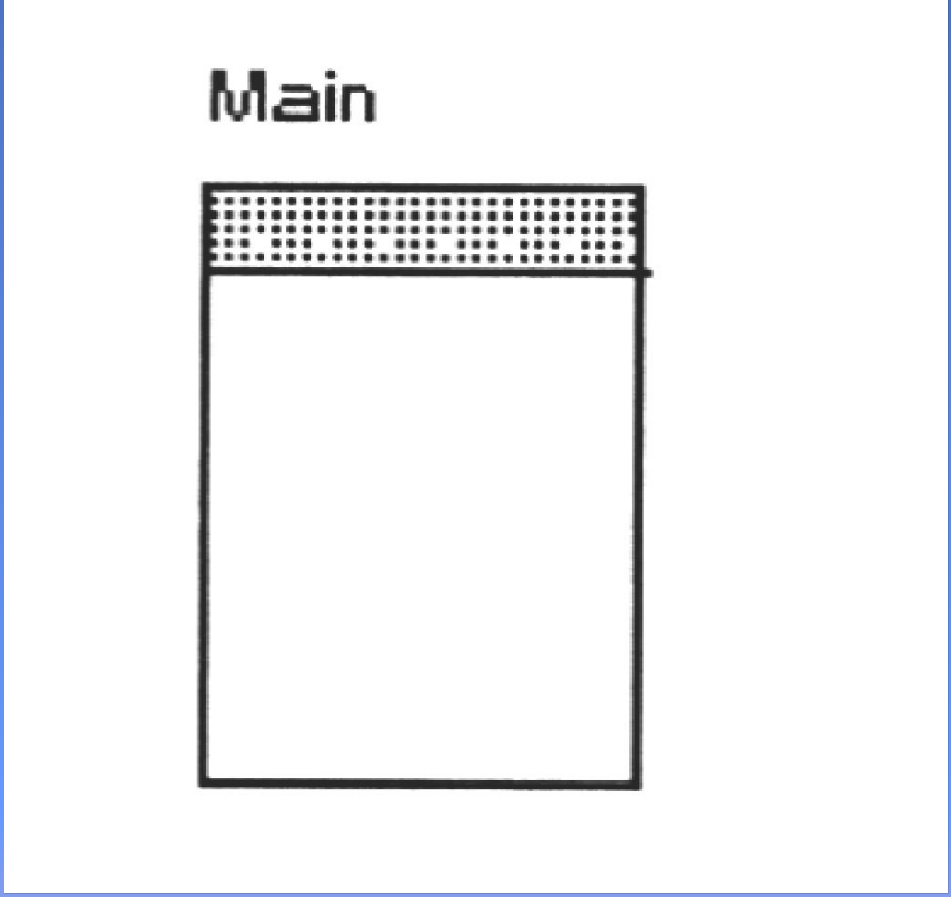


Рис. 2. Пример изображения главной программы

Обозначение **спецификации подпрограммы** и **тела подпрограммы** на диаграммах компоновки представлено на рис. 3.

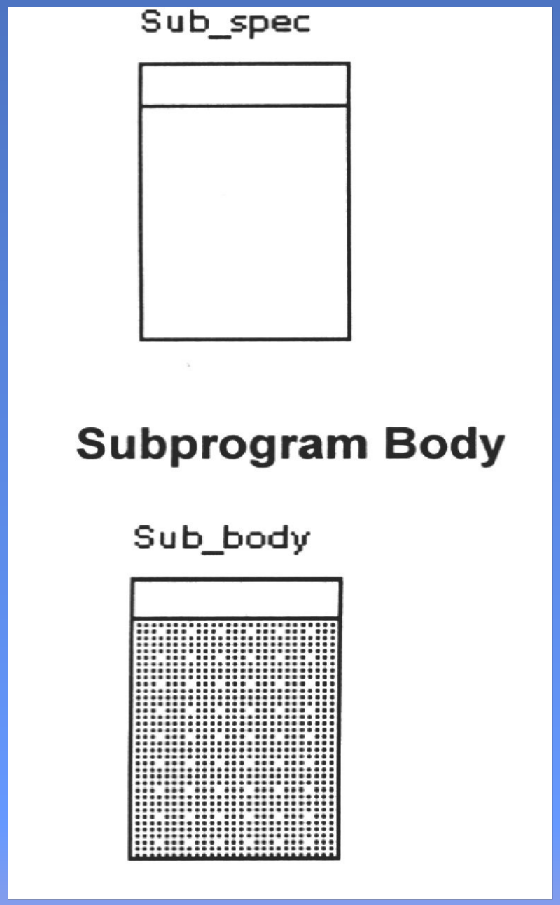


Рис. 3. Обозначения спецификации подпрограммы и тела подпрограммы

Обозначение **спецификации задачи** и **тела задачи** на диаграммах компоновки представлено на рис. 4.

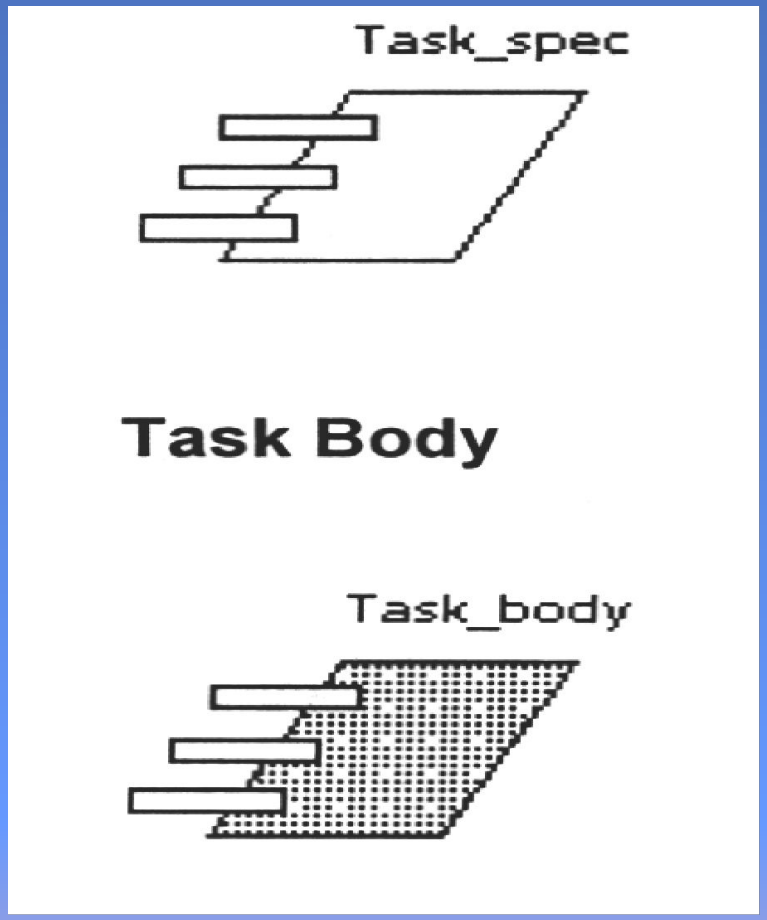


Рис. 4.Обозначение спецификации подпрограммы и тела подпрограммы

Исполняемые модули на диаграммах компонент обозначаются как представлено на рис. 4 (**Task_spec**).

Библиотеки динамической компоновки обозначаются на диаграммах как представлено на рис. 1.

Интерфейс

Интерфейс определяет ограниченную часть компоненты, связанную с представлением операций пользовательского интерфейса. На диаграммах интерфейс обозначается **маленьким кружком с линией**, направленной к компоненте, реализующий этот интерфейс (рис. 5).

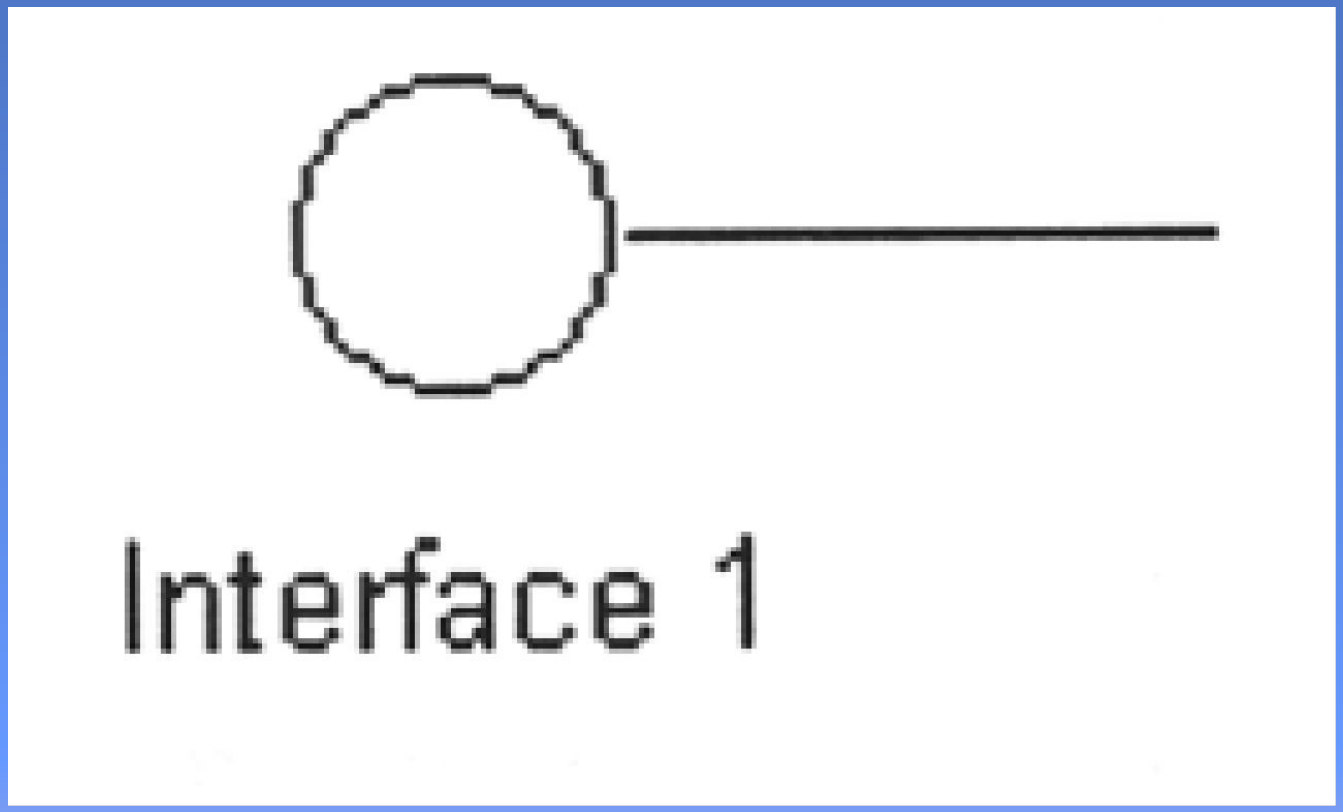


Рис. 5. Обозначение интерфейса на диаграммах компонент

Связи

Между компонентами или модулями может существовать **связь**. Связь которая, которая существует между модулями есть **компиляционная связь**. На диаграммах связь обозначается прерывистой стрелкой, выходящей из зависимого модуля.

На рис. 6 представлен пример диаграммы компонент со связями.

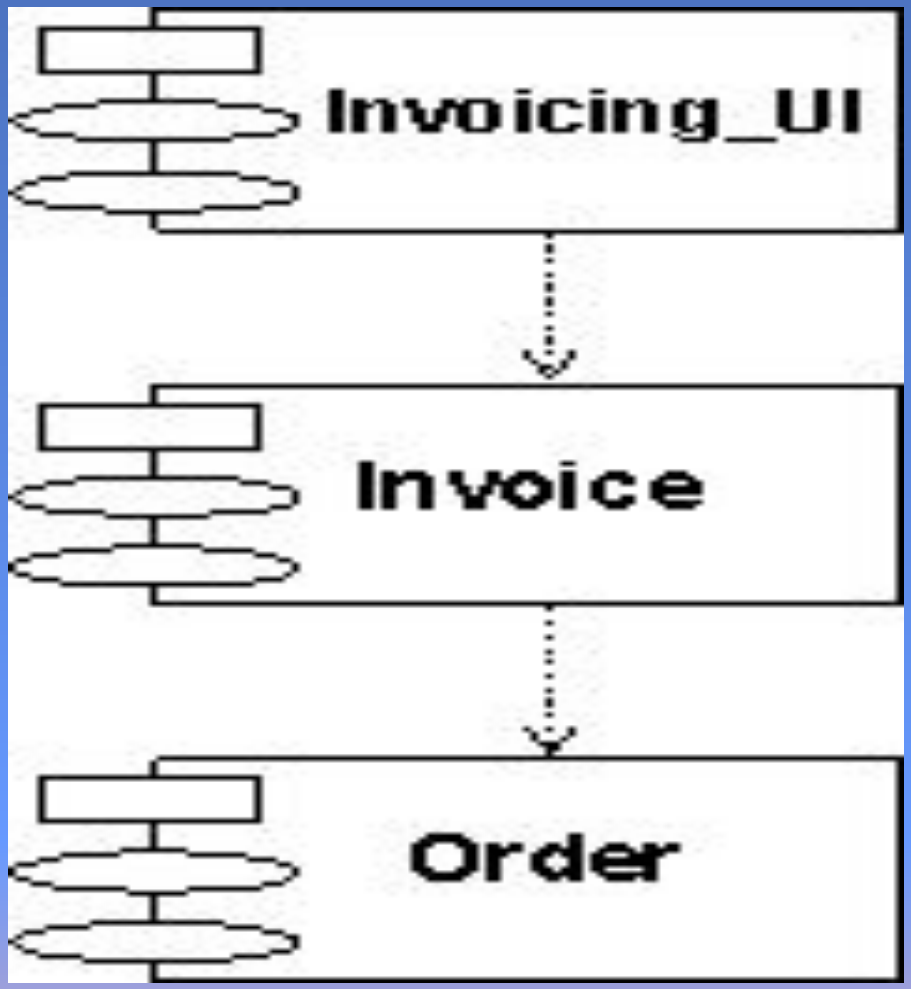
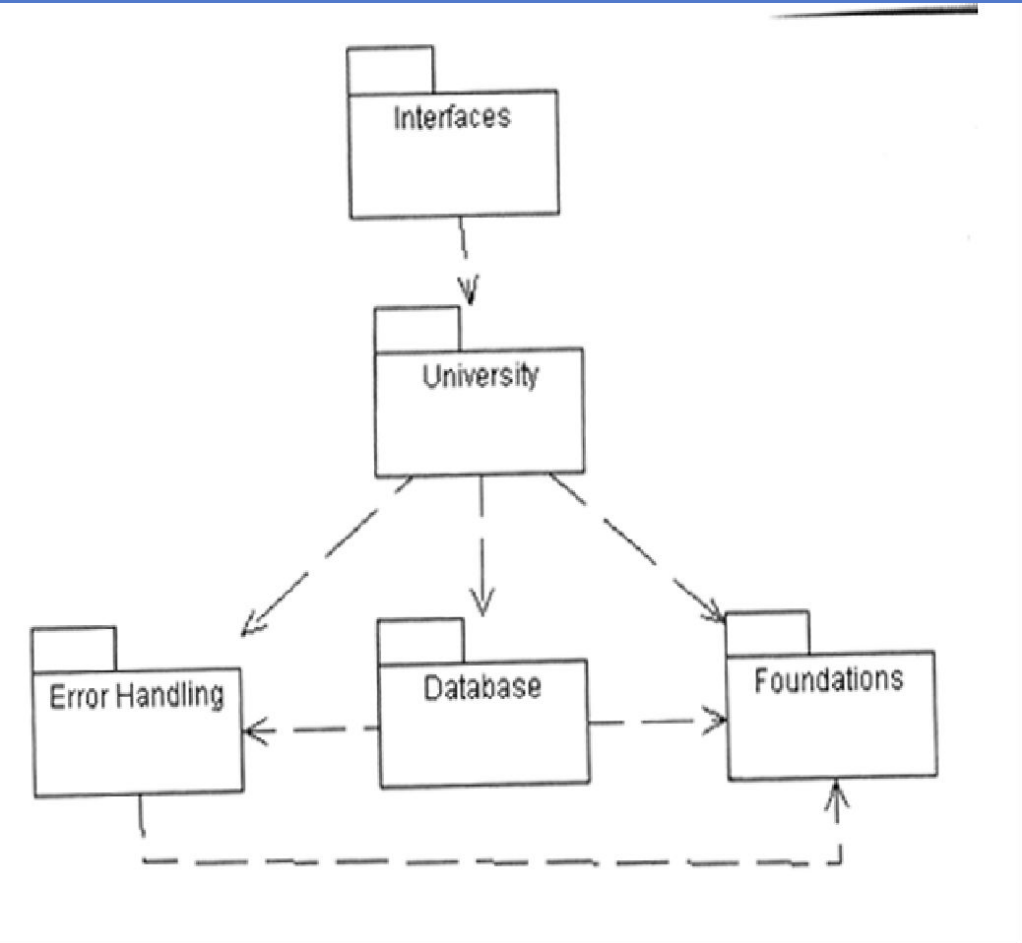


Рис. 6. Пример диаграммы компонент

Пример главной диаграммы с подсистемами компонент



Диаграммы размещения

Диаграммы размещения

Диаграммы размещения используются, для отражения **конфигурации оборудования и программного обеспечения** в реально действующей системе.

Основные элементы диаграммы:

процессоры;

устройства;

соединения.

Процессор (иначе **компьютер**) - часть аппаратуры, **способная выполнять программы**.

Устройство это часть оборудования, на котором **программы не выполняются**.

Для обозначения **компьютеров** или **процессоров**, прочих **устройств** и их **соединений** на диаграммах размещения используются обозначения, представленные на рис.1.

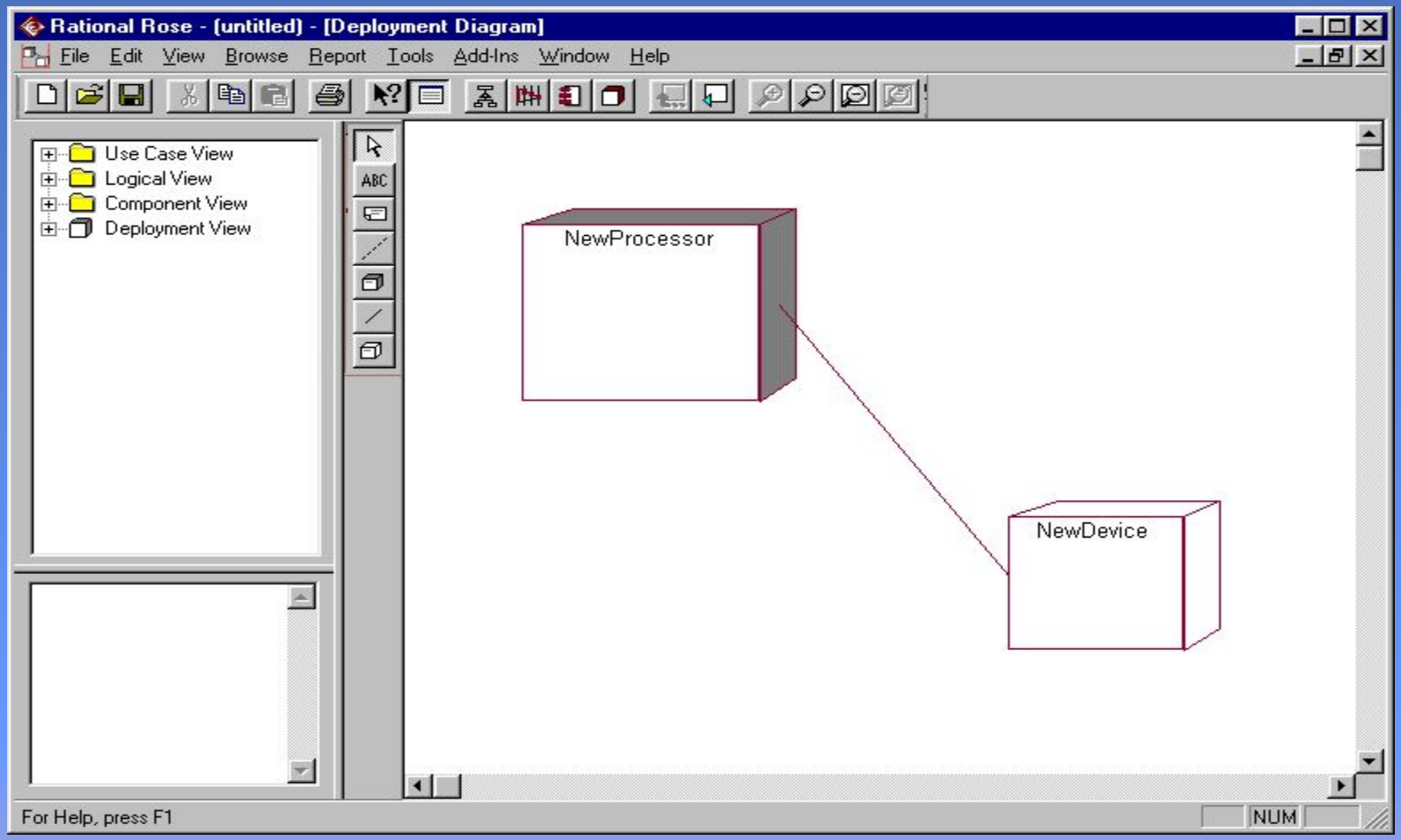


Рис. 1. Пример обозначения процессоров, устройств и связей между ними

На диаграммах каждый компьютер и устройство должны иметь свое **имя**.

Никаких ограничений на имена процессоров и устройств нет, так как они обозначают "**железо**", а не программы.

Можно дополнить значок процессора или компьютера **списком процессов** или **программ**, выполняющихся на нем, например, как представлено на рис. 2.

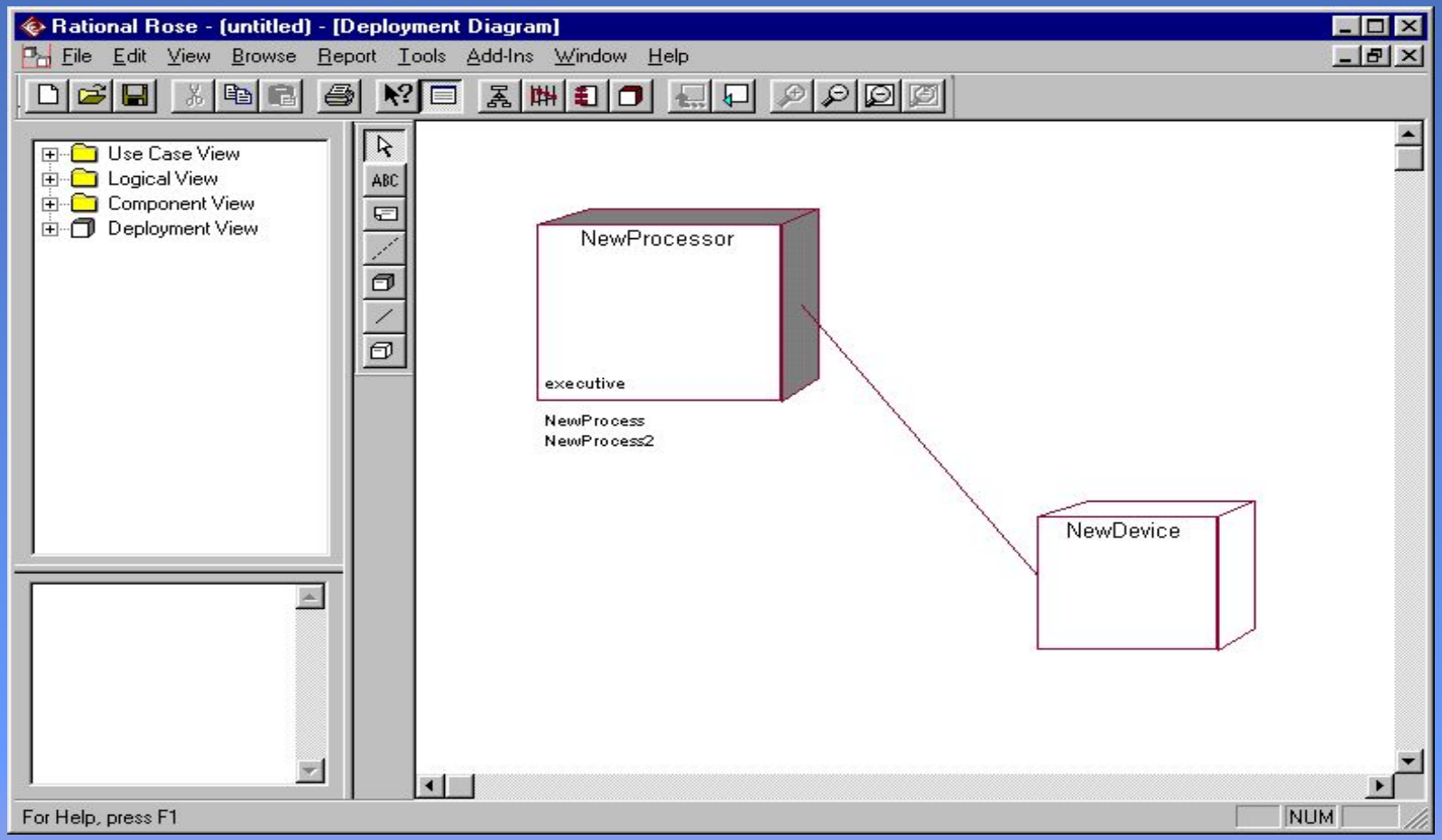


Рис. 2. Пример обозначения процессоров с процессами, устройств и связей между ними

Соединения на диаграмме изображается **линией**.

Соединение представляет непосредственную связь между аппаратурой, например, RS232.

На рис. 3 представлен пример диаграммы размещения.

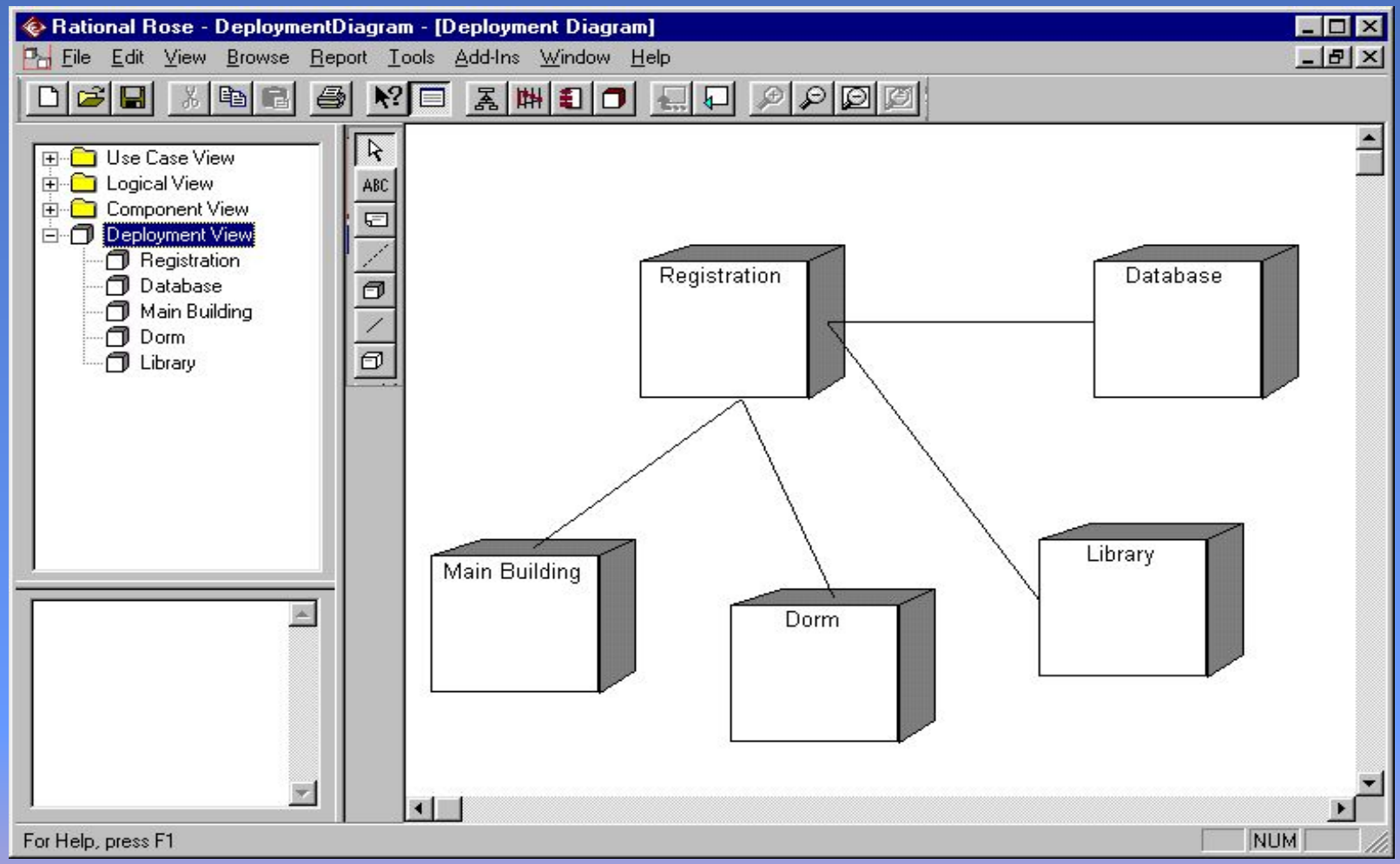


Рис. 3. Пример диаграммы размещения

Для документирования процессов и устройств используются **спецификации**.

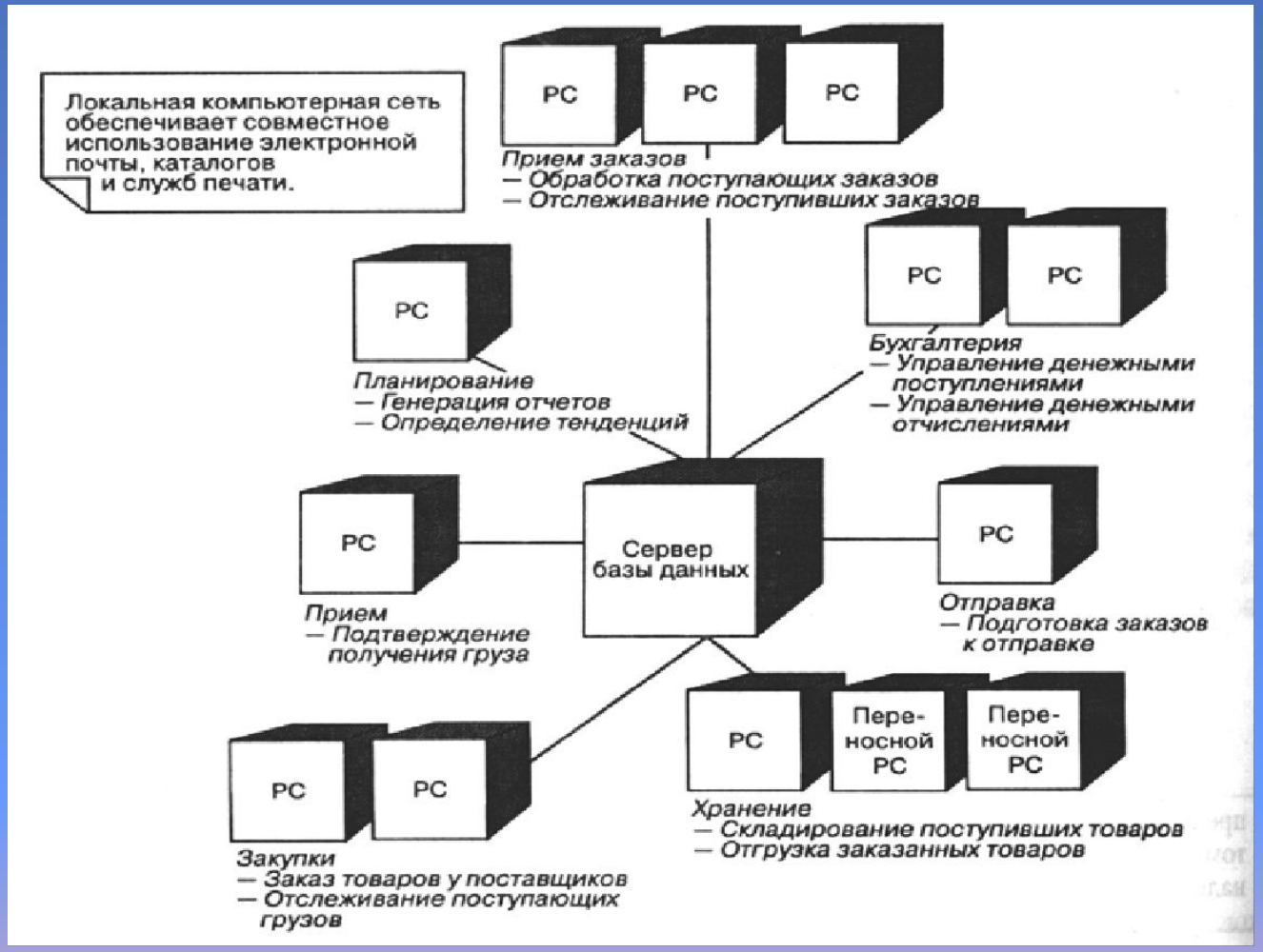
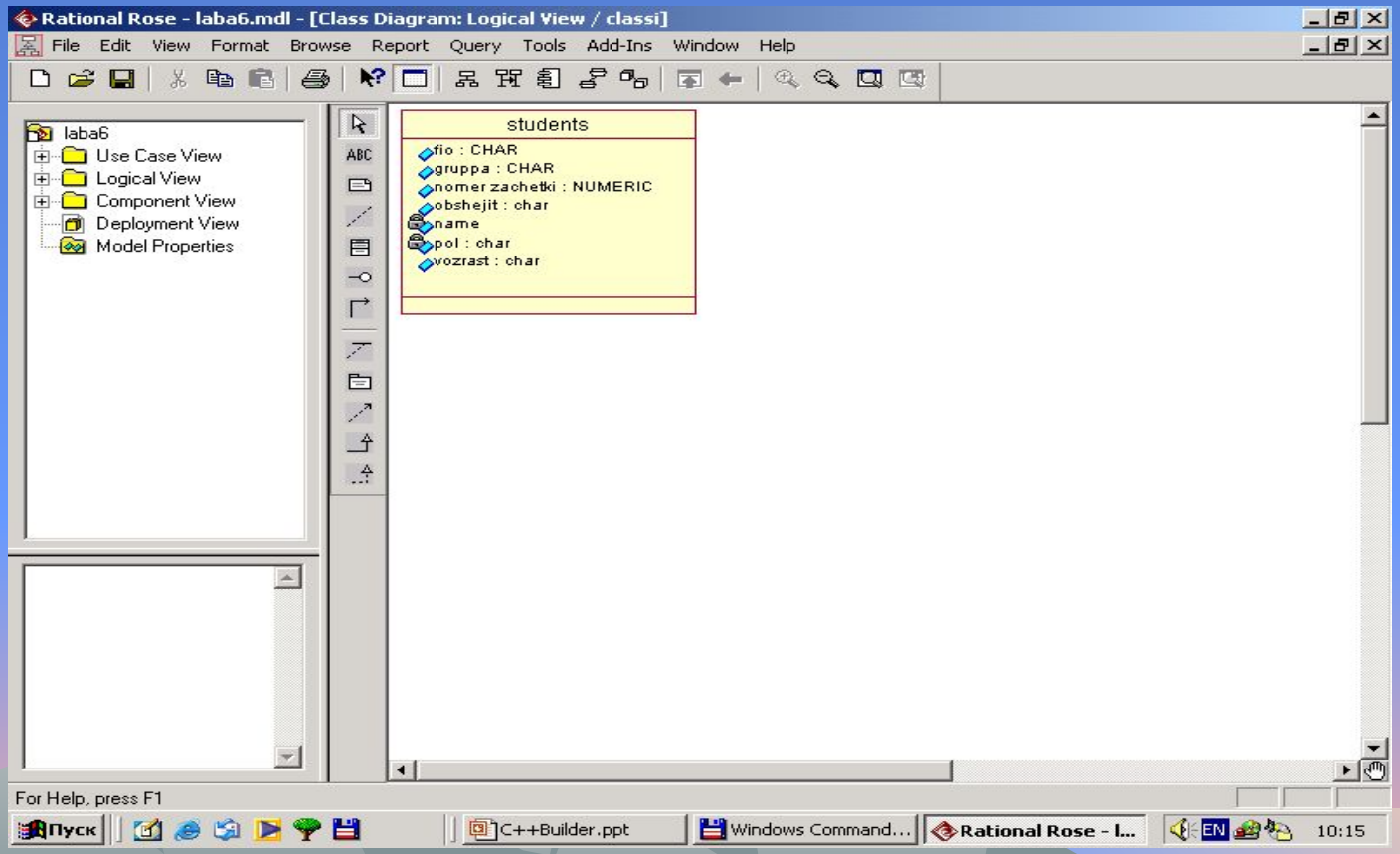


Рис. 4. Пример диаграммы размещения

Разработка программных средств в C++Builder с использованием диаграммы классов

Пример 1.

- Диаграмма классов



Пример 1.

- **Сгенерированный код на языке C++ CASE-средством Rational Rose**
- **Текс содержит описания конструктора и деструктура класса student.**
- //## begin module%1.2%.codegen_version preserve=yes
- // Read the documentation to learn more about C++ code generator
- // versioning.
- //## end module%1.2%.codegen_version

- //## begin module%407A8ED70148.cm preserve=no
- // %X% %Q% %Z% %W%
- //## end module%407A8ED70148.cm

- //## begin module%407A8ED70148.cp preserve=no
- //## end module%407A8ED70148.cp

- //## Module: students%407A8ED70148; Pseudo Package body
- //## Source file: E:\RR2000\Rose 2000\C++\source\students.cpp

Текст программы для примера 1 (students.cpp)

- //## begin module%407A8ED70148.additionalIncludes preserve=no
- //## end module%407A8ED70148.additionalIncludes

- //## begin module%407A8ED70148.includes preserve=yes
- //## end module%407A8ED70148.includes

- // students
- #include "students.h"
- //## begin module%407A8ED70148.additionalDeclarations preserve=yes
- //## end module%407A8ED70148.additionalDeclarations

- // Class students

Текст программы для примера 1 (students.cpp)

```
students::students()
  //## begin students::students%407A8ED70148_const.hasinit preserve=no
  //## end students::students%407A8ED70148_const.hasinit
  //## begin students::students%407A8ED70148_const.initialization preserve=yes
  //## end students::students%407A8ED70148_const.initialization
{
  //## begin students::students%407A8ED70148_const.body preserve=yes
  //## end students::students%407A8ED70148_const.body
}

students::students(const students &right)
  //## begin students::students%407A8ED70148_copy.hasinit preserve=no
  //## end students::students%407A8ED70148_copy.hasinit
  //## begin students::students%407A8ED70148_copy.initialization preserve=yes
  //## end students::students%407A8ED70148_copy.initialization
{
  //## begin students::students%407A8ED70148_copy.body preserve=yes
  //## end students::students%407A8ED70148_copy.body
}
```

Текст программы для примера 1 (students.cpp)

- students::~students()
- {
- //## begin students::~students%407A8ED70148_dest.body preserve=yes
- //## end students::~students%407A8ED70148_dest.body
- }

- students & students::operator=(const students &right)
- {
- //## begin students::operator=%407A8ED70148_assign.body preserve=yes
- //## end students::operator=%407A8ED70148_assign.body
- }

Текст программы для примера 1 (students.cpp)

```
int students::operator==(const students &right) const
{
  //## begin students::operator==%407A8ED70148_eq.body preserve=yes
  //## end students::operator==%407A8ED70148_eq.body
}

int students::operator!=(const students &right) const
{
  //## begin students::operator!=%407A8ED70148_neq.body preserve=yes
  //## end students::operator!=%407A8ED70148_neq.body
}


// Additional Declarations
  //## begin students%407A8ED70148.declarations preserve=yes
  //## end students%407A8ED70148.declarations

//## begin module%407A8ED70148.epilog preserve=yes
//## end module%407A8ED70148.epilog
```

Текст программы для примера 1 (students.h)

- **Содержит описание класса student, его атрибутов и методов**

```
//## begin module%1.2%.codegen_version preserve=yes
//   Read the documentation to learn more about C++ code generator
//   versioning.
//## end module%1.2%.codegen_version

//## begin module%407A8ED70148.cm preserve=no
//  %X% %Q% %Z% %W%
//## end module%407A8ED70148.cm

//## begin module%407A8ED70148.cp preserve=no
//## end module%407A8ED70148.cp

//## Module: students%407A8ED70148; Pseudo Package specification
//## Source file: E:\RR2000\Rose 2000\C++\source\students.h
```

Текст программы для примера 1 (students.h)

- #ifndef students_h
- #define students_h 1

- #define NUMERIC int
- #define CHAR char*
- //## begin module%407A8ED70148.additionalIncludes preserve=no
- //## end module%407A8ED70148.additionalIncludes

- //## begin module%407A8ED70148.includes preserve=yes
- //## end module%407A8ED70148.includes

- //## begin module%407A8ED70148.additionalDeclarations preserve=yes
- //## end module%407A8ED70148.additionalDeclarations

- //## begin students%407A8ED70148.preface preserve=yes
- //## end students%407A8ED70148.preface

Текст программы для примера 1 (students.h)

```
//## Class: students%407A8ED70148
//## Category: <Top Level>
//## Persistence: Transient
//## Cardinality/Multiplicity: n

class students
{
  //## begin students%407A8ED70148.initialDeclarations preserve=yes
  //## end students%407A8ED70148.initialDeclarations

  public:
    //## Constructors (generated)
      students();

      students(const students &right);

```

Текст программы для примера 1 (students.h)

```
//## Destructor (generated)
    ~students();

  //## Assignment Operation (generated)
    students & operator=(const students &right);

  //## Equality Operations (generated)
    int operator==(const students &right) const;

    int operator!=(const students &right) const;

  //## Get and Set Operations for Class Attributes (generated)

    //## Attribute: fio%407A908200DA
    const char* get_fio () ;
    void set_fio (char* value);

    //## Attribute: gruppa%407A909700AB
    const char* get_gruppa () const;
    void set_gruppa (char* value);
```

Текст программы для примера 1 (students.h)

```
//## Attribute: nomer zachetki%407A90A1007D
      const int get_nomer_zachetki () const;
      void set_nomer_zachetki (int value);

      //## Attribute: obshejit%407A90AE00AB
      const char* get_obshejit () const;
      void set_obshejit (char* value);

      //## Attribute: vozrast%40B1E4E20073
      const int get_vozrast () const;
      void set_vozrast (int value);

    // Additional Public Declarations
      //## begin students%407A8ED70148.public preserve=yes
      //## end students%407A8ED70148.public
```

Текст программы для примера 1 (students.h)

```
protected:
    // Additional Protected Declarations
      //## begin students%407A8ED70148.protected preserve=yes
      //## end students%407A8ED70148.protected

  private:
    //## Get and Set Operations for Class Attributes (generated)

      //## Attribute: name%40B1D897002A
      const char* get_name () const;
      void set_name (char* value);
```

Текст программы для примера 1 (students.h)

```
//## Attribute: pol%40B1E4C603A0
      const char get_pol () const;
      void set_pol (char value);

    // Additional Private Declarations
      //## begin students%407A8ED70148.private preserve=yes
      //## end students%407A8ED70148.private

  private: //## implementation
    // Data Members for Class Attributes

      //## begin students::fio%407A908200DA.attr preserve=no  public: CHAR {U}
      char* fio;
      //## end students::fio%407A908200DA.attr
```

Текст программы для примера 1 (students.h)

```
//## begin students::gruppa%407A909700AB.attr preserve=no  public: CHAR {U}
    CHAR gruppa;
    //## end students::gruppa%407A909700AB.attr

    //## begin students::nomer zachetki%407A90A1007D.attr preserve=no  public: NUMERIC {U}
    NUMERIC nomer_zachetki;
    //## end students::nomer zachetki%407A90A1007D.attr

    //## begin students::obshejit%407A90AE00AB.attr preserve=no  public: char {U}
    char* obshejit;
    //## end students::obshejit%407A90AE00AB.attr

    //## begin students::name%40B1D897002A.attr preserve=no  private: void {U}
    char* name;
    //## end students::name%40B1D897002A.attr

    //## begin students::pol%40B1E4C603A0.attr preserve=no  private: char {U}
    char pol;
    //## end students::pol%40B1E4C603A0.attr
```

Текст программы для примера 1 (students.h)

```
//## begin students::vozrast%40B1E4E20073.attr preserve=no  public: NUMERIC {U}
        int vozrast;
        //## end students::vozrast%40B1E4E20073.attr

      // Additional Implementation Declarations
        //## begin students%407A8ED70148.implementation preserve=yes
        //## end students%407A8ED70148.implementation

};

//## begin students%407A8ED70148.postscript preserve=yes
//## end students%407A8ED70148.postscript

// Class students
```

Текст программы для примера 1 (students.h)

```
//## Get and Set Operations for Class Attributes (inline)

inline  const CHAR students::get_fio ()
{
  //## begin students::get_fio%407A908200DA.get preserve=no
  return fio;
  //## end students::get_fio%407A908200DA.get
}

inline void students::set_fio (CHAR value)
{
  //## begin students::set_fio%407A908200DA.set preserve=no
  fio = value;
  //## end students::set_fio%407A908200DA.set
}

inline const CHAR students::get_gruppa () const
{
  //## begin students::get_gruppa%407A909700AB.get preserve=no
  return gruppa;
  //## end students::get_gruppa%407A909700AB.get
}
```

Текст программы для примера 1 (students.h)

```
inline void students::set_gruppa (CHAR value)
{
  //## begin students::set_gruppa%407A909700AB.set preserve=no
  gruppa = value;
  //## end students::set_gruppa%407A909700AB.set
}

inline const NUMERIC students::get_nomer_zachetki () const
{
  //## begin students::get_nomer_zachetki%407A90A1007D.get preserve=no
  return nomer_zachetki;
  //## end students::get_nomer_zachetki%407A90A1007D.get
}

inline void students::set_nomer_zachetki (NUMERIC value)
{
  //## begin students::set_nomer_zachetki%407A90A1007D.set preserve=no
  nomer_zachetki = value;
  //## end students::set_nomer_zachetki%407A90A1007D.set
}
```

Текст программы для примера 1 (students.h)

```
inline const char* students::get_obshejit () const
{
  //## begin students::get_obshejit%407A90AE00AB.get preserve=no
  return obshejit;
  //## end students::get_obshejit%407A90AE00AB.get
}

inline void students::set_obshejit (char* value)
{
  //## begin students::set_obshejit%407A90AE00AB.set preserve=no
  obshejit = value;
  //## end students::set_obshejit%407A90AE00AB.set
}

inline const char* students::get_name () const
{
  //## begin students::get_name%40B1D897002A.get preserve=no
  return name;
  //## end students::get_name%40B1D897002A.get
}

inline void students::set_name (char* value)
{
  //## begin students::set_name%40B1D897002A.set preserve=no
  name = value;
  //## end students::set_name%40B1D897002A.set
}
```

Текст программы для примера 1 (students.h)

```
inline const char students::get_pol () const
{
  //## begin students::get_pol%40B1E4C603A0.get preserve=no
  return pol;
  //## end students::get_pol%40B1E4C603A0.get
}

inline void students::set_pol (char value)
{
  //## begin students::set_pol%40B1E4C603A0.set preserve=no
  pol = value;
  //## end students::set_pol%40B1E4C603A0.set
}

inline const int students::get_vozrast () const
{
  //## begin students::get_vozrast%40B1E4E20073.get preserve=no
  return vozrast;
  //## end students::get_vozrast%40B1E4E20073.get
}
```

Текст программы для примера 1 (students.h)

```
inline void students::set_vozrast (int value)
{
  //## begin students::set_vozrast%40B1E4E20073.set preserve=no
  vozrast = value;
  //## end students::set_vozrast%40B1E4E20073.set
}

//## begin module%407A8ED70148.epilog preserve=yes
//## end module%407A8ED70148.epilog


#endif
```

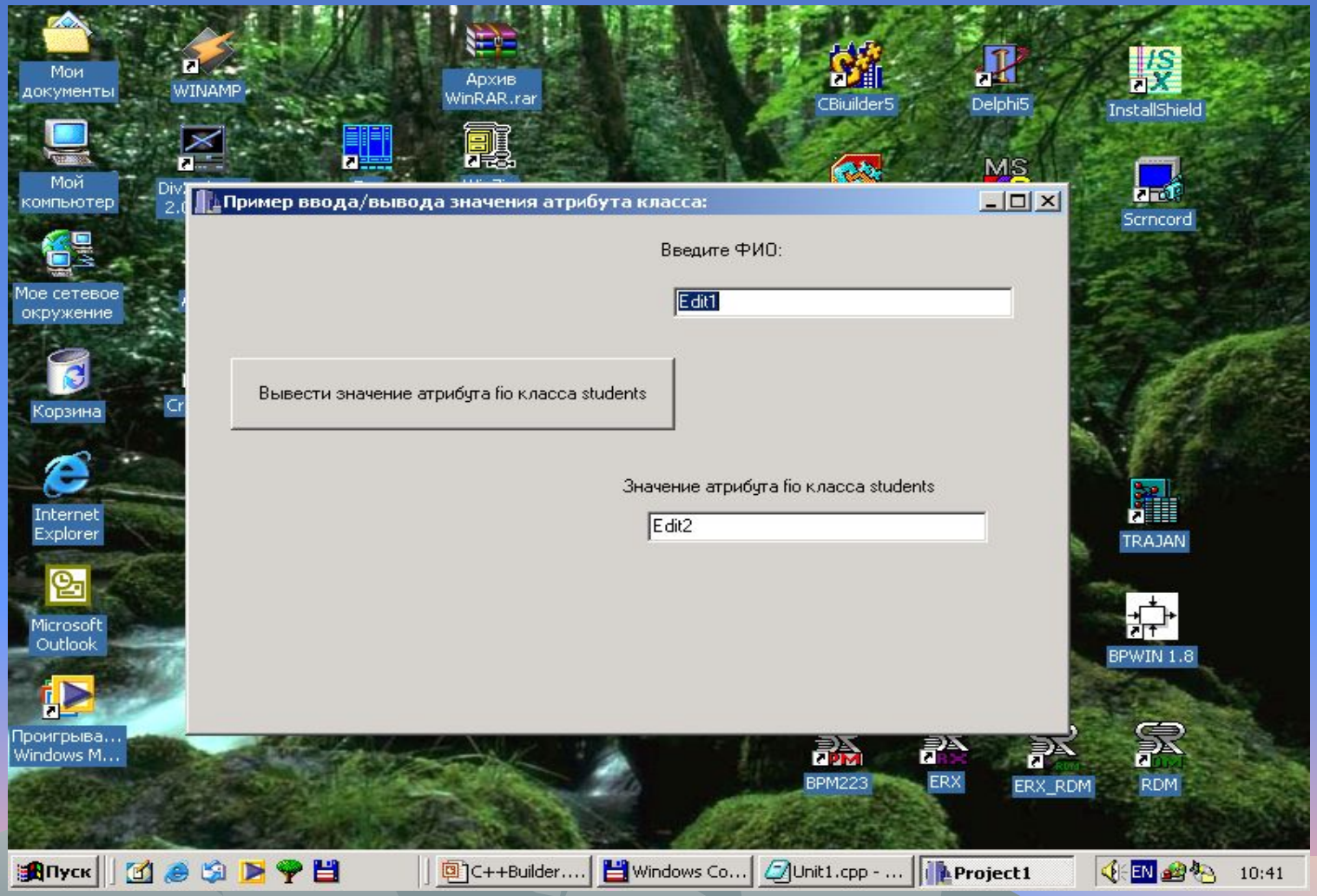
Текст программы Project1.cpp

- **Этот файл содержит описание главной функции. Содержание этого файла генерируется программной средой C++Builder.**

```
//---------------------------------------------------------------------------

#include <vcl.h>
#pragma hdrstop
USERES("Project1.res");
USEFORM("Unit1.cpp", Form1);
USEUNIT("students.cpp");
USE("students.h", File);
//---------------------------------------------------------------------------
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
        try
        {
                Application->Initialize();
                Application->CreateForm(__classid(TForm1), &Form1);
                Application->Run();
        }
        catch (Exception &exception)
        {
                Application->ShowException(&exception);
        }
        return 0;
}
//---------------------------------------------------------------------------
```

Форма пользовательского интерфейса



Текст функций, выполняемых при нажатии на кнопку

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
#include"students.h"
#include"students.cpp"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------------------------

void __fastcall TForm1::Button1Click(TObject *Sender)
{
students *pstudents;
pstudents=new students;
pstudents->set_fio( Edit1->Text.c_str());
Edit2->Text=pstudents->get_fio();
MessageBox(NULL,pstudents->get_fio(),"Значение атрибута fio класса students",MB_OK);
}
//---------------------------------------------------------------------------
```

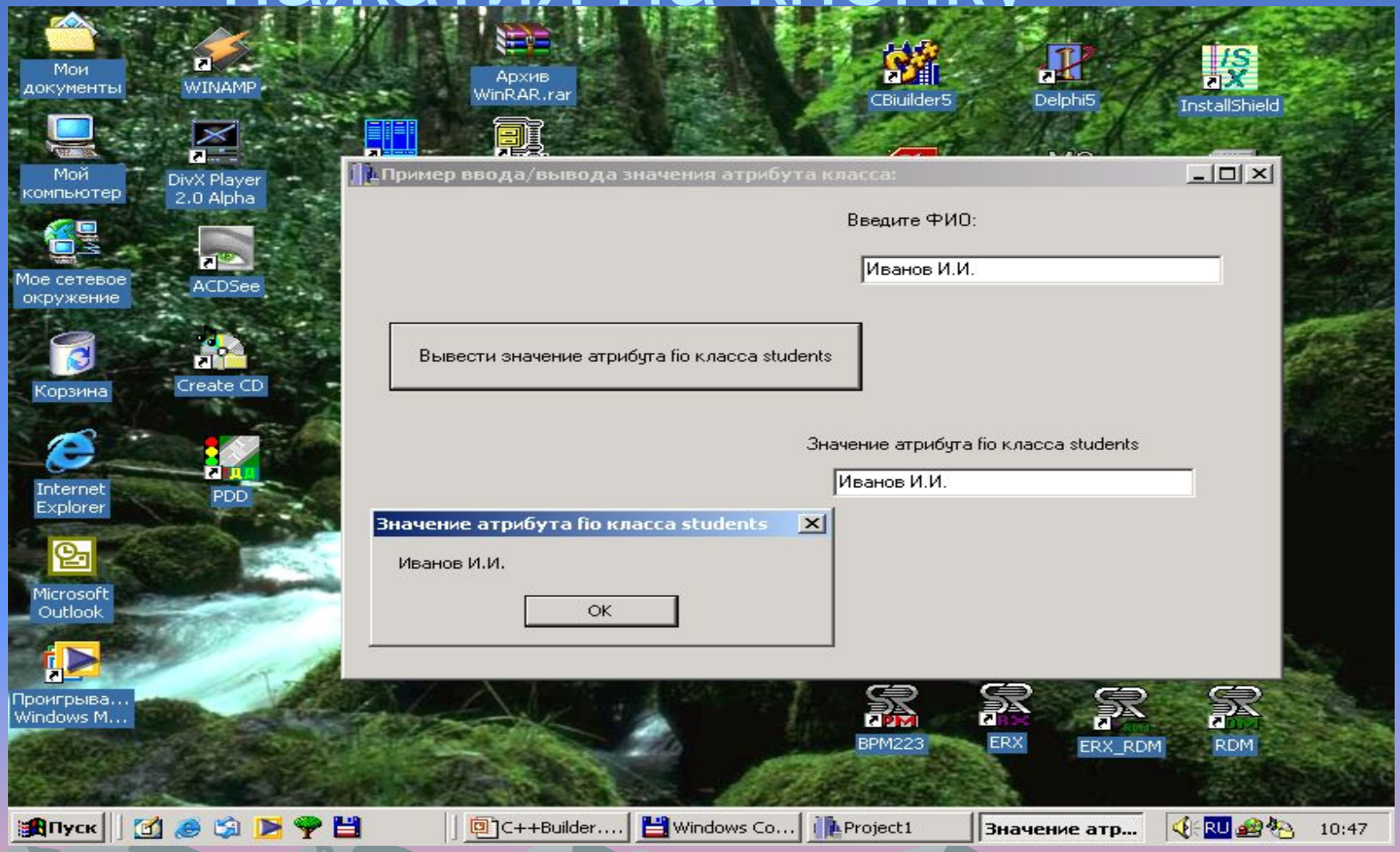
Файл, содержащий описание формы

- Автоматически генерируется в C++Builder.

```
//---------------------------------------------------------------------------

#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//---------------------------------------------------------------------------
class TForm1 : public TForm
{
__published:        // IDE-managed Components
        TEdit *Edit1;
        TEdit *Edit2;
        TButton *Button1;
        TLabel *Label1;
        TLabel *Label2;
        void __fastcall Button1Click(TObject *Sender);
private:    // User declarations
public:                 // User declarations
        __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
```

Интерфейс программного средства после ввода Ф.И.О. и нажатия на кнопку



ЯЗЫК SQL

ЯЗЫК SQL

- Как и большинство современных реляционных языков, SQL основан на исчислении кортежей. В результате, каждый запрос, сформулированный с помощью исчисления кортежей (или иначе говоря, реляционной алгебры), может быть также сформулирован с помощью SQL. Однако, он имеет способности, лежащие за пределами реляционной алгебры или исчисления. Вот список некоторых дополнительных свойств, предоставленных SQL, которые не являются частью реляционной алгебры или исчисления:
- Команды вставки, удаления или изменения данных.
- Арифметические возможности: в SQL возможно вызвать арифметические операции, так же как и сравнения, например A < B + 3. Заметим, что + или других арифметических операторов нет ни в реляционной алгебре ни в реляционном исчислении.
- Команды присвоения и печати: возможно напечатать отношение, созданное запросом и присвоить вычисленному отношению имя отношения.
- Итоговые функции: такие операции как *average*, *sum*, *max*, и т.д. могут применяться к столбцам отношения для получения единичной величины.

Выборка

- Наиболее часто используемая команда SQL - это оператор SELECT, используемый для получения данных. Синтаксис:
- SELECT [ALL|DISTINCT] { * | *expr_1* [AS *c_alias_1*] [, ... [, *expr_k* [AS *c_alias_k*]]]} FROM *table_name_1* [*t_alias_1*] [, ... [, *table_name_n* [*t_alias_n*]]] [WHERE *condition*] [GROUP BY *name_of_attr_i* [,... [, *name_of_attr_j*]] [HAVING *condition*]] [{UNION [ALL] | INTERSECT | EXCEPT} SELECT ...] [ORDER BY *name_of_attr_i* [ASC|DESC] [, ... [, *name_of_attr_j* [ASC|DESC]]]];

Простые выборки

- **Пример 2-4. Простой ограничивающий запрос**
- Получить все кортежи из таблицы PART, где атрибут PRICE больше 10:
- SELECT * FROM PART WHERE PRICE > 10;
  Получаемая таблица:
- PNO | PNAME | PRICE
- 3 | Bolt | 15
- 4 | Cam | 25
- Использовав "*" в операторе SELECT, получаем все атрибуты из таблицы.

Простые выборки

- Если мы хотим получить только атрибуты PNAME и PRICE из таблицы PART, то используем следующее выражение:
- SELECT PNAME, PRICE FROM PART WHERE PRICE > 10; В этом случае получим:
- PNAME | PRICE
- Bolt | 15
- Cam | 25
- Заметим, что SQL SELECT соответствует "проекции" в реляционной алгебре, а не "выборке"

Простые выборки

- Ограничения в операторе WHERE могут также быть логически соединены с помощью ключевых слов OR, AND, и NOT:
- SELECT PNAME, PRICE FROM PART WHERE PNAME = 'Bolt' AND (PRICE = 0 OR PRICE < 15); приведёт к результату:
- PNAME | PRICE
- Bolt | 15

Простые выборки

- Арифметические операции могут использоваться в списке объектов и операторе WHERE. Например, если нам надо знать сколько будут стоить две штуки одной детали, то используем следующий запрос:
- SELECT PNAME, PRICE * 2 AS DOUBLE FROM PART WHERE PRICE * 2 < 50; и получим: PNAME | DOUBLE

  Screw | 20

  Nut | 16

  Bolt | 30

Заметим, что слово DOUBLE после ключевого слова AS - это новый заголовок второго столбца. Эта техника может быть использована для любого элемента списка объектов, для того чтобы задать новый заголовок столбцу результата. Этот новый заголовок часто называют псевдонимом. Псевдонимы не могут просто использоваться в запросе.

Соединения

- Следующий пример показывает, как осуществлять *соединения* в SQL.
- Для объединения трёх таблиц SUPPLIER, PART и SELLS по их общим атрибутам, мы формулируем следующее выражение:
- SELECT S.SNAME, P.PNAME FROM SUPPLIER S, PART P, SELLS SE WHERE S.SNO = SE.SNO AND P.PNO = SE.PNO;

Соединения

и получаем следующую таблицу в качестве результата:

SNAME	PNAME
Smith	Screw
Smith	Nut
Jones	Cam
Adams	Screw
Adams	Bolt
Blake	Nut
Blake	Bolt
Blake	Cam

Соединения

В операторе FROM мы вводим псевдоним имени для каждого отношения, так как в отношениях есть общие названия атрибутов (SNO и PNO). Теперь мы можем различить общие имена атрибутов, просто предварив имя атрибута псевдонимом с точкой. Во-первых, определяется декартово произведение SUPPLIER × PART × SELLS . Потом выбираются только те кортежи, которые удовлетворяют условиям, заданным в операторе WHERE (т.е. где общие именованные атрибуты равны). Наконец, убираются все колонки кроме S.SNAME и P.PNAME.

Итоговые операторы

- SQL снабжён итоговыми операторами (например AVG, COUNT, SUM, MIN, MAX), которые принимают название атрибута в качестве аргумента. Значение итогового оператора высчитывается из всех значений заданного атрибута(столбца) всей таблицы. Если в запросе указана группа, то вычисления выполняются только над значениями группы (смотри следующий раздел).
- **Пример. Итоги**
- Если мы хотим узнать среднюю стоимость всех деталей в таблице PART, то используем следующий запрос:
- SELECT AVG(PRICE) AS AVG_PRICE FROM PART;
- Результат:
- AVG_PRICE
- 14.5

Итоговые операторы

- Если мы хотим узнать количество деталей, хранящихся в таблице PART, то используем выражение:
- SELECT COUNT(PNO) FROM PART; и получим: COUNT
- 4

Итоги по группам

- SQL позволяет разбить кортежи таблицы на группы. После этого итоговые операторы, описанные выше, могут применяться к группам (т.е. значение итогового оператора вычисляется не из всех значений указанного столбца, а над всеми значениями группы. Таким образом, итоговый оператор вычисляет индивидуально для каждой группы.)
- Разбиение кортежей на группы выполняется с помощью ключевых слов **GROUP BY** и следующим за ними списком атрибутов, которые определяют группы. Если мы имеем **GROUP BY A1, ⃛, Ak** мы разделяем отношение на группы так, что два кортежа будут в одной группе, если у них соответствуют все атрибуты A1, ⃛, Ak.

Итоги по группам

- Если мы хотим узнать сколько деталей продаёт каждый поставщик, то мы так сформулируем запрос:
- SELECT S.SNO, S.SNAME, COUNT(SE.PNO) FROM SUPPLIER S, SELLS SE WHERE S.SNO = SE.SNO GROUP BY S.SNO, S.SNAME; и получим:
- SNO | SNAME | COUNT
- 1 | Smith | 2
- 2 | Jones | 1
- 3 | Adams | 2
- 4 | Blake | 3

Итоги по группам

- Теперь давайте посмотрим что здесь происходит. Во-первых, соединяются таблицы SUPPLIER и SELLS:
- S.SNO | S.SNAME | SE.PNO
- 1 | Smith | 1
- 1 | Smith | 2
- 2 | Jones | 4
- 3 | Adams | 1
- 3 | Adams | 3
- 4 | Blake | 2
- 4 | Blake | 3
- 4 | Blake | 4

Having

Оператор HAVING выполняет ту же работу что и оператор WHERE, но принимает к рассмотрению только те группы, которые удовлетворяют определению оператора HAVING. Выражения в операторе HAVING должны вызывать итоговые функции. Каждое выражение, использующее только простые атрибуты, принадлежат оператору WHERE. С другой стороны каждое выражение вызывающее итоговую функцию должно помещаться в оператор HAVING.

Having

- **Пример. Having**
- Если нас интересуют поставщики, продающие более одной детали, используем запрос:
- SELECT S.SNO, S.SNAME, COUNT(SE.PNO) FROM SUPPLIER S, SELLS SE WHERE S.SNO = SE.SNO GROUP BY S.SNO, S.SNAME HAVING COUNT(SE.PNO) > 1; и получим:
- SNO | SNAME | COUNT
- 1 | Smith | 2
- 3 | Adams | 2
- 4 | Blake | 3

Подзапросы

В операторах WHERE и HAVING используются подзапросы (вложенные выборки), которые разрешены в любом месте, где ожидается значение. В этом случае значение должно быть получено предварительно подсчитав подзапрос. Использование подзапросов увеличивает выражающую мощность SQL.

Подзапросы

- **Пример. Вложенная выборка**
- Если мы хотим узнать все детали, имеющие цену больше чем деталь 'Screw', то используем запрос:
- SELECT * FROM PART WHERE PRICE > (SELECT PRICE FROM PART WHERE PNAME='Screw');
- Результат:
- PNO | PNAME | PRICE
- 3 | Bolt | 15
- 4 | Cam | 25

Подзапросы

Если мы посмотрим на запрос выше, то увидим ключевое слово SELECT два раза. Первый начинает запрос - мы будем называть его внешним запросом SELECT - и второй в операторе WHERE, который начинает вложенный запрос - мы будем называть его внутренним запросом SELECT. Для каждого кортежа внешнего SELECT внутренний SELECT необходимо вычислить. После каждого вычисления мы узнаём цену кортежа с названием 'Screw' и мы можем проверить выше ли цена из текущего кортежа.

Подзапросы

- Если мы хотим узнать поставщиков, которые ничего не продают (например, чтобы удалить этих поставщиков из базы данных) используем:
- SELECT * FROM SUPPLIER S WHERE NOT EXISTS (SELECT * FROM SELLS SE WHERE SE.SNO = S.SNO);
- В нашем примере результат будет пустым, потому что каждый поставщик продаёт хотя бы одну деталь. Заметим, что мы использовали S.SNO из внешнего SELECT внутри оператора WHERE в внутреннем SELECT. Как описывалось выше подзапрос вычисляется для каждого кортежа из внешнего запроса т.е. значение для S.SNO всегда берётся из текущего кортежа внешнего SELECT.

Объединение, пересечение, исключение

- Эти операции вычисляют объединение, пересечение и теоретико-множественное вычитание кортежей из двух подзапросов.
- **Пример. Объединение, пересечение, исключение**
- Следующий запрос - пример для UNION(объединение):
- SELECT S.SNO, S.SNAME, S.CITY FROM SUPPLIER S WHERE S.SNAME = 'Jones' UNION SELECT S.SNO, S.SNAME, S.CITY FROM SUPPLIER S WHERE S.SNAME = 'Adams';
- даёт результат:
- SNO | SNAME | CITY
- 2 | Jones | Paris
- 3 | Adams | Vienna

Объединение, пересечение, исключение

- Вот пример для INTERSECT(пересечение):
- SELECT S.SNO, S.SNAME, S.CITY FROM SUPPLIER S WHERE S.SNO > 1 INTERSECT SELECT S.SNO, S.SNAME, S.CITY FROM SUPPLIER S WHERE S.SNO > 2;
- даёт результат:
- SNO | SNAME | CITY
- 2 | Jones | Paris
- Возвращаются только те кортежи, которые есть в обоих частях запроса и имеют $SNO=2$.

Объединение, пересечение, исключение

- Наконец, пример для EXCEPT(исключение):
- SELECT S.SNO, S.SNAME, S.CITY FROM SUPPLIER S WHERE S.SNO > 1 EXCEPT SELECT S.SNO, S.SNAME, S.CITY FROM SUPPLIER S WHERE S.SNO > 3;
- даёт результат:
- SNO | SNAME | CITY
- 2 | Jones | Paris
- 3 | Adams | Vienna

Создание таблицы

- Самая основная команда определения данных - это та, которая создаёт новое отношение (новую таблицу). Синтаксис команды CREATE TABLE:
- CREATE TABLE *table_name* (*name_of_attr_1 type_of_attr_1* [, *name_of_attr_2 type_of_attr_2* [, ...]]);

Создание таблицы

- **Пример. Создание таблицы**
- Для создания таблиц используются следующие SQL выражения:
- CREATE TABLE SUPPLIER (SNO INTEGER, SNAME VARCHAR(20), CITY VARCHAR(20));
- CREATE TABLE PART (PNO INTEGER, PNAME VARCHAR(20), PRICE DECIMAL(4 , 2));
- CREATE TABLE SELLS (SNO INTEGER, PNO INTEGER);

Типы данных SQL

- Список некоторых типов данных, которые поддерживает SQL:
- INTEGER: знаковое полнословное двоичное целое (31 бит для представления данных).
- SMALLINT: знаковое полсловное двоичное целое (15 бит для представления данных).
- DECIMAL (*p*[,*q*]): знаковое упакованное десятичное число с *p* знаками представления данных, с возможным *q* знаками справа от десятичной точки. (15 ≥ *p* ≥ *q*q ≥ 0). Если *q* опущено, то предполагается что оно равно 0.
- FLOAT: знаковое двусловное число с плавающей точкой.
- CHAR(*n*): символьная строка с постоянной длиной *n*.
- VARCHAR(*n*): символьная строка с изменяемой длиной, максимальная длина *n*.

Создание индекса

Индексы используются для ускорения доступа к отношению. Если отношение R проиндексировано по атрибуту A, то мы можем получить все кортежи *t* имеющие *t*(A) = *a* за время приблизительно пропорциональное числу таких кортежей *t*, в отличие от времени, пропорциональному размеру R.

Создание индекса

- Для создания индекса в SQL используется команда CREATE INDEX. Синтаксис:
- CREATE INDEX *index_name* ON *table_name* (*name_of_attribute*);
- **Пример. Создание индекса**
- Для создания индекса с именем I по атрибуту SNAME отношения SUPPLIER используем следующее выражение:
- CREATE INDEX I ON SUPPLIER (SNAME);
- Созданный индекс обслуживается автоматически, т.е. при вставке ново кортежа в отношение SUPPLIER, индекс I будет перестроен. Заметим, что пользователь может ощутить изменения в при существовании индекса только по увеличению скорости.

Создание представлений

- Представление можно рассматривать как *виртуальную таблицу*, т.е. таблицу, которая в базе данных не существует *физически*, но для пользователя она как-бы там есть. По сравнению, если мы говорим о *базовой таблице*, то мы имеем в виду таблицу, физически хранящую каждую строку где-то на физическом носителе.
- Представления не имеют своих собственных, физически самостоятельных, различимых хранящихся данных. Вместо этого, система хранит определение представления (т.е. правила о доступе к физически хранящимся базовым таблицам в порядке претворения их в представление) где-то в системных каталогах

Создание представлений

- Для определения представлений в SQL используется команда **CREATE VIEW**. Синтаксис:
- CREATE VIEW *view_name* AS *select_stmt* где *select_stmt*
- Заметим, что *select_stmt* не выполняется при создании представления. Оно только сохраняется в *системных каталогах* и выполняется всякий раз когда делается запрос представления.

Создание представлений

- Пусть дано следующее определение представления:
- CREATE VIEW London_Suppliers AS SELECT S.SNAME, P.PNAME FROM SUPPLIER S, PART P, SELLS SE WHERE S.SNO = SE.SNO AND P.PNO = SE.PNO AND S.CITY = 'London';

Создание представлений

- Теперь мы можем использовать это *виртуальное отношение* London_Suppliers как если бы оно было ещё одной базовой таблицей:
- SELECT * FROM London_Suppliers WHERE P.PNAME = 'Screw';
- которое возвращает следующую таблицу: SNAME | PNAME
- Smith | Screw

Создание представлений

Для вычисления этого результата система базы данных в начале выполняет *скрытый* доступ к базовым таблицам SUPPLIER, SELLS и PART. Это делается с помощью выполнения заданных запросов в определении представления к этим базовым таблицам. После, это дополнительное определедение (заданное в запросе к представлению) можно использовать для получения результирующей таблицы.

Drop Table, Drop Index, Drop View

- Для уничтожения таблицы (включая все кортежи, хранящиеся в этой таблице) используется команда DROP TABLE:
- DROP TABLE *table_name*;
- Для уничтожения таблицы SUPPLIER используется следующее выражение:
- DROP TABLE SUPPLIER;
- Команда DROP INDEX используется для уничтожения индекса:
- DROP INDEX *index_name*;
- Наконец, для уничтожения заданного представления используется команда DROP VIEW:
- DROP VIEW *view_name*;

Манипулирование данными

- **Insert Into**
- После создания таблицы её можно заполнять кортежами с помощью команды **INSERT INTO**. Синтаксис:
- INSERT INTO *table_name* (*name_of_attr_1* [, *name_of_attr_2* [,...]]) VALUES (*val_attr_1* [, *val_attr_2* [, ...]]);

Манипулирование данными

- Чтобы вставить первый кортеж в отношение SUPPLIER используется следующее выражение:
- INSERT INTO SUPPLIER (SNO, SNAME, CITY) VALUES (1, 'Smith', 'London');
- Чтобы вставить первый кортеж в отношение SELLS используется:
- INSERT INTO SELLS (SNO, PNO) VALUES (1, 1);

Обновление

- Для изменения одного или более значений атрибутов кортежей в отношении используется команда UPDATE. Синтаксис:
- UPDATE *table_name* SET *name_of_attr_1* = *value_1* [, ... [, *name_of_attr_k* = *value_k*]] WHERE *condition*;
- Чтобы изменить значение атрибута PRICE детали 'Screw' в отношении PART используется:
- UPDATE PART SET PRICE = 15 WHERE PNAME = 'Screw';
- Новое значение атрибута PRICE кортежа, чьё имя равно 'Screw' теперь стало 15.

Удаление

- Для удаления кортежа из отдельной таблицы используется команда DELETE FROM. Синтаксис:
- DELETE FROM *table_name* WHERE *condition*;
- Чтобы удалить поставщика называемого 'Smith', из таблицы SUPPLIER используем следующее выражение:
- DELETE FROM SUPPLIER WHERE SNAME = 'Smith';

Системные каталоги

В каждой системе базы данных SQL определены *системные каталоги*, которые используются для хранения записей о таблицах, представлениях, индексах и т.д. К системным каталогам также можно строить запросы, как если бы они были нормальными отношениями. Например, один каталог используется для определения представлений. В этом каталоге хранятся запросы об определении представлений. Всякий раз когда делается запрос к представлению, система сначала берёт *запрос определения представления* из этого каталога и материализует представление перед тем, как обработать запрос пользователя.

Встраивание SQL

- В этом разделе мы опишем в общих чертах как SQL может быть встроен в конечный язык (например в C). Есть две главных причины, по которым мы хотим пользоваться SQL из конечного языка:
- Существуют запросы, которые нельзя сформулировать на чистом SQL(т.е. рекурсивные запросы). Чтобы выполнить такие запросы, нам необходим конечный язык, обладающий большей мощностью выразительности, чем SQL.
- Просто нам необходим доступ к базе данных из другого приложения, которое написано на конечном языке (например, система бронирования билетов с графическим интерфейсом пользователя написана на C и информация об оставшихся билетах хранится в базе данных, которую можно получить с помощью встроенного SQL).

Встраивание SQL

- Программа, использующая встроенный SQL в конечном языке, состоит из выражений конечного языка и выражений *встроенного SQL* (ESQL). Каждое выражение ESQL начинается с ключевых слов **EXEC SQL**. Выражения ESQL преобразуются в выражения на конечном языке с помощью *прекомпилятора* (который обычно вставляет вызовы библиотечных процедур, которые выполняют различные команды SQL).

Если мы посмотрим на все примеры из *Выборка* мы поймём, что результатом запроса очень часто являются множество кортежей. Большинство конечных языков не предназначено для работы с множествами, поэтому нам нужен механизм доступа к каждому отдельному кортежу из множества кортежей, возвращаемого выражением SELECT. Этот механизм можно предоставить, определив *курсор*. После этого, мы можем использовать команду FETCH для получения кортежа и установления курсора на следующий кортеж.

РАЗРАБОТКА ПРОТОТИПОВ ИНФОРМАЦИОННЫХ СИСТЕМ, ОСНОВАННЫХ НА СУБД, В CASE-СРЕДСТВЕ RATIONAL ROSE ENTERPRISE И СРЕДЕ ПРОГРАММИРОВАНИЯ C++BUILDER

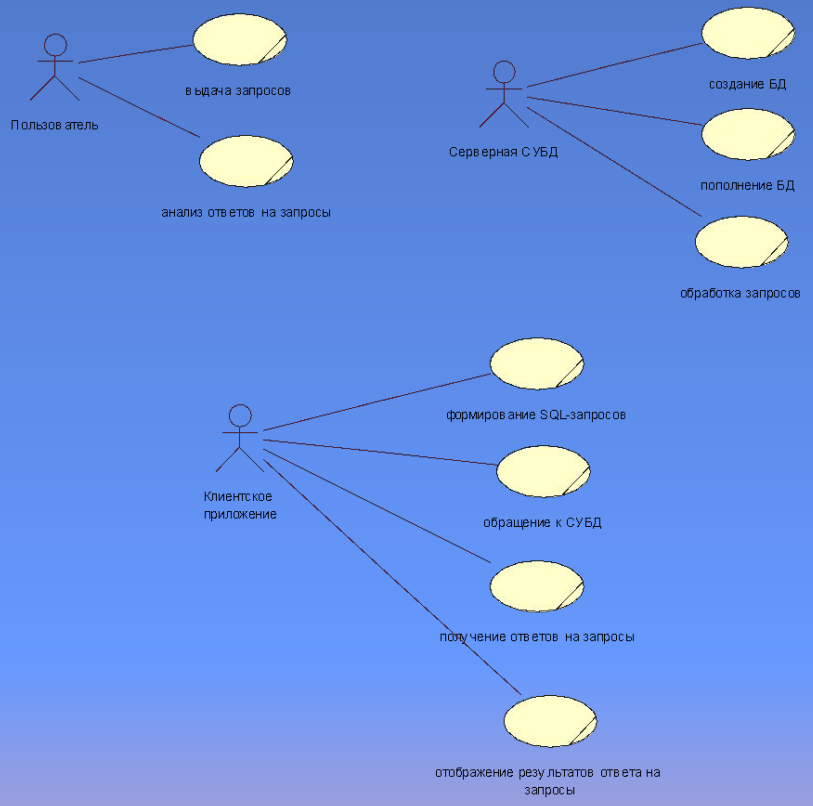
- Создание информационных систем, основанных на СУБД, используется для решения различных практических задач. Одной из таких задач является принятие решений в различных сферах деятельности человека, в том числе в экономической сфере. В условиях нарастающей конкурентной борьбы в экономической сфере решение таких задач при помощи компьютеров является наиболее востребованным.
- Приведем результаты работ, выполненных на этапах анализа (см. рис. 1, 2 и 3) и проектирования (см. рис. 4, 5 и 6) создания прототипа информационной системы, используемой для выбора программных продуктов фирмы Rational Rose.

РАЗРАБОТКА ПРОТОТИПОВ ИНФОРМАЦИОННЫХ СИСТЕМ, ОСНОВАННЫХ НА СУБД, В CASE-СРЕДСТВЕ RATIONAL ROSE ENTERPRISE И СРЕДЕ ПРОГРАММИРОВАНИЯ C++BUILDER

- Рис. 1. Диагра деятельности

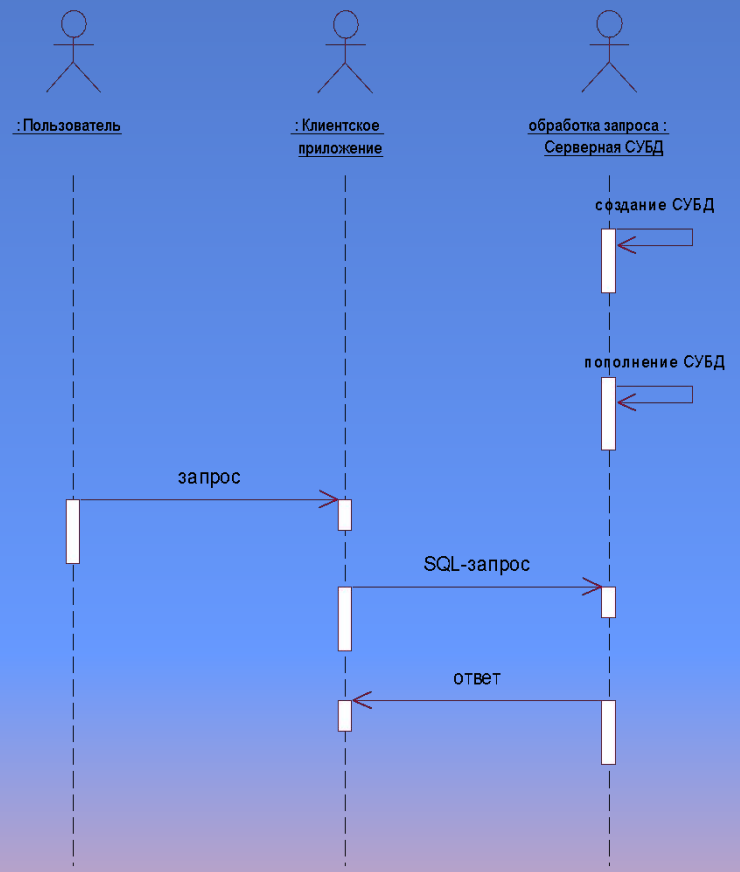
РАЗРАБОТКА ПРОТОТИПОВ ИНФОРМАЦИОННЫХ СИСТЕМ, ОСНОВАННЫХ НА СУБД, В CASE-СРЕДСТВЕ RATIONAL ROSE ENTERPRISE И СРЕДЕ ПРОГРАММИРОВАНИЯ C++BUILDER

- Рис. 2. Диаграмма прецедентов



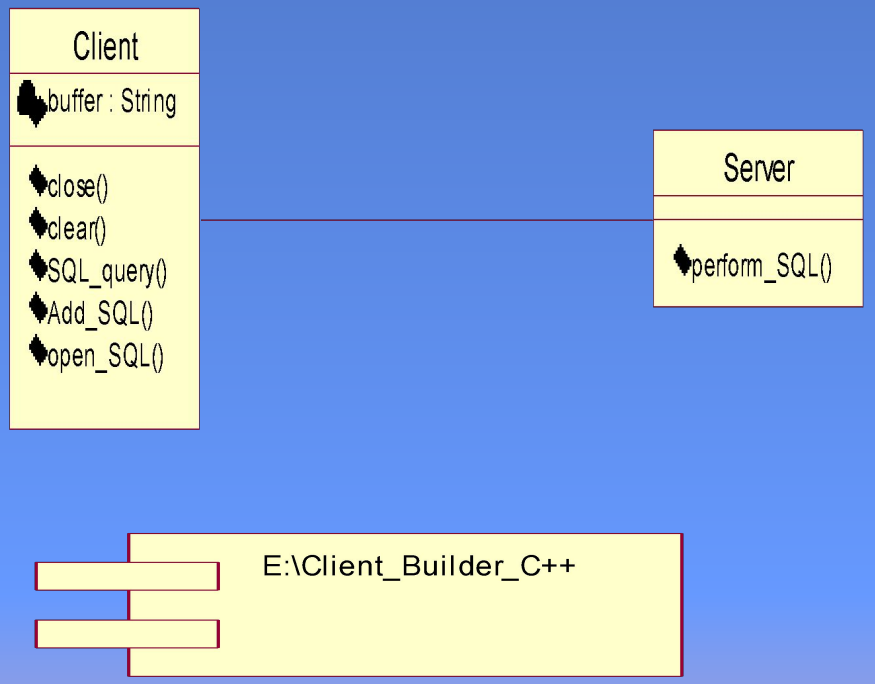
РАЗРАБОТКА ПРОТОТИПОВ ИНФОРМАЦИОННЫХ СИСТЕМ, ОСНОВАННЫХ НА СУБД, В CASE-СРЕДСТВЕ RATIONAL ROSE ENTERPRISE И СРЕДЕ ПРОГРАММИРОВАНИЯ C++BUILDER

- Рис. 3. Диаграмма последовательностей



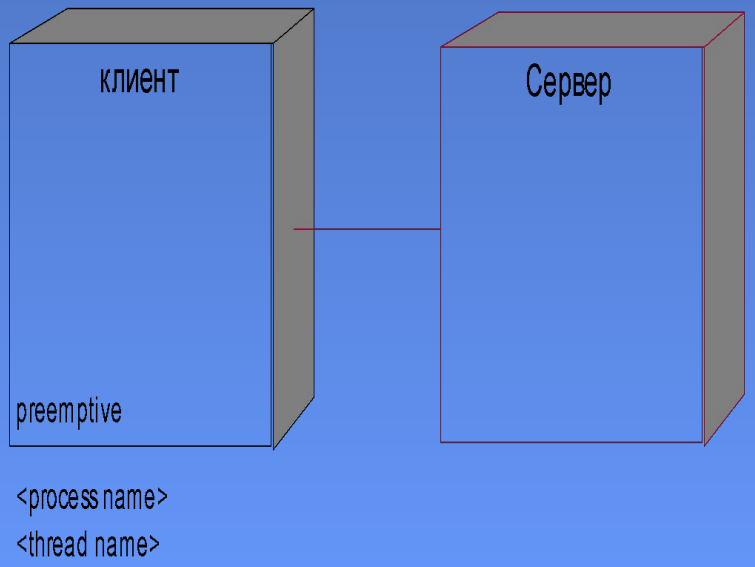
РАЗРАБОТКА ПРОТОТИПОВ ИНФОРМАЦИОННЫХ СИСТЕМ, ОСНОВАННЫХ НА СУБД, В CASE-СРЕДСТВЕ RATIONAL ROSE ENTERPRISE И СРЕДЕ ПРОГРАММИРОВАНИЯ C++BUILDER

- Рис. 4. Диаграмма классов
- Рис. 3.5 Диаграмма компонент



РАЗРАБОТКА ПРОТОТИПОВ ИНФОРМАЦИОННЫХ СИСТЕМ, ОСНОВАННЫХ НА СУБД, В CASE-СРЕДСТВЕ RATIONAL ROSE ENTERPRISE И СРЕДЕ ПРОГРАММИРОВАНИЯ C++BUILDER

- Рис. 6. Диаграмма размещения

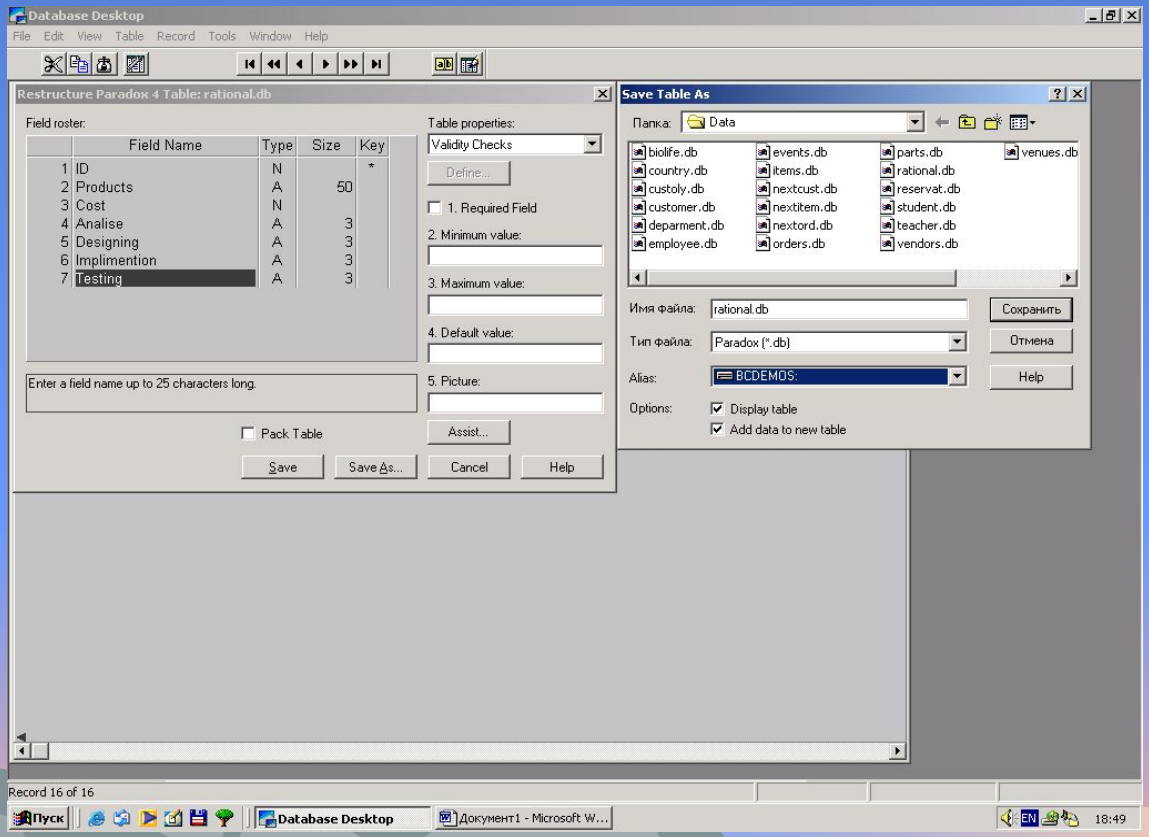


РАЗРАБОТКА ПРОТОТИПОВ ИНФОРМАЦИОННЫХ СИСТЕМ, ОСНОВАННЫХ НА СУБД, В CASE-СРЕДСТВЕ RATIONAL ROSE ENTERPRISE И СРЕДЕ ПРОГРАММИРОВАНИЯ C++BUILDER

- Для разработки прототипа информационной системы, основанного на СУБД, в среде программирования C++Builder необходимо открыть Database Desktop, определить структуры таблиц (наименования столбцов, ключи и типы столбцов) и создать таблицы (см. рис. 7 и 8).

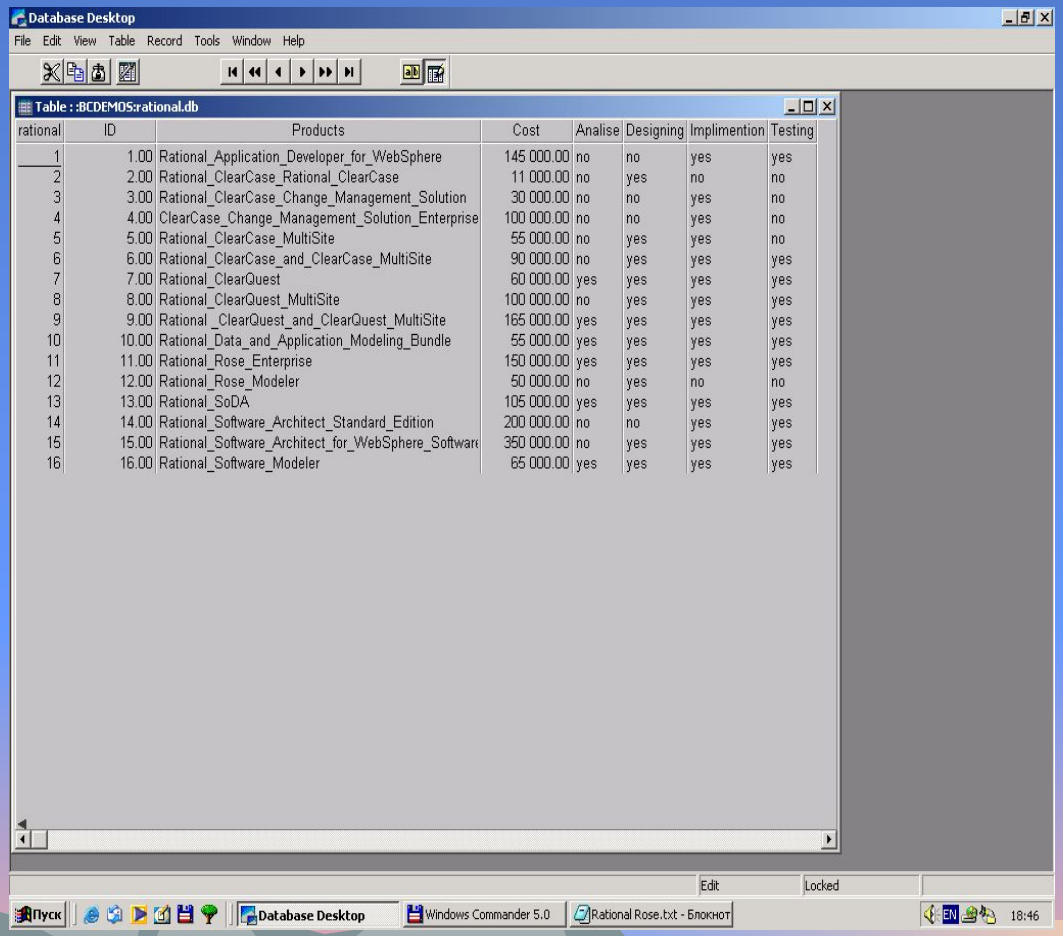
РАЗРАБОТКА ПРОТОТИПОВ ИНФОРМАЦИОННЫХ СИСТЕМ, ОСНОВАННЫХ НА СУБД, В CASE-СРЕДСТВЕ RATIONAL ROSE ENTERPRISE И СРЕДЕ ПРОГРАММИРОВАНИЯ C++BUILDER

- Рис. 7. Структура таблицы



РАЗРАБОТКА ПРОТОТИПОВ ИНФОРМАЦИОННЫХ СИСТЕМ, ОСНОВАННЫХ НА СУБД, В CASE-СРЕДСТВЕ RATIONAL ROSE ENTERPRISE И СРЕДЕ ПРОГРАММИРОВАНИЯ C++BUILDER

- Рис. 8. Таблица



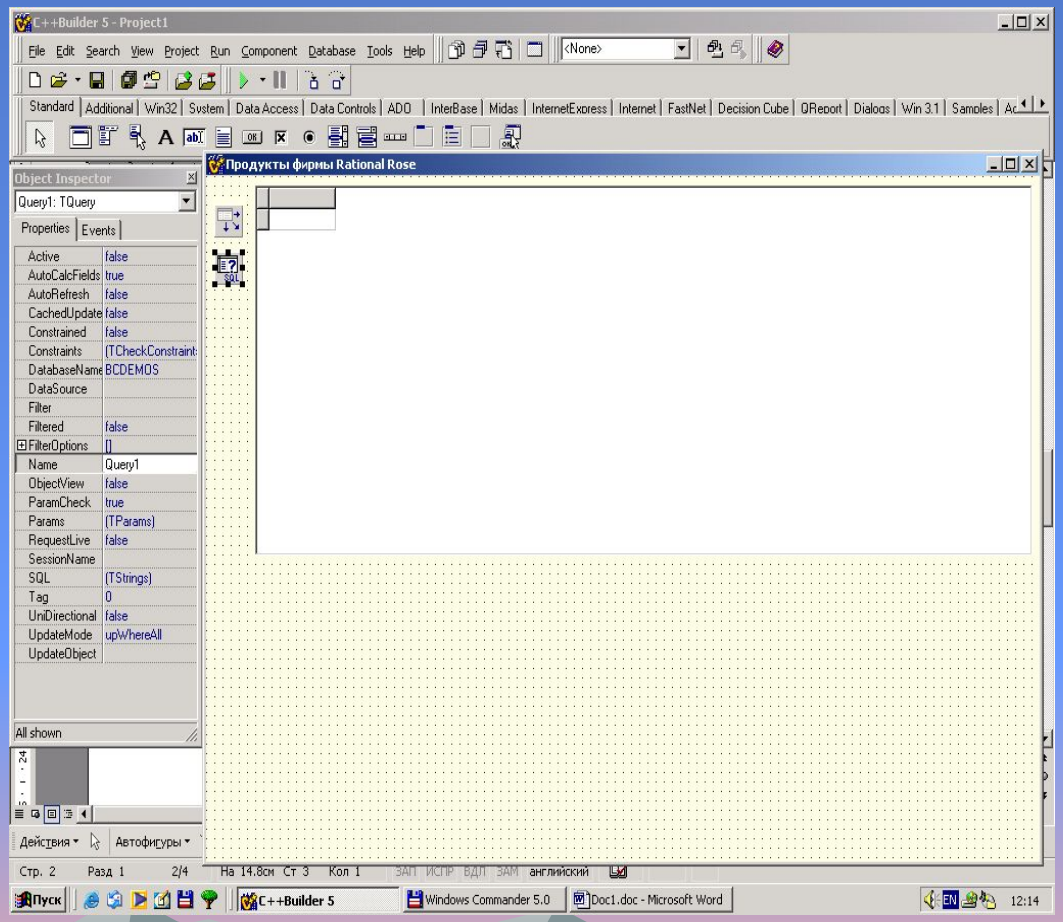
rational	ID	Products	Cost	Analise	Designing	Implimention	Testing
1	1.00	Rational_Application_Developer_for_WebSphere	145 000.00	no	no	yes	yes
2	2.00	Rational_ClearCase_Rational_ClearCase	11 000.00	no	yes	no	no
3	3.00	Rational_ClearCase_Change_Management_Solution	30 000.00	no	no	yes	no
4	4.00	ClearCase_Change_Management_Solution_Enterprise	100 000.00	no	no	yes	no
5	5.00	Rational_ClearCase_MultiSite	55 000.00	no	yes	yes	no
6	6.00	Rational_ClearCase_and_ClearCase_MultiSite	90 000.00	no	yes	yes	yes
7	7.00	Rational_ClearQuest	60 000.00	yes	yes	yes	yes
8	8.00	Rational_ClearQuest_MultiSite	100 000.00	no	yes	yes	yes
9	9.00	Rational _ClearQuest_and_ClearQuest_MultiSite	165 000.00	yes	yes	yes	yes
10	10.00	Rational_Data_and_Application_Modeling_Bundle	55 000.00	yes	yes	yes	yes
11	11.00	Rational_Rose_Enterprise	150 000.00	yes	yes	yes	yes
12	12.00	Rational_Rose_Modeler	50 000.00	no	yes	no	no
13	13.00	Rational_SoDA	105 000.00	yes	yes	yes	yes
14	14.00	Rational_Software_Architect_Standard_Edition	200 000.00	no	no	yes	yes
15	15.00	Rational_Software_Architect_for_WebSphere_Softwar	350 000.00	no	yes	yes	yes
16	16.00	Rational_Software_Modeler	65 000.00	yes	yes	yes	yes

РАЗРАБОТКА ПРОТОТИПОВ ИНФОРМАЦИОННЫХ СИСТЕМ, ОСНОВАННЫХ НА СУБД, В CASE-СРЕДСТВЕ RATIONAL ROSE ENTERPRISE И СРЕДЕ ПРОГРАММИРОВАНИЯ C++BUILDER

- Открыть среду программирования C++Builder и выбрать объекты Query1, DataSource1 и DBGrid1. После выбора этих объектов устанавливаются их свойства (см. рис. 9, 10 и 11).
- Рассмотрим реализацию динамических SQL-запросов:
- показать всю информацию;
- показать программные продукты, стоимость которых не превышает заданное значение;
- показать программные продукт, которые поддерживают заданные этапы жизненного цикла разработки программных средств;
- показать программные продукт, имеющие максимальную или минимальную стоимость.
- Интерфейсы разработанного прототипа программных средств приведен на рис. 12 (получен ответ на 1-й запрос), 13 (получен ответ на 2-й запрос), 14 (получен ответ на 3-й запрос) и 15 (получен ответ на 4-й запрос).

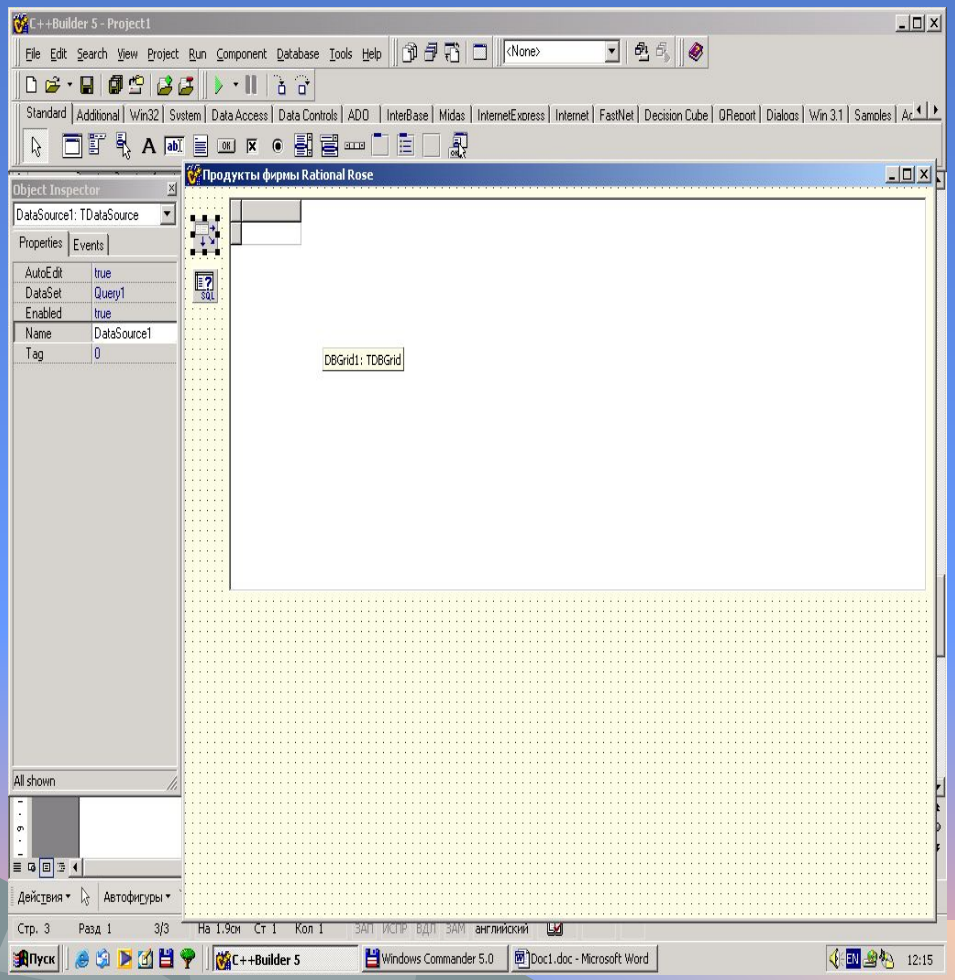
РАЗРАБОТКА ПРОТОТИПОВ ИНФОРМАЦИОННЫХ СИСТЕМ, ОСНОВАННЫХ НА СУБД, В CASE-СРЕДСТВЕ RATIONAL ROSE ENTERPRISE И СРЕДЕ ПРОГРАММИРОВАНИЯ C++BUILDER

- Рис. 9. Свойства объекта Query1



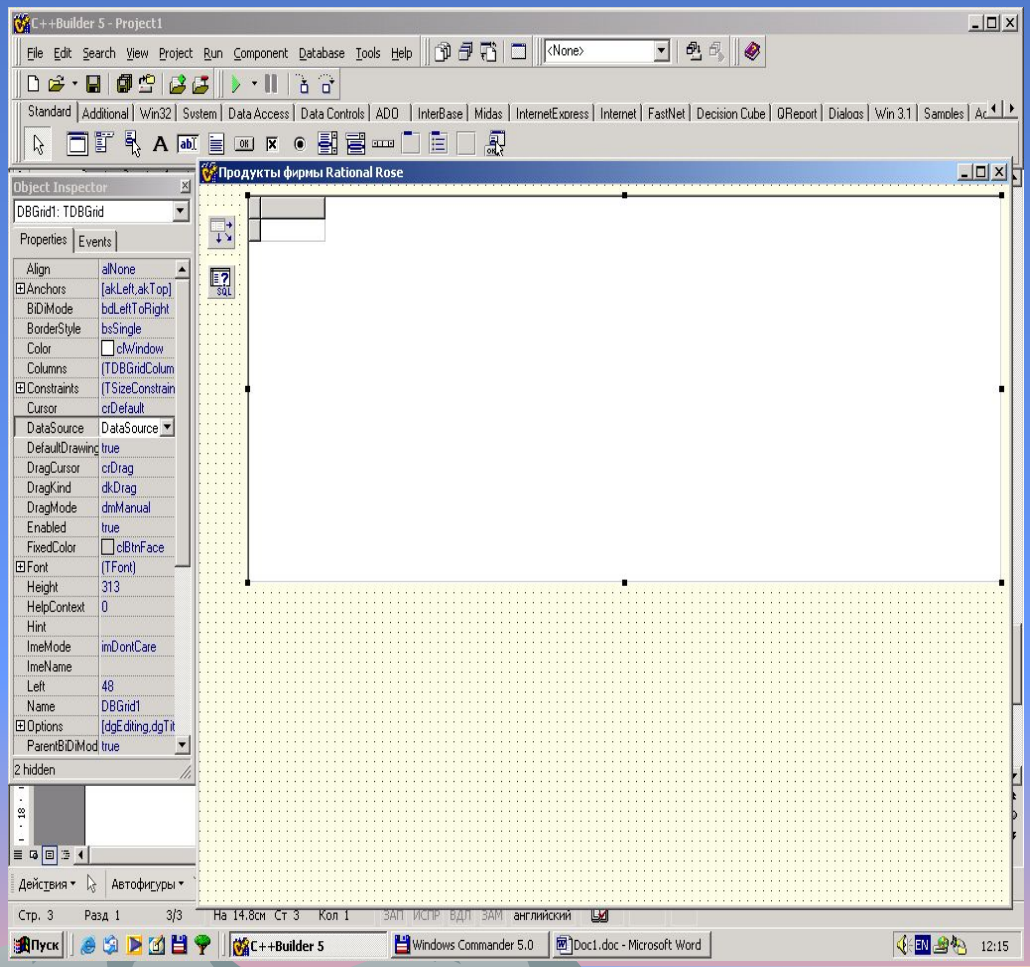
РАЗРАБОТКА ПРОТОТИПОВ ИНФОРМАЦИОННЫХ СИСТЕМ, ОСНОВАННЫХ НА СУБД, В CASE-СРЕДСТВЕ RATIONAL ROSE ENTERPRISE И СРЕДЕ ПРОГРАММИРОВАНИЯ C++BUILDER

- Рис. 10. Свойства объекта DataSource1



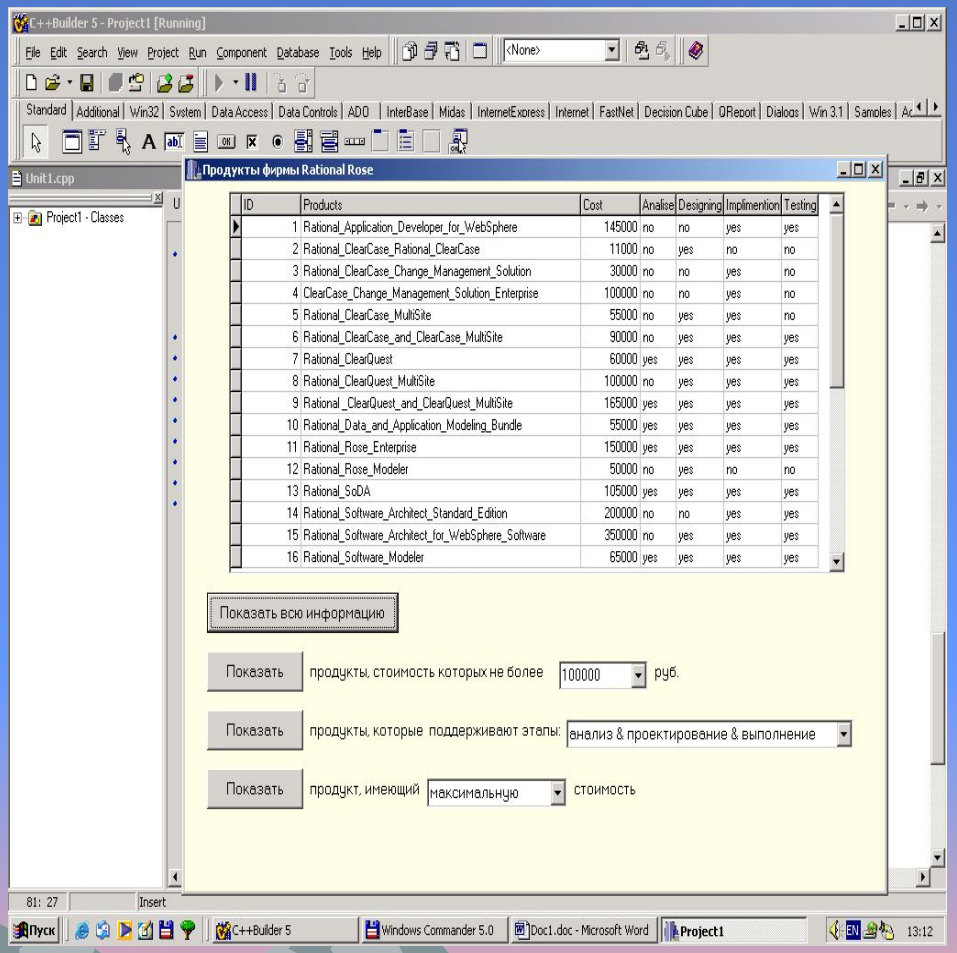
РАЗРАБОТКА ПРОТОТИПОВ ИНФОРМАЦИОННЫХ СИСТЕМ, ОСНОВАННЫХ НА СУБД, В CASE-СРЕДСТВЕ RATIONAL ROSE ENTERPRISE И СРЕДЕ ПРОГРАММИРОВАНИЯ C++BUILDER

- Рис. 11. Свойства объекта DBGrid1



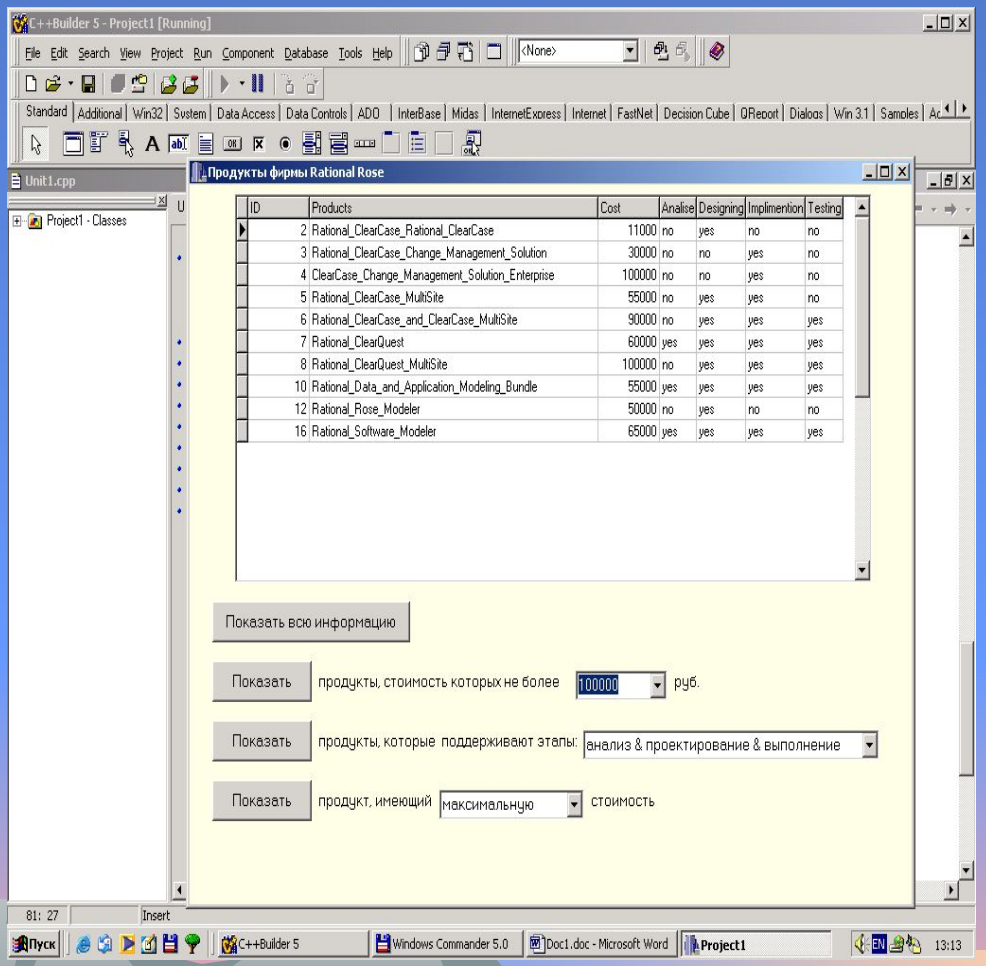
РАЗРАБОТКА ПРОТОТИПОВ ИНФОРМАЦИОННЫХ СИСТЕМ, ОСНОВАННЫХ НА СУБД, В CASE-СРЕДСТВЕ RATIONAL ROSE ENTERPRISE И СРЕДЕ ПРОГРАММИРОВАНИЯ C++BUILDER

- Рис. 12. Интерфейс разработанного прототипа программных средств (получен ответ на 1-й запрос)



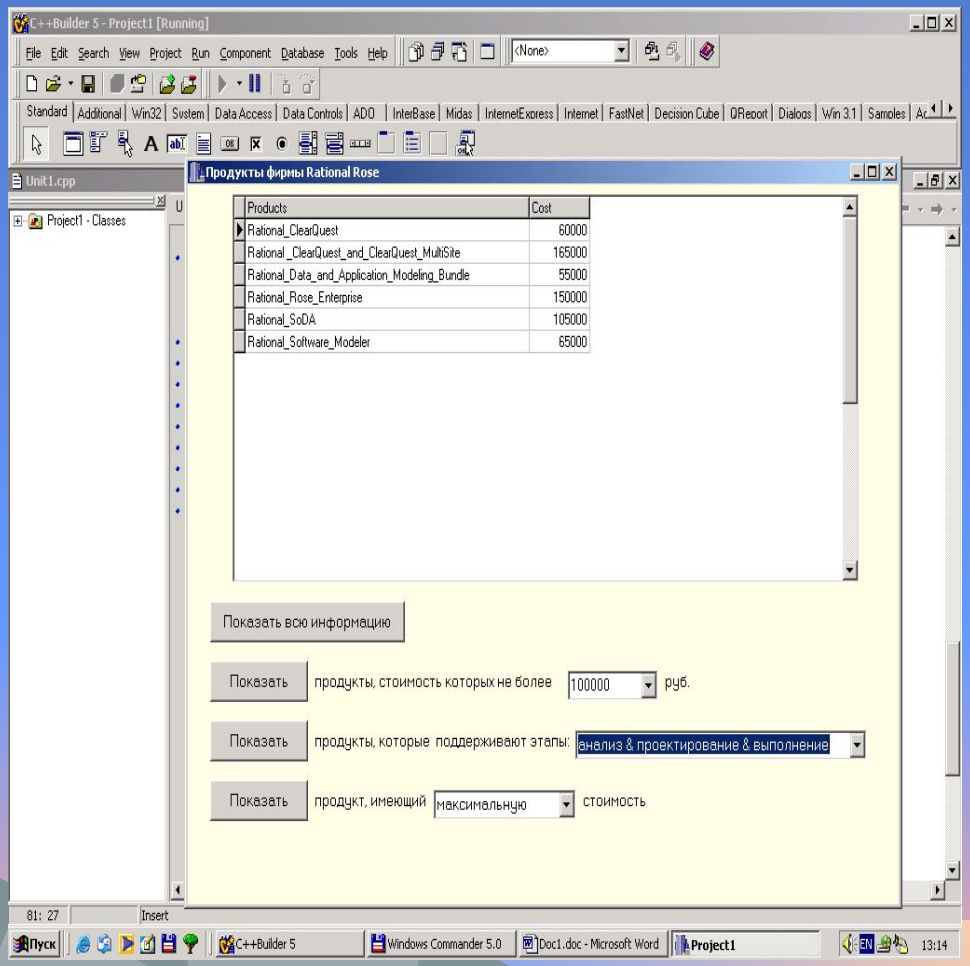
РАЗРАБОТКА ПРОТОТИПОВ ИНФОРМАЦИОННЫХ СИСТЕМ, ОСНОВАННЫХ НА СУБД, В CASE-СРЕДСТВЕ RATIONAL ROSE ENTERPRISE И СРЕДЕ ПРОГРАММИРОВАНИЯ C++BUILDER

- Рис. 13. Интерфейс разработанного прототипа программных средств (получен ответ на 2-й запрос)



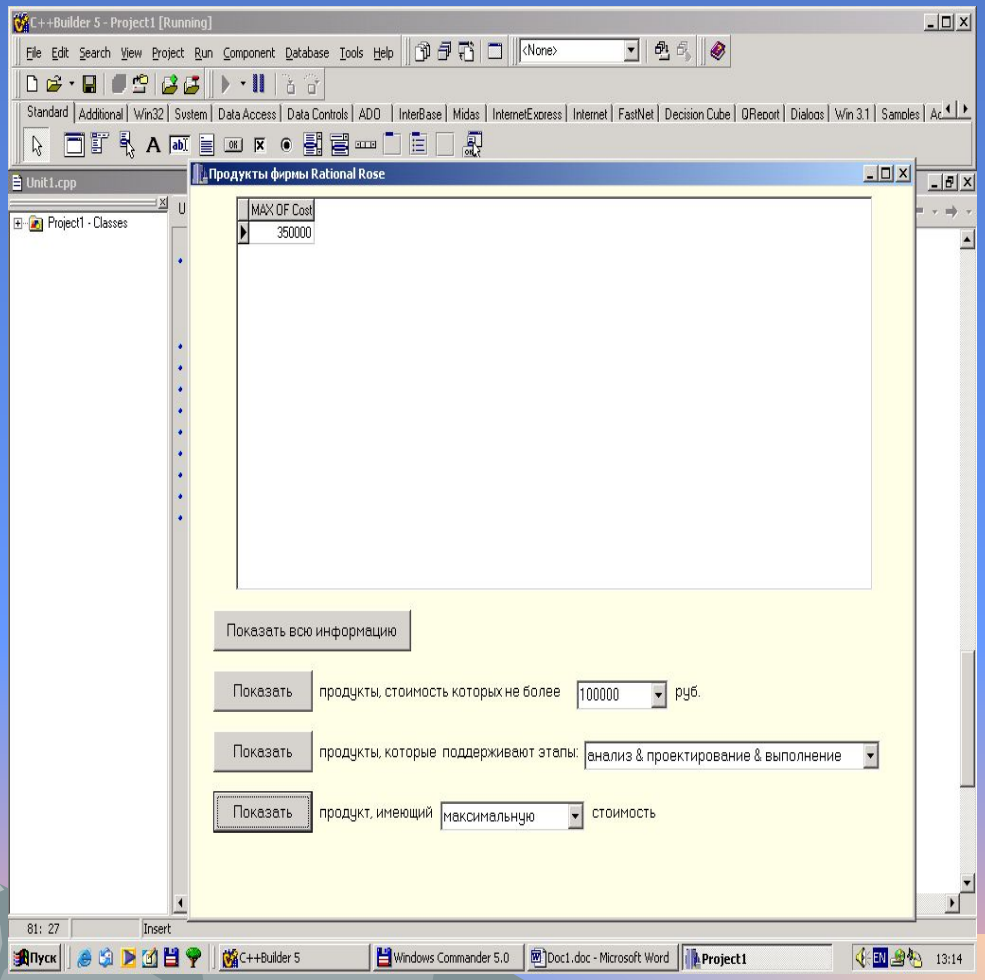
РАЗРАБОТКА ПРОТОТИПОВ ИНФОРМАЦИОННЫХ СИСТЕМ, ОСНОВАННЫХ НА СУБД, В CASE-СРЕДСТВЕ RATIONAL ROSE ENTERPRISE И СРЕДЕ ПРОГРАММИРОВАНИЯ C++BUILDER

- Рис. 14. Интерфейс разработанного прототипа программных средств (получен ответ на 3-й запрос)



РАЗРАБОТКА ПРОТОТИПОВ ИНФОРМАЦИОННЫХ СИСТЕМ, ОСНОВАННЫХ НА СУБД, В CASE-СРЕДСТВЕ RATIONAL ROSE ENTERPRISE И СРЕДЕ ПРОГРАММИРОВАНИЯ C++BUILDER

- Рис.15. Интерфейс разработанного прототипа программных средств (получен ответ на 4-й запрос)



РАЗРАБОТКА ПРОТОТИПОВ ИНФОРМАЦИОННЫХ СИСТЕМ, ОСНОВАННЫХ НА СУБД, В CASE-СРЕДСТВЕ RATIONAL ROSE ENTERPRISE И СРЕДЕ ПРОГРАММИРОВАНИЯ C++BUILDER

- Приведем программный код, реализующий динамические SQL-запросы.

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------------------------
#include <stdio.h>
void __fastcall TForm1::Button1Click(TObject *Sender)
{
char buffer[250];
Query1->Close();
Query1->SQL->Clear();
sprintf(buffer,"Select * from rational.db");
Query1->SQL->Add(buffer);
Query1->Open();
}
//---------------------------------------------------------------------------
```

РАЗРАБОТКА ПРОТОТИПОВ ИНФОРМАЦИОННЫХ СИСТЕМ, ОСНОВАННЫХ НА СУБД, В CASE-СРЕДСТВЕ RATIONAL ROSE ENTERPRISE И СРЕДЕ ПРОГРАММИРОВАНИЯ C++BUILDER

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
char buffer[250];
Query1->Close();
Query1->SQL->Clear();
sprintf(buffer,"Select * from rational.db where cost<=%f",atof(ComboBox1->Text.c_str()));
Query1->SQL->Add(buffer);
Query1->Open();
}
//---------------------------------------------------------------------------
```

```
void __fastcall TForm1::Button3Click(TObject *Sender)
{
char buffer[250];
Query1->Close();
Query1->SQL->Clear();
if (!strcmp(ComboBox2->Text.c_str(),"анализ"))
   sprintf(buffer,"Select Products,Cost from rational.db where Analise='yes'");
if (!strcmp(ComboBox2->Text.c_str(),"проектирование"))
   sprintf(buffer,"Select Products,Cost from rational.db where Designing='yes'");
if (!strcmp(ComboBox2->Text.c_str(),"анализ & проектирование"))
   sprintf(buffer,"Select Products,Cost from rational.db where Analise='yes' and Designing='yes'");
if (!strcmp(ComboBox2->Text.c_str(),"выполнение"))
   sprintf(buffer,"Select Products,Cost from rational.db where Implimention='yes'");
if (!strcmp(ComboBox2->Text.c_str(),"проектирование & выполнение"))
   sprintf(buffer,"Select Products,Cost from rational.db where Designing='yes'and Implimention='yes'");
if (!strcmp(ComboBox2->Text.c_str(),"анализ & проектирование & выполнение"))
   sprintf(buffer,"Select Products,Cost from rational.db where Analise='yes' and Designing='yes'and
Implimention='yes'");
if (!strcmp(ComboBox2->Text.c_str(),"тестирование"))
   sprintf(buffer,"Select Products,Cost from rational.db where Testing='yes'");
if (!strcmp(ComboBox2->Text.c_str(),"выполнение & тестирование"))
   sprintf(buffer,"Select Products,Cost from rational.db where Implimention='yes' and Testing='yes'");
if (!strcmp(ComboBox2->Text.c_str(),"проектирование & выполнение & тестирование"))
   sprintf(buffer,"Select Products,Cost from rational.db where Designing='yes' and Implimention='yes' and
Testing='yes'");
if (!strcmp(ComboBox2->Text.c_str(),"анализ & проектирование & выполнение & тестирование"))
   sprintf(buffer,"Select Products,Cost from rational.db where Analise='yes' and Designing='yes' and
Implimention='yes' and Testing='yes'");
Query1->SQL->Add(buffer);
Query1->Open();
}
```

РАЗРАБОТКА ПРОТОТИПОВ ИНФОРМАЦИОННЫХ СИСТЕМ, ОСНОВАННЫХ НА СУБД, В CASE-СРЕДСТВЕ RATIONAL ROSE ENTERPRISE И СРЕДЕ ПРОГРАММИРОВАНИЯ C++BUILDER

```
//---------------------------------------------------------------------------
void __fastcall TForm1::Button4Click(TObject *Sender)
{
char buffer[250];
Query1->Close();
Query1->SQL->Clear();
if (!strcmp(ComboBox3->Text.c_str(),"минимальную"))
   sprintf(buffer,"Select MIN(Cost) from rational.db");
if (!strcmp(ComboBox3->Text.c_str(),"максимальную"))
   sprintf(buffer,"Select MAX(Cost) from rational.db");
Query1->SQL->Add(buffer);
Query1->Open();
}
//---------------------------------------------------------------------------
```

Этапы реинжиниринга бизнес-процессов в университете

- Проект по БПР в университете включает следующие шесть этапов.

1. Разработка будущего образа университета – спецификация целей университета, исходя из существующей правовой, Устава университета, стратегий развития и потребностей в специалистах сферы образования, конъюнктуры целевых отраслей трудоустройства выпускников университета и заказов на научно-исследовательские и опытно-конструкторские работы, а также из текущего состояния университета.
2. Создание функциональной модели университета. На этом этапе руководство университетом вместе с экспертами (руководителями соответствующих структурных подразделений) и разработчиками информационных систем должны создать детальное описание модели функционирования университета, т.е. определить и документировать его основные бизнес-процессы, а также оценить их эффективность.
3. Перепроектирование основных бизнес-процессов. На этом этапе руководство университетом вместе с экспертами (менеджерами) и с участием разработчиков информационных систем должны разработать более эффективные структуры управления и процедур функционирования по сравнению с существующими.

Этапы реинжиниринга бизнес-процессов в университете

4. Детальная проработка бизнес-процессов университета. На этом этапе проектируются различные виды работ, подготавливается система мотивации, организуются команды по выполнению работ и группы поддержки качества, создаются программы подготовки специалистов и т.д.
5. Разработка поддерживающих корпоративных информационных и информационно-управляющих систем (КИС и КИУС). На этом этапе определяются имеющиеся ресурсы (аппаратное и программное обеспечение) и реализуется информационно-управляющая система университета.
6. Внедрение перепроектированных процессов. Интеграция и тестирование разработанных процессов и информационно-управляющей системы, обучение сотрудников, установка информационно-управляющей системы, переход к новой работе университета (создание новых должностных инструкций сотрудников).

Перечисленные этапы выполняются не последовательно, а, по крайней мере, частично параллельно, причем некоторые этапы повторяются. Более того, для разработки общего представления будущего университета надо сначала разобраться в существующей деятельности университета.

CASE-средства создания информационных систем

CASE-средства фирмы Platinum technology

Методология IDEF

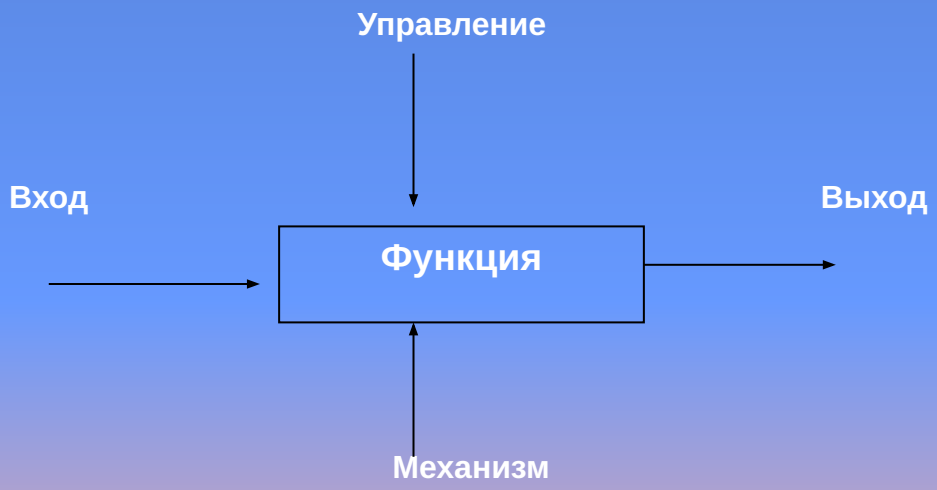
- Наиболее удобным языком моделирования бизнес-процессов является IDEF0, предложенный более 20 лет назад Дугласом Россом (SoftTech, Inc.) и называвшийся первоначально SADT – Structured Analysis and Design Technique. В начале 70-х годов вооруженные силы США применили подмножество SADT, касающееся моделирования процессов, для реализации проектов в рамках программы ICAM (Integrated Computer-Aided Manufacturing). В дальнейшем это подмножество SADT было принято в качестве федерального стандарта США под наименованием IDEF0. В последние годы разработана серия стандартов, включающая стандарты IDEF0 и IDEF1X, рассмотренные в ниже, IDEF3, IDEF4, IDEF5, описания которых приведены в приложениях, а также IDEF6 (Design Rationale Capture), IDEF8 (User Interface Modeling), IDEF9 (Scenario-Driven IS Design), IDEF10 (Implementation Architecture Modeling), IDEF11(Information Artifact Modeling), IDEF11(Organization Modeling), IDEF13 (Three Schema Mapping Design) и IDEF14 (Network Design).

Методология IDEF

- В IDEF0 система представляется как совокупность взаимодействующих работ или функций. Такая чисто функциональная ориентация является принципиальной - функции системы анализируются независимо от объектов, которыми они оперируют. Это позволяет более четко смоделировать логику и взаимодействие процессов организации.
- В основу IDEF0 положен формализованный язык, характеризующийся точными правилами построения функциональной модели и ее понимания. Алфавит этого языка включает совокупность графических символов, из которых строятся функциональные схемы (выражения) в соответствии с правилами синтаксиса, т.е. правилами построения схем. Для аналитического представления функциональной модели, а также для описания свойств и семантики используются естественные и логико-математические языки.

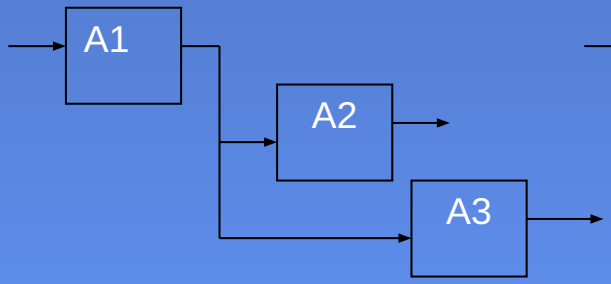
Функциональный блок

- Стрелки играют роль интерфейса и означают либо предметы (материальные объекты), либо информационные объекты – данные.
- Функции должны иметь имена, выраженные грамматической формой глагола. Стрелки помечаются уникальными метками, выраженными грамматической формой существительного и называются ICOM-метками (Input, Control, Output, Mechanism).

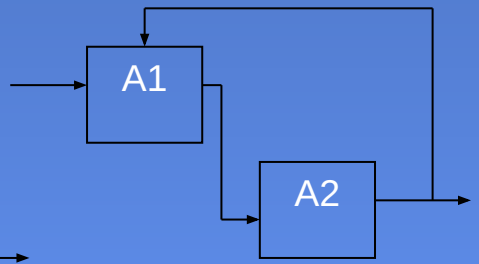


Основные правила соединения блоков

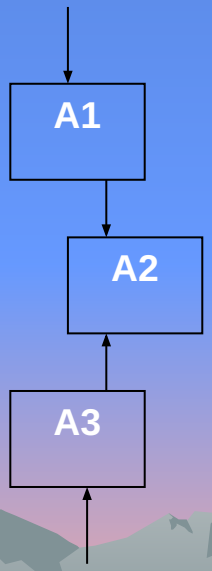
А- одновременное действие; Б -обратная связь; В- взаимный переход меток; Г- разветвление стрелки для управления функциями А1 и А2



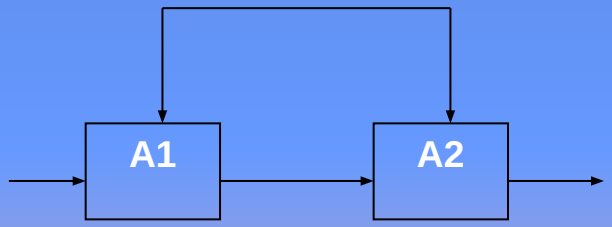
А)



Б)



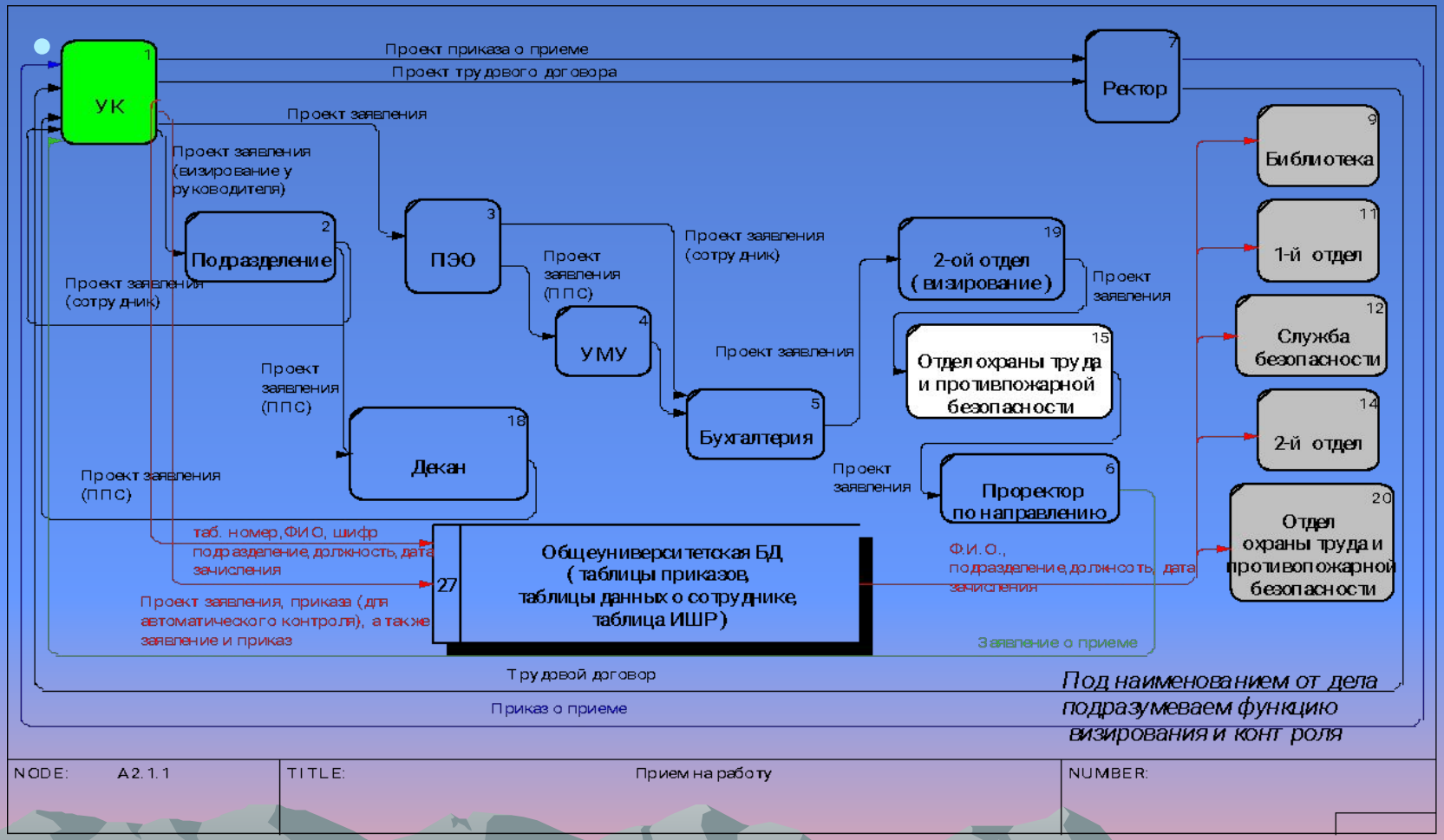
В)



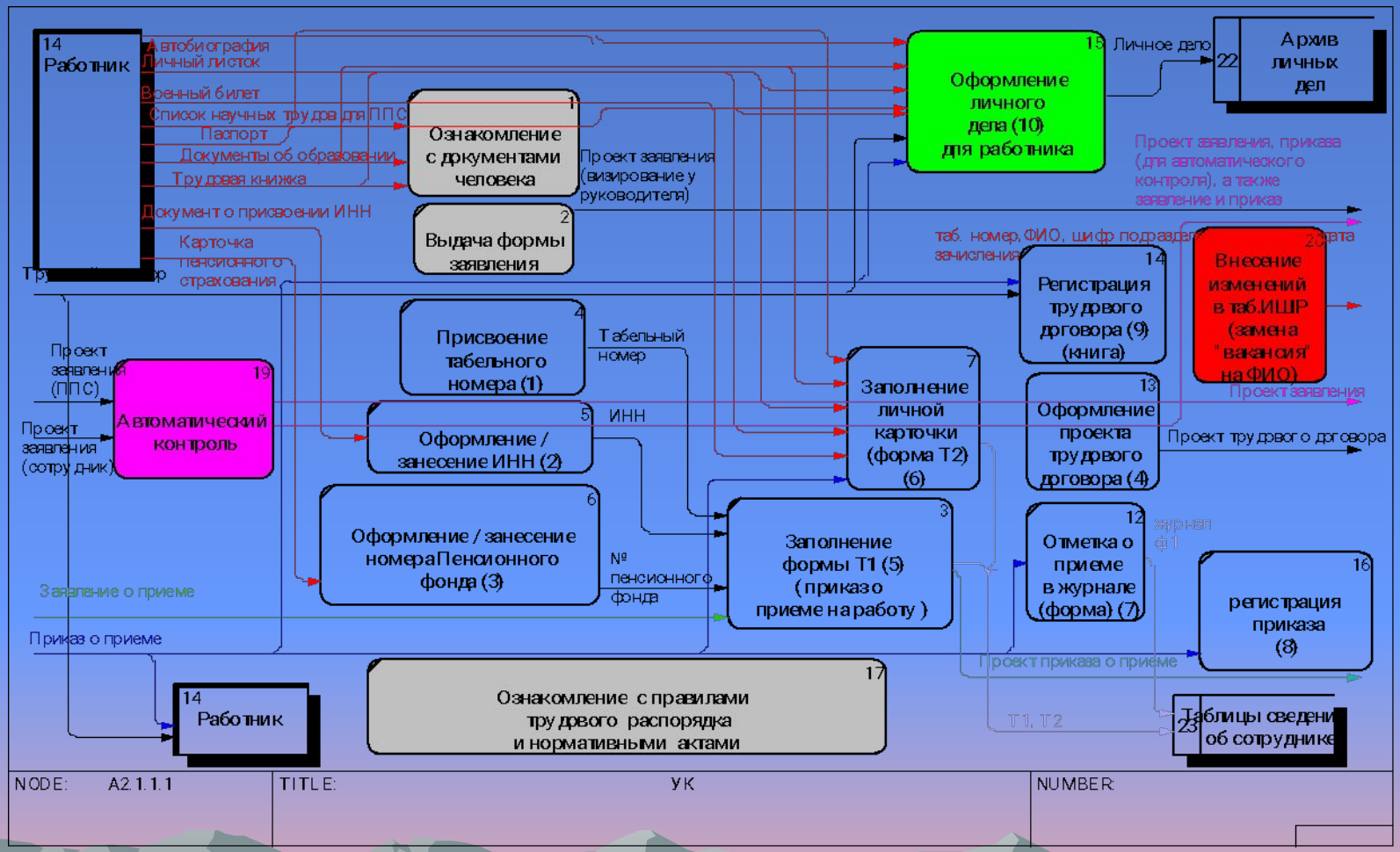
Г)

- Схему блоков, соединенных по приведенным выше правилам, называют диаграммой соответствующего уровня иерархии.
- Модель IDEF0 представляет иерархически образованный набор диаграмм. Иерархия диаграмм определяется схемой декомпозиции блоков. Под иерархией будем понимать уровни раскрытия блоков функциональной модели по мере уменьшения их сложности. Каждый блок может быть декомпозирован на другой диаграмме.

Бизнес-процесс «Прием на работу сотрудников»



Бизнес-процесс «Прием на работу сотрудников»



Стандарт IDEF1X

- Рассмотрим методологию IDEF1X. Методология IDEF1X представляет собой формализованный язык семантического (контекстного) моделирования данных, основанный на концепции "Сущность - Отношение" (Entity-Relatioship). Это инструмент для анализа информационных структур систем различной природы. Информационное моделирование заключается в анализе логической структуры информации об объектах системы. Логическая структура является необходимым дополнением функциональной модели, детализируя объекты, которыми манипулируют функции системы. Теоретической базой построения информационных моделей является теория баз данных.
- Приведем краткое описание языка моделирования логических структур данных.

Правила определения сущностей

- **Сущность -** множество реальных или абстрактных объектов, обладающих общими атрибутами или характеристиками.
- ***Правила определения сущностей***
- 1. Сущность должна иметь уникальное имя.
- 2. Сущность обладает одним или несколькими атрибутами, которые либо принадлежат сущности, либо наследуются через отношения.
- 3. Сущность обладает одним или несколькими атрибутами, которые однозначно идентифицируют каждый образец сущности и называются ключом или составным ключом.
- 4. Каждая сущность может обладать любым количеством отношений с другими сущностями.
- 5. Если внешний ключ целиком используется в составе первичного ключа, то сущность является зависимой от идентификатора.
- Сущность должна обладать **атрибутом** или **комбинацией атрибутов,** чьи значения однозначно определяют каждый экземпляр сущности. Эти атрибуты образуют первичный ключ сущности.

Правила определения атрибутов

- 1. Каждый атрибут каждой сущности обладает уникальным именем.
- 2. Сущность может обладать любым количеством атрибутов.
- 3. При идентифицирующем отношении сущность "потомок" наследует атрибут и/или атрибуты, составляющие первичный ключ сущности "родителя".

Первичные и альтернативные ключи

- Возможный ключ - это один или несколько атрибутов, чьи значения однозначно определяют каждый экземпляр сущности. При существовании нескольких возможных ключей один из них назначается первичным, а остальные формируют альтернативные ключи

Правила определения отношений

- 1. При определении отношения типа «родитель - потомок» экземпляр «потомка» связан с одним "родителем". Экземпляр «родитель» может быть связан с любым числом экземпляров «потомков».
- 2. В идентифицирующем отношении сущность-«потомок» всегда является зависимой от идентифицирующей сущности.
- 3. Сущность может быть связана с любым количеством других сущностей как в качестве "потомка", так и в качестве "родителя".
- 4. Отношения определяются мощностью: 0,1 или более.

Отношения категоризации

- Отношения полной категоризации - это отношения между двумя или более сущностями, в которых каждый экземпляр одной сущности, названной общей сущностью, связан в точности с одним экземпляром сущности, называемой сущностями-категориями.

Правила определения отношений категоризации

- 1. Сущность типа "категория" может иметь только одну общую сущность.
- 2. Сущность-категория, принадлежащая одному отношению категоризации, может **быть** общей сущностью в другом отношении категоризации.
- 3. Сущность может быть общей сущностью в любом количестве отношений категоризации.
- 4. Атрибуты первичного ключа сущности-категории должны совпадать с атрибутами первичного ключа общей сущности.
- Все экземпляры сущности-категории имеют одно и то же значение дискриминатора, и все экземпляры других категорий должны иметь другие значения дискриминатора.

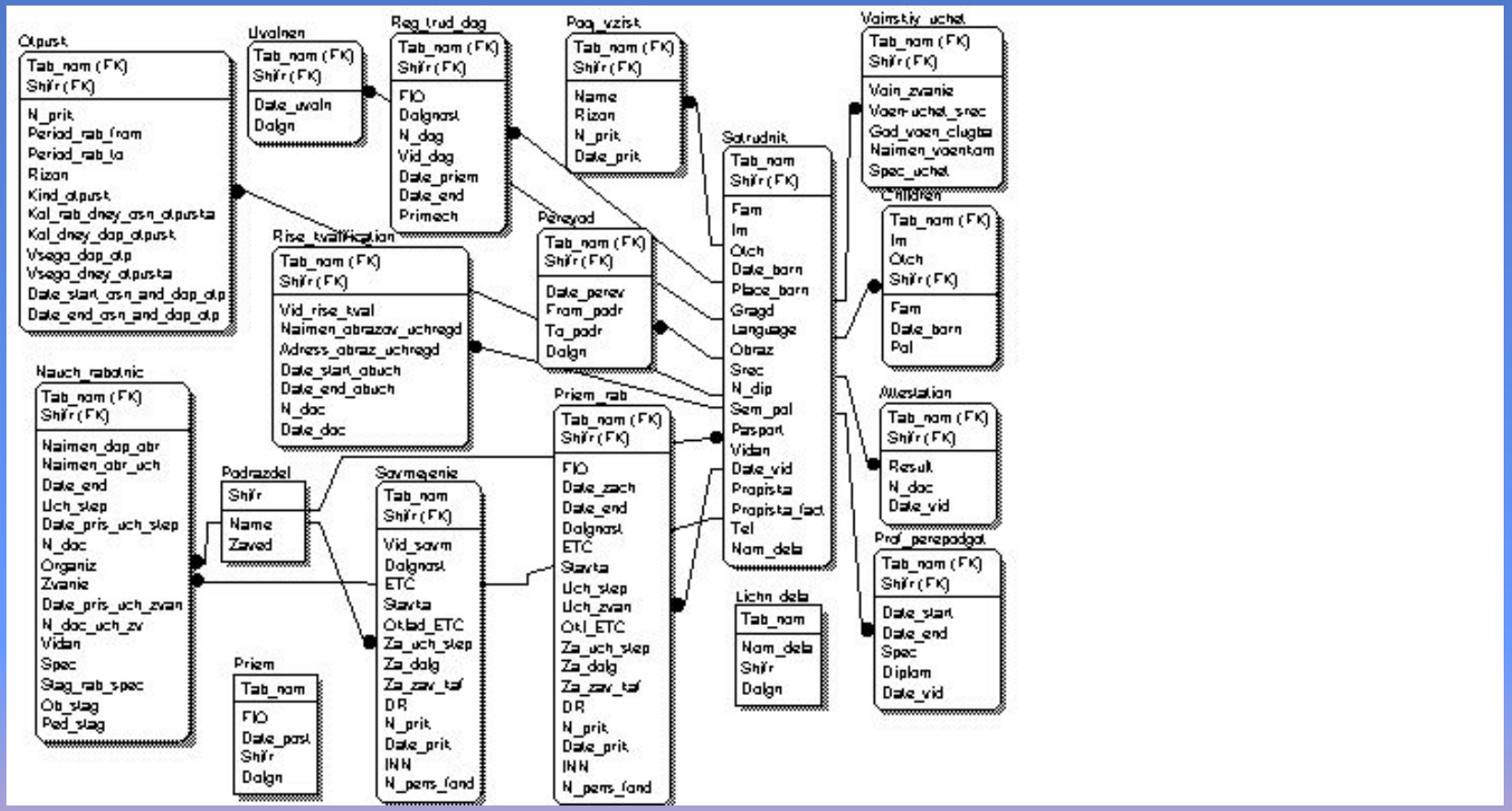
Основные правила формирования информационной модели

- 1. Все стрелки (входные, выходные, управляющие, механизмов исполнения) становятся потенциальными сущностями, а функции, связывающие их, трансформируются в отношения между этими сущностями. Для этого составляется пул (список потенциальных сущностей).
- 2. Число сущностей и связей в IDEF1X модели считается необозримым, если их количество превышает 25-30. Поэтому далее рассматривается совокупность сущностей и отношений для каждой функции.

Основные правила формирования информационной модели

- 1. Функциональный подход представляет совокупность сущностей и отношений в целом как информационную структуру обобщенного документа или отчета.
- 2. Функциональный подход обычно ограничивается рассмотрением не более 25-30 сущностей.
- 3. Информационная модель функции должна позволять:
- •воспроизвести структуру документа и часть информации в нем;
- •воспроизвести информацию порождаемого документа.
- Это информация обычно в форме вопросов:
- • где что-либо хранится?
- •где что-либо может храниться?
- Текстовые пояснения заносятся в глоссарий или оформляются гипертекстом. На основании определения типов отношений, анализа функций и дальнейшего изучения предметной области определяются атрибуты.
- Отметим, что списки имен потенциальных сущностей и отношений автоматически формируются **по** функциональной модели. Поэтому **их** классификация и применение в информационной модели являются последовательным раскрытием их информационной структуры в контексте синтаксиса функциональной модели и семантики предметной области.
- Построенная по указанным выше правилам информационная модель будет являться адекватным отображением информационной структуры сущностей и их отношений.
- При реализации информационной модели может возникнуть необходимость приведения ее к какой-либо нормализованной форме: 1-й или 2-й, или 3-й нормальной форме.

Фрагмент диаграммы «сущность-связь» учета сотрудников



Стандарт IDEF3

- **Предназначение IDEF3**
- IDEF3 является стандартом документирования технологических процессов, происходящих на предприятии, и предоставляет инструментарий для наглядного исследования и моделирования их сценариев. Сценарием (Scenario) мы называем описание последовательности изменений свойств объекта, в рамках рассматриваемого процесса (например, описание последовательности этапов обработки детали в цеху и изменение её свойств после прохождения каждого этапа). Исполнение каждого сценария сопровождается соответствующим документооборотом, который состоит из двух основных потоков: документов, определяющих структуру и последовательность процесса (технологических указаний, описаний стандартов и т.д.), и документов, отображающих ход его выполнения (результатов тестов и экспертиз, отчетов о браке, и т.д.). Для эффективного управления любым процессом, необходимо иметь детальное представление об его сценарии и структуре сопутствующего документооборота. Средства документирования и моделирования IDEF3 позволяют выполнять следующие задачи.

Стандарт IDEF3

- Документировать имеющиеся данные о технологии процесса, выявленные, скажем, в процессе опроса компетентных сотрудников, ответственных за организацию рассматриваемого процесса.
- Определять и анализировать точки влияния потоков сопутствующего документооборота на сценарий технологических процессов.
- Определять ситуации, в которых требуется принятие решения, влияющего на жизненный цикл процесса, например изменение конструктивных, технологических или эксплуатационных свойств конечного продукта.
- Содействовать принятию оптимальных решений при реорганизации технологических процессов.
- Разрабатывать имитационные модели технологических процессов, по принципу "КАК БУДЕТ, ЕСЛИ..."

Два типа диаграмм в IDEF3

- Существуют два типа диаграмм в стандарте IDEF3, представляющие описание одного и того же сценария технологического процесса в разных ракурсах. Диаграммы относящиеся к первому типу называются диаграммами Описания Последовательности Этапов Процесса (Process Flow Description Diagrams, PFDD), а ко второму - диаграммами Состояния Объекта в и его Трансформаций Процессе (Object State Transition Network, OSTN). Предположим, требуется описать процесс окраски детали в производственном цеху на предприятии. С помощью диаграмм PFDD документируется последовательность и описание стадий обработки детали в рамках исследуемого технологического процесса.
- Диаграммы OSTN используются для иллюстрации трансформаций детали, которые происходят на каждой стадии обработки.
- На следующем примере, опишем, как графические средства IDEF3 позволяют документировать вышеуказанный производственный процесс окраски детали. В целом, этот процесс состоит непосредственно из самой окраски, производимой на специальном оборудовании и этапа контроля ее качества, который определяет, нужно ли деталь окрасить заново (в случае несоответствия стандартам и выявления брака) или отправить ее в дальнейшую обработку.

Пример диаграммы PFDD

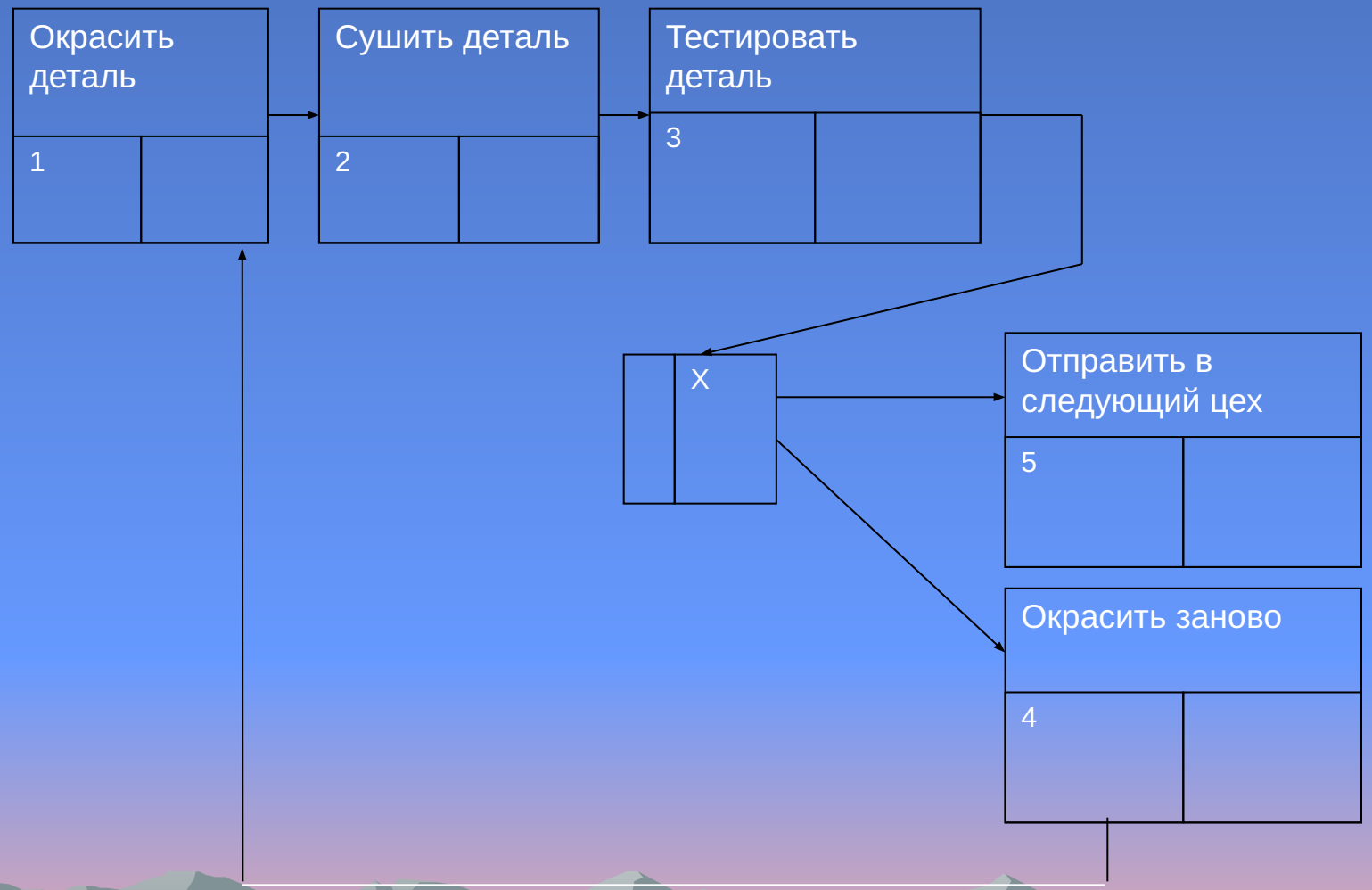


Диаграмма PFDD

- На рисунке изображена диаграмма PFDD, являющаяся графическим отображение сценария обработки детали. Прямоугольники на диаграмме PFDD называются функциональными элементами или элементами поведения (Unit of Behavior, UOB) и обозначают событие, стадию процесса или принятие решения. Каждый UOB имеет свое имя, отображаемое в глагольном наклонении и уникальный номер. Стрелки или линии являются отображением перемещения детали между UOB-блоками в ходе процесса. Линии бывают следующих видов.
- - Старшая (Precedence) - сплошная линия, связывающая UOB. Рисуется слева направо или сверху вниз.
- - Отношения (Relational Link)- пунктирная линия, использующаяся для изображения связей между UOB
- - Потоки объектов (Object Flow)- стрелка с двумя наконечниками используется для описания того факта, что объект (деталь) используется в двух или более единицах работы, например, когда объект порождается в одной работе и используется в другой.
- Объект, обозначенный J1 - называется перекрестком (Junction). Перекрестки используются для отображения логики взаимодействия стрелок (потоков) при слиянии и разветвлении или для отображения множества событий, которые могут или должны быть завершены перед началомследующей работы. Различают перекрестки для слияния (Fan-in Junction) и разветвления (Fan-out Junction) стрелок. Перекресток не может использоваться одновременно для слияния и для разветвления. При внесении перекрестка в диаграмму необходимо указать тип перекрестка.

Классификация возможных типов перекрестков

- Asynchronous AND-Все предшествующие процессы должны быть завершены.Все следующие процессы должны быть запущены.
- Synchronous AN-Все предшествующие процессы завершены одновременно. Все следующие процессы запускаются одновременно.
- Asynchronous OR-Один или несколько предшествующих процессов должны быть завершены.Один или несколькоследующих процессовдолжны быть запущены.
- Synchronous OR-Один или несколько предшествующихпроцессов.Завершаются одновременно.Один или несколькоСледующих процессовзапускаются одновременно.
- XOR (Exclusive OR)-Только один предшествующий процесс завершен.Только один следующий процесс запускается.

Диаграмма PFDD

- Все перекрестки в PFDD диаграмме нумеруются, каждый номер имеет префикс "J".
- Сценарий, отображаемый на диаграмме, можно описать в следующем виде.
- Деталь поступает в окрасочный цех, подготовленной к окраске. В процессе окраски наносится один слой эмали при высокой температуре. После этого, производится сушка детали, после которой начинается этап проверки качества нанесенного слоя. Если тест подтверждает недостаточное качество нанесенного слоя (недостаточную толщину, неоднородность и т.д.), то деталь заново пропускается через цех окраски. Если деталь успешно проходит контроль качества, то она отправляется в следующий цех для дальнейшей обработки.

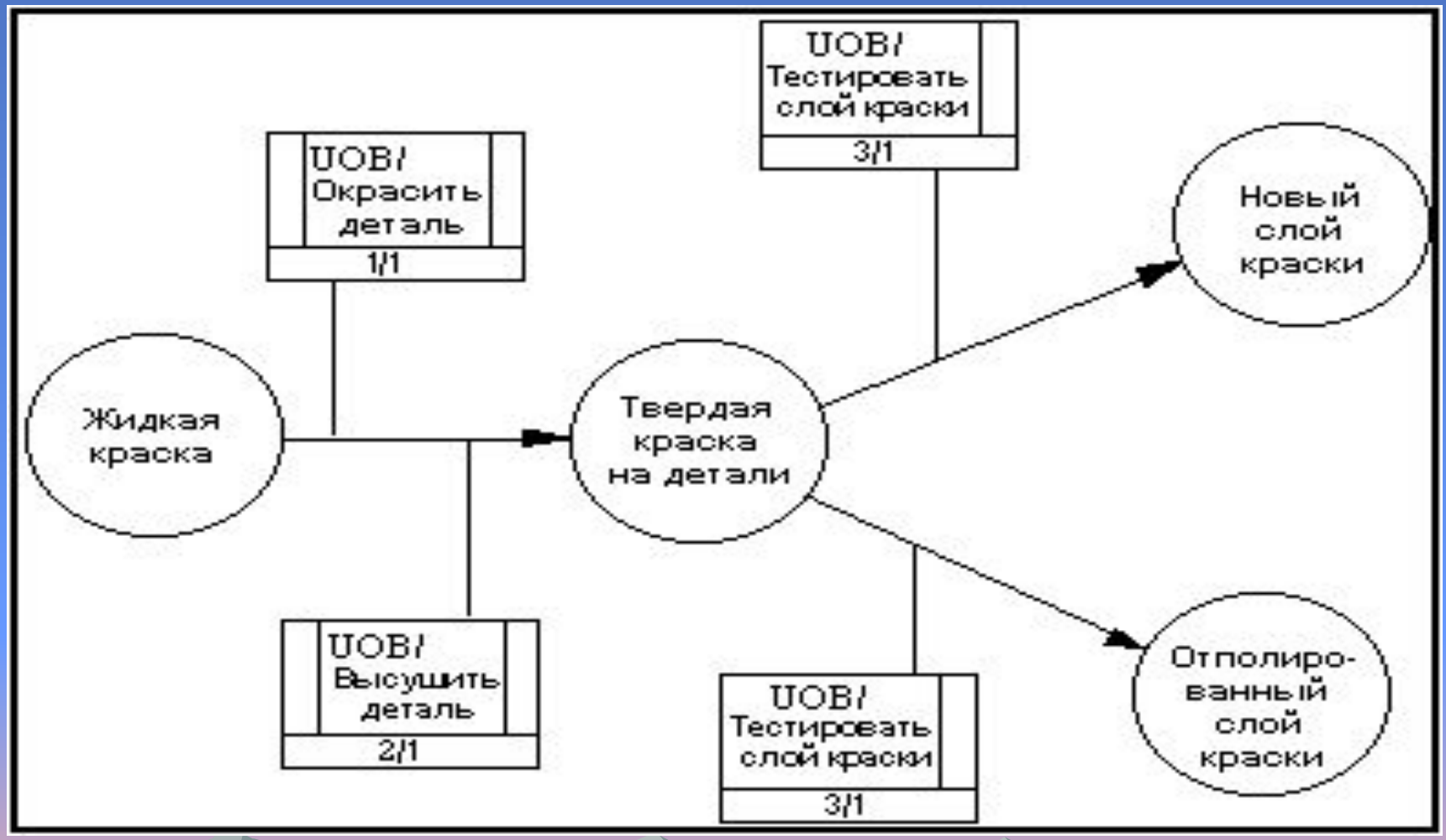
Диаграмма PFDD

- Каждый функциональный блок UOB может иметь последовательность декомпозиций, и, следовательно, может быть детализирован с любой необходимой точностью. Под декомпозицией мы понимаем представление каждого UOB с помощью отдельной IDEF3 диаграммы. Например, мы можем декомпозировать UOB "Окрасить Деталь", представив его отдельным процессом и построив для него свою PFDD диаграмму. При этом эта диаграмма будет называться дочерней, по отношению к изображенной на рисунке, а та, соответственно родительской. Номера UOB дочерних диаграмм имеют сквозную нумерацию, т.е., если родительский UOB имеет номер "1", то блоки UOB на его декомпозиции будут соответственно иметь номера "1.1", "1.2" и т.д. Применение принципа декомпозиции в IDEF3 позволяет структурировано описывать процессы с любым требуемым уровнем детализации.

Диаграмма OSTN

- Если диаграммы PFDD технологический процесс "С точки зрения наблюдателя", то другой класс диаграмм IDEF3 OSTN позволяет рассматривать тот же самый процесс "С точки зрения объекта". На рис.2 представлено отображение процесса окраски с точки зрения OSTN диаграммы. Состояния объекта (в нашем случае детали) и Изменение состояния являются ключевыми понятиями OSTN диаграммы. Состояния объекта отображаются окружностями, а их изменения направленными линиями. Каждая линия имеет ссылку на соответствующий функциональный блок UOB, в результате которого произошло отображаемое ей изменение состояния объекта.

Пример OSTN диаграммы



Стандарт IDEF5

- Исторически, понятие онтологии появилось в одной из ветвей философии, называемой метафизикой, которая изучает устройство реального мира. Основной характерной чертой онтологического анализа является, в частности, разделение реального мира на составляющие и классы объектов (at its joints) и определение их онтологий, или же совокупности фундаментальных свойств, которые определяют их изменения и поведение. Таким образом, естественная наука представляет собой типичный пример онтологического исследования. Например, атомная физика классифицирует и изучает свойства наиболее фундаментальных объектов реального мира, таких как элементарные частицы, а биология, в свою очередь, описывает характерные свойства живых организмов, населяющих планету.

Стандарт IDEF5

- Однако фундаментальные и естественные науки не обладают достаточным инструментарием для того, чтобы полностью охватить область, представляющую интерес для онтологического исследования. Например, существует большое количество сложных формаций или систем, созданных и поддерживаемых человеком, таких как производственные фабрики, военные базы, коммерческие предприятия и т.д. Эти формации представляют собой совокупность взаимосвязанных между собой объектов и процессов, в которых эти объекты тем или иным образом участвуют. Онтологическое исследование подобных сложных систем позволяет накопить ценную информацию об их работе, результаты анализа которой будут иметь решающее мнение при проведении процесса реорганизации существующих и построении новых систем.
- Методология IDEF5 обеспечивает наглядное представление данных, полученных в результате обработки онтологических запросов в простой естественной графической форме.

Основные принципы онтологического анализа

- Онтологический анализ обычно начинается с составления словаря терминов, который используется при обсуждении и исследовании характеристик объектов и процессов, составляющих рассматриваемую систему, а также создания системы точных определений этих терминов. Кроме того, документируются основные логические взаимосвязи между соответствующими введенным терминам понятиями. В дальнейшем мы не будем делать различия между понятиями и терминами. Результатом этого анализа является онтология системы, или же совокупность словаря терминов, точных их определений взаимосвязей между ними.
- Таким образом, онтология включает в себя совокупность терминов и правила, согласно которым эти термины могут быть скомбинированы для построения достоверных утверждений о состоянии рассматриваемой системы в некоторый момент времени. Кроме того, на основе этих утверждений, могут быть сделаны соответствующие выводы, позволяющие вносить изменения в систему, для повышения эффективности её функционирования.

Основные принципы онтологического анализа

- В любой системе существует две основные категории предметов восприятия, такие как сами объекты, составляющие систему (физические и интеллектуальные) и взаимосвязи между этими объектами, характеризующие состояние системы. В терминах онтологии, понятие взаимосвязи, однозначно описывает или, другими словами, является точным дескриптором зависимости между объектами системы в реальном мире, а термины - являются, соответственно, точными дескрипторами самих реальных объектов.
- При построении онтологии, в первую очередь происходит создание списка или базы данных дескрипторов и с помощью них, если их набор достаточен, создается модель системы. Таким образом, на начальном этапе должны быть выполнены следующие задачи.
- 1) Создание и документирования словаря терминов
- 2) Описание правил и ограничений, согласно которым на базе введенной терминологии формируются достоверные утверждения,
- описывающие состояние системы.
- 3) Построение модели, которая на основе существующих утверждений, позволяет формировать необходимые дополнительные утверждения.

Основные принципы онтологического анализа

- Что мы имеем в виду под необходимыми дополнительными утверждениями? Дело в том, что при рассмотрении каждой системы существует огромное количество утверждений, достоверно отображающих ее состояние в различных разрезах, а построенная онтологическим способом модель должна выбирать из них наиболее полезные для эффективного рассмотрения в том или ином контексте. Дополнительно, эта модель помогает описывать поведение объектов и соответствующее изменение взаимосвязей между ними, или, другими словами, поведение системы.
- Таким образом, онтология представляет собой некий словарь данных, включающий в себя и терминологию и модель поведения системы.

Концепции IDEF5

- Процесс построения онтологии, согласно методологии IDEF5 состоит из пяти основных действий:
- 1) Изучение и систематизирование начальных условий. Это действие устанавливает основные цели и контексты проекта разработки онтологии, а также распределяет роли между членами проекта
- 2) Сбор и накапливание данных. На этом этапе происходит сбор и накапливание необходимых начальных данных для построения онтологии
- 3) Анализ данных. Эта стадия заключается в анализе и группировке собранных данных и предназначена для облегчения построения терминологии.
- 4) Начальное развитие онтологии. На этом этапе формируется предварительная онтология, на основе отобранных данных.
- 5) Уточнение и утверждение онтологии - Заключительная стадия процесса.

Язык описания онтологий в IDEF5

- Для поддержания процесса построения онтологий в IDEF5 существуют специальные онтологические языки: схематический язык (Schematic Language-SL) и язык доработок и уточнений (Elaboration Language-EL). SL является наглядным графическим языком, специально предназначенным для изложения компетентными специалистами в рассматриваемой области системы основных данных в форме онтологической информации (См. рисунок 1). Этот несложный язык позволяет естественным образом представлять основную информацию в начальном развитии онтологии и дополнять существующие онтологии новыми данными. EL представляет собой структурированный текстовой язык, который позволяет детально характеризовать элементы онтологии.
- Язык SL позволяет строить разнообразные типы диаграмм и схем в IDEF5. Основная цель всех этих диаграмм - наглядно и визуально представлять основную онтологическую информацию.
- Несмотря на кажущееся сходство, семантика и обозначения схематичного языка SL существенно отличается от семантики и обозначений других графических языков. Дело в том, что часть элементов графической схемы SL может быть изменен или вовсе не приниматься во внимание языком EL. Причина этого состоит в том, что основной целью применения SL является создание лишь вспомогательной структурированной конструкции онтологии, и графические элементы SL не несут достаточной информации для полного представления и анализа системы, тем самым они не предназначены для сохранения при конечном этапе проекта. Тщательный анализ, обеспечение полноты представления структуры данных, полученных в результате онтологического исследования, являются задачей применения языка EL.

Виды диаграмм IDEF5

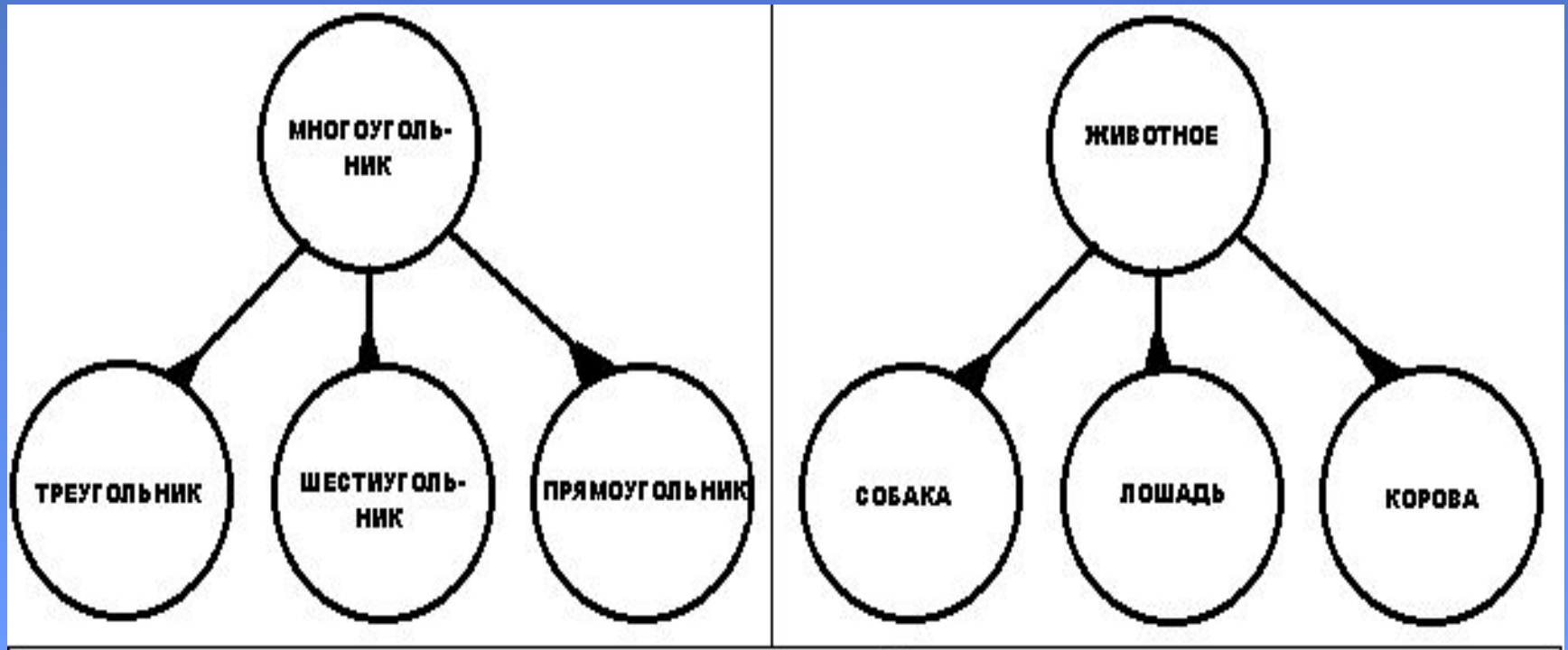


РИСУНОК 2. ВИДЫ ДИАГРАММ IDEF5: ДИАГРАММА СТРОГОЙ КЛАССИФИКАЦИИ (СЛЕВА) И ДИАГРАММА ЕСТЕСТВЕННОЙ КЛАССИФИКАЦИИ (СПРАВА)

Виды схем и диаграмм IDEF5

- Как правило, наиболее важные и заметные зависимости между объектами всегда являются преобладающими, когда конкретные люди высказывают свои знания и мнения, касающиеся той или иной системы. Подобные взаимосвязи явным образом описываются языками IDEF5.
- Всего существует четыре основных вида схем, которые наглядно используются для накопления информации об онтологии в достаточно прозрачной графической форме.
- 1. Диаграмма классификации. Диаграмма классификации обеспечивает механизм для логической систематизации знаний, накопленных при изучении системы. Существует два типа таких диаграмм: Диаграмма строгой классификации (Description Subsumption - DS) и диаграмма естественной или видовой классификации (Natural Kind Classification - NKC). Основное отличие диаграммы DS заключается в том, что определяющие свойства классов высшего и всех последующих уровней являются необходимым и достаточным признаком принадлежности объекта к тому или иному классу. На рисунке 2 приведен пример такой диаграммы, построенной на основе тривиальной возможности классификации многоугольников по количеству углов. Из геометрии известно точное математическое определение многоугольника, суть определяющих свойств родительского класса. Определяющим свойством каждого дочернего класса дополнительно является количество углов в многоугольнике. Очевидно, зная это определяющее свойство для любого многоугольника, можно однозначно отнести его к тому или иному дочернему классу. С помощью диаграмм DS, как правило, классифицируются логические объекты.

Виды схем и диаграмм IDEF5

- Диаграммы естественной классификации или же диаграммы NKC, наоборот, не предполагают того, что свойства класса являются необходимым и достаточным признаком для принадлежности к ним тех или иных объектов. В этом виде диаграмм определение свойств класса является более общим. Пример такой диаграммы также приведен на рис.2.
- 2. Композиционная схема. Композиционные схемы (Composition Schematics) являются механизмом графического представления состава классов онтологии и фактически представляют собой инструменты онтологического исследования по принципу "Что из чего состоит". В частности, композиционные схемы позволяют наглядно отображать состав объектов, относящихся к тому или иному классу. На рисунке 3 изображена композиционная схема шариковой ручки, относящейся к классу шариковых автоматических ручек. В данном случае шариковая ручка является системой, к которой мы применяем методы онтологического исследования. С помощью композиционной схемы мы наглядно документируем, что авторучка состоит из нижней и верхней трубки, нижняя трубка в свою очередь включает в себя кнопку и фиксирующий механизм, а верхняя трубка включает в себя стержень и пружину.

Виды схем и диаграмм IDEF5

- Схема взаимосвязей. Схемы взаимосвязей (Relation Schematics) позволяют разработчикам визуализировать и изучать взаимосвязи между различными классами объектов в системе. В некоторых случаях схемы взаимосвязей используются для отображения зависимостей между самими же классовыми взаимосвязями. Мотивацией для развития подобной возможности послужило то тривиальное правило, что все вновь разработанные концепции всегда базируются на уже существующих и изученных. Это тесно согласуется с теорией Новака и Гоуэна (Novak & Gowin, 1984), суть которой в том, что изучение любой системы часто происходит от частного к общему, то есть, происходит изыскание и исследование новой частной информации, влияющее на конечные характеристики более общей концепции, к которой эта информация имела прямое отношение. Исходя из этой гипотезы, естественным образом изучения новой или плохо понимаемой взаимосвязи является соотнесение ее с достаточно изученной взаимосвязью, для исследования характеристик их сосуществования.

Виды схем и диаграмм IDEF5

- 4.Диаграмма состояния объекта. Диаграмма состояния объекта (Object State Schemantic) позволяет документировать тот или иной процесс с точки зрения изменения состояния объекта. В происходящих процессах могут произойти два типа изменения объекта: объект может поменять свое состояние или класс. Между этими двумя видами изменений по сути не существует принципиальной разницы: объекты, относящиеся к определенному классу K в начальном состоянии в течение процесса могут просто перейти к его дочернему или просто родственному классу. Например, полученная в процессе нагревания теплая вода, уже относится не к классу ВОДА, а к его дочернему классу ТЕПЛАЯ ВОДА. Однако при формальном описании процесса, во избежание путаницы, целесообразно разделять оба вида изменений, и для такого разделения используется обозначения следующего вида: "класс: состояние". Например теплая вода будетописываться следующим образом: "вода:теплая", холодная - "вода:холодная" и так далее. Таким образом, диаграммы состояния в IDEF5 наглядно представляют изменения состояния или класса объекта в течение всего хода процесса. Пример такой диаграммы приведен на рис.4

Виды схем и диаграмм IDEF5

РИСУНОК 4. ПРИМЕР ДИАГРАММЫ СОСТОЯНИЯ

CASE-средства американской фирмы Computer Systems Advisers, Inc.

CASE-система SILVERRUN

- CASE-система SILVERRUN американской фирмы Computer Systems Advisers, Inc. используется для инструментального обеспечения анализа и проектирования информационных систем бизнес-класса. Она применима для поддержки любой методологии, основанной на раздельном построении функциональной и информационной моделей (диаграмм потоков данных и диаграмм "сущность-связь").
- Настройка на конкретную методологию обеспечивается выбором требуемого графического изображения символов моделей и набора правил проверки проектных спецификаций. В системе имеются готовые настройки для наиболее распространенных методологий:. Gane/Sarson, Yourdon/DeMarco, Merise, Ward/Mellor, Information Engineering. Для каждого проектного понятия имеется возможность добавления собственных описателей.
- Системные отчеты позволяют получать рабочие проектные документы с контролируемым содержанием. Функции импорта/экспорта позволяют работать с текстовой информацией изменяемого формата (определяемые пользователем разделители записей и полей), что дает возможность передавать проектные данные в большинство СУБД и электронных таблиц.

Модуль построения диаграмм потоков данных (BPM - Business Process Modeler)

- Архитектура системы SILVERRUN позволяет наращивать среду разработки по мере необходимости. Система состоит из трех модулей, каждый из которых является самостоятельным продуктом и может приобретаться и использоваться без связи с остальными модулями.
- **Модуль построения диаграмм потоков данных (BPM - Business Process Modeler)** позволяет моделировать функционирование обследуемой организации или создаваемой информационной системы. Диаграммы потоков данных являются самым распространенным средством моделирования процессов в системах бизнес-класса. В модуле SILVERRUN BPM обеспечена возможность удобной работы с моделями большой сложности: автоматическая перенумерация, работа с деревом процессов (включая визуальное "перетаскивание" ветвей), отсоединение и присоединение частей модели для коллективной разработки. Диаграммы могут изображаться в нескольких предопределенных нотациях, включая Yordon/DeMarco и Gane/Sarson. Имеется также возможность видоизменять предопределенные и создавать собственные нотации, в том числе добавлять в число изображаемых на схеме дескрипторов определенные пользователем поля.

Модуль концептуального моделирования данных (ERX - Entity Relationship Export)

- **Модуль концептуального моделирования данных (ERX - Entity Relationship Export)** обеспечивает построение концептуальных моделей данных "сущность - связь", в которых допускается соединять одной связью более двух сущностей, а также присваивать связям атрибуты. Такие возможности облегчают построение обобщенной модели данных, не привязанной к реализации. Встроенная экспертная система помогает создать корректную нормализованную модель через ответы на содержательные вопросы о взаимосвязи данных. Возможно автоматическое построение модели данных из описаний структур данных. Анализ функциональных за висимостей атрибутов дает возможность проверить соответствие модели требованиям третьей нормальной формы и обеспечить их выполнение. Проверенная модель передается в модуль RDM.

Модуль реляционного моделирования (RDM - Relational Data Modeler)

- **Модуль реляционного моделирования (RDM - Relational Data Modeler)** позволяет создавать детализированные модели "сущность-связь", предназначенные для реализации в реляционной базе данных. В этом модуле документируются все конструкции, связанные с построением базы данных: индексы, триггеры, хранимые процедуры и т.д. Гибкая изменяемая нотация и расширяемость репозитория позволяют работать по любой методологии. Возможность создавать подсхемы соответствует подходу ANSI SPARC к представлению схемы базы данных. На языке подсхем моделируются как узлы распределенной обработки, так и пользовательские представления. Этот модуль обеспечивает проектирование и полное документирование реляционных баз данных любой сложности.
- Для автоматической генерации схем баз данных отдельно поставляются мосты к наиболее известным РСУБД: DB2, Informix, Ingres, Oracle, Progress, SQL Base, SQL Server, Sybase. Все мосты обеспечивают реверсивный инжиниринг, позволяя загрузить в SILVERRUN RDM информацию из каталогов СУБД и строить модели уже имеющихся баз данных. Это дает возможность документировать и перепроектировать сложные базы данных, уже находящиеся в эксплуатации.
- Для передачи данных в средства разработки приложений имеются мосты к языкам четвертого поколения: PowerBuilder, Progress, SQLWindows, UNIFACE. Разработка приложения существенно облегчается, так как информационное содержание создаваемого рабочего места автоматически заносится в репозиторий приложения.

Менеджер репозитория рабочей группы (WRM - Workgroup Repository Manager)

- **Менеджер репозитория рабочей группы (WRM - Workgroup Repository Manager)** применяется как словарь данных, а также обеспечивает интеграцию модулей SILVERRUN в единую среду проектирования.
- Система SILVERRUN реализована на четырех платформах - Windows, Macintosh, OS/2, Solaris - с возможностью прозрачного обмена проектными данными между ними.
- Все модули графического моделирования системы SILVERRUN предоставляют удобный графический интерфейс, наборы пиктограмм, использование разного рода палитр, управление цветами (включая печать) и шрифтами. Любая графическая модель может быть разделена на подмодели для групповой работы, а результаты этой работы интегрированы в общую модель. Детальные описания моделей и их компонентов могут быть дополнены неограниченным набором определенных пользователем полей. Доступ к репозиторию осуществляется как из меню *каждого* из модулей, так и непосредственно из графической модели. Вся проектная информация выводится на печать в виде графических схем и текстовых форматируемых отчетов на любом устройстве, подключенном к используемой графической среде, или экспортируется в нужном формате в ASCII-файлы.

Система SILVERRUN

- Система SILVERRUN применяется в фирмах самого разнообразного профиля: Apple Computer, IBM, Intel, Texas Instruments, Northern Telecom, Citibank, World Bank, Montreal Exchange, American Express, American Airlines, Pepsi-Cola, Polaroid, Pizza Hut. В настоящее время используются тысячи инсталляций продуктов фирмы.
- Наличие собственного исследовательского центра и работа по консалтингу в области информационных технологий дают уверенность в постоянном соответствии продуктов фирмы CSA самым современным технологиям.

Методология создания информационных систем «Datarun»

- Высокая динамичность рынка требует от организаций быстрого развития информационно-технологической инфраструктуры. Одной из ее наиболее важных и дорогостоящих составляющих является информационная система, для реализации которой применяются современные технологии: архитектура клиент/сервер, распределенные базы данных, сложные сети коммуникаций, развитые интерфейсы пользователя. Все это ставит перед разработчиком проблему выбора инструментальных средств и технологий для ведения проекта.
- Создание сложной информационной системы невозможно без единого интегрированного подхода к процессу разработки. Такой подход часто оформляется в виде коммерчески доступной методологии проектирования. Методология служит двум целям:
- обеспечивает концептуальную основу для всего процесса разработки;
- предоставляет технологию руководства проектом.
- Многие методологии применялись в течение ряда лет с разной степенью успеха. Часто разнообразие используемых в них моделей приводит к получению огромного количества документации, не сосредоточенной на результатах. Множественные перекрывающиеся модели процессов и данных создают избыточность, которая преподносится как перекрестный контроль.

Методология создания информационных систем «Datarun»

- DATARUN - уникальная концепция в ряду методов. Эта методология гарантирует, что на каждой стадии выполняется только существенная для целей проекта работа, облегчающая быстрое создание приложений. Повторения и избыточность в спецификациях исключаются, создается управляемая, основанная на моделях форма итеративной разработки. Исходные версии объектов доступны для непосредственного использования на следующих фазах проектного цикла. Создаваемая информационная система описывается рядом последовательных моделей, каждая из которых является развитием предыдущей и наследует правила и данные, определенные в более ранних моделях. Наследование свойств позволяет многократно использовать различные спецификации на всех уровнях прикладного проекта.
- Методология DATARUN ведет заказчика и разработчика информационной системы по всем этапам жизненного цикла проекта, от стадии первоначальной экономической оценки затрат на проект до выхода реального приложения. Она позволяет координировать и контролировать работу всех групп лиц, занятых в работе над проектом.

Методология создания информационных систем «Datarun»

- Методология DATARUN обеспечена средствами автоматизированной поддержки:
- Для управления проектной деятельностью имеется система Software Engineering Companion, позволяющая детально расписывать ведение проекта, распределять проектные роли среди исполнителей, контролировать выполнение заданий.
- Детальное изложение техник моделирования данных и бизнес-функций, проектирования баз данных, создания приложений содержится в гипертекстовой системе DATARUN Software Engineering Guidelines.
- Автоматизация проведения проектных работ обеспечивается CASE-системой SILVERRUN.
- Предоставляемая этими средствами среда проектирования дает возможность руководителю проекта контролировать выполнение работ. Каждый участник проекта, подключившись к системе, может уточнить содержание и сроки выполнения порученной ему работы, детально изучить технику ее выполнения в гипертексте по технологиям, и вызвать инструмент (модуль SILVERRUN) для реального выполнения работы.
- Такой автоматизированный комплекс поддержки выполнения проектов, основанный на современной методологии проектирования и эффективном CASE-средстве, создает все необходимые условия для быстрого создания сложных информационных систем с высоким качеством. Основные этапы методологии datarun

Построение бизнес-модели предметной области

- Строятся функциональная (диаграмма потоков данных(DFD)) и информационная (диагармма «сущность-связь» (ER)) модели предметной области. При построении функциональной модели выявляются первичные структуры данных, которые преобразуются в сущности ER-модели. Результат: **Модель бизнес-процессов,** содержащая **Первичные структуры данных,** и **Концептуальная модель данных.**

Построение архитектуры информационной системы

- Принимается решение, из каких приложений (подсистем) будет состоять система. Анализируются существующие системы. Архитектура системы документируется в виде DFD, где функции представляют компоненты приложений с указанием используемой информации (путем ссылки на сущности и связи ER-модели). ER-модель также делится на группы сущностей, обрабатываемых приложениями. Результат: **Архитектура информационной системы.**

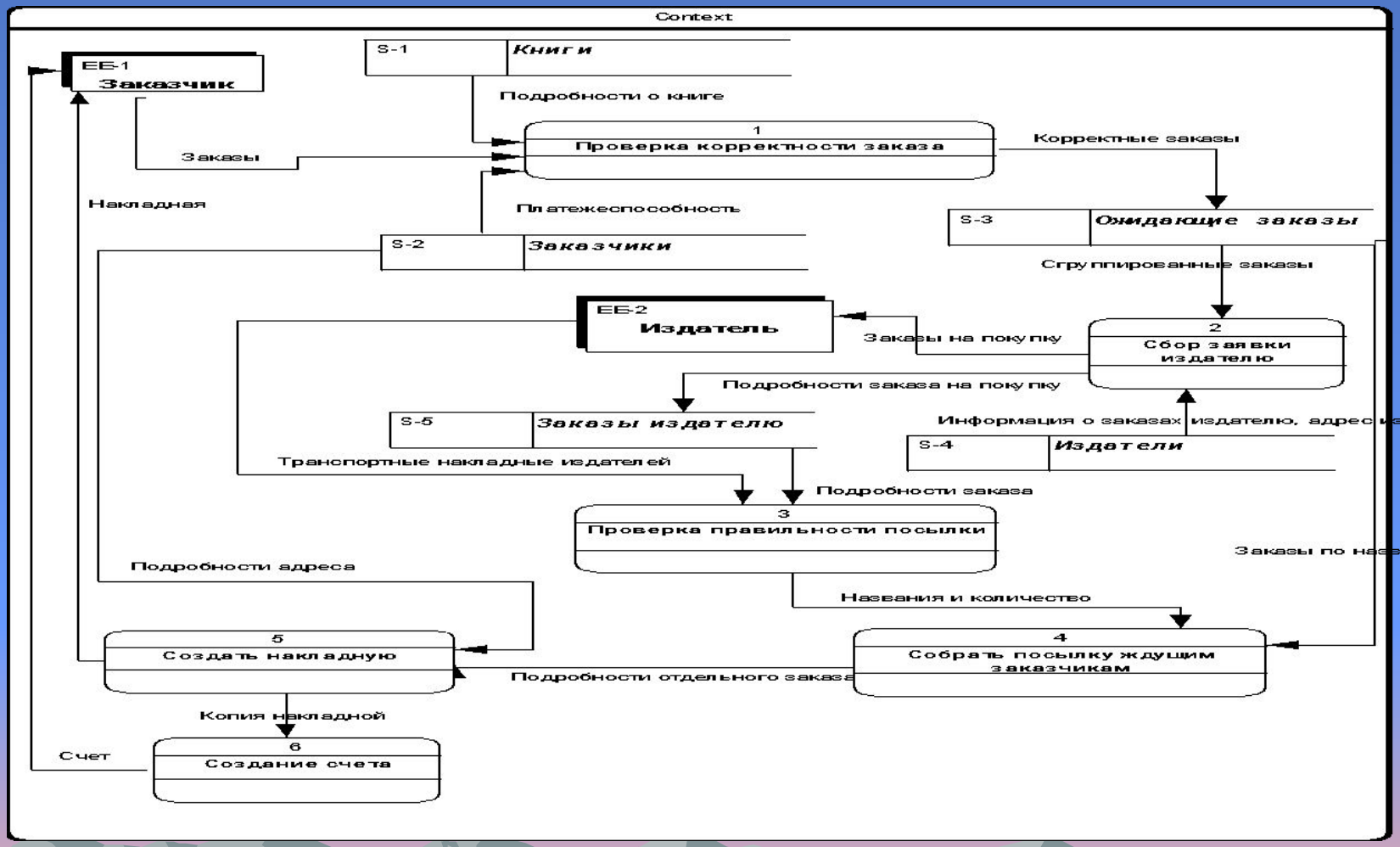
Проектирование приложений (подсистем)

- На основе концептуальной ER модели строится реляционная модель данных. Части ER-модели, соответствующие различным приложениям, оформляются как подсхемы базы данных.
- Для каждого приложения создается (возможно, разными группами разработчиков) детальный проект. Строится модель системных процессов (программных функций) и подсхемы базы данных для каждой функции (модель данных интерфейса / спецификация интерфейса). Поскольку все приложения работают с подсхемами одной базы данных, обеспечивается их совместная работа.
- Результат: **Реляционная модель данных.** Для каждого приложения **- Модель данных приложения. Модель системных процессов.** Для каждого интерфейса в приложении **- Модель данных интерфейса. Модель спецификации интерфейса, Модель представления интерфейса.**

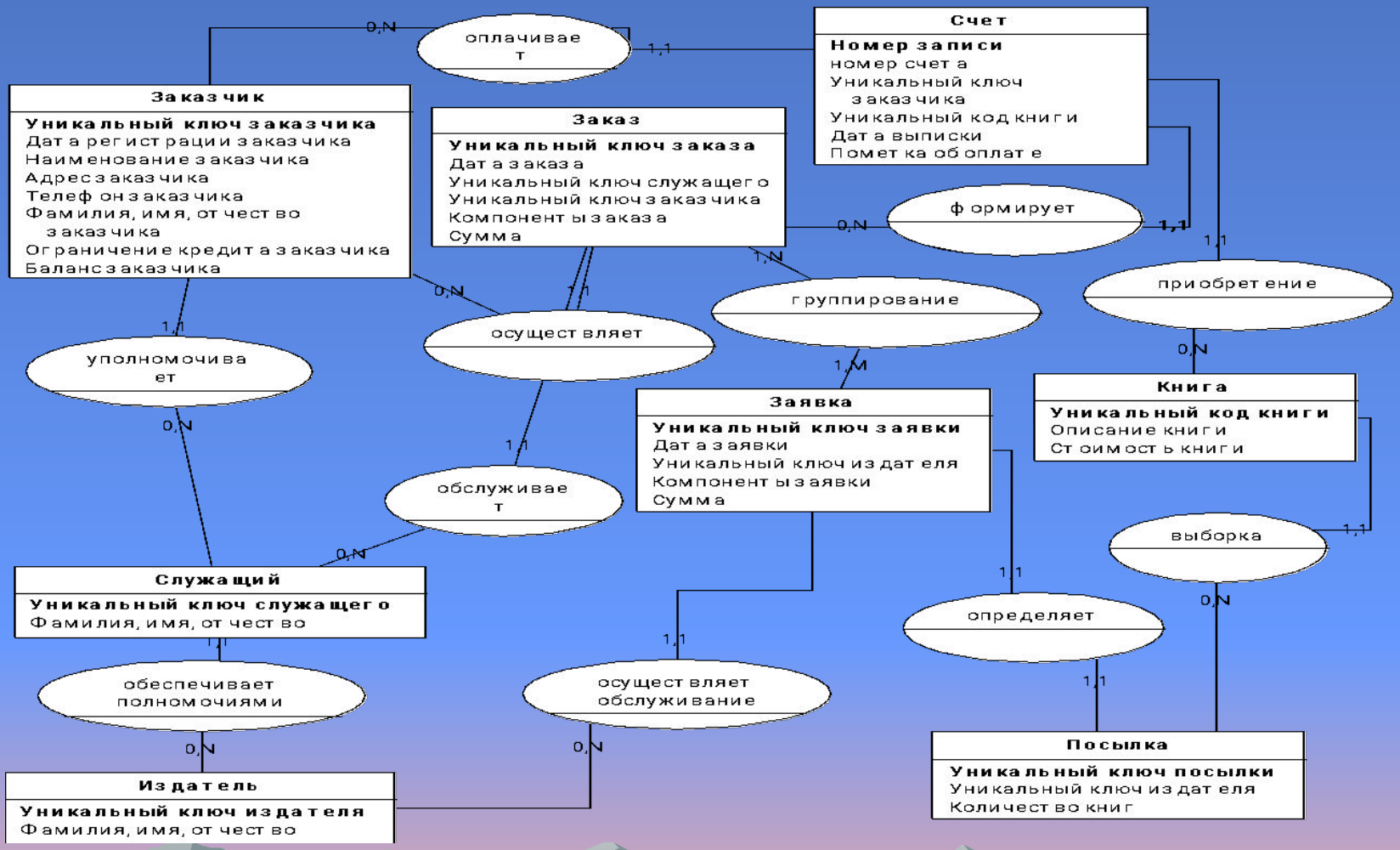
Создание приложений

- Из модели базы данных генерируется код для ее создания на сервере. Программируются системные процессы. При этом возможно разнесение процессов по узлам распределенной системы (часть процессов реализуется как хранимые процедуры на сервере, часть - как сервисы монитора транзакций, часть - как программы клиентской части). Интерфейс приложений (обычно составляющий до 70-80% всей системы) может быть быстро создан перенесением соответствующей ему подсхемы базы данных в среду языка 4-го поколения).
- Созданные приложения объединяются в единую среду и тестируются на совместимость. Поскольку все приложения строились на основе общей глобальной модели данных, достигается высокая степень интеграции.

Фрагмент диаграммы бизнес-процесса, созданный в CASE-средстве SILVERRUN-BPM



Фрагмент диаграммы «Сущность-связь», выполненный в SILVERRUN-ERX



Методология RAD

- Одним из возможных подходов к разработке ПО в рамках спиральной модели ЖЦ является получившая в последнее время широкое распространение методология быстрой разработки приложений RAD (Rapid Application Development). Под этим термином обычно понимается процесс разработки ПО, содержащий 3 элемента:
- небольшую команду программистов (от 2 до 10 человек);
- короткий, но тщательно проработанный производственный график (от 2 до 6 мес.);
- повторяющийся цикл, при котором разработчики, по мере того, как приложение начинает обретать форму, запрашивают и реализуют в продукте требования, полученные через взаимодействие с заказчиком.

Методология RAD

- Команда разработчиков должна представлять из себя группу профессионалов, имеющих опыт в анализе, проектировании, генерации кода и тестировании ПО с использованием CASE-средств. Члены коллектива должны также уметь трансформировать в рабочие прототипы предложения конечных пользователей.
- Жизненный цикл ПО по методологии RAD состоит из четырех фаз:
- фаза анализа и планирования требований;
- фаза проектирования;
- фаза построения;
- фаза внедрения.

Методология RAD

- На фазе анализа и планирования требований пользователи системы определяют функции, которые она должна выполнять, выделяют наиболее приоритетные из них, требующие проработки в первую очередь, описывают информационные потребности. Определение требований выполняется в основном силами пользователей под руководством специалистов-разработчиков. Ограничивается масштаб проекта, определяются временные рамки для каждой из последующих фаз. Кроме того, определяется сама возможность реализации данного проекта в установленных рамках финансирования, на данных аппаратных средствах и т.п. Результатом данной фазы должны быть список и приоритетность функций будущей ИС, предварительные функциональные и информационные модели ИС.
- На фазе проектирования часть пользователей принимает участие в техническом проектировании системы под руководством специалистов-разработчиков. CASE-средства используются для быстрого получения работающих прототипов приложений. Пользователи, непосредственно взаимодействуя с ними, уточняют и дополняют требования к системе, которые не были выявлены на предыдущей фазе.

Методология RAD

- Более подробно рассматриваются процессы системы. Анализируется и, при необходимости, корректируется функциональная модель. Каждый процесс рассматривается детально. При необходимости для каждого элементарного процесса создается частичный прототип: экран, диалог, отчет, устраняющий неясности или неоднозначности. Определяются требования разграничения доступа к данным. На этой же фазе происходит определение набора необходимой документации.
- После детального определения состава процессов оценивается количество функциональных элементов разрабатываемой системы и принимается решение о разделении ИС на подсистемы, поддающиеся реализации одной командой разработчиков за приемлемое для RAD-проектов время - порядка 60 - 90 дней. С использованием CASE-средств проект распределяется между различными командами (делится функциональная модель).

Методология RAD

- Результатом данной фазы должны быть:
- общая информационная модель системы;
- функциональные модели системы в целом и подсистем, реализуемых отдельными командами разработчиков;
- точно определенные с помощью CASE-средства интерфейсы между автономно разрабатываемыми подсистемами;
- построенные прототипы экранов, отчетов, диалогов.

Методология RAD

- Все модели и прототипы должны быть получены с применением тех CASE-средств, которые будут использоваться в дальнейшем при построении системы. Данное требование вызвано тем, что в традиционном подходе при передаче информации о проекте с этапа на этап может произойти фактически неконтролируемое искажение данных. Применение единой среды хранения информации о проекте позволяет избежать этой опасности.
- В отличие от традиционного подхода, при котором использовались специфические средства прототипирования, не предназначенные для построения реальных приложений, а прототипы выбрасывались после того, как выполняли задачу устранения неясностей в проекте, в подходе RAD каждый прототип развивается в часть будущей системы. Таким образом, на следующую фазу передается более полная и полезная информация.

Методология RAD

- На фазе построения выполняется непосредственно сама быстрая разработка приложения. На данной фазе разработчики производят итеративное построение реальной системы на основе полученных в предыдущей фазе моделей, а также требований нефункционального характера. Программный код частично формируется при помощи автоматических генераторов, получающих информацию непосредственно из репозитория CASE-средств. Конечные пользователи на этой фазе оценивают получаемые результаты и вносят коррективы, если в процессе разработки система перестает удовлетворять определенным ранее требованиям. Тестирование системы осуществляется непосредственно в процессе разработки.
- После окончания работ каждой отдельной команды разработчиков производится постепенная интеграция данной части системы с остальными, формируется полный программный код, выполняется тестирование совместной работы данной части приложения с остальными, а затем тестирование системы в целом.

Методология RAD

- Завершается физическое проектирование системы:
- определяется необходимость распределения данных;
- производится анализ использования данных;
- производится физическое проектирование базы данных;
- определяются требования к аппаратным ресурсам;
- определяются способы увеличения производительности;
- завершается разработка документации проекта.

Методология RAD

- Результатом фазы является готовая система, удовлетворяющая всем согласованным требованиям.
- На фазе внедрения производится обучение пользователей, организационные изменения и параллельно с внедрением новой системы осуществляется работа с существующей системой (до полного внедрения новой). Так как фаза построения достаточно непродолжительна, планирование и подготовка к внедрению должны начинаться заранее, как правило, на этапе проектирования системы. Приведенная схема разработки ИС не является абсолютной. Возможны различные варианты, зависящие, например, от начальных условий, в которых ведется разработка: разрабатывается совершенно новая система; уже было проведено обследование предприятия и существует модель его деятельности; на предприятии уже существует некоторая ИС, которая может быть использована в качестве начального прототипа или должна быть интегрирована с разрабатываемой.

Методология RAD

- Следует, однако, отметить, что методология RAD, как и любая другая, не может претендовать на универсальность, она хороша в первую очередь для относительно небольших проектов, разрабатываемых для конкретного заказчика. Если же разрабатывается типовая система, которая не является законченным продуктом, а представляет собой комплекс типовых компонент, централизованно сопровождаемых, адаптируемых к программно-техническим платформам, СУБД, средствам телекоммуникации, организационно-экономическим особенностям объектов внедрения и интегрируемых с существующими разработками, на первый план выступают такие показатели проекта, как управляемость и качество, которые могут войти в противоречие с простотой и скоростью разработки. Для таких проектов необходимы высокий уровень планирования и жесткая дисциплина проектирования, строгое следование заранее разработанным протоколам и интерфейсам, что снижает скорость разработки.

Методология RAD

- Методология RAD неприменима для построения сложных расчетных программ, операционных систем или программ управления космическими кораблями, т.е. программ, требующих написания большого объема (сотни тысяч строк) уникального кода.
- Не подходят для разработки по методологии RAD приложения, в которых отсутствует ярко выраженная интерфейсная часть, наглядно определяющая логику работы системы (например, приложения реального времени) и приложения, от которых зависит безопасность людей (например, управление самолетом или атомной электростанцией), так как итеративный подход предполагает, что первые несколько версий наверняка не будут полностью работоспособны, что в данном случае исключается.

Методология RAD

- Оценка размера приложений производится на основе так называемых функциональных элементов (экраны, сообщения, отчеты, файлы и т.п.) Подобная метрика не зависит от языка программирования, на котором ведется разработка. Размер приложения, которое может быть выполнено по методологии RAD, для хорошо отлаженной среды разработки ИС с максимальным повторным использованием программных компонентов, определяется следующим образом:
- < 1000 функциональных элементов один человек1000-4000 функциональных элементов одна команда разработчиков> 4000 функциональных элементов4000 функциональных элементов на одну команду разработчиков

Методология RAD

- В качестве итога перечислим основные принципы методологии RAD:
- разработка приложений итерациями;
- необязательность полного завершения работ на каждом из этапов жизненного цикла;
- обязательное вовлечение пользователей в процесс разработки ИС;
- необходимое применение CASE-средств, обеспечивающих целостность проекта;
- применение средств управления конфигурацией, облегчающих внесение изменений в проект и сопровождение готовой системы;
- необходимое использование генераторов кода;
- использование прототипирования, позволяющее полнее выяснить и удовлетворить потребности конечного пользователя;
- тестирование и развитие проекта, осуществляемые одновременно с разработкой;
- ведение разработки немногочисленной хорошо управляемой командой профессионалов;
- грамотное руководство разработкой системы, четкое планирование и контроль выполнения работ.

Стратегическое планирование как начальный этап анализа требований в ЖЦ ИС

- Каждую цель, которую необходимо достигнуть, представляем в виде И/ИЛИ-графа «цель-подцель», в котором она является корнем.
- Примечание. В случае необходимости достижения нескольких целей осуществляется добавление родительской вершины для достигаемых целей в качестве корня дерева.
- Вершинам назначаются веса, которым соответствуют коэффициенты значимости целей (или степени важности целей) с точки зрения эксперта или группы экспертов и ЛПР. Цели могут взаимодействовать (коррелировать) между собой. Для отображения взаимодействия целей введены следующие типы направленных дуг, используемых для задания вспомогательных отношений между вершинами:
- R1 – отношение “кооперирования” целей;
- R2 – отношение “взаимного препятствования” целей достижению друг друга;
- R3 – отношение “способствования достижения цели” достижению другой цели;
- R4 – отношение “препятствования достижения цели” достижению другой цели.

Стратегическое планирование как начальный этап анализа требований в ЖЦ ИС

- Базисные элементы и отношения между ними, используемые в диаграммном методе, положенном в основу построения блока целеполагания ИС, представлены на рис. 2.1 и 2.2. Дуги нагружены «весами», которые соответствуют силе влияния целей друг на друга с точки зрения эксперта или группы экспертов. В диаграммном методе для обозначения «весов» используются следующие обозначения:
- vi – вес вершины i (коэффициент значимости вершины Ці);
- R1ij – коэффициент силы кооперирования целей Ці и Цj;
- R2ij – коэффициент силы рассогласования целей Ці и Цj;
- R3ij – коэффициент силы способствования достижения цели Ці достижению цели Цj;
- R4ij – коэффициент силы препятствования цели Ці достижению цели Цj.

Базисные элементы

цель

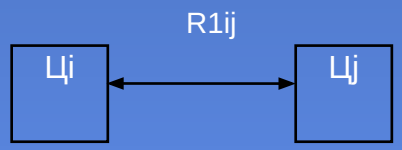
ограничение

условие достижения цели

Идеальный ресурс

Реальный ресурс

Вспомогательные отношения между базисными элементами



Отношение кооперирования целей

Отношение рассогласования целей

Отношение способствования достижения цели Цi достижению цели Цj

Отношение препятствования цели Цi достижению цели Цj

Стратегическое планирование как начальный этап анализа требований в ЖЦ ИС

- Вспомогательные отношения не устанавливаются между целью и ее подцелями, поскольку подцели используются для описания альтернативных путей достижения цели. Условием является логическая формула, используемая экспертом для задания условий достижения целей, соответствующих вершинам графа. Ограничением является логическая формула, определяющая соотношение между идеальными и реальными ресурсами. При разрешимости заключительной вершины осуществляется изменение реальных ресурсов на значение идеальных ресурсов, соответствующих данной вершине. (Если IR>0, то RR уменьшаются (RR=RR-IR) и, если IR<0, то RR увеличиваются (RR=RR+IR), где RR-значение реального ресурса, например, RR.люди; IR- значение идеального ресурса, которые задаются для концевых вершин).

Стратегическое планирование как начальный этап анализа требований в ЖЦ ИС

- В качестве математической модели используется И/ИЛИ-граф, модифицированный в соответствии с вышеописанными требованиями проблемной среды. Таким образом, вершине И/ИЛИ-графа соответствует цель, ребру – отношение «цель-подцель», также введены 4 типа направленных дуг, которым соответствуют отношения. Для вершин и направленных дуг определяются «веса». Для каждой вершины, если необходимо, задаются условия, ограничения, накладываемые на ее достижение, также определяются значения идеальных ресурсов, которые необходимо затратить на достижение цели с точки зрения эксперта или группы экспертов, и значения реальных ресурсов, имеющихся для ее достижения. Перечисленные выше параметры задаются экспертом или группой экспертов и используются в эвристической функции, используемой для нахождения наилучшего пути достижения поставленной цели.

Стратегическое планирование как начальный этап анализа требований в ЖЦ ИС

- Выбор наилучшего пути достижения поставленной цели основан на эвристическом поиске в И/ИЛИ-графе. Приведем определения разрешимой и неразрешимой вершин, являющихся модификацией определений, которые используются в алгоритме поиска наилучшего пути достижения цели.
- **Определение разрешимых вершин**
- Заключительные вершины, соответствующие элементарным целям, разрешимы тогда и только тогда, когда указанные ограничения и условия выполнены.
- Вершина, не являющаяся заключительной и имеющая дочерние вершины типа «ИЛИ», разрешима тогда и только тогда, когда выполняется условие, ей соответствующее, и разрешима, по крайней мере, одна из дочерних вершин.
- Вершина, не являющая заключительной и имеющая дочерние вершины типа «И», разрешима тогда и только тогда, когда условие выполнено и разрешимы все ее дочерние вершины.

Стратегическое планирование как начальный этап анализа требований в ЖЦ ИС

- **Определение неразрешимых вершин**
- Заключительные вершины неразрешимы тогда и только тогда, когда указанные ограничения и/или условия не выполнены.
- Вершина, не являющаяся заключительной и имеющая дочерние вершины типа «ИЛИ», неразрешима тогда и только тогда, когда соответствующее ей условие не выполнено и/или неразрешимы все ее дочерние вершины.
- Вершина, не являющая заключительной и имеющая дочерние вершины типа «И», неразрешима тогда и только тогда, когда условие не выполнено и/или неразрешима, по крайней мере, одна из ее дочерних вершин.
- В основе алгоритма поиска в И/ИЛИ-графе лежит эвристическая функция, которая позволяет сократить пространство поиска за счет упорядочивания процесса раскрытия вершин и не рассмотрения неперспективных путей достижения цели. Определим эвристическую функцию.

Стратегическое планирование как начальный этап анализа требований в ЖЦ ИС

- Введем пересчитанный коэффициент значимости wi вершины (цели) i, который является аргументом эвристической функции.
- , где

$$w_i = \begin{cases} v_i, если\ имеет\ место\ (1) \\ v_i * (\frac{\sum\limits_k R1ik}{\sum\limits_k R2ik} + \frac{\sum\limits_k R3ik}{\sum\limits_k R4ik}), если\ имеет\ место\ (2) \end{cases}$$

(1) – для вершин типа «ИЛИ», а также для вершин типа «И», если между вершинами типа «И» не определены отношения;

(2) – для вершин типа «И», между которыми определено хотя бы одно отношение;

- vi∈]0,10]; R1ik, R2ik, R3ik, R4ik ∈]0, 10], i=1..M, где M -число вершин в И/ИЛИ-графе.

Стратегическое планирование как начальный этап анализа требований в ЖЦ ИС

- Зададим эвристическую функцию h(i), которая минимизируется при нахождении наилучшего пути достижения корневой цели: h(i)⇒min.
- h(i)=

$$\left\{\begin{array}{l} 1/w_i, если\ имеет\ место\ (1) \\ 1/w_i + \min_j[h(i_j)], j = 1..k, если\ имеет\ место\ (2) \\ 1/w_i + \sum_j h(i_j), j = 1..k, если\ имеет\ место\ (3) \end{array}\right\}$$

- , где

  (1) – для заключительных вершин;

  (2) – для вершин, имеющих дочерние вершины i1,i2,…,ik типа «ИЛИ»;

  (3) – для вершин, имеющих дочерние вершины i1,i2,…,ik типа «И», образующие группу вершин типа «И».

Библиографический список

- *«Информационная инженерия»*
- Modern Software Engineering. Foundation and Current Perspectives.- Edited by Peter A.Ng., Raymond T. Yeh. – New York: VAN NOSTRAND REINHOLD, 1990. – 581 p.
- Cooling J.E. Software Design for Real-time Systems. Raymond T CHAPMAN AND HALL (University and Professional Division), 1991. – 505 p.
- Калянов Г.Н. CASE структурный системный анализ (автоматизация и применение). – М: «Лори», 1996.
- Виханский О.С. Стратегическое управление: Учебник.-2-е изд., перераб. и доп. – М.: Гардарика, 1998. – 296 с.
- Липаев В.В. Документирование и управление конфигурацией программных средств. Методы и стандарты. Серия «Информатизация России на пороге XXI века». – М.: СИНТЕГ, 1998. – 220 с.
- Липаев В.В. Системное проектирование сложных программных средств для информационных систем. Серия «Информатизация России на пороге XXI века». – М.: СИНТЕГ, 1999. – 224 с.
- Липаев В.В. Качество программных средств. Методические рекомендации. –М.: Янус-К, 2002. – 399 с.

Библиографический список

- **Нильсон Н.** Искусственный интеллект. Методы поиска и решений. –М.: Мир, 1973. – 270 с.
- **Вагин В.Н.** Дедукция и обобщение в системах принятия решений.–М.: Наука, 1988.
- **Вагин В.Н., Головина Е.Ю., Загорянская А.А., Фомина М.В.** Достоверный и правдоподобный вывод в интеллектуальных системах/ Под ред. В.Н. Вагина, Д. А. Поспелова. – М.:ФИЗМАТЛИТ, 2004. –704 с.–ISBN 5-9221-0474-8
- **Башмаков А.И., Башмаков И.А.** Интеллектуальные информационные технологии: Учеб. пособие.–М.: Изд-во МГТУ им. Н.Э. Баумана, 2005.–304 с.:ил. – (Информатика в техническом университете).
- **Головина Е.Ю.** Технологии создания корпоративных информационных систем с использованием интеллектуальных методов (Монография. Серия «Технология, оборудования и автоматизация машиностроительных производств»). – М.:Янус-К, 2002. – 107 с., ил. ISBN 5-8037-0085-1

Библиографический список

- **Головина Е.Ю.** Объектно-ориентированные и интеллектуальные технологии создания информационных систем:/учебное пособие/ Е.Ю. Головина.–М.: Издательский дом МЭИ, 2008.–95 с.
- **Головина Е.Ю.** Корпоративные информационные системы и методы их разработки:/учебное пособие/ Е.Ю. Головина.–М.: Издательский дом МЭИ, 2008.–94 с.
- **Головина Е.Ю.** Учебное пособие по курсу «CASE-технология проектирования программного обеспечения». Серия «Технология, оборудования и автоматизация машиностроительных производств». Модели и методы проектирования информационных систем. – М.: МГТУ «СТАНКИН», 2002. – 92 с.
- http://www.idef.ru
- http://www.citforum.ru
- http://www.interface.ru

Библиографический список

- ***«Реинжиниринг бизнес-процессов»***
- **Hammer M., Champy J.** Reengineering the Corporation: A Manifesto for Business Revolution. – New York: HarperCollins, 1993.
- **Ойхман В.Г., Попов Э.В.** Реинжиниринг бизнес-процессов.– М.: Финансы и статистика,1997. – 396 с.
- **Попов Э.В.** Реинжиниринг бизнес-процессов и искусственный интеллект// Новости Искусственного Интеллекта. 1996. – № 4.
- **Попов Э.В., Кисель Е.Б., Фоминых И.Б., Шапот М.Д.** Статические и динамические экспертные системы. Учебное пособие. – М.: Финансы и статистика, 1996.
- **Попов Э.В., Шапот М.Д.** Реинжиниринг бизнес-процессов и информационные технологии // Открытые системы. 1996. – № 1

Библиографический список

- ***«Объектно-ориентированная методология»***
- **Буч Г.** Объектно-ориентированное проектирование с примерами применения : Пер. с англ. – Совместное издание фирмы "Диалектика", г. Киев и АО "ИВК" г. Москва, 1992.
- **Shlaer S., Stephen J.** Mellor Object Lifecycles: Modeling the World in States – Prentice-Hall, Englewood Cliffs, N.J., 1992.
- **Буч Г.** Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е изд./Пер. с англ. – М.: Издательство Бином, СПб: Невский диалект, 1998. – 560 с.
- **Коуд П., Норт Д., Мейфилд М.** Объектные модели. Стратегии шаблоны и приложения. –М.: Из-во Лори, 1999. – 429 с.
- **Фаулер М., Скотт К.** UML в кратком изложении. Применение стандартного языка объектного моделирования: Пер. с англ. – М.: Мир, 1999. – 191 с., ил.
- **Архангельский А.Я.** Программирование в C++Builder 6. 2-изд.– М.:ООО «Бином-Пресс», 2005. – 1168 с.:ил.

Библиографический список

- *«CASE-средства RATIONAL ROSE»*
- **Applying** Use Cases: a practical guide / Ceri Schneider and Jason P. Winters. 208 p., 1998. http://www.awl.com/cseng/titles/0-201-30981-5/
- **The Rational** Unified Process: an introduction / Philippe Kruchten. 255 p. http://www.awl.com/cseng/titles/0-201-60459-0/
- **Visual** Modeling With Rational Rose And UML / Terry Quatrany. 222 p. http://www.awl.com/cseng/titles/0-201-31016-3/
- Web: www.rational.com
- Web: www.rosearchitect.com