**1.Классификация и жизненный цикл автоматизированных информационных систем.**

**1.1 Общая классификация**

Информация в современном мире превратилась в один из наиболее важных ресурсов, а информационные системы (ИС) стали необходимым инструментом практически во всех сферах деятельности.

Разнообразие задач, решаемых с помощью ИС, привело к появлению множества разнотипных систем, отличающихся принципами построения и заложенными в них правилами обработки информации.

Информационные системы можно **классифицировать** по целому ряду различных признаков. В основу рассматриваемой классификации положены наиболее существенные признаки, определяющие функциональные возможности и особенности построения современных систем. В зависимости от объема решаемых задач, используемых технических средств, организации функционирования, информационные системы делятся на ряд групп (классов) ([рис. 1.1](http://www.intuit.ru/department/se/devis/1/devis_1.html#image.1.1#image.1.1)).

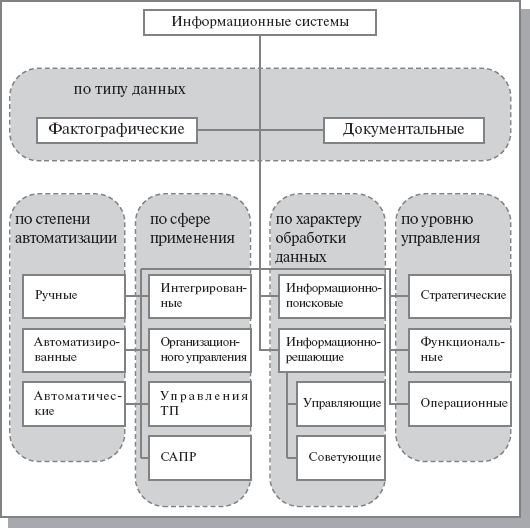
**По типу хранимых данных** ИС делятся на фактографические и документальные. Фактографические системы предназначены для хранения и обработки структурированных данных в виде чисел и текстов. Над такими данными можно выполнять различные операции. В документальных системах информация представлена в виде документов, состоящих из наименований, описаний, рефератов и текстов. Поиск по неструктурированным данным осуществляется с использованием семантических признаков. Отобранные документы предоставляются пользователю, а обработка данных в таких системах практически не производится.

Рисунок 1.1 Классификация информационных систем

Основываясь на **степени автоматизации информационных процессов** в системе управления фирмой, информационные системы делятся на ручные, автоматические и автоматизированные.

**Ручные** ИС характеризуются отсутствием современных технических средств переработки информации и выполнением всех операций человеком.

В **автоматических** ИС все операции по переработке информации выполняются без участия человека.

**Автоматизированные** ИС предполагают участие в процессе обработки информации и человека, и технических средств, причем главная роль в выполнении рутинных операций обработки данных отводится компьютеру. Именно этот класс систем соответствует современному представлению понятия "информационная система".

В зависимости от **характера обработки** данных ИС делятся на **информационно-поисковые и информационно-решающие.**

**Информационно-поисковые** системы производят ввод, систематизацию, хранение, выдачу информации по запросу пользователя без сложных преобразований данных. (Например, ИС библиотечного обслуживания, резервирования и продажи билетов на транспорте, бронирования мест в гостиницах и пр.)

**Информационно-решающие** системы осуществляют, кроме того, операции переработки информации по определенному алгоритму. По **характеру использования выходной информации** такие системы принято делить на **управляющие и советующие.**

Результирующая информация управляющих ИС непосредственно трансформируется в принимаемые человеком решения. Для этих систем характерны задачи расчетного характера и обработка больших объемов данных. (Например, ИС планирования производства или заказов, бухгалтерского учета.)

Советующие ИС вырабатывают информацию, которая принимается человеком к сведению и учитывается при формировании управленческих решений, а не инициирует конкретные действия. Эти системы имитируют интеллектуальные процессы обработки знаний, а не данных. (Например, экспертные системы.)

В зависимости от **сферы применения** различают следующие классы ИС.

Информационные системы **организационного управлени**я - предназначены для автоматизации функций управленческого персонала как промышленных предприятий, так и непромышленных объектов (гостиниц, банков, магазинов и пр.).

Основными функциями подобных систем являются: оперативный контроль и регулирование, оперативный учет и анализ, перспективное и оперативное планирование, бухгалтерский учет, управление сбытом, снабжением и другие экономические и организационные задачи.

ИС **управления технологическими процессами (ТП)** - служат для автоматизации функций производственного персонала по контролю и управлению производственными операциями. В таких системах обычно предусматривается наличие развитых средств измерения параметров технологических процессов (температуры, давления, химического состава и т.п.), процедур контроля допустимости значений параметров и регулирования технологических процессов.

ИС **автоматизированного проектирования (САПР)** - предназначены для автоматизации функций инженеров-проектировщиков, конструкторов, архитекторов, дизайнеров при создании новой техники или технологии. Основными функциями подобных систем являются: инженерные расчеты, создание графической документации (чертежей, схем, планов), создание проектной документации, моделирование проектируемых объектов.

**Интегрированные (корпоративные) ИС** - используются для автоматизации всех функций фирмы и охватывают весь цикл работ от планирования деятельности до сбыта продукции. Они включают в себя ряд модулей (подсистем), работающих в едином информационном пространстве и выполняющих функции поддержки соответствующих направлений деятельности. Общепринятая международная классификация корпоративных систем приведена в следующем параграфе.

Анализ современного состояния рынка ИС показывает устойчивую тенденцию роста спроса на информационные системы организационного управления. Причем спрос продолжает расти именно на интегрированные системы управления. Автоматизация отдельной функции, например, бухгалтерского учета или сбыта готовой продукции, считается уже пройденным этапом для многих предприятий.

В [таблице 1.](http://www.intuit.ru/department/se/devis/1/devis_1.html#table.1.2#table.1.2)1 приведен перечень наиболее популярных в настоящее время программных продуктов для реализации ИС организационного управления различных классов.

**Таблица 1.1. Классификация рынка информационных систем**

| **Локальные системы** | **Малые интегрированные системы** | **Средние интегрированные системы** | **Крупные интегрированные системы (IC)** |
| --- | --- | --- | --- |
| * БЭСТ * Инотек * Инфософт * Супер-Менеджер * Турбо-Бухгалтер * Инфо-Бухгалтер | * Concorde XAL Exact * NS-2000 Platinum PRO/MIS * Scala SunSystems * БЭСТ-ПРО * 1C-Предприятие * БОСС-Корпорация * Галактика * Парус * Ресурс * Эталон | * Microsoft-Business Solutions - Navision, * Axapta D Edwards (Robertson & Blums) * MFG-Pro (QAD/BMS) * SyteLine (COKAП/SYMIX) | * SAP/R3 (SAP AG) * Baan (Baan) * BPCS (ITS/SSA) * Oracle Applications (Оracle) |

Существует классификация ИС в зависимости от **уровня управления**, на котором система используется.

Информационная система оперативного уровня - поддерживает исполнителей, обрабатывая данные о сделках и событиях (счета, накладные, зарплата, кредиты, поток сырья и материалов). Информационная система оперативного уровня является связующим звеном между фирмой и внешней средой.

Задачи, цели, источники информации и алгоритмы обработки на оперативном уровне заранее определены и в высокой степени структурированы.

Информационные системы специалистов - поддерживают работу с данными и знаниями, повышают продуктивность и производительность работы инженеров и проектировщиков. Задача подобных информационных систем - интеграция новых сведений в организацию и помощь в обработке бумажных документов.

Информационные системы уровня менеджмента - используются работниками среднего управленческого звена для мониторинга, контроля, принятия решений и администрирования. Основные функции этих информационных систем:

* сравнение текущих показателей с прошлыми;
* составление периодических отчетов за определенное время, а не выдача отчетов по текущим событиям, как на оперативном уровне;
* обеспечение доступа к архивной информации и т.д.

Стратегическая информационная система - компьютерная информационная система, обеспечивающая поддержку принятия решений по реализации стратегических перспективных целей развития организации.

Информационные системы стратегического уровня помогают высшему звену управленцев решать неструктурированные задачи, осуществлять долгосрочное планирование. Основная задача - сравнение происходящих во внешнем окружении изменений с существующим потенциалом фирмы. Они призваны создать общую среду компьютерной телекоммуникационной поддержки решений в неожиданно возникающих ситуациях. Используя самые совершенные программы, эти системы способны в любой момент предоставить информацию из многих источников. Некоторые стратегические системы обладают ограниченными аналитическими возможностями.

С точки зрения **программно-аппаратной реализации** можно выделить ряд типовых архитектур ИС.

Традиционные архитектурные решения основаны на использовании выделенных файл-серверов или серверов баз данных. Существуют также варианты архитектур корпоративных информационных систем, базирующихся на технологии Internet (Intranet-приложения). Следующая разновидность архитектуры информационной системы основывается на концепции "хранилища данных" (DataWarehouse) - интегрированной информационной среды, включающей разнородные информационные ресурсы. И, наконец, для построения глобальных распределенных информационных приложений используется архитектура интеграции информационно-вычислительных компонентов на основе объектно-ориентированного подхода.

Индустрия разработки автоматизированных информационных систем управления зародилась в 1950-х - 1960-х годах и к концу века приобрела вполне законченные формы.

На первом этапе основным подходом в проектировании ИС был метод "снизу-вверх", когда система создавалась как набор приложений, наиболее важных в данный момент для поддержки деятельности предприятия. Основной целью этих проектов было не создание тиражируемых продуктов, а обслуживание текущих потребностей конкретного учреждения. Такой подход отчасти сохраняется и сегодня. В рамках "лоскутной автоматизации" достаточно хорошо обеспечивается поддержка отдельных функций, но практически полностью отсутствует стратегия развития комплексной системы автоматизации, а объединение функциональных подсистем превращается в самостоятельную и достаточно сложную проблему.

Создавая свои отделы и управления автоматизации, предприятия пытались "обустроиться" своими силами. Однако периодические изменения технологий работы и должностных инструкций, сложности, связанные с разными представлениями пользователей об одних и тех же данных, приводили к непрерывным доработкам программных продуктов для удовлетворения все новых и новых пожеланий отдельных работников. Как следствие - и работа программистов, и создаваемые ИС вызывали недовольство руководителей и пользователей системы.

Следующий этап связан с осознанием того факта, что существует потребность в достаточно стандартных программных средствах автоматизации деятельности различных учреждений и предприятий. Из всего спектра проблем разработчики выделили наиболее заметные: автоматизацию ведения бухгалтерского аналитического учета и технологических процессов. Системы начали проектироваться "сверху-вниз", т.е. в предположении, что одна программа должна удовлетворять потребности многих пользователей.

Сама идея использования универсальной программы накладывает существенные ограничения на возможности разработчиков по формированию структуры базы данных, экранных форм, по выбору алгоритмов расчета. Заложенные "сверху" жесткие рамки не дают возможности гибко адаптировать систему к специфике деятельности конкретного предприятия: учесть необходимую глубину аналитического и производственно-технологического учета, включить необходимые процедуры обработки данных, обеспечить интерфейс каждого рабочего места с учетом функций и технологии работы конкретного пользователя. Решение этих задач требует серьезных доработок системы. Таким образом, материальные и временные затраты на внедрение системы и ее доводку под требования заказчика обычно значительно превышают запланированные показатели.

Согласно статистическим данным, собранным Standish Group (США), из 8380 проектов, обследованных в США в 1994 году, неудачными оказались более 30% проектов, общая стоимость которых превышала 80 миллиардов долларов. При этом оказались выполненными в срок лишь 16% от общего числа проектов, а перерасход средств составил 189% от запланированного бюджета.

В то же время, заказчики ИС стали выдвигать все больше требований, направленных на обеспечение возможности комплексного использования корпоративных данных в управлении и планировании своей деятельности.

Таким образом, возникла насущная необходимость формирования новой методологии построения информационных систем.

**Цель такой методологии заключается в регламентации процесса проектирования ИС и обеспечении управления этим процессом с тем, чтобы гарантировать выполнение требований как к самой ИС, так и к характеристикам процесса разработки**. Основными задачами, решению которых должна способствовать методология проектирования корпоративных ИС, являются следующие:

* обеспечивать создание корпоративных ИС, отвечающих целям и задачам организации, а также предъявляемым требованиям по автоматизации деловых процессов заказчика;
* гарантировать создание системы с заданным качеством в заданные сроки и в рамках установленного бюджета проекта;
* поддерживать удобную дисциплину сопровождения, модификации и наращивания системы;
* обеспечивать преемственность разработки, т.е. использование в разрабатываемой ИС существующей информационной инфраструктуры организации (задела в области информационных технологий).

Внедрение методологии должно приводить к снижению сложности процесса создания ИС за счет полного и точного описания этого процесса, а также применения современных методов и технологий создания ИС на всем жизненном цикле ИС - от замысла до реализации.

Проектирование ИС охватывает три основные области:

* проектирование объектов данных, которые будут реализованы в базе данных;
* проектирование программ, экранных форм, отчетов, которые будут обеспечивать выполнение запросов к данным;
* учет конкретной среды или технологии, а именно: топологии сети, конфигурации аппаратных средств, используемой архитектуры (файл-сервер или клиент-сервер), параллельной обработки, распределенной обработки данных и т.п.

Проектирование информационных систем всегда начинается с определения цели проекта. В общем виде цель проекта можно определить как решение ряда взаимосвязанных задач, включающих в себя обеспечение на момент запуска системы и в течение всего времени ее эксплуатации:

* требуемой функциональности системы и уровня ее адаптивности к изменяющимся условиям функционирования;
* требуемой пропускной способности системы;
* требуемого времени реакции системы на запрос;
* безотказной работы системы;
* необходимого уровня безопасности;
* простоты эксплуатации и поддержки системы.

Согласно современной методологии, процесс создания ИС представляет собой процесс построения и последовательного преобразования ряда согласованных моделей на всех этапах жизненного цикла (ЖЦ) ИС. На каждом этапе ЖЦ создаются специфичные для него модели - организации, требований к ИС, проекта ИС, требований к приложениям и т.д. Модели формируются рабочими группами команды проекта, сохраняются и накапливаются в репозитарии проекта. Создание моделей, их контроль, преобразование и предоставление в коллективное пользование осуществляется с использованием специальных программных инструментов - CASE-средств.

Процесс создания ИС делится на ряд **этапов** (стадий [[1]](http://www.intuit.ru/department/se/devis/popup.lit.html#1)), ограниченных некоторыми временными рамками и заканчивающихся выпуском конкретного продукта (моделей, программных продуктов, документации и пр.).

Обычно выделяют следующие этапы создания ИС: формирование требований к системе, проектирование, реализация, тестирование, ввод в действие, эксплуатация и сопровождение [[1]](http://www.intuit.ru/department/se/devis/popup.lit.html#1) [[2]](http://www.intuit.ru/department/se/devis/popup.lit.html#2). (Последние два этапа далее не рассматриваются, поскольку выходят за рамки тематики книги.)

Начальным этапом процесса создания ИС является моделирование бизнес-процессов, протекающих в организации и реализующих ее цели и задачи. Модель организации, описанная в терминах бизнес-процессов и бизнес-функций, позволяет сформулировать основные требования к ИС. Это фундаментальное положение методологии обеспечивает объективность в выработке требований к проектированию системы. Множество моделей описания требований к ИС затем преобразуется в систему моделей, описывающих концептуальный проект ИС. Формируются модели архитектуры ИС, требований к программному обеспечению (ПО) и информационному обеспечению (ИО). Затем формируется архитектура ПО и ИО, выделяются корпоративные БД и отдельные приложения, формируются модели требований к приложениям и проводится их разработка, тестирование и интеграция.

Целью начальных этапов создания ИС, выполняемых на стадии анализа деятельности организации, является формирование требований к ИС, корректно и точно отражающих цели и задачи организации-заказчика. Чтобы специфицировать процесс создания ИС, отвечающей потребностям организации, нужно выяснить и четко сформулировать, в чем заключаются эти потребности. Для этого необходимо определить требования заказчиков к ИС и отобразить их на языке моделей в требования к разработке проекта ИС так, чтобы обеспечить соответствие целям и задачам организации.

Задача формирования требований к ИС является одной из наиболее ответственных, трудно формализуемых и наиболее дорогих и тяжелых для исправления в случае ошибки. Современные инструментальные средства и программные продукты позволяют достаточно быстро создавать ИС по готовым требованиям. Но зачастую эти системы не удовлетворяют заказчиков, требуют многочисленных доработок, что приводит к резкому удорожанию фактической стоимости ИС. Основной причиной такого положения является неправильное, неточное или неполное определение требований к ИС на этапе анализа.

На этапе проектирования прежде всего формируются модели данных. Проектировщики в качестве исходной информации получают результаты анализа. Построение логической и физической моделей данных является основной частью проектирования базы данных. Полученная в процессе анализа информационная модель сначала преобразуется в логическую, а затем в физическую модель данных.

Параллельно с проектированием схемы базы данных выполняется проектирование процессов, чтобы получить спецификации (описания) всех модулей ИС. Оба эти процесса проектирования тесно связаны, поскольку часть бизнес-логики обычно реализуется в базе данных (ограничения, триггеры, хранимые процедуры). Главная цель проектирования процессов заключается в отображении функций, полученных на этапе анализа, в модули информационной системы. При проектировании модулей определяют интерфейсы программ: разметку меню, вид окон, горячие клавиши и связанные с ними вызовы.

Конечными продуктами этапа проектирования являются:

* схема базы данных (на основании ER-модели, разработанной на этапе анализа);
* набор спецификаций модулей системы (они строятся на базе моделей функций).

Кроме того, на этапе проектирования осуществляется также разработка архитектуры ИС, включающая в себя выбор платформы (платформ) и операционной системы (операционных систем). В неоднородной ИС могут работать несколько компьютеров на разных аппаратных платформах и под управлением различных операционных систем. Кроме выбора платформы, на этапе проектирования определяются следующие характеристики архитектуры:

* будет ли это архитектура "файл-сервер" или "клиент-сервер";
* будет ли это 3-уровневая архитектура со следующими слоями: сервер, ПО промежуточного слоя (сервер приложений), клиентское ПО;
* будет ли база данных централизованной или распределенной. Если база данных будет распределенной, то какие механизмы поддержки согласованности и актуальности данных будут использоваться;
* будет ли база данных однородной, то есть, будут ли все серверы баз данных продуктами одного и того же производителя (например, все серверы только Oracle или все серверы только DB2 UDB). Если база данных не будет однородной, то какое ПО будет использовано для обмена данными между СУБД разных производителей (уже существующее или разработанное специально как часть проекта);.
* будут ли для достижения должной производительности использоваться параллельные серверы баз данных (например, Oracle Parallel Server, DB2 UDB и т.п.).

Этап проектирования завершается разработкой технического проекта ИС.

На этапе реализации осуществляется создание программного обеспечения системы, установка технических средств, разработка эксплуатационной документации.

Этап тестирования обычно оказывается распределенным во времени.

После завершения разработки отдельного модуля системы выполняют автономный тест, который преследует две основные цели:

* обнаружение отказов модуля (жестких сбоев);
* соответствие модуля спецификации (наличие всех необходимых функций, отсутствие лишних функций).

После того как автономный тест успешно пройдет, модуль включается в состав разработанной части системы и группа сгенерированных модулей проходит тесты связей, которые должны отследить их взаимное влияние.

Далее группа модулей тестируется на надежность работы, то есть проходят, во-первых, тесты имитации отказов системы, а во-вторых, тесты наработки на отказ. Первая группа тестов показывает, насколько хорошо система восстанавливается после сбоев программного обеспечения, отказов аппаратного обеспечения. Вторая группа тестов определяет степень устойчивости системы при штатной работе и позволяет оценить время безотказной работы системы. В комплект тестов устойчивости должны входить тесты, имитирующие пиковую нагрузку на систему.

Затем весь комплект модулей проходит системный тест - тест внутренней приемки продукта, показывающий уровень его качества. Сюда входят тесты функциональности и тесты надежности системы.

Последний тест информационной системы - приемо-сдаточные испытания. Такой тест предусматривает показ информационной системы заказчику и должен содержать группу тестов, моделирующих реальные бизнес-процессы, чтобы показать соответствие реализации требованиям заказчика.

Необходимость контролировать процесс создания ИС, гарантировать достижение целей разработки и соблюдение различных ограничений (бюджетных, временных и пр.) привело к широкому использованию в этой сфере методов и средств программной инженерии: структурного анализа, объектно-ориентированного моделирования, CASE-систем.

**1.2 Классификация корпоративных АИС**

**MRP (Material Resources Planning)**

Первые мысли о возможности использования средств вычислительной техники (СВТ) для планирования деятельности предприятий (в том числе, планирования производственных процессов) возникли еще в начале 60-х гг. С целью решения с помощью СВТ вышерассмотренных проблем и была разработана методология планирования потребности в материалах **MRP (material requirements planning)**. Суть концепции MRP состоит в том, чтобы минимизировать издержки, связанные со складскими запасами и на различных участках в производстве. В основе этой концепции лежит понятие **спецификации изделия (bill of material — BOM)**, которое показывает зависимость спроса на сырье, полуфабрикаты и др. от плана выпуска готовой продукции (с учетом времени). На основе плана выпуска продукции, спецификации изделия и учета особенностей технологической цепочки и осуществляется расчет потребностей производства в материалах (обязательно привязанный к конкретным срокам).

Концепция MRP легла в основу построения так называемых MRP-систем. **Главной задачей MRP-систем является обеспечение наличия на складе необходимого количества требуемых материалов/комплектующих в любой момент времени в рамках срока планирования.** Программные системы, реализованные на базе MRP-методологии, позволили оптимально регулировать поставки комплектующих для производства продукции, контролировать складские запасы и саму технологию производства. Кроме того, использование MRP-систем позволило уменьшить объем постоянных складских запасов.

Сначала с помощью MRP-систем просто формировался на основе утвержденной производственной программы план заказов на определенный период. Это не вполне удовлетворяло возрастающие потребности предприятий. С целью повышения эффективности планирования в конце 70-х гг. в MRP-системах была реализована идея воспроизведения замкнутого цикла **(Cloosed Loop Material Requirment Planning)**, подразумевающая составление производственной программы и ее контроль на цеховом уровне. К базовым функциям планирования производственных мощностей и планирования потребностей в материалах были добавлены дополнительные функции (например, контроля соответствия количества произведенной продукции количеству использованных в процессе сборки комплектующих, составления регулярных отчетов о задержках заказов, об объемах и динамике продаж продукции, о поставщиках и др.). Cозданные в процессе работы модифицированной MRP-системы отчеты анализировались и учитывались на дальнейших этапах планирования, изменяя (при необходимости) программу производства и план заказов (обеспечивая, тем самым, гибкость планирования по отношению к таким внешним факторам, как уровень спроса, текущее состояние дел у поставщиков комплектующих и др.).

**ERP (Enterprise Resource Planning)**

Основные понятия производственного менеджмента (в том числе и термин «ERP») можно считать вполне устоявшимися. В этой области признанным «стандартом де-факто» служит терминология Американской ассоциации по управлению запасами и производством (*American Production and Inventory Control Society, APICS*). Основные термины и определения приводятся в Словаре APICS, который регулярно обновляется по мере развития теории и практики управления. Именно в этом издании содержится наиболее полное и точное определение ERP-системы.

В соответствии со Словарем APICS, термин «ERP-система» (*Enterprise Resource Planning* — Управление ресурсами предприятия) может употребляться в двух значениях.

Во-первых, это — информационная система для идентификации и планирования всех ресурсов предприятия, которые необходимы для осуществления продаж, производства, закупок и учета в процессе выполнения клиентских заказов.

Во-вторых (в более общем контексте), это — методология эффективного планирования и управления всеми ресурсами предприятия, которые необходимы для осуществления продаж, производства, закупок и учета при исполнении заказов клиентов в сферах производства, дистрибьюции и оказания услуг.

Таким образом, термин ERP может означать не только информационную систему, но и соответствующую методологию управления, реализуемую и поддерживаемую этой информационной системой.

**Основные функции ERP-системы**

Большинство современных ERP-систем построены по модульному принципу, что дает заказчику возможность выбора и внедрения лишь тех модулей, которые ему действительно необходимы. Модули разных ERP-систем могут отличаться как по названиям, так и по содержанию. Тем не менее, есть некоторый набор функций, который может считаться типовым для программных продуктов класса ERP. Такими типовыми функциями являются

* ведение конструкторских и технологических спецификаций. Такие спецификации определяют состав конечного изделия, а также материальные ресурсы и операции, необходимые для его изготовления (включая маршрутизацию);
* управление спросом и формирование планов продаж и производства. Эти функции предназначены для прогноза спроса и планирования выпуска продукции;
* планирование потребностей в материалах. Позволяют определить объемы различных видов материальных ресурсов (сырья, материалов, комплектующих), необходимых для выполнения производственного плана, а также сроки поставок, размеры партий и т.д.;
* управление запасами и закупочной деятельностью>. Позволяют организовать ведение договоров, реализовать схему централизованных закупок, обеспечить учет и оптимизацию складских запасов и т.д.;
* планирование производственных мощностей. Эта функция позволяет контролировать наличие доступных мощностей и планировать их загрузку. Включает укрупненное планирование мощностей (для оценки реалистичности производственных планов) и более детальное планирование, вплоть до отдельных рабочих центров;
* финансовые функции. В эту группу входят функции финансового учета, управленческого учета, а также оперативного управления финансами;
* функции управления проектами. Обеспечивают планирование задач проекта и ресурсов, необходимых для их реализации.
* **ERP II (Enterprise Resource & Relationship Processing)**
* По определению, данному Gartner Group ERP II, - это бизнес-стратегия предприятия, принадлежащего к определенной отрасли, и набор ключевых для данной отрасли приложений, помогающих клиентам и акционерам компаний увеличивать стоимость бизнеса за счет эффективной ИТ-поддержки и оптимизации операционных и финансовых процессов как внутри своего предприятия, так и во внешнем мире - в рамках сотрудничества с другими корпорациями.
* Основная идея ERP II заключается в выходе за рамки задач по оптимизации внутренних процессов организации: кроме интеграции таких традиционных для ERP систем областей деятельности предприятия, как управление финансами, бухгалтерский учет, управление продажами и покупками, отношения с дебиторами и кредиторами, управление персоналом, производство, управление запасами, системы класса ERP II позволяют управлять взаимоотношениями с клиентами, цепочками поставок, вести торговлю через Интернет.
* Функции и преимущества систем, которые реализованы в ERPII.
* **Управление взаимоотношениями с клиентами (CRM)**
* Новая функциональность ERP II, позволяет эффективно управлять контактами с клиентами, рекламными кампаниями, сбытом, проводить маркетинговые исследования. Это достигается за счет создания персональных профилей клиентов, классификации клиентов по различным категориям, определения целевых групп для рекламных кампаний, планирования и контроля взаимодействия с клиентами (телефонные звонки, визиты, рассылка рекламных и маркетинговых материалов и т. д.), упрощенного доступа к данным о существующих и потенциальных клиентах, поставщиках. Существует также возможность тарифицировать различные типы контактов и точно оценивать затраты на проведение различных мероприятий. Благодаря интеграции с другими подсистемами появляется возможность сопоставления с результатов маркетинговых кампаний с фактическими данными продаж. Это позволяет повысить координацию работы служб маркетинга, сбыта и снабжения.
* Сервисным подразделениям система дает возможность организовать эффективное управление послепродажным обслуживанием клиентов. Функциональные средства позволяют регистрировать действия, осуществляемые компанией после продажи товаров. При этом заявки на сервисное обслуживание могут быть инициированы как самими клиентами, так и автоматически, согласно сервисному контракту. Данные в сервисный заказ могут быть внесены как сотрудником, принявшим звонок, так и работником сервисной службы. В системе имеется возможность определять приоритетные сервисные заказы и загрузку персонала, выбирать персонал и работников технической службы по их занятости, профессиональной ориентации. Проанализировав исторические данные о сервисных работах можно выявлять наиболее "слабые места" в производимой продукции. Интеграция с подсистемой управления запасами позволяет при приеме заказа на сервисное обслуживание выяснить наличие запасных частей, а в случае их отсутствия определить сроки поступления на склад или зарегистрировать заявку на приобретение. Клиент в любой момент может проследить, как выполнялся заказ и его статус.
* **Электронная коммерция**
* Для продвижения своего бизнеса в Интернет необходимо, прежде всего, выстроить и отработать свои бизнес процессы до открытия Интернет представительства. Отсутствие "надежного тыла" становиться серьезной проблемой для компании.
* Это обусловлено тем, что при обычной схеме работы задача часто решается следующим образом: создается каталог продукции с описанием товаров и интерактивной формой заказа, клиент заказывает товар и ждет его доставки. Однако зачастую, осуществляя заказ, клиент, да и сами менеджеры, не имеют понятия, есть ли товар на складе и в случае отсутствия, сколько времени понадобится, чтобы доставить товар на склад, а затем переправить его клиенту. Более того, если единожды компания осуществляла специальный заказ для клиента, данные об этом не сохраняются, и при вторичном заказе компания вновь обрабатывает его как новый.
* Это приводит к простоям, затягивает сроки доставки заказов, что, в конечном счете, влечет за собой полное разочарование клиентов. Функциональность систем ERP II, в свою очередь, позволяют контролировать склад, закупки и поставки и организовывать систему работы оптимальным образом. Помимо оптимизации рабочих процессов, решения ERP II позволяет создать клиентскую базу, хранить и анализировать данные по предпочтениям. Компания также получают возможность прогнозировать сроки последующих заказов постоянных покупателей и предоставлять более персонализированный сервис, что служит выработке приверженности покупателей к сотрудничеству и установлению длительных отношений с клиентами. Таким образом, использование ERP-систем при ведении электронной торговли позволяет компаниям добиться преимущества по отношению к конкурентам.
* Что касается рынка B2B, то этот сегмент в СНГ развит все еще довольно слабо. Это связано, с тем, что системы B2B весьма дорогостоящи, и внедрять их могут позволить только крупные предприятия сырьевой направленности или предприятия, выпускающие высоко ликвидные товары. Но с появлением на рынке систем ERP II, ориентированных на средние предприятия, компании получают возможность развивать торговые площадки в Интернет, которые оптимально будут отвечать их нуждам - обеспечивать электронный документооборот, контролировать движение заказов и т.д. Это позволит компаниям использовать свои Интернет - ресурсы для взаимодействия с поставщиками или дистрибьюторами.
* Использование систем класса ERP II в данном случае позволяет оптимизировать процессы, как закупок, так и продаж. Информация о заказах, полученных через Интернет, интегрируется с данными склада, отделов доставки, продаж, сервисных центров, что позволяет создать единый профиль клиента, эффективно обрабатывать заказы и быстро отвечать на них, создавать и хранить данные обо всех его обращениях, анализировать их и прогнозировать новые обращения.
* Клиент, в свою очередь, отправив заказ через Интернет - представительство, получает возможность контролировать процесс обработки заказа: автоматическое уведомление о его принятии и начале работы, данные о подготовке заказа, когда его заявка получена складом, сведения об отгрузке, когда заказ сформирован и отправлен, а также пакет необходимых документов.
* Аналогично используется система при закупках. Менеджер по закупкам через систему получает доступ к каталогу товаров поставщика, выбирает необходимые товары и отправляет заказ. Возможны варианты, когда система автоматически формирует заказ на покупку, учитывая оптимальные объемы закупок, текущие потребности, необходимые сроки поставки и в стандартном формате пересылает поставщику. Система у поставщика обрабатывает полученный заказ, вычисляет срок поставки и посылает уведомление о ходе выполнения заказа.

**CRM (Customer Relationship Management)**

В современных условиях жесткой ценовой конкуренции предприятия вынуждены искать новые неценовые конкурентные преимущества. Одним из таких преимуществ является достижение уникальных взаимоотношений с клиентами, повышение их лояльности к компании.

Лояльность, как правило, измеряется количеством совершенных повторных покупок или намерением совершить таковые. Проанализируем общеизвестные факты, имеющие отношение к лояльности потребителей:

* Затраты на привлечение нового клиента в среднем в пять раз больше, чем на удержание существующего.
* Удовлетворенный клиент расскажет о удачной покупке в среднем 5 своим знакомым. Неудовлетворенный - минимум 10.
* Большая часть клиентов окупается лишь через год работы с ними (а это значит, что, если клиент "ушел" до этого срока, то он принес убытки)
* Около 50% существующих клиентов компании не прибыльны из-за неэффективного взаимодействия с ними.
* Лояльный клиент менее чувствителен к повышению цен и не требует интенсивной рекламы.
* Большая часть компаний из списка Fortune 500 теряет 50% своих клиентов каждые 5 лет.

Выводы напрашиваются сами собой:

1. В современных условиях эффективность работы предприятия напрямую зависит от лояльности ее клиентов.
2. Повышение лояльности клиентов - важнейший неценовой способ борьбы с конкуренцией.
3. Для большинства компаний лояльность покупателей является единственно важным источником долгосрочного роста и прибыльности.

В связи с этим и встает вопрос об управлении взаимоотношениями с клиентами, причем, не просто об управлении, а о его научной организации и технической поддержке. Все это и обозвали в середине 90-х годов неким термином CRM. Но если тогда это было просто модной новинкой, то в настоящее время, когда клиентам уже недостаточно просто приятных цен и широкого ассортимента, **CRM стал стандартным подходом цивилизованных компаний к решению вопроса о борьбе с конкурентами.** **Эффективность внедрения** напрямую зависит от того, насколько руководством компании осознается факт , что CRM - это не просто ПО или технология, но также и определенная культура работы компании, т.е. от того, как рассматриваются **различные аспекты CRM**.

При внедрении CRM соблюдается два основных принципа функционирования этой системы. Во-первых, это всеохватность (т.е. каждый этап взаимодействия предприятия с потребителями, как реальными, так и потенциальными, должен быть учтен в CRM ), а во-вторых, взаимосвязанность (т.е все эти учитываемые этапы должны быть связаны между собой для получения правильных результатов анализа на основе CRM-технологий).

**EAM (Enterprise Asset Management)**

Методология EAM (Enterprise Asset Management) - управление основными фондами предприятия.Позволяет сократить простои оборудования, затраты на техобслуживание, ремонты и материально-техническое снабжение. EAM-системы позволяют согласованно управлять следующими процессами:

 техническое обслуживание и ремонт (ТОиР);

 материально-техническое снабжение (МТС);

 управление складскими запасами (запчасти для ТОиР);

 управление финансами (в области ТОиР и МТС);

 управление персоналом (в области ТОиР и МТС);

 управление документами (в области ТОиР и МТС).

Исторически EAM-системы возникли из CMMS-систем — систем управления ремонтами. В настоящее время модули EAM входят также в состав крупных пакетов управленческого программного обеспечения — ERP-систем (mySAP Business Suite, IFS Applications, Oracle E-Business Suite и др.).На Западе, где конкуренция жестче и необходимость снижать издержки очевидна, EAM-системы — непременный инструмент работы предприятий фондоёмких отраслей (то есть энергетики, ЖКХ, промышленности с процессным типом производства, транспорта, добывающей промышленности и вооруженных сил).

Методология HR (Human Resources) - управление персоналом.

HR подразумевает:

- управление персоналом;

- расчет заработной платы;

- табельный учет;

- самообслуживания персонала и т.д.

**1.3 Жизненный цикл АИС**

Методология проектирования информационных систем описывает процесс создания и сопровождения систем в виде жизненного цикла (ЖЦ) ИС, представляя его как некоторую последовательность стадий и выполняемых на них процессов. Для каждого этапа определяются состав и последовательность выполняемых работ, получаемые результаты, методы и средства, необходимые для выполнения работ, роли и ответственность участников и т.д. Такое формальное описание ЖЦ ИС позволяет спланировать и организовать процесс коллективной разработки и обеспечить управление этим процессом.

Жизненный цикл ИС можно представить как ряд событий, происходящих с системой в процессе ее создания и использования.

Модель жизненного цикла отражает различные состояния системы, начиная с момента возникновения необходимости в данной ИС и заканчивая моментом ее полного выхода из употребления. Модель жизненного цикла - структура, содержащая процессы, действия и задачи, которые осуществляются в ходе разработки, функционирования и сопровождения программного продукта в течение всей жизни системы, от определения требований до завершения ее использования.

В настоящее время известны и используются следующие модели жизненного цикла:

* Каскадная модель (рис. 1.2 ) предусматривает последовательное выполнение всех этапов проекта в строго фиксированном порядке. Переход на следующий этап означает полное завершение работ на предыдущем этапе.
* Поэтапная модель с промежуточным контролем ([рис. 1.](http://www.intuit.ru/department/se/devis/2/devis_2.html#image.2.2#image.2.2)3). Разработка ИС ведется итерациями с циклами обратной связи между этапами. Межэтапные корректировки позволяют учитывать реально существующее взаимовлияние результатов разработки на различных этапах; время жизни каждого из этапов растягивается на весь период разработки.
* Спиральная модель ([рис. 1.](http://www.intuit.ru/department/se/devis/2/devis_2.html#image.2.3#image.2.3)4). На каждом витке спирали выполняется создание очередной версии продукта, уточняются требования проекта, определяется его качество и планируются работы следующего витка.Особое внимание уделяется начальным этапам разработки - анализу и проектированию, где реализуемость тех или иных технических решений проверяется и обосновывается посредством создания прототипов (макетирования).

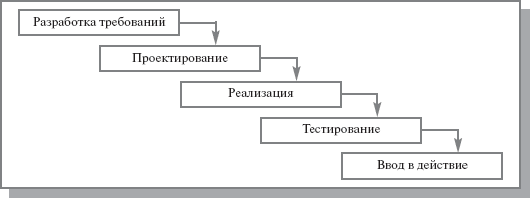


Рис. 1.2. Каскадная модель ЖЦ ИС

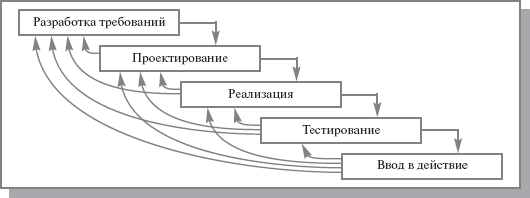


Рис. 1.3. Поэтапная модель с промежуточным контролем

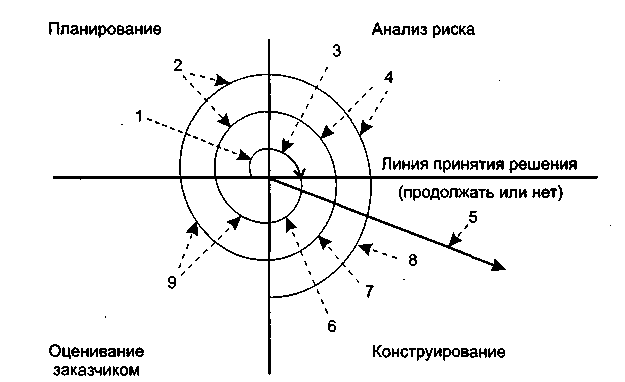


Рис. 1.4.  Спиральная модель ЖЦ ИС   
: *1* — начальный сбор требований и планирование проекта;

2 — та же работа, но на основе рекомендаций заказчика; 3 — анализ риска на основе

начальных требований; 4 — анализ риска на основе реакции заказчика; 5 — переход

к комплексной системе; 6 — начальный макет системы; 7 — следующий уровень макета;

8 — сконструированная система; 9 — оценивание заказчиком

На практике наибольшее распространение получили две основные модели жизненного цикла:

* каскадная модель (характерна для периода 1970-1985 гг.);
* спиральная модель (характерна для периода после 1986.г.).

В ранних проектах достаточно простых ИС каждое приложение представляло собой единый, функционально и информационно независимый блок. Для разработки такого типа приложений эффективным оказался каскадный способ. Каждый этап завершался после полного выполнения и документального оформления всех предусмотренных работ.

Можно выделить следующие положительные стороны применения каскадного подхода:

* на каждом этапе формируется законченный набор проектной документации, отвечающий критериям полноты и согласованности;
* выполняемые в логической последовательности этапы работ позволяют планировать сроки завершения всех работ и соответствующие затраты.

Каскадный подход хорошо зарекомендовал себя при построении относительно простых ИС, когда в самом начале разработки можно достаточно точно и полно сформулировать все требования к системе. Основным недостатком этого подхода является то, что реальный процесс создания системы никогда полностью не укладывается в такую жесткую схему, постоянно возникает потребность в возврате к предыдущим этапам и уточнении или пересмотре ранее принятых решений. В результате реальный процесс создания ИС оказывается соответствующим поэтапной модели с промежуточным контролем.

Однако и эта схема не позволяет оперативно учитывать возникающие изменения и уточнения требований к системе. Согласование результатов разработки с пользователями производится только в точках, планируемых после завершения каждого этапа работ, а общие требования к ИС зафиксированы в виде технического задания на все время ее создания. Таким образом, пользователи зачастую получают систему, не удовлетворяющую их реальным потребностям.

Спиральная модель ЖЦ была предложена для преодоления перечисленных проблем. На этапах анализа и проектирования реализуемость технических решений и степень удовлетворения потребностей заказчика проверяется путем создания прототипов. Каждый виток спирали соответствует созданию работоспособного фрагмента или версии системы. Это позволяет уточнить требования, цели и характеристики проекта, определить качество разработки, спланировать работы следующего витка спирали. Таким образом углубляются и последовательно конкретизируются детали проекта и в результате выбирается обоснованный вариант, который удовлетворяет действительным требованиям заказчика и доводится до реализации.

Как показано на рис. 1.4, модель определяет четыре действия, представляемые четырьмя квадрантами спирали.

1. Планирование — определение целей, вариантов и ограничений.

2. Анализ риска — анализ вариантов и распознавание/выбор риска.

3. Конструирование — разработка продукта следующего уровня.

4. Оценивание — оценка заказчиком текущих результатов конструирования.

Интегрирующий аспект спиральной модели очевиден при учете радиального измерения спирали. С каждой итерацией по спирали (продвижением от центра к периферии) строятся все более полные версии ПО.

В первом витке спирали определяются начальные цели, варианты и ограничения, распознается и анализируется риск. Если анализ риска показывает неопределенность требований, на помощь разработчику и заказчику приходит макетирование (используемое в квадранте конструирования). Для дальнейшего определения проблемных и уточненных требований может быть использовано моделирование. Заказчик оценивает инженерную (конструкторскую) работу и вносит предложения по модификации (квадрант оценки заказчиком). Следующая фаза планирования и анализа риска базируется на предложениях заказчика. В каждом цикле по спирали результаты анализа риска формируются в виде «продолжать, не продолжать». Если риск слишком велик, проект может быть остановлен.

В большинстве случаев движение по спирали продолжается, с каждым шагом продвигая разработчиков к более общей модели системы. В каждом цикле по спирали требуется конструирование (нижний правый квадрант), которое может быть реализовано классическим жизненным циклом или макетированием. Заметим, что количество действий по разработке (происходящих в правом нижнем квадранте) возрастает по мере продвижения от центра спирали.

Итеративная разработка отражает объективно существующий спиральный цикл создания сложных систем. Она позволяет переходить на следующий этап, не дожидаясь полного завершения работы на текущем и решить главную задачу - 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 вводятся временные ограничения на каждый из этапов жизненного цикла, и переход осуществляется в соответствии с планом, даже если не вся запланированная работа закончена. Планирование производится на основе статистических данных, полученных в предыдущих проектах, и личного опыта разработчиков.

Несмотря на настойчивые рекомендации компаний - вендоров и экспертов в области проектирования и разработки ИС, многие компании продолжают использовать каскадную модель вместо какого-либо варианта итерационной модели. Основные причины, по которым каскадная модель сохраняет свою популярность, следующие [[3]](http://www.intuit.ru/department/se/devis/popup.lit.html#3):

1. **Привычка** - многие ИТ-специалисты получали образование в то время, когда изучалась только каскадная модель, поэтому она используется ими и в наши дни.
2. **Иллюзия снижения рисков** участников проекта (заказчика и исполнителя).Каскадная модель предполагает разработку законченных продуктов на каждом этапе: технического задания, технического проекта, программного продукта и пользовательской документации. Разработанная документация позволяет не только определить требования к продукту следующего этапа, но и определить обязанности сторон, объем работ и сроки, при этом окончательная оценка сроков и стоимости проекта производится на начальных этапах, после завершения обследования. Очевидно, что если требования к информационной системе меняются в ходе реализации проекта, а качество документов оказывается невысоким (требования неполны и/или противоречивы), то в действительности использование каскадной модели создает лишь иллюзию определенности и на деле увеличивает риски, уменьшая лишь ответственность участников проекта. При формальном подходе менеджер проекта реализует только те требования, которые содержатся в спецификации, опирается на документ, а не н а реальные потребности бизнеса. Есть два основных типа контрактов на разработку ПО. Первый тип предполагает выполнение определенного объема работ за определенную сумму в определенные сроки (fixed price). Второй тип предполагает повременную оплату работы (time work). Выбор того или иного типа контракта зависит от степени определенности задачи. Каскадная модель с определенными этапами и их результатами лучше приспособлена для заключения контракта с оплатой по результатам работы, а именно этот тип контрактов позволяет получить полную оценку стоимости проекта до его завершения. Более вероятно заключение контракта с повременной оплатой на небольшую систему, с относительно небольшим весом в структуре затрат предприятия. Разработка и внедрение интегрированной информационной системы требует существенных финансовых затрат, поэтому используются контракты с фиксированной ценой, и, следовательно, каскадная модель разработки и внедрения. Спиральная модель чаще применяется при разработке информационной системы силами собственного отдела ИТ предприятия.
3. **Проблемы внедрения** при использовании итерационной модели. В некоторых областях спиральная модель не может применяться, поскольку невозможно использование/тестирование продукта, обладающего неполной функциональностью (например, военные разработки, атомная энергетика и т.д.). Поэтапное итерационное внедрение информационной системы для бизнеса возможно, но сопряжено с организационными сложностями (перенос данных, интеграция систем, изменение бизнес-процессов, учетной политики, обучение пользователей). Трудозатраты при поэтапном итерационном внедрении оказываются значительно выше, а управление проектом требует настоящего искусства. Предвидя указанные сложности, заказчики выбирают каскадную модель, чтобы "внедрять систему один раз".

Каждая из стадий создания системы предусматривает выполнение определенного объема работ, которые представляются в виде процессов ЖЦ. Процесс определяется как совокупность взаимосвязанных действий, преобразующих входные данные в выходные. Описание каждого процесса включает в себя перечень решаемых задач, исходных данных и результатов.

Существует целый ряд стандартов, регламентирующих ЖЦ ПО, а в некоторых случаях и процессы разработки.

Значительный вклад в теорию проектирования и разработки информационных систем внесла компания IBM, предложив еще в середине 1970-х годов методологию BSP (Business System Planning - методология организационного планирования). Метод структурирования информации с использованием матриц пересечения бизнес-процессов, функциональных подразделений, функций систем обработки данных (информационных систем), информационных объектов, документов и баз данных, предложенный в BSP, используется сегодня не только в ИТ-проектах, но и проектах по реинжинирингу бизнес-процессов, изменению организационной структуры. Важнейшие шаги процесса BSP, их последовательность (получить поддержку высшего руководства, определить процессы предприятия, определить классы данных, провести интервью, обработать и организовать данные интервью) можно встретить практически во всех формальных методиках, а также в проектах, реализуемых на практике.

Среди наиболее известных стандартов можно выделить следующие:

* ГОСТ 34.601-90 - распространяется на автоматизированные системы и устанавливает стадии и этапы их создания. Кроме того, в стандарте содержится описание содержания работ на каждом этапе. Стадии и этапы работы, закрепленные в стандарте, в большей степени соответствуют каскадной модели жизненного цикла [[4]](http://www.intuit.ru/department/se/devis/popup.lit.html#4).
* ISO/IEC 12207:1995 - стандарт на процессы и организацию жизненного цикла. Распространяется на все виды заказного ПО. Стандарт не содержит описания фаз, стадий и этапов [[5]](http://www.intuit.ru/department/se/devis/popup.lit.html#5).
* Custom Development Method (методика Oracle) по разработке прикладных информационных систем - технологический материал, детализированный до уровня заготовок проектных документов, рассчитанных на использование в проектах с применением Oracle. Применяется CDM для классической модели ЖЦ (предусмотрены все работы/задачи и этапы), а также для технологий "быстрой разработки" (Fast Track) или "облегченного подхода", рекомендуемых в случае малых проектов.
* Rational Unified Process (RUP) предлагает итеративную модель разработки, включающую четыре фазы: начало, исследование, построение и внедрение. Каждая фаза может быть разбита на этапы (итерации), в результате которых выпускается версия для внутреннего или внешнего использования. Прохождение через четыре основные фазы называется циклом разработки, каждый цикл завершается генерацией версии системы. Если после этого работа над проектом не прекращается, то полученный продукт продолжает развиваться и снова минует те же фазы. Суть работы в рамках RUP - это создание и сопровождение моделей на базе UML [[6]](http://www.intuit.ru/department/se/devis/popup.lit.html#6).
* Microsoft Solution Framework (MSF) сходна с RUP, так же включает четыре фазы: анализ, проектирование, разработка, стабилизация, является итерационной, предполагает использование объектно-ориентированного моделирования. MSF в сравнении с RUP в большей степени ориентирована на разработку бизнес-приложений.
* Extreme Programming (XP). Экстремальное программирование (самая новая среди рассматриваемых методологий) сформировалось в 1996 году. В основе методологии командная работа, эффективная коммуникация между заказчиком и исполнителем в течение всего проекта по разработке ИС, а разработка ведется с использованием последовательно дорабатываемых прототипов.

В соответствии с базовым международным стандартом ISO/IEC 12207 все процессы ЖЦ ПО делятся на три группы:

1. **Основные процессы:**
   * приобретение;
   * поставка;
   * разработка;
   * эксплуатация;
   * сопровождение.
2. **Вспомогательные процессы:**
   * документирование;
   * управление конфигурацией;
   * обеспечение качества;
   * разрешение проблем;
   * аудит;
   * аттестация;
   * совместная оценка;
   * верификация.
3. **Организационные процессы:**
   * создание инфраструктуры;
   * управление;
   * обучение;
   * усовершенствование.

Позднее был разработан и в 2002 г. опубликован стандарт на процессы жизненного цикла систем (ISO/IEC 15288 System life cycle processes). К разработке стандарта были привлечены специалисты различных областей: системной инженерии, программирования, управления качеством, человеческими ресурсами, безопасностью и пр. Был учтен практический опыт создания систем в правительственных, коммерческих, военных и академических организациях. Стандарт применим для широкого класса систем, но его основное предназначение - поддержка создания компьютеризированных систем.

Согласно стандарту ISO/IEC серии 15288 [[7]](http://www.intuit.ru/department/se/devis/popup.lit.html#7) в структуру ЖЦ следует включать следующие группы процессов:

1. **Договорные процессы:**
   * приобретение (внутренние решения или решения внешнего поставщика);
   * поставка (внутренние решения или решения внешнего поставщика).
2. **Процессы предприятия:**
   * управление окружающей средой предприятия;
   * инвестиционное управление;
   * управление ЖЦ ИС;
   * управление ресурсами;
   * управление качеством.
3. **Проектные процессы:**
   * планирование проекта;
   * оценка проекта;
   * контроль проекта;
   * управление рисками;
   * управление конфигурацией;
   * управление информационными потоками;
   * принятие решений.
4. **Технические процессы:**
   * определение требований;
   * анализ требований;
   * разработка архитектуры;
   * внедрение;
   * интеграция;
   * верификация;
   * переход;
   * аттестация;
   * эксплуатация;
   * сопровождение;
   * утилизация.
5. **Специальные процессы:**
   * определение и установка взаимосвязей исходя из задач и целей.

Стадии создания системы, предусмотренные в стандарте ISO/IEC 15288, несколько отличаются от рассмотренных выше. Перечень стадий и основные результаты, которые должны быть достигнуты к моменту их завершения, приведены в [таблице 2.2](http://www.intuit.ru/department/se/devis/2/devis_2.html#table.2.2#table.2.2).

**Таблица 2.2. Стадии создания систем (ISO/IEC 15288)**

|  |  |  |
| --- | --- | --- |
| **№ п/п** | **Стадия** | **Описание** |
| 1 | Формирование концепции | Анализ потребностей, выбор концепции и проектных решений |
| 2 | Разработка | Проектирование системы |
| 3 | Реализация | Изготовление системы |
| 4 | Эксплуатация | Ввод в эксплуатацию и использование системы |
| 5 | Поддержка | Обеспечение функционирования системы |
| 6 | Снятие с эксплуатации | Прекращение использования, демонтаж, архивирование системы |
|  |  |  |

empty

empty

**2.Каноническое проектирование ИС**

Организация канонического проектирования ИС ориентирована на использование главным образом каскадной модели жизненного цикла ИС. Стадии и этапы работы описаны в стандарте ГОСТ 34.601-90.

В зависимости от сложности объекта автоматизации и набора задач, требующих решения при создании конкретной ИС, стадии и этапы работ могут иметь различную трудоемкость. Допускается объединять последовательные этапы и даже исключать некоторые из них на любой стадии проекта. Допускается также начинать выполнение работ следующей стадии до окончания предыдущей.

**2.1 Стадии и этапы создания ИС**

Стадии и этапы создания ИС, выполняемые организациями-участниками, прописываются в договорах и *технических заданиях* на выполнение работ:

Стадия 1. **Формирование требований к ИС**.

На начальной стадии проектирования выделяют следующие этапы работ:

* *обследование* объекта и обоснование необходимости создания ИС;
* формирование требований пользователей к ИС;
* оформление отчета о выполненной работе и тактико- *технического задания* на разработку.

Стадия 2. **Разработка концепции ИС**.

* изучение объекта автоматизации;
* проведение необходимых научно-исследовательских работ;
* разработка вариантов концепции ИС, удовлетворяющих требованиям пользователей;
* оформление отчета и утверждение концепции.

Стадия 3. **Техническое задание**.

* разработка и утверждение *технического задания* на создание ИС.

Стадия 4. **Эскизный проект**.

* разработка предварительных проектных решений по системе и ее частям;
* разработка эскизной документации на ИС и ее части.

Стадия 5. **Технический проект**.

* разработка проектных решений по системе и ее частям;
* разработка документации на ИС и ее части;
* разработка и оформление документации на поставку комплектующих изделий;
* разработка заданий на проектирование в смежных частях проекта.

Стадия 6. **Рабочая документация**.

* разработка *рабочей документации* на ИС и ее части;
* разработка и адаптация программ.

Стадия 7. **Ввод в действие**.

* подготовка объекта автоматизации;
* подготовка персонала;
* комплектация ИС поставляемыми изделиями (программными и техническими средствами, программно-техническими комплексами, информационными изделиями);
* строительно-монтажные работы;
* пусконаладочные работы;
* проведение *предварительных испытаний* ;
* проведение *опытной эксплуатации* ;
* проведение *приемочных испытаний*.

Стадия 8. **Сопровождение ИС**.

* выполнение работ в соответствии с гарантийными обязательствами;
* послегарантийное обслуживание.

***Oбследование*** - это изучение и диагностический анализ организационной структуры предприятия, его деятельности и существующей системы обработки информации. Материалы, полученные в результате *обследования*, используются для:

* обоснования разработки и поэтапного внедрения систем;
* составления *технического задания* на разработку систем;
* разработки технического и рабочего проектов систем.

На этапе *обследования* целесообразно выделить две составляющие: определение стратегии внедрения ИС и детальный анализ деятельности организации.

Основная задача первого этапа *обследования* - оценка реального объема проекта, его целей и задач на основе выявленных функций и информационных элементов автоматизируемого объекта высокого уровня [[8]](http://www.intuit.ru/department/se/devis/popup.lit.html#8). Эти задачи могут быть реализованы или заказчиком ИС самостоятельно, или с привлечением консалтинговых организаций. Этап предполагает тесное взаимодействие с основными потенциальными пользователями системы и бизнес-экспертами. Основная задача взаимодействия - получить полное и однозначное понимание требований заказчика. Как правило, нужная информация может быть получена в результате интервью, бесед или семинаров с руководством, экспертами и пользователями.

По завершении этой стадии *обследования* появляется возможность определить вероятные технические подходы к созданию системы и оценить затраты на ее реализацию (затраты на аппаратное обеспечение, закупаемое программное обеспечение и разработку нового программного обеспечения ).

Результатом этапа определения стратегии является документ ( *технико-экономическое обоснование проекта* ), где четко сформулировано, что получит заказчик, если согласится финансировать проект, когда он получит готовый продукт (график выполнения работ) и сколько это будет стоить (для крупных проектов должен быть составлен график финансирования на разных этапах работ). В документе желательно отразить не только затраты, но и выгоду проекта, например время окупаемости проекта, ожидаемый экономический эффект (если его удается оценить).

Ориентировочное содержание этого документа:

* ограничения, риски, критические факторы, которые могут повлиять на успешность проекта;
* совокупность условий, при которых предполагается эксплуатировать будущую систему: архитектура системы, аппаратные и программные ресурсы, условия функционирования, обслуживающий персонал и пользователи системы;
* сроки завершения отдельных этапов, форма приемки/сдачи работ, привлекаемые ресурсы, меры по защите информации;
* описание выполняемых системой функций;
* возможности развития системы;
* информационные объекты системы;
* интерфейсы и распределение функций между человеком и системой;
* требования к программным и информационным компонентам ПО, требования к СУБД;
* что не будет реализовано в рамках проекта.

На этапе детального анализа деятельности организации изучаются задачи, обеспечивающие реализацию функций управления, организационная структура, штаты и содержание работ по управлению предприятием, а также характер подчиненности вышестоящим органам управления. На этом этапе должны быть выявлены:

* инструктивно-методические и директивные материалы, на основании которых определяются состав подсистем и перечень задач;
* возможности применения новых методов решения задач.

Аналитики собирают и фиксируют информацию в двух взаимосвязанных формах:

* функции - информация о событиях и процессах, которые происходят в бизнесе;
* сущности - информация о вещах, имеющих значение для организации и о которых что-то известно.

При изучении каждой функциональной задачи управления определяются:

* наименование задачи; сроки и периодичность ее решения;
* степень формализуемости задачи;
* источники информации, необходимые для решения задачи;
* показатели и их количественные характеристики;
* порядок корректировки информации;
* действующие алгоритмы расчета показателей и возможные методы контроля;
* действующие средства сбора, передачи и обработки информации;
* действующие средства связи;
* принятая точность решения задачи;
* трудоемкость решения задачи;
* действующие формы представления исходных данных и результатов их обработки в виде документов;
* потребители результатной информации по задаче.

Одной из наиболее трудоемких, хотя и хорошо формализуемых задач этого этапа является описание документооборота организации. При *обследовании* документооборота составляется схема маршрута движения документов, которая должна отразить:

* количество документов;
* место формирования показателей документа;
* взаимосвязь документов при их формировании;
* маршрут и длительность движения документа;
* место использования и хранения данного документа;
* внутренние и внешние информационные связи;
* объем документа в знаках.

По результатам *обследования* устанавливается перечень задач управления, решение которых целесообразно автоматизировать, и очередность их разработки.

На этапе *обследования* следует классифицировать планируемые функции системы по степени важности. Один из возможных форматов представления такой классификации - MuSCoW [[9]](http://www.intuit.ru/department/se/devis/popup.lit.html#9).

Эта аббревиатура расшифровывается так: Must have - необходимые функции; Should have - желательные функции; Could have - возможные функции; Won't have - отсутствующие функции.

Функции первой категории обеспечивают критичные для успешной работы системы возможности.

Реализация функций второй и третьей категорий ограничивается временными и финансовыми рамками: разрабатывается то, что необходимо, а также максимально возможное в порядке приоритета число функций второй и третьей категорий.

Последняя категория функций особенно важна, поскольку необходимо четко представлять границы проекта и набор функций, которые будут отсутствовать в системе.

Модели деятельности организации создаются в двух видах:

* ***модель "как есть"*** ("as-is")- отражает существующие в организации бизнес-процессы;
* ***модель "как должно быть"*** ("to-be") - отражает необходимые изменения бизнес-процессов с учетом внедрения ИС.

На этапе анализа необходимо привлекать к работе группы тестирования для решения следующих задач:

* получения сравнительных характеристик предполагаемых к использованию аппаратных платформ, операционных систем, СУБД, иного окружения;
* разработки плана работ по обеспечению надежности информационной системы и ее тестирования.

Привлечение тестировщиков на ранних этапах разработки является целесообразным для любых проектов. Если проектное решение оказалось неудачным и это обнаружено слишком поздно (на этапе разработки или, что еще хуже, на этапе внедрения в эксплуатацию), то исправление ошибки проектирования обходится очень дорого. Чем раньше группы тестирования выявляют ошибки в информационной системе, тем ниже стоимость сопровождения системы. Время на тестирование системы и на исправление обнаруженных ошибок следует предусматривать не только на этапе разработки, но и на этапе проектирования.

Для автоматизации тестирования следует использовать системы отслеживания ошибок (bug tracking). Это позволяет иметь единое хранилище ошибок, отслеживать их повторное появление, контролировать скорость и эффективность исправления ошибок, видеть наиболее нестабильные компоненты системы, а также поддерживать связь между группой разработчиков и группой тестирования (уведомления об изменениях по e-mail и т.п.). Чем больше проект, тем сильнее потребность в bug tracking.

Результаты *обследования* представляют объективную основу для формирования *технического задания* на информационную систему.

***Техническое задание*** - это документ, определяющий цели, требования и основные исходные данные, необходимые для разработки автоматизированной системы управления.

При разработке *технического задания* необходимо решить следующие задачи:

* установить общую цель создания ИС, определить состав подсистем и функциональных задач;
* разработать и обосновать требования, предъявляемые к подсистемам;
* разработать и обосновать требования, предъявляемые к информационной базе, математическому и программному обеспечению, комплексу технических средств (включая средства связи и передачи данных);
* установить общие требования к проектируемой системе;
* определить перечень задач создания системы и исполнителей;
* определить этапы создания системы и сроки их выполнения;
* провести предварительный расчет затрат на создание системы и определить уровень экономической эффективности ее внедрения.

Типовые требования к составу и содержанию *технического задания* приведены в [таблице 2.1](http://www.intuit.ru/department/se/devis/3/devis_3.html#table.3.1).

|  |  |  |
| --- | --- | --- |
| **Таблица 2.1. Состав и содержание технического задания (ГОСТ 34.602- 89)** | | |
| **№ п\п** | **Раздел** | **Содержание** |
| 1 | Общие сведения | * полное наименование системы и ее условное обозначение * шифр темы или шифр (номер) договора; * наименование предприятий разработчика и заказчика системы, их реквизиты * перечень документов, на основании которых создается ИС * плановые сроки начала и окончания работ * сведения об источниках и порядке финансирования работ * порядок оформления и предъявления заказчику результатов работ по созданию системы, ее частей и отдельных средств |
| 2 | Назначение и цели создания (развития) системы | * вид автоматизируемой деятельности * перечень объектов, на которых предполагается использование системы * наименования и требуемые значения технических, технологических, производственно-экономических и др. показателей объекта, которые должны быть достигнуты при внедрении ИС |
| 3 | Характеристика объектов автоматизации | * краткие сведения об объекте автоматизации * сведения об условиях эксплуатации и характеристиках окружающей среды |
| 4 | Требования к системе | Требования к системе в целом:   * требования к структуре и функционированию системы (перечень подсистем, уровни иерархии, степень централизации, способы информационного обмена, режимы функционирования, взаимодействие со смежными системами, перспективы развития системы) * требования к персоналу (численность пользователей, квалификация, режим работы, порядок подготовки) * показатели назначения (степень приспособляемости системы к изменениям процессов управления и значений параметров) * требования к надежности, безопасности, эргономике, транспортабельности, эксплуатации, техническому обслуживанию и ремонту, защите и сохранности информации, защите от внешних воздействий, к патентной чистоте, по стандартизации и унификации   Требования к функциям (по подсистемам) :   * перечень подлежащих автоматизации задач * временной регламент реализации каждой функции * требования к качеству реализации каждой функции, к форме представления выходной информации, характеристики точности, достоверности выдачи результатов * перечень и критерии отказов   Требования к видам обеспечения:   * математическому (состав и область применения мат. моделей и методов, типовых и разрабатываемых алгоритмов) * информационному (состав, структура и организация данных, обмен данными между компонентами системы, информационная совместимость со смежными системами, используемые классификаторы, СУБД, контроль данных и ведение информационных массивов, процедуры придания юридической силы выходным документам) * лингвистическому (языки программирования, языки взаимодействия пользователей с системой, системы кодирования, языки ввода- вывода) * программному (независимость программных средств от платформы, качество программных средств и способы его контроля, использование фондов алгоритмов и программ) * техническому * метрологическому * организационному (структура и функции эксплуатирующих подразделений, защита от ошибочных действий персонала) * методическому (состав нормативно-технической документации) |
| 5 | Состав и содержание работ по созданию системы | * перечень стадий и этапов работ * сроки исполнения * состав организаций — исполнителей работ * вид и порядок экспертизы технической документации * программа обеспечения надежности * программа метрологического обеспечения |
| 6 | Порядок контроля и приемки системы | * виды, состав, объем и методы испытаний системы * общие требования к приемке работ по стадиям * статус приемной комиссии |
| 7 | Требования к составу и содержанию работ по подготовке объекта автоматизации к вводу системы в действие | * преобразование входной информации к машиночитаемому виду * изменения в объекте автоматизации * сроки и порядок комплектования и обучения персонала |
| 8 | Требования к документированию | * перечень подлежащих разработке документов * перечень документов на машинных носителях |
| 9 | Источники разработки | документы и информационные материалы, на основании которых разрабатывается ТЗ и система |

***Эскизный проект*** предусматривает разработку предварительных проектных решений по системе и ее частям.

Выполнение стадии эскизного проектирования не является строго обязательной. Если основные проектные решения определены ранее или достаточно очевидны для конкретной ИС и объекта автоматизации, то эта стадия может быть исключена из общей последовательности работ.

Содержание *эскизного проекта* задается в ТЗ на систему. Как правило, на этапе эскизного проектирования определяются:

* функции ИС;
* функции подсистем, их цели и ожидаемый эффект от внедрения;
* состав комплексов задач и отдельных задач;
* концепция информационной базы и ее укрупненная структура;
* функции системы управления базой данных;
* состав вычислительной системы и других технических средств;
* функции и параметры основных программных средств.

По результатам проделанной работы оформляется, согласовывается и утверждается документация в объеме, необходимом для описания полной совокупности принятых проектных решений и достаточном для дальнейшего выполнения работ по созданию системы.

На основе *технического задания* (и *эскизного проекта* ) разрабатывается *технический проект* ИС. ***Технический проект*** системы - это техническая документация, содержащая общесистемные проектные решения, алгоритмы решения задач, а также оценку экономической эффективности автоматизированной системы управления и перечень мероприятий по подготовке объекта к внедрению.

На этом этапе осуществляется комплекс научно-исследовательских и экспериментальных работ для выбора основных проектных решений и расчет экономической эффективности системы.

Состав и содержание *технического проекта* приведены в [таблице 2.2](http://www.intuit.ru/department/se/devis/3/devis_3.html#table.3.2).

|  |  |  |
| --- | --- | --- |
| **Таблица 2.2. Содержание технического проекта** | | |
| **№ п\п** | **Раздел** | **Содержание** |
| 1 | Пояснительная записка | * основания для разработки системы * перечень организаций разработчиков * краткая характеристика объекта с указанием основных технико-экономических показателей его функционирования и связей с другими объектами * краткие сведения об основных проектных решениях по функциональной и обеспечивающим частям системы |
| 2 | Функциональная и организационная структура системы | * обоснование выделяемых подсистем, их перечень и назначение * перечень задач, решаемых в каждой подсистеме, с краткой характеристикой их содержания * схема информационных связей между подсистемами и между задачами в рамках каждой подсистемы |
| 3 | Постановка задач и алгоритмы решения | * организационно-экономическая сущность задачи (наименование, цель решения, краткое содержание, метод, периодичность и время решения задачи, способы сбора и передачи данных, связь задачи с другими задачами, характер использования результатов решения, в которых они используются) * экономико-математическая модель задачи (структурная и развернутая форма представления) * входная оперативная информация ( характеристика показателей, диапазон изменения, формы представления) * нормативно-справочная информация ( НСИ) (содержание и формы представления) * информация, хранимая для связи с другими задачами * информация, накапливаемая для последующих решений данной задачи * информация по внесению изменений ( система внесения изменений и перечень информации, подвергающейся изменениям) * алгоритм решения задачи ( последовательность этапов расчета, схема, расчетные формулы) * контрольный пример (набор заполненных данными форм входных документов, условные документы с накапливаемой и хранимой информацией, формы выходных документов, заполненные по результатам решения экономико-технической задачи и в соответствии с разработанным алгоритмом расчета) |
| 4 | Организация информационной базы | * источники поступления информации и способы ее передачи * совокупность показателей, используемых в системе * состав документов, сроки и периодичность их поступления * основные проектные решения по организации фонда НСИ * состав НСИ, включая перечень реквизитов, их определение, диапазон изменения и перечень документов НСИ * перечень массивов НСИ, их объем, порядок и частота корректировки информации * структура фонда НСИ с описанием связи между его элементами; требования к технологии создания и ведения фонда * методы хранения, поиска, внесения изменений и контроля * определение объемов и потоков информации НСИ * контрольный пример по внесению изменений в НСИ * предложения по унификации документации |
| 5 | Альбом форм документов |  |
| 6 | Система математического обеспечения | * обоснование структуры математического обеспечения * обоснование выбора системы программирования * перечень стандартных программ |
| 7 | Принцип построения комплекса технических средств | * описание и обоснование схемы технологического процесса обработки данных * обоснование и выбор структуры комплекса технических средств и его функциональных групп * обоснование требований к разработке нестандартного оборудования * комплекс мероприятий по обеспечению надежности функционирования технических средств |
| 8 | Расчет экономической эффективности системы | * сводная смета затрат, связанных с эксплуатацией систем * расчет годовой экономической эффективности, источниками которой являются оптимизация производственной структуры хозяйства (объединения), снижение себестоимости продукции за счет рационального использования производственных ресурсов и уменьшения потерь, улучшения принимаемых управленческих решений |
| 9 | Мероприятия по подготовке объекта к внедрению системы | * перечень организационных мероприятий по совершенствованию бизнес-процессов * перечень работ по внедрению системы, которые необходимо выполнить на стадии рабочего проектирования, с указанием сроков и ответственных лиц |
| 10 | Ведомость документов |  |

В завершение стадии технического проектирования производится разработка документации на поставку серийно выпускаемых изделий для комплектования ИС, а также определяются технические требования и составляются ТЗ на разработку изделий, не изготовляемых серийно.

На стадии " ***рабочая документация*** " осуществляется создание программного продукта и разработка всей сопровождающей документации. Документация должна содержать все необходимые и достаточные сведения для обеспечения выполнения работ по вводу ИС в действие и ее эксплуатации, а также для поддержания уровня эксплуатационных характеристик (качества) системы. Разработанная документация должна быть соответствующим образом оформлена, согласована и утверждена.

Для ИС, которые являются разновидностью автоматизированных систем, устанавливают следующие основные виды испытаний: предварительные, *опытная эксплуатация* и приемочные. При необходимости допускается дополнительно проведение других видов испытаний системы и ее частей.

В зависимости от взаимосвязей частей ИС и объекта автоматизации испытания могут быть автономные или комплексные. Автономные испытания охватывают части системы. Их проводят по мере готовности частей системы к сдаче в *опытную эксплуатацию*. Комплексные испытания проводят для групп взаимосвязанных частей или для системы в целом.

Для планирования проведения всех видов испытаний разрабатывается документ "Программа и методика испытаний". Разработчик документа устанавливается в договоре или ТЗ. В качестве приложения в документ могут включаться тесты или контрольные примеры.

***Предварительные испытания*** проводят для определения работоспособности системы и решения вопроса о возможности ее приемки в *опытную эксплуатацию*. *Предварительные испытания* следует выполнять после проведения разработчиком отладки и тестирования поставляемых программных и технических средств системы и представления им соответствующих документов об их готовности к испытаниям, а также после ознакомления персонала ИС с эксплуатационной документацией.

***Опытную эксплуатацию*** системы проводят с целью определения фактических значений количественных и качественных характеристик системы и готовности персонала к работе в условиях ее функционирования, а также определения фактической эффективности и корректировки, при необходимости, документации.

***Приемочные испытания*** проводят для определения соответствия системы *техническому заданию*, оценки качества *опытной эксплуатации* и решения вопроса о возможности приемки системы в постоянную эксплуатацию.

**2.2 Типовое проектирование ИС**

Методы *типового проектирования ИС* достаточно подробно рассмотрены в литературе [[10]](http://www.intuit.ru/department/se/devis/popup.lit.html#10). В данном курсе приведены основные определения и представлено задание для разработки проекта ИС методом *типового проектирования* (кейс "Проектирование ИС предприятия оптовой торговли лекарственными препаратами").

***Типовое проектирование ИС*** предполагает создание системы из готовых типовых элементов.Основополагающим требованием для применения методов *типового проектирования* является возможность декомпозиции проектируемой ИС на множество составляющих компонентов (подсистем, комплексов задач, программных модулей и т.д.). Для реализации выделенных компонентов выбираются имеющиеся на рынке *типовые проектные решения*, которые настраиваются на особенности конкретного предприятия.

***Типовое проектное решение (ТПР)*** - это тиражируемое (пригодное к многократному использованию) проектное решение.

Принятая классификация *ТПР* основана на уровне декомпозиции системы. Выделяются следующие классы *ТПР*:

* элементные *ТПР* - типовые решения по задаче или по отдельному виду обеспечения задачи (информационному, программному, техническому, математическому, организационному);
* подсистемные *ТПР* - в качестве элементов типизации выступают отдельные подсистемы, разработанные с учетом функциональной полноты и минимизации внешних информационных связей;
* объектные *ТПР* - типовые отраслевые проекты, которые включают полный набор функциональных и обеспечивающих подсистем ИС.

Каждое типовое решение предполагает наличие, кроме собственно функциональных элементов (программных или аппаратных), документации с детальным описанием *ТПР* и процедур настройки в соответствии с требованиями разрабатываемой системы.

Основные особенности различных классов *ТПР* приведены в [таблице 2.3](http://www.intuit.ru/department/se/devis/3/devis_3.html#table.3.3).

|  |  |  |
| --- | --- | --- |
| **Таблица 2.3. Достоинства и недостатки ТПР** | | |
| **Класс ТПР Реализация ТПР** | **Достоинства** | **Недостатки** |
| Элементные *ТПР* Библиотеки методо-ориентированных программ | * обеспечивается применение модульного подхода к проектированию и документированию ИС | * большие затраты времени на сопряжение разнородных элементов вследствие информационной, программной и технической несовместимости * большие затраты времени на доработку *ТПР* отдельных элементов |
| Подсистемные *ТПР* Пакеты прикладных программ | * достигается высокая степень интеграции элементов ИС * позволяют осуществлять: модульное проектирование; параметрическую настройку программных компонентов на различные объекты управления * обеспечивают: сокращение затрат на проектирование и программирование взаимосвязанных компонентов; хорошее документирование отображаемых процессов обработки информации | * адаптивность *ТПР* недостаточна с позиции непрерывного инжиниринга деловых процессов * возникают проблемы в комплексировании разных функциональных подсистем, особенно в случае использования решений нескольких производителей программного обеспечения |
| Объектные *ТПР* Отраслевые проекты ИС | * комплексирование всех компонентов ИС за счет методологического единства и информационной, программной и технической совместимости * открытость архитектуры — позволяет устанавливать *ТПР* на разных программно-технических платформах * масштабируемость — допускает конфигурацию ИС для переменного числа рабочих мест * конфигурируемость — позволяет выбирать необходимое подмножество компонентов | * проблемы привязки типового проекта к конкретному объекту управления, что вызывает в некоторых случаях даже необходимость изменения организационно-экономической структуры объекта автоматизации |

Для реализации *типового проектирования* используются два подхода: *параметрически-ориентированное* и ***модельно-ориентированное проектирование***.

***Параметрически-ориентированное проектирование*** включает следующие этапы: определение критериев оценки пригодности пакетов прикладных программ (ППП) для решения поставленных задач, анализ и оценка доступных ППП по сформулированным критериям, выбор и закупка наиболее подходящего пакета, настройка параметров (доработка) закупленного ППП.

Критерии оценки ППП делятся на следующие группы:

* назначение и возможности пакета;
* отличительные признаки и свойства пакета;
* требования к техническим и программным средствам;
* документация пакета;
* факторы финансового порядка;
* особенности установки пакета;
* особенности эксплуатации пакета;
* помощь поставщика по внедрению и поддержанию пакета;
* оценка качества пакета и опыт его использования;
* перспективы развития пакета.

Внутри каждой группы критериев выделяется некоторое подмножество частных показателей, детализирующих каждый из десяти выделенных аспектов анализа выбираемых ППП. Достаточно полный перечень показателей можно найти в литературе [[10]](http://www.intuit.ru/department/se/devis/popup.lit.html#10).

Числовые значения показателей для конкретных ППП устанавливаются экспертами по выбранной шкале оценок (например, 10-бальной). На их основе формируются групповые оценки и комплексная оценка пакета (путем вычисления средневзвешенных значений). Нормированные взвешивающие коэффициенты также получаются экспертным путем.

***Модельно-ориентированное проектирование*** заключается в адаптации состава и характеристик типовой ИС в соответствии с моделью объекта автоматизации.

Технология проектирования в этом случае должна обеспечивать единые средства для работы как с моделью типовой ИС, так и с моделью конкретного предприятия.

Типовая ИС в специальной базе метаинформации - репозитории - содержит модель объекта автоматизации, на основе которой осуществляется конфигурирование программного обеспечения. Таким образом, *модельно-ориентированное проектирование ИС* предполагает, прежде всего, построение модели объекта автоматизации с использованием специального программного инструментария (например, SAP Business Engineering Workbench (BEW), BAAN Enterprise Modeler). Возможно также создание системы на базе *типовой модели ИС* из репозитория, который поставляется вместе с программным продуктом и расширяется по мере накопления опыта проектирования информационных систем для различных отраслей и типов производства.

Репозиторий содержит *базовую (ссылочную) модель ИС*, *типовые (референтные) модели* определенных классов ИС, модели конкретных ИС предприятий.

***Базовая модель ИС*** в репозитории содержит описание бизнес-функций, бизнес-процессов, бизнес-объектов, бизнес-правил, организационной структуры, которые поддерживаются программными модулями типовой ИС.

***Типовые модели*** описывают конфигурации информационной системы для определенных отраслей или типов производства.

Модель конкретного предприятия строится либо путем выбора фрагментов основной или *типовой модели* в соответствии со специфическими особенностями предприятия (BAAN Enterprise Modeler), либо путем автоматизированной адаптации этих моделей в результате экспертного опроса (SAP Business Engineering Workbench).

Построенная модель предприятия в виде метаописания хранится в репозитории и при необходимости может быть откорректирована. На основе этой модели автоматически осуществляется конфигурирование и настройка информационной системы.

Бизнес-правила определяют условия корректности совместного применения различных компонентов ИС и используются для поддержания целостности создаваемой системы.

Модель бизнес-функций представляет собой иерархическую декомпозицию функциональной деятельности предприятия (подробное описание см. в разделе "Анализ и моделирование функциональной области внедрения ИС").

Модель бизнес-процессов отражает выполнение работ для функций самого нижнего уровня модели бизнес-функций (подробное описание см. в разделе "Спецификация функциональных требований к ИС"). Для отображения процессов используется модель управления событиями (ЕРС - Event-driven Process Chain). Именно модель бизнес-процессов позволяет выполнить настройку программных модулей - приложений информационной системы в соответствии с характерными особенностями конкретного предприятия.

Модели бизнес-объектов используются для интеграции приложений, поддерживающих исполнение различных бизнес-процессов (подробное описание см. в разделе "Этапы проектирования ИС с применением UML").

Модель организационной структуры предприятия представляет собой традиционную иерархическую структуру подчинения подразделений и персонала (подробное описание см. в разделе "Анализ и моделирование функциональной области внедрения ИС").

Внедрение типовой информационной системы начинается с анализа требований к конкретной ИС, которые выявляются на основе результатов предпроектного *обследования* объекта автоматизации (см. раздел "Анализ и моделирование функциональной области внедрения ИС"). Для оценки соответствия этим требованиям программных продуктов может использоваться описанная выше методика оценки ППП. После выбора программного продукта на базе имеющихся в нем референтных моделей строится предварительная модель ИС, в которой отражаются все особенности реализации ИС для конкретного предприятия. Предварительная модель является основой для выбора *типовой модели* системы и определения перечня компонентов, которые будут реализованы с использованием других программных средств или потребуют разработки с помощью имеющихся в составе типовой ИС инструментальных средств (например, ABAP в SAP, Tools в BAAN).

Реализация типового проекта предусматривает выполнение следующих операций:

* установку глобальных параметров системы;
* задание структуры объекта автоматизации;
* определение структуры основных данных;
* задание перечня реализуемых функций и процессов;
* описание интерфейсов;
* описание отчетов;
* настройку авторизации доступа;
* настройку системы архивирования.

**3.Методологии моделирования предметной области**

### 3.1Структурная модель предметной области

В основе проектирования ИС лежит моделирование предметной области. Для того чтобы получить адекватный предметной области проект ИС в виде системы правильно работающих программ, необходимо иметь целостное, системное представление модели, которое отражает все аспекты функционирования будущей информационной системы. При этом под моделью предметной области понимается некоторая система, имитирующая структуру или функционирование исследуемой предметной области и отвечающая основному требованию – быть адекватной этой области.

Предварительное моделирование предметной области позволяет сократить время и сроки проведения проектировочных работ и получить более эффективный и качественный проект. Без проведения моделирования предметной области велика вероятность допущения большого количества ошибок в решении стратегических вопросов, приводящих к экономическим потерям и высоким затратам на последующее перепроектирование системы. Вследствие этого все современные технологии проектирования ИС основываются на использовании методологии моделирования предметной области.

К моделям предметных областей предъявляются следующие требования:

* формализация, обеспечивающая однозначное описание структуры предметной области;
* понятность для заказчиков и разработчиков на основе применения графических средств отображения модели;
* реализуемость, подразумевающая наличие средств физической реализации модели предметной области в ИС;
* обеспечение оценки эффективности реализации модели предметной области на основе определенных методов и вычисляемых показателей.

Для реализации перечисленных требований, как правило, строится **система моделей**, которая отражает структурный и оценочный аспекты функционирования предметной области.

**Структурный аспект** предполагает построение:

* объектной структуры, отражающей состав взаимодействующих в процессах материальных и информационных объектов предметной области;
* функциональной структуры, отражающей взаимосвязь функций (действий) по преобразованию объектов в процессах;
* структуры управления, отражающей события и бизнес-правила, которые воздействуют на выполнение процессов;
* организационной структуры, отражающей взаимодействие организационных единиц предприятия и персонала в процессах;
* технической структуры, описывающей топологию расположения и способы коммуникации комплекса технических средств.

Для отображения структурного аспекта моделей предметных областей в основном используются графические методы, которые должны гарантировать представление информации о компонентах системы. Главное требование к графическим методам документирования — простота. Графические методы должны обеспечивать возможность структурной декомпозиции спецификаций системы с максимальной степенью детализации и согласований описаний на смежных уровнях декомпозиции.

С моделированием непосредственно связана проблема **выбора языка** представления проектных решений, позволяющего как можно больше привлекать будущих пользователей системы к ее разработке. Язык моделирования – это нотация, в основном графическая, которая используется для описания проектов. Нотация представляет собой совокупность графических объектов, используемых в модели. Нотация является синтаксисом языка моделирования . Язык моделирования, с одной стороны, должен делать решения проектировщиков понятными пользователю, с другой стороны, предоставлять проектировщикам средства достаточно формализованного и однозначного определения проектных решений, подлежащих реализации в виде программных комплексов, образующих целостную систему программного обеспечения.

Графическое изображение нередко оказывается наиболее емкой формой представления информации. При этом проектировщики должны учитывать, что графические методы документирования не могут полностью обеспечить декомпозицию проектных решений от постановки задачи проектирования до реализации программ ЭВМ. Трудности возникают при переходе от этапа анализа системы к этапу проектирования и в особенности к программированию.

Главный **критерий адекватности структурной модели** предметной области заключается в функциональной полноте разрабатываемой ИС.

**Оценочные аспекты** моделирования предметной области связаны с разрабатываемыми показателями эффективности автоматизируемых процессов, к которым относятся:

* время решения задач;
* стоимостные затраты на обработку данных;
* надежность процессов;
* косвенные показатели эффективности, такие, как объемы производства, производительность труда, оборачиваемость капитала, рентабельность и т.д.

Для расчета показателей эффективности, как правило, используются статические методы функционально-стоимостного анализа (ABC) и динамические методы имитационного моделирования.

В основе различных методологий моделирования предметной области ИС лежат принципы последовательной детализации абстрактных категорий. Обычно модели строятся на трех уровнях: на внешнем уровне ( **определении требований** ), на концептуальном уровне ( **спецификации требований** ) и внутреннем уровне ( **реализации требований** ). Так, на внешнем уровне модель отвечает на вопрос, что должна делать система, то есть определяется состав основных компонентов системы: объектов, функций, событий, организационных единиц, технических средств. **На концептуальном уровне** модель отвечает на вопрос, как должна функционировать система? Иначе говоря, определяется характер взаимодействия компонентов системы одного и разных типов. На внутреннем уровне модель отвечает на вопрос: с помощью каких программно-технических средств реализуются требования к системе? С позиции жизненного цикла ИС описанные уровни моделей соответственно строятся на этапах анализа требований, логического (технического) и физического (рабочего) проектирования. Рассмотрим особенности построения моделей предметной области на трех уровнях детализации.

#### Объектная структура

Объект — это сущность, которая используется при выполнении некоторой функции или операции (преобразования, обработки, формирования и т.д.). Объекты могут иметь динамическую или статическую природу: динамические объекты используются в одном цикле воспроизводства, например заказы на продукцию, счета на оплату, платежи; статические объекты используются во многих циклах воспроизводства, например, оборудование, персонал, запасы материалов.

**На внешнем уровне** детализации модели выделяются основные виды материальных объектов (например, сырье и материалы, полуфабрикаты, готовые изделия, услуги) и основные виды информационных объектов или документов (например, заказы, накладные, счета и т.д.).

**На концептуальном уровне** построения модели предметной области уточняется состав классов объектов, определяются их атрибуты и взаимосвязи. Таким образом строится обобщенное представление структуры предметной области.

Далее концептуальная модель на внутреннем уровне отображается в виде файлов базы данных, входных и выходных документов ЭИС. Причем динамические объекты представляются единицами переменной информации или документами, а статические объекты — единицами условно-постоянной информации в виде списков, номенклатур, ценников, справочников, классификаторов. Модель базы данных как постоянно поддерживаемого информационного ресурса отображает хранение условно-постоянной и накапливаемой переменной информации, используемой в повторяющихся информационных процессах.

#### Функциональная структура

Функция ( операция ) представляет собой некоторый преобразователь входных объектов в выходные. Последовательность взаимосвязанных по входам и выходам функций составляет бизнес-процесс. Функция бизнес-процесса может порождать объекты любой природы (материальные, денежные, информационные). Причем бизнес-процессы и информационные процессы, как правило, неразрывны, то есть функции материального процесса не могут осуществляться без информационной поддержки. Например, отгрузка готовой продукции осуществляется на основе документа "Заказ", который, в свою очередь, порождает документ "Накладная", сопровождающий партию отгруженного товара.

Функция может быть представлена одним действием или некоторой совокупностью действий. В последнем случае каждой функции может соответствовать некоторый процесс, в котором могут существовать свои подпроцессы, и т.д., пока каждая из подфункций не будет представлять некоторую недекомпозируемую последовательность действий.

**На внешнем уровне** моделирования определяется список основных бизнес-функций или видов бизнес-процессов. Обычно таких функций насчитывается 15–20.

**На концептуальном уровне** выделенные функции декомпозируются и строятся иерархии взаимосвязанных функций.

**На внутреннем уровне** отображается структура информационного процесса в компьютере: определяются иерархические структуры программных модулей, реализующих автоматизируемые функции.

#### Структура управления

В совокупности функций бизнес-процесса возможны альтернативные или циклические последовательности в зависимости от различных условий протекания процесса. Эти условия связаны с происходящими событиями во внешней среде или в самих процессах и с образованием определенных состояний объектов (например, заказ принят, отвергнут, отправлен на корректировку). **События** вызывают выполнение функций, которые, в свою очередь, изменяют состояния объектов и формируют новые события, и т.д., пока не будет завершен некоторый бизнес-процесс. Тогда последовательность событий составляет конкретную реализацию бизнес-процесса.

Каждое событие описывается с двух точек зрения: **информационной** и **процедурной**. Информационно событие отражается в виде некоторого сообщения, фиксирующего факт выполнения некоторой функции изменения состояния или появления нового. Процедурно событие вызывает выполнение новой функции, и поэтому для каждого состояния объекта должны быть заданы описания этих вызовов. Таким образом, события выступают в связующей роли для выполнения функций бизнес-процессов.

**На внешнем уровне** определяются список внешних событий, вызываемых взаимодействием предприятия с внешней средой (платежи налогов, процентов по кредитам, поставки по контрактам и т.д.), и список целевых установок, которым должны соответствовать бизнес-процессы (регламент выполнения процессов, поддержка уровня материальных запасов, уровень качества продукции и т.д.).

**На концептуальном уровне** устанавливаются бизнес-правила, определяющие условия вызова функций при возникновении событий и достижении состояний объектов.

**На внутреннем уровне** выполняется формализация бизнес-правил в виде триггеров или вызовов программных модулей.

#### Организационная структура

Организационная структура представляет собой совокупность организационных единиц, как правило, связанных иерархическими и процессными отношениями. Организационная единица — это подразделение, представляющее собой объединение людей (персонала) для выполнения совокупности общих функций или бизнес-процессов. В функционально-ориентированной организационной структуре организационная единица выполняет набор функций, относящихся к одной функции управления и входящих в различные процессы. В процессно-ориентированной структуре организационная единица выполняет набор функций, входящих в один тип процесса и относящихся к разным функциям управления.

**На внешнем уровне** строится структурная модель предприятия в виде иерархии подчинения организационных единиц или списков взаимодействующих подразделений.

**На концептуальном уровне** для каждого подразделения задается организационно-штатная структура должностей (ролей персонала).

**На внутреннем уровне** определяются требования к правам доступа персонала к автоматизируемым функциям информационной системы.

#### Техническая структура

Топология определяет территориальное размещение технических средств по структурным подразделениям предприятия, а коммуникация — технический способ реализации взаимодействия структурных подразделений.

**На внешнем уровне** модели определяются типы технических средств обработки данных и их размещение по структурным подразделениям.

**На концептуальном уровне** определяются способы коммуникаций между техническими комплексами структурных подразделений: физическое перемещение документов, машинных носителей, обмен информацией по каналам связи и т.д.

**На внутреннем уровне** строится модель "клиент-серверной" архитектуры вычислительной сети.

Описанные модели предметной области нацелены на проектирование отдельных компонентов ИС: данных, функциональных программных модулей, управляющих программных модулей, программных модулей интерфейсов пользователей, структуры технического комплекса. Для более качественного проектирования указанных компонентов требуется построение моделей, увязывающих различные компоненты ИС между собой. В простейшем случае в качестве таких моделей взаимодействия могут использоваться матрицы перекрестных ссылок: "объекты-функции", "функции-события", "организационные единицы — функции ", "организационные единицы — объекты", "организационные единицы — технические средства" и т д. Такие матрицы не наглядны и не отражают особенности реализации взаимодействий.

Для правильного отображения взаимодействий компонентов ИС важно осуществлять совместное моделирование таких компонентов, особенно с содержательной точки зрения объектов и функций. Методология структурного системного анализа существенно помогает в решении таких задач.

Структурным анализом принято называть метод исследования системы, который начинается с ее общего обзора, а затем детализируется, приобретая иерархическую структуру с все большим числом уровней. Для таких методов характерно: разбиение на уровни абстракции с ограниченным числом элементов (от 3 до 7); ограниченный контекст, включающий только существенные детали каждого уровня; использование строгих формальных правил записи; последовательное приближение к результату. Структурный анализ основан на двух базовых принципах – "разделяй и властвуй" и принципе иерархической упорядоченности. Решение трудных проблем путем их разбиения на множество меньших независимых задач (так называемых "черных ящиков") и организация этих задач в древовидные иерархические структуры значительно повышают понимание сложных систем. Определим ключевые понятия структурного анализа.

Операция – элементарное (неделимое) действие, выполняемое на одном рабочем месте.

Функция – совокупность операций, сгруппированных по определенному признаку.

Бизнес-процесс — связанная совокупность функций, в ходе выполнения которой потребляются определенные ресурсы и создается продукт (предмет, услуга, научное открытие, идея), представляющая ценность для потребителя.

Подпроцесс – это бизнес-процесс, являющийся структурным элементом некоторого бизнес-процесса и представляющий ценность для потребителя.

Бизнес-модель – структурированное графическое описание сети процессов и операций, связанных с данными, документами, организационными единицами и прочими объектами, отражающими существующую или предполагаемую деятельность предприятия.

Существуют различные методологии структурного моделирования предметной области, среди которых следует выделить **функционально-ориентированные и объектно-ориентированные методологии**.

### 3.2 Функционально-ориентированные(структурные) и объектно-ориентированные методологии описания предметной области

Процесс бизнес-моделирования может быть реализован в рамках различных методик, отличающихся прежде всего своим подходом к тому, что представляет собой моделируемая организация. В соответствии с различными представлениями об организации методики принято делить на объектные и функциональные (структурные).

Объектные методики рассматривают моделируемую организацию как набор взаимодействующих объектов – производственных единиц. Объект определяется как осязаемая реальность – предмет или явление, имеющие четко определяемое поведение. Целью применения данной методики является выделение объектов, составляющих организацию, и распределение между ними ответственностей за выполняемые действия.

Функциональные методики, наиболее известной из которых является методика IDEF, рассматривают организацию как набор функций, преобразующий поступающий поток информации в выходной поток. Процесс преобразования информации потребляет определенные ресурсы. Основное отличие от объектной методики заключается в четком отделении функций (методов обработки данных) от самих данных.

С точки зрения бизнес-моделирования каждый из представленных подходов обладает своими преимуществами. Объектный подход позволяет построить более устойчивую к изменениям систему, лучше соответствует существующим структурам организации. Функциональное моделирование хорошо показывает себя в тех случаях, когда организационная структура находится в процессе изменения или вообще слабо оформлена. Подход от выполняемых функций интуитивно лучше понимается исполнителями при получении от них информации об их текущей работе.

### 3.3Семейство стандартов и методов IDEF

В создании информационной системы принимает участие большой коллектив специалистов, работающих в разных областях. Прежде всего, сама фирма пытается упростить себе жизнь, привлекая собственных экспертов, далее сотрудники других организаций берут на себя обязанности в тех направлениях, которые в собственной организации не представлены.

И существенным моментом является выработка языка, на котором будут общаться программисты, бухгалтера, директора, начальники и т.д. Таким «языком» можно смело назвать методы **IDEF** и аналогичные, построенные на тех же принципах.

Развитие методик было начато в 1945 г. Ludwig von Bertallanfly. В основе лежат следующие постулаты: **любая система есть совокупность систем, система всегда имеет некоторые границы, система преобразует входы в выходы используя механизмы, у системы есть управляющее воздействие, описание системы зависит от цели, области и точки зрения**.

Собственно IDEF есть некоторое формализованное описание разложения изучаемой системы на подсистемы (**декомпозиция**) в зависимости от области применения, состава экспертной группы и назначений. Техника моделирования функциональной декомпозиции была развита в 60-х годах Douglas T.Ross для улучшения производства. Первым был стандарт IDEF0 (называвшийся в начале SADT), упрощенная версия которого принята ВВС США в 70-х годах. Использование методик моделирования бизнес процессов дало хорошие результаты в процессе реорганизации производства и управления. В дальнейшем, используя методологию проектирования, IDEF развился и продолжает развиваться. Описание наиболее распространенных стандартов можно найти на [www.idef.com](http://www.idef.com).

IDEF — методологии семейства [ICAM](http://ru.wikipedia.org/w/index.php?title=ICAM&action=edit&redlink=1) (Integrated Computer-Aided Manufacturing) для решения задач [моделирования](http://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D0%B5%D0%BB%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) [сложных систем](http://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D0%BE%D0%B6%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), позволяет отображать и анализировать модели деятельности широкого спектра сложных систем в различных разрезах. При этом широта и глубина обследования процессов в системе определяется самим разработчиком, что позволяет не перегружать создаваемую модель излишними данными.

IDEF — методологии создавались в рамках предложенной [ВВС США](http://ru.wikipedia.org/wiki/%D0%92%D0%92%D0%A1_%D0%A1%D0%A8%D0%90) программы компьютеризации промышленности — ICAM, в ходе реализации которой выявилась потребность в разработке методов анализа процессов взаимодействия в производственных (промышленных) системах. Принципиальным требованием при разработке рассматриваемого семейства методологий была возможность эффективного обмена информацией между всеми специалистами — участниками программы ICAM (отсюда название: Icam DEFinition — IDEF другой вариант — Integrated DEFinition). После опубликования стандарта он был успешно применен в самых различных областях бизнеса, показав себя эффективным средством анализа, конструирования и отображения [бизнес-процессов](http://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%B7%D0%BD%D0%B5%D1%81-%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81) (к слову сказать, он активно применяется и в российских [госструктурах](http://ru.wikipedia.org/w/index.php?title=%D0%93%D0%BE%D1%81%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0&action=edit&redlink=1), например в [Государственной Налоговой Инспекции](http://ru.wikipedia.org/w/index.php?title=%D0%93%D0%BE%D1%81%D1%83%D0%B4%D0%B0%D1%80%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F_%D0%9D%D0%B0%D0%BB%D0%BE%D0%B3%D0%BE%D0%B2%D0%B0%D1%8F_%D0%98%D0%BD%D1%81%D0%BF%D0%B5%D0%BA%D1%86%D0%B8%D1%8F&action=edit&redlink=1)). Более того, собственно с широким применением IDEF (и предшествующей методолoгии — SADT) и связано возникновение основных идей популярного ныне понятия — [BPR](http://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B8%D0%BD%D0%B6%D0%B8%D0%BD%D0%B8%D1%80%D0%B8%D0%BD%D0%B3_%D0%B1%D0%B8%D0%B7%D0%BD%D0%B5%D1%81-%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D0%B2) ([бизнес-процесс](http://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%B7%D0%BD%D0%B5%D1%81-%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81) [реинжиниринг](http://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%80%D0%B0%D1%82%D0%BD%D0%B0%D1%8F_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0)).

IDEF – Integration Definition – особенно полезны они оказались при создании информационных систем управления, поскольку позволяют «найти» общий язык между людьми различных специальностей, договорится о том что и как будет делаться, позволяют выявить информационные потоки и взаимодействия, в формализованном виде хорошо представить достаточно сложные системы. Используя различные стандарты можно построить модели любых систем управления с той или иной степенью точности.

В некотором смысле процесс построения модели подобен построению иерархии сущностей, которые определяются конечной целью, и набором правил в соответствии, с которыми строится модель. При этом основными логическими (и соответственно графическими) объектами является понятие сущности и связи между сущностями (геометрические фигуры и стрелки между ними). Для различных методов различными являются категории сущностей и связей между ними. Следующим важным моментом для построения модели является свод правил, по которым они строятся, т.е. что с чем и каким образом может быть связано, т.е. формализации логики работы или управления, опять-таки своеобразны для каждой методики.

Внешними атрибутами являются словари терминов, которыми оперируют эксперты, детальные описания что под чем понимается и чью точку зрения отражают. Важным моментом является система **оценок и правил их сверток** (ABC анализ), что позволяет на выходе моделей получать некоторые весьма важные количественные показатели.

Необходимо также иметь в виду, что не существует «абсолютной» модели, т.е. которая полностью описывает деятельность какого-либо предприятия. Любое моделирование происходит на уровне описания одной из сторон деятельности или управления (например: бухгалтерский учет, документооборот, технологический цикл и т.п.). Да и задачи, решаемые в результате моделирования, совершенно различны: в одном случае это анализ технологий фирмы на предмет их улучшения, в другом случае – структурная реорганизация предприятия, в третьем – создание информационной системы управления, и т.д. Конечно, в некотором смысле такие модели подобны – более того, взаимосвязаны. Более того, чтобы построить наиболее функциональную систему автоматизации, необходимо построить достаточное количество подобных моделей.

Как правило, методы IDEF устанавливают стандарт описания деятельности человека в соответствии с некоторым мировоззрением. Рассмотрим некоторые из них.

Прежде всего, постараемся дать определение, что же такое метод и что позволяет достичь методики моделирования.

Непротиворечивое, надежное создание правильных архитектурных представлений, независимо от того, были ли искусственные аппроксимации системы или описательные представления, требует использования методического руководства. Это наблюдение доказывает потребность во многом "архитектурных представлений" и, соответственно, многих методов.

Методы, и их связанные архитектурные представления, сосредотачиваются на ограниченном наборе характеристик системы и явно игнорируют те, которые непосредственно не участвуют в принятии решении. Методы не были предназначены, чтобы оценивать и представлять любое возможное состояние или поведенческую характеристику системы при анализе.

Изобилие методов специального назначения, которые обычно предназначены для узкого круга экспертов, требует наличия механизмов интеграции с другими методами.

IDEF семейство методов предназначено, чтобы создать равновесие между методами специального назначения, чья эффективность ограничена спецификой применения, и "первоклассными методами", которые пытаются включить все, что может быть когда-либо необходимым.

Это равновесие поддерживается внутри IDEF методов, обеспечивая явные механизмы для интегрирования.

**IDEF0 (Business Process)**

Метод IDEF0 предназначен для моделирования организации бизнес процессов предприятия. Основными понятиями являются «работа» и «стрелки». «Работа» есть поименованный процесс, функция или задача, которая происходит в течение некоторого периода времени и производит ощутимые результаты.

«Стрелки» используются для представления направления или потока объектов или данных, показывают необходимые объекты, используемые или создаваемые работой, должны входить или покидать работу в одной точке. Каждая стрелка должна представлять только одну классификацию или категорию, если в модели не оговорено другое. «Стрелки» бывают следующих типов: ВХОД, УПРАВЛЕНИЕ, ВЫХОД, МЕХАНИЗМ, ВЫЗОВ.

Стрелка входа представляет информацию о материале, который используется или потребляется работой, изменяется работой, запускает эту работу.

Стрелка управления представляет информацию о материале, который регулирует или направляет работу, стратегии, процедуры, определяет, когда и как или будет ли выполняться работа, определяет, какие выходы производятся.

Стрелка выхода представляет информацию о материале, который создается или производится работой. Обычно – это причина, по которой работа выполняется.

Рис. 1 Простейшая диаграмма IDEF0

Стрелка механизма представляет лицо(а) или машину(ы), которые выполняют работу или поддерживают работу.



Стрелка вызова представляет особый случай механизма, который индицирует, что внешний процесс выполняет работу.

Декомпозиция диаграмм заключается в разбиении работ на дочерние (как правило, в пределах 3-6 дочерних работ).

**IDEF3 (Workflow)**

IDEF3 наиболее подходит для описания логики взаимодействия информационных потоков. Диаграммы Workflow могут быть использованы в моделировании бизнес процессов для анализа завершенности процедур обработки информации. С их помощью можно описывать сценарии действий сотрудников организаций, например, последовательность обработки заказа или события, которые необходимо обработать за конечное время. Каждый сценарий сопровождается описанием процесса и может быть использован для документирования каждой функции. В IDEF3 стрелки могут расщепляться или сливаться только через «перекрестки», которые отображают временную логику событий. Существует пять типов перекрестков: синхронный и асинхронный «и», синхронный и асинхронный «или», исключающий «или». Более того, стрелки могут связывать единицы работ, быть отношениями или потоками объектов.

IDEF3 достаточно хорошо описывает системы, где существенным моментом являются временные сдвиги (например в области принятия решений), либо хорошо формализуется логика технологических процессов.

**DFD (Dataflow)**

Общие принципы построения модели в методологии DFD сходны с IDEF0. Модель представляет совокупность иерархически зависимых диаграмм, прямоугольники изображают работы или процессы, стрелки – это то же некоторые данные. Построение модели осуществляется сверху вниз путем проведения декомпозиции крупных работ на мелкие.

Диаграммы потоков данных (DFD) используются для описания документооборота и обработки информации. Их можно использовать как дополнение к модели IDEF0 для более наглядного отображения текущих операций документооборота в корпоративных системах обработки информации. DFD описывают функции обработки информации (работы), документы (стрелки), объекты, сотрудников или отделы, которые участвуют в обработке информации (внешние ссылки) и таблицы для хранения документов (хранилища данных). В отличии от IDEF0 для стрелок нет понятия вход, выход, управление или механизм и неважно, в какую грань работы они входят или из какой грани выходят.

**HFD (Hierarchy Flow Diagram)**

HFD предназначен, в первую очередь, для построения иерархии управления. Основными понятиями являются узлы уровней административного управления предприятия. Тем не менее, основная методология декомпозиции сохраняется. Декомпозиция осуществляется по уровням управления, начиная с самого верхнего. Стрелки между уровнями являются правилами делегирования принятия решения. Важным понятием также является понятие задачи и связей между узловыми точками и задачами. В некотором приближении концепцию построения иерархии задач можно рассматривать как декомпозицию общих задач на более мелкие и только для систем административного управления.

**ABC (activity based costing) анализ.**

ABC анализ - расчет себестоимости, базирующийся на понятии стоимости работы, некое соглашение об учете, используемое для сбора затрат, связанных с работами, с целью определить общую стоимость процесса. ABC необходим для того, чтобы понять происхождение выходных затрат, облегчить объектный выбор работ бизнеса в правильной попытке улучшения технологического процесса (business process improvement - BPI) или реконструкции процессов (business process re-engineering - BPR). ABC анализ заключается в следующем: для конкретной модели назначается некоторая совокупность разноплановых характеристик (временных, финансовых, весовых и т.д.). Каждая сущность имеет конкретные значения характеристик, причем при переходе от нижних уровней к верхним определены правила расчетов, т.е. правила свертки.

ABC анализ помогает оценить стоимостные, временные и иные характеристики построенных моделей, соответственно бизнес процессов предприятия. Решение обратной задачи, перехода от фактических показателей к желаемым и построения соответствующей модели (“to be”), позволит значительно повысить эффективность технологических процессов и системы управления.

Обязательным общим для всех стандартов является понятие словаря терминов, который крайне необходим для однозначного понимания описываемых процессов.

Успех корпорации зависит от ее способности эффективно использовать информационные потоки и правильно организовывать бизнес процесс, что провоцирует крупные инвестиции в информационные технологии.

Однако, наряду с преимуществами использования информационных технологий, появились новые проблемы: возрастающие затраты на техническое обслуживание, устарелые и негибкие системы передачи и обработки данных, рабочие проекты создания программного обеспечения, которые будут не в состоянии завершиться вовремя и под выделенный бюджет.

Если не выполняются соответствующий анализ и планирование, то корпорации не смогут получить полное значение из их вклада в информационные технологии. Они могут также попасть в ловушку зацикливания постоянного изобретение системы, поскольку предыдущие решения не соответствуют требования пользователей, либо технологии устарели со временем, либо не правильно спроектирована система.

Эффективный способ избежать такого зацикливания через анализ объединенной системы, через создание информационного проекта и модели, через реализацию по методикам типа объектно-ориентированного подхода, что позволяет контролировать и управлять полным эксплуатационным циклом информационных объектов в доводке систем программного обеспечения. Это обеспечивает трассируемость практически всех компонентов программного обеспечения вплоть до исходных текстов, отдельных составляющих проекта, требований, и объектов в прикладной области.

**IDEF** — методологии семейства [ICAM](http://ru.wikipedia.org/w/index.php?title=ICAM&action=edit&redlink=1) (Integrated Computer-Aided Manufacturing) для решения задач [моделирования](http://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D0%B5%D0%BB%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) [сложных систем](http://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D0%BE%D0%B6%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), позволяет отображать и анализировать модели деятельности широкого спектра сложных систем в различных разрезах. При этом широта и глубина обследования процессов в системе определяется самим разработчиком, что позволяет не перегружать создаваемую модель излишними данными.

В настоящий момент к семейству IDEF можно отнести следующие стандарты:

* [IDEF0](http://ru.wikipedia.org/wiki/IDEF0) — Function Modeling — методология функционального моделирования. С помощью наглядного графического языка IDEF0 изучаемая система предстает перед разработчиками и аналитиками в виде набора взаимосвязанных функций (функциональных блоков — в терминах IDEF0). Как правило, моделирование средствами IDEF0 является первым этапом изучения любой системы. Методологию IDEF0 можно считать следующим этапом развития хорошо известного графического языка описания функциональных систем [SADT](http://ru.wikipedia.org/wiki/SADT) (Structured Analysis and Design Technique);
* [IDEF1](http://ru.wikipedia.org/wiki/IDEF1) — Information Modeling — методология моделирования информационных потоков внутри системы, позволяющая отображать и анализировать их структуру и взаимосвязи;
* [IDEF1X](http://ru.wikipedia.org/w/index.php?title=IDEF1X&action=edit&redlink=1) (IDEF1 Extended) — Data Modeling — методология построения реляционных структур (баз данных), относится к типу методологий «Сущность-взаимосвязь» ([ER — Entity-Relationship](http://ru.wikipedia.org/wiki/ER-%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85)) и, как правило, используется для моделирования реляционных баз данных, имеющих отношение к рассматриваемой системе;
* [IDEF2](http://ru.wikipedia.org/w/index.php?title=IDEF2&action=edit&redlink=1) — Simulation Model Design — методология динамического моделирования развития систем. В связи с весьма серьезными сложностями анализа динамических систем от этого стандарта практически отказались, и его развитие приостановилось на самом начальном этапе. В настоящее время присутствуют алгоритмы и их компьютерные реализации, позволяющие превращать набор статических диаграмм IDEF0 в динамические модели, построенные на базе «раскрашенных сетей Петри» (CPN — Color Petri Nets);
* [IDEF3](http://ru.wikipedia.org/wiki/IDEF3) — Process Description Capture — Документирование технологических процессов,

IDEF3 — методология документирования процессов, происходящих в системе (например, на предприятии), описываются сценарий и последовательность операций для каждого процесса. IDEF3 имеет прямую взаимосвязь с методологией IDEF0 — каждая функция (функциональный блок) может быть представлена в виде отдельного процесса средствами IDEF3;

* [IDEF4](http://ru.wikipedia.org/w/index.php?title=IDEF4&action=edit&redlink=1) — Object-Oriented Design — методология построения объектно-ориентированных систем, позволяют отображать структуру объектов и заложенные принципы их взаимодействия, тем самым позволяя анализировать и оптимизировать сложные объектно-ориентированные системы;
* [IDEF5](http://ru.wikipedia.org/w/index.php?title=IDEF5&action=edit&redlink=1) — Ontology Description Capture — Стандарт онтологического исследования сложных систем. С помощью методологии IDEF5 [онтология](http://ru.wikipedia.org/wiki/%D0%9E%D0%BD%D1%82%D0%BE%D0%BB%D0%BE%D0%B3%D0%B8%D1%8F) системы может быть описана при помощи определенного словаря терминов и правил, на основании которых могут быть сформированы достоверные утверждения о состоянии рассматриваемой системы в некоторый момент времени. На основе этих утверждений формируются выводы о дальнейшем развитии системы и производится её оптимизация;
* [IDEF6](http://ru.wikipedia.org/w/index.php?title=IDEF6&action=edit&redlink=1) — Design Rationale Capture — Обоснование проектных действий. Назначение IDEF6 состоит в облегчении получения «знаний о способе» моделирования, их представления и использования при разработке систем управления предприятиями. Под «знаниями о способе» понимаются причины, обстоятельства, скрытые мотивы, которые обуславливают выбранные методы моделирования. Проще говоря, «знания о способе» интерпретируются как ответ на вопрос: «почему модель получилась такой, какой получилась?» Большинство методов моделирования фокусируются на собственно получаемых моделях, а не на процессе их создания. Метод IDEF6 акцентирует внимание именно на процессе создания модели;
* [IDEF7](http://ru.wikipedia.org/w/index.php?title=IDEF7&action=edit&redlink=1) — Information System Auditing — Аудит информационных систем. Этот метод определён как востребованный, однако так и не был полностью разработан;
* [IDEF8](http://ru.wikipedia.org/w/index.php?title=IDEF8&action=edit&redlink=1) — User Interface Modeling — Метод разработки интерфейсов взаимодействия оператора и системы (пользовательских интерфейсов). Современные среды разработки пользовательских интерфейсов в большей степени создают внешний вид интерфейса. IDFE8 фокусирует внимание разработчиков интерфейса на программировании желаемого взаимного поведения интерфейса и пользователя на трех уровнях: выполняемой операции (что это за операция); сценарии взаимодействия, определяемом специфической ролью пользователя (по какому сценарию она должна выполняться тем или иным пользователем); и, наконец, на деталях интерфейса (какие элементы управления, предлагает интерфейс для выполнения операции);
* [IDEF9](http://ru.wikipedia.org/w/index.php?title=IDEF9&action=edit&redlink=1) — Scenario-Driven IS Design (Business Constraint Discovery method) — Метод исследования бизнес ограничений был разработан для облегчения обнаружения и анализа ограничений в условиях которых действует предприятие. Обычно, при построении моделей описанию ограничений, оказывающих влияние на протекание процессов на предприятии уделяется недостаточное внимание. Знания об основных ограничениях и характере их влияния, закладываемые в модели, в лучшем случае остаются неполными, несогласованными, распределенными нерационально, но часто их вовсе нет. Это не обязательно приводит к тому, что построенные модели нежизнеспособны, просто их реализация столкнется с непредвиденными трудностями, в результате чего их потенциал будет не реализован. Тем не менее в случаях, когда речь идет именно о совершенствовании структур или адаптации к предсказываемым изменениям, знания о существующих ограничениях имеют критическое значение;
* [IDEF10](http://ru.wikipedia.org/w/index.php?title=IDEF10&action=edit&redlink=1) — Implementation Architecture Modeling — Моделирование архитектуры выполнения. Этот метод определён как востребованный, однако так и не был полностью разработан;
* [IDEF11](http://ru.wikipedia.org/w/index.php?title=IDEF11&action=edit&redlink=1) — Information Artifact Modeling. Этот метод определён как востребованный, однако так и не был полностью разработан;
* [IDEF12](http://ru.wikipedia.org/w/index.php?title=IDEF12&action=edit&redlink=1) — Organization Modeling — Организационное моделирование. Этот метод определён как востребованный, однако так и не был полностью разработан;
* [IDEF13](http://ru.wikipedia.org/w/index.php?title=IDEF13&action=edit&redlink=1) — Three Schema Mapping Design — Трёхсхемное проектирование преобразования данных. Этот метод определён как востребованный, однако так и не был полностью разработан;
* [IDEF14](http://ru.wikipedia.org/w/index.php?title=IDEF14&action=edit&redlink=1) — Network Design — Метод проектирования компьютерных сетей, основанный на анализе требований, специфических сетевых компонентов, существующих конфигураций сетей. Также он обеспечивает поддержку решений, связанных с рациональным управлением материальными ресурсами, что позволяет достичь существенной экономии.

#### 3.4 Объектно-ориентированная методика

Принципиальное отличие между функциональным и объектным подходом заключается в способе декомпозиции системы. Объектно-ориентированный подход использует объектную декомпозицию, при этом статическая структура описывается в терминах **объектов и связей** между ними, а поведение системы описывается в терминах **обмена сообщениями** между объектами. Целью методики является построение бизнес-модели организации, позволяющей перейти от модели сценариев использования к модели, определяющей отдельные объекты, участвующие в реализации бизнес-функций.

Концептуальной основой объектно-ориентированного подхода является объектная модель, которая строится с учетом следующих принципов:

* абстрагирование;
* инкапсуляция;
* модульность;
* иерархия;
* типизация;
* параллелизм;
* устойчивость.

Основными понятиями объектно-ориентированного подхода являются объект и класс.

**Объект — предмет или явление, имеющее четко определенное поведение и обладающие состоянием, поведением и индивидуальностью**. Структура и поведение схожих объектов определяют общий для них класс. **Класс – это множество объектов, связанных общностью структуры и поведения**. Следующую группу важных понятий объектного подхода составляют наследование и полиморфизм. Понятие **полиморфизм** может быть интерпретировано как способность класса принадлежать более чем одному типу. **Наследование** означает построение новых классов на основе существующих с возможностью добавления или переопределения данных и методов.

Важным качеством объектного подхода является согласованность моделей деятельности организации и моделей проектируемой информационной системы от стадии формирования требований до стадии реализации. По объектным моделям может быть прослежено отображение реальных сущностей моделируемой предметной области (организации) в объекты и классы информационной системы.

Большинство существующих методов объектно-ориентированного подхода включают язык моделирования и описание процесса моделирования. **Процесс** – это описание шагов, которые необходимо выполнить при разработке проекта. В качестве языка моделирования объектного подхода используется унифицированный язык моделирования UML, который содержит стандартный набор диаграмм для моделирования.

Диаграмма (Diagram) — это графическое представление множества элементов. Чаще всего она изображается в виде связного графа с вершинами (сущностями) и ребрами (отношениями) и представляет собой некоторую проекцию системы.

Объектно-ориентированный подход обладает следующими преимуществами:

* Объектная декомпозиция дает возможность создавать модели меньшего размера путем использования общих механизмов, обеспечивающих необходимую экономию выразительных средств. Использование объектного подхода существенно повышает уровень унификации разработки и пригодность для повторного использования, что ведет к созданию среды разработки и переходу к сборочному созданию моделей.
* Объектная декомпозиция позволяет избежать создания сложных моделей, так как она предполагает эволюционный путь развития модели на базе относительно небольших подсистем.
* Объектная модель естественна, поскольку ориентированна на человеческое восприятие мира.

К недостаткам объектно-ориентированного подхода относятся высокие начальные затраты. Этот подход не дает немедленной отдачи. Эффект от его применения сказывается после разработки двух–трех проектов и накопления повторно используемых компонентов. Диаграммы, отражающие специфику объектного подхода, менее наглядны.

#### 3.4 Сравнение существующих методик

В **функциональных моделях** (DFD-диаграммах потоков данных, SADT-диаграммах) главными структурными компонентами являются функции ( операции, действия, работы), которые на диаграммах связываются между собой потоками объектов.

Несомненным достоинством функциональных моделей является реализация структурного подхода к проектированию ИС по принципу "сверху-вниз", когда каждый функциональный блок может быть декомпозирован на множество подфункций и т.д., выполняя, таким образом, модульное проектирование ИС. Для функциональных моделей характерны процедурная строгость декомпозиции ИС и наглядность представления.

При функциональном подходе объектные модели данных в виде ER-диаграмм "объект — свойство — связь" разрабатываются отдельно. Для проверки корректности моделирования предметной области между функциональными и объектными моделями устанавливаются взаимно однозначные связи.

Главный недостаток функциональных моделей заключается в том, что процессы и данные существуют отдельно друг от друга — помимо функциональной декомпозиции существует структура данных, находящаяся на втором плане. Кроме того, не ясны условия выполнения процессов обработки информации, которые динамически могут изменяться.

Перечисленные недостатки функциональных моделей снимаются в **объектно-ориентированных моделях**, в которых главным структурообразующим компонентом выступает класс объектов с набором функций, которые могут обращаться к атрибутам этого класса.

Для классов объектов характерна иерархия обобщения, позволяющая осуществлять **наследование** не только атрибутов (свойств) объектов от вышестоящего класса объектов к нижестоящему классу, но и функций (методов).

В случае наследования функций можно абстрагироваться от конкретной реализации процедур ( **абстрактные типы данных** ), которые отличаются для определенных подклассов ситуаций. Это дает возможность обращаться к подобным программным модулям по общим именам ( **полиморфизм** ) и осуществлять повторное использование программного кода при модификации программного обеспечения. Таким образом, адаптивность объектно-ориентированных систем к изменению предметной области по сравнению с функциональным подходом значительно выше.

При объектно-ориентированном подходе изменяется и принцип проектирования ИС. Сначала выделяются классы объектов, а далее в зависимости от возможных состояний объектов (жизненного цикла объектов) определяются методы обработки (функциональные процедуры), что обеспечивает наилучшую реализацию динамического поведения информационной системы.

Для объектно-ориентированного подхода разработаны графические методы моделирования предметной области, обобщенные в языке унифицированного моделирования UML. Однако по наглядности представления модели пользователю-заказчику объектно-ориентированные модели явно уступают функциональным моделям.

При выборе методики моделирования предметной области обычно в качестве критерия выступает степень ее динамичности. Для более регламентированных задач больше подходят функциональные модели, для более адаптивных бизнес-процессов (управления рабочими потоками, реализации динамических запросов к информационным хранилищам) — объектно-ориентированные модели. Однако в рамках одной и той же ИС для различных классов задач могут требоваться различные виды моделей, описывающих одну и ту же проблемную область. В таком случае должны использоваться комбинированные модели предметной области.

### 3.5 Синтетическая методика

Как можно видеть из представленного обзора, каждая из рассмотренных методик позволяет решить задачу построения формального описания рабочих процедур исследуемой системы. Все методики позволяют построить модель "как есть" и "как должно быть". С другой стороны, каждая из этих методик обладает существенными недостатками. Их можно суммировать следующим образом: недостатки применения отдельной методики лежат не в области описания реальных процессов, а в неполноте методического подхода.

Функциональные методики в целом лучше дают представление о существующих функциях в организации, о методах их реализации, причем чем выше степень детализации исследуемого процесса, тем лучше они позволяют описать систему. Под лучшим описанием в данном случае понимается наименьшая ошибка при попытке по полученной модели предсказать поведение реальной системы. На уровне отдельных рабочих процедур их описание практически однозначно совпадает с фактической реализацией в потоке работ.

На уровне общего описания системы функциональные методики допускают значительную степень произвола в выборе общих интерфейсов системы, ее механизмов и т.д., то есть в определении границ системы. Хорошо описать систему на этом уровне позволяет объектный подход, основанный на понятии сценария использования. Ключевым является понятие о сценарии использования как о сеансе взаимодействия действующего лица с системой, в результате которого действующее лицо получает нечто, имеющее для него ценность. Использование критерия ценности для пользователя дает возможность отбросить не имеющие значения детали потоков работ и сосредоточиться на тех функциях системы, которые оправдывают ее существование. Однако и в этом случае задача определения границ системы, выделения внешних пользователей является сложной.

Технология потоков данных, исторически возникшая первой, легко решает проблему границ системы, поскольку позволяет за счет анализа информационных потоков выделить внешние сущности и определить основной внутренний процесс. Однако отсутствие выделенных управляющих процессов, потоков и событийной ориентированности не позволяет предложить эту методику в качестве единственной.

Наилучшим способом преодоления недостатков рассмотренных методик является формирование **синтетической методики**, объединяющей различные этапы отдельных методик. При этом из каждой методики необходимо взять часть методологии, наиболее полно и формально изложенную, и обеспечить возможность обмена результатами на различных этапах применения синергетической методики. В бизнес-моделировании неявным образом идет формирование подобной синергетической методики.

Идея **синтетической методики** заключается в последовательном применении функционального и объектного подхода с учетом возможности реинжиниринга существующей ситуации.

Рассмотрим применение синтетической методики на примере разработки административного регламента.

При построении административных регламентов выделяются следующие стадии:

1. Определение границ системы. На этой стадии при помощи **анализа потоков данных выделяют внешние сущности** и собственно моделируемую систему.
2. Выделение сценариев использования системы. На этой стадии **при помощи критерия** полезности **строят** для каждой внешней сущности **набор сценариев использования системы**.
3. Добавление системных сценариев использования. На этой стадии **определяют сценарии, необходимые для реализации целей системы**, отличных от целей пользователей.
4. Построение диаграммы активностей по сценариям использования. На этой стадии строят **набор действий системы**, приводящих к реализации сценариев использования;
5. Функциональная **декомпозиция диаграмм активностей** как контекстных диаграмм методики IDEF0.
6. Формальное описание отдельных функциональных активностей в виде административного регламента (с применением различных нотаций ).
7. **Структурные методологии проектирования**

Моделирование деловых процессов, как правило, выполняется с помощью case-средств. К таким средствам относятся BPwin (PLATINUM technology) – теперь это **AllFusion Business Modeler**, Silverrun (Silverrun technology), Oracle Designer (Oracle), Rational Rose (Rational Software) и др. Функциональные возможности инструментальных средств структурного моделирования деловых процессов будут рассмотрены на примере case-средства BPwin.

**AllFusion Business Modeler** поддерживает три методологии моделирования: функциональное моделирование (IDEF0); описание бизнес-процессов (IDEF3); диаграммы потоков данных (DFD).

**4.1 Построение модели IDEF0**

На начальных этапах создания ИС необходимо понять, как работает организация, которую собираются автоматизировать. Руководитель хорошо знает работу в целом, но не в состоянии вникнуть в детали работы каждого рядового сотрудника. Рядовой сотрудник хорошо знает, что творится на его рабочем месте, но может не знать, как работают коллеги. Поэтому для описания работы предприятия необходимо построить модель, которая будет адекватна предметной области и содержать в себе знания всех участников бизнес-процессов организации.

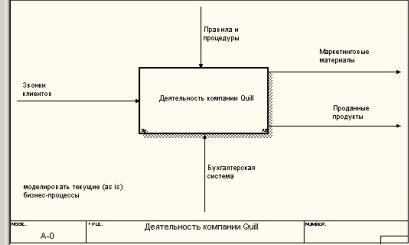
Наиболее удобным языком моделирования бизнес-процессов является IDEF0, где система представляется как совокупность взаимодействующих работ или функций. Такая чисто функциональная ориентация является принципиальной — функции системы анализируются независимо от объектов, которыми они оперируют. Это позволяет более четко смоделировать логику и взаимодействие процессов организации.

Процесс моделирования системы в IDEF0 начинается с создания контекстной диаграммы — диаграммы наиболее абстрактного уровня описания системы в целом, содержащей определение субъекта моделирования, цели и точки зрения на модель.

Под субъектом понимается сама система, при этом необходимо точно установить, что входит в систему, а что лежит за ее пределами, другими словами, определить, что будет в дальнейшем рассматриваться как компоненты системы, а что как внешнее воздействие. На определение субъекта системы будут существенно влиять позиция, с которой рассматривается система, и цель моделирования — вопросы, на которые построенная модель должна дать ответ. Другими словами, в начале необходимо определить область моделирования. Описание области как системы в целом, так и ее компонентов является основой построения модели. Хотя предполагается, что в ходе моделирования область может корректироваться, она должна быть в основном сформулирована изначально, поскольку именно область определяет направление моделирования. При формулировании области необходимо учитывать два компонента — широту и глубину. Широта подразумевает определение границ модели — что будет рассматриваться внутри системы, а что снаружи. Глубина определяет, на каком уровне детализации модель является завершенной. При определении глубины системы необходимо помнить об ограничениях времени — трудоемкость построения модели растет в геометрической прогрессии с увеличением глубины декомпозиции. После определения границ модели предполагается, что новые объекты не должны вноситься в моделируемую систему.

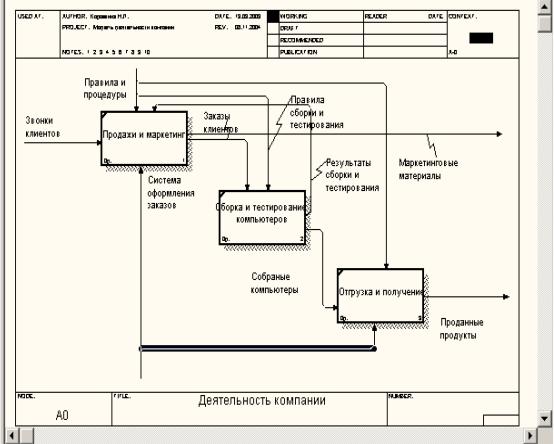
Базовая единица бизнес-модели IDEF0- Работа - **Activity**

Работы **(Activity)**обозначают поименованные процессы, функции или задачи, которые происходят в течение определенного времени и имеют распознаваемые результаты. Работы изображаются в виде прямоугольников. Все работы должны быть названы и определены. Имя работы должно быть выражено отглагольным существительным, обозначающим действие (например, "Деятельность компании", "Прием заказа" и т.д.). Работа "Деятельность компании" может иметь, например, следующее определение: "Это учебная модель, описывающая деятельность компании". При создании новой модели (меню File/New) автоматически создается контекстная диаграмма с единственной работой, изображающей систему в целом ([рис. 4.](http://localhost:3232/department/se/devis/7/devis_7.html#image.7.6)1).



**Рис. 4.1.**  Пример контекстной диаграммы

Диаграммы декомпозиции содержат родственные работы, т. е. дочерние работы, имеющие общую родительскую работу. Для создания диаграммы декомпозиции следует щелкнуть по кнопке



**Рис. 4.2.**  Пример диаграммы декомпозиции

Если оказывается, что количество работ недостаточно, то работу можно добавить в диаграмму, щелкнув сначала по кнопке на палитре инструментов, а затем по свободному месту на диаграмме.

Работы на диаграммах декомпозиции обычно располагаются по диагонали от левого верхнего угла к правому нижнему.

Такой порядок называется порядком доминирования. Согласно этому принципу расположения в левом верхнем углу помещается самая важная работа или работа, выполняемая по времени первой. Далее вправо вниз располагаются менее важные или выполняемые позже работы. Такое размещение облегчает чтение диаграмм, кроме того, на нем основывается понятие взаимосвязей работ (см. ниже).

Каждая из работ на диаграмме декомпозиции может быть в свою очередь декомпозирована. На диаграмме декомпозиции работы нумеруются автоматически слева направо. Номер работы показывается в правом нижнем углу. В левом верхнем углу изображается небольшая диагональная черта, которая показывает, что данная работа не была декомпозирована. Так, на [рис. 4.](http://localhost:3232/department/se/devis/7/devis_7.html#image.7.9)2 все работы еще не были декомпозированы.

Стрелки(Arrow) описывают взаимодействие работ и представляют собой некую информацию, выраженную существительными.(Например, "Звонки клиентов", "Правила и процедуры", "Бухгалтерская система".)

В IDEF0 различают пять типов стрелок:

**Вход**(**Input**) — материал или информация, которые используются или преобразуются работой для получения результата (выхода). Допускается, что работа может не иметь ни одной стрелки входа. Каждый тип стрелок подходит к определенной стороне прямоугольника, изображающего работу, или выходит из нее. Стрелка входа рисуется как входящая в левую грань работы. При описании технологических процессов (для этого и был придуман IDEF0) не возникает проблем определения входов. Действительно, "Звонки клиентов" на [рис. 4.](http://localhost:3232/department/se/devis/7/2.html#image.7.6)1 — это нечто, что перерабатывается в процессе "Деятельность компании" для получения результата. При моделировании ИС, когда стрелками являются не физические объекты, а данные, не все так очевидно. Например, при "Приеме пациента" карта пациента может быть и на входе и на выходе, между тем качество этих данных меняется. Другими словами, в нашем примере для того, чтобы оправдать свое назначение, стрелки входа и выхода должны быть точно определены с тем, чтобы указать на то, что данные действительно были переработаны (например, на выходе — "Заполненная карта пациента"). Очень часто сложно определить, являются ли данные входом или управлением. В этом случае подсказкой может служить информация о том, перерабатываются/изменяются ли данные в работе или нет. Если изменяются, то, скорее всего, это вход, если нет — управление.

**Управление**(**Control**) — правила, стратегии, процедуры или стандарты, которыми руководствуется работа. Каждая работа должна иметь хотя бы одну стрелку управления. Стрелка управления рисуется как входящая в верхнюю грань работы. На [рис. 3.6](http://localhost:3232/department/se/devis/7/2.html#image.7.6) стрелка "Правила и процедуры" — управление для работы "Деятельность компании". Управление влияет на работу, но не преобразуется работой. Если цель работы — изменить процедуру или стратегию, то такая процедура или стратегия будет для работы входом. В случае возникновения неопределенности в статусе стрелки (управление или вход) рекомендуется рисовать стрелку управления.

**Выход**(**Output**) — материал или информация, которые производятся работой. Каждая работа должна иметь хотя бы одну стрелку выхода. Работа без результата не имеет смысла и не должна моделироваться. Стрелка выхода рисуется как исходящая из правой грани работы. На [рис. 4.](http://localhost:3232/department/se/devis/7/2.html#image.7.6)1 стрелки "Маркетинговые материалы" и "Проданные продукты" являются выходом для работы "Деятельность компании".

**Механизм**(**Mechanism**) — ресурсы, которые выполняют работу, например персонал предприятия, станки, устройства и т. д. Стрелка механизма рисуется как входящая в нижнюю грань работы. На [рис. 4.](http://localhost:3232/department/se/devis/7/2.html#image.7.6)1 стрелка "Бухгалтерская система" является механизмом для работы "Деятельность компании". По усмотрению аналитика стрелки механизма могут не изображаться в модели.

**Цель моделирования**

Цель моделирования определяется из ответов на следующие вопросы:

Почему этот процесс должен быть смоделирован?

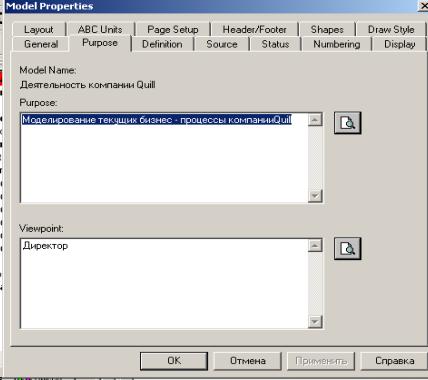
Что должна показывать модель?

Что может получить клиент?

Точка зрения (Viewpoint).

Под точкой зрения понимается перспектива, с которой наблюдалась система при построении модели. Хотя при построении модели учитываются мнения различных людей, все они должны придерживаться единой точки зрения на модель. Точка зрения должна соответствовать цели и границам моделирования. Как правило, выбирается точка зрения человека, ответственного за моделируемую работу в целом.

IDEF0-модель предполагает наличие четко сформулированной цели, единственного субъекта моделирования и одной точки зрения. Для внесения области, цели и точки зрения в модели IDEF0 в BPwin следует выбрать пункт меню Model/Model Properties, вызывающий диалог Model Properties ([рис. 4.3](http://localhost:3232/department/se/devis/7/devis_7.html#image.7.3)). В закладке Purpose следует внести цель и точку зрения, а в закладку Definition — определение модели и описание области.



**Рис. 4.3.**  Диалог задания свойств модели

В закладке Status того же диалога можно описать статус модели (черновой вариант, рабочий, окончательный и т. д.), время создания и последнего редактирования (отслеживается в дальнейшем автоматически по системной дате). В закладке Source описываются источники информации для построения модели (например, "Опрос экспертов предметной области и анализ документации"). Закладка General служит для внесения имени проекта и модели, имени и инициалов автора и временных рамок модели — AS-IS и ТО-ВЕ.

Модели AS-IS и ТО-ВЕ. Обычно сначала строится модель существующей организации работы — AS-IS (как есть). Анализ функциональной модели позволяет понять, где находятся наиболее слабые места, в чем будут состоять преимущества новых бизнес-процессов и насколько глубоким изменениям подвергнется существующая структура организации бизнеса. Детализация бизнес-процессов позволяет выявить недостатки организации даже там, где функциональность на первый взгляд кажется очевидной. Найденные в модели AS-IS недостатки можно исправить при создании модели ТО-ВЕ (как будет) — модели новой организации бизнес-процессов.

Технология проектирования ИС подразумевает сначала создание модели AS-IS, ее анализ и улучшение бизнес-процессов, то есть создание модели ТО-ВЕ, и только на основе модели ТО-ВЕ строится модель данных, прототип и затем окончательный вариант ИС.

Иногда текущая AS-IS и будущая ТО-ВЕ модели различаются очень сильно, так что переход от начального к конечному состоянию становится неочевидным. В этом случае необходима третья модель, описывающая процесс перехода от начального к конечному состоянию системы, поскольку такой переход — это тоже бизнес-процесс.

Результат описания модели можно получить в отчете Model Report. Диалог настройки отчета по модели вызывается из пункта меню Tools/Reports/Model Report.

В диалоге настройки следует выбрать необходимые поля, при этом автоматически отображается очередность вывода информации в отчет

Модель в нотации IDEF0 представляет собой совокупность иерархически упорядоченных и взаимосвязанных диаграмм. Каждая диаграмма является единицей описания системы и располагается на отдельном листе.

Модель может содержать четыре типа диаграмм:

контекстную диаграмму (в каждой модели может быть только одна контекстная диаграмма);

диаграммы декомпозиции;

диаграммы дерева узлов;

диаграммы только для экспозиции (FEO).

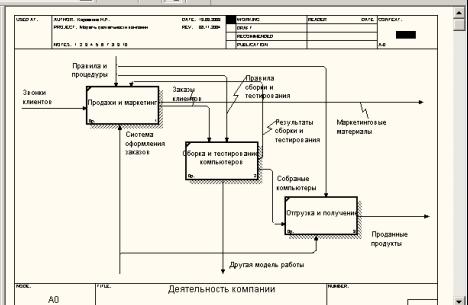
Контекстная диаграмма является вершиной древовидной структуры диаграмм и представляет собой самое общее описание системы и ее взаимодействия с внешней средой. После описания системы в целом проводится разбиение ее на крупные фрагменты. Этот процесс называется функциональной декомпозицией, а диаграммы, которые описывают каждый фрагмент и взаимодействие фрагментов, называются диаграммами декомпозиции. После декомпозиции контекстной диаграммы проводится декомпозиция каждого большого фрагмента системы на более мелкие и так далее, до достижения нужного уровня подробности описания. После каждого сеанса декомпозиции проводятся сеансы экспертизы — эксперты предметной области указывают на соответствие реальных бизнес-процессов созданным диаграммам. Найденные несоответствия исправляются, и только после прохождения экспертизы без замечаний можно приступать к следующему сеансу декомпозиции. Так достигается соответствие модели реальным бизнес-процессам на любом и каждом уровне модели. Синтаксис описания системы в целом и каждого ее фрагмента одинаков во всей модели.

Диаграмма дерева узлов показывает иерархическую зависимость работ, но не взаимосвязи между работами. Диаграмм деревьев узлов может быть в модели сколь угодно много, поскольку дерево может быть построено на произвольную глубину и не обязательно с корня.

диаграммы для экспозиции (FEO) строятся для иллюстрации отдельных фрагментов модели,.

**4.2 Механизм действия стрелок**

**Вызов**(**Call**) — специальная стрелка, указывающая на другую модель работы. Стрелка вызова рисуется как исходящая из нижней грани работы. На [рис. 4.](http://localhost:3232/department/se/devis/7/devis_7.html#image.7.10)4 стрелка "Другая модель работы" является вызовом для работы "Изготовление изделия". Стрелка вызова используется для указания того, что некоторая работа выполняется за пределами моделируемой системы. В BPwin стрелки вызова используются в механизме слияния и разделения моделей.



**Рис. 4.4.**  Стрелка вызова, появляющаяся при расщеплении модели

**Граничные** стрелки. Стрелки на контекстной диаграмме служат для описания взаимодействия системы с окружающим миром. Они могут начинаться у границы диаграммы и заканчиваться у работы, или наоборот. Такие стрелки называются граничными.

Для внесения граничной стрелки входа следует:

щелкнуть по кнопке с символом стрелки

http://localhost:3232/department/se/devis/7/kno1.jpg;

в палитре инструментов перенести курсор к левой стороне экрана, пока не появится начальная штриховая полоска;

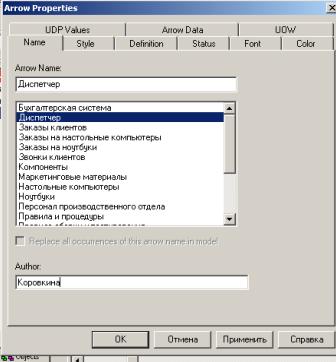
щелкнуть один раз по полоске (откуда выходит стрелка) и еще раз в левой части работы со стороны входа (где заканчивается стрелка);

вернуться в палитру инструментов и выбрать опцию редактирования стрелки

http://localhost:3232/department/se/devis/7/kno2.jpg;

щелкнуть правой кнопкой мыши на линии стрелки, во всплывающем меню выбрать Name и добавить имя стрелки в закладке Name диалога IDEF0 Arrow Properties.

Стрелки управления, входа, механизма и выхода изображаются аналогично. Имена вновь внесенных стрелок ([рис. 4.](http://localhost:3232/department/se/devis/7/devis_7.html#image.7.11)5) автоматически заносятся в словарь Arrow Dictionary.

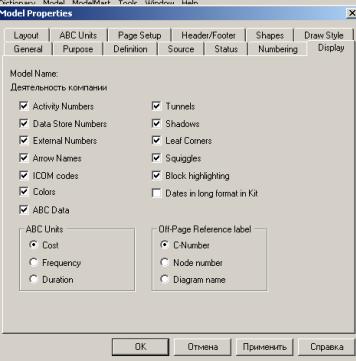


**Рис. 4.5.**  Диалог IDEF0 Arrow Properties

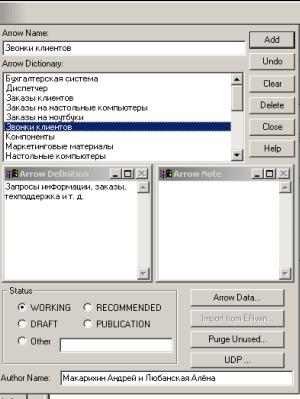
**ICOM-коды**. Диаграмма декомпозиции предназначена для детализации работы. В отличие от моделей, отображающих структуру организации, работа на диаграмме верхнего уровня в IDEF0 — это не элемент управления нижестоящими работами. Работы нижнего уровня — это то же самое, что работы верхнего уровня, но в более детальном изложении. Как следствие этого границы работы верхнего уровня — это то же самое, что границы диаграммы декомпозиции. ICOM (аббревиатура от Input, Control, Output и Mechanism) — коды, предназначенные для идентификации граничных стрелок. Код ICOM содержит префикс, соответствующий типу стрелки (I, С, О или М), и порядковый номер.

BPwin вносит ICOM-коды автоматически. Для отображения ICOM-кодов следует включить опцию ICOM codes на закладке Display диалога Model Properties (меню Model/Model Properties) ([рис.4.](http://localhost:3232/department/se/devis/7/devis_7.html#image.7.12)6).

Словарь **стрелок** редактируется при помощи специального редактора Arrow Dictionary Editor, в котором определяется стрелка и вносится относящийся к ней комментарий ([рис. 4.](http://localhost:3232/department/se/devis/7/devis_7.html#image.7.13)7). Словарь стрелок решает очень важную задачу. Диаграммы создаются аналитиком для того, чтобы провести сеанс экспертизы, т. е. обсудить диаграмму со специалистом предметной области. В любой предметной области формируется профессиональный жаргон, причем очень часто жаргонные выражения имеют нечеткий смысл и воспринимаются разными специалистами по-разному. В то же время аналитик — автор диаграмм должен употреблять те выражения, которые наиболее понятны экспертам. Поскольку формальные определения часто сложны для восприятия, аналитик вынужден употреблять профессиональный жаргон, а чтобы не возникло неоднозначных трактовок, в словаре стрелок каждому понятию можно дать расширенное и, если это необходимо, формальное определение.



**Рис. 4.6.**  Включение опции ICOM codes на закладке Display

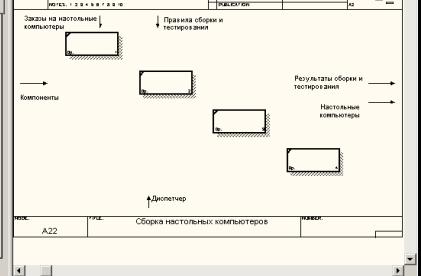


**Рис. 4.7.**  Редактирование словаря стрелок

Содержимое словаря стрелок можно распечатать в виде отчета (меню Tools/ Report /Arrow Report...) и получить толковый словарь терминов предметной области, использующихся в модели.

**Несвязанные граничные** стрелки **(unconnected border arrow)**. При декомпозиции работы входящие в нее и исходящие из нее стрелки (кроме стрелки вызова) автоматически появляются на диаграмме декомпозиции (миграция стрелок), но при этом не касаются работ. Такие стрелки называются несвязанными и воспринимаются в BPwin как синтаксическая ошибка.

На [рис. 4.](http://localhost:3232/department/se/devis/7/devis_7.html#image.7.14)8 приведен фрагмент диаграммы декомпозиции с несвязанными стрелками, генерирующийся BPwin при декомпозиции работы **"Сборка настольных компьютеров"** (см. [рис.](http://localhost:3232/department/se/devis/7/2.html#image.7.9) ). Для связывания стрелок входа, управления или механизма необходимо перейти в режим редактирования стрелок, щелкнуть по наконечнику стрелки и потом по соответствующему сегменту работы. Для связывания стрелки выхода необходимо перейти в режим редактирования стрелок, щелкнуть по сегменту выхода работы и затем по стрелке.



**Рис.4.8.**  Пример несвязанных стрелок

**Внутренние** стрелки. Для связи работ между собой используются внутренние стрелки, то есть стрелки, которые не касаются границы диаграммы, начинаются у одной и кончаются у другой работы.

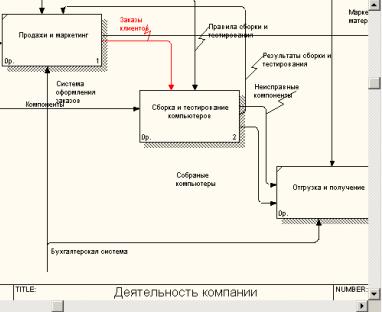
Для рисования внутренней стрелки необходимо в режиме рисования стрелок щелкнуть по сегменту (например, выхода) одной работы и затем по сегменту (например, входа) другой. В IDEF0 различают пять типов связей работ.

**Связь по входу**(**output-input**), когда стрелка выхода вышестоящей работы (далее — просто выход) направляется на вход нижестоящей (например, на [рис. 4.](http://localhost:3232/department/se/devis/7/devis_7.html#image.7.15)9 стрелка **"Собранные компьютеры"** связывает работы **"Сборка и тестирование компьютеров"** и **"Отгрузка и получение"**).



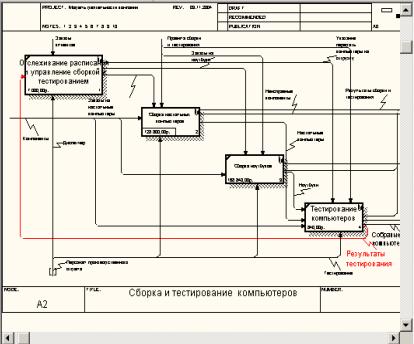
**Рис. 4.9.**  Связь по входу

**Связь по управлению**(**output-control**), когда выход вышестоящей работы направляется на управление нижестоящей. Связь по управлению показывает доминирование вышестоящей работы. Данные или объекты выхода вышестоящей работы не меняются в нижестоящей. На [рис. 4.](http://localhost:3232/department/se/devis/7/devis_7.html#image.7.16)10 стрелка **"Заказы клиентов"** связывает работы **"Продажи и маркетинг"** и **"Сборка и тестирование компьютеров"**.



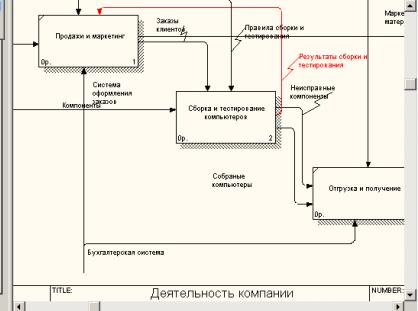
**Рис. 4.10.**  Связь по управлению

**Обратная связь по входу**(**output-input feedback**), когда выход нижестоящей работы направляется на вход вышестоящей. Такая связь, как правило, используется для описания циклов. На [рис. 4.](http://localhost:3232/department/se/devis/7/devis_7.html#image.7.17)11 стрелка "Результаты тестирования" связывает работы **"Тестирование компьютеров"** и **"Отслеживание расписания и управление сборкой и тестированием"**.



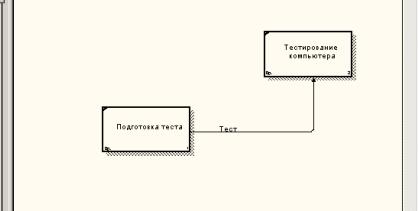
**Рис. 4.11.**  Обратная связь по входу

**Обратная связь по управлению**(**output-control feedback**), когда выход нижестоящей работы направляется на управление вышестоящей (стрелка "Результаты сборки и тестирования", [рис. 4.1](http://localhost:3232/department/se/devis/7/devis_7.html#image.7.18)2). Обратная связь по управлению часто свидетельствует об эффективности бизнес-процесса. На [рис. 4.1](http://localhost:3232/department/se/devis/7/devis_7.html#image.7.18)2 объем продаж может быть повышен путем непосредственного регулирования процессов сборки и тестирования компьютеров (выхода) работы "Сборки и тестирование компьютеров".



**Рис. 4.12.**  Обратная связь по управлению

**Связь выход-механизм**(**output-mechanism**), когда выход одной работы направляется на механизм другой. Эта взаимосвязь используется реже остальных и показывает, что одна работа подготавливает ресурсы, необходимые для проведения другой работы ([рис. 4.1](http://localhost:3232/department/se/devis/7/devis_7.html#image.7.19)3).

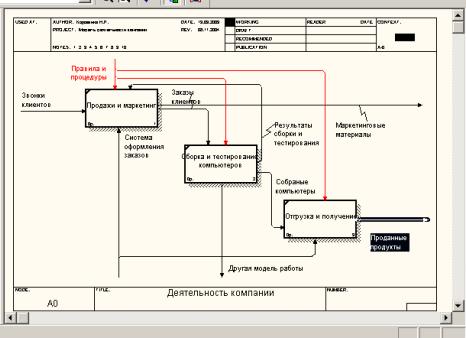


**Рис. 4.13.**  Связь выход-механизм

**Явные** стрелки. Явная стрелка имеет источником одну-единственную работу и назначением тоже одну-единственную работу.

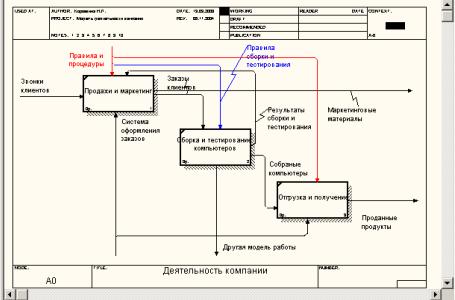
**Разветвляющиеся и сливающиеся** стрелки. Одни и те же данные или объекты, порожденные одной работой, могут использоваться сразу в нескольких других работах. С другой стороны, стрелки, порожденные в разных работах, могут представлять собой одинаковые или однородные данные или объекты, которые в дальнейшем используются или перерабатываются в одном месте. Для моделирования таких ситуаций в IDEF0 используются разветвляющиеся и сливающиеся стрелки. Для разветвления стрелки нужно в режиме редактирования стрелки щелкнуть по фрагменту стрелки и по соответствующему сегменту работы. Для слияния двух стрелок выхода нужно в режиме редактирования стрелки сначала щелкнуть по сегменту выхода работы, а затем по соответствующему фрагменту стрелки.

Смысл разветвляющихся и сливающихся стрелок передается именованием каждой ветви стрелок. Существуют определенные правила именования таких стрелок. Рассмотрим их на примере разветвляющихся стрелок. Если стрелка именована до разветвления, а после разветвления ни одна из ветвей не именована, то подразумевается, что каждая ветвь моделирует те же данные или объекты, что и ветвь до разветвления ([рис. 4.](http://localhost:3232/department/se/devis/7/devis_7.html#image.7.20)14).



**Рис. 4.14.**  Пример именования разветвляющейся стрелки

Если стрелка именована до разветвления, а после разветвления какая-либо из ветвей тоже именована, то подразумевается, что эти ветви соответствуют именованию. Если при этом какая-либо ветвь после разветвления осталась неименованной, то подразумевается, что она моделирует те же данные или объекты, что и ветвь до разветвления ([рис. 4.1](http://localhost:3232/department/se/devis/7/devis_7.html#image.7.21)5).



**Рис. 4.15.**  Пример именования разветвляющейся стрелки

Недопустима ситуация, когда стрелка до разветвления не именована, а после разветвления не именована какая-либо из ветвей. BPwin определяет такую стрелку как синтаксическую ошибку.

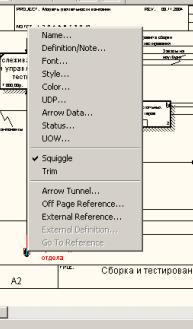
Правила именования сливающихся стрелок полностью аналогичны — ошибкой будет считаться стрелка, которая после слияния не именована, а до слияния не именована какая-либо из ее ветвей. Для именования отдельной ветви разветвляющихся и сливающихся стрелок следует выделить на диаграмме только одну ветвь, после чего вызвать редактор имени и присвоить имя стрелке. Это имя будет соответствовать только выделенной ветви.

**Туннелирование стрелок**. Вновь внесенные граничные стрелки на диаграмме декомпозиции нижнего уровня изображаются в квадратных скобках и автоматически не появляются на диаграмме верхнего уровня ([рис. 4.16](http://localhost:3232/department/se/devis/7/devis_7.html#image.7.22)).



**Рис. 4.16.**  Неразрешенная (unresolved) стрелка

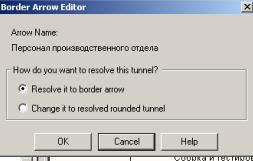
Для их "перетаскивания" наверх нужно щелкнуть правой кнопкой мыши по квадратным скобкам граничной стрелки и в контекстном меню выбрать команду Arrow Tunnel ([рис. 4.](http://localhost:3232/department/se/devis/7/devis_7.html#image.7.23)17).



**Рис. 4.17.**  Выбор команды из контекстного меню

Появляется диалог Border Arrow Editor ([рис. 3.24](http://localhost:3232/department/se/devis/7/devis_7.html#image.7.24)).

Если щелкнуть по кнопке Resolve Border Arrow, стрелка мигрирует на диаграмму верхнего уровня, если по кнопке Change To Tunnel — стрелка будет туннелирована и не попадет на другую диаграмму. Туннельная стрелка изображается с круглыми скобками на конце ([рис. 4.](http://localhost:3232/department/se/devis/7/devis_7.html#image.7.25)18).



**Рис. 4.18.**  Диалог Border Arrow Editor



**Рис. 4.19.**  Типы туннелирования стрелок

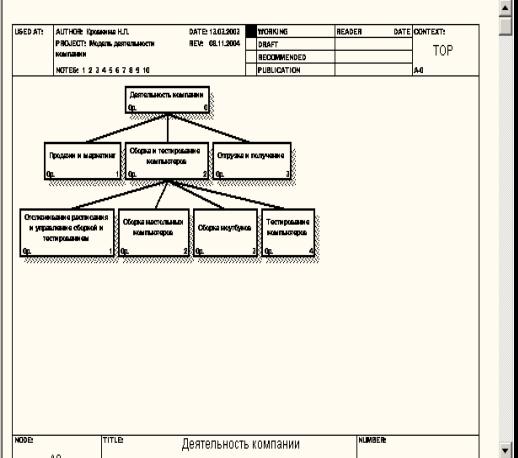
Туннелирование может быть применено для изображения малозначимых стрелок. Если на какой-либо диаграмме нижнего уровня необходимо изобразить малозначимые данные или объекты, которые не обрабатываются или не используются работами на текущем уровне, то их необходимо направить на вышестоящий уровень (на родительскую диаграмму). Если эти данные не используются на родительской диаграмме, их нужно направить еще выше, и т. д. В результате малозначимая стрелка будет изображена на всех уровнях и затруднит чтение всех диаграмм, на которых она присутствует. Выходом является туннелирование стрелки на самом нижнем уровне. Такое туннелирование называется "не-в-родительской-диаграмме".

Другим примером туннелирования может быть ситуация, когда стрелка механизма мигрирует с верхнего уровня на нижний, причем на нижнем уровне этот механизм используется одинаково во всех работах без исключения. (Предполагается, что не нужно детализировать стрелку механизма, т. е. стрелка механизма на дочерней работе именована до разветвления, а после разветвления ветви не имеют собственного имени). В этом случае стрелка механизма на нижнем уровне может быть удалена, после чего на родительской диаграмме она может быть туннелирована, а в комментарии к стрелке или в словаре можно указать, что механизм будет использоваться во всех работах дочерней диаграммы декомпозиции. Такое туннелирование называется "не-в-дочерней-работе" ([рис. 4.](http://localhost:3232/department/se/devis/7/devis_7.html#image.7.25)19).

**4.2.1 Нумерация работ и диаграмм**. Все работы модели нумеруются. Номер состоит из префикса и числа. Может быть использован префикс любой длины, но обычно используют префикс А. Контекстная (корневая) работа дерева имеет номер А0. Работы i декомпозиции А0 имеют номера А1, А2, A3 и т. д. Работы декомпозиции нижнего уровня имеют номер родительской работы и очередной порядковый номер, например работы декомпозиции A3 будут иметь номера А31, А32, АЗЗ, А34 и т. д. Работы образуют иерархию, где каждая работа может иметь одну родительскую и несколько дочерних работ, образуя дерево. Такое дерево называют деревом узлов, а вышеописанную нумерацию — нумерацией по узлам. Диаграммы IDEF0 имеют двойную нумерацию. Во-первых, диаграммы имеют номера по узлу. Контекстная диаграмма всегда имеет номер А-0, декомпозиция контекстной диаграммы — номер А0, остальные диаграммы декомпозиции — номера по соответствующему узлу (например, A1, A2, А21, А213 и т. д.). BPwin автоматически поддерживает нумерацию по узлам, т. е. при проведении декомпозиции создается новая диаграмма и ей автоматически присваивается соответствующий номер. В результате проведения экспертизы диаграммы могут уточняться и изменяться, следовательно, могут быть созданы различные версии одной и той же (с точки зрения ее расположения в дереве узлов) диаграммы декомпозиции. BPwin позволяет иметь в модели только одну диаграмму декомпозиции в данном узле. Прежние версии диаграммы можно хранить в виде бумажной копии либо как FEO-диаграмму. (К сожалению, при создании FEO-диаграмм отсутствует возможность отката, т. е. из диаграммы можно получить декомпозиции FEO, но не наоборот.) В любом случае следует отличать различные версии одной и той же диаграммы. Для этого существует специальный номер — C-number, который должен присваиваться автором модели вручную. C-number — это произвольная строка, но рекомендуется придерживаться стандарта, когда номер состоит из буквенного префикса и порядкового номера, причем в качестве префикса используются инициалы автора диаграммы, а порядковый номер отслеживается автором вручную, например МСВ00021.

**1.2.2 Диаграммы дерева узлов и FEO**

Диаграмма деревьев узлов показывает иерархию работ в модели и позволяет рассмотреть всю модель целиком, но не показывает взаимосвязи между работами ([рис. 4.](http://localhost:3232/department/se/devis/7/devis_7.html#image.7.26)20).Процесс создания модели работ является итерационным, следовательно, работы могут менять свое расположение в дереве узлов многократно. Чтобы не запутаться и проверить способ декомпозиции, следует после каждого изменения создавать диаграмму дерева узлов. Впрочем, BPwin имеет мощный инструмент навигации по модели — Model Explorer, который позволяет представить иерархию работ и диаграмм в удобном и компактном виде, однако составляющей стандарта IDEF0.

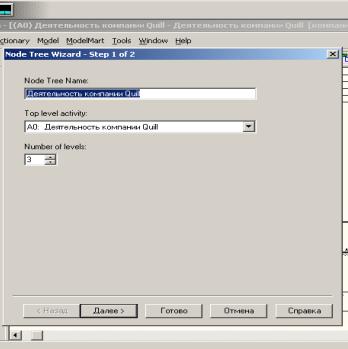


**Рис. 4.20.**  Диаграмма дерева узлов

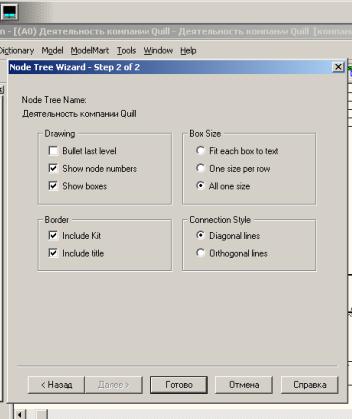
Для создания диаграммы дерева узлов следует выбрать в меню пункт Diagram/Add Node Tree ([рис. 4.22](http://localhost:3232/department/se/devis/7/devis_7.html#image.7.27)). Возникает диалог формирования диаграммы дерева узлов Node Tree Definition ([рис. 4.2](http://localhost:3232/department/se/devis/7/devis_7.html#image.7.28)3, [4. 24](http://localhost:3232/department/se/devis/7/devis_7.html#image.7.29)).



**Рис. 4.22.**  Выбор команды для формирования диаграммы дерева узлов



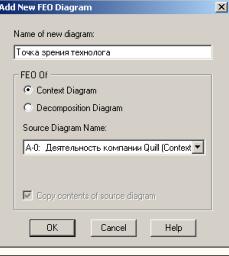
**Рис. 4.23.**  Диалог настройки диаграммы дерева узлов (шаг 1)



**Рис. 4.24.**  Диалог настройки диаграммы дерева узлов (шаг 2)

В диалоге Node Tree Definition следует указать глубину дерева — Number of Levels (по умолчанию — 3) и корень дерева (по умолчанию — родительская работа текущей диаграммы). По умолчанию нижний уровень декомпозиции показывается в виде списка, остальные работы — в виде прямоугольников. Для отображения всего дерева в виде прямоугольников следует выключить опцию Bullet Last Level. При создании дерева узлов следует указать имя диаграммы, поскольку, если в нескольких диаграммах в качестве корня на дереве узлов использовать одну и ту же работу, все эти диаграммы получат одинаковый номер (номер узла + постфикс N, например AON) и в списке открытых диаграмм (пункт меню Window) их можно будет различить только по имени.

Диаграммы "только для экспозиции" (FEO) часто используются в модели для иллюстрации других точек зрения, для отображения отдельных деталей, которые не поддерживаются явно синтаксисом IDEF0. Диаграммы FEO позволяют нарушить любое синтаксическое правило, поскольку по сути являются просто картинками — копиями стандартных диаграмм и не включаются в анализ синтаксиса. Для создания диаграммы FEO следует выбрать пункт меню Diagram/Add FEO Diagram. В возникающем диалоге Add New FEO Diagram следует указать имя диаграммы FEO и тип родительской диаграммы ([рис. 4.](http://localhost:3232/department/se/devis/7/devis_7.html#image.7.30)25).

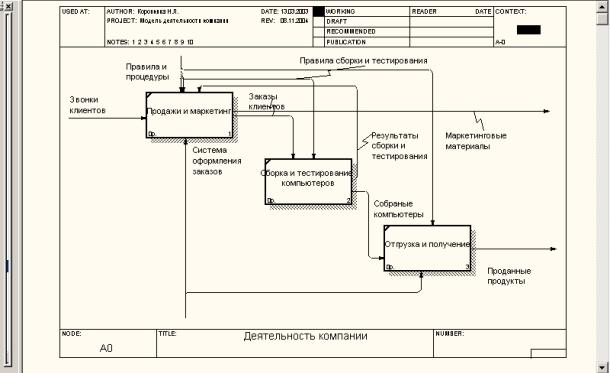


**Рис. 4.25.**  Диалог создания FEO-диаграммы

Новая диаграмма получает номер, который генерируется автоматически (номер родительской диаграммы по узлу + постфикс F, например A1F).

**Каркас диаграммы**

На [рис. 4.](http://localhost:3232/department/se/devis/7/devis_7.html#image.7.31)26 показан типичный пример диаграммы декомпозиции с граничными рамками, которые называются каркасом диаграммы.

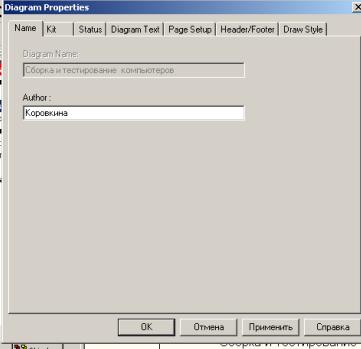


**Рис. 4.26.**  Пример диаграммы декомпозиции с каркасом

Каркас содержит заголовок (верхняя часть рамки) и подвал (нижняя часть). Заголовок каркаса используется для отслеживания диаграммы в процессе моделирования. Нижняя часть используется для идентификации и позиционирования в иерархии диаграммы.

Смысл элементов каркаса приведен в [табл. 4.1](http://localhost:3232/department/se/devis/7/devis_7.html#table.7.1) и [4](http://localhost:3232/department/se/devis/7/6.html#table.7.2).2

Значения полей каркаса задаются в диалоге Diagram Properties (меню Diagram /Diagram Properties) — [рис. 4.2](http://localhost:3232/department/se/devis/7/devis_7.html#image.7.32)7.



**Рис. 4.27.**  Диалог Diagram Properties

Таблица 4.1. Поля заголовка каркаса (слева направо)

|  |  |
| --- | --- |
| **Поле** | **Смысл** |
| Used At | Используется для указания на родительскую работу в случае, если на текущую диаграмму ссылались посредством стрелки вызова |
| Autor, Date, Rev, Project | Имя создателя диаграммы, дата создания и имя проекта, в рамках которого была создана диаграмма. REV-дата последнего редактирования диаграммы |
| Notes 123456789 10 | Используется при проведении сеанса экспертизы. Эксперт должен (на бумажной копии диаграммы) указать число замечаний, вычеркивая цифру из списка каждый раз при внесении нового замечания |
| Status | Статус отображает стадию создания диаграммы, отображая все этапы публикации |
| Working | Новая диаграмма, кардинально обновленная диаграмма или новый автор диаграммы |
| Draft | Диаграмма прошла первичную экспертизу и готова к дальнейшему обсуждению |
| Recommended | Диаграмма и все ее сопровождающие документы прошли экспертизу. Новых изменений не ожидается |
| Publication | Диаграмма готова к окончательной печати и публикации |
| Reader | Имя читателя (эксперта) |
| Date | Дата прочтения (экспертизы) |
| Context | Схема расположения работ в диаграмме верхнего уровня |

Работа, являющаяся родительской, показана темным прямоугольником, остальные – светлым. На контекстной диаграмме (А-0) показана надпись ТОР. В левом нижнем углу показывается номер по узлу родительской диаграммы:

http://localhost:3232/department/se/devis/7/table1.jpg

**4.3 Слияние и расщепление моделей**

Возможность слияния и расщепления моделей обеспечивает коллективную работу над проектом. Так, руководитель проекта может создать декомпозицию верхнего уровня и дать задание аналитикам продолжить декомпозицию каждой ветви дерева в виде отдельных моделей. После окончания работы над отдельными ветвями все подмодели могут быть слиты в единую модель. С другой стороны, отдельная ветвь модели может быть отщеплена для использования в качестве независимой модели, для доработки или архивирования.

Таблица 4.2. Поля подвала каркаса (слева направо)

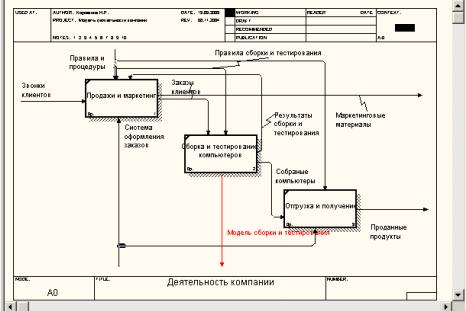
|  |  |
| --- | --- |
| **Поле** | **Смысл** |
| Node | Номер узла диаграммы (номер родительской работы) |
| Title | Имя диаграммы. По умолчанию — имя родительской работы |
| Number | C-Number, уникальный номер версии диаграммы |
| Page | Номер страницы, может использоваться как номер страницы при формировании папки |

BPwin использует для слияния и разветвления моделей стрелки вызова. Для слияния необходимо выполнить следующие условия:

Обе сливаемые модели должны быть открыты в BPwin.

Имя модели-источника, которое присоединяют к модели-цели, должно совпадать с именем стрелки вызова работы в модели-цели.

Стрелка вызова должна исходить из недекомпозируемой работы (работа должна иметь диагональную черту в левом верхнем углу) ([рис. 4.](http://localhost:3232/department/se/devis/7/devis_7.html#image.7.33)28).

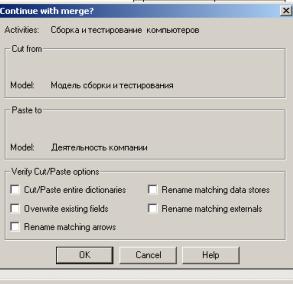


**Рис. 4.28.**  Стрелка вызова работы "Сборка и тестирование компьютеров" модели-цели

Имена контекстной работы подсоединяемой модели-источника и работы на модели-цели, к которой мы подсоединяем модель-источник, должны совпадать.Модель-источник должна иметь, по крайней мере, одну диаграмму декомпозиции.

Для слияния моделей нужно щелкнуть правой кнопкой мыши по работе со стрелкой вызова в модели-цели и во всплывающем меню выбрать пункт Merge Model.

Появляется диалог, в котором следует указать опции слияния модели ([рис. 4.](http://localhost:3232/department/se/devis/7/devis_7.html#image.7.34)29). При слиянии моделей объединяются и словари стрелок и работ. В случае одинаковых определений возможна перезапись определений или принятие определений из модели-источника. То же относится к именам стрелок, хранилищам данных и внешним ссылкам. (Хранилища данных и внешние ссылки — объекты диаграмм потоков данных, DFD, будут рассмотрены ниже.)



**Рис. 4.29.**  Диалог Continue with merge

После подтверждения слияния (кнопка OK) модель-источник подсоединяется к модели-цели, стрелка вызова исчезает, а работа, от которой отходила стрелка вызова, становится декомпозируемой — к ней подсоединяется диаграмма декомпозиции первого уровня модели-источника. Стрелки, касающиеся работы на диаграмме модели-цели, автоматически не мигрируют в декомпозицию, а отображаются как неразрешенные. Их следует туннелировать вручную.

В процессе слияния модель-источник остается неизменной, и к модели-цели подключается фактически ее копия. Не нужно путать слияние моделей с синхронизацией. Если в дальнейшем модель-источник будет редактироваться, эти изменения автоматически не попадут в соответствующую ветвь модели-цели.

Разделение моделей производится аналогично. Для отщепления ветви от модели следует щелкнуть правой кнопкой мыши по декомпозированной работе (работа не должна иметь диагональной черты в левом верхнем углу) и выбрать во всплывающем меню пункт Split Model. В появившемся диалоге Split Options следует указать имя создаваемой модели. После подтверждения расщепления в старой модели работа станет недекомпозированной (признак — диагональная черта в левом верхнем углу), будет создана стрелка вызова, ее имя будет совпадать с именем новой модели, и, наконец, будет создана новая модель, причем имя контекстной работы будет совпадать с именем работы, от которой была "оторвана" декомпозиция.

**4.4 Создание отчетов в BPwin**

BPwin имеет мощный инструмент генерации отчетов. Отчеты по модели вызываются из пункта меню Report. Всего имеется семь типов отчетов:

Model Report. Включает информацию о контексте модели — имя модели, точку зрения, область, цель, имя автора, дату создания и др.

Diagram Report. Отчет по конкретной диаграмме. Включает список объектов (работ, стрелок, хранилищ данных, внешних ссылок и т. д.).

Diagram Object Report. Наиболее полный отчет по модели. Может включать полный список объектов модели (работ, стрелок с указанием их типа и др.) и свойства, определяемые пользователем.

Activity Cost Report. Отчет о результатах стоимостного анализа. Будет рассмотрен ниже.

Arrow Report. Отчет по стрелкам. Может содержать информацию из словаря стрелок, информацию о работе-источнике, работе-назначении стрелки и информацию о разветвлении и слиянии стрелок.

Data Usage Report. Отчет о результатах связывания модели процессов и модели данных. (Будет рассмотрен ниже.)

Model Consistency Report. Отчет, содержащий список синтаксических ошибок модели.



http://localhost:3232/img/empty.gif

**Рис. 4.30.**  Диалог настройки отчета Diagram Object Report

http://localhost:3232/img/empty.gif

1. **Потоковые диаграммы потоков данных**

**5.1 Диаграммы потоков данных (DFD)**

Диаграммы потоков данных (Data Flow Diagramming) являются основным средством моделирования функциональных требований к проектируемой системе. Требования представляются в виде иерархии процессов, связанных потоками данных. Диаграммы потоков данных показывают, как каждый процесс преобразует свои входные данные в выходные, и выявляют отношения между этими процессами. DFD-диаграммы успешно используются как дополнение к модели IDEF0 для описания документооборота и обработки информации. Подобно IDEF0, DFD представляет моделируемую систему как сеть связанных работ. Основные компоненты DFD (как было сказано выше) – процессы или работы, внешние сущности, потоки данных, накопители данных (хранилища).

В BPwin для построения диаграмм потоков данных используется нотация Гейна-Сарсона.

Для того чтобы дополнить модель IDEF0 диаграммой DFD, нужно в процессе декомпозиции в диалоге Activity Box Count "кликнуть" по радио-кнопке DFD. В палитре инструментов на новой диаграмме DFD появляются новые кнопки:

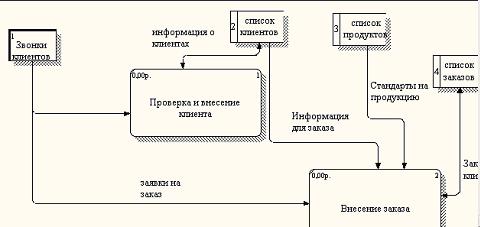
(**External Reference**) — добавить в диаграмму внешнюю ссылку;

(**Data store**) — добавить в диаграмму хранилище данных;

**Diagram Dictionary Editor** – ссылка на другую страницу. В отличие от IDEF0 этот инструмент позволяет направить стрелку на любую диаграмму (а не только на верхний уровень).

В отличие от стрелок IDEF0, которые представляют собой жесткие взаимосвязи, стрелки DFD показывают, как объекты (включая данные) двигаются от одной работы к другой. Это представление потоков совместно с хранилищами данных и внешними сущностями делает модели DFD более похожими на физические характеристики системы — движение объектов, хранение объектов, поставка и распространение объектов ([рис. 5.](http://localhost:3232/department/se/devis/8/devis_8.html#image.8.9)1).

В отличие от IDEF0, где система рассматривается как взаимосвязанные работы, DFD рассматривает систему как совокупность предметов. Контекстная диаграмма часто включает работы и внешние ссылки. Работы обычно именуются по названию системы, например **"Система обработки информации"**. Включение внешних ссылок в контекстную диаграмму не отменяет требования методологии четко определить цель, область и единую точку зрения на моделируемую систему.



**Рис. 5.1.**  Пример диаграммы DFD

В DFD **работы** (процессы) представляют собой функции системы, преобразующие входы в выходы. Хотя работы изображаются прямоугольниками со скругленными углами, смысл их совпадает со смыслом работ IDEF0 и IDEF3. Так же, как процессы IDEF3, они имеют входы и выходы, но не поддерживают управления и механизмы, как IDEF0 ([рис. 4.9](http://localhost:3232/department/se/devis/8/devis_8.html#image.8.9)) (блоки "Проверка и внесение клиентов", "Внесение заказов").

Внешние сущности **изображают входы в систему и/или выходы из системы**. Внешние сущности изображаются в виде прямоугольника с тенью и обычно располагаются по краям диаграммы ([рис. 5.](http://localhost:3232/department/se/devis/8/devis_8.html#image.8.9)1, блок "Звонки клиентов). Одна внешняя сущность может быть использована многократно на одной или нескольких диаграммах. Обычно такой прием используют, чтобы не рисовать слишком длинных и запутанных стрелок.

Потоки работ изображаются **стрелками** и **описывают движение объектов из одной части системы в другую**. Поскольку в DFD каждая сторона работы не имеет четкого назначения, как в IDEF0, стрелки могут подходить и выходить из любой грани прямоугольника работы. В DFD также применяются двунаправленные стрелки для описания диалогов типа "команда-ответ" между работами, между работой и внешней сущностью и между внешними сущностями ([рис. 5.](http://localhost:3232/department/se/devis/8/devis_8.html#image.8.9)1).

В отличие от стрелок, описывающих объекты в движении, хранилища данных изображают объекты в покое ([рис. 5.](http://localhost:3232/department/se/devis/8/devis_8.html#image.8.10)2).

Хранилище данных

**Рис. 5.2.**  Хранилище данных

В материальных системах хранилища данных изображаются там, где объекты ожидают обработки, например в очереди. В системах обработки информации хранилища данных **являются механизмом, который позволяет сохранить данные для последующих** процессов.

В DFD стрелки могут сливаться и разветвляться, что позволяет описать декомпозицию стрелок. Каждый новый сегмент сливающейся или разветвляющейся стрелки может иметь собственное имя.

Диаграммы DFD могут быть построены с использованием традиционного структурного анализа, подобно тому, как строятся диаграммы IDEF0.

В DFD номер каждой работы может включать префикс, номер родительской работы (А) и номер объекта. Номер объекта — это уникальный номер работы на диаграмме. Например, работа может иметь номер А.12.4. Уникальный номер имеют хранилища данных и внешние сущности независимо от их расположения на диаграмме. Каждое хранилище данных имеет префикс D и уникальный номер, например D5. Каждая внешняя сущность имеет префикс Е и уникальный номер, например Е5.

**5.2. Метод описания процессов IDEF3**

Наличие в диаграммах DFD элементов для обозначения источников, приемников и хранилищ данных позволяет более эффективно и наглядно описать процесс документооборота. Однако для описания логики взаимодействия информационных потоков более подходит IDEF3, называемая также workflow diagramming, — методология моделирования, использующая графическое описание информационных потоков, взаимоотношений между процессами обработки информации и объектов, являющихся частью этих процессов. Диаграммы Workflow могут быть использованы в моделировании бизнес-процессов для анализа завершенности процедур обработки информации. С их помощью можно описывать сценарии действий сотрудников организации, например последовательность обработки заказа или события, которые необходимо обработать за конечное время. Каждый сценарий сопровождается описанием процесса и может быть использован для документирования каждой функции.

IDEF3 — это метод, имеющий основной целью дать возможность аналитикам описать ситуацию, когда процессы выполняются в определенной последовательности, а также описать объекты, участвующие совместно в одном процессе.

Техника описания набора данных IDEF3 является частью структурного анализа. В отличие от некоторых методик описаний процессов IDEF3 не ограничивает аналитика чрезмерно жесткими рамками синтаксиса, что может привести к созданию неполных или противоречивых моделей.

IDEF3 может быть также использован как метод создания процессов. IDEF3 дополняет IDEF0 и содержит все необходимое для построения моделей, которые в дальнейшем могут быть использованы для имитационного анализа.

Каждая работа в IDEF3 описывает какой-либо сценарий бизнес-процесса и может являться составляющей другой работы. Поскольку сценарий описывает цель и рамки модели, важно, чтобы работы именовались отглагольным существительным, обозначающим процесс действия, или фразой, содержащей такое существительное.

Точка зрения на модель должна быть документирована. Обычно это точка зрения человека, ответственного за работу в целом. Также необходимо документировать цель модели — те вопросы, на которые призвана ответить модель.

**Диаграмма** является основной единицей описания в IDEF3. Важно правильно построить диаграммы, поскольку они предназначены для чтения другими людьми (а не только автором).

**Единицы работы** — **Unit of Work (UOW)** — также называемые **работами** (activity), являются центральными компонентами модели. В IDEF3 работы изображаются прямоугольниками с прямыми углами и имеют имя, выраженное отглагольным **существительным**, обозначающим процесс действия, одиночным или в составе фразы, и номер (идентификатор); другое имя существительное в составе той же фразы обычно отображает основной выход (результат) работы (например, "Изготовление изделия"). Часто имя существительное в имени работы меняется в процессе моделирования, поскольку модель может уточняться и редактироваться. Идентификатор работы присваивается при создании и не меняется никогда. Даже если работа будет удалена, ее идентификатор не будет вновь использоваться для других работ. Обычно номер работы состоит из номера родительской работы и порядкового номера на текущей диаграмме.

Связи **показывают взаимоотношения работ**. Все связи в IDEF3 однонаправлены и могут быть направлены куда угодно, но обычно диаграммы IDEF3 стараются построить так, чтобы связи были направлены слева направо. В IDEF3 различают **три типа стрелок, изображающих** связи, стиль которых устанавливается через меню Edit/Arrow Style:

**Старшая (Precedence)**

http://localhost:3232/department/se/devis/8/str.jpg

сплошная линия, связывающая единицы работ (UOW). Рисуется слева направо или сверху вниз. Показывает, что работа-источник должна закончиться прежде, чем работа-цель начнется.

**Отношения (Relational Link)**

http://localhost:3232/department/se/devis/8/str1.jpg

пунктирная линия, использующаяся для изображения связей между единицами работ (UOW) а также между единицами работ и объектами ссылок.

Потоки объектов (Object Flow)

http://localhost:3232/department/se/devis/8/str2.jpg

стрелка с двумя наконечниками, применяется для описания того факта, что объект используется в двух или более единицах работы, например, когда объект порождается в одной работе и используется в другой.

**Старшая** связь показывает, что работа-источник заканчивается ранее, чем начинается работа-цель. Часто результатом работы-источника становится объект, необходимый для запуска работы-цели. В этом случае стрелку, обозначающую объект, изображают с двойным наконечником. Имя стрелки должно ясно идентифицировать отображаемый объект. Поток объектов имеет ту же семантику, что и старшая стрелка.

Отношение показывает, что стрелка является альтернативой старшей стрелке или потоку объектов в смысле задания последовательности выполнения работ — работа-источник не обязательно должна закончиться, прежде чем работа-цель начнется. Более того, работа-цель может закончиться прежде, чем закончится работа-источник.

Окончание одной работы может служить сигналом к началу нескольких работ, или же одна работа для своего запуска может ожидать окончания нескольких работ. **Для отображения логики взаимодействия стрелок при слиянии и разветвлении или для отображения множества событий, которые могут или должны быть завершены перед началом следующей работы, используются** перекрестки **(Junction)**. Различают перекрестки для слияния (Fan-in Junction) и разветвления стрелок (Fan-out Junction). Перекресток не может использоваться одновременно для слияния и для разветвления. Для внесения перекрестка служит кнопка

http://localhost:3232/department/se/devis/8/kno.jpg

— (добавить в диаграмму перекресток — Junction) в палитре инструментов. В диалоге Select Junction Type необходимо указать тип перекрестка.

Смысл каждого типа приведен в [таблице 5.1](http://localhost:3232/department/se/devis/8/devis_8.html#table.8.1).

Все перекрестки на диаграмме нумеруются, каждый номер имеет префикс J. Можно редактировать свойства перекрестка при помощи диалога Junction Properties, который вызывается в контекстном меню перекрестка командой Definition/Note. В отличие от IDEF0 и DFD в IDEF3 стрелки могут сливаться и разветвляться только через перекрестки.

**Объект ссылки** в IDEF3 выражает некую идею, концепцию или данные, которые нельзя связать со стрелкой, перекрестком или работой. Для внесения объекта ссылки служит кнопка

http://localhost:3232/department/se/devis/8/kno1.jpg

— (добавить в диаграмму объект ссылки — Referent) в палитре инструментов. Объект ссылки изображается в виде прямоугольника, похожего на прямоугольник работы

http://localhost:3232/department/se/devis/8/kno2.jpg

. Имя объекта ссылки задается в диалоге Referent (пункт Name контекстного меню), в качестве имени можно использовать имя какой-либо стрелки с других диаграмм или имя сущности из модели данных. Объекты ссылки должны быть связаны с единицами работ или перекрестками пунктирными линиями. Официальная спецификация IDEF3 различает **три стиля объектов ссылок** — **безусловные** (unconditional), **синхронные** (synchronous) и **асинхронные** (asynchronous). BPwin поддерживает только безусловные объекты ссылок. Синхронные и асинхронные объекты ссылок, используемые в диаграммах переходов состояний объектов, не поддерживаются.

Таблица 5.1. Типы перекрестков

|  |  |  |
| --- | --- | --- |
| **Наименование Обозначение** | **Смысл в случае слияния стрелок (Fan-in Junction)** | **Смысл в случае разветвления стрелок (Fan-out Junction)** |
| Asynchronous ANDhttp://localhost:3232/department/se/devis/8/table1.jpg | Все предшествующие процессы должны быть завершены | Все следующие процессы должны быть запущены |
| Synchronous ANDhttp://localhost:3232/department/se/devis/8/table3.jpg | Все предшествующие процессы завершены одновременно | Все следующие процессы запускаются одновременно |
| Asynchronous ORhttp://localhost:3232/department/se/devis/8/table2.jpg | Один или несколько предшествующих процессов должны быть завершены | Один или несколько следующих процессов должны быть запущены |
| Synchronous ORhttp://localhost:3232/department/se/devis/8/table4.jpg | Один или несколько предшествующих процессов завершены одновременно | Один или несколько следующих процессов запускаются одновременно |
| http://localhost:3232/department/se/devis/8/table5.jpgXOR (Exclusive OR) | Только один предшествующий процесс завершен | Только один следующий процесс запускается |

При внесении объектов ссылок помимо имени следует указывать тип объекта ссылки. Типы объектов ссылок приведены в [таблице 5.2](http://localhost:3232/department/se/devis/8/devis_8.html#table.8.2)

В IDEF3 **декомпозиция** используется для детализации работ. Методология IDEF3 позволяет декомпозировать работу многократно, т. е. работа может иметь множество дочерних работ. Это позволяет в одной модели описать альтернативные потоки. Возможность множественной декомпозиции предъявляет дополнительные требования к нумерации работ. Так, номер работы состоит из номера родительской работы, версии декомпозиции и собственного номера работы на текущей диаграмме.

Рассмотрим процесс декомпозиции диаграмм IDEF3, включающий взаимодействие автора (аналитика) и одного или нескольких экспертов предметной области.

Перед проведением сеанса экспертизы у экспертов предметной области должны быть документированные сценарии и рамки модели, для того чтобы понять цели декомпозиции. Обычно эксперт предметной области передает аналитику текстовое описание сценария. В дополнение к этому может существовать документация, описывающая интересующие процессы. Из этой информации аналитик должен составить предварительный список работ (отглагольные существительные, обозначающие процесс) и объектов (существительные, обозначающие результат выполнения работы), которые необходимы для перечисленных работ. В некоторых случаях целесообразно создать графическую модель для представления ее эксперту предметной области.

Таблица 5.2. Типы объектов ссылок

|  |  |
| --- | --- |
| **Тип объекта ссылки** | **Цель описания** |
| OBJECT | Описывает участие важного объекта в работе |
| GOTO | Инструмент циклического перехода (в повторяющейся последовательности работ), возможно на текущей диаграмме, но не обязательно. Если все работы цикла присутствуют на текущей диаграмме, цикл может также изображаться стрелкой, возвращающейся на стартовую работу. GOTO может ссылаться на перекресток |
| UOB (Unit of behaviour) | Применяется, когда необходимо подчеркнуть множественное использование какой-либо работы, но без цикла. Например, работа "Контроль качества" может быть использована в процессе "Изготовление изделия" несколько раз, после каждой единичной операции. Обычно этот тип ссылки не используется для моделирования автоматически запускающихся работ |
| NOTE | Используется для документирования важной информации, относящейся к каким-либо графическим объектам на диаграмме. NOTE является альтернативой внесению текстового объекта в диаграмму |
| ELAB (Elaboration) | Используется для усовершенствования графиков или их более детального описания. Обычно употребляется для детального описания разветвления и слияния стрелок на перекрестках |

На [рисунке 5.](http://localhost:3232/department/se/devis/8/devis_8.html#image.8.11)3 представлено описание процесса "Сборка настольных компьютеров" в методологии IDEF3.

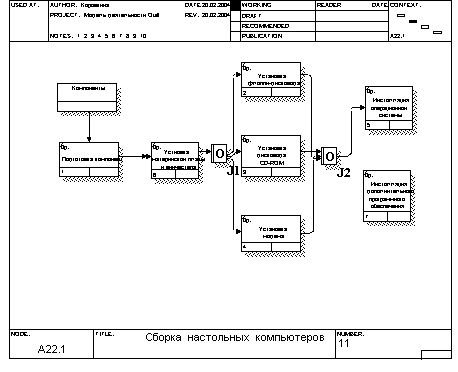
Поскольку разные фрагменты модели IDEF3 могут быть созданы разными группами аналитиков в разное время, IDEF3 поддерживает простую схему нумерации работ в рамках всей модели. Разные аналитики оперируют разными диапазонами номеров, работая при этом независимо. Пример выделения диапазона приведен в [табл. 4.3](http://localhost:3232/department/se/devis/8/devis_8.html#table.8.3).

В результате дополнения диаграмм IDEF0 диаграммами DFD и IDEF3 может быть создана смешанная модель, которая наилучшим образом описывает все стороны деятельности предприятия. Иерархию работ в смешанной модели можно увидеть в окне Model Explorer ([рис. 5.](http://localhost:3232/department/se/devis/8/devis_8.html#image.8.12)4). Модели в нотации IDEF0 изображаются зеленым цветом, в IDEF3 — желтым, в DFD — голубым.

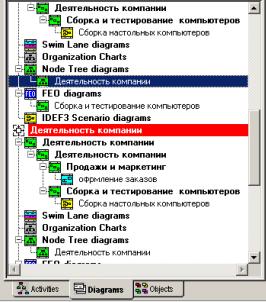
Таблица 5.3. Диапазоны номеров работ

|  |
| --- |
| Аналитик |
| Диапазон номеров IDEF3 |
| Иванов |
| 1-999 |
| Петров |
| 1000-1999 |
| Сидоров |

2000-2999



**Рис. 5.3.**  Описание процесса в методологии IDEF3



**Рис. 5.4.**  Представление смешанной модели в окне Model Explorer

**5.4 Имитационное моделирование**

**Оценочные аспекты** моделирования предметной области связаны с разрабатываемыми показателями эффективности автоматизируемых процессов.

Метод функционального моделирования позволяет оптимизировать существующие на предприятии бизнес-процессы, однако для оптимизации конкретных технологических операций функциональной модели может быть недостаточно. В этом случае целесообразно использовать имитационное моделирование.

Имитационное моделирование – это метод, позволяющий строить модели, учитывающие время выполнения операций, и обеспечивающий наиболее полные средства анализа динамики бизнес-процессов. Имитационные модели описывают не только потоки сущностей, информации и управления, но и различные метрики. Полученную модель можно "проиграть" во времени и получить статистику происходящих процессов так, как это было бы в реальности. В имитационной модели изменения процессов и данных ассоциируются с событиями. "Проигрывание" модели заключается в последовательном переходе от одного события к другому.

Связь между имитационными моделями и моделями процессов заключается в возможности преобразования модели процессов в имитационную модель. Имитационная модель дает больше информации для анализа системы, в свою очередь результаты такого анализа могут быть причиной модификации модели процессов.

Одним из наиболее эффективных инструментов имитационного моделирования является система ARENA, разработанная фирмой System Modeling Corporation. Система позволяет строить имитационные модели, проигрывать их и анализировать результаты.

Имитационная модель включает следующие основные элементы:

Источники и стоки (Create и Dispose). Источники — это элементы, от которых в модель поступает информация или объекты. По смыслу они близки к понятиям "внешняя ссылка" на DFD-диаграммах или "объект ссылки" на диаграммах IDEF3. Скорость поступления данных или объектов от источника обычно задается статистической функцией. Сток — это устройство для приема информации или объектов.

Очереди (Queues). Понятие очереди близко к понятию хранилища данных на DFD-диаграммах — это место, где объекты ожидают обработки. Время обработки объектов в разных работах может быть разным. В результате перед некоторыми работами могут накапливаться объекты, ожидающие своей очереди. Часто целью имитационного моделирования является минимизация количества объектов в очередях.

Процессы (Process) — это аналог работ в модели процессов. В имитационной модели может быть задана производительность процессов.

Построение модели производится путем переноса из панели инструментов в рабочее пространство модулей **Create, Dispose и Process**. Связи между модулями устанавливаются автоматически, но могут быть переопределены вручную. Далее модулям назначаются свойства. Для контроля проигрывания модели необходимо в модель добавить модуль Simulate и задать для него параметры. Результаты проигрывания модели отображаются в автоматически генерируемых отчетах.

BPwin не имеет собственных инструментов, позволяющих создавать имитационные модели, однако дает возможность экспортировать модель IDEF3 в специализированное средство создания таких моделей. Для экспорта модели необходимо настроить свойства, определяемые пользователем UDP, специально включенные в BPwin для целей экспорта.

Функциональные и имитационные модели тесно взаимосвязаны и эффективно дополняют друг друга. Имитационные модели дают больше информации для анализа системы, результаты которого могут быть причиной модификации модели процессов. Целесообразно сначала строить функциональную модель, а на ее основе — имитационную.

1. **Моделирование данных**

* **http://localhost:3232/img/empty.gif**

http://localhost:3232/img/empty.gif

**6.1 Предназначение стандарта IDEF1**

**Метод IDEFI** основан на подходе Чена и позволяет построить модель данных, эквивалентную реляционной модели в третьей нормальной форме. На основе совершенствования метода IDEFI создана его новая версия — метод IDEFIX, разработанный с учетом таких требований, как простота для изучения и возможность автоматизации. IDEFIX-диаграммы используются в ряде распространенных CASE-средств (в частности, ERwin, Design/IDEF).

Деятельность любого предприятия можно представить как непрерывное изменение состояния физических и интеллектуальных объектов, имеющих отношение к предприятию, таких как сотрудники, средства производства, производимые продукты, идеи, финансы и т.д. Для эффективного менеджмента этим процессом, каждое изменение того или иного объекта должно иметь свое документальное отображение. Этими отображениями служат личные дела сотрудников, отчеты, рекламная продукция, служебные записки и т.д. Их совокупность назовем информационной областью предприятия. Движение информации (например, документооборот) и изменение ее назовем информационными потоками. Очевидно, что любому бизнес процессу, а также любому изменению физических объектов должен соответствовать определенный информационный поток. Более того, руководство, при построении стратегических планов развития и управлении деятельностью предприятия, (издавая приказы, распоряжения и т.д.), фактически руководствуется информационными потоками и вносит в них изменения, таким образом осуществляя информационный менеджмент.

Стандарт IDEF1 был разработан как инструмент для анализа и изучения взаимосвязей между информационными потоками в рамках коммерческой деятельности предприятия. Целью подобного исследования является дополнение и структуризация существующей информации и обеспечение качественного менеджмента информационными потоками. Необходимость в подобной реорганизации информационной области как правило возникает на начальном этапе построения корпоративной информационной системы, и методология IDEF1 позволяет достаточно наглядно обнаружить "черные дыры" и слабые места в существующей структуре информационных потоков. Применение методологии IDEF1, как инструмента построения наглядной модели информационной структуры предприятия по принципу "Как должно быть" позволяет решить следующие задачи:

* Выяснить структуру и содержание существующих потоков информации на предприятии
* Определить какие проблемы, выявленные в результате функционального анализа и анализа потребностей, вызваны недостатком управления соответствующей информацией.
* Выявить, информационные потоки, требующие дополнительного управления для эффективной реализации модели.

С помощью IDEF1 происходит изучение существующей информации о различных объектах в области деятельности предприятия. Характерно то, что IDEF1-модель включает в рассмотрение не только автоматизированные компоненты, базы данных и соответствующую им информацию, но также и реальные объекты, такие как сами сотрудники, кабинеты, телефоны и т.д. Миссия методологии IDEF1 состоит в том, чтобы выявить и четко постулировать потребности в информационном менеджменте в рамках коммерческой деятельности предприятия. В отличие от методов разработки структур баз данных (например, IDEF1X), IDEF1 является аналитическим методом и используется преимущественно для выполнения следующих действий:

* Определения самой информации и структуры ее потоков, имеющей отношение к деятельности предприятия
* Определение существующих правил и законов, по которым осуществляется движение информационных потоков, а также принципов управления ими.
* Выяснение взаимосвязей между существующими информационными потоками в рамках предприятия.
* Выявление проблем, возникающих вследствие недостатка качественного информационного менеджмента.

Результаты анализа информационных потоков могут быть использованы для стратегического и тактического планирования деятельности предприятия и улучшения информационного менеджмента.

Однако основной целью использования методологии IDEF1 все же остается исследование движения потоков информации и принципов управления ими на начальном этапе процесса проектирования корпоративной информационно-аналитической системы, которая будет способствовать более эффективному использованию информационного пространства. Наглядные модели IDEF1 обеспечивают базис для построения мощной и гибкой информационной системы.

Основные преимущества IDEF1

Методология IDEF1 позволяет на основе простых графических изображений моделировать информационные взаимосвязи и различия между:

1. Реальными объектами
2. Физическими и абстрактными зависимостями, существующими среди реальных объектов
3. Информацией, относящейся к реальным объектам
4. Структурой данных, используемой для приобретения, накопления, применения и управления информацией.

Одним из основных преимуществ методологии IDEF1 является обеспечение последовательного и строго структурированного процесса анализа информационных потоков в рамках деятельности предприятия. Другим отличительным свойством IDEF1 является широко развитая модульность, позволяющая эффективно выявлять и корректировать неполноту и неточности существующей структуры информации, на всем протяжении этапа моделирования.

**6.2 Концепции моделирования IDEF1**

При построении информационной модели проектировщик всегда оперирует с двумя основными глобальными областями, каждой из которой соответствует множество характерных объектов. Первой из этих областей является реальный мир, или же совокупность физических и интеллектуальных объектов, таких, как люди, места, вещи, идеи и т.д., а также все свойства этих объектов и зависимости между ними. Второй же является информационная область. Она включает в себя существующие информационные отображения объектов первой области и их свойств. Информационное отображение, по существу, не является объектом реального мира, однако изменение его, как правило, является следствием некоторого изменения соответствующего ему объекта реального мира. Методология IDEF1 разработана как инструмент для исследования статического соответствия вышеуказанных областей и установления строгих правил и механизмов изменения объектов информационной области при изменении соответствующих им объектов реального мира.

**6.2.1 Терминология и семантика IDEF1**

Методология IDEF1 разделяет элементы структуры информационной области, их свойства и взаимосвязи на *классы.* Центральным понятием методологии IDEF1 является понятие сущности. Класс сущностей представляет собой совокупность информации, накопленной и хранящейся в рамках предприятия и соответствующей определенному объекту или группе объектов реального мира. Основными концептуальными свойствами сущностей в IDEF1 являются:

1) Устойчивость. Информация, имеющая отношение к той или иной сущности постоянно накапливается.

2) Уникальность. Любая сущность может быть однозначно идентифицирована из другой сущности.

Каждая сущность имеет своё имя и атрибуты. Атрибуты представляют собой характерные свойства и признаки объектов реального мира, относящихся к определенной сущности. Класс атрибутов представляет собой набор пар, состоящих из имени атрибута и его значения для определенной сущности. Атрибуты, по которым можно однозначно отличить одну сущность от другой называются ключевыми атрибутами. Каждая сущность может характеризоваться несколькими ключевыми атрибутами. Класс взаимосвязей в IDEF1 представляет собой совокупность взаимосвязей между сущностями. Взаимосвязь между двумя отдельными сущностями считается существующей в том случае, класс атрибутов одной сущности содержит ключевые атрибуты другой сущности. Каждый из вышеописанных классов имеет свое условное графическое отображение, согласно методологии IDEF1.

IDEF1X является методом для разработки реляционных баз данных и использует условный синтаксис, специально разработанный для удобного построения концептуальной схемы. **Концептуальной схемой** мы называем универсальное представление структуры данных в рамках коммерческого предприятия, независимое от конечной реализации базы данных и аппаратной платформы. Будучи статическим методом разработки, IDEF1X изначально не предназначен для динамического анализа по принципу "AS IS", тем не менее, он иногда применяется в этом качестве, как альтернатива методу IDEF1. Использование метода IDEF1X наиболее целесообразно для построения логической структуры базы данных после того, как все информационные ресурсы исследованы (скажем с помощью метода IDEF1) и решение о внедрении реляционной базы данных, как части корпоративной информационной системы, было принято. Однако не стоит забывать, что средства моделирования IDEF1X специально разработаны для построения реляционных информационных систем, и если существует необходимость проектирования другой системы, скажем объектно-ориентированной, то лучше избрать другие методы моделирования.

Существует несколько очевидных причин, по которым IDEF1X не следует применять в случае построения нереляционных систем. Во-первых, IDEF1X требует от проектировщика определить ключевые атрибуты, для того чтобы отличить одну сущность от другой, в то время как объектно-ориентированные системы не требуют задания ключевых ключей, в целях идентифицирования объектов. Во-вторых, в тех случаях, когда более чем один атрибут является однозначно идентифицирующим сущность, проектировщик должен определить один из этих атрибутов первичным ключом, а все остальные вторичными. И, таким образом, построенная проектировщиком IDEF1X-модель и переданная для окончательной реализации программисту является некорректной для применения методов объектно-ориентированной реализации, и предназначена для построения реляционной системы.

**6.2.2 Концепция и семантика IDEF1X**

**Сущности в IDEF1X и их атрибуты.**

Хотя терминология IDEF1X практически совпадает с терминологией IDEF1, существует ряд фундаментальных отличий в теоретических концепциях этих методологий. Сущность в IDEF1X описывает собой совокупность или набор экземпляров похожих по свойствам, но однозначно отличаемых друг от друга по одному или нескольким признакам. Каждый экземпляр является реализацией сущности. Таким образом, сущность в IDEF1X описывает конкретный набор экземпляров реального мира, в отличие от сущности в IDEF1, которая представляет собой абстрактный набор информационных отображений реального мира.

*Сущность (Entity) — множество экземпляров реальных или абстрактных объектов (людей, событий, состояний, идей, предметов и др.), обладающих общими атрибутами или характеристиками.*

Любой объект системы может быть представлен только одной сущностью, которая должна быть уникально идентифицирована. При этом имя сущности должно отражать тип или класс объекта, а не его конкретный экземпляр (например, АЭРОПОРТ, а не ВНУКОВО). Сущность описывается в диаграмме IDEF1X графическим объектом в виде прямоугольника.

Каждый экземпляр сущности должен однозначно идентифицироваться и отличаться от всех других экземпляров данного типа сущности. Таким образом, каждая сущность должна обладать некоторыми свойствами:

* иметь уникальное имя -уникальный *идентификатор*; к одному и тому же имени должна всегда применяться одна и та же интерпретация; одна и та же интерпретация не может применяться к различным именам, если только они не являются псевдонимами;
* иметь один или несколько атрибутов, которые либо принадлежат сущности, либо наследуются через связь;
* иметь один или несколько атрибутов, которые однозначно идентифицируют каждый экземпляр сущности.

*Атрибут (Attribute) — любая характеристика сущности, значимая для рассматриваемой предметной области и предназначенная для квалификации, идентификации, классификации, количественной характеристики или выражения состояния сущности.*

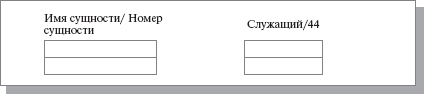
Атрибут представляет тип характеристик или свойств, ассоциированных с множеством реальных или абстрактных объектов (людей, мест, событий, состояний, идей, предметов и т.д.). Экземпляр атрибута — это определенная характеристика отдельного элемента множества. Экземпляр атрибута определяется типом характеристики и ее значением, называемым значением атрибута. На диаграмме атрибуты ассоциируются с конкретными сущностями. Таким образом, экземпляр сущности должен обладать единственным определенным значением для ассоциированного атрибута.

Каждая сущность может обладать любым количеством связей с другими сущностями модели.

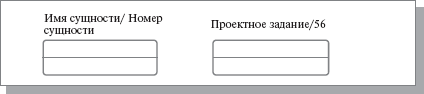
*Связь (Relationship) — поименованная ассоциация между двумя сущностями, значимая для рассматриваемой предметной области. Связь — это ассоциация между сущностями, при которой каждый экземпляр одной сущности ассоциирован с произвольным (в том числе нулевым) количеством экземпляров второй сущности, и наоборот.*

Сокращение ER (Entity-Relationship) часто применяется для обозначения диаграмм IDF1Х – ER-диаграммы или диаграммы "сущность-связь". Однако, у ER и IDF1X, несмотря на существенные элементы идентичности, есть отличия о которых будет сказано чуть ниже.

В методе IDEFIX сущность является независимой от идентификаторов или просто независимой, если каждый экземпляр сущности может быть однозначно идентифицирован без определения его отношений с другими сущностями. Сущность называется зависимой от идентификаторов или просто зависимой, если однозначная идентификация экземпляра сущности зависит от его отношения к другой сущности (рис.6.1,6.2).



**Рис. 6.1.**  Независимые от идентификации сущности



**Рис. 6.2.**  Зависимые от идентификации сущности

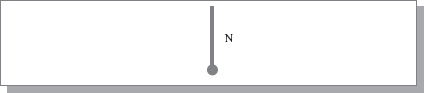
Каждой сущности присваиваются уникальные имя и номер, разделяемые косой чертой "/" и помещаемые над блоком.

Связь может дополнительно определяться с помощью указания степени или мощности (количества экземпляров сущности-потомка, которое может порождать каждый экземпляр сущности-родителя). В IDEFIX могут быть выражены следующие мощности связей:

* каждый экземпляр сущности-родителя может иметь ноль, один или более одного связанного с ним экземпляра сущности-потомка;
* каждый экземпляр сущности-родителя должен иметь не менее одного связанного с ним экземпляра сущности-потомка;
* каждый экземпляр сущности-родителя должен иметь не более одного связанного с ним экземпляра сущности-потомка;
* каждый экземпляр сущности-родителя связан с некоторым фиксированным числом экземпляров сущности-потомка.

Если экземпляр сущности-потомка однозначно определяется своей связью с сущностью-родителем, то связь называется идентифицирующей, в противном случае — неидентифицирующей.

Связь изображается линией, проводимой между сущностью-родителем и сущностью-потомком, с точкой на конце линии у сущности-потомка (рис. 6.3). Мощность связей может принимать следующие значения: N — ноль, один или более, Z — ноль или один, Р — один или более. По умолчанию мощность связей принимается равной N.

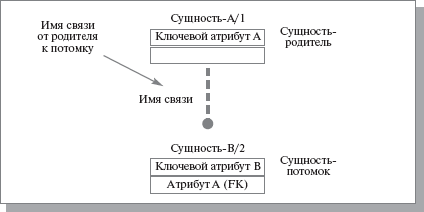
  
**Рис. 6.3.**  Графическое изображение мощности связи

Идентифицирующая связь между сущностью-родителем и сущностью-потомком изображается сплошной линией. Сущность-потомок в идентифицирующей связи является зависимой от идентификатора сущностью. Сущность-родитель в идентифицирующей связи может быть как независимой, так и зависимой от идентификатора сущностью (это определяется ее связями с другими сущностями).

Пунктирная линия изображает неидентифицирующую связь (рис. 6.4). Сущность-потомок в неидентифицирующей связи будет не зависимой от идентификатора, если она не является также сущностью-потомком в какой-либо идентифицирующей связи.

Атрибуты изображаются в виде списка имен внутри блока сущности. Атрибуты, определяющие первичный ключ, размещаются наверху списка и отделяются от других атрибутов горизонтальной чертой (рис. 6.4 ).

Сущности могут иметь также **внешние ключи** (Foreign Key), которые могут использоваться в качестве части или целого первичного ключа или неключевого атрибута. Для обозначения внешнего ключа внутрь блока сущности помещают имена атрибутов, после которых следуют буквы FK в скобках (рис. 6.4).

  
Рис. 6.4.  Неидентифицирующая связь

При разработке модели, зачастую, приходится сталкиваться с сущностями, уникальность которых зависит от значений атрибута внешнего ключа. Для этих сущностей (для уникального определения каждой сущности) внешний ключ должен быть частью первичного ключа дочернего объекта.

Дочерняя сущность, уникальность которой зависит от атрибута внешнего ключа, называется **зависимой сущностью**. В примере на рис. 6.5 сущность СОТРУДНИК является зависимой сущностью потому, что его идентификация зависит от сущности ОТДЕЛ. В обозначениях IDEF1X зависимые сущности представлены в виде закругленных прямоугольников.

Зависимые сущности далее классифицируются на сущности, которые не могут существовать без родительской сущности и сущности, которые не могут быть идентифицированы без использования ключа родителя (сущности, зависящие от идентификации). Сущность СОТРУДНИК принадлежит ко второму типу зависимых сущностей, так как сотрудники могут существовать и без отдела.

Напротив, существуют ситуации в которых сущность зависит от существования другой сущности. Рассмотрим две сущности: ЗАПРОС, используемый для отслеживания запросов покупателей, и ПОЗИЦИЯ ЗАПРОСА, который отслеживает отдельные элементы в ЗАПРОСе. Связь между этими двумя сущностями может быть выражена в виде ЗАПРОС <содержит> один или несколько ПОЗИЦИЙ ЗАПРОСА. В этом случае, ПОЗИЦИЯ ЗАПРОСА зависит от существования ЗАКАЗА.

Сущности, независящие при идентификации от других объектов в модели, называются независимыми **сущностями**. В вышеописанном примере сущность ОТДЕЛ можно считать независимой. В IDEF1X независимые сущности представлены в виде прямоугольников.

**Типы связей между сущностями. Идентифицирующие и неидентифицирующие связи.**

В IDEF1X концепция зависимых и независимых сущностей усиливается типом взаимосвязей между двумя сущностями. Если вы хотите, чтобы внешний ключ передавался в дочернюю сущность (и, в результате, создавал зависимую сущность), то можете создать **идентифицирующую связь** между родительской и дочерней сущность. Идентифицирующие взаимосвязи обозначаются сплошной линией между сущностями.

**Неидентифицирующие связи**, являющиеся уникальными для IDEF1X, также связывают родительскую сущность с дочерней. Неидентифицирующие связи используются для отображения другого типа передачи атрибутов внешних ключей - передача в область данных дочерней сущности (под линией).

Неидентифицирующие связи отображаются пунктирной линией между объектами. Так как переданные ключи в неидентифицирующей связи не являются составной частью первичного ключа дочерней сущности, то этот вид связи не проявляется ни в одной идентифицирующей зависимости. В этом случае и ОТДЕЛ, и СОТРУДНИК рассматриваются как независимые сущности.

Тем не менее, взаимосвязь может отражать зависимость существования, если бизнес правило для взаимосвязи определяет то, что внешний ключ не может принимать значение NULL. Если внешний ключ должен существовать, то это означает, что запись в дочерней сущности может существовать только при наличии ассоциированной с ним родительской записи.

Основным преимуществом IDEF1X, по сравнению с другими многочисленными методами разработки реляционных баз данных, такими как ER и ENALIM является жесткая и строгая стандартизация моделирования. Установленные стандарты позволяют избежать различной трактовки построенной модели, которая несомненно является значительным недостатком ER.

 С помощью ERD осуществляется детализация накопителей документируются информационные аспекты бизнес-системы, включая идентификацию объектов, важных для предметной области (сущностей), свойств этих объектов (атрибутов) и их связей с другими объектами (отношений).

Наиболее распространенными методами для построения ERD-диаграмм являются метод Баркера и метод IDEFI. Метод Баркера основан на нотации, предложенной автором, и используется в case-средстве Oracle Designer.



Рисунок 6.5

### 6.3 Отображение модели данных в инструментальном средстве ERwin

ERwin имеет два уровня представления модели — логический и физический.

**Логический уровень — это абстрактный взгляд на данные, когда данные представляются так, как выглядят в реальном мире**, и могут называться так, как они называются в реальном мире, например "Постоянный клиент", "Отдел" или "Фамилия сотрудника". Объекты модели, представляемые на логическом уровне, называются сущностями и атрибутами. Логическая модель данных может быть построена на основе другой логической модели, например на основе модели процессов. Логическая модель данных является универсальной и никак не связана с конкретной реализацией СУБД.

Физическая модель данных, напротив, зависит от конкретной СУБД, фактически являясь отображением системного каталога. В физической модели содержится информация обо всех объектах БД. Поскольку стандартов на объекты БД не существует (например, нет стандарта на типы данных), физическая модель зависит от конкретной реализации СУБД. Следовательно, одной и той же логической модели могут соответствовать несколько разных физических моделей. Если в логической модели не имеет значения, какой конкретно тип данных имеет атрибут, то в физической модели важно описать всю информацию о конкретных физических объектах — таблицах, колонках, индексах, процедурах и т.д.

#### Документирование модели

Многие СУБД имеют ограничение на именование объектов (например, ограничение на длину имени таблицы или запрет использования специальных символов — пробела и т. п.). Зачастую разработчики ИС имеют дело с нелокализованными версиями СУБД. Это означает, что объекты БД могут называться короткими словами, только латинскими символами и без использования специальных символов (т. е. нельзя назвать таблицу, используя предложение — ее можно назвать только одним словом). Кроме того, проектировщики БД нередко злоупотребляют "техническими" наименованиями, в результате таблица и колонки получают наименования типа RTD\_324 или CUST\_A12 и т.д. Полученную в результате структуру могут понять только специалисты (а чаще всего — только авторы модели), ее невозможно обсуждать с экспертами предметной области. Разделение модели на логическую и физическую позволяет решить эту проблему. На физическом уровне объекты БД могут называться так, как того требуют ограничения СУБД. На логическом уровне можно этим объектам дать синонимы — имена более понятные неспециалистам, в том числе на кириллице и с использованием специальных символов. Например, таблице CUST\_A12 может соответствовать сущность Постоянный клиент. Такое соответствие позволяет лучше документировать модель и дает возможность обсуждать структуру данных с экспертами предметной области.

#### Масштабирование

Создание модели данных, как правило, начинается с разработки логической модели. После описания логической модели проектировщик может выбрать необходимую СУБД, и ERwin автоматически создаст соответствующую физическую модель. На основе физической модели ERwin может сгенерировать системный каталог СУБД или соответствующий SQL-скрипт. Этот процесс называется прямым проектированием (Forward Engineering). Тем самым достигается масштабируемость — создав одну логическую модель данных, можно сгенерировать физические модели под любую поддерживаемую ERwin СУБД. С другой стороны, ERwin способен по содержимому системного каталога или SQL-скрипту воссоздать физическую и логическую модель данных (Reverse Engineering). На основе полученной логической модели данных можно сгенерировать физическую модель для другой СУБД и затем создать ее системный каталог. Следовательно, ERwin позволяет решить задачу по переносу структуры данных с одного сервера на другой. Например, можно перенести структуру данных с Oracle на Informix (или наоборот) или перенести структуру dbf-файлов в реляционную СУБД, тем самым облегчив переход от файл-серверной к клиент-серверной ИС. Однако, формальный перенос структуры "плоских" таблиц на реляционную СУБД обычно неэффективен. Для того чтобы извлечь выгоды от перехода на клиент-серверную технологию, структуру данных следует модифицировать.

**Интерфейс ERwin. Уровни отображения модели**

Интерфейс выполнен в стиле Windows-приложений, достаточно прост и интуитивно понятен. Рассмотрим кратко основные функции ERwin по отображению модели.

Каждому уровню отображения модели соответствует своя палитра инструментов. На логическом уровне палитра инструментов имеет следующие кнопки:

кнопку указателя (режим мыши) — в этом режиме можно установить фокус на каком-либо объекте модели;

кнопку внесения сущности;

кнопку категории (категория, или категориальная связь, — специальный тип связи между сущностями, которая будет рассмотрена ниже);

кнопку внесения текстового блока;

кнопку перенесения атрибутов внутри сущностей и между ними;

кнопки создания связей: идентифицирующую, "многие-ко-многим" и неидентифицирующую.

На **физическом уровне** палитра инструментов имеет:

* вместо кнопки категорий — кнопку внесения представлений (view);
* вместо кнопки связи "многие-ко-многим" — кнопку связей представлений.

Для создания моделей данных в ERwin можно использовать две нотации: IDEFIX и IE (Information Engineering). В дальнейшем будет рассматриваться нотация IDEFIX.

ERwin имеет несколько уровней отображения диаграммы: уровень сущностей, уровень атрибутов, уровень определений, уровень первичных ключей и уровень иконок. Переключиться между первыми тремя уровнями можно с использованием кнопок панели инструментов. Переключиться на другие уровни отображения можно при помощи контекстного меню, которое появляется, если "кликнуть" по любому месту диаграммы, не занятому объектами модели. В контекстном меню следует выбрать пункт Display Level.

### 6.3.1 Создание логической модели данных

#### Уровни логической модели

Различают три уровня логической модели, отличающихся по глубине представления информации о данных:

* диаграмма сущность-связь (Entity Relationship Diagram, ERD);
* модель данных, основанная на ключах (Key Based model, KB);
* полная атрибутивная модель (Fully Attributed model, FA).

Диаграмма сущность-связь представляет собой модель данных верхнего уровня. Она включает сущности и взаимосвязи, отражающие основные бизнес-правила предметной области. Такая диаграмма не слишком детализирована, в нее включаются основные сущности и связи между ними, которые удовлетворяют основным требованиям, предъявляемым к ИС. Диаграмма сущность-связь может включать связи "многие-ко-многим" и не включать описание ключей. Как правило, ERD используется для презентаций и обсуждения структуры данных с экспертами предметной области.

Модель данных, основанная на ключах, — более подробное представление данных. Она включает описание всех сущностей и первичных ключей и предназначена для представления структуры данных и ключей, которые соответствуют предметной области.

Полная атрибутивная модель — наиболее детальное представление структуры данных: представляет данные в третьей нормальной форме и включает все сущности, атрибуты и связи.

#### Сущности и атрибуты

Основные компоненты диаграммы ERwin — это сущности, атрибуты и связи. Каждая сущность является множеством подобных индивидуальных объектов, называемых экземплярами. Каждый экземпляр индивидуален и должен отличаться от всех остальных экземпляров. Атрибут выражает определенное свойство объекта. С точки зрения БД (физическая модель) сущности соответствует таблица, экземпляру сущности — строка в таблице, а атрибуту — колонка таблицы.

Построение модели данных предполагает определение сущностей и атрибутов, т. е. необходимо определить, какая информация будет храниться в конкретной сущности или атрибуте. Сущность можно определить как объект, событие или концепцию, информация о которых должна сохраняться. сущности должны иметь наименование с четким смысловым значением, именоваться существительным в единственном числе, не носить "технических" наименований и быть достаточно важными для того, чтобы их моделировать. Именование сущности в единственном числе облегчает в дальнейшем чтение модели. Фактически имя сущности дается по имени ее экземпляра. Примером может быть сущности Заказчик (но не Заказчики!) с атрибутами Номер заказчика, Фамилия заказчика и Адрес заказчика. На уровне физической модели ей может соответствовать таблица Customer с колонками Customer\_number, Customer\_name и Customer\_address. Каждая сущность должна быть полностью определена с помощью текстового описания. Для внесения дополнительных комментариев и определений к сущности служат свойства, определенные пользователем (UDP). Использование (UDP) аналогично их использованию в BPwin.

Как было указано выше, каждый атрибут хранит информацию об определенном свойстве сущности, а каждый экземпляр сущности должен быть уникальным. Атрибут или группа атрибутов, которые идентифицируют сущность, называется первичным ключем.

Очень важно дать атрибуту правильное имя. Атрибуты должны именоваться в единственном числе и иметь четкое смысловое значение. Соблюдение этого правила позволяет частично решить проблему нормализации данных уже на этапе определения атрибутов. Например, создание в сущности Сотрудник атрибута Телефоны сотрудника противоречит требованиям нормализации, поскольку атрибут должен быть атомарным, т. е. не содержать множественных значений. Согласно синтаксису IDEFIX имя атрибута должно быть уникально в рамках модели (а не только в рамках сущности!). По умолчанию при попытке внесения уже существующего имени атрибута ERwin переименовывает его.

Каждый атрибут должен быть определен, при этом следует избегать циклических определений, например, когда термин 1 определяется через термин 2, термин 2 — через термин 3, а термин 3 в свою очередь — через термин 1. Часто приходится создавать производные атрибуты, т. е. атрибуты, значение которых можно вычислить из других атрибутов. Примером производного атрибута может служить Возраст сотрудника, который может быть вычислен из атрибута Дата рождения сотрудника. Такой атрибут может привести к конфликтам; действительно, если вовремя не обновить значение атрибута Возраст сотрудника, он может противоречить значению атрибута Дата рождения сотрудника. Производные атрибуты — ошибка нормализации, однако их вводят для повышения производительности системы, чтобы не проводить вычисления, которые на практике могут быть сложными.

#### Связи

Связь является логическим соотношением между сущностями. Каждая связь должна именоваться глаголом или глагольной фразой. Имя связи выражает некоторое ограничение или бизнес-правило и облегчает чтение диаграммы. По умолчанию имя связи на диаграмме не показывается. На логическом уровне можно установить идентифицирующую связь "один-ко-многим", связь "многие-ко-многим" и неидентифицирующую связь "один-ко-многим".

В IDEFIX различают зависимые и независимые сущности. Тип сущности определяется ее связью с другими сущностями. Идентифицирующая связь устанавливается между независимой (родительский конец связи) и зависимой (дочерний конец связи) сущностями. Когда рисуется идентифицирующая связь, ERwin автоматически преобразует дочернюю сущность в зависимую. Зависимая сущность изображается прямоугольником со скругленными углами. Экземпляр зависимой сущности определяется только через отношение к родительской сущности. При установлении идентифицирующей связи атрибуты первичного ключа родительской сущности автоматически переносятся в состав первичного ключа дочерней сущности. Эта операция дополнения атрибутов дочерней сущности при создании связи называется миграцией атрибутов.

При установлении неидентифицирующей связи дочерняя сущность остается независимой, а атрибуты первичного ключа родительской сущности мигрируют в состав неключевых компонентов родительской сущности. Неидентифицирующая связь служит для связывания независимых сущностей.

В дочерней сущности новые атрибуты помечаются как внешний ключ(Foreign key — FK).

Идентифицирующая связь показывается на диаграмме сплошной линией с жирной точкой на дочернем конце связи, неидентифицирующая – пунктирной (см. [рис. 10.](http://localhost:3232/department/se/devis/10/2.html#image.10.6)3).

Мощность связей (Cardinality) — служит для обозначения отношения числа экземпляров родительской сущности к числу экземпляров дочерней.

Различают четыре типа сущности:

* общий случай, когда одному экземпляру родительской сущности соответствуют 0, 1 или много экземпляров дочерней сущности; не помечается каким-либо символом;
* символом Р помечается случай, когда одному экземпляру родительской сущности соответствуют 1 или много экземпляров дочерней сущности (исключено нулевое значение);
* символом Z помечается случай, когда одному экземпляру родительской сущности соответствуют 0 или 1 экземпляр дочерней сущности (исключены множественные значения);
* цифрой помечается случай точного соответствия, когда одному экземпляру родительской сущности соответствует заранее заданное число экземпляров дочерней сущности.

Имя связи (Verb Phrase) — фраза, характеризующая отношение между родительской и дочерней сущностями. Для связи "один-ко-многим", идентифицирующей или неидентифицирующей, достаточно указать имя, характеризующее отношение от родительской к дочерней сущности (Parent-to-Child). Для связи многие-ко-многим следует указывать имена как Parent-to-Child, так и Child-to-Parent.

#### Типы сущностей и иерархия наследования

Как было указано выше, связи определяют, является ли сущность независимой или зависимой. Различают несколько **типов зависимых** сущностей.

**Характеристическая** — зависимая дочерняя сущность, которая связана только с одной родительской и по смыслу хранит информацию о характеристиках родительской сущности (рис. 6.7).

Пример характеристической сущности "Хобби"  
**Рис. 6.6.**  Пример характеристической сущности "Хобби"

**Ассоциативная** — сущность, связанная с несколькими родительскими сущностями. Такая сущность содержит информацию о связях сущностей.

**Именующая** — частный случай ассоциативной сущности, не имеющей собственных атрибутов (только атрибуты родительских сущностей, мигрировавших в качестве внешнего ключа).

**Категориальная** — дочерняя сущность в иерархии наследования.

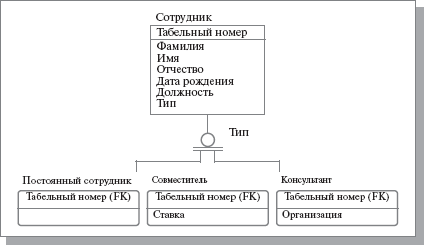
Иерархия наследования (или иерархия категорий) представляет собой особый тип объединения сущностей, которые разделяют общие характеристики. Например, в организации работают служащие, занятые полный рабочий день (постоянные служащие), и совместители. Из их общих свойств можно сформировать обобщенную сущность (родовой предок) Сотрудник (рис. 6.7), чтобы представить информацию, общую для всех типов служащих. Специфическая для каждого типа информация может быть расположена в категориальных сущностях (потомках) Постоянный сотрудник и Совместитель.

Обычно иерархию наследования создают, когда несколько сущностей имеют общие по смыслу атрибуты, либо когда сущности имеют общие по смыслу связи (например, если бы Постоянный сотрудник и Совместитель имели сходную по смыслу связь "работает в" с сущностью Организация), либо когда это диктуется бизнес-правилами.

Для каждой категории можно указать дискриминатор — атрибут родового предка, который показывает, как отличить одну категориальную сущность от другой (атрибут Тип на рис. 6.7).

  
**Рис. 6.7.**  Иерархия наследования. Неполная категория

Иерархии категорий делятся на два типа — полные и неполные. В полной категории одному экземпляру родового предка (сущность Служащий, рис. 6.8) обязательно соответствует экземпляр в каком-либо потомке, т. е. в примере служащий обязательно является либо совместителем, либо консультантом, либо постоянным сотрудником.

  
**Рис. 6.8.**  Иерархия наследования. Полная категория

Если категория еще не выстроена полностью и в родовом предке могут существовать экземпляры, которые не имеют соответствующих экземпляров в потомках, то такая категория будет неполной. На [рис. 6.](http://localhost:3232/department/se/devis/10/devis_10.html#image.10.8)7 показана неполная категория — сотрудник может быть не только постоянным или совместителем, но и консультантом, однако сущность Консультант еще не внесена в иерархию наследования.

#### Ключи

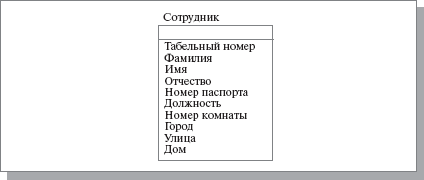
Как было сказано выше, каждый экземпляр сущности должен быть уникален и должен отличаться от других атрибутов.

Первичный ключ (primary key - PK) — это атрибут или группа атрибутов, однозначно идентифицирующая экземпляр сущности. атрибуты первичного ключа на диаграмме не требуют специального обозначения — это те атрибуты, которые находятся в списке атрибутов выше горизонтальной линии (см., например, [рис. 10.9](http://localhost:3232/department/se/devis/10/devis_10.html#image.10.9)).

В одной сущности могут оказаться несколько атрибутов или наборов атрибутов, претендующих на роль первичного ключа. Такие претенденты называются иногда потенциальными ключами (candidate key), а чаще альтернативными ключами (alternate key – AK) . Альтернативный ключ— это потенциальный ключ, не ставший первичным.

Ключи могут быть сложными, т. е. содержащими несколько атрибутов. Сложные первичные ключи не требуют специального обозначения — это список атрибутов, расположенных выше горизонтальной линии.

Рассмотрим кандидатов на роль первичного ключа сущности Сотрудник ([рис. 6.](http://localhost:3232/department/se/devis/10/devis_10.html#image.10.10)9).

  
**Рис. 6.9.**  Определение первичного ключа для сущности "Сотрудник"

Здесь можно выделить следующие потенциальные ключи:

1. Табельный номер;
2. Номер паспорта;
3. Фамилия + Имя + Отчество.

Для того чтобы стать первичным, потенциальный ключ должен удовлетворять ряду требований:

**Уникальность**. Два экземпляра не должны иметь одинаковых значений возможного ключа. потенциальный ключ № 3 (Фамилия + Имя + Отчество) является плохим кандидатом, поскольку в организации могут работать полные тезки.

**Компактность**. Сложный возможный ключ не должен содержать ни одного атрибута, удаление которого не приводило бы к утрате уникальности. Для обеспечения уникальности ключа № 3 дополним его атрибутами Дата рождения и Цвет волос. Если бизнес-правила говорят, что сочетания атрибутов Фамилия + Имя + Отчество + Дата рождения достаточно для однозначной идентификации сотрудника, то Цвет волос оказывается лишним, т. е. ключ Фамилия + Имя + Отчество + Дата рождения + Цвет волос не является компактным.

При выборе первичного ключа предпочтение должно отдаваться более простым ключам, т. е. ключам, содержащим меньшее количество атрибутов. В приведенном примере ключи № 1 и 2 предпочтительней ключа № 3.

Атрибуты ключа не должны содержать нулевых значений. Значение атрибутов ключа не должно меняться в течение всего времени существования экземпляра сущности. Сотрудница организации может выйти замуж и сменить как фамилию, так и паспорт. Поэтому ключи № 2 и 3 не подходят на роль первичного ключа.

Каждая сущность должна иметь по крайней мере один потенциальный ключ. Многие сущности имеют только один потенциальный ключ. Такой ключ становится первичным. Некоторые сущности могут иметь более одного возможного ключа. Тогда один из них становится первичным, а остальные — альтернативными ключами.

**6.4 Нормализация данных**

Нормализация - процесс проверки и реорганизации сущностей и атри­бутов с целью удовлетворения требований к реляционной модели данных. Нормализация позволяет быть уверенным, что каждый атрибут определен для своей сущности, значительно сократить объем памяти для хранения информации и устранить аномалии в организации хранения данных. В ре­зультате проведения нормализации должна быть создана структура данных, при которой информация о каждом факте хранится только в одном месте. Процесс нормализации сводится к последовательному приведению структу­ры данных к нормальным формам - формализованным требованиям к ор­ганизации данных. Известны шесть нормальных форм:

* первая нормальная форма (1NF);
* вторая нормальная форма (2NF);
* третья нормальная форма (3NF);
* нормальная форма Бойса - Кодда (усиленная 3NF);
* четвертая нормальная форма (4NF);
* пятая нормальная форма (5NF).

На практике обычно ограничиваются приведением данных к третьей нормальной форме (полная атрибутивная модель, FA, см. 2.2.1). В данном подразделе будут достаточно кратко рассмотрены первые три нормальные формы и, в качестве иллюстрации, четвертая нормальная форма.

Для углубленного изучения нормализации следует рекомендовать книгу К. Дж. Дейта "Введение в системы баз данных" (Киев;М.:Диалектика, 1998).

Нормальные формы основаны на понятии функциональной зависимо­сти (в дальнейшем будет использоваться термин "зависимость"). Приведем формальное определение для функциональной зависимости.

**Функциональная зависимость (FD).** Атрибут В сущности Е функцио­нально зависит от атрибута А сущности Е тогда и только тогда, когда каж­дое значение А в Е связало с ним точно одно значение В в Е, т. е. А одно­значно определяет В.

**Полная функциональная зависимость.** Атрибут В сущности Е полностью функционально зависит от ряда атрибутов А сущности Е тогда и только тогда, когда В функционально зависит от А и не зависит ни от какого под­ряда А.

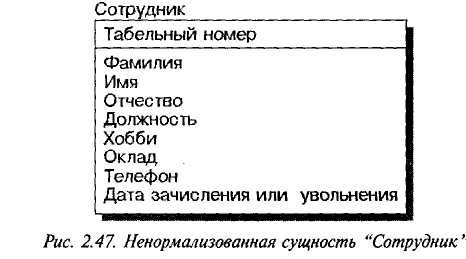


Рисунок 6.10 Пример ненормализованной сущности «Сотрудник»

На рис. 6.10 в сущности ***Сотрудник*** значение атрибутов ***Фамилия, Имя*** и ***Отчество*** однозначно определяются значением атрибута ***Табельный но­мер,*** т. е. атрибуты ***Фамилия, Имя*** и ***Отчество*** зависят от атрибута ***Табель­ный номер.*** Функциональные зависимости определяются бизнес-правилами предметной области. Так, если оклад сотрудника определяется только должностью, то атрибут *Оклад* зависит от атрибута ***Должность;*** если оклад зависит еще, например, от стажа, то такой зависимости нет. В нижеследующих примерах будем считать для определенности, что такая зависимость есть.

Рассмотрим нормальные формы.

**Первая нормальная форма (1NF).** Сущность находится в первой нор­мальной форме тогда и только тогда, когда все атрибуты содержат атомар­ные значения. Среди атрибутов не должно встречаться повторяющихся групп, т. е. несколько значений для каждого экземпляра. На рис. 6.10 атри­буты ***Телефон*** и *Хобби* являются нарушением первой нормальной формы. Что будет, если у сотрудника несколько рабочих телефонов? Запись значе­ния колонки через разделитель, например "124-56-78, 124-56-79, 124-56-90" или "Аквалангист, мотоциклист, шахматист", приводит к ряду проблем. Размера поля может не хватить для хранения данных (нельзя увеличивать список телефонов до бесконечности), по такой колонке невозможно по­строить индекс и т. д. и т. п. Сущность, приведенная на рис. 6.11, не явля­ется решением проблемы. Что будет, если у сотрудника появится четвертый телефон или третье хобби? Эту информацию будет негде хранить.



Рисунок 6.11 Еще один пример ненормализованной сущности

Другой ошибкой нормализации является хранение в одном атрибуте разных по смыслу значений. На рис. 6.10 атрибут ***Дата зачисления или увольнения*** хранит информацию как о зачислении, так и об увольнении со­трудника. Если хранится только одно значение, то невозможно понять, какая именно дата внесена. Если внести атрибут-признак типа даты, тип можно будет определить, но останется возможность хранения только одной даты для каждого сотрудника.

Для приведения сущности к первой нормальной форме следует:

* разделить сложные атрибуты на атомарные,
* создать новую сущность,
* перенести в нее все "повторяющиеся" атрибуты,
* выбрать возможный ключ для нового РК (или создать новый РК).
* установить идентифицирующую связь от прежней сущности к новой,  
  РК прежней сущности станет внешним ключом (FK) для новой сущности.

На рис. 6.12 показана сущность ***Сотрудник,*** приведенная к первой нор­мальной форме.

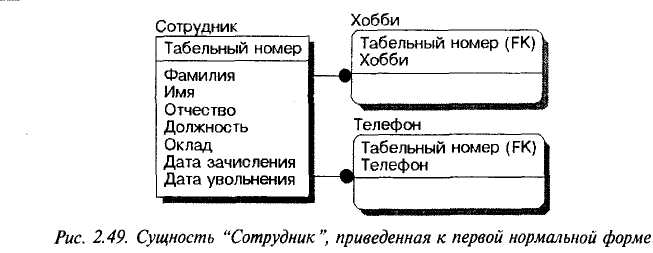


Рисунок 6.12 Нормализованная сущность "Сотрудник"

**Вторая нормальная форма (2NF).** Сущность находится во второй нор­мальной форме, если она находится в первой нормальной форме и каждый неключевой атрибут полностью зависит от первичного ключа (не должно быть зависимости от части ключа). Вторая нормальная форма имеет смысл только для сущностей, имеющих сложный первичный ключ.

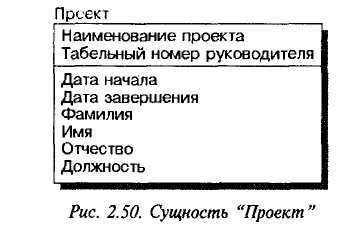


Рисунок 6.13 Ненормализованная сущность "Проект"

Предположим, сущность ***Проект*** содержит информацию о проекте, ко­торым руководит сотрудник, причем информация содержится как непо­средственно о проекте, так и о руководителе проекта (рис. 6.13). Атрибуты ***Фамилия, Имя, Отчество*** и ***Должность*** зависят только от атрибута ***Табель­ный номер руководителя,*** но вовсе не от ***Наименования проекта.*** Другими словами, имеется зависимость только от части ключа.

Для приведения сущности ко второй нормальной форме следует:

* выделить атрибуты, которые зависят только от части первичного ключа, создать новую сущность;
* поместить атрибуты, зависящие от части ключа, в их собственную (новую) сущность;
* установить идентифицирующую связь от прежней сущности к новой (рис. 6.14).



Рисунок 6.14 Нормализованная сущность "Проект"

Вторая нормальная форма позволяет избежать следующих аномалий при выполнении операций:

Обновление (UPDATE). Имеет место дублирование данных о сотрудни­ке, если он руководит несколькими проектами. Если данные о сотруднике изменяются, необходимо менять несколько записей (по числу ведомых проектов).

Вставка (INSERT). Невозможно ввести данные о сотруднике, если он в данный момент не руководит проектами.

Удаление (DELETE). Если сотрудник временно прекращает руководство проектами, данные о нем теряются.

На рис. 6.14 показана сущность ***Проект,*** приведенная ко второй нор­мальной форме.

**Третья нормальная форма (3NF).** Сущность находится в третьей нор­мальной форме, если она находится во второй нормальной форме и ника­кой неключевой атрибут не зависит от другого неключевого атрибута (не должно быть взаимозависимости между неключевыми атрибутами).

На рис. 6.15 сущность *Сотрудник* находится во второй нормальной форме (имеется только один атрибут первичного ключа, поэтому не может быть зависимости неключевых атрибутов от части ключа), но неключевой атрибут ***Оклад*** зависит от другого неключевого атрибута - ***Должности.***

Для приведения сущности к третьей нормальной форме следует:

* создать новую сущность и перенести в нее атрибуты с одной и той же зависимостью от неключевого атрибута;
* использовать атрибут(ы), определяющий эту зависимость, в качестве первичного ключа новой сущности;
* установить неидентифицирующую связь от новой сущности к старой (рис. 2.52).

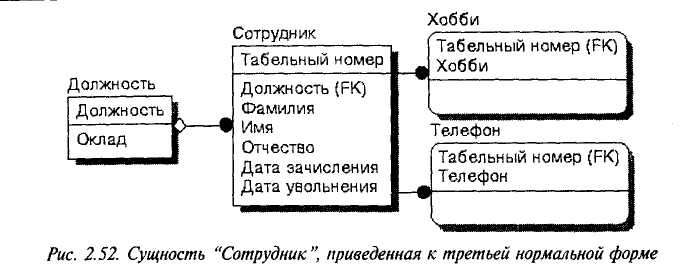


Рисунок 6.15 Сущность "Сотрудник", приведенная к третьей нормальной форме

В третьей нормальной форме каждый атрибут сущности зависит от клю­ча, от всего ключа целиком и ни от чего другого, кроме как от ключа. Третья нормальная форма также позволяет избежать ряда аномалий:

Обновление (UPDATE). Имеет место дублирование данных об окладе, если должность занимают несколько сотрудников. Если оклад соответст­вующих должности меняется, необходимо менять несколько записей (по числу сотрудников на одной должности).

Вставка (INSERT). Невозможно ввести данные об окладе, соответст­вующем должности, если в данный момент нет сотрудника, занимающего эту должность.

Удаление (DELETE). В случае удаления из таблицы сотрудника, зани­мающего уникальную должность, данные об окладе теряются.

**Четвертая нормальная форма (4NF)** требует отсутствия многозначных зависимостей между атрибутами.

В примере на рис. 6.16 (слева) преподаватель читает лекции по несколь­ким предметам и курирует несколько групп студентов. Одна группа студен­тов может изучать несколько предметов, одному предмету могут обучаться несколько групп студентов. Имеется многозначная зависимость между ат­рибутами ***Предмет*** и ***Группа.*** При этом возможна аномалия: если у препо­давателя появляется новая группа, приходится добавлять несколько запи­сей, по числу читаемых предметов.

Для приведения сущности к четвертой нормальной форме следует соз­дать новую сущность и перенести атрибуты с многозначной зависимостью в разные сущности (рис. 6.16, справа). Связь между новыми сущностями при этом устанавливать нельзя, поскольку в результате миграции атрибутов внешних ключей атрибуты с многозначной зависимостью вновь окажутся в одной сущности. Ссылочную целостность в этом случае следует поддержи­вать при помощи триггеров.

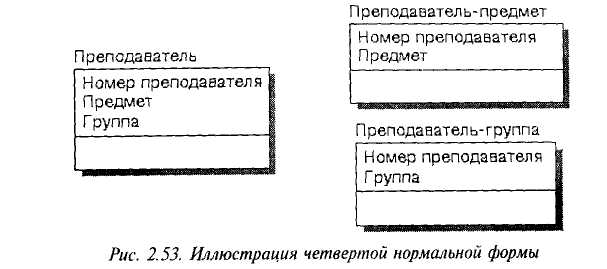


Рисунок 6.16 Иллюстрация четвертой нормальной формы

**6.5 Поддержка нормализации в ERwin.** ERwin не содержит полного алгорит­ма нормализации и не может проводить нормализацию автоматически, од­нако его возможности облегчают создание нормализованной модели дан­ных. Запрет на присвоение неуникальных имен атрибутов в рамках модели (при соответствующей установке опции Unique Name) облегчает соблюде­ние правила "один факт - в одном месте". Имена ролей атрибутов внешних ключей и унификация атрибутов также облегчают построение нормализо­ванной модели.

**Денормализация.** В результате нормализации все взаимосвязи данных становятся правильно определены, исключаются аномалии при оперирова­нии с данными, модель данных становится легче поддерживать. Однако часто нормализация данных не ведет к повышению производительности ИС в целом. Рассмотрим примеры на рис. 6.10 и 6.15. Для получения пол­ной информации о сотруднике из ненормализованной структуры данных достаточно обратиться к одной таблице (см. рис. 6.10). После приведения структуры данных к третьей нормальной форме (рис. 6.15) информация о сотруднике содержится уже в четырех таблицах. Хотя общее количество строк в этих таблицах может быть меньше, чем в исходной (до нормализа­ции), теперь для получения полной информации о сотруднике серверу БД необходимо обращаться одновременно к четырем таблицам (объединение таблиц, join). Время выполнения запроса с объединением может во много раз превосходить время выполнения запроса к одной таблице, другими сло­вами, в приведенном примере общая производительность ИС в результате нормализации скорее всего упадет. В целях повышения производительно­сти при переходе на физический уровень приходится сознательно отходить от нормальных форм для того, чтобы использовать возможности конкрет­ного сервера или ИС в целом.

В отличие от процесса нормализации денормализация не может быть представлена в виде четко сформулированных правил. К сожалению, в ка­ждом конкретном случае приходится искать конкретные решения, которые используют специфику ИС и предметной области и не могут быть тиражи­рованы.

Пример денормализации приведен на рис. 6.17.

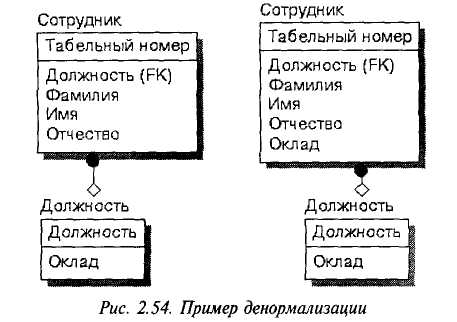


Рисунок 6.17 Пример денормализации

Слева данные находятся в третьей нормальной форме, но для получения из БД информации о сотруднике, включая его оклад, приходится обра­щаться к таблицам ***Должность*** и ***Сотрудник.*** Если в таблицу ***Сотрудник*** добавить колонку ***Оклад*** (рис. 6.17, справа), то тогда при выборке инфор­мации **о** сотруднике достаточно обратиться только к таблице ***Сотрудник*** (исключается объединение). При этом нарушается третья нормальная фор­ма и возникают аномалии, в том числе аномалии при обновлении (если оклад соответствующих должности меняется, необходимо менять несколько записей по числу сотрудников на одной должности). Для решения пробле­мы можно делать выборку только из таблицы ***Сотрудник,*** а обновлять зна­чение оклада только в таблице ***Должность.*** Но при этом возникает проти­воречие между старым значением оклада, хранящимся в таблице ***Сотруд­ник,*** и новыми данными, хранящимися в таблице ***Должность.*** Чтобы избе­жать противоречия, можно создать утилиту (процедуру сервера), которая будет запускаться во время минимальной загрузки сервера, например но­чью, и выравнивать значения колонок. Если при выборке должности всегда необходимо самое свежее значение, то такое решение неприемлемо. Если задача позволяет подождать до начала следующего дня после редактирова­ния справочника ***Должность,*** то такое решение вполне допустимо.

Заметим, что приведенный пример следует воспринимать исключитель­но как иллюстрацию, а не как руководство к действию.

**Поддержка денормализации в ERwin.** Денормализация, как правило, проводится на уровне физической модели. ERwin позволяет сохранить на уровне логической модели нормализованную структуру, при этом постро­ить на уровне физической модели структуру (возможно, денормализован-ную), которая обеспечивает лучшую производительность, используя осо­бенности конкретной СУБД и бизнес-правил предметной области.

ERwin имеет следующую функциональность для поддержки денормали­зации:

Сущности, атрибуты, ключи и домены можно создавать только на уров­не логической модели, включив в соответствующих редакторах опцию Logical Only. Такие объекты не будут ото­бражаться на уровне физической модели и не будут создаваться при гене­рации БД.

Таблицы, колонки, домены и индексы можно создавать только на уров­не физической модели. Например, на уровне только физической модели может быть создана колонка ***Оклад*** таблицы ***Сотрудник,*** см. рис. 6.17.

При автоматическом разрешении связи многие-ко-многим в физической модели создается новая таблица и структура данных может быть дополнена только на уровне физической модели.

ERwin не содержит полного алгортма нормализации и не может проводить нормализацию автоматически, однако его возможности облегчают создание нормализованной модели данных. Запрет на присвоение неуникальных имен атрибутов в рамках модели (при соответствующей установке опции Unique Name) облегчает соблюдение правила "один факт — в одном месте". Имена ролей атрибутов внешних ключей и унификация атрибутов также облегчают построение нормализованной модели.

**6.5 Правила ссылочной целостности**

Правила ссылочной целостности (referential integrity (RI) - логические конструкции, которые выражают бизнес-правила использования данных и представляют собой правила вставки, замены и удаления. При генерации схемы БД на основе опций логической модели, задаваемых в закладке Rolename/RI Actions, будут сгенерированы правила декларативной ссылоч­ной целостности, которые должны быть предписаны для каждой связи, и триггеры, обеспечивающие ссылочную целостность. Триггеры представляют собой программы, выполняемые всякий раз при выполнении команд встав­ки, замены или удаления (INSERT, UPDATE или DELETE).



Рисунок 6.18 Связи сущностей «Команда-Игрок-Гол»

Допустим, существует идентифицирующая связь между сущностями Команда и Игрок. Что будет, если удалить команду?(Рис. 6.18). Экземпляр сущности Игрок не может существовать без команды (атрибут первичного ключа В какой команде иг­рает. Номер команды не может принимать значение NULL), следователь­но, нужно либо запретить удаление команды, пока в ней числится хотя бы один игрок (для удаления команды сначала нужно удалить всех игроков), либо сразу удалять вместе с командой всех ее игроков. Такие правила уда­ления называются "ограничение" и "каскад" (Parent RESTRICT и Parent CASCADE, см. рис. 2.25). Заметим, что сущности Игрок и Гол, в свою оче­редь, тоже связаны идентифицирующей связью и в случае удаления каска­дом команды будут удалены все игроки команды и все голы, которые они забивали. Выполнение команды на удаление одной строки реально может привести к удалению тысячи строк в БД, поэтому использовать правило удаления каскадом следует с осторожностью. В том случае, если установле­но правило ограничения удаления, при попытке выполнить удаление ко­манды, в которой есть хотя бы один игрок, сервер реляционной СУБД воз­вратит ошибку.

Если установлена необязательная неидентифицирующая связь между сущностями Отдел и Сотрудник. Экземпляр сущности Сотрудник может существовать без ссылки на отдел (атрибут внешнего ключа Где ра­ботает. Номер отдела может принимать значение NULL). В этом случае возможно установление правила установки в нуль - SET NULL. При удале­нии отдела атрибут внешнего ключа сущности Сотрудник - Где работает. Номер отдела примет значение NULL. Это означает, что при удалении от­дела сотрудник остается работать в организации не будучи приписан к ка­кому-либо отделу и информация о нем сохраняется.

Возможна установка еще двух правил удаления (если таковые поддер­живаются СУБД):SET DEFAULT - при удалении атрибуту внешнего ключа присваивается значение по умолчанию. Например, при удалении команды игроки могут быть переведены в другую команду.

NONE - при удалении значение атрибута внешнего ключа не меняется. Запись об игроке "повисает в воздухе", т. е. ссылается на несуществующую уже команду. Такая ситуация характерна для "плоских" таблиц. Например, если информация об игроках и командах хранится в dbf-файлах, можно удалить запись о команде, при этом файл игроков "ничего не будет знать" о том, что соответствующей команды не существует. Поэтому в настольных или файл-серверных системах функциональность, обеспечивающая правила ссылочной целостности, реализуется в клиентском приложении.

Правила удаления управляют тем, что будет происходить в БД при уда­лении строки. Аналогично правила вставки и обновления управляют тем,что будет происходить с БД, если строки изменяются или добавляются.

ERwin автоматически присваивает каждой связи значение ссылочной целостности, устанавливаемой по умолчанию, прежде чем добавить ее в диаграмму. Некоторые режимы RI, присваиваемые ERwin по умолчанию или выбранные пользователем приведены в табл. 6.1.

Таблица Значения RI, присваимаемые в Erwin по умолчанию,

а также возможные режимы для каждого типа связи

| *Сущность* | *Идентифицирующая связь* | *Неидентифицирующая связь (Nulls Allowed)* | *Неидентифицирующая связь (No Nulls)* | *Категориальная связь* |
| --- | --- | --- | --- | --- |
| Child Delete  Возможные режимы | RESTRICT, CASCADE,  NONE | RESTRICT, CASCADE,  NONE,  SET NULL,  SET DEFAULT | RESTRICT, CASCADE,  NONE,  SET DEFAULT | RESTRICT, CASCADE,  NONE |
| Child Delete  Режимы по умолчанию | NONE | NONE | NONE | NONE |
| Child Insert  Возможные режимы | RESTRICT, CASCADE,  NONE | RESTRICT, CASCADE,  NONE,  SET NULL,  SET DEFAULT | RESTRICT, CASCADE,  NONE,  SET DEFAULT | RESTRICT, CASCADE,  NONE |
| Child Delete  Режимы по умолчанию | RESTRICT | SET NULL | RESTRICT | RESTRICT |
| Child Update  Возможные режимы | RESTRICT, CASCADE,  NONE | RESTRICT, CASCADE,  NONE,  SET NULL,  SET DEFAULT | RESTRICT, CASCADE,  NONE,  SET DEFAULT | RESTRICT, CASCADE,  NONE |
| Child Update  Режимы по умолчанию | RESTRICT | SET NULL | RESTRICT | RESTRICT |
| Parent Delete  Возможные режимы | RESTRICT, CASCADE,  NONE | RESTRICT, CASCADE,  NONE,  SET NULL,  SET DEFAULT | RESTRICT, CASCADE,  NONE,  SET DEFAULT | RESTRICT, CASCADE,  NONE |
| Parent Delete  Режимы по умолчанию | RESTRICT | SET NULL | RESTRICT | CASCADE |
| Parent Insert  Возможные режимы | RESTRICT, CASCADE,  NONE | RESTRICT, CASCADE,  NONE,  SET NULL,  SET DEFAULT | RESTRICT, CASCADE,  NONE,  SET DEFAULT | RESTRICT, CASCADE,  NONE |
| Parent Insert  Режимы по умолчанию | NONE | NONE | NONE | NONE |
| Parent Update  Возможные режимы | RESTRICT, CASCADE,  NONE | RESTRICT, CASCADE,  NONE,  SET NULL,  SET DEFAULT | RESTRICT, CASCADE,  NONE,  SET DEFAULT | RESTRICT, CASCADE,  NONE |
| Parent Update  Режимы по умолчанию | RESTRICT | SET NULL | RESTRICT | CASCADE |

#### 6.6 Домены

Домен можно определить как совокупность значений, из которых берутся значения атрибутов. Каждый атрибут может быть определен только на одном домене, но на каждом домене может быть определено множество атрибутов. В понятие домена входит не только тип данных, но и область значений данных. Например, можно определить домен "Возраст" как положительное целое число и определить атрибут Возраст сотрудника как принадлежащий этому домену.

В ERwin домен может быть определен только один раз и использоваться как в логической, так и в физической модели.

Домены позволяют облегчить работу с данными как разработчикам на этапе проектирования, так и администраторам БД на этапе эксплуатации системы. На логическом уровне домены можно описать без конкретных физических свойств. На физическом уровне они автоматически получают специфические свойства, которые можно изменить вручную. Так, домен "Возраст" может иметь на логическом уровне тип Number, на физическом уровне колонкам домена будет присвоен тип INTEGER.

Каждый домен может быть описан, снабжен комментарием или свойством, определенным пользователем (UDP).

#### 6.7 Создание физической модели данных

Физическая модель содержит всю информацию, необходимую для реализации конкретной БД. Различают два уровня физической модели:

* трансформационную модель;
* модель СУБД.

Трансформационная модель содержит информацию для реализации отдельного проекта, который может быть частью общей ИС и описывать подмножество предметной области. Данная модель позволяет проектировщикам и администраторам БД лучше представить, какие объекты БД хранятся в словаре данных, и проверить, насколько физическая модель удовлетворяет требованиям к ИС.

**Модель СУБД** автоматически генерируется из трансформационной модели и является точным отображением системного каталога СУБД.

Физический уровень представления модели зависит от выбранного сервера. ERwin поддерживает более 20 реляционных и нереляционных БД.

По умолчанию ERwin генерирует имена таблиц и индексов по шаблону на основе имен соответствующих сущностей и ключей логической модели, которые в дальнейшем могут быть откорректированы вручную. Имена таблиц и колонок будут сгенерированы по умолчанию на основе имен сущностей и атрибутов логической модели.

**Правила валидации и значения по умолчанию**

ERwin поддерживает правила валидации для колонок, а также значение, присваиваемое колонкам по умолчанию.

**Правило валидац**ии задает список допустимых значений для конкретной колонки и/или правила проверки допустимых значений. В список допустимых значений можно вносить новые значения. ERwin позволяет сгенерировать правила валидации соответственно синтаксису выбранной СУБД с учетом границ диапазона или списка значений.

**Значение по умолч**анию – значение, которое нужно ввести в колонку, если никакое другое значение не задано явным образом во время ввода данных. С каждой колонкой или доменом можно связать значение по умолчанию. Список значений можно редактировать.

После создания правила валидации и значения по умолчанию их можно присвоить одной или нескольким колонкам или доменами.

**Индексы**

В БД данные обычно хранятся в том порядке, в котором их ввели в таблицу. Многие реляционные СУБД имеют страничную организацию, при которой таблица может храниться фрагментарно в разных областях диска, причем строки таблицы располагаются на страницах неупорядоченно. Такой способ позволяет быстро вводить новые данные, но затрудняет поиск данных.

Чтобы решить проблему поиска, СУБД используют объекты, называемые индексами. Индекс содержит отсортированную по колонке или нескольким колонкам информацию и указывает на строки, в которых хранится конкретное значение колонки. Поскольку значения в индексе хранятся в определенном порядке, при поиске просматривать нужно значительно меньший объем данных, что существенно уменьшает время выполнения запроса. Индекс рекомендуется создавать для тех колонок, по которым часто производится поиск.

При генерации схемы физической БД ERwin автоматически создает индекс на основе первичного ключа каждой таблицы, а также на основе всех альтернативных ключей и внешних ключей, поскольку эти колонки наиболее часто используются для поиска данных. Можно отказаться от генерации индексов по умолчанию и создать собственные индексы. Для увеличения эффективности поиска администратор БД должен анализировать часто выполняемые запросы и на основе анализа создавать собственные индексы.

**Триггеры и хранимые процедуры**

Триггеры и хранимые процедуры – это именованные блоки кода SQL, которые заранее откомпилированы и хранятся на сервере для того, чтобы быстро производить обработку запросов, валидацию данных и другие часто выполняемые функции. Хранение и выполнение кода на сервере позволяет создавать код только один раз, а не в каждом приложении, работающем с БД. Это экономит время при написании и сопровождении программ. При этом гарантируется, что целостность данных и бизнес-правила поддерживаются независимо от того, какое именно клиентское приложение обращается к данным. Триггеры и хранимые процедуры не требуется пересылать по сети из клиентского приложения, что значительно снижает сетевой трафик.

Хранимой процедурой называется именованный набор предварительно откомпилированных команд SQL, который может вызываться из клиентского приложения или из другой хранимой процедуры.

Триггером называется процедура, которая выполняется автоматически как реакция на событие. Таким событием может быть вставка, изменение или удаление строки в существующей таблице. Триггер сообщает СУБД, какие действия нужно выполнить при выполнении команд SQL INSERT, UPDATE или DELETE для обеспечения дополнительной функциональности, выполняемой на сервере.

**Триггер ссылочной целостности** – это особый вид триггера, используемый для поддержания целостности между двумя таблицами, которые связаны между собой. Если строка в одной таблице вставляется, изменяется или удаляется, то триггер ссылочной целостности сообщает СУБД, что нужно делать с теми строками в других таблицах, у которых значение внешнего ключа совпадает со значением первичного ключа вставленной строки (измененной или удаленной строки).

Для генерации триггеров ERwin использует механизм шаблонов – специальных скриптов, использующих макрокоманды. При генерации кода триггера вместо макрокоманд подставляются имена таблиц, колонок, переменные и другие фрагменты кода, соответствующие синтаксису выбранной СУБД. Шаблоны триггеров ссылочной целостности, генерируемые ERwin по умолчанию, можно изменять.

Для создания и редактирования хранимых процедур ERwin располагает специальными редакторами, аналогичными редакторам, используемым для создания триггеров. В отличие от триггера хранимая процедура не выполняется в ответ на какое-то событие, а вызывается из другой программы, которая передает на сервер имя процедуры. Хранимая процедура более гибкая, чем триггер, поскольку может вызывать другие хранимые процедуры. Ей можно передавать параметры, и она может возвращать параметры, значения и сообщения.

### 6.8 Проектирование хранилищ данных

В хранилища данных помещают данные, которые редко меняются. Хранилища ориентированы на выполнение аналитических запросов, обеспечивающих поддержку принятия решений для руководителей и менеджеров. При проектировании хранилищ данных необходимо выполнять следующие требования:

* хранилище должно иметь понятную для пользователей структуру данных;
* должны быть выделены статические данные, которые модифицируются по расписанию (ежедневно, еженедельно, ежеквартально);
* должны быть упрощены требования к запросам для исключения запросов, требующих множественных утверждений SQL в традиционных реляционных СУБД;
* должна обеспечиваться поддержка сложных запросов SQL, требующих обработки миллионов записей.

Как видно из этих требований, по своей структуре реляционные СУБД существенно отличаются от хранилищ данных. Нормализация данных в реляционных СУБД приводит к созданию множества связанных между собой таблиц. Выполнение сложных запросов неизбежно приводит к объединению многих таблиц, что значительно увеличивает время отклика. Проектирование хранилища данных подразумевает создание денормализованной структуры данных, ориентированных в первую очередь на высокую производительность при выполнении аналитических запросов. Нормализация делает модель хранилища слишком сложной, затрудняет ее понимание и снижает скорость выполнения запроса. Для эффективного проектирования хранилищ данных ERwin использует размерную модель – методологию проектирования, предназначенную специально для разработки хранилищ данных. Размерное моделирование сходно с моделированием связей и сущностей для реляционной модели, но имеет другую цель. Реляционная модель акцентируется на целостности и эффективности ввода данных. Размерная модель ориентирована в первую очередь на выполнение сложных запросов

В размерном моделировании принят стандарт модели, называемый схемой "звезда", которая обеспечивает высокую скорость выполнения запроса посредством денормализации и разделения данных. Невозможно создать универсальную структуру данных, обеспечивающую высокую скорость обработки любого запроса, поэтому схема "звезда" строится для обеспечения наивысшей производительности при выполнении самого важного запроса (или группы запросов).

Схема "звезда" обычно содержит одну большую таблицу, называемую таблицей факта, помещенную в центре. Ее окружают меньшие таблицы, называемые таблицами размерности, которые связаны с таблицей факта радиальными связями.

Для создания БД со схемой "звезда" необходимо проанализировать бизнес-правила предметной области для выяснения центрального запроса. Данные, обеспечивающие выполнение этого запроса, должны быть помещены в центральную таблицу. При проектировании хранилища важно определить источник данных, метод, которым данные извлекаются, преобразуются и фильтруются, прежде чем они импортируются в хранилище. Знания об источнике данных позволяют поддерживать регулярное обновление и проверку качества данных.

#### Вычисление размера БД

ERwin позволяет рассчитать приблизительный размер БД в целом, а также таблиц, индексов и других объектов через определенный период времени после начала эксплуатации ИС. Расчет строится на основе следующих параметров: начальное количество строк; максимальное количество строк; прирост количества строк в месяц. Результаты расчетов сводятся в отчет.

#### Прямое и обратное проектирование

Прямым проектированием называется процесс генерации физической схемы БД из логической модели. При генерации физической схемы ERwin включает триггеры ссылочной целостности, хранимые процедуры, индексы, ограничения и другие возможности, доступные при определении таблиц в выбранной СУБД.

Обратным проектированием называется процесс генерации логической модели из физической БД. Обратное проектирование позволяет конвертировать БД из одной СУБД в другую. После создания логической модели БД путем обратного проектирования можно переключиться на другой сервер и произвести прямое проектирование.

Кроме режима прямого и обратного проектирования программа обеспечивает синхронизацию между логической моделью и системным каталогом СУБД на протяжении всего жизненного цикла создания ИС.

# empty7. Базис языка визуального моделирования

Для создания моделей анализа и проектирования объектно-ориентированных программных систем используют языки визуального моделирования. Появившись сравнительно недавно, в период с 1989 по 1997 год, эти языки уже имеют представительную историю развития.

E:\UML_Picture\РисункиVisio\ИсторияUML.wmf

В настоящее время различают три поколения языков визуального моделирования. И если первое поколение образовали 10 языков, то численность второго поколения уже превысила 50 языков. Среди наиболее популярных языков 2-го поколения можно выделить: язык Буча (G. Booch), язык Рамбо (J. Rumbaugh), язык Джекобсона (I. Jacobson), язык Коада-Йордона (Coad-Yourdon), язык Шлеера-Меллора (Shlaer-Mellor) и т. д [41], [64], [69]. Каждый язык вводил свои выразительные средства, ориентировался на собственный синтаксис и семантику, иными словами — претендовал на роль единственного и неповторимого языка. В результате разработчики (и пользователи этих языков) перестали понимать друг друга. Возникла острая необходимость унификации языков.

Идея унификации привела к появлению языков 3-го поколения. В качестве стандартного языка третьего поколения был принят Unified Modeling Language (UML), создававшийся в 1994-1997 годах (основные разработчики — три «amigos» Г. Буч, Дж. Рамбо, И. Джекобсон)- версия 1.4-1.5, которая описывается в данном конспекте лекций [53]. Данная глава посвящена определению базовых понятий языка UML. В настоящее время разработана версия UML 2.12 (см. Рис.

## Унифицированный язык моделирования

UML — стандартный язык для написания моделей анализа, проектирования и реализации объектно-ориентированных программных систем [23], [53], [67]. UML может использоваться для визуализации, спецификации, конструирования и документирования результатов программных проектов. UML — это не визуальный язык программирования, но его модели прямо транслируются в текст на языках программирования (Java, C++, Visual Basic, Ada 95, Object Pascal) и даже в таблицы для реляционной БД.

Для визуализации, специфицирования, конструирования и документирования программных систем необходимо рассматривать их с различных точек зрения ([см. главу 1](http://interway.narod.ru/context/prog/UML/1/gl1.htm)). Все, кто имеет отношение к проекту, - конечные пользователи, аналитики, разработчики, системные интеграторы, тестировщики, технические писатели и менеджеры проектов - преследуют собственные интересы, и каждый смотрит на создаваемую систему, по-разному в различные моменты ее жизни. Системная архитектура является, пожалуй, наиболее важным артефактом, который используется для управления всевозможными точками зрения и тем самым способствует итеративной и инкрементной разработке системы на всем протяжении ее жизненного цикла.

Архитектура - это совокупность существенных решений касательно:

организации программной системы;

выбора структурных элементов, составляющих систему, и их интерфейсов;

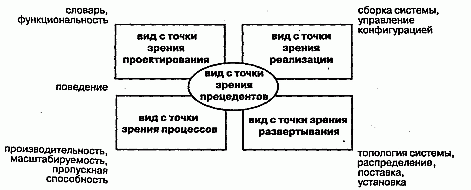
поведения этих элементов, специфицированного в кооперациях с другими элементами;

составления из этих структурных и поведенческих элементов все более и более крупных подсистем;

архитектурного стиля, направляющего и определяющего всю организацию системы: статические и динамические элементы, их интерфейсы, кооперации и способ их объединения.

Архитектура программной системы охватывает не только ее структурные и поведенческие аспекты, но и использование, функциональность, производительность, гибкость, возможности повторного применения, полноту, экономические и технологические ограничения и компромиссы, а также эстетические вопросы.

Как показано на рис. 2.20, архитектура программной системы наиболее оптимально может быть описана с помощью пяти взаимосвязанных видов или представлений, каждый из которых является одной из возможных проекций организации и структуры системы и заостряет внимание на определенном аспекте ее функционирования ([см. главу 31](http://interway.narod.ru/context/prog/UML/1/gl31.htm)).



**Рис. 2.20** Моделирование системной архитектуры

*Вид с точки зрения прецедентов* (Use case view) охватывает прецеденты, которые описывают поведение системы, наблюдаемое конечными пользователями, аналитиками и тестировщиками. Этот вид специфицирует не истинную организацию программной системы, а те движущие силы, от которых зависит формирование системной архитектуры. В языке **UML** статические аспекты этого вида передаются диаграммами прецедентов, а динамические - диаграммами взаимодействия, состояний и действий.

*Вид с точки зрения проектирования* (Design view) охватывает классы, интерфейсы и кооперации, формирующие словарь задачи и ее решения. Этот вид поддерживает прежде всего функциональные требования, предъявляемые к системе,то есть те услуги, которые она должна предоставлять конечным пользователям. С помощью языка **UML** статические аспекты этого вида можно передавать диаграммами классов и объектов, а динамические - диаграммами взаимодействия, состояний и действий.

*Вид с точки зрения процессов* (Process view) охватывает нити и процессы, формирующие механизмы параллелизма и синхронизации в системе. Этот вид описывает главным образом производительность, масштабируемость и пропускную способность системы. В **UML** его статические и динамические аспекты визуализируются теми же диаграммами, что и для вида с точки зрения проектирования, но особое внимание при этом уделяется активным классам, которые представляют соответствующие нити и процессы.

*Вид с точки зрения реализации* (Implementation view) охватывает компоненты и файлы, используемые для сборки и выпуска конечного программного продукта. Этот вид предназначен в первую очередь для управления конфигурацией версий системы, составляемых из независимых (до некоторой степени) компонентов и файлов, которые могут по-разному объединяться между собой. В языке **UML** статические аспекты этого вида передают с помощью диаграмм компонентов, а динамические - с помощью диаграмм взаимодействия, состояний и действий.

*Вид с точки зрения развертывания* (Deployment view) охватывает узлы, формирующие топологию аппаратных средств системы, на которой она выполняется. В первую очередь он связан с распределением, поставкой и установкой частей, составляющих физическую систему. Его статические аспекты описываются диаграммами развертывания, а динамические - диаграммами взаимодействия, состояний и действий.

Каждый из перечисленных видов может считаться вполне самостоятельным, так что лица, имеющие отношение к разработке системы, могут сосредоточиться на изучении только тех аспектов архитектуры, которые непосредственно их касаются. Но нельзя забывать о том, что эти виды взаимодействуют друг с другом. Например, узлы вида с точки зрения развертывания содержат компоненты, описанные для вида с точки зрения реализации, а те, в свою очередь, представляют собой физическое воплощение классов, интерфейсов, коопераций и активных классов из видов с точки зрения проектирования и процессов. **UML** позволяет отобразить каждый из пяти перечисленных видов и их взаимодействия

Словарь UML образуют три разновидности строительных блоков: предметы, отношения, диаграммы.

Предметы — это абстракции, которые являются основными элементами в модели, отношения связывают эти предметы, диаграммы группируют коллекции предметов.

## Предметы в UML

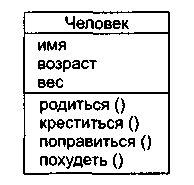
В UML имеются четыре разновидности предметов:

* структурные предметы;
* предметы поведения;
* группирующие предметы;
* поясняющие предметы.

Эти предметы являются базовыми объектно-ориентированными строительными блоками. Они используются для написания моделей.

*Структурные предметы* являются существительными в UML-моделях. Они представляют статические части модели — понятийные или физические элементы. Перечислим восемь разновидностей структурных предметов.

1. *Класс* — описание множества объектов, которые разделяют одинаковые свойства, операции, отношения и семантику (смысл). Класс реализует один или несколько интерфейсов. Как показано на рис. 10.1, графически класс отображается в виде прямоугольника, обычно включающего секции с именем, свойствами (атрибутами) и операциями.



**Рис. 10.1.** Классы

1. *Интерфейс —* набор операций, которые определяют услуги класса или компонента. Интерфейс описывает поведение элемента, видимое извне. Интерфейс может представлять полные услуги класса или компонента или часть таких услуг. Интерфейс определяет набор спецификаций операций (их сигнатуры), а не набор реализаций операций. Графически интерфейс изображается в виде кружка с именем, как показано на рис. 10.2. Имя интерфейса обычно начинается с буквы «I». Интерфейс редко показывают самостоятельно. Обычно его присоединяют к классу или компоненту, который реализует интерфейс.
2. *Кооперация* (сотрудничество) определяет взаимодействие и является совокупностью ролей и других элементов, которые работают вместе для обеспечения коллективного поведения более сложного, чем простая сумма всех элементов. Таким образом, кооперации имеют как структурное, так и поведенческое измерения. Конкретный класс может участвовать в нескольких кооперациях. Эти кооперации представляют реализацию паттернов (образцов), которые формируют систему. Как показано на рис. 10.3, графически кооперация изображается как пунктирный эллипс, в который вписывается ее имя.



**Рис. 10.3.** Кооперации

1. *Актер —* набор согласованных ролей, которые могут играть пользователи при взаимодействии с системой (ее элементами Use Case). Каждая роль требует от системы определенного поведения. Как показано на рис. 10.4, актер изображается как проволочный человечек с именем.



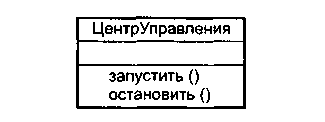
**Рис. 10.4.** Актеры

1. *Элемент Use Case* (Прецедент) — описание последовательности действий (или нескольких последовательностей), выполняемых системой в интересах отдельного актера и производящих видимый для актера результат. В модели элемент Use Case применяется для структурирования предметов поведения. Элемент Use Case реализуется кооперацией. Как показано на рис. 10.5, элемент Use Case изображается как эллипс, в который вписывается его имя.



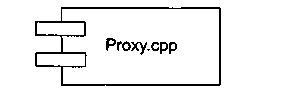
**Рис. 10.5.** Элементы Use Case

1. *Активный класс —* класс, чьи объекты имеют один или несколько процессов (или потоков) и поэтому могут инициировать управляющую деятельность. Активный класс похож на обычный класс за исключением того, что его объекты действуют одновременно с объектами других классов. Как показано на рис. 10.6, активный класс изображается как утолщенный прямоугольник, обычно включающий имя, свойства (атрибуты) и операции.



**Рис. 10.6.** Активные классы

1. *Компонент —* физическая и заменяемая часть системы, которая соответствует набору интерфейсов и обеспечивает реализацию этого набора интерфейсов. В систему включаются как компоненты, являющиеся результатами процесса разработки (файлы исходного кода), так и различные разновидности используемых компонентов (СОМ+-компоненты, Java Beans). Обычно компонент — это физическая упаковка различных логических элементов (классов, интерфейсов и сотрудничеств). Как показано на рис. 10.7, компонент изображается как прямоугольник с вкладками, обычно включающий имя.



**Рис. 10.7.** Компоненты

1. *Узел —* физический элемент, который существует в период работы системы и представляет ресурс, обычно имеющий память и возможности обработки. В узле размещается набор компонентов, который может перемещаться от узла к узлу. Как показано на рис. 10.8, узел изображается как куб с именем.



**Рис. 10.8.** Узлы

*Предметы поведения —* динамические части UML-моделей. Они являются глаголами моделей, представлением поведения во времени и пространстве. Существует две основные разновидности предметов поведения.

1. *Взаимодействие* — поведение, заключающее в себе набор сообщений, которыми обменивается набор объектов в конкретном контексте для достижения определенной цели. Взаимодействие может определять динамику как совокупности объектов, так и отдельной операции. Элементами взаимодействия являются сообщения, последовательность действий (поведение, вызываемое сообщением) и связи (соединения между объектами). Как показано на рис. 10.9, сообщение изображается в виде направленной линии с именем ее операции.



**Рис. 10.9.** Сообщения

1. *Конечный автомат —* поведение, которое определяет последовательность состояний объекта или взаимодействия, выполняемые в ходе его существования в ответ на события (и с учетом обязанностей по этим событиям). С помощью конечного автомата может определяться поведение индивидуального класса или кооперации классов. Элементами конечного автомата являются состояния, переходы (от состояния к состоянию), события (предметы, вызывающие переходы) и действия (реакции на переход). Как показано на рис. 10.10, состояние изображается как закругленный прямоугольник, обычно включающий его имя и его подсостояния (если они есть).



**Рис. 10.10.** Состояния

Эти два элемента — взаимодействия и конечные автоматы — являются базисными предметами поведения, которые могут включаться в UML-модели. Семантически эти элементы ассоциируются с различными структурными элементами (прежде всего с классами, сотрудничествами и объектами).

*Группирующие предметы* — организационные части UML-моделей. Это ящики, по которым может быть разложена модель. Предусмотрена одна разновидность группирующего предмета — пакет.

*Пакет —* общий механизм для распределения элементов по группам. В пакет могут помещаться структурные предметы, предметы поведения и даже другие группировки предметов. В отличие от компонента (который существует в период выполнения), пакет — чисто концептуальное понятие. Это означает, что пакет существует только в период разработки. Как показано на рис. 10.11, пакет изображается как папка с закладкой, на которой обозначено его имя и, иногда, его содержание.



**Рис. 10.11.** Пакеты

*Поясняющие предметы —* разъясняющие части UML-моделей. Они являются замечаниями, которые можно применить для описания, объяснения и комментирования любого элемента модели. Предусмотрена одна разновидность поясняющего предмета — примечание.

*Примечание —* символ для отображения ограничений и замечаний, присоединяемых к элементу или совокупности элементов. Как показано на рис. 10.12, примечание изображается в виде прямоугольника с загнутым углом, в который вписывается текстовый или графический комментарий.



**Рис. 10.12.** Примечания

## Отношения в UML

В UML имеются четыре разновидности отношений:

1) зависимость;

2) ассоциация;

3) обобщение;

4) реализация.

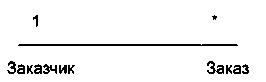
Эти отношения являются базовыми строительными блоками отношений. Они используются при написании моделей.

1. *Зависимость* — семантическое отношение между двумя предметами, в котором изменение в одном предмете (независимом предмете) может влиять на семантику другого предмета (зависимого предмета). Как показано на рис. 10.13, зависимость изображается в виде пунктирной линии, возможно направленной на независимый предмет и иногда имеющей метку.



**Рис. 10.13.** Зависимости

1. *Ассоциация —* структурное отношение, которое описывает набор связей, являющихся соединением между объектами. Агрегация — это специальная разновидность ассоциации, представляющая структурное отношение между целым и его частями. Как показано на рис. 10.14, ассоциация изображается в виде сплошной линии, возможно направленной, иногда имеющей метку и часто включающей другие «украшения», такие как мощность и имена ролей.



**Рис. 10.14.** Ассоциации

1. *Обобщение* — отношение специализации/обобщения, в котором объекты специализированного элемента (потомка, ребенка) могут заменять объекты обобщенного элемента (предка, родителя). Иначе говоря, потомок разделяет структуру и поведение родителя. Как показано на рис. 10.15, обобщение изображается в виде сплошной стрелки с полым наконечником, указывающим на родителя.



**Рис. 10.15.** Обобщения

1. *Реализация —* семантическое отношение между классификаторами, где один классификатор определяет контракт, который другой классификатор обязуется выполнять (к классификаторам относят классы, интерфейсы, компоненты, элементы Use Case, кооперации). Отношения реализации применяют в двух случаях: между интерфейсами и классами (или компонентами), реализующими их; между элементами Use Case и кооперациями, которые реализуют их. Как показано на рис. 10.16, реализация изображается как нечто среднее между обобщением и зависимостью.



**Рис. 10.16.** Реализации

## Диаграммы в UML

*Диаграмма* — графическое представление множества элементов, наиболее часто изображается как связный граф из вершин (предметов) и дуг (отношений). Диаграммы рисуются для визуализации системы с разных точек зрения, затем они отображаются в систему. Обычно диаграмма дает неполное представление элементов, которые составляют систему. Хотя один и тот же элемент может появляться во всех диаграммах, на практике он появляется только в некоторых диаграммах. Теоретически диаграмма может содержать любую комбинацию предметов и отношений, на практике ограничиваются малым количеством комбинаций, которые соответствуют пяти представлениям архитектуры ПС. По этой причине UML включает девять видов диаграмм:

1) диаграммы классов;

2) диаграммы объектов;

3) диаграммы Use Case (диаграммы прецедентов);

4) диаграммы последовательности;

5) диаграммы сотрудничества (кооперации);

6) диаграммы схем состояний;

7) диаграммы деятельности;

8) компонентные диаграммы;

9) диаграммы размещения (развертывания).

*Диаграмма классов* показывает набор классов, интерфейсов, сотрудничеств и их отношений. При моделировании объектно-ориентированных систем диаграммы классов используются наиболее часто. Диаграммы классов обеспечивают статическое проектное представление системы. Диаграммы классов, включающие активные классы, обеспечивают статическое представление процессов системы.

*Диаграмма объектов* показывает набор объектов и их отношения. Диаграмма объектов представляет статический «моментальный снимок» с экземпляров предметов, которые находятся в диаграммах классов. Как и диаграммы классов, эти диаграммы обеспечивают статическое проектное представление или статическое представление процессов системы (но с точки зрения реальных или фототипичных случаев).

*Диаграмма Use Case* (диаграмма прецедентов) показывает набор элементов Use Case, актеров и их отношений. С помощью диаграмм Use Case для системы создается статическое представление Use Case. Эти диаграммы особенно важны при организации и моделировании поведения системы, задании требований заказчика к системе.

Диаграммы последовательности и диаграммы сотрудничества — это разновидности диаграмм взаимодействия.

*Диаграмма взаимодействия* показывает взаимодействие, включающее набор объектов и их отношений, а также пересылаемые между объектами сообщения. Диаграммы взаимодействия обеспечивают динамическое представление системы.

*Диаграмма последовательности —* это диаграмма взаимодействия, которая выделяет упорядочение сообщений по времени.

*Диаграмма сотрудничества* (диаграмма кооперации) — это диаграмма взаимодействия, которая выделяет структурную организацию объектов, посылающих и принимающих сообщения. Диаграммы последовательности и диаграммы сотрудничества изоморфны, что означает, что одну диаграмму можно трансформировать в другую диаграмму.

*Диаграмма схем состояний* показывает конечный автомат, представляет состояния, переходы, события и действия. Диаграммы схем состояний обеспечивают динамическое представление системы. Они особенно важны при моделировании поведения интерфейса, класса или сотрудничества. Эти диаграммы выделяют такое поведение объекта, которое управляется событиями, что особенно полезно при моделировании реактивных систем.

*Диаграмма деятельности —* специальная разновидность диаграммы схем состояний, которая показывает поток от действия к действию внутри системы. Диаграммы деятельности обеспечивают динамическое представление системы. Они особенно важны при моделировании функциональности системы и выделяют поток управления между объектами.

*Компонентная диаграмма* показывает организацию набора компонентов и зависимости между компонентами. Компонентные диаграммы обеспечивают статическое представление реализации системы. Они связаны с диаграммами классов в том смысле, что в компонент обычно отображается один или несколько классов, интерфейсов или коопераций.

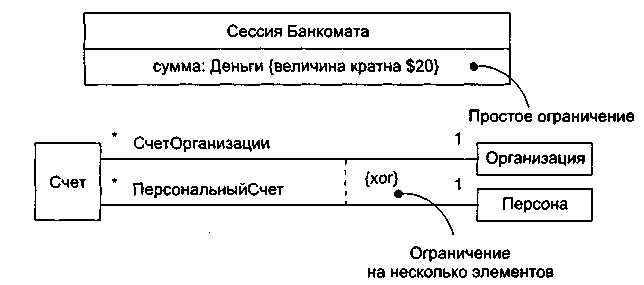
*Диаграмма размещения* (диаграмма развертывания) показывает конфигурацию обрабатывающих узлов периода выполнения, а также компоненты, живущие в них. Диаграммы размещения обеспечивают статическое представление размещения системы. Они связаны с компонентными диаграммами в том смысле, что узел обычно включает один или несколько компонентов.

## Механизмы расширения в UML

UML — развитый язык, имеющий большие возможности, но даже он не может отразить все нюансы, которые могут возникнуть при создании различных моделей. Поэтому UML создавался как открытый язык, допускающий контролируемые рас-. ширения. Механизмами расширения в UML являются:

* ограничения;
* теговые величины;
* стереотипы.

*Ограничение* (constraint) расширяет семантику строительного UML-блока, позволяя добавить новые правила или модифицировать существующие. Ограничение показывают как текстовую строку, заключенную в фигурные скобки {}. Например, на рис. 10.17 введено простое ограничение на свойство *сумма* класса *Сессия Банкомата —* его значение должно быть кратно 20. Кроме того, здесь показано ограничение на два элемента (две ассоциации), оно располагается возле пунктирной линии, соединяющей элементы, и имеет следующий смысл — владельцем конкретного счета не может быть и организация, и персона.



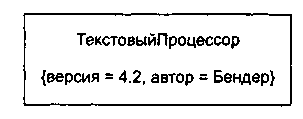
**Рис. 10.17.** Ограничения

*Теговая величина* (tagged value) расширяет характеристики строительного UML-блока, позволяя создать новую информацию в спецификации конкретного элемента. Теговую величину показывают как строку в фигурных скобках {}. Строка имеет вид

имя теговой величины = значение.

Иногда (в случае предопределенных тегов) указывается только имя теговой величины.

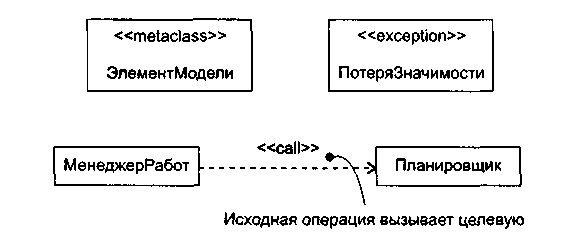
Отметим, что при работе с продуктом, имеющим много реализаций, полезно отслеживать версию и автора определенных блоков. Версия и автор не принадлежат к основным понятиям UML. Они могут быть добавлены к любому строительному блоку (например, к классу) введением в блок новых теговых величин. Например, на рис. 10.18 класс ТекстовыйПроцессор расширен путем явного указания его версии и автора.



**Рис. 10.18.** Расширение класса

*Стереотип* (stereotype) расширяет словарь языка, позволяет создавать новые виды строительных блоков, производные от существующих и учитывающие специфику новой проблемы. Элемент со стереотипом является вариацией существующего элемента, имеющей такую же форму, но отличающуюся по сути. У него могут быть дополнительные ограничения и теговые величины, а также другое визуальное представление. Он иначе обрабатывается при генерации программного кода. Отображают стереотип как имя, указываемое в двойных угловых скобках (или в угловых кавычках).

Примеры элементов со стереотипами приведены на рис. 10.19. Стереотип «exception» говорит о том, что класс ПотеряЗначимости теперь рассматривается как специальный класс, которому, положим, разрешается только генерация и обработка сигналов исключений. Особые возможности метакласса получил класс ЭлементМодели. Кроме того, здесь показано применение стереотипа «call» к отношению зависимости (у него появился новый смысл).



**Рис. 10.19.** Стереотипы

Таким образом, механизмы расширения позволяют адаптировать UML под нужды конкретных проектов и под новые программные технологии. Возможно добавление новых строительных блоков, модификация спецификаций существующих блоков и даже изменение их семантики. Конечно, очень важно обеспечить контролируемое введение расширений.

## Контрольные вопросы

1. Сколько поколений языков визуального моделирования вы знаете?
2. Назовите численность языков визуального моделирования 2-го поколения.
3. Какая необходимость привела к созданию языка визуального моделирования третьего поколения?
4. Поясните назначение UML.
5. Какие строительные блоки образуют словарь UML? Охарактеризуйте их.
6. Какие разновидности предметов UML вы знаете? Их назначение?
7. Перечислите известные вам разновидности структурных предметов UML.
8. Перечислите известные вам разновидности предметов поведения UML.
9. Перечислите известные вам группирующие предметы UML.
10. Перечислите известные вам поясняющие предметы UML.
11. Какие разновидности отношений предусмотрены в UML? Охарактеризуйте каждое отношение.
12. Дайте характеристику диаграммы классов.
13. Дайте характеристику диаграммы объектов.
14. Охарактеризуйте диаграмму Use Case.
15. Охарактеризуйте диаграммы взаимодействия.
16. Дайте характеристику диаграммы последовательности.
17. Дайте характеристику диаграммы сотрудничества.
18. Охарактеризуйте диаграмму схем состояний.
19. Охарактеризуйте диаграмму деятельности.
20. Дайте характеристику компонентной диаграммы.
21. Охарактеризуйте диаграмму размещения.
22. Для чего служат механизмы расширения в UML?
23. Поясните механизм ограничений в UML.
24. Объясните механизм теговых величин в UML.
25. В чем суть механизма стереотипов UML?

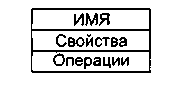
# 8. Статические модели объектно-ориентированных программных систем

Статические модели обеспечивают представление структуры систем в терминах базовых строительных блоков и отношений между ними. «Статичность» этих моделей состоит в том, что здесь не показывается динамика изменений системы во времени. Вместе с тем следует понимать, что эти модели несут в себе не только структурные описания, но и описания операций, реализующих заданное поведение системы. Основным средством для представления статических моделей являются диаграммы классов [8], [23], [53], [67]. Вершины диаграмм классов нагружены классами, а дуги (ребра) — отношениями между ними. Диаграммы используются:

* в ходе анализа — для указания ролей и обязанностей сущностей, которые обеспечивают поведение системы;
* в ходе проектирования — для фиксации структуры классов, которые формируют системную архитектуру.

## Вершины в диаграммах классов

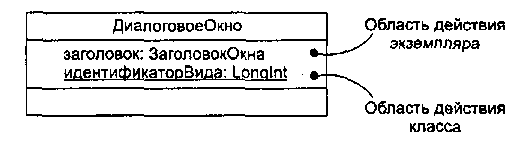
Итак, вершина в диаграмме классов — класс. Обозначение класса показано на рис. 11.1.



**Рис. 11.1.** Обозначение класса

Имя класса указывается всегда, свойства и операции — выборочно. Предусмотрено задание области действия свойства (операции). Если свойство (операция) подчеркивается, его областью действия является класс, в противном случае областью Действия является экземпляр (рис. 11.2).

Что это значит? Если областью действия свойства является класс, то все его экземпляры (объекты) используют общее значение этого свойства, в противном случае у каждого экземпляра свое значение свойства.



**Рис. 11.2.** Свойства уровней класса и экземпляра

### Свойства

Общий синтаксис представления свойства имеет вид

Видимость Имя [Множественность]: Тип = НачальнЗначение {Характеристики}

Рассмотрим видимость и характеристики свойств.

В языке UML определены три уровня видимости:

|  |  |
| --- | --- |
| public  protected  private | Любой клиент класса может использовать свойство (операцию), обозначается символом +  Любой наследник класса может использовать свойство (операцию), обозначается символом #  Свойство (операция) может использоваться только самим классом, обозначается символом - |

**ПРИМЕЧАНИЕ**

Если видимость не указана, считают, что свойство объявлено с публичной видимостью.

Определены три характеристики свойств:

|  |  |
| --- | --- |
| changeable  addOnly  frozen | Нет ограничений на модификацию значения свойства  Для свойств с множественностью, большей единицы; дополнительные значения могут быть добавлены, но после создания значение не может удаляться или изменяться  После инициализации объекта значение свойства не изменяется |

ПРИМЕЧАНИЕ

Если характеристика не указана, считают, что свойство объявлено с характеристикой changeable.

Примеры объявления свойств:

|  |  |
| --- | --- |
| начало  + начало  начало : Координаты  имяфамилия [0..1] : String  левыйУгол : Координаты=(0, 10)  сумма : Integer {frozen} | Только имя  Видимость и имя  Имя и тип  Имя, множественность, тип  Имя, тип, начальное значение  Имя и характеристика |

### Операции

Общий синтаксис представления операции имеет вид

Видимость Имя (Список Параметров): ВозвращаемыйТип {Характеристики}

Примеры объявления операций:

|  |  |
| --- | --- |
| записать  + записать  зарегистрировать) и: Имя, ф: Фамилия)  балансСчета ( ) : Integer  нагревать ( ) (guarded) | Только имя  Видимость и имя  Имя и параметры  Имя и возвращаемый тип  Имя и характеристика |

В сигнатуре операции можно указать ноль или более параметров, форма представления параметра имеет следующий синтаксис:

Направление Имя : Тип = ЗначениеПоУмолчанию

Элемент Направление может принимать одно из следующих значений:

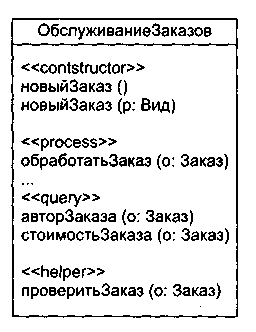
|  |  |
| --- | --- |
| in  out  inout | Входной параметр, не может модифицироваться  Выходной параметр, может модифицироваться для передачи информации в вызывающий объект  Входной параметр, может модифицироваться |

Допустимо применение следующих характеристик операций:

|  |  |
| --- | --- |
| leaf  isQuery  sequential  guarded  concurrent | Конечная операция, операция не может быть полиморфной и не может переопределяться (в цепочке наследования)  Выполнение операции не изменяет состояния объекта  В каждый момент времени в объект поступает только один вызов операций. Как следствие, в каждый момент времени выполняется только одна операция объекта. Другими словами, допустим только один поток вызовов (поток управления)  Допускается одновременное поступление в объект нескольких вызовов, но в каждый момент времени обрабатывается только один вызов охраняемой операции. Иначе говоря, параллельные потоки управления исполняются последовательно (за счет постановки вызовов в очередь)  В объект поступает несколько потоков вызовов операций (из параллельных потоков управления). Разрешается параллельное (и множественное) выполнение операции. Подразумевается, что такие операции являются атомарными |

### Организация свойств и операций

Известно, что пиктограмма класса включает три секции (для имени, для свойств и для операций). Пустота секции не означает, что у класса отсутствуют свойства или операции, просто в данный момент они не показываются. Можно явно определить наличие у класса большего количества свойств или атрибутов. Для этого в конце показанного списка проставляются три точки. Как показано на рис. 11.3, в длинных списках свойств и операций разрешается группировка — каждая группа начинается со своего стереотипа.



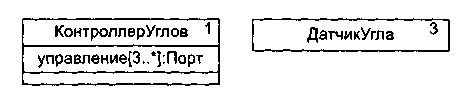
**Рис. 11.3.** Стереотипы для характеристик класса

### Множественность

Иногда бывает необходимо ограничить количество экземпляров класса:

* задать ноль экземпляров (в этом случае класс превращается в утилиту, которая предлагает свои свойства и операции);
* задать один экземпляр (класс-singleton);
* задать конкретное количество экземпляров;
* не ограничивать количество экземпляров (это случай, предполагаемый по умолчанию).

Количество экземпляров класса называется его множественностью. Выражение множественности записывается в правом верхнем углу значка класса. Например, как показано на рис. 11.4, КонтроллерУглов — это класс-singleton, а для класса ДатчикУгла разрешены три экземпляра.

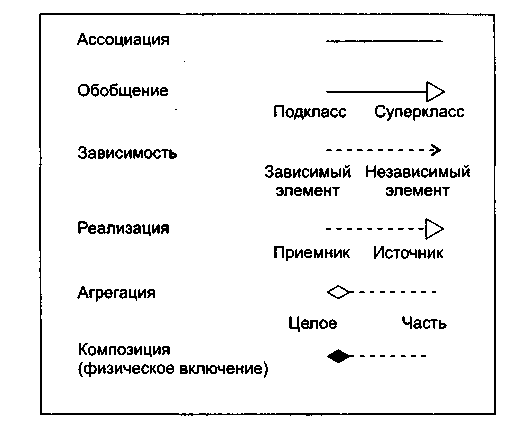


**Рис. 11.4.** Множественность

Множественность применима не только к классам, но и к свойствам. Множественность свойства задается выражением в квадратных скобках, записанным после его имени. Например, на рисунке заданы три и более экземпляра свойства Управление (в экземпляре класса КонтроллерУглов).

## Отношения в диаграммах классов

Отношения, используемые в диаграммах классов, показаны на рис. 11.5.



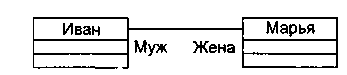
**Рис. 11.5.** Отношения в диаграммах классов

*Ассоциации* отображают структурные отношения между экземплярами классов, то есть соединения между объектами. Каждая ассоциация может иметь метку — *имя,* которое описывает природу отношения. Как показано на рис. 11.6, имени можно придать направление — достаточно добавить треугольник направления, который указывает направление, заданное для чтения имени.



**Рис. 11.6.** Имена ассоциаций

Когда класс участвует в ассоциации, он играет в этом отношении определенную роль. Как показано на рис. 11.7, *роль* определяет, каким представляется класс на одном конце ассоциации для класса на противоположном конце ассоциации.

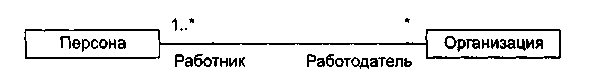


**Рис. 11.7.** Роли

Один и тот же класс в разных ассоциациях может играть разные роли. Часто важно знать, как много объектов может соединяться через экземпляр ассоциации. Это количество называется ложностью роли в ассоциации, записывается в виде выражения, задающего диапазон величин или одну величину (рис. 11.8).

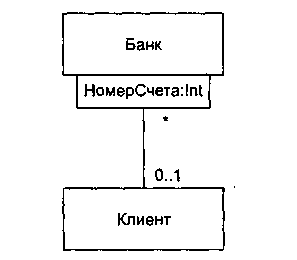
Запись мощности на одном конце ассоциации определяет количество объектов, соединяемых с каждым объектом на противоположном конце ассоциации. Например, можно задать следующие варианты мощности:

* 5 — точно пять;
* \* — неограниченное количество;
* 0..\* — ноль или более;
* 1..\* — один или более;
* 3..7 — определенный диапазон;
* 1..3, 7 — определенный диапазон или число.



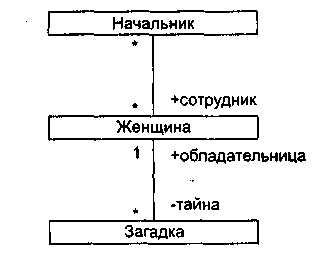
**Рис. 11. 8.** Мощность

Достаточно часто возникает следующая проблема — как для объекта на одном конце ассоциации выделить набор объектов на противоположном конце? Например, рассмотрим взаимодействие между банком и клиентом — вкладчиком. Как показано на рис. 11.9, мы устанавливаем ассоциацию между классом Банк и классом Клиент. В контексте Банка мы имеем НомерСчета, который позволяет идентифицировать конкретного Клиента. В этом смысле НомерСчета является атрибутом ассоциации. Он не является характеристикой Клиента, так как Клиенту не обязательно знать служебные параметры его счета. Теперь для данного экземпляра Банка и данного значения НомераСчета можно выявить ноль или один экземпляр Клиента. В UML для решения этой проблемы вводится *квалификатор —* атрибут ассоциации, чьи значения выделяют набор объектов, связанных с объектом через ассоциацию. Квалификатор изображается маленьким прямоугольником, присоединенным к концу ассоциации. В прямоугольник вписывается свойство — атрибут ассоциации.



**Рис. 11.9.** Квалификация

Кроме того, роли в ассоциациях могут иметь пометки *видимости.* Например, на рис. 11.10 показаны ассоциации между Начальником и Женщиной, а также между Женщиной и Загадкой. Для данного экземпляра Начальника можно определить соответствующие экземпляры Женщины. С другой стороны, Загадка приватна для Женщины, поэтому она недоступна извне. Как показано на рисунке, из объекта Начальника можно перемещаться к экземплярам Женщины (и наоборот), но нельзя видеть экземпляры Загадки для объектов Женщины.

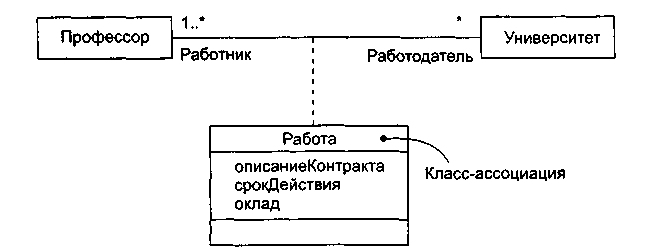


**Рис. 11.10.** Видимость

На конце ассоциации можно задать три уровня видимости, добавляя символ видимости к имени роли:

* по умолчанию для роли задается публичная видимость;
* приватная видимость указывает, что объекты на данном конце недоступны любым объектам вне ассоциации;
* защищенная видимость (protected) указывает, что объекты на данном конце недоступны любым объектам вне ассоциации, за исключением потомков того класса, который указан на противоположном конце ассоциации.

В языке UML ассоциации могут иметь свойства. Как показано на рис, 11.11, такие возможности отображаются с помощью классов-ассоциаций. Эти классы присоединяются к линии ассоциации пунктирной линией и рассматриваются как классы со свойствами ассоциаций или как ассоциации со свойствами классов.



**Рис. 11.11.** Класс-ассоциация

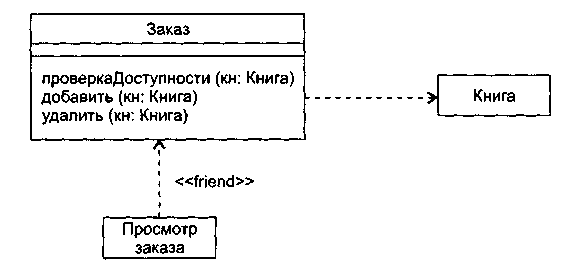
Свойства класса-ассоциации характеризуют не один, а пару объектов, в данном случае — пару экземпляров, Профессор и Университет.

Отношения агрегации и композиции в языке UML считаются разновидностями ассоциации, применяемыми для отображения структурных отношений между «целым» (агрегатом) и его «частями». *Агрегация* показывает отношение по ссылке (в агрегат включены только указатели на части), *композиция —* отношение физического включения (в агрегат включены сами части).

*Зависимость* является отношением использования между клиентом (зависимым элементом) и поставщиком (независимым элементом). Обычно операции клиента:

* вызывают операции поставщика;
* имеют сигнатуры, в которых возвращаемое значение или аргументы принадлежат классу поставщика.

Например, на рис. 11.12 показана зависимость класса Заказ от класса Книга, так как Книга используется в операциях проверкаДоступности, добавить и удалить класса Заказ.



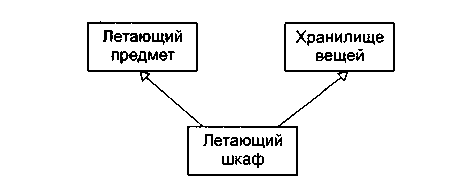
**Рис. 11.12.** Отношения зависимости

На этом рисунке изображена еще одна зависимость, которая показывает, что класс Просмотр Заказа использует класс Заказ. Причем Заказ ничего не знает о Просмотре Заказа. Данная зависимость помечена стереотипом «friend», который расширяет простую зависимость, определенную в языке. Отметим, что отношение зависимости очень разнообразно — в настоящее время язык предусматривает 17 разновидностей зависимости, различаемых по стереотипам.

*Обобщение —* отношение между общим предметом (суперклассом) и специализированной разновидностью этого предмета (подклассом). Подкласс может иметь одного родителя (один суперкласс) или несколько родителей (несколько суперклассов). Во втором случае говорят о множественном наследовании.

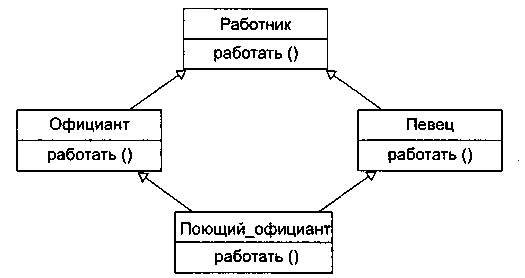
Как показано на рис. 11.13, подкласс Летающий шкаф является наследником суперклассов Летающий предмет и Хранилище вещей. Этому подклассу достаются в наследство все свойства и операции двух классов-родителей.

Множественное наследование достаточно сложно и коварно, имеет много «подводных камней». Например, подкласс Яблочный\_Пирог не следует производить от суперклассов Пирог и Яблоко. Это типичное неправильное использование множественного наследования: потомок наследует все свойства от его родителя, хотя обычно не все свойства применимы к потомку. Очевидно, что Яблочный\_Пирог является Пирогом, но не является Яблоком, так как пироги не растут на деревьях.



**Рис. 11.13.** Множественное наследование

Еще более сложные проблемы возникают при наследовании от двух классов, имеющих общего родителя. Говорят, что в результате образуется ромбовидная решетка наследования (рис. 11.14).

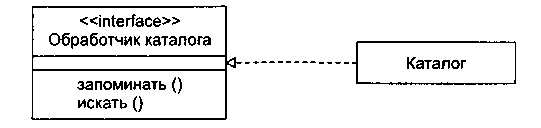


**Рис. 11.14.** Ромбовидная решетка наследования

Полагаем, что в подклассах Официант и Певец операция работать суперкласса Работник переопределена в соответствии с обязанностью подкласса (работа официанта состоит в обслуживании едой, а певца — в пении). Возникает вопрос — какую версию операции работать унаследует Поющий\_официант? А что делать со свойствами, доставшимися в наследство от родителей и общего прародителя? Хотим ли мы иметь несколько копий свойства или только одну?

Все эти проблемы увеличивают сложность реализации, приводят к введению многочисленных правил для обработки особых случаев.

*Реализация* — семантическое отношение между классами, в котором класс-приемник выполняет реализацию операций интерфейса класса-источника. Например, на рис. 11.15 показано, что класс Каталог должен реализовать интерфейс Обработчик каталога, то есть Обработчик каталога рассматривается как источник, а Каталог — как приемник.

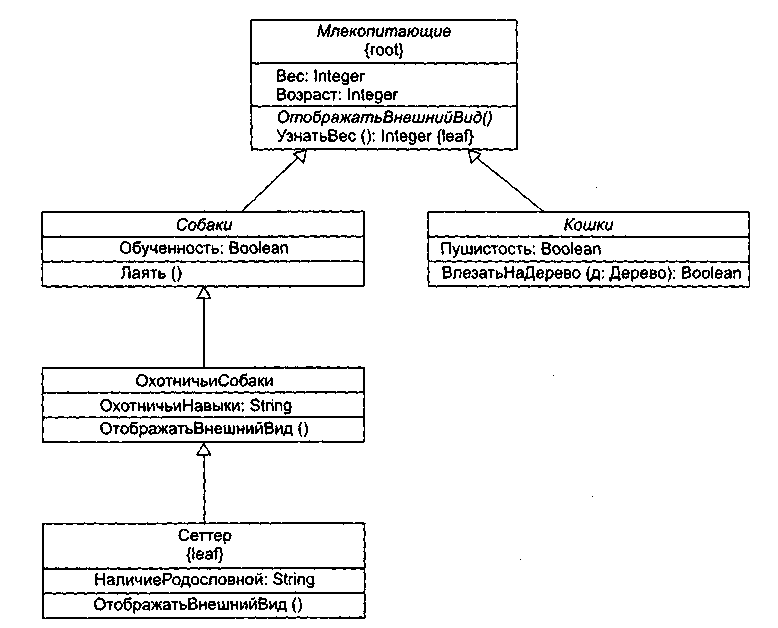


**Рис. 11.15.** Реализация интерфейса

Интерфейс Обработчик каталога позволяет клиентам взаимодействовать с объектами класса Каталог без знания той дисциплины доступа, которая здесь реализована (LIFO — последний вошел, первый вышел; FIFO — первый вошел, первый вышел и т. д.).

## Деревья наследования

При использовании отношений обобщения строится иерархия классов. Некоторые классы в этой иерархии могут быть абстрактными. *Абстрактным* называют класс, который не может иметь экземпляров. Имена абстрактных классов записываются курсивом. Например, на рис. 11.16 показаны абстрактные классы *Млекопитающие, Собаки, Кошки.*



**Рис. 11.16.** Абстрактность и полиморфизм

Кроме того, здесь имеются конкретные классы ОхотничьиСобаки, Сеттер, каждый из которых может иметь экземпляры.

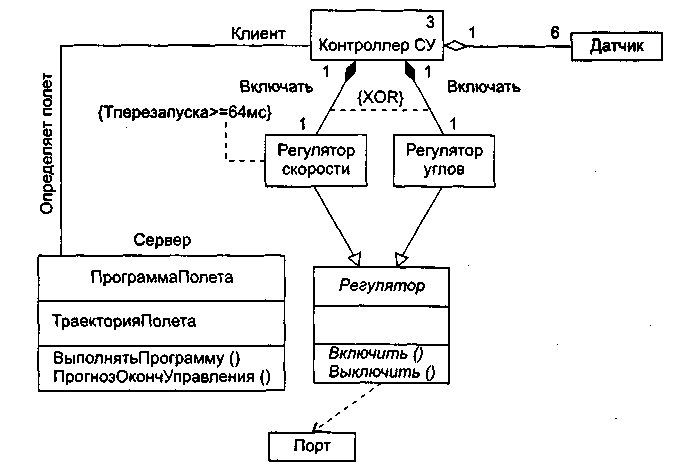
Обычно класс наследует какие-то характеристики класса-родителя и передает свои характеристики классу-потомку. Иногда требуется определить *конечный* класс, который не может иметь детей. Такие классы помечаются теговой величиной (характеристикой) leaf, записываемой за именем класса. Например, на рисунке показан конечный класс Сеттер.

Иногда полезно отметить *корневой* класс, который не может иметь родителей. Такой класс помечается теговой величиной (характеристикой) root, записываемой за именем класса. Например, на рисунке показан корневой класс *Млекопитающие.*

Аналогичные свойства имеют и операции. Обычно операция является полиморфной, это значит, что в различных точках иерархии можно определять операции с похожей сигнатурой. Такие операции из дочерних классов переопределяют поведение соответствующих операций из родительских классов. При обработке сообщения (в период выполнения) производится полиморфный выбор одной из операций иерархии в соответствии с типом объекта. Например, ОтображатьВнешнийВид () и ВлезатьНаДерево (дуб) — полиморфные операции. К тому же операция *Млекопитающие::ОтображатьВнешнийВид ( )* является абстрактной, то есть неполной и требующей для своей реализации потомка. Имя абстрактной операции записывается курсивом (как и имя класса). С другой стороны, Млекопитающие::УзнатьВес () — конечная операция, что отмечается характеристикой leaf. Это значит, что операция не полиморфна и не может перекрываться.

## Примеры диаграмм классов

В качестве первого примера на рис. 11.17 показана диаграмма классов системы управления полетом летательного аппарата.



**Рис. 11.17.** Диаграмма классов системы управления полетом

Здесь представлен класс ПрограммаПолета, который имеет свойство ТраекторияПолета, операцию-модификатор ВыполнятьПрограмму () и операцию-селектор ПрогнозОкончУправления (). Имеется ассоциация между этим классом и классом Контроллер СУ — экземпляры программы задают параметры движения, которые должны обеспечивать экземпляры контроллера.

Класс Контроллер СУ — агрегат, чьи экземпляры включают по одному экземпляру классов Регулятор скорости и Регулятор углов, а также по шесть экземпляров класса Датчик. Экземпляры Регулятора скорости и Регулятора углов включены в агрегат физически (с помощью отношения *композиция),* а экземпляры Датчика — по ссылке, то есть экземпляр Контроллера СУ включает лишь указатели на объекты-датчики. Регулятор скорости и Регулятор углов — это подклассы абстрактного суперкласса *Регулятор,* который передает им в наследство абстрактные операции *Включить ( )* и *Выключить ().* В свою очередь, класс *Регулятор* использует конкретный класс Порт.

Как видим, ассоциация имеет имя (Определяет полет), роли участников ассоциации явно указаны (Сервер, Клиент). Отношения композиции также имеют имена (Включать), причем на эти отношения наложено ограничение — контроллер не может включать Регулятор скорости и Регулятор углов одновременно.

Для класса Контроллер СУ задано ограничение на множественность — допускается не более трех экземпляров этого класса. Класс Регулятор скорости имеет ограничение другого типа — повторное включение его экземпляра разрешается не раньше, чем через 64 мс.

В качестве второго примера на рис. 11.18 приведена диаграмма классов для информационной системы театра. Эту систему образует 6 классов.

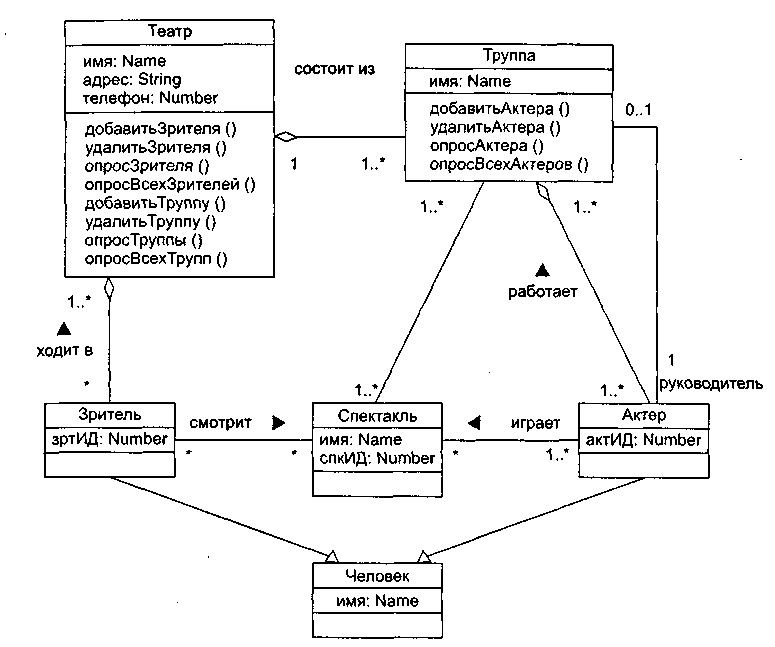
Классы-агрегаты Театр и Труппа имеют операции добавления и удаления своих частей, которые включаются в агрегаты по ссылке. Частями Театра являются Зрители и Труппы, а частями Труппы — Актеры. Отношения агрегации между классом Театр и классами Труппа и Зритель слегка отличны. Театр может состоять из одной или нескольких трупп, но каждая труппа находится в одном и только одном театре. С другой стороны, в театр может ходить любое количество зрителей (включая нулевое количество), причем зритель может посещать один или несколько театров.

Между классами Труппа и Актер существуют два отношения — агрегация и ассоциация. Агрегация показывает, что каждый актер работает в одной или нескольких труппах, а в каждой труппе должен быть хотя бы один актер. Ассоциация отображает, что каждой труппой управляет только один актер — художественный руководитель, а некоторые актеры не являются руководителями.

Ассоциация между классами Спектакль и Актер фиксирует, что в спектакле должен быть занят хотя бы один актер, впрочем, актер может играть в любом количестве спектаклей (или вообще может ничего не играть).

Между классами Спектакль и Зритель тоже определена ассоциация. Она поясняет, что зритель может смотреть любое число спектаклей, а на каждом спектакле может быть любое число зрителей.

И наконец, на диаграмме отображены два отношения наследования, утверждающие, что и в зрителях, и в актерах есть человеческое начало.



**Рис. 11.18.** Диаграмма классов информационной системы театра

## Контрольные вопросы

1. Поясните назначение статических моделей объектно-ориентированных программных систем.
2. Что является основным средством для представления статических моделей?
3. Как используются статические модели?
4. Какие секции входят в графическое обозначение класса?
5. Какие секции класса можно не показывать?
6. Какие имеются разновидности области действия свойства (операции)?
7. Поясните общий синтаксис представления свойства.
8. Какие уровни видимости вы знаете? Их смысл?
9. Какие характеристики свойств вам известны?
10. Поясните общий синтаксис представления операции.
11. Какой вид имеет форма представления параметра операции?
12. Какие характеристики операций вам известны?
13. Что означают три точки в списке свойств (операций)?
14. Как организуется группировка свойств (операций)?
15. Как ограничить количество экземпляров класса?
16. Перечислите известные вам «украшения» отношения ассоциации.
17. Может ли статическая модель программной системы не иметь отношений ассоциации?
18. Какой смысл имеет квалификатор? К чему он относится?
19. Какие отношения могут иметь пометки видимости и что эти пометки обозначают?
20. Какой смысл имеет класс-ассоциация?
21. Чем отличается агрегация от композиции? Разновидностями какого отношения (в UML) они являются?
22. Что обозначает в UML простая зависимость?
23. Какой смысл имеет отношение обобщения?
24. Какие недостатки у множественного наследования?
25. Перечислите недостатки ромбовидной решетки наследования.
26. В чем смысл отношения реализации?
27. Что обозначает мощность «многие-ко-многим» и в каких отношениях она применяется?
28. Что такое абстрактный класс (операция) и как он (она) отображается?
29. Как запретить полиморфизм операции?
30. Как обозначить корневой класс?

# 9. Динамические модели объектно-ориентированных программных систем

Динамические модели обеспечивают представление поведения систем. «Динамизм» этих моделей состоит в том, что в них отражается изменение состояний в процессе работы системы (в зависимости от времени). Средства языка UML для создания динамических моделей многочисленны и разнообразны [8], [23], [41], [53], [67]. Эти средства ориентированы не только на собственно программные системы, но и на отображение требований заказчика к поведению таких систем.

## Моделирование поведения программной системы

Для моделирования поведения системы используют:

* автоматы;
* взаимодействия.

Автомат (State machine) описывает поведение в терминах последовательности состояний, через которые проходит объект в течение своей жизни. Взаимодействие (Interaction) описывает поведение в терминах обмена сообщениями между объектами.

Таким образом, автомат задает поведение системы как цельной, единой сущности; моделирует жизненный цикл единого объекта. В силу этого автоматный подход удобно применять для формализации динамики отдельного трудного для понимания блока системы.

Взаимодействия определяют поведение системы в виде коммуникаций между его частями (объектами), представляя систему как сообщество совместно работающих объектов. Именно поэтому взаимодействия считают основным аппаратом для фиксации полной динамики системы.

Автоматы отображают с помощью:

* диаграмм схем состояний;
* диаграмм деятельности.

Взаимодействия отображают с помощью:

* диаграмм сотрудничества (кооперации);
* диаграмм последовательности.

## Диаграммы схем состояний

Диаграмма схем состояний — одна из пяти диаграмм UML, моделирующих динамику систем. Диаграмма схем состояний отображает конечный автомат, выделяя поток управления, следующий от состояния к состоянию. Конечный автомат — поведение, которое определяет последовательность состояний в ходе существования объекта. Эта последовательность рассматривается как ответ на события и включает реакции на эти события.

Диаграмма схем состояний показывает:

1) набор состояний системы;

2) события, которые вызывают переход из одного состояния в другое;

3) действия, которые происходят в результате изменения состояния.

В языке UML состоянием называют период в жизни объекта, на протяжении которого он удовлетворяет какому-то условию, выполняет определенную деятельность или ожидает некоторого события. Как показано на рис. 12.1, состояние изображается как закругленный прямоугольник, обычно включающий его имя и подсостоя-ния (если они есть).



**Рис. 12.1.** Обозначение состояния

Переходы между состояниями отображаются помеченными стрелками (рис. 12.2).



**Рис. 12.2.** Переходы между состояниями

На рис. 12.2 обозначено: Событие — происшествие, вызывающее изменение состояния, Действие — набор операций, запускаемых событием.

Иначе говоря, события вызывают переходы, а действия являются реакциями на переходы.

Примеры событий:

|  |  |
| --- | --- |
| баланс < 0  помехи  уменьшить(Давление)  after (5 seconds)  when (time = 16:30) | Изменение в состоянии  Сигнал (объект с именем)  Вызов действия  Истечение периода времени  Наступление абсолютного момента времени |

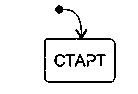
Примеры действий:

|  |  |
| --- | --- |
| Кассир. прекратитьВыплаты( )  flt:= new(Фильтp); Ш.убратьПомехи( )  send Ник. привет | Вызов одной операции  Вызов двух операций  Посылка сигнала в объект Ник |

**ПРИМЕЧАНИЕ**

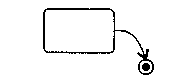
Для отображения посылки сигнала используют специальное обозначение — перед именем сигнала указывают служебное слово send.

Для отображения перехода в начальное состояние принято обозначение, показанное на рис. 12.3.



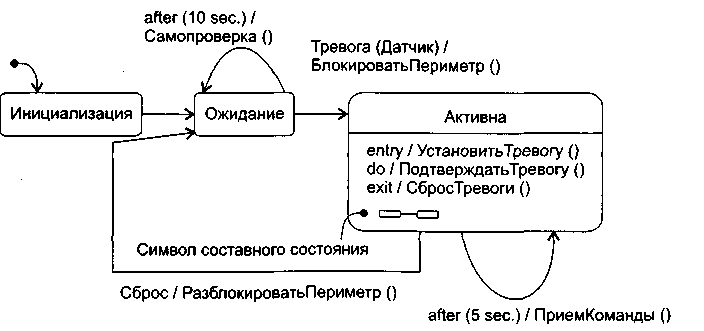
**Рис. 12.3.** Переход в начальное состояние

Соответственно, обозначение перехода в конечное состояние имеет вид, представленный на рис. 12.4.



**Рис. 12.4.** Переход в конечное состояние

В качестве примера на рис. 12.5 показана диаграмма схем состояний для системы охранной сигнализации.



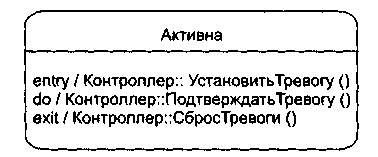
**Рис. 12.5.** Диаграмма схем состояний системы охранной сигнализации

Из рисунка видно, что система начинает свою жизнь в состоянии Инициализация, затем переходит в состояние Ожидание. В этом состоянии через каждые 10 секунд (по событию after (10 sec.)) выполняется самопроверка системы (операция Самопроверка ()). При наступлении события Тревога (Датчик) реализуются действия, связанные с блокировкой периметра охраняемого объекта, — исполняется операция БлокироватьПериметр() и осуществляется переход в состояние Активна. В активном состоянии через каждые 5 секунд по событию after (5 sec.) запускается операция ПриемКоманды(). Если команда получена (наступило событие Сброс), система возвращается в состояние Ожидание. В процессе возврата разблокируется периметр охраняемого объекта (операция РазблокироватьПериметр()).

### Действия в состояниях

Для указания действий, выполняемых при входе в состояние и при выходе из состояния, используются метки entry и exit соответственно.

Например, как показано на рис. 12.6, при входе в состояние Активна выполняется операция УстановитьТревогу() из класса Контроллер, а при выходе из состояния — операция СбросТревоги().



**Рис. 12.6.** Входные и выходные действия и деятельность в состоянии Активна

Действие, которое должно выполняться, когда система находится в данном состоянии, указывается после метки do. Считается, что такое действие начинается при входе в состояние и заканчивается при выходе из него. Например, в состоянии Активна это действие ПодтверждатьТревогу().

### Условные переходы

Между состояниями возможны различные типы переходов. Обычно переход инициируется событием. Допускаются переходы и без событий. Наконец, разрешены условные или охраняемые переходы.

Правила пометки стрелок условных переходов иллюстрирует рис. 12.7.

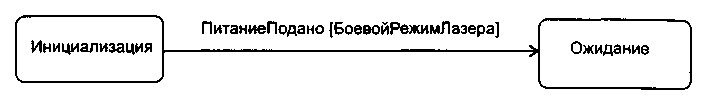


**Рис. 12.7.** Обозначение условного перехода

*Порядок выполнения условного перехода:*

1. происходит событие;
2. вычисляется условие УсловиеПерехода;
3. при УсловиеПерехода=true запускается переход и активизируется действие, в противном случае переход не выполняется.

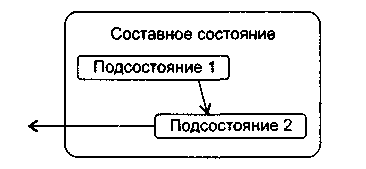
Пример условного перехода между состояниями Инициализация и Ожидание приведен на рис. 12.8. Он происходит по событию ПитаниеПодано, но только в том случае, если достигнут боевой режим лазера.



**Рис. 12.8.** Условный переход между состояниями

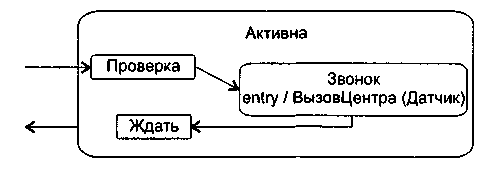
### Вложенные состояния

Одной из наиболее важных характеристик конечных автоматов в UML является подсостояние. Подсостояние позволяет значительно упростить моделирование сложного поведения. Подсостояние — это состояние, вложенное в другое состояние. На рис. 12.9 показано составное состояние, содержащее в себе два подсостояния.



**Рис. 12.9.** Обозначение подсостояний

На рис. 12.10 приведена внутренняя структура составного состояния Активна.

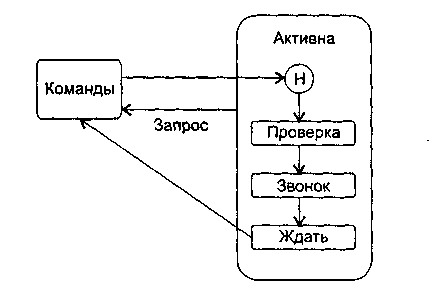


**Рис. 12.10.** Переходы в состоянии Активна

Семантика вложенности такова: если система находится в состоянии Активна, то она должна быть точно в одном из подсостояний: Проверка, Звонок, Ждать. В свою очередь, в подсостояние могут вкладываться другие подсостояния. Степень вложенности подсостояний не ограничивается. Данная семантика соответствует случаю последовательных подсостояний.

Возможно наличие параллельных подсостояний — они выполняются параллельно внутри составного состояния. Графически изображения параллельных подсостояний отделяются друг от друга пунктирными линиями.

Иногда при возврате в составное состояние возникает необходимость попасть в то его подсостояние, которое в прошлый раз было последним. Такое подсостояние называют историческим. Информация об историческом состоянии запоминается. Как показано на рис. 12.11, подобная семантика переходов отображается значком истории — буквой Н внутри кружка.



**Рис. 12.11.** Историческое состояние

При первом посещении состояния Активна автомат не имеет истории, поэтому происходит простой переход в подсостояние Проверка. Предположим, что в подсостоя-нии Звонок произошло событие Запрос. Средства управления заставляют автомат покинуть подсостояние Звонок (и состояние Активна) и вернуться в состояние Команды. Когда работа в состоянии Команды завершается, выполняется возврат в историческое подсостояние состояния Активна. Поскольку теперь автомат запомнил историю, он переходит прямо в подсостояние Звонок (минуя подсостояние Проверка).

Как показано на рис. 12.12, для обозначения составного состояния, имеющего внутри себя скрытые (не показанные на диаграмме) подсостояния, используется символ «очки».



**Рис. 12.12.** Символ состояния со скрытыми подсостояниями

## Диаграммы деятельности

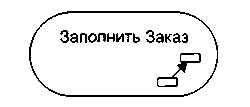
Диаграмма деятельности представляет особую форму конечного автомата, в которой показываются процесс вычислений и потоки работ. В ней выделяются не обычные состояния объекта, а состояния выполняемых вычислений — состояния действий. При этом полагается, что процесс вычислений не прерывается внешними событиями. Словом, диаграммы деятельности очень похожи на блок-схемы алгоритмов.

Основной вершиной в диаграмме деятельности является состояние действия (рис. 12.13), которое изображается как прямоугольник с закругленными боковыми сторонами.



**Рис. 12.13.** Состояние действия

Состояние действия считается атомарным (действие нельзя прервать) и выполняется за один квант времени, его нельзя подвергнуть декомпозиции. Если нужно представить сложное действие, которое можно подвергнуть дальнейшей декомпозиции (разбить на ряд более простых действий), то используют состояние под-деятельности. Изображение состояния под-деятельности содержит пиктограмму в правом нижнем углу (рис. 12.14).



**Рис. 12.14.** Состояние под-деятельности

Фактически в данную вершину вписывается имя другой диаграммы, имеющей внутреннюю структуру.

Переходы между вершинами — состояниями действий — изображаются в виде стрелок. Переходы выполняются по окончании действий.

Кроме того, в диаграммах деятельности используются вспомогательные вершины:

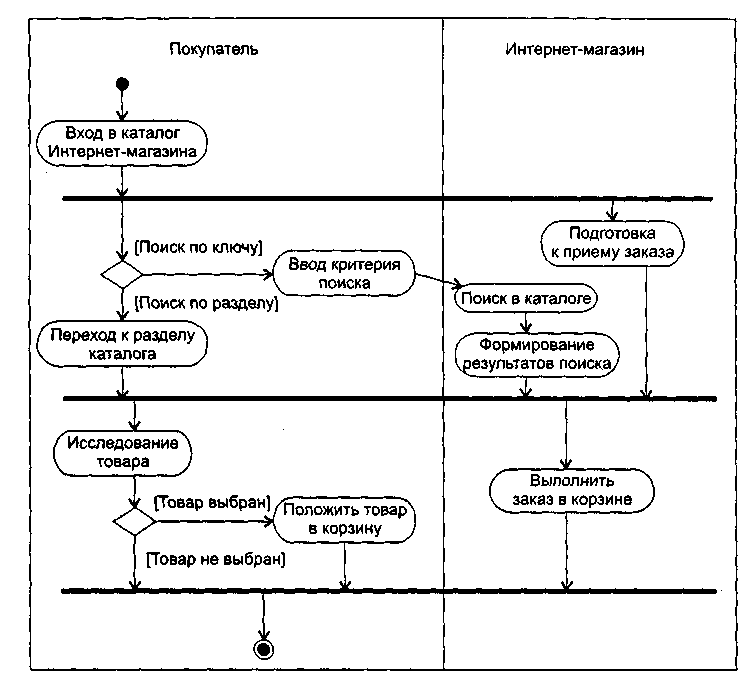
* решение (ромбик с одной входящей и несколькими исходящими стрелками);
* объединение (ромбик с несколькими входящими и одной исходящей стрелкой);
* линейка синхронизации — разделение (жирная горизонтальная линия с одной входящей и несколькими исходящими стрелками);
* линейка синхронизации — слияние (жирная горизонтальная линия с несколькими входящими и одной исходящей стрелкой);
* начальное состояние (черный кружок);
* конечное состояние (незакрашенный кружок, в котором размещен черный кружок меньшего размера).

Вершина «решение» позволяет отобразить разветвление вычислительного процесса, исходящие из него стрелки помечаются сторожевыми условиями ветвления.

Вершина «объединение» отмечает точку слияния альтернативных потоков действий.

Линейки синхронизации позволяют показать параллельные потоки действий, отмечая точки их синхронизации при запуске (момент разделения) и при завершении (момент слияния).

Пример диаграммы деятельности приведен на рис. 12.15. Эта диаграмма описывает деятельность покупателя в Интернет-магазине. Здесь представлены две точки ветвления — для выбора способа поиска товара и для принятия решения о покупке. Присутствуют три линейки синхронизации: верхняя отражает разделение на два параллельных процесса, средняя отражает и разделение, и слияние процессов, а нижняя — только слияние процессов.



**Рис. 12.15.** Диаграмма деятельности покупателя в Интернет-магазине

Дополнительно на этой диаграмме показаны две плавательные дорожки — дорожка покупателя и дорожка магазина, которые разделены вертикальной линией. Каждая дорожка имеет имя и фиксирует область деятельности конкретного лица, обозначая зону его ответственности.

## Диаграммы взаимодействия

Диаграммы взаимодействия предназначены для моделирования динамических аспектов системы. Диаграмма взаимодействия показывает взаимодействие, включающее набор объектов и их отношений, а также пересылаемые между объектами сообщения. Существуют две разновидности диаграммы взаимодействия — диаграмма последовательности и диаграмма сотрудничества. Диаграмма последовательности — это диаграмма взаимодействия, которая выделяет упорядочение сообщений по времени. Диаграмма сотрудничества — это диаграмма взаимодействия, которая выделяет структурную организацию объектов, посылающих и принимающих сообщения. Элементами диаграмм взаимодействия являются участники взаимодействия — объекты, связи, сообщения.

## Диаграммы сотрудничества

Диаграммы сотрудничества отображают взаимодействие объектов в процессе функционирования системы. Такие диаграммы моделируют сценарии поведения системы. В русской литературе диаграммы сотрудничества часто называют диаграммами кооперации.

Обозначение объекта показано на рис. 12.16.



**Рис. 12.16.** Обозначение объекта

Имя объекта подчеркивается и указывается всегда, свойства указываются выборочно. Синтаксис представления имени имеет вид

ИмяОбъекта : ИмяКласса

*Примеры записи имени:*

|  |  |
| --- | --- |
| Адам : Человек  : Пользователь  мойКомпьютер  агент : | Имя объекта и класса  Только имя класса (анонимный объект)  Только имя объекта (подразумевается, что имя класса известно)  Объект — сирота (подразумевается, что имя класса неизвестно) |

Синтаксис представления свойства имеет вид

Имя : Тип = Значение

*Примеры записи свойства:*

|  |  |
| --- | --- |
| номер:Телефон = "7350-420"  активен = True | Имя, тип, значение  Имя и значение |

Объекты взаимодействуют друг с другом с помощью связей — каналов для передачи сообщений. Связь между парой объектов рассматривается как экземпляр ассоциации между их классами. Иными словами, связь между двумя объектами существует только тогда, когда имеется ассоциация между их классами. Неявно все классы имеют ассоциацию сами с собой, следовательно, объект может послать сообщение самому себе.

Итак, связь — это путь для пересылки сообщения. Путь может быть снабжен характеристикой видимости. Характеристика видимости проставляется как стандартный стереотип над дальним концом связи. В языке предусмотрены следующие стандартные стереотипы видимости:

|  |  |
| --- | --- |
| «global»  «local»  «parameter»  «self» | Объект-поставщик находится в глобальной области определения  Объект-поставщик находится в локальной области определения объекта-клиента  Объект-поставщик является параметром операции объекта-клиента  Один и тот же объект является и клиентом, и поставщиком |

Сообщение — это спецификация передачи информации между объектами в ожидании того, что будет обеспечена требуемая деятельность. Прием сообщения рассматривается как событие.

Результатом обработки сообщения обычно является действие. В языке UML моделируются следующие разновидности действий:

|  |  |
| --- | --- |
| Вызов  Возврат  Посылка(Send)  Создание  Уничтожение | В объекте запускается операция  Возврат значения в вызывающий объект  В объект посылается сигнал  Создание объекта, выполняется по стандартному сообщению «create»  Уничтожение объекта, выполняется по стандартному сообщению «destroy» |

Для записи сообщений в языке UML принят следующий синтаксис:

ВозврВеличина := ИмяСообщения (Аргументы),

где ВозврВеличина задает величину, возвращаемую как результат обработки сообщения.

*Примеры записи сообщений:*

|  |  |
| --- | --- |
| Коорд := ТекущПоложение(самолетТ1)  оповещение( )  УстановитьМаршрут(х)  «create» | Вызов операции, возврат значения  Посылка сигнала  Вызов операции с действительным параметром  Стандартное сообщение для создания объекта |

Когда объект посылает сообщение в другой объект (делегируя некоторое действие получателю), объект-получатель, в свою очередь, может послать сообщение в третий объект, и т. д. Так формируется поток сообщений — последовательность управления. Очевидно, что сообщения в последовательности должны быть пронумерованы. Номера записываются перед именами сообщений, направления сообщений указываются стрелками (размещаются над линиями связей).

Наиболее общую форму управления задает процедурный или вложенный поток (поток синхронных сообщений). Как показано на рис. 12.17, процедурный поток рисуется стрелками с заполненными наконечниками.



**Рис. 12.17.** Поток синхронных сообщений

Здесь сообщение 2.1 : Напиток : = Изготовить(Смесь№3) определено как первое сообщение, вложенное во второе сообщение 2 : Заказать(Смесь№3) последовательности, а сообщение 2.2 : Принести(Напиток) — как второе вложенное сообщение. Все сообщения процедурной последовательности считаются синхронными. Работа с синхронным сообщением подчиняется следующему правилу: передатчик ждет до тех пор, пока получатель не примет и не обработает сообщение. В нашем примере это означает, что третье сообщение будет послано только после обработки сообщений 2.1 и 2.2. Отметим, что степень вложенности сообщений может быть любой. Главное, чтобы соблюдалось правило: последовательность сообщений внешнего уровня возобновляется только после завершения вложенной последовательности.

Менее общую форму управления задает асинхронный поток управления. Как показано на рис. 12.18, асинхронный поток рисуется обычными стрелками. Здесь все сообщения считаются асинхронными, при которых передатчик не ждет реакции от получателя сообщения. Такой вид коммуникации имеет семантику почтового ящика — получатель принимает сообщение по мере готовности. Иными словами, передатчик и получатель не синхронизируют свою работу, скорее, один объект «избавляется» от сообщения для другого объекта. В нашем примере сообщение ПодтверждениеВызова определено как второе сообщение в последовательности.



**Рис. 12.18.** Поток асинхронных сообщений

Помимо рассмотренных линейных потоков управления, можно моделировать и более сложные формы — итерации и ветвления.

Итерация представляет повторяющуюся последовательность сообщений. После номера сообщения итерации добавляется выражение

\*[i := 1 .. n].

Оно означает, что сообщение итерации будет повторяться заданное количество раз. Например, четырехкратное повторение первого сообщения РисоватьСторонуПрямоугольника можно задать выражением

1\*[1 := 1 .. 4] : РисоватьСторонуПрямоугольника(i)

Для моделирования ветвления после номера сообщения добавляется выражение условия, например: [х>0]. Сообщение альтернативной ветви помечается таким же номером, но с другим условием: [х<=0]. Пример итерационного и разветвляющегося потока сообщений приведен на рис. 12.19.

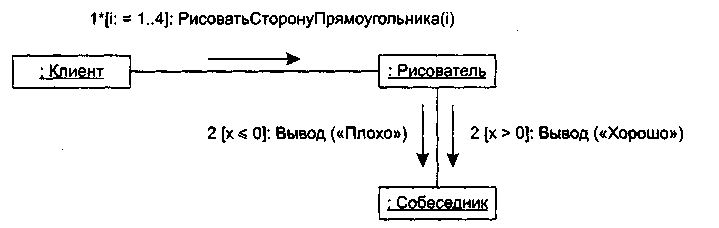
Здесь первое сообщение повторяется 4 раза, а в качестве второго выбирается одно из двух сообщений (в зависимости от значения переменной х). В итоге экземпляр рисователя нарисует на экране прямоугольное окно, а экземпляр собеседника выведет в него соответствующее донесение.

Таким образом, для формирования диаграммы сотрудничества выполняются следующие действия:

1) отображаются объекты, которые участвуют во взаимодействии;

2) рисуются связи, соединяющие эти объекты;

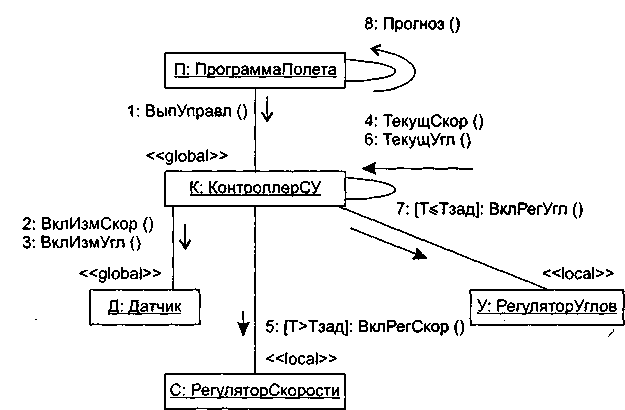
3) связи помечаются сообщениями, которые посылают и получают выделенные объекты.



**Рис. 12.19.** Итерация и ветвление

В итоге формируется ясное визуальное представление потока управления (в контексте структурной организации сотрудничающих объектов).

В качестве примера на рис. 12.20 приведена диаграмма сотрудничества системы управления полетом летательного аппарата.



**Рис. 12.20.** Диаграмма сотрудничества системы управления полетом

На данной диаграмме представлены пять объектов, явно показаны характеристики видимости всех связей системы. Поток управления в системе включает восемь сообщений: четыре асинхронных и четыре синхронных сообщения. Экземпляр Контроллера СУ ждет приема и обработки сообщений:

* ВклРегСкор( );
* ВрРегУгл();
* ТекущСкор();
* ТекущУгл( ).

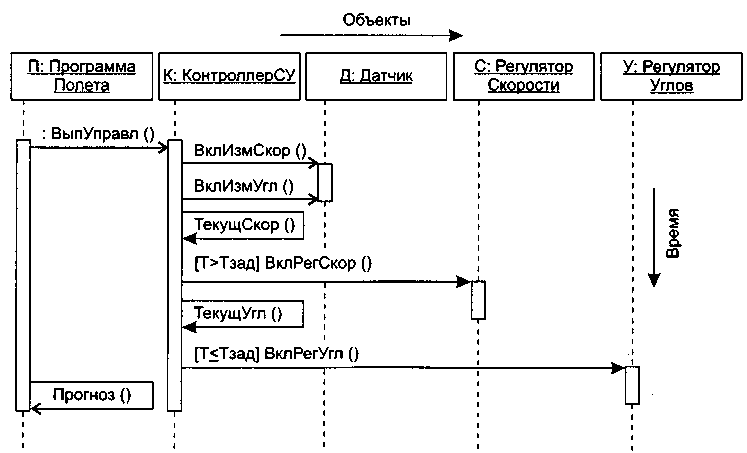
Порядок следования сообщений задан их номерами. Для пятого и седьмого сообщений указаны условия:

* включение Регулятора Скорости происходит, если относительное время полета *Т* больше заданного периода *Тзад*;
* включение Регулятора Углов обеспечивается, если относительное время поле-; та меньше или равно заданному периоду.

## Диаграммы последовательности

Диаграмма последовательности — вторая разновидность диаграмм взаимодействия. Отражая сценарий поведения в системе, эта диаграмма обеспечивает более наглядное представление порядка передачи сообщений. Правда, она не позволяет показать такие детали, которые видны на диаграмме сотрудничества (структурные характеристики объектов и связей).

Графически диаграмма последовательности — разновидность таблицы, которая показывает объекты, размещенные вдоль оси *X,* и сообщения, упорядоченные по времени вдоль оси *Y.*



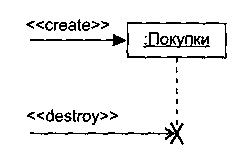
**Рис. 12.21.** Диаграмма последовательности системы управления полетом

Как показано на рис. 12.21, объекты, участвующие во взаимодействии, помещаются на вершине диаграммы, вдоль оси *X.* Обычно слева размещается объект, инициирующий взаимодействие, а справа — объекты по возрастанию подчиненности. Сообщения, посылаемые и принимаемые объектами, помещаются вдоль оси *Y* в порядке возрастания времени от вершины к основанию диаграммы. Используются те же синтаксис и обозначения синхронизации, что и в диаграммах сотрудничества. Таким образом, обеспечивается простое визуальное представление потока управления во времени.

От диаграмм сотрудничества диаграммы последовательности отличают две важные характеристики.

Первая характеристика — *линия жизни* объекта.

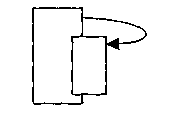
Линия жизни объекта — это вертикальная пунктирная линия, которая обозначает период существования объекта. Большинство объектов существуют на протяжении всего взаимодействия, их линии жизни тянутся от вершины до основания диаграммы. Впрочем, объекты могут создаваться в ходе взаимодействия. Их линии жизни начинаются с момента приема сообщения «create». Кроме того, объекты могут уничтожаться в ходе взаимодействия. Их линии жизни заканчиваются с момента приема сообщения «destroy». Как представлено на рис. 12.22, уничтожение линии жизни отмечается пометкой X в конце линии:



**Рис. 12.22.** Создание и уничтожение объекта

Вторая характеристика — *фокус управления.*

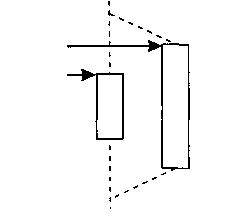
Фокус управления — это высокий тонкий прямоугольник, отображающий период времени, в течение которого объект выполняет действие (свою или подчиненную процедуру). Вершина прямоугольника отмечает начало действия, а основание — его завершение. Момент завершения может маркироваться сообщением возврата, которое показывается пунктирной стрелкой. Можно показать вложение фокуса управления (например, рекурсивный вызов собственной операции). Для этого второй фокус управления рисуется немного правее первого (рис. 12.23).



**Рис. 12.23.** Вложение фокусов управления

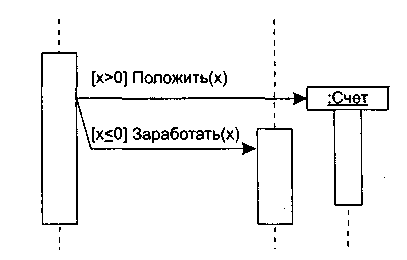
*Замечания.*

1. Для отображения «условности» линия жизни может быть разделена на несколько параллельных линий жизни. Каждая отдельная линия соответствует условному ветвлению во взаимодействии. Далее в некоторой точке линии жизни могут быть снова слиты (рис. 12.24).



**Рис. 12.24.** Параллельные линии жизни

1. Ветвление показывается множеством стрелок, идущих из одной точки. Каждая стрелка отмечается сторожевым условием (рис. 12.25).



**Рис. 12.25.** Ветвление

## Диаграммы Use Case

Диаграмма Use Case определяет поведение системы с точки зрения пользователя. Диаграмма Use Case рассматривается как главное средство для первичного моделирования динамики системы, используется для выяснения требований к разрабатываемой системе, фиксации этих требований в форме, которая позволит проводить дальнейшую разработку. В русской литературе диаграммы Use Case часто называют диаграммами прецедентов, или диаграммами вариантов использования.

В состав диаграмм Use Case входят элементы Use Case, актеры, а также отношения зависимости, обобщения и ассоциации. Как и другие диаграммы, диаграммы Use Case могут включать примечания и ограничения. Кроме того, диаграммы Use Case могут содержать пакеты, используемые для группировки элементов модели в крупные фрагменты.

### Актеры и элементы Use Case

Вершинами в диаграмме Use Case являются актеры и элементы Use Case. Их обозначения показаны на рис. 12.26.

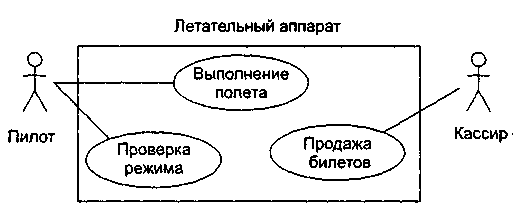
Актеры представляют внешний мир, нуждающийся в работе системы. Элементы Use Case представляют действия, выполняемые системой в интересах актеров.



**Рис. 12.26.** Обозначения актера и элемента Use Case

*Актер —* это роль объекта вне системы, который прямо взаимодействует с ее частью — конкретным элементом (элементом Use Case). Различают актеров и пользователей. Пользователь — это физический объект, который использует систему. Он может играть несколько ролей и поэтому может моделироваться несколькими актерами. Справедливо и обратное — актером могут быть разные пользователи.

Например, для коммерческого летательного аппарата можно выделить двух актеров: пилота и кассира. Сидоров — пользователь, который иногда действует как пилот, а иногда — как кассир. Как изображено на рис. 12.27, в зависимости от роли Сидоров взаимодействует с разными элементами Use Case.



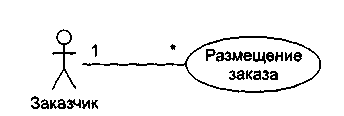
**Рис. 12.27.** Модель Use Case

*Элемент Use Case* — это описание последовательности действий (или нескольких последовательностей), которые выполняются системой и производят для отдельного актера видимый результат.

Один актер может использовать несколько элементов Use Case, и наоборот, один элемент Use Case может иметь несколько актеров, использующих его. Каждый элемент Use Case задает определенный путь использования системы. Набор всех элементов Use Case определяет полные функциональные возможности системы.

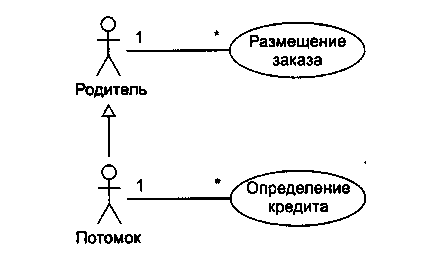
### Отношения в диаграммах Use Case

Между актером и элементом Use Case возможен только один вид отношения — ассоциация, отображающая их взаимодействие (рис. 12.28). Как и любая другая ассоциация, она может быть помечена именем, ролями, мощностью.



**Рис. 12.28.** Отношение ассоциации

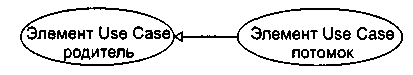
Между актерами допустимо отношение обобщения (рис. 12.29), означающее, что экземпляр потомка может взаимодействовать с такими же разновидностями экземпляров элементов Use Case, что и экземпляр родителя.



**Рис. 12.29.** Отношение обобщения между актерами

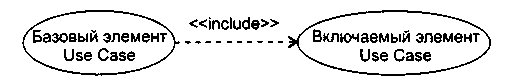
Между элементами Use Case определены отношение обобщения и две разновидности отношения зависимости — включения и расширения.

Отношение обобщения (рис. 12.30) фиксирует, что потомок наследует поведение родителя. Кроме того, потомок может дополнить или переопределить поведение родителя. Элемент Use Case, являющийся потомком, может замещать элемент Use Case, являющийся родителем, в любом месте диаграммы.



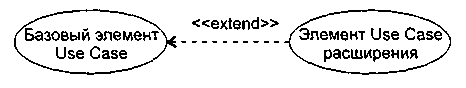
**Рис. 12.30.** Отношение обобщения между элементами Use Case

Отношение включения (рис. 12.31) между элементами Use Case означает, что базовый элемент Use Case *явно* включает поведение другого элемента Use Case в точке, которая определена в базе. Включаемый элемент Use Case никогда не используется самостоятельно — его конкретизация может быть только частью другого, большего элемента Use Case. Отношение включения является примером отношения делегации. При этом в отдельное место (включаемый элемент Use Case) помещается определенный набор обязанностей системы. Далее остальные части системы могут агрегировать в себя эти обязанности (при необходимости).

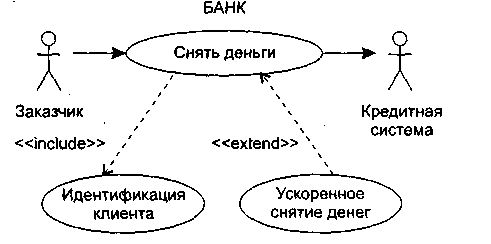


**Рис. 12.31.** Отношение включения между элементами Use Case

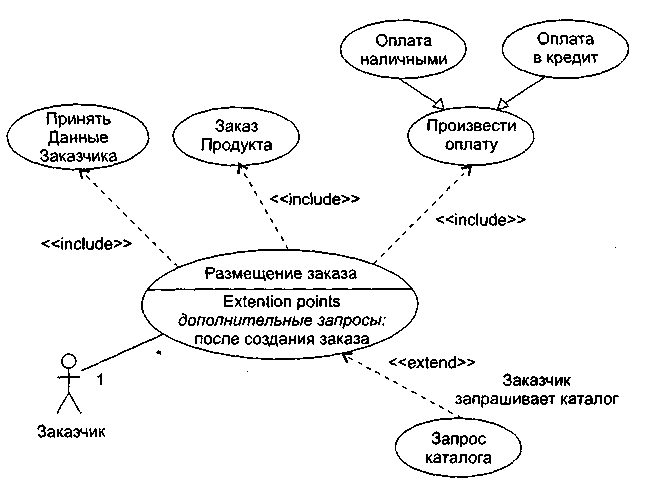
Отношение расширения (рис. 12.32) между элементами Use Case означает, что базовый элемент Use Case *неявно* включает поведение другого элемента Use Case в точке, которая определяется косвенно расширяющим элементом Use Case. Базовый элемент Use Case может быть автономен, но при определенных условиях его поведение может расширяться поведением из другого элемента Use Case. Базовый элемент Use Case может расширяться только в определенных точках — точках расширения. Отношение расширения применяется для моделирования выбираемого поведения системы. Таким способом можно отделить обязательное поведение от необязательного поведения. Например, можно использовать отношение расширения для отдельного подпотока, который выполняется только при определенных условиях, находящихся вне поля зрения базового элемента Use Case. Наконец, можно моделировать отдельные потоки, вставка которых в определенную точку управляется актером.



**Рис. 12.32.** Отношение расширения между элементами Use Case



**Рис. 12.33.** Простейшая диаграмма Use Case для банка



**Рис. 12.34.** Диаграмма Use Case для обслуживания заказчика

Пример простейшей диаграммы Use Case, в которой использованы отношения включения и расширения, приведен на рис. 12.33.

Как показано на рис. 12.34, внутри элемента Use Case может быть дополнительная секция с заголовком Extention points. В этой области перечисляются точки расширения. В указанную здесь точку *дополнительные запросы* вставляется последовательность действий от расширяющего элемента Use Case Запрос каталога. Для справки отмечено, что точка расширения размещена после действий, обеспечивающих создание заказа. На этом же рисунке отображены отношения наследования между элементами Use Case. Видно, что элементы Use Case Оплата наличными и Оплата в кредит наследуют поведение элемента Use Case Произвести оплату и являются его специализациями.

### Работа с элементами Use Case

Элемент Use Case описывает, *что* должна делать система, но не определяет, *как* она должна это делать. При моделировании это позволяет отделять внешнее представление системы от ее внутреннего представления.

Поведение элемента Use Case описывается потоком событий. Начальное описание выполняется в текстовой форме, прозрачной для пользователя системы. В потоке событий выделяют:

* основной поток и альтернативные потоки поведения;
* как и когда стартует и заканчивается элемент Use Case;
* когда элемент Use Case взаимодействует с актерами;
* какими данными обмениваются актер и система.

Для уточнения и формализации потоков событий используют диаграммы последовательности. Обычно одна диаграмма последовательности определяет главный поток в элементе Use Case, а другие диаграммы — потоки исключений.

В общем случае один элемент Use Case описывает набор последовательностей, в котором каждая последовательность представляет возможный поток событий. Каждая последовательность называется сценарием. Сценарий — конкретная последовательность действий, которая иллюстрирует поведение. Сценарии являются для элемента Use Case почти тем же, чем являются экземпляры для класса. Говорят, что сценарий — это экземпляр элемента Use Case.

## Спецификация элементов Use Case

Спецификация элемента Use Case — основной источник информации для выполнения анализа и проектирования системы. Очень важно, чтобы содержание спецификации было представлено в полной и конструктивной форме. В общем случае спецификация включает главный поток, подпотоки и альтернативные потоки поведения. В качестве шаблона спецификации представим описание элемента Use Case *«Покупать авиабилет»* для модели информационной системы авиакассы.

**Предусловие:** перед началом этого элемента Use Case должен быть выполнен элемент Use Case «Заполнить базу данных авиарейсов».

### Главный поток

Этот элемент Use Case начинается, когда покупатель регистрируется в системе и вводит свой пароль. Система проверяет, правилен ли пароль (Е-1), и предлагает покупателю выбрать одно из действий: СОЗДАТЬ, УДАЛИТЬ, ПРОВЕРИТЬ, ВЫПОЛНИТЬ, ВЫХОД.

1. Если выбрано действие СОЗДАТЬ, выполняется подпоток S-1: создать заказ авиабилета.

2. Если выбрано действие УДАЛИТЬ, выполняется подпоток S-2: удалить заказ авиабилета.

3. Если выбрано действие ПРОВЕРИТЬ, выполняется подпоток S-3: проверить заказ авиабилета.

4. Если выбрано действие ВЫПОЛНИТЬ, выполняется подпоток S-4: реализовать заказ авиабилета.

5. Если выбрано действие ВЫХОД, элемент Use Case заканчивается.

### Подпотоки

**S-1: создать заказ авиабилета.** Система отображает диалоговое окно, содержащее поля для пункта назначения и даты полета. Покупатель вводит пункт назначения и дату полета (Е-2). Система отображает параметры авиарейсов (Е-3). Покупатель выбирает авиарейс. Система связывает покупателя с выбранным авиарейсом (Е-4). Возврат к началу элемента Use Case.

**S-2: удалить заказ авиабилета.** Система отображает параметры заказа. Покупатель подтверждает решение о ликвидации заказа (Е-5). Система удаляет связь с покупателем (Е-6). Возврат к началу элемента Use Case.

**S-3: проверить заказ авиабилета.** Система выводит (Е-7) и отображает параметры заказа авиабилета: номер рейса, пункт назначения, дата, время, место, цену. Когда покупатель указывает, что он закончил проверку, выполняется возврат к началу элемента Use Case.

**S-4: реализовать заказ авиабилета.** Система запрашивает параметры кредитной карты покупателя. Покупатель вводит параметры своей кредитной карты (Е-8). Возврат к началу элемента Use Case.

### Альтернативные потоки

**Е-1: введен неправильный ID-номер покупателя.** Покупатель может повторить ввод ID-номера или прекратить элемент Use Case.

**Е-2: введены неправильные пункт назначения/дата полета.** Покупатель может повторить ввод пункта назначения/даты полета или прекратить элемент Use Case.

**Е-3: нет подходящих авиарейсов.** Покупатель информируется, что в данное время такой полет невозможен. Возврат к началу элемента Use Case.

**Е-4: не может быть создана связь между покупателем и авиарейсом.** Информация сохраняется, система создаст эту связь позже. Элемент Use Case продолжается.

**Е-5: введен неправильный номер заказа.** Покупатель может повторить ввод правильного номера заказа или прекратить элемент Use Case.

**Е-6: не может быть удалена связь между покупателем и авиарейсом.** Информация сохраняется, система будет удалять эту связь позже. Элемент Use Case продолжается.

**Е-7: система не может вывести информацию заказа.** Возврат к началу элемента Use Case.

**Е-8: некорректные параметры кредитной карты.** Покупатель может повторить ввод параметров карты или прекратить элемент Use Case.

Таким образом, в данной спецификации зафиксировано, что внутри элемента Use Case находится один основной поток и двенадцать вспомогательных потоков действий. В дальнейшем разработчик может принять решение о выделении из этого элемента Use Case самостоятельных элементов Use Case. Очевидно, что если самостоятельный элемент Use Case содержит подпоток, то его следует подключать к базовому элементу Use Case отношением include. В свою очередь, самостоятельный элемент Use Case с альтернативным потоком подключается к базовому элементу Use Case отношением extend.

## Пример диаграммы Use Case

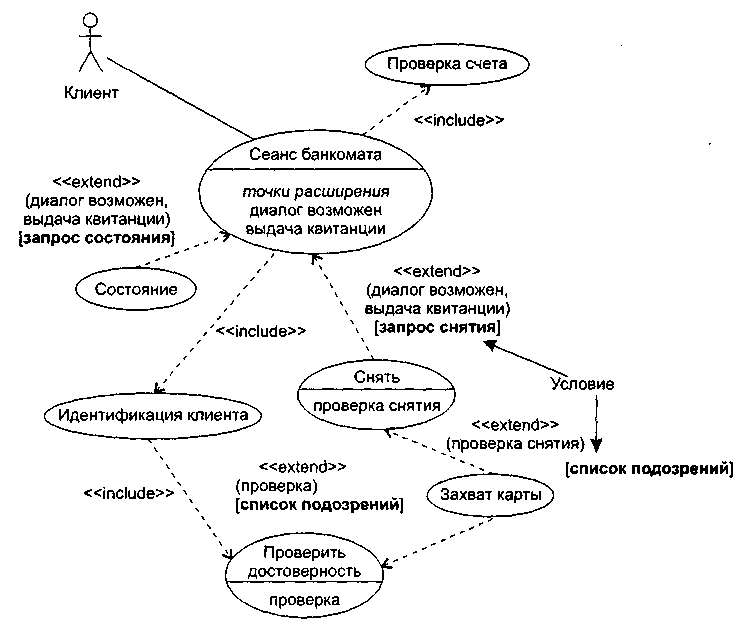
Наибольшие трудности при построении диаграмм Use Case вызывает применение отношений включения и расширения. Очень важно разобраться в отличительных особенностях этих отношений, специфике взаимодействия элементов Use Case, соединяемых с их помощью.

Пример диаграммы Use Case, в которой использованы отношения включения и расширения, приведен на рис. 12.35.

В этой диаграмме один базовый элемент Use Case Сеанс банкомата, который взаимодействует с актером Клиент. К базовому элементу Use Case подключены два расширяющих элемента Use Case (Состояние, Снять) и два включаемых элемента Use Case (Идентификация клиента, Проверка счета). В свою очередь, к элементу Use Case Идентификация клиента подключен включаемый элемент Use Case Проверить достоверность, а к элементу Use Case Снять — расширяющий элемент Use Case Захват карты (он же расширяет элемент Use Case Проверить достоверность).

Видим, что элемент Use Case Сеанс банкомата имеет две точки расширения (диалог возможен, выдача квитанции), а элементы Use Case Снять и Проверить достоверность — по одной точке расширения (проверка снятия и проверка соответственно). В точки расширения возможна вставка поведения из расширяющего элемента Use Case. Вставка происходит, если выполняется условие расширения:

* для расширяющего элемента Use Case Состояние — это условие запрос состояния;
* для расширяющего элемента Use Case Снять — это условие запрос снятия;
* для расширяющего элемента Use Case Захват карты — это условие список подозрений.



**Рис. 12.35.** Использование включения и расширения

Для расширяемого (базового) элемента Use Case эти условия являются внешними, то есть формируемыми вне его. Иными словами, элементу Use Case Сеанс банкомата ничего не известно об условиях запрос состояния и запрос снятия, а элементам Use Case Снять и Проверить достоверность — об условии список подозрений. Условия расширения являются следствиями событий, происходящих во внешней среде.

Стрелки расширения в диаграмме подписаны. Помимо стереотипа, здесь указаны:

* в круглых скобках — имена точек расширения;
* в квадратных скобках — условие расширения.

Описание расширяющего элемента Use Case разделено на сегменты, каждый сегмент обслуживает одну точку расширения базового элемента Use Case.

Количество сегментов расширяющего элемента Use Case равно количеству точек расширения базового элемента Use Case. Первый сегмент расширяющего элемента Use Case начинается с условия расширения, условие записывается только один раз, его действие распространяется и на все остальные сегменты.

Поведение базового элемента Use Case задается внутренним потоком событий, вплоть до точки расширения. В точке расширения возможно выполнение расширяющего элемента Use Case, после чего возобновляется работа внутреннего потока.

Спецификации элементов Use Case рассматриваемой диаграммы имеют следующий вид:

*Элемент Use Case* Сеанс банкомата

|  |  |
| --- | --- |
| include (Идентификация клиента)  include (Проверка счета)  (диалог возможен)  напечатать заголовок квитанции  (выдача квитанции)  конец сеанса | //включение  //включение  //первая точка расширения  //вторая точка расширения |

*Расширяющий элемент Use Case* Состояние

|  |  |
| --- | --- |
| сегмент | //начало первого сегмента |
| принять запрос состояния | //условие расширения |
| отобразить информацию о состоянии счета |  |
| сегмент | //вторая точка расширения |
| конец сеанса |  |

*Расширяющий элемент Use Case* Снять

|  |  |
| --- | --- |
| сегмент | //начало первого сегмента |
| принять запрос снятия | //условие расширения |
| определить сумму |  |
| (проверка снятия) | //точка расширения |
| сегмент | //начало второго сегмента |
| напечатать снимаемую сумму |  |
| выдать наличные деньги |  |

*Расширяющий элемент Use Case* Захват карты

|  |  |
| --- | --- |
| сегмент  принять список подозрений  проглотить карту  конец сеанса | //начало единственного сегмента  //условие расширения |

*Включаемый элемент Use Case* Идентификация клиента

|  |  |
| --- | --- |
| получить имя клиента  include (Проверить достоверность)  получить номер счета клиента | //включение |

*Включаемый элемент Use Case* Проверка счета

|  |
| --- |
| установить соединение с базойданных счетов  получить состояние и ограничения счета |

*Включаемый элемент Use Case* Проверить достоверность

|  |
| --- |
| установить соединение с базой данных клиентов  получить параметры клиента  (проверка) //точка расширения |

Опишем возможное поведение модели, задаваемое этой диаграммой.

Актер Клиент инициирует действия базового элемента Use Case Сеанс банкомата. На первом шаге запускается включаемый элемент Use Case Идентификация клиента. Этот элемент Use Case получает имя клиента и запускает элемент Use Case Проверить достоверность, в результате чего устанавливается соединение с базой данных клиентов и получаются параметры клиента.

Если к этому моменту исполняется условие расширения список подозрений, то «срабатывает» расширяющий элемент Use Case Захват карты, карта арестовывается и работа системы прекращается.

В противном случае происходит возврат к элементу Use Case Идентификация клиента, который получает номер счета клиента и возвращает управление базовому элементу Use Case.

Базовый элемент Use Case переходит ко второму шагу работы — вызывает включаемый элемент Use Case Проверка счета, который устанавливает соединение с базой данных счетов и получает состояние и ограничения счета.

Управление опять возвращается к базовому элементу Use Case. Базовый элемент Use Case переходит к первой точке расширения диалог возможен. В этой точке возможно подключение одного из двух расширяющих элементов Use Case.

Положим, что к этому моменту выполняется условие расширения запрос состояния, поэтому запускается первый сегмент элемента Use Case Состояние. В результате отображается информация о состоянии счета и управление передается базовому элементу Use Case. В базовом элементе Use Case печатается заголовок квитанции и обеспечивается переход ко второй точке расширения выдача квитанции.

Поскольку в активном состоянии продолжает находиться расширяющий элемент Use Case Состояние, запускается его второй сегмент — в квитанции печатается информация о состоянии счета.

В последний раз управление возвращается к базовому элементу Use Case — завершается сеанс работы банкомата.

## Построение модели требований

Напомним, что основное назначение диаграмм Use Case — определение требований заказчика к будущему программному приложению. Обсудим разработку ПО для машины утилизации, которая принимает использованные бутылки, банки, ящики. Для определения элементов Use Case, которые должны выполняться в системе, вначале определяют актеров.

**Выбор актеров**

Поиск актеров — большая работа. Сначала выделяют первичных актеров, использующих систему по прямому назначению. Каждый из первичных актеров участвует в выполнении одной или нескольких главных задач системы. В нашем примере первичным актером является Потребитель. Потребитель кладет в машину бутылки, получает квитанцию от машины.

Кроме первичных, существуют и вторичные актеры. Они наблюдают и обслуживают систему. Вторичные актеры существуют только для того, чтобы первичные актеры могли использовать систему. В нашем примере вторичным актером является Оператор. Оператор обслуживает машину и получает дневные отчеты о ее работе. Мы не будем нуждаться в операторе, если не будет потребителей.

Таким образом, внешняя среда машины утилизации имеет вид, представленный на рис. 12.36.



**Рис. 12.36.** Внешняя среда машины утилизации

Деление актеров на первичных и вторичных облегчает выбор системной архитектуры в терминах основного функционального назначения. Системную структуру определяют в основном первичные актеры. Именно от них в систему приходят главные изменения. Поэтому полное выделение первичных актеров гарантирует, что архитектура системы будет настроена на большинство важных пользователей.

**Определение элементов Use Case**

После выбора внешней среды можно выявить внутренние функциональные возможности системы. Для этого определяются элементы Use Case.

Каждый элемент Use Case задает некоторый путь использования системы, выполнение некоторой части функциональных возможностей. Полная совокупность элементов Use Case определяет все существующие пути использования системы.

Элемент Use Case — это последовательность взаимодействий в диалоге, выполняемом актером и системой. Запускается элемент Use Case актером, поэтому удобно выявлять элементы Use Case с помощью актеров.

Рассматривая каждого актера, мы решаем, какие элементы Use Case он может выполнять. Для этого изучается описание системы (с точки зрения актера) или проводится обсуждение с теми, кто будет действовать как актер.

Перейдем к примеру. Потребитель — первичный актер, поэтому начнем с этой роли. Этот актер должен выполнять возврат утилизируемых элементов. Так формируется элемент Use Case Возврат элемента. Приведем его текстовое описание:

Начинается, когда потребитель начинает возвращать банки, бутылки, ящики. Для каждого элемента, помещенного в машину утилизации, система увеличивает количество элементов, принятых от Потребителя, и общее количество элементов этого типа за день.

После сдачи всех элементов Потребитель нажимает кнопку квитанции, чтобы получить квитанцию, на которой напечатаны названия возвращенных элементов и общая сумма возврата.

Следующий актер — Оператор. Он получает дневной отчет об элементах, сданных за день. Это образует элемент Use Case Создание дневного отчета. Его описание:

Начинается оператором, когда он хочет получить информацию об элементах, сданных за день.

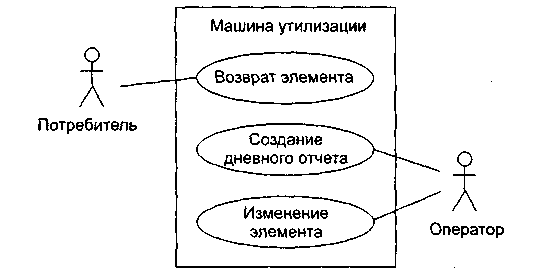
Система печатает количество элементов каждого типа и общее количество элементов, полученных за день.

Доя подготовки к созданию нового дневного отчета сбрасывается в ноль параметр Общее количество.

Кроме того, актер Оператор может изменять параметры сдаваемых элементов. Назовем соответствующий элемент Use Case Изменение элемента. Его описание:

Могут изменяться цена и размер каждого возвращаемого элемента. Могут добавляться новые типы элементов.

После выявления всех элементов диаграмма Use Case для системы принимает вид, показанный на рис. 12.37.



**Рис. 12.37.** Диаграмма Use Case для машины утилизации

Чаще всего полные описания элементов Use Case формируются за несколько итераций. На каждом шаге в описание вводятся дополнительные детали. Например, окончательное описание Возврата элемента может иметь следующий вид:

Когда потребитель возвращает сдаваемый элемент, элемент измеряется системой. Измерения позволяют определить тип элемента. Если тип допустим, то увеличивается количество элементов этого типа, принятых от Потребителя, и общее количество элементов этого типа за день.

Если тип недопустим, то на панели машины высвечивается «недействительно».

Когда Потребитель нажимает кнопку квитанции, принтер печатает дату. Производятся вычисления. По каждому типу принятых элементов печатается информация: название, принятое количество, цена, итого для типа. В конце печатается сумма, которую должен получить потребитель.

Не всегда очевидно, как распределить функциональные возможности по отдельным элементам Use Case и что является вариантом одного и того же элемента Use Case. Основной критерий выбора — сложность элемента Use Case. При анализе вариантов поведения рассматривают их различия. Если различия малы, варианты встраивают в один элемент Use Case. Если различия велики, то варианты описываются как отдельные элементы Use Case.

Обычно элемент Use Case задает одну основную и несколько альтернативных последовательностей событий.

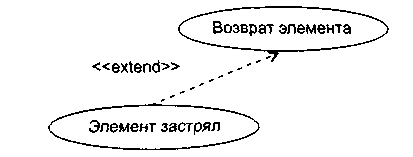
Каждый элемент Use Case выделяет частный аспект функциональных возможностей системы. Поэтому элементы Use Case обеспечивают инкрементную схему анализа функций системы. Можно независимо разрабатывать элементы Use Case для разных функциональных областей, а позднее соединить их вместе (для формирования полной модели требований).

**Вывод**: на основе элементов Use Case в каждый момент времени можно концентрировать внимание на одной частной проблеме, что позволяет вести параллельную разработку.

Расширение функциональных возможностей

Для добавления в элемент Use Case новых действий удобно применять отношение расширения. С его помощью базовый элемент Use Case может быть расширен новым элементом Use Case.

В нашем примере поведение системы не определено для случая, когда сдаваемый элемент застрял. Введем элемент Use Case Элемент Застрял, который будет расширять базовый элемент Use Case Возврат Элемента (рис. 12.38).



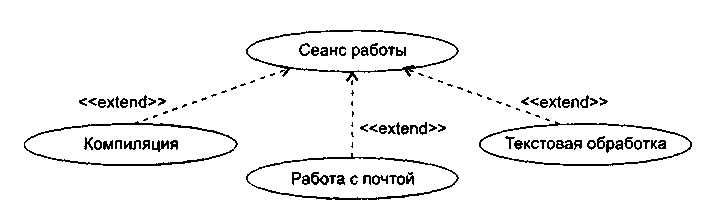
**Рис 12.38.** Расширение элемента Use Case возврат элемента

Описание элемента Use Case Элемент застрял может иметь следующий вид:

Если элемент застрял, для вызова Оператора вырабатывается сигнал тревоги. После удаления застрявшего элемента Оператор сбрасывает сигнал тревоги. В результате Потребитель может продолжить сдачу элементов. Величина ИТОГО сохраняет правильное значение. Цена застрявшего элемента не засчитывается.

Таким образом, описание базового элемента остается прежним, простым. Еще один пример приведен на рис. 12.39.

Здесь мы видим только один базовый элемент Use Case Сеанс работы. Все остальные элементы Use Case могут добавляться как расширения. Базовый элемент Use Case при этом остается почти без изменений.



**Рис. 12.39.** Применение отношения расширения

Отношение расширения определяет прерывание базового элемента Use Case, которое происходит для вставки другого элемента Use Case. Базовый элемент Use Case не знает, будет выполняться прерывание или нет. Вычисление условий прерывания находится вне компетенции базового элемента Use Case.

В расширяющем элементе Use Case указывается ссылка на то место базового элемента Use Case, куда он будет вставляться (при прерывании). После выполнения расширяющего элемента Use Case продолжается выполнение базового элемента Use Case.

Обычно расширения используют:

* для моделирования вариантных частей элементов Use Case;
* для моделирования сложных и редко выполняемых альтернативных последовательностей;
* для моделирования подчиненных последовательностей, которые выполняются только в определенных случаях;
* для моделирования систем с выбором на основе меню.

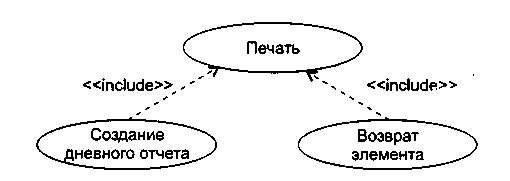
Главное, что следует помнить: решение о выборе, подключении варианта на основе расширения принимается вне базового элемента Use Case. Если же вы вводите в базовый элемент Use Case условную конструкцию, конструкцию выбора, то придется применять отношение включения. Это случай, когда «штурвал управления» находится в руках базового элемента Use Case.

**Уточнение модели требований**

Уточнение модели сводится к выявлению одинаковых частей в элементах Use Case и извлечению этих частей. Любые изменения в такой части, выделенной в отдельный элемент Use Case, будут автоматически влиять на все элементы Use Case, которые используют ее совместно.

Извлеченные элементы Use Case называют абстрактными. Они не могут быть конкретизированы сами по себе, применяются для описания одинаковых частей в других, конкретных элементах Use Case. Таким образом, описания абстрактных элементов Use Case используются в описаниях конкретных элементов Use Case. Говорят, что конкретный элемент Use Case находится в отношении «включает» с абстрактным элементом Use Case.

Вернемся к нашему примеру. В этом примере два конкретных элемента Use Case Возврат элемента и Создание дневного отчета имеют общую часть — действия, обеспечивающие печать квитанции. Поэтому, как показано на рис. 12.40, можно выделить абстрактный элемент Use Case Печать. Этот элемент Use Case будет специализироваться на выполнении распечаток.

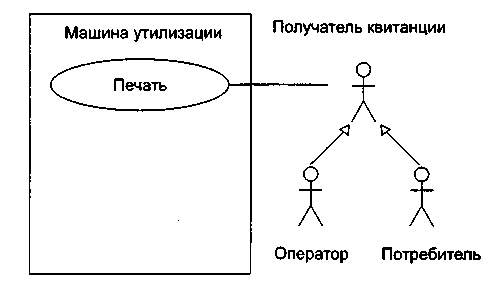


**Рис. 12.40.** Применение отношения включения

В свою очередь, абстрактные элементы Use Case могут использоваться другими абстрактными элементами Use Case. Так образуется иерархия. При построении иерархии абстрактных элементов Use Case руководствуются правилом: выделение элементов Use Case прекращается при достижении уровня отдельных операций над объектами.

Выделение абстрактных элементов Use Case можно упростить с помощью абстрактных актеров.

Абстрактный актер — это общий фрагмент роли в нескольких конкретных актерах. Абстрактный актер выражает подобия в элементах Use Case. Конкретные актеры находятся в отношении наследования с абстрактным актером. Так, в машине утилизации конкретные актеры имеют одно общее поведение: они могут получать квитанцию. Поэтому можно определить одного абстрактного актера — Получателя квитанции. Как показано на рис. 12.41, наследниками этого актера являются Потребитель и Оператор.



**Рис. 12.41.** Выделение абстрактного актера

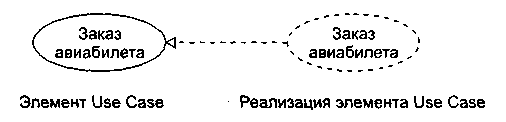
**Выводы:**

1. Абстрактные элементы Use Case находят извлечением общих последовательностей из различных элементов Use Case.
2. Отношение «включает» применяется, если несколько элементов Use Case имеют общее поведение. Цель: устранить повторения, ликвидировать избыточность.
3. Кроме того, это отношение часто используют для ограничения сложности большого элемента Use Case.
4. Отношение «расширяет» применяется, когда описывается вариация, дополняющая нормальное поведение.

## Кооперации и паттерны

Кооперации (сотрудничества) являются средством представления комплексных решений в разработке ПО на высшем, архитектурном уровне. С одной стороны, *^* кооперации обеспечивают компактность цельной спецификации программного продукта, с другой стороны — несут в себе реализации потоков управления и данных, а также структур данных.

В терминологии фирмы Rational (вдохновителя и организатора побед языка UML) кооперации называют реализациями элементов Use Case, да и обозначения их весьма схожи (рис. 12.42).



**Рис. 12.42.** Элемент Use Case и его реализация

Обратите внимание на то, что и связаны эти элементы отношением реализации: кооперация реализует конкретный элемент Use Case.

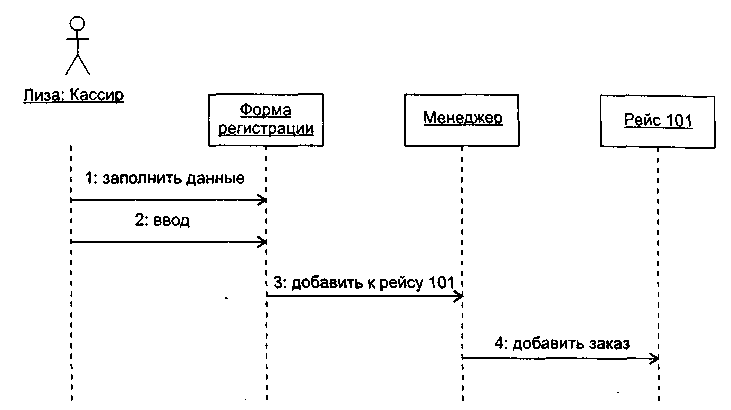
Кооперации содержат две составляющие — статическую (структурную) и динамическую (поведенческую).

Статическая составляющая кооперации задает структуру совместно работающих классов и других элементов (интерфейсов, компонентов, узлов). Чаще всего для этого используют одну или несколько диаграмм классов. Динамическая составляющая кооперации определяет поведение совместно работающих элементов. Обычно для определения применяют одну или несколько диаграмм последовательности.

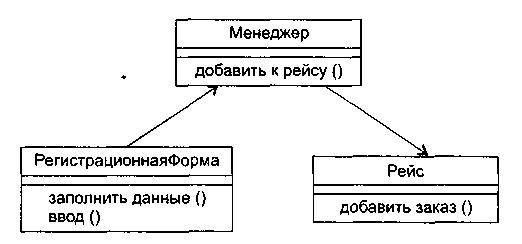
Таким образом, если заглянуть под «обложку» кооперации, мы увидим набор разнообразных диаграмм. Например, требования к информационной системе авиакассы задаются множеством элементов Use Case, каждый из которых реализуется отдельной кооперацией. Все эти кооперации применяют одни и те же классы, но все же имеют разную функциональную организацию. В частности, поведение кооперации для заказа авиабилета может описываться диаграммой последовательности, показанной на рис. 12.43.

Соответственно, структура кооперации для заказа авиабилета может иметь вид, представленный на рис. 12.44.

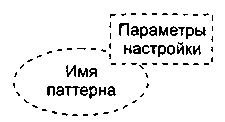
Важно понимать, что кооперации отражают понятийный аспект архитектуры системы. Один и тот же элемент может участвовать в различных кооперациях. Ведь речь здесь идет не о владении элементом, а только о его применении.



**Рис. 12.43.** Динамическая составляющая кооперации Заказ авиабилета



**Рис. 12.44.** Статическая составляющая кооперации Заказ авиабилета



**Рис. 12.45.** Обозначение паттерна

Параметризованные, то есть настраиваемые кооперации называют паттернами (образцами). Паттерн является решением типичной проблемы в определенном контексте. Обозначение паттерна имеет вид, представленный на рис. 12.45.

На место параметров настройки паттерна подставляются различные фактические параметры, в результате создаются разные кооперации.

Паттерны рассматриваются как крупные строительные блоки. Их использование приводит к существенному сокращению затрат на анализ и проектирование ПО. повышению качества и правильности разработки на логическом уровне, ведь паттерны создаются опытными профессионалами и отражают проверенные и оптимизированные решения [26], [31], [68].

Итак, паттерны — это наборы готовых решений, рецепты, предлагающие к повторному использованию самое ценное для разработчика — сплав мирового опыта по созданию ПО.

## Бизнес-модели

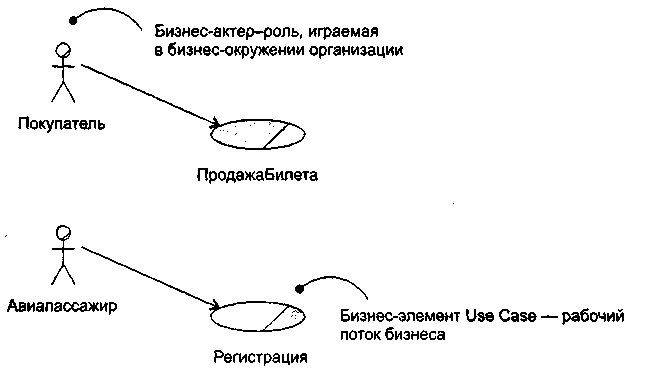
Достаточно часто перед тем, как решиться на заказ ПО, организация проводит бизнес-моделирование. Цели бизнес-моделирования:

* отобразить структуру и процессы деятельности организации;
* обеспечить ясное, комплексное и, главное, одинаковое понимание нужд организации как сотрудниками, так и будущими разработчиками ПО;
* сформировать реальные требования к программному обеспечению деятельности организации.

Для достижения этих целей разрабатываются две модели: Q бизнес-модель Use Case; а бизнес-объектная модель.

Бизнес-модель Use Case задает внешнее представление бизнес-процессов организации (с точки зрения внешней среды — клиентов и партнеров).

Как показано на рис. 12.57, бизнес-модель Use Case строится с помощью бизнес-актеров и бизнес-элементов Use Case — простого расширения средств, используемых в обычных диаграммах Use Case.



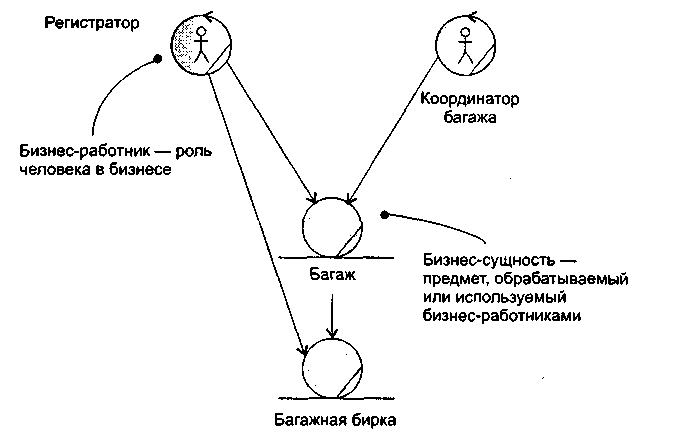
**Рис. 12.57.** Фрагмент бизнес-модели Use Case для аэропорта

Бизнес-актеры определяют внешние сущности и людей, с которыми взаимодействует бизнес. Бизнес-актер представляет собой человека, но информационная система, взаимодействующая с бизнесом, также может играть роль такого актера.

Бизнес-элементы Use Case изображают различные рабочие потоки бизнеса. Последовательности действий в бизнес-элементах Use Case обычно описываются диаграммами деятельности.

Бизнес-объектная модель отражает внутреннее представление бизнес-процессов организации (с точки зрения ее сотрудников).

Как показано на рис. 12.58, бизнес-объектная модель строится с помощью бизнес-работников и бизнес-сущностей — классов со специальными стереотипами. Эти классы имеют специальные графические обозначения.



**Рис. 12.58.** Фрагмент бизнес-объектной модели аэропорта

Бизнес-работник — абстракция человека, действующего в бизнесе. Бизнес-сущности являются «предметами», обрабатываемыми или используемыми бизнес-работниками по мере выполнения бизнес-элемента Use Case. Например, бизнес-сущность представляет собой документ или существенную часть продукта. Фактически бизнес-объектная модель отображается с помощью диаграмм классов.

## Контрольные вопросы

1. Поясните два подхода к моделированию поведения системы. Объясните достоинства и недостатки каждого из этих подходов.
2. Охарактеризуйте вершины и дуги диаграммы схем состояний. В чем состоит назначение этой диаграммы?
3. Как отображаются действия в состояниях диаграммы схем состояний?
4. Как показываются условные переходы между состояниями?
5. Как задаются вложенные состояния в диаграммах схем состояний?
6. Поясните понятие исторического подсостояния.
7. Охарактеризуйте средства и возможности диаграммы деятельности.
8. Когда не следует применять диаграмму деятельности?
9. Какие средства диаграммы деятельности позволяют отобразить параллельные действия?
10. Зачем в диаграмму деятельности введены плавательные дорожки?
11. Как представляется имя объекта в диаграмме сотрудничества?
12. Поясните синтаксис представления свойства в диаграмме сотрудничества.
13. Какие стереотипы видимости используются в диаграмме сотрудничества? Поясните их смысл.
14. В какой форме записываются сообщения в языке UML? Поясните смысл сообщения.
15. В каком отношении находятся сообщения и действия? Перечислите разновидности действий.
16. Чем отличается процедурный поток от асинхронного потока сообщений?
17. Как указывается повторение сообщений?
18. Как показать ветвление сообщений?
19. Что общего в диаграмме последовательности и диаграмме сотрудничества? Чем они отличаются друг от друга?
20. Как отображается порядок передачи сообщений в диаграмме последовательности?
21. Когда удобнее применять диаграммы последовательности?
22. Из каких элементов состоит диаграмма Use Case?
23. Какие отношения разрешены между элементами диаграммы Use Case?
24. Для чего применяют диаграммы Use Case?
25. Чем отличаются друг от друга отношения включения и расширения с точки зрения управления?
26. Каково назначение спецификации элемента Use Case и как она оформляется?
27. Что такое сценарий элемента Use Case?
28. Как документируется отношение включения?
29. Как документируется отношение расширения?
30. Каков порядок построения модели требований?
31. Каково назначение кооперации? Какие составляющие ее образуют?
32. Могут ли разные кооперации использовать одинаковые классы? Обоснуйте ответ.
33. Что такое паттерн?
34. Чем паттерн отличается от кооперации? Чем они схожи?
35. Как описывается паттерн?
36. Что нужно сделать для применения паттерна?
37. Каковы цели бизнес-моделирования?
38. Из каких частей состоит бизнес-модель? На что похожи эти части? В чем их своеобразие?

# 10. Модели реализации объектно-ориентированных программных систем

Статические и динамические модели описывают логическую организацию системы, отражают логический мир программного приложения. Модели реализации обеспечивают представление системы в физическом мире, рассматривая вопросы упаковки логических элементов в компоненты и размещения компонентов в аппаратных узлах [8], [23], [53], [67].

## Компонентные диаграммы

Компонентная диаграмма — первая из двух разновидностей диаграмм реализации, моделирующих физические аспекты объектно-ориентированных систем. Компонентная диаграмма показывает организацию набора компонентов и зависимости между компонентами.

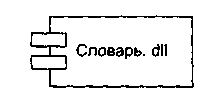
Элементами компонентных диаграмм являются компоненты и интерфейсы, а также отношения зависимости и реализации. Как и другие диаграммы, компонентные диаграммы могут включать примечания и ограничения. Кроме того, компонентные диаграммы могут содержать пакеты или подсистемы, используемые для группировки элементов модели в крупные фрагменты.

### Компоненты

По своей сути компонент является физическим фрагментом реализации системы, который заключает в себе программный код (исходный, двоичный, исполняемый), сценарные описания или наборы команд операционной системз (имеются в виду командные файлы). Язык UML дает следующее определение.

*Компонент —* физическая и заменяемая часть системы, которая соответствует набору интерфейсов и обеспечивает реализацию этого набора интерфейсов.

Интерфейс — очень важная часть понятия «компонент», его мы обсудим в следующем подразделе. Графически компонент изображается как прямоугольник с вкладками, обычно включающий имя (рис. 13.1).



**Рис. 13.1.** Обозначение компонента

Компонент — базисный строительный блок физического представления ПО, поэтому интересно сравнить его с базисным строительным блоком логического представления ПО — классом.

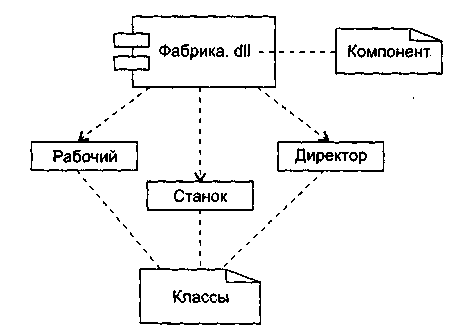
Сходные характеристики компонента и класса:

* наличие имени;
* реализация набора интерфейсов;
* участие в отношениях зависимости;
* возможность быть вложенным;
* наличие экземпляров (экземпляры компонентов можно использовать только в диаграммах размещения).

Вы скажете — много общего. И тем не менее между компонентами и классами есть существенная разница, ее характеризует табл. 13.1.

**Таблица 13.1.** Различия компонентов и классов

|  |  |
| --- | --- |
| **№** | **Описание** |
| 1  2  3 | Классы — логические абстракции, компоненты — физические предметы, которые живут в мире битов. В частности, компоненты могут «жить» в физических узлах, а классы лишены такой возможности  Компоненты являются физическими упаковками, контейнерами, инкапсулирующими в себе различные логические элементы. Они — элементы абстракций другого уровня  Классы имеют свойства и операции. Компоненты имеют только операции, которые доступны через их интерфейсы |



**Рис.** 13.2. Классы в компоненте

О чем говорят эти различия? Во-первых, класс не может «дышать» воздухом физического мира реализации. Ему нужен скафандр. Таким скафандром является компонент.

Во-вторых, им не жить друг без друга — пустые скафандры никому не нужны. Причем в скафандре-компоненте может находиться несколько классов и коопераций. Итак, в скафандре — физической реализации — располагается набор логики. Как показано на рис. 13.2, с помощью отношения зависимости можно явно отобразить отношение между компонентом и классами, которые он реализует. Правда, чаще всего такие отношения не отображаются. Их удобно представлять в компонентной спецификации.

В-третьих, класс — душа нараспашку (он может даже показать свои свойства). Компонент всегда застегнут на все пуговицы (правда, из него торчат интерфейсные разъемы операций).

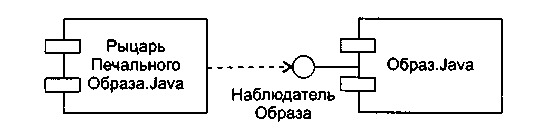
Теперь уместно перейти к обсуждению интерфейсов.

### Интерфейсы

Интерфейс — список операций, которые определяют услуги класса или компонента. Образно говоря, интерфейс — это разъем, который торчит из ящичка компонента. С помощью интерфейсных разъемов компоненты стыкуются друг с другом, объединяясь в систему.

Еще одна аналогия. Интерфейс подобен абстрактному классу, у которого отсутствуют свойства и работающие операции, а есть только абстрактные операции (не имеющие тел). Если хотите, интерфейс похож на улыбку чеширского кота из правдивой истории об Алисе, где кот отдельно и улыбка отдельно. Все операции интерфейса открыты и видимы клиенту (в противном случае они потеряли бы всякий смысл). Итак, операции интерфейса только именуют предлагаемые услуги, не более того.

Очень важна взаимосвязь между компонентом и интерфейсом. Возможны два способа отображения взаимосвязи между компонентом и его интерфейсами. В первом, свернутом способе, как показано на рис. 13.3, интерфейс изображается в форме пиктограммы. Компонент Образ.java, который реализует интерфейс, соединяется со значком интерфейса (кружком) НаблюдательОбраза простой линией. Компонент РыцарьПечальногоОбраза.jауа, который использует интерфейс, связан с ним отношением зависимости.



**Рис. 13.3.** Представление интерфейса в форме пиктограммы

Второй способ представления интерфейса иллюстрирует рис. 13.4. Здесь используется развернутая форма изображения интерфейса, в которой могут показываться его операции. Компонент, который реализует интерфейс, подключается к нему отношением реализации. Компонент, который получает доступ к услугам другого компонента через интерфейс, по-прежнему подключается к интерфейсу отношением зависимости.



**Рис. 13.4.** Развернутая форма представления интерфейса

По способу связи компонента с интерфейсом различают:

* экспортируемый интерфейс — тот, который компонент реализует и предлагает как услугу клиентам;
* импортируемый интерфейс — тот, который компонент использует как услугу другого компонента.

У одного компонента может быть несколько экспортируемых и несколько импортируемых интерфейсов.

Тот факт, что между двумя компонентами всегда находится интерфейс, устраняет их прямую зависимость. Компонент, использующий интерфейс, будет функционировать правильно вне зависимости от того, какой компонент реализует этот интерфейс. Это очень важно и обеспечивает гибкую замену компонентов в интересах развития системы.

### Компоновка системы

За последние полвека разработчики аппаратуры прошли путь от компьютеров размером с комнату до крошечных «ноутбуков», обеспечивших возросшие функциональные возможности. За те же полвека разработчики программного обеспечения прошли путь от больших систем на Ассемблере и Фортране до еще больших систем на C++ и Java. Увы, но программный инструментарий развивается медленнее, чем аппаратный инструментарий. В чем главный секрет аппаратчиков? — спросят у аппаратчика-мальчиша программеры-буржуины.

Этот секрет — компоненты. Разработчик аппаратуры создает систему из готовых аппаратных компонентов (микросхем), выполняющих определенные функции и предоставляющих набор услуг через ясные интерфейсы. Задача конструкторов упрощается за счет повторного использования результатов, полученных другими.

Повторное использование — магистральный путь развития программного инструментария. Создание нового ПО из существующих, работоспособных программных компонентов приводит к более надежному и дешевому коду. При этом сроки разработки существенно сокращаются.

Основная цель программных компонентов — допускать сборку системы из двоичных заменяемых частей. Они должны обеспечить начальное создание системы из компонентов, а затем и ее развитие — добавление новых компонентов и замену некоторых старых компонентов без перестройки системы в целом. Ключ к воплощению такой возможности — интерфейсы. После того как интерфейс определен, к выполняемой системе можно подключить любой компонент, который удовлетворяет ему или обеспечивает этот интерфейс. Для расширения системы производятся компоненты, которые обеспечивают дополнительные услуги через новые интерфейсы. Такой подход основывается на особенностях компонента, перечисленных в табл. 13.2.

**Таблица 13.2.** Особенности компонента

|  |
| --- |
| Компонент физичен. Он живет в мире битов, а не логических понятий и не зависит от языка программирования |
| Компонент — заменяемый элемент. Свойство заменяемости позволяет заменить один компонент другим компонентом, который удовлетворяет тем же интерфейсам. Механизм замены оговорен современными компонентными моделями (COM, COM+, CORBA, Java Beans), требующими незначительных преобразований или предоставляющими утилиты, которые автоматизируют механизм |
| Компонент является частью системы, он редко автономен. Чаще компонент сотрудничает с другими компонентами и существует в архитектурной или технологической среде, предназначенной для его использования. Компонент связан и физически, и логически, он обозначает фрагмент большой системы |
| Компонент соответствует набору интерфейсов и обеспечивает реализацию этого набора интерфейсов |

**Вывод**: компоненты — базисные строительные блоки, из которых может проектироваться и составляться система. Компонент может появляться на различных уровнях иерархии представления сложной системы. Система на одном уровне абстракции может стать простым компонентом на более высоком уровне абстракции.

### Разновидности компонентов

Мир современных компонентов достаточно широк и разнообразен. В языке UML для обозначения новых разновидностей компонентов используют механизм стереотипов. Стандартные стереотипы, предусмотренные в UML для компонентов, представлены в табл. 13.3.

**Таблица 13.3.** Разновидности компонентов

|  |  |
| --- | --- |
| **Стереотип** | **Описание** |
| «executable»  «library»  «file»  «table»  «document» | Компонент, который может выполняться в физическом узле (имеет расширение .ехе)  Статическая или динамическая объектная библиотека (имеет расширение .dll)  Компонент, который представляет файл, содержащий исходный код или данные (имеет расширение .ini)  Компонент, который представляет таблицу базы данных (имеет расширение .tbl)  Компонент, который представляет документ (имеет расширение .hip) |

В языке UML не определены пиктограммы для перечисленных стереотипов, применяемые на практике пиктограммы компонентов показаны на рис. 13.5-13.9.



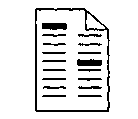
**Рис. 13.5.** Пиктограмма исполняемого **Рис. 13.6.** Пиктограмма объектной

элемента библиотеки



**Рис. 13.7.** Пиктограмма документа **Рис. 13.8.** Пиктограмма таблицы

с исходным кодом или данными базы данных



**Рис. 13.9.** Пиктограмма документа

## Использование компонентных диаграмм

Компонентные диаграммы используют для моделирования статического представления реализации системы. Это представление поддерживает управление конфигурацией системы, составляемой из компонентов. Подразумевается, что для получения работающей системы существуют различные способы сборки компонентов.

Компонентные диаграммы показывают отношения:

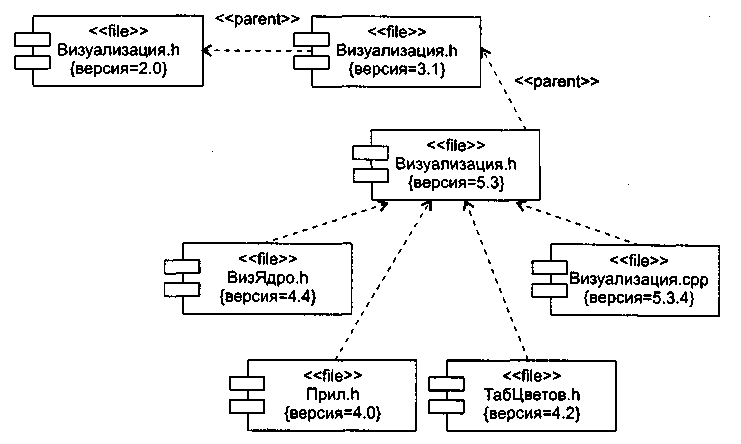
* периода компиляции (среди текстовых компонентов);
* периода сборки, линковки (среди объектных двоичных компонентов);
* периода выполнения (среди машинных компонентов).

Рассмотрим типовые варианты применения компонентных диаграмм.

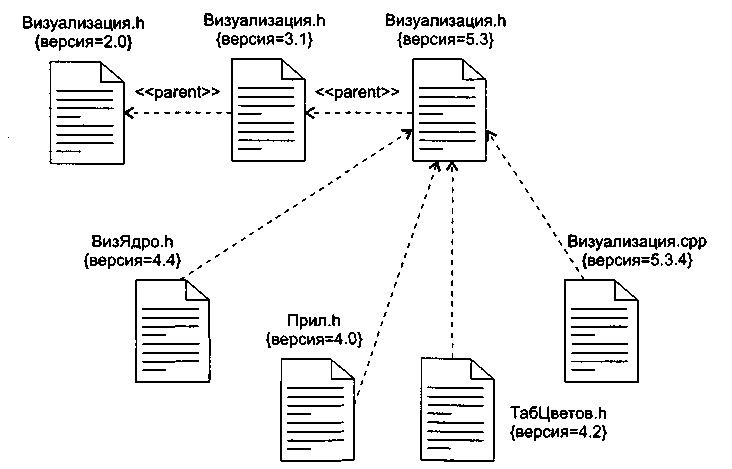
### Моделирование программного текста системы

При разработке сложных систем программный текст (исходный код) разбросан по многим файлам исходного кода. При использовании Java исходный код сохраняется в .java-файлах, при использовании C++ — в заголовочных файлах (.h-фай-лах) и телах (.срр-файлах), при использовании Ada 95 — в спецификациях (.ads-файлах) и реализациях (.adb-файлах).

Между файлами существуют многочисленные зависимости компиляции. Если к этому добавить, что по мере разработки рождаются новые версии файлов, то становится очевидной необходимость управления конфигурацией системы, визуализации компиляционных зависимостей.



**Рис. 13.10.** Моделирование исходного кода



**Рис. 13.11.** Моделирование исходного кода с использованием пиктограмм

В качестве примера на рис. 13.10 приведена компонентная диаграмма, где изображены файлы исходного кода, используемые для построения библиотеки Визуализация.dll. Имеются четыре заголовочных файла (Визуализация.h, ВизЯдро.h, Прил.h, ТабЦветов.h), которые представляют исходный код для спецификации определенных классов. Файл реализации здесь один (Визуализация.срр), он является реализацией одного из заголовков. Отметим, что для каждого файла явно указана его версия, причем для файла Визуализация.h показаны три версии и история их появления. На рис. 13.11 повторяется та же диаграмма, но здесь для обозначения компонентов использованы специальные пиктограммы.

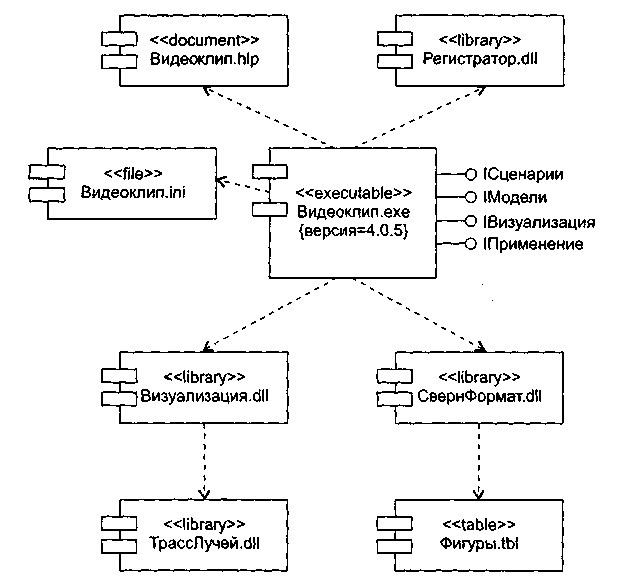
### 

### Моделирование реализации системы

Реализация системы может включать большое количество разнообразных компонентов:

* исполняемых элементов;
* динамических библиотек;
* файлов данных;
* справочных документов;
* файлов инициализации;
* файлов регистрации;
* сценариев;
* файлов установки.

Моделирование этих компонентов, отношений между ними — важная часть управления конфигурацией системы.

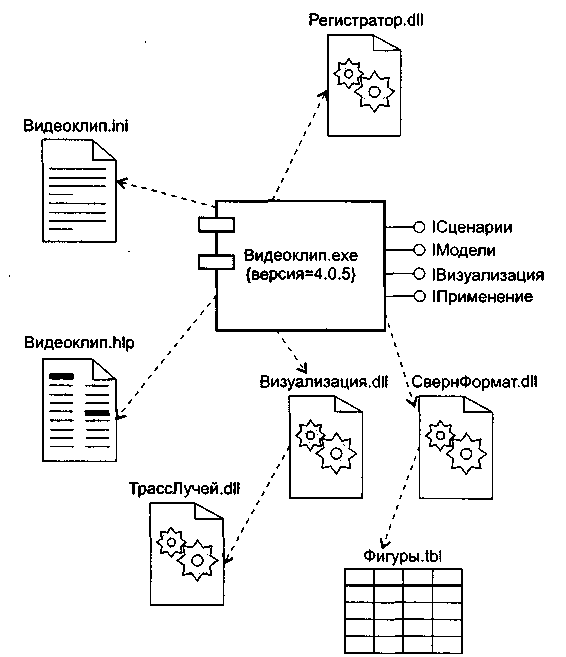


**Рис. 13.12.** Моделирование реализации системы

Например, на рис. 13.12 показана часть реализации системы, группируемая вокруг исполняемого элемента Видеоклип.ехе. Здесь изображены четыре библиотеки (Регистратор.dll, СвернФормат.dll, Визуализация.dll, ТрассЛучей.dll), один документ (Видеоклип.hlp), один простой файл (Видеоклип.ini),атакже таблица базы данных (Фигуры.tbl). Вдиаграмме указаны отношения зависимости, существующие между компонентами.

Для исполняемого компонента Видеоклип.ехе указан номер версии (с помощью пгеговой величины), представлены его экспортируемые интерфейсы (IСценарии, IВизуализация, IМодели, IПрименение). Эти интерфейсы образуют API компонента «интерфейс прикладного программирования).

На рис. 13.13 повторяется та же диаграмма, моделирующая реализацию, но здесь для обозначения компонентов использованы специальные пиктограммы.



**Рис. 13.13.** Моделирование реализации с использованием пиктограмм

# 7. Базис языка визуального моделирования

Для создания моделей анализа и проектирования объектно-ориентированных программных систем используют языки визуального моделирования. Появившись сравнительно недавно, в период с 1989 по 1997 год, эти языки уже имеют представительную историю развития.

E:\UML_Picture\РисункиVisio\ИсторияUML.wmf

В настоящее время различают три поколения языков визуального моделирования. И если первое поколение образовали 10 языков, то численность второго поколения уже превысила 50 языков. Среди наиболее популярных языков 2-го поколения можно выделить: язык Буча (G. Booch), язык Рамбо (J. Rumbaugh), язык Джекобсона (I. Jacobson), язык Коада-Йордона (Coad-Yourdon), язык Шлеера-Меллора (Shlaer-Mellor) и т. д [41], [64], [69]. Каждый язык вводил свои выразительные средства, ориентировался на собственный синтаксис и семантику, иными словами — претендовал на роль единственного и неповторимого языка. В результате разработчики (и пользователи этих языков) перестали понимать друг друга. Возникла острая необходимость унификации языков.

Идея унификации привела к появлению языков 3-го поколения. В качестве стандартного языка третьего поколения был принят Unified Modeling Language (UML), создававшийся в 1994-1997 годах (основные разработчики — три «amigos» Г. Буч, Дж. Рамбо, И. Джекобсон)- версия 1.4-1.5, которая описывается в данном конспекте лекций [53]. Данная глава посвящена определению базовых понятий языка UML. В настоящее время разработана версия UML 2.12 (см. Рис.

## Унифицированный язык моделирования

UML — стандартный язык для написания моделей анализа, проектирования и реализации объектно-ориентированных программных систем [23], [53], [67]. UML может использоваться для визуализации, спецификации, конструирования и документирования результатов программных проектов. UML — это не визуальный язык программирования, но его модели прямо транслируются в текст на языках программирования (Java, C++, Visual Basic, Ada 95, Object Pascal) и даже в таблицы для реляционной БД.

Для визуализации, специфицирования, конструирования и документирования программных систем необходимо рассматривать их с различных точек зрения ([см. главу 1](http://interway.narod.ru/context/prog/UML/1/gl1.htm)). Все, кто имеет отношение к проекту, - конечные пользователи, аналитики, разработчики, системные интеграторы, тестировщики, технические писатели и менеджеры проектов - преследуют собственные интересы, и каждый смотрит на создаваемую систему, по-разному в различные моменты ее жизни. Системная архитектура является, пожалуй, наиболее важным артефактом, который используется для управления всевозможными точками зрения и тем самым способствует итеративной и инкрементной разработке системы на всем протяжении ее жизненного цикла.

Архитектура - это совокупность существенных решений касательно:

организации программной системы;

выбора структурных элементов, составляющих систему, и их интерфейсов;

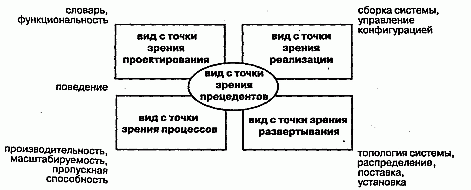
поведения этих элементов, специфицированного в кооперациях с другими элементами;

составления из этих структурных и поведенческих элементов все более и более крупных подсистем;

архитектурного стиля, направляющего и определяющего всю организацию системы: статические и динамические элементы, их интерфейсы, кооперации и способ их объединения.

Архитектура программной системы охватывает не только ее структурные и поведенческие аспекты, но и использование, функциональность, производительность, гибкость, возможности повторного применения, полноту, экономические и технологические ограничения и компромиссы, а также эстетические вопросы.

Как показано на рис. 2.20, архитектура программной системы наиболее оптимально может быть описана с помощью пяти взаимосвязанных видов или представлений, каждый из которых является одной из возможных проекций организации и структуры системы и заостряет внимание на определенном аспекте ее функционирования ([см. главу 31](http://interway.narod.ru/context/prog/UML/1/gl31.htm)).



**Рис. 2.20** Моделирование системной архитектуры

*Вид с точки зрения прецедентов* (Use case view) охватывает прецеденты, которые описывают поведение системы, наблюдаемое конечными пользователями, аналитиками и тестировщиками. Этот вид специфицирует не истинную организацию программной системы, а те движущие силы, от которых зависит формирование системной архитектуры. В языке **UML** статические аспекты этого вида передаются диаграммами прецедентов, а динамические - диаграммами взаимодействия, состояний и действий.

*Вид с точки зрения проектирования* (Design view) охватывает классы, интерфейсы и кооперации, формирующие словарь задачи и ее решения. Этот вид поддерживает прежде всего функциональные требования, предъявляемые к системе,то есть те услуги, которые она должна предоставлять конечным пользователям. С помощью языка **UML** статические аспекты этого вида можно передавать диаграммами классов и объектов, а динамические - диаграммами взаимодействия, состояний и действий.

*Вид с точки зрения процессов* (Process view) охватывает нити и процессы, формирующие механизмы параллелизма и синхронизации в системе. Этот вид описывает главным образом производительность, масштабируемость и пропускную способность системы. В **UML** его статические и динамические аспекты визуализируются теми же диаграммами, что и для вида с точки зрения проектирования, но особое внимание при этом уделяется активным классам, которые представляют соответствующие нити и процессы.

*Вид с точки зрения реализации* (Implementation view) охватывает компоненты и файлы, используемые для сборки и выпуска конечного программного продукта. Этот вид предназначен в первую очередь для управления конфигурацией версий системы, составляемых из независимых (до некоторой степени) компонентов и файлов, которые могут по-разному объединяться между собой. В языке **UML** статические аспекты этого вида передают с помощью диаграмм компонентов, а динамические - с помощью диаграмм взаимодействия, состояний и действий.

*Вид с точки зрения развертывания* (Deployment view) охватывает узлы, формирующие топологию аппаратных средств системы, на которой она выполняется. В первую очередь он связан с распределением, поставкой и установкой частей, составляющих физическую систему. Его статические аспекты описываются диаграммами развертывания, а динамические - диаграммами взаимодействия, состояний и действий.

Каждый из перечисленных видов может считаться вполне самостоятельным, так что лица, имеющие отношение к разработке системы, могут сосредоточиться на изучении только тех аспектов архитектуры, которые непосредственно их касаются. Но нельзя забывать о том, что эти виды взаимодействуют друг с другом. Например, узлы вида с точки зрения развертывания содержат компоненты, описанные для вида с точки зрения реализации, а те, в свою очередь, представляют собой физическое воплощение классов, интерфейсов, коопераций и активных классов из видов с точки зрения проектирования и процессов. **UML** позволяет отобразить каждый из пяти перечисленных видов и их взаимодействия

Словарь UML образуют три разновидности строительных блоков: предметы, отношения, диаграммы.

Предметы — это абстракции, которые являются основными элементами в модели, отношения связывают эти предметы, диаграммы группируют коллекции предметов.

## Предметы в UML

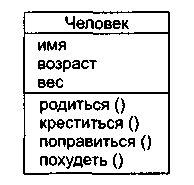
В UML имеются четыре разновидности предметов:

* структурные предметы;
* предметы поведения;
* группирующие предметы;
* поясняющие предметы.

Эти предметы являются базовыми объектно-ориентированными строительными блоками. Они используются для написания моделей.

*Структурные предметы* являются существительными в UML-моделях. Они представляют статические части модели — понятийные или физические элементы. Перечислим восемь разновидностей структурных предметов.

1. *Класс* — описание множества объектов, которые разделяют одинаковые свойства, операции, отношения и семантику (смысл). Класс реализует один или несколько интерфейсов. Как показано на рис. 10.1, графически класс отображается в виде прямоугольника, обычно включающего секции с именем, свойствами (атрибутами) и операциями.



**Рис. 10.1.** Классы

1. *Интерфейс —* набор операций, которые определяют услуги класса или компонента. Интерфейс описывает поведение элемента, видимое извне. Интерфейс может представлять полные услуги класса или компонента или часть таких услуг. Интерфейс определяет набор спецификаций операций (их сигнатуры), а не набор реализаций операций. Графически интерфейс изображается в виде кружка с именем, как показано на рис. 10.2. Имя интерфейса обычно начинается с буквы «I». Интерфейс редко показывают самостоятельно. Обычно его присоединяют к классу или компоненту, который реализует интерфейс.
2. *Кооперация* (сотрудничество) определяет взаимодействие и является совокупностью ролей и других элементов, которые работают вместе для обеспечения коллективного поведения более сложного, чем простая сумма всех элементов. Таким образом, кооперации имеют как структурное, так и поведенческое измерения. Конкретный класс может участвовать в нескольких кооперациях. Эти кооперации представляют реализацию паттернов (образцов), которые формируют систему. Как показано на рис. 10.3, графически кооперация изображается как пунктирный эллипс, в который вписывается ее имя.



**Рис. 10.3.** Кооперации

1. *Актер —* набор согласованных ролей, которые могут играть пользователи при взаимодействии с системой (ее элементами Use Case). Каждая роль требует от системы определенного поведения. Как показано на рис. 10.4, актер изображается как проволочный человечек с именем.



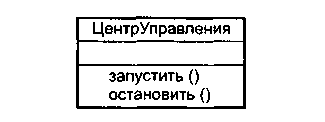
**Рис. 10.4.** Актеры

1. *Элемент Use Case* (Прецедент) — описание последовательности действий (или нескольких последовательностей), выполняемых системой в интересах отдельного актера и производящих видимый для актера результат. В модели элемент Use Case применяется для структурирования предметов поведения. Элемент Use Case реализуется кооперацией. Как показано на рис. 10.5, элемент Use Case изображается как эллипс, в который вписывается его имя.



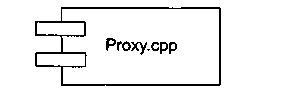
**Рис. 10.5.** Элементы Use Case

1. *Активный класс —* класс, чьи объекты имеют один или несколько процессов (или потоков) и поэтому могут инициировать управляющую деятельность. Активный класс похож на обычный класс за исключением того, что его объекты действуют одновременно с объектами других классов. Как показано на рис. 10.6, активный класс изображается как утолщенный прямоугольник, обычно включающий имя, свойства (атрибуты) и операции.



**Рис. 10.6.** Активные классы

1. *Компонент —* физическая и заменяемая часть системы, которая соответствует набору интерфейсов и обеспечивает реализацию этого набора интерфейсов. В систему включаются как компоненты, являющиеся результатами процесса разработки (файлы исходного кода), так и различные разновидности используемых компонентов (СОМ+-компоненты, Java Beans). Обычно компонент — это физическая упаковка различных логических элементов (классов, интерфейсов и сотрудничеств). Как показано на рис. 10.7, компонент изображается как прямоугольник с вкладками, обычно включающий имя.



**Рис. 10.7.** Компоненты

1. *Узел —* физический элемент, который существует в период работы системы и представляет ресурс, обычно имеющий память и возможности обработки. В узле размещается набор компонентов, который может перемещаться от узла к узлу. Как показано на рис. 10.8, узел изображается как куб с именем.



**Рис. 10.8.** Узлы

*Предметы поведения —* динамические части UML-моделей. Они являются глаголами моделей, представлением поведения во времени и пространстве. Существует две основные разновидности предметов поведения.

1. *Взаимодействие* — поведение, заключающее в себе набор сообщений, которыми обменивается набор объектов в конкретном контексте для достижения определенной цели. Взаимодействие может определять динамику как совокупности объектов, так и отдельной операции. Элементами взаимодействия являются сообщения, последовательность действий (поведение, вызываемое сообщением) и связи (соединения между объектами). Как показано на рис. 10.9, сообщение изображается в виде направленной линии с именем ее операции.



**Рис. 10.9.** Сообщения

1. *Конечный автомат —* поведение, которое определяет последовательность состояний объекта или взаимодействия, выполняемые в ходе его существования в ответ на события (и с учетом обязанностей по этим событиям). С помощью конечного автомата может определяться поведение индивидуального класса или кооперации классов. Элементами конечного автомата являются состояния, переходы (от состояния к состоянию), события (предметы, вызывающие переходы) и действия (реакции на переход). Как показано на рис. 10.10, состояние изображается как закругленный прямоугольник, обычно включающий его имя и его подсостояния (если они есть).



**Рис. 10.10.** Состояния

Эти два элемента — взаимодействия и конечные автоматы — являются базисными предметами поведения, которые могут включаться в UML-модели. Семантически эти элементы ассоциируются с различными структурными элементами (прежде всего с классами, сотрудничествами и объектами).

*Группирующие предметы* — организационные части UML-моделей. Это ящики, по которым может быть разложена модель. Предусмотрена одна разновидность группирующего предмета — пакет.

*Пакет —* общий механизм для распределения элементов по группам. В пакет могут помещаться структурные предметы, предметы поведения и даже другие группировки предметов. В отличие от компонента (который существует в период выполнения), пакет — чисто концептуальное понятие. Это означает, что пакет существует только в период разработки. Как показано на рис. 10.11, пакет изображается как папка с закладкой, на которой обозначено его имя и, иногда, его содержание.



**Рис. 10.11.** Пакеты

*Поясняющие предметы —* разъясняющие части UML-моделей. Они являются замечаниями, которые можно применить для описания, объяснения и комментирования любого элемента модели. Предусмотрена одна разновидность поясняющего предмета — примечание.

*Примечание —* символ для отображения ограничений и замечаний, присоединяемых к элементу или совокупности элементов. Как показано на рис. 10.12, примечание изображается в виде прямоугольника с загнутым углом, в который вписывается текстовый или графический комментарий.



**Рис. 10.12.** Примечания

## Отношения в UML

В UML имеются четыре разновидности отношений:

1) зависимость;

2) ассоциация;

3) обобщение;

4) реализация.

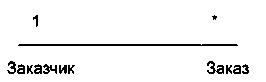
Эти отношения являются базовыми строительными блоками отношений. Они используются при написании моделей.

1. *Зависимость* — семантическое отношение между двумя предметами, в котором изменение в одном предмете (независимом предмете) может влиять на семантику другого предмета (зависимого предмета). Как показано на рис. 10.13, зависимость изображается в виде пунктирной линии, возможно направленной на независимый предмет и иногда имеющей метку.



**Рис. 10.13.** Зависимости

1. *Ассоциация —* структурное отношение, которое описывает набор связей, являющихся соединением между объектами. Агрегация — это специальная разновидность ассоциации, представляющая структурное отношение между целым и его частями. Как показано на рис. 10.14, ассоциация изображается в виде сплошной линии, возможно направленной, иногда имеющей метку и часто включающей другие «украшения», такие как мощность и имена ролей.



**Рис. 10.14.** Ассоциации

1. *Обобщение* — отношение специализации/обобщения, в котором объекты специализированного элемента (потомка, ребенка) могут заменять объекты обобщенного элемента (предка, родителя). Иначе говоря, потомок разделяет структуру и поведение родителя. Как показано на рис. 10.15, обобщение изображается в виде сплошной стрелки с полым наконечником, указывающим на родителя.



**Рис. 10.15.** Обобщения

1. *Реализация —* семантическое отношение между классификаторами, где один классификатор определяет контракт, который другой классификатор обязуется выполнять (к классификаторам относят классы, интерфейсы, компоненты, элементы Use Case, кооперации). Отношения реализации применяют в двух случаях: между интерфейсами и классами (или компонентами), реализующими их; между элементами Use Case и кооперациями, которые реализуют их. Как показано на рис. 10.16, реализация изображается как нечто среднее между обобщением и зависимостью.



**Рис. 10.16.** Реализации

## Диаграммы в UML

*Диаграмма* — графическое представление множества элементов, наиболее часто изображается как связный граф из вершин (предметов) и дуг (отношений). Диаграммы рисуются для визуализации системы с разных точек зрения, затем они отображаются в систему. Обычно диаграмма дает неполное представление элементов, которые составляют систему. Хотя один и тот же элемент может появляться во всех диаграммах, на практике он появляется только в некоторых диаграммах. Теоретически диаграмма может содержать любую комбинацию предметов и отношений, на практике ограничиваются малым количеством комбинаций, которые соответствуют пяти представлениям архитектуры ПС. По этой причине UML включает девять видов диаграмм:

1) диаграммы классов;

2) диаграммы объектов;

3) диаграммы Use Case (диаграммы прецедентов);

4) диаграммы последовательности;

5) диаграммы сотрудничества (кооперации);

6) диаграммы схем состояний;

7) диаграммы деятельности;

8) компонентные диаграммы;

9) диаграммы размещения (развертывания).

*Диаграмма классов* показывает набор классов, интерфейсов, сотрудничеств и их отношений. При моделировании объектно-ориентированных систем диаграммы классов используются наиболее часто. Диаграммы классов обеспечивают статическое проектное представление системы. Диаграммы классов, включающие активные классы, обеспечивают статическое представление процессов системы.

*Диаграмма объектов* показывает набор объектов и их отношения. Диаграмма объектов представляет статический «моментальный снимок» с экземпляров предметов, которые находятся в диаграммах классов. Как и диаграммы классов, эти диаграммы обеспечивают статическое проектное представление или статическое представление процессов системы (но с точки зрения реальных или фототипичных случаев).

*Диаграмма Use Case* (диаграмма прецедентов) показывает набор элементов Use Case, актеров и их отношений. С помощью диаграмм Use Case для системы создается статическое представление Use Case. Эти диаграммы особенно важны при организации и моделировании поведения системы, задании требований заказчика к системе.

Диаграммы последовательности и диаграммы сотрудничества — это разновидности диаграмм взаимодействия.

*Диаграмма взаимодействия* показывает взаимодействие, включающее набор объектов и их отношений, а также пересылаемые между объектами сообщения. Диаграммы взаимодействия обеспечивают динамическое представление системы.

*Диаграмма последовательности —* это диаграмма взаимодействия, которая выделяет упорядочение сообщений по времени.

*Диаграмма сотрудничества* (диаграмма кооперации) — это диаграмма взаимодействия, которая выделяет структурную организацию объектов, посылающих и принимающих сообщения. Диаграммы последовательности и диаграммы сотрудничества изоморфны, что означает, что одну диаграмму можно трансформировать в другую диаграмму.

*Диаграмма схем состояний* показывает конечный автомат, представляет состояния, переходы, события и действия. Диаграммы схем состояний обеспечивают динамическое представление системы. Они особенно важны при моделировании поведения интерфейса, класса или сотрудничества. Эти диаграммы выделяют такое поведение объекта, которое управляется событиями, что особенно полезно при моделировании реактивных систем.

*Диаграмма деятельности —* специальная разновидность диаграммы схем состояний, которая показывает поток от действия к действию внутри системы. Диаграммы деятельности обеспечивают динамическое представление системы. Они особенно важны при моделировании функциональности системы и выделяют поток управления между объектами.

*Компонентная диаграмма* показывает организацию набора компонентов и зависимости между компонентами. Компонентные диаграммы обеспечивают статическое представление реализации системы. Они связаны с диаграммами классов в том смысле, что в компонент обычно отображается один или несколько классов, интерфейсов или коопераций.

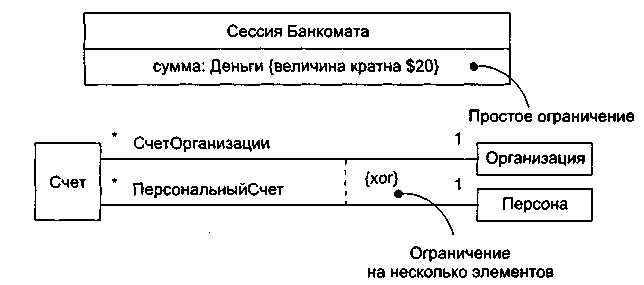
*Диаграмма размещения* (диаграмма развертывания) показывает конфигурацию обрабатывающих узлов периода выполнения, а также компоненты, живущие в них. Диаграммы размещения обеспечивают статическое представление размещения системы. Они связаны с компонентными диаграммами в том смысле, что узел обычно включает один или несколько компонентов.

## Механизмы расширения в UML

UML — развитый язык, имеющий большие возможности, но даже он не может отразить все нюансы, которые могут возникнуть при создании различных моделей. Поэтому UML создавался как открытый язык, допускающий контролируемые рас-. ширения. Механизмами расширения в UML являются:

* ограничения;
* теговые величины;
* стереотипы.

*Ограничение* (constraint) расширяет семантику строительного UML-блока, позволяя добавить новые правила или модифицировать существующие. Ограничение показывают как текстовую строку, заключенную в фигурные скобки {}. Например, на рис. 10.17 введено простое ограничение на свойство *сумма* класса *Сессия Банкомата —* его значение должно быть кратно 20. Кроме того, здесь показано ограничение на два элемента (две ассоциации), оно располагается возле пунктирной линии, соединяющей элементы, и имеет следующий смысл — владельцем конкретного счета не может быть и организация, и персона.



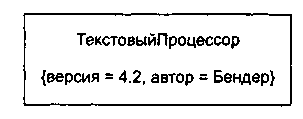
**Рис. 10.17.** Ограничения

*Теговая величина* (tagged value) расширяет характеристики строительного UML-блока, позволяя создать новую информацию в спецификации конкретного элемента. Теговую величину показывают как строку в фигурных скобках {}. Строка имеет вид

имя теговой величины = значение.

Иногда (в случае предопределенных тегов) указывается только имя теговой величины.

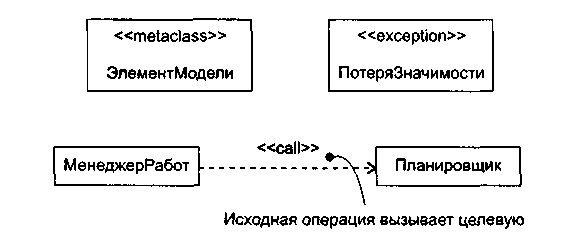
Отметим, что при работе с продуктом, имеющим много реализаций, полезно отслеживать версию и автора определенных блоков. Версия и автор не принадлежат к основным понятиям UML. Они могут быть добавлены к любому строительному блоку (например, к классу) введением в блок новых теговых величин. Например, на рис. 10.18 класс ТекстовыйПроцессор расширен путем явного указания его версии и автора.



**Рис. 10.18.** Расширение класса

*Стереотип* (stereotype) расширяет словарь языка, позволяет создавать новые виды строительных блоков, производные от существующих и учитывающие специфику новой проблемы. Элемент со стереотипом является вариацией существующего элемента, имеющей такую же форму, но отличающуюся по сути. У него могут быть дополнительные ограничения и теговые величины, а также другое визуальное представление. Он иначе обрабатывается при генерации программного кода. Отображают стереотип как имя, указываемое в двойных угловых скобках (или в угловых кавычках).

Примеры элементов со стереотипами приведены на рис. 10.19. Стереотип «exception» говорит о том, что класс ПотеряЗначимости теперь рассматривается как специальный класс, которому, положим, разрешается только генерация и обработка сигналов исключений. Особые возможности метакласса получил класс ЭлементМодели. Кроме того, здесь показано применение стереотипа «call» к отношению зависимости (у него появился новый смысл).



**Рис. 10.19.** Стереотипы

Таким образом, механизмы расширения позволяют адаптировать UML под нужды конкретных проектов и под новые программные технологии. Возможно добавление новых строительных блоков, модификация спецификаций существующих блоков и даже изменение их семантики. Конечно, очень важно обеспечить контролируемое введение расширений.

## Контрольные вопросы

1. Сколько поколений языков визуального моделирования вы знаете?
2. Назовите численность языков визуального моделирования 2-го поколения.
3. Какая необходимость привела к созданию языка визуального моделирования третьего поколения?
4. Поясните назначение UML.
5. Какие строительные блоки образуют словарь UML? Охарактеризуйте их.
6. Какие разновидности предметов UML вы знаете? Их назначение?
7. Перечислите известные вам разновидности структурных предметов UML.
8. Перечислите известные вам разновидности предметов поведения UML.
9. Перечислите известные вам группирующие предметы UML.
10. Перечислите известные вам поясняющие предметы UML.
11. Какие разновидности отношений предусмотрены в UML? Охарактеризуйте каждое отношение.
12. Дайте характеристику диаграммы классов.
13. Дайте характеристику диаграммы объектов.
14. Охарактеризуйте диаграмму Use Case.
15. Охарактеризуйте диаграммы взаимодействия.
16. Дайте характеристику диаграммы последовательности.
17. Дайте характеристику диаграммы сотрудничества.
18. Охарактеризуйте диаграмму схем состояний.
19. Охарактеризуйте диаграмму деятельности.
20. Дайте характеристику компонентной диаграммы.
21. Охарактеризуйте диаграмму размещения.
22. Для чего служат механизмы расширения в UML?
23. Поясните механизм ограничений в UML.
24. Объясните механизм теговых величин в UML.
25. В чем суть механизма стереотипов UML?

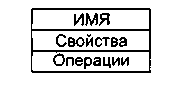
# 8. Статические модели объектно-ориентированных программных систем

Статические модели обеспечивают представление структуры систем в терминах базовых строительных блоков и отношений между ними. «Статичность» этих моделей состоит в том, что здесь не показывается динамика изменений системы во времени. Вместе с тем следует понимать, что эти модели несут в себе не только структурные описания, но и описания операций, реализующих заданное поведение системы. Основным средством для представления статических моделей являются диаграммы классов [8], [23], [53], [67]. Вершины диаграмм классов нагружены классами, а дуги (ребра) — отношениями между ними. Диаграммы используются:

* в ходе анализа — для указания ролей и обязанностей сущностей, которые обеспечивают поведение системы;
* в ходе проектирования — для фиксации структуры классов, которые формируют системную архитектуру.

## Вершины в диаграммах классов

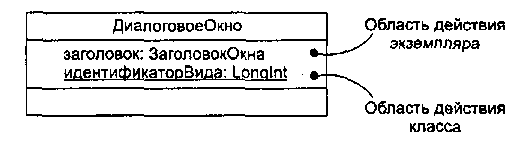
Итак, вершина в диаграмме классов — класс. Обозначение класса показано на рис. 11.1.



**Рис. 11.1.** Обозначение класса

Имя класса указывается всегда, свойства и операции — выборочно. Предусмотрено задание области действия свойства (операции). Если свойство (операция) подчеркивается, его областью действия является класс, в противном случае областью Действия является экземпляр (рис. 11.2).

Что это значит? Если областью действия свойства является класс, то все его экземпляры (объекты) используют общее значение этого свойства, в противном случае у каждого экземпляра свое значение свойства.



**Рис. 11.2.** Свойства уровней класса и экземпляра

### Свойства

Общий синтаксис представления свойства имеет вид

Видимость Имя [Множественность]: Тип = НачальнЗначение {Характеристики}

Рассмотрим видимость и характеристики свойств.

В языке UML определены три уровня видимости:

|  |  |
| --- | --- |
| public  protected  private | Любой клиент класса может использовать свойство (операцию), обозначается символом +  Любой наследник класса может использовать свойство (операцию), обозначается символом #  Свойство (операция) может использоваться только самим классом, обозначается символом - |

**ПРИМЕЧАНИЕ**

Если видимость не указана, считают, что свойство объявлено с публичной видимостью.

Определены три характеристики свойств:

|  |  |
| --- | --- |
| changeable  addOnly  frozen | Нет ограничений на модификацию значения свойства  Для свойств с множественностью, большей единицы; дополнительные значения могут быть добавлены, но после создания значение не может удаляться или изменяться  После инициализации объекта значение свойства не изменяется |

ПРИМЕЧАНИЕ

Если характеристика не указана, считают, что свойство объявлено с характеристикой changeable.

Примеры объявления свойств:

|  |  |
| --- | --- |
| начало  + начало  начало : Координаты  имяфамилия [0..1] : String  левыйУгол : Координаты=(0, 10)  сумма : Integer {frozen} | Только имя  Видимость и имя  Имя и тип  Имя, множественность, тип  Имя, тип, начальное значение  Имя и характеристика |

### Операции

Общий синтаксис представления операции имеет вид

Видимость Имя (Список Параметров): ВозвращаемыйТип {Характеристики}

Примеры объявления операций:

|  |  |
| --- | --- |
| записать  + записать  зарегистрировать) и: Имя, ф: Фамилия)  балансСчета ( ) : Integer  нагревать ( ) (guarded) | Только имя  Видимость и имя  Имя и параметры  Имя и возвращаемый тип  Имя и характеристика |

В сигнатуре операции можно указать ноль или более параметров, форма представления параметра имеет следующий синтаксис:

Направление Имя : Тип = ЗначениеПоУмолчанию

Элемент Направление может принимать одно из следующих значений:

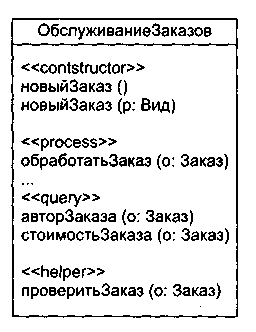
|  |  |
| --- | --- |
| in  out  inout | Входной параметр, не может модифицироваться  Выходной параметр, может модифицироваться для передачи информации в вызывающий объект  Входной параметр, может модифицироваться |

Допустимо применение следующих характеристик операций:

|  |  |
| --- | --- |
| leaf  isQuery  sequential  guarded  concurrent | Конечная операция, операция не может быть полиморфной и не может переопределяться (в цепочке наследования)  Выполнение операции не изменяет состояния объекта  В каждый момент времени в объект поступает только один вызов операций. Как следствие, в каждый момент времени выполняется только одна операция объекта. Другими словами, допустим только один поток вызовов (поток управления)  Допускается одновременное поступление в объект нескольких вызовов, но в каждый момент времени обрабатывается только один вызов охраняемой операции. Иначе говоря, параллельные потоки управления исполняются последовательно (за счет постановки вызовов в очередь)  В объект поступает несколько потоков вызовов операций (из параллельных потоков управления). Разрешается параллельное (и множественное) выполнение операции. Подразумевается, что такие операции являются атомарными |

### Организация свойств и операций

Известно, что пиктограмма класса включает три секции (для имени, для свойств и для операций). Пустота секции не означает, что у класса отсутствуют свойства или операции, просто в данный момент они не показываются. Можно явно определить наличие у класса большего количества свойств или атрибутов. Для этого в конце показанного списка проставляются три точки. Как показано на рис. 11.3, в длинных списках свойств и операций разрешается группировка — каждая группа начинается со своего стереотипа.



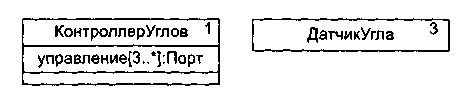
**Рис. 11.3.** Стереотипы для характеристик класса

### Множественность

Иногда бывает необходимо ограничить количество экземпляров класса:

* задать ноль экземпляров (в этом случае класс превращается в утилиту, которая предлагает свои свойства и операции);
* задать один экземпляр (класс-singleton);
* задать конкретное количество экземпляров;
* не ограничивать количество экземпляров (это случай, предполагаемый по умолчанию).

Количество экземпляров класса называется его множественностью. Выражение множественности записывается в правом верхнем углу значка класса. Например, как показано на рис. 11.4, КонтроллерУглов — это класс-singleton, а для класса ДатчикУгла разрешены три экземпляра.

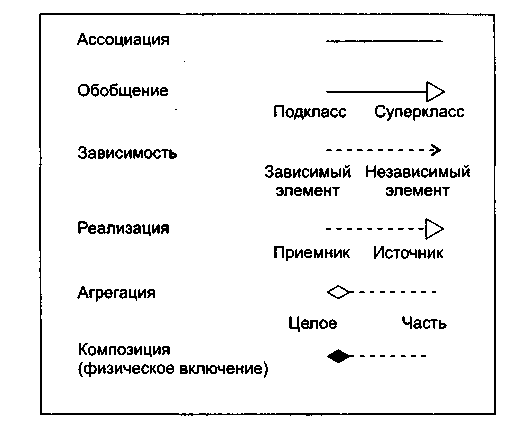


**Рис. 11.4.** Множественность

Множественность применима не только к классам, но и к свойствам. Множественность свойства задается выражением в квадратных скобках, записанным после его имени. Например, на рисунке заданы три и более экземпляра свойства Управление (в экземпляре класса КонтроллерУглов).

## Отношения в диаграммах классов

Отношения, используемые в диаграммах классов, показаны на рис. 11.5.



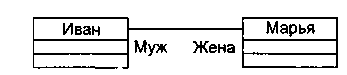
**Рис. 11.5.** Отношения в диаграммах классов

*Ассоциации* отображают структурные отношения между экземплярами классов, то есть соединения между объектами. Каждая ассоциация может иметь метку — *имя,* которое описывает природу отношения. Как показано на рис. 11.6, имени можно придать направление — достаточно добавить треугольник направления, который указывает направление, заданное для чтения имени.



**Рис. 11.6.** Имена ассоциаций

Когда класс участвует в ассоциации, он играет в этом отношении определенную роль. Как показано на рис. 11.7, *роль* определяет, каким представляется класс на одном конце ассоциации для класса на противоположном конце ассоциации.

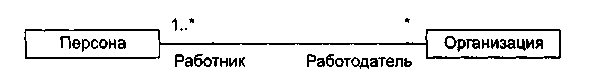


**Рис. 11.7.** Роли

Один и тот же класс в разных ассоциациях может играть разные роли. Часто важно знать, как много объектов может соединяться через экземпляр ассоциации. Это количество называется ложностью роли в ассоциации, записывается в виде выражения, задающего диапазон величин или одну величину (рис. 11.8).

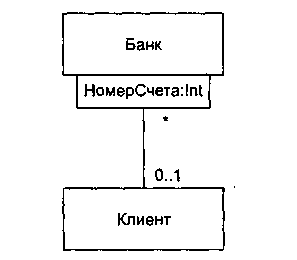
Запись мощности на одном конце ассоциации определяет количество объектов, соединяемых с каждым объектом на противоположном конце ассоциации. Например, можно задать следующие варианты мощности:

* 5 — точно пять;
* \* — неограниченное количество;
* 0..\* — ноль или более;
* 1..\* — один или более;
* 3..7 — определенный диапазон;
* 1..3, 7 — определенный диапазон или число.



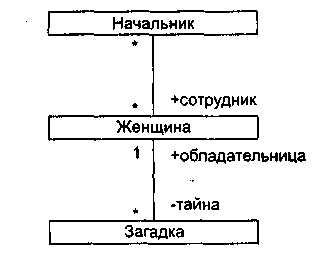
**Рис. 11. 8.** Мощность

Достаточно часто возникает следующая проблема — как для объекта на одном конце ассоциации выделить набор объектов на противоположном конце? Например, рассмотрим взаимодействие между банком и клиентом — вкладчиком. Как показано на рис. 11.9, мы устанавливаем ассоциацию между классом Банк и классом Клиент. В контексте Банка мы имеем НомерСчета, который позволяет идентифицировать конкретного Клиента. В этом смысле НомерСчета является атрибутом ассоциации. Он не является характеристикой Клиента, так как Клиенту не обязательно знать служебные параметры его счета. Теперь для данного экземпляра Банка и данного значения НомераСчета можно выявить ноль или один экземпляр Клиента. В UML для решения этой проблемы вводится *квалификатор —* атрибут ассоциации, чьи значения выделяют набор объектов, связанных с объектом через ассоциацию. Квалификатор изображается маленьким прямоугольником, присоединенным к концу ассоциации. В прямоугольник вписывается свойство — атрибут ассоциации.



**Рис. 11.9.** Квалификация

Кроме того, роли в ассоциациях могут иметь пометки *видимости.* Например, на рис. 11.10 показаны ассоциации между Начальником и Женщиной, а также между Женщиной и Загадкой. Для данного экземпляра Начальника можно определить соответствующие экземпляры Женщины. С другой стороны, Загадка приватна для Женщины, поэтому она недоступна извне. Как показано на рисунке, из объекта Начальника можно перемещаться к экземплярам Женщины (и наоборот), но нельзя видеть экземпляры Загадки для объектов Женщины.

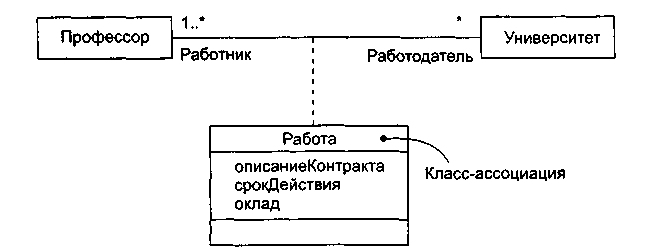


**Рис. 11.10.** Видимость

На конце ассоциации можно задать три уровня видимости, добавляя символ видимости к имени роли:

* по умолчанию для роли задается публичная видимость;
* приватная видимость указывает, что объекты на данном конце недоступны любым объектам вне ассоциации;
* защищенная видимость (protected) указывает, что объекты на данном конце недоступны любым объектам вне ассоциации, за исключением потомков того класса, который указан на противоположном конце ассоциации.

В языке UML ассоциации могут иметь свойства. Как показано на рис, 11.11, такие возможности отображаются с помощью классов-ассоциаций. Эти классы присоединяются к линии ассоциации пунктирной линией и рассматриваются как классы со свойствами ассоциаций или как ассоциации со свойствами классов.



**Рис. 11.11.** Класс-ассоциация

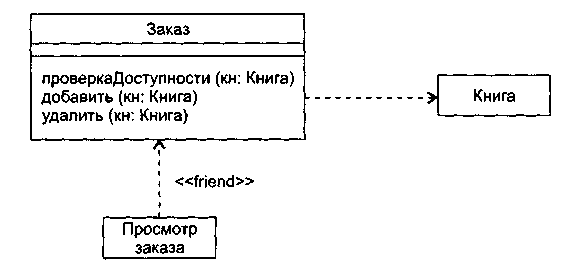
Свойства класса-ассоциации характеризуют не один, а пару объектов, в данном случае — пару экземпляров, Профессор и Университет.

Отношения агрегации и композиции в языке UML считаются разновидностями ассоциации, применяемыми для отображения структурных отношений между «целым» (агрегатом) и его «частями». *Агрегация* показывает отношение по ссылке (в агрегат включены только указатели на части), *композиция —* отношение физического включения (в агрегат включены сами части).

*Зависимость* является отношением использования между клиентом (зависимым элементом) и поставщиком (независимым элементом). Обычно операции клиента:

* вызывают операции поставщика;
* имеют сигнатуры, в которых возвращаемое значение или аргументы принадлежат классу поставщика.

Например, на рис. 11.12 показана зависимость класса Заказ от класса Книга, так как Книга используется в операциях проверкаДоступности, добавить и удалить класса Заказ.



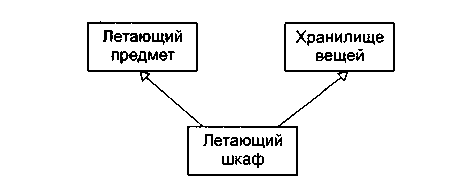
**Рис. 11.12.** Отношения зависимости

На этом рисунке изображена еще одна зависимость, которая показывает, что класс Просмотр Заказа использует класс Заказ. Причем Заказ ничего не знает о Просмотре Заказа. Данная зависимость помечена стереотипом «friend», который расширяет простую зависимость, определенную в языке. Отметим, что отношение зависимости очень разнообразно — в настоящее время язык предусматривает 17 разновидностей зависимости, различаемых по стереотипам.

*Обобщение —* отношение между общим предметом (суперклассом) и специализированной разновидностью этого предмета (подклассом). Подкласс может иметь одного родителя (один суперкласс) или несколько родителей (несколько суперклассов). Во втором случае говорят о множественном наследовании.

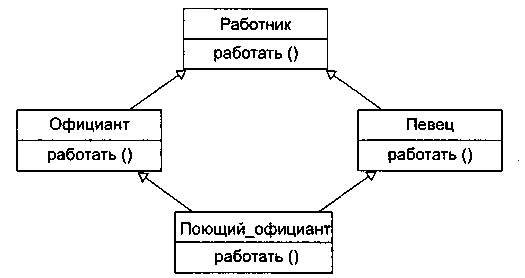
Как показано на рис. 11.13, подкласс Летающий шкаф является наследником суперклассов Летающий предмет и Хранилище вещей. Этому подклассу достаются в наследство все свойства и операции двух классов-родителей.

Множественное наследование достаточно сложно и коварно, имеет много «подводных камней». Например, подкласс Яблочный\_Пирог не следует производить от суперклассов Пирог и Яблоко. Это типичное неправильное использование множественного наследования: потомок наследует все свойства от его родителя, хотя обычно не все свойства применимы к потомку. Очевидно, что Яблочный\_Пирог является Пирогом, но не является Яблоком, так как пироги не растут на деревьях.



**Рис. 11.13.** Множественное наследование

Еще более сложные проблемы возникают при наследовании от двух классов, имеющих общего родителя. Говорят, что в результате образуется ромбовидная решетка наследования (рис. 11.14).

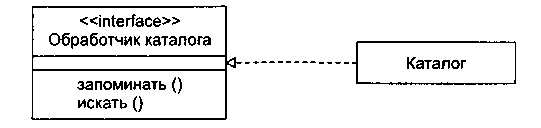


**Рис. 11.14.** Ромбовидная решетка наследования

Полагаем, что в подклассах Официант и Певец операция работать суперкласса Работник переопределена в соответствии с обязанностью подкласса (работа официанта состоит в обслуживании едой, а певца — в пении). Возникает вопрос — какую версию операции работать унаследует Поющий\_официант? А что делать со свойствами, доставшимися в наследство от родителей и общего прародителя? Хотим ли мы иметь несколько копий свойства или только одну?

Все эти проблемы увеличивают сложность реализации, приводят к введению многочисленных правил для обработки особых случаев.

*Реализация* — семантическое отношение между классами, в котором класс-приемник выполняет реализацию операций интерфейса класса-источника. Например, на рис. 11.15 показано, что класс Каталог должен реализовать интерфейс Обработчик каталога, то есть Обработчик каталога рассматривается как источник, а Каталог — как приемник.

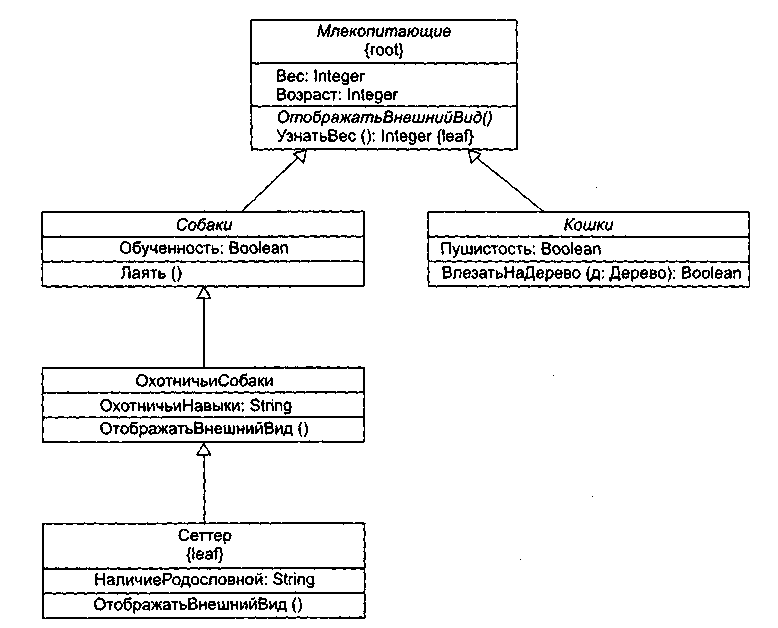


**Рис. 11.15.** Реализация интерфейса

Интерфейс Обработчик каталога позволяет клиентам взаимодействовать с объектами класса Каталог без знания той дисциплины доступа, которая здесь реализована (LIFO — последний вошел, первый вышел; FIFO — первый вошел, первый вышел и т. д.).

## Деревья наследования

При использовании отношений обобщения строится иерархия классов. Некоторые классы в этой иерархии могут быть абстрактными. *Абстрактным* называют класс, который не может иметь экземпляров. Имена абстрактных классов записываются курсивом. Например, на рис. 11.16 показаны абстрактные классы *Млекопитающие, Собаки, Кошки.*



**Рис. 11.16.** Абстрактность и полиморфизм

Кроме того, здесь имеются конкретные классы ОхотничьиСобаки, Сеттер, каждый из которых может иметь экземпляры.

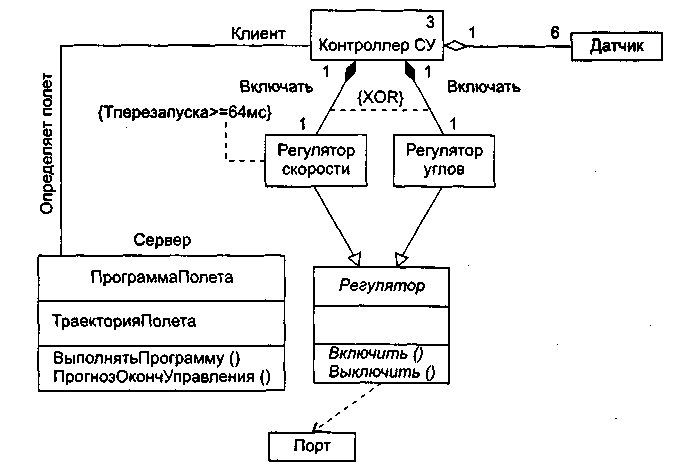
Обычно класс наследует какие-то характеристики класса-родителя и передает свои характеристики классу-потомку. Иногда требуется определить *конечный* класс, который не может иметь детей. Такие классы помечаются теговой величиной (характеристикой) leaf, записываемой за именем класса. Например, на рисунке показан конечный класс Сеттер.

Иногда полезно отметить *корневой* класс, который не может иметь родителей. Такой класс помечается теговой величиной (характеристикой) root, записываемой за именем класса. Например, на рисунке показан корневой класс *Млекопитающие.*

Аналогичные свойства имеют и операции. Обычно операция является полиморфной, это значит, что в различных точках иерархии можно определять операции с похожей сигнатурой. Такие операции из дочерних классов переопределяют поведение соответствующих операций из родительских классов. При обработке сообщения (в период выполнения) производится полиморфный выбор одной из операций иерархии в соответствии с типом объекта. Например, ОтображатьВнешнийВид () и ВлезатьНаДерево (дуб) — полиморфные операции. К тому же операция *Млекопитающие::ОтображатьВнешнийВид ( )* является абстрактной, то есть неполной и требующей для своей реализации потомка. Имя абстрактной операции записывается курсивом (как и имя класса). С другой стороны, Млекопитающие::УзнатьВес () — конечная операция, что отмечается характеристикой leaf. Это значит, что операция не полиморфна и не может перекрываться.

## Примеры диаграмм классов

В качестве первого примера на рис. 11.17 показана диаграмма классов системы управления полетом летательного аппарата.



**Рис. 11.17.** Диаграмма классов системы управления полетом

Здесь представлен класс ПрограммаПолета, который имеет свойство ТраекторияПолета, операцию-модификатор ВыполнятьПрограмму () и операцию-селектор ПрогнозОкончУправления (). Имеется ассоциация между этим классом и классом Контроллер СУ — экземпляры программы задают параметры движения, которые должны обеспечивать экземпляры контроллера.

Класс Контроллер СУ — агрегат, чьи экземпляры включают по одному экземпляру классов Регулятор скорости и Регулятор углов, а также по шесть экземпляров класса Датчик. Экземпляры Регулятора скорости и Регулятора углов включены в агрегат физически (с помощью отношения *композиция),* а экземпляры Датчика — по ссылке, то есть экземпляр Контроллера СУ включает лишь указатели на объекты-датчики. Регулятор скорости и Регулятор углов — это подклассы абстрактного суперкласса *Регулятор,* который передает им в наследство абстрактные операции *Включить ( )* и *Выключить ().* В свою очередь, класс *Регулятор* использует конкретный класс Порт.

Как видим, ассоциация имеет имя (Определяет полет), роли участников ассоциации явно указаны (Сервер, Клиент). Отношения композиции также имеют имена (Включать), причем на эти отношения наложено ограничение — контроллер не может включать Регулятор скорости и Регулятор углов одновременно.

Для класса Контроллер СУ задано ограничение на множественность — допускается не более трех экземпляров этого класса. Класс Регулятор скорости имеет ограничение другого типа — повторное включение его экземпляра разрешается не раньше, чем через 64 мс.

В качестве второго примера на рис. 11.18 приведена диаграмма классов для информационной системы театра. Эту систему образует 6 классов.

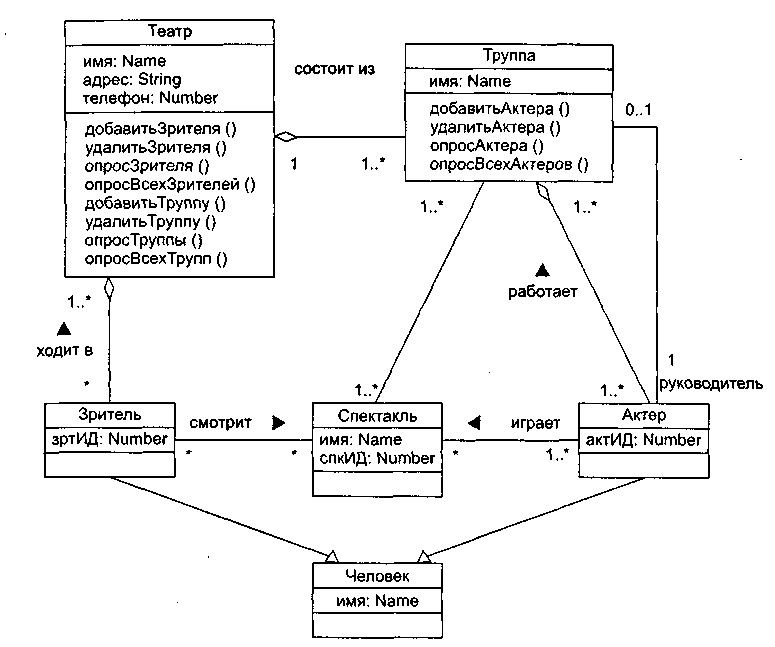
Классы-агрегаты Театр и Труппа имеют операции добавления и удаления своих частей, которые включаются в агрегаты по ссылке. Частями Театра являются Зрители и Труппы, а частями Труппы — Актеры. Отношения агрегации между классом Театр и классами Труппа и Зритель слегка отличны. Театр может состоять из одной или нескольких трупп, но каждая труппа находится в одном и только одном театре. С другой стороны, в театр может ходить любое количество зрителей (включая нулевое количество), причем зритель может посещать один или несколько театров.

Между классами Труппа и Актер существуют два отношения — агрегация и ассоциация. Агрегация показывает, что каждый актер работает в одной или нескольких труппах, а в каждой труппе должен быть хотя бы один актер. Ассоциация отображает, что каждой труппой управляет только один актер — художественный руководитель, а некоторые актеры не являются руководителями.

Ассоциация между классами Спектакль и Актер фиксирует, что в спектакле должен быть занят хотя бы один актер, впрочем, актер может играть в любом количестве спектаклей (или вообще может ничего не играть).

Между классами Спектакль и Зритель тоже определена ассоциация. Она поясняет, что зритель может смотреть любое число спектаклей, а на каждом спектакле может быть любое число зрителей.

И наконец, на диаграмме отображены два отношения наследования, утверждающие, что и в зрителях, и в актерах есть человеческое начало.



**Рис. 11.18.** Диаграмма классов информационной системы театра

## Контрольные вопросы

1. Поясните назначение статических моделей объектно-ориентированных программных систем.
2. Что является основным средством для представления статических моделей?
3. Как используются статические модели?
4. Какие секции входят в графическое обозначение класса?
5. Какие секции класса можно не показывать?
6. Какие имеются разновидности области действия свойства (операции)?
7. Поясните общий синтаксис представления свойства.
8. Какие уровни видимости вы знаете? Их смысл?
9. Какие характеристики свойств вам известны?
10. Поясните общий синтаксис представления операции.
11. Какой вид имеет форма представления параметра операции?
12. Какие характеристики операций вам известны?
13. Что означают три точки в списке свойств (операций)?
14. Как организуется группировка свойств (операций)?
15. Как ограничить количество экземпляров класса?
16. Перечислите известные вам «украшения» отношения ассоциации.
17. Может ли статическая модель программной системы не иметь отношений ассоциации?
18. Какой смысл имеет квалификатор? К чему он относится?
19. Какие отношения могут иметь пометки видимости и что эти пометки обозначают?
20. Какой смысл имеет класс-ассоциация?
21. Чем отличается агрегация от композиции? Разновидностями какого отношения (в UML) они являются?
22. Что обозначает в UML простая зависимость?
23. Какой смысл имеет отношение обобщения?
24. Какие недостатки у множественного наследования?
25. Перечислите недостатки ромбовидной решетки наследования.
26. В чем смысл отношения реализации?
27. Что обозначает мощность «многие-ко-многим» и в каких отношениях она применяется?
28. Что такое абстрактный класс (операция) и как он (она) отображается?
29. Как запретить полиморфизм операции?
30. Как обозначить корневой класс?

# 9. Динамические модели объектно-ориентированных программных систем

Динамические модели обеспечивают представление поведения систем. «Динамизм» этих моделей состоит в том, что в них отражается изменение состояний в процессе работы системы (в зависимости от времени). Средства языка UML для создания динамических моделей многочисленны и разнообразны [8], [23], [41], [53], [67]. Эти средства ориентированы не только на собственно программные системы, но и на отображение требований заказчика к поведению таких систем.

## Моделирование поведения программной системы

Для моделирования поведения системы используют:

* автоматы;
* взаимодействия.

Автомат (State machine) описывает поведение в терминах последовательности состояний, через которые проходит объект в течение своей жизни. Взаимодействие (Interaction) описывает поведение в терминах обмена сообщениями между объектами.

Таким образом, автомат задает поведение системы как цельной, единой сущности; моделирует жизненный цикл единого объекта. В силу этого автоматный подход удобно применять для формализации динамики отдельного трудного для понимания блока системы.

Взаимодействия определяют поведение системы в виде коммуникаций между его частями (объектами), представляя систему как сообщество совместно работающих объектов. Именно поэтому взаимодействия считают основным аппаратом для фиксации полной динамики системы.

Автоматы отображают с помощью:

* диаграмм схем состояний;
* диаграмм деятельности.

Взаимодействия отображают с помощью:

* диаграмм сотрудничества (кооперации);
* диаграмм последовательности.

## Диаграммы схем состояний

Диаграмма схем состояний — одна из пяти диаграмм UML, моделирующих динамику систем. Диаграмма схем состояний отображает конечный автомат, выделяя поток управления, следующий от состояния к состоянию. Конечный автомат — поведение, которое определяет последовательность состояний в ходе существования объекта. Эта последовательность рассматривается как ответ на события и включает реакции на эти события.

Диаграмма схем состояний показывает:

1) набор состояний системы;

2) события, которые вызывают переход из одного состояния в другое;

3) действия, которые происходят в результате изменения состояния.

В языке UML состоянием называют период в жизни объекта, на протяжении которого он удовлетворяет какому-то условию, выполняет определенную деятельность или ожидает некоторого события. Как показано на рис. 12.1, состояние изображается как закругленный прямоугольник, обычно включающий его имя и подсостоя-ния (если они есть).



**Рис. 12.1.** Обозначение состояния

Переходы между состояниями отображаются помеченными стрелками (рис. 12.2).



**Рис. 12.2.** Переходы между состояниями

На рис. 12.2 обозначено: Событие — происшествие, вызывающее изменение состояния, Действие — набор операций, запускаемых событием.

Иначе говоря, события вызывают переходы, а действия являются реакциями на переходы.

Примеры событий:

|  |  |
| --- | --- |
| баланс < 0  помехи  уменьшить(Давление)  after (5 seconds)  when (time = 16:30) | Изменение в состоянии  Сигнал (объект с именем)  Вызов действия  Истечение периода времени  Наступление абсолютного момента времени |

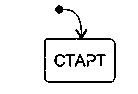
Примеры действий:

|  |  |
| --- | --- |
| Кассир. прекратитьВыплаты( )  flt:= new(Фильтp); Ш.убратьПомехи( )  send Ник. привет | Вызов одной операции  Вызов двух операций  Посылка сигнала в объект Ник |

**ПРИМЕЧАНИЕ**

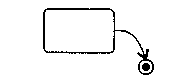
Для отображения посылки сигнала используют специальное обозначение — перед именем сигнала указывают служебное слово send.

Для отображения перехода в начальное состояние принято обозначение, показанное на рис. 12.3.



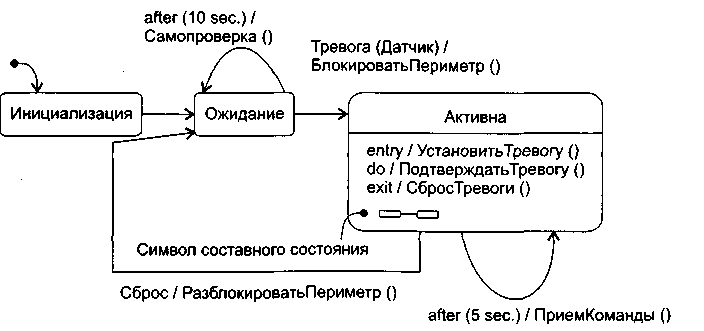
**Рис. 12.3.** Переход в начальное состояние

Соответственно, обозначение перехода в конечное состояние имеет вид, представленный на рис. 12.4.



**Рис. 12.4.** Переход в конечное состояние

В качестве примера на рис. 12.5 показана диаграмма схем состояний для системы охранной сигнализации.



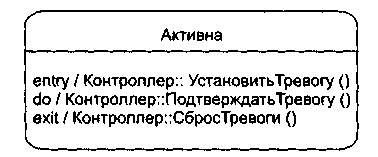
**Рис. 12.5.** Диаграмма схем состояний системы охранной сигнализации

Из рисунка видно, что система начинает свою жизнь в состоянии Инициализация, затем переходит в состояние Ожидание. В этом состоянии через каждые 10 секунд (по событию after (10 sec.)) выполняется самопроверка системы (операция Самопроверка ()). При наступлении события Тревога (Датчик) реализуются действия, связанные с блокировкой периметра охраняемого объекта, — исполняется операция БлокироватьПериметр() и осуществляется переход в состояние Активна. В активном состоянии через каждые 5 секунд по событию after (5 sec.) запускается операция ПриемКоманды(). Если команда получена (наступило событие Сброс), система возвращается в состояние Ожидание. В процессе возврата разблокируется периметр охраняемого объекта (операция РазблокироватьПериметр()).

### Действия в состояниях

Для указания действий, выполняемых при входе в состояние и при выходе из состояния, используются метки entry и exit соответственно.

Например, как показано на рис. 12.6, при входе в состояние Активна выполняется операция УстановитьТревогу() из класса Контроллер, а при выходе из состояния — операция СбросТревоги().



**Рис. 12.6.** Входные и выходные действия и деятельность в состоянии Активна

Действие, которое должно выполняться, когда система находится в данном состоянии, указывается после метки do. Считается, что такое действие начинается при входе в состояние и заканчивается при выходе из него. Например, в состоянии Активна это действие ПодтверждатьТревогу().

### Условные переходы

Между состояниями возможны различные типы переходов. Обычно переход инициируется событием. Допускаются переходы и без событий. Наконец, разрешены условные или охраняемые переходы.

Правила пометки стрелок условных переходов иллюстрирует рис. 12.7.

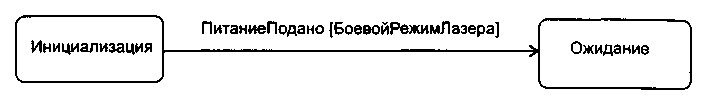


**Рис. 12.7.** Обозначение условного перехода

*Порядок выполнения условного перехода:*

1. происходит событие;
2. вычисляется условие УсловиеПерехода;
3. при УсловиеПерехода=true запускается переход и активизируется действие, в противном случае переход не выполняется.

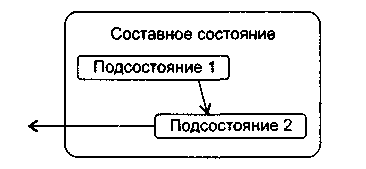
Пример условного перехода между состояниями Инициализация и Ожидание приведен на рис. 12.8. Он происходит по событию ПитаниеПодано, но только в том случае, если достигнут боевой режим лазера.



**Рис. 12.8.** Условный переход между состояниями

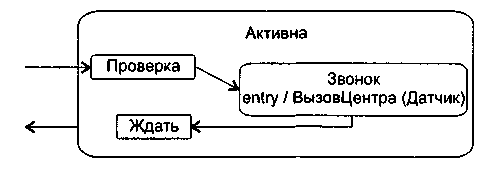
### Вложенные состояния

Одной из наиболее важных характеристик конечных автоматов в UML является подсостояние. Подсостояние позволяет значительно упростить моделирование сложного поведения. Подсостояние — это состояние, вложенное в другое состояние. На рис. 12.9 показано составное состояние, содержащее в себе два подсостояния.



**Рис. 12.9.** Обозначение подсостояний

На рис. 12.10 приведена внутренняя структура составного состояния Активна.

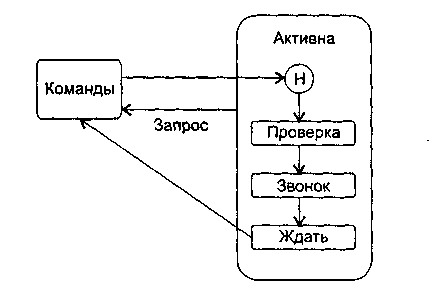


**Рис. 12.10.** Переходы в состоянии Активна

Семантика вложенности такова: если система находится в состоянии Активна, то она должна быть точно в одном из подсостояний: Проверка, Звонок, Ждать. В свою очередь, в подсостояние могут вкладываться другие подсостояния. Степень вложенности подсостояний не ограничивается. Данная семантика соответствует случаю последовательных подсостояний.

Возможно наличие параллельных подсостояний — они выполняются параллельно внутри составного состояния. Графически изображения параллельных подсостояний отделяются друг от друга пунктирными линиями.

Иногда при возврате в составное состояние возникает необходимость попасть в то его подсостояние, которое в прошлый раз было последним. Такое подсостояние называют историческим. Информация об историческом состоянии запоминается. Как показано на рис. 12.11, подобная семантика переходов отображается значком истории — буквой Н внутри кружка.



**Рис. 12.11.** Историческое состояние

При первом посещении состояния Активна автомат не имеет истории, поэтому происходит простой переход в подсостояние Проверка. Предположим, что в подсостоя-нии Звонок произошло событие Запрос. Средства управления заставляют автомат покинуть подсостояние Звонок (и состояние Активна) и вернуться в состояние Команды. Когда работа в состоянии Команды завершается, выполняется возврат в историческое подсостояние состояния Активна. Поскольку теперь автомат запомнил историю, он переходит прямо в подсостояние Звонок (минуя подсостояние Проверка).

Как показано на рис. 12.12, для обозначения составного состояния, имеющего внутри себя скрытые (не показанные на диаграмме) подсостояния, используется символ «очки».



**Рис. 12.12.** Символ состояния со скрытыми подсостояниями

## Диаграммы деятельности

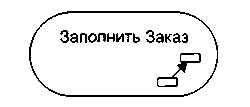
Диаграмма деятельности представляет особую форму конечного автомата, в которой показываются процесс вычислений и потоки работ. В ней выделяются не обычные состояния объекта, а состояния выполняемых вычислений — состояния действий. При этом полагается, что процесс вычислений не прерывается внешними событиями. Словом, диаграммы деятельности очень похожи на блок-схемы алгоритмов.

Основной вершиной в диаграмме деятельности является состояние действия (рис. 12.13), которое изображается как прямоугольник с закругленными боковыми сторонами.



**Рис. 12.13.** Состояние действия

Состояние действия считается атомарным (действие нельзя прервать) и выполняется за один квант времени, его нельзя подвергнуть декомпозиции. Если нужно представить сложное действие, которое можно подвергнуть дальнейшей декомпозиции (разбить на ряд более простых действий), то используют состояние под-деятельности. Изображение состояния под-деятельности содержит пиктограмму в правом нижнем углу (рис. 12.14).



**Рис. 12.14.** Состояние под-деятельности

Фактически в данную вершину вписывается имя другой диаграммы, имеющей внутреннюю структуру.

Переходы между вершинами — состояниями действий — изображаются в виде стрелок. Переходы выполняются по окончании действий.

Кроме того, в диаграммах деятельности используются вспомогательные вершины:

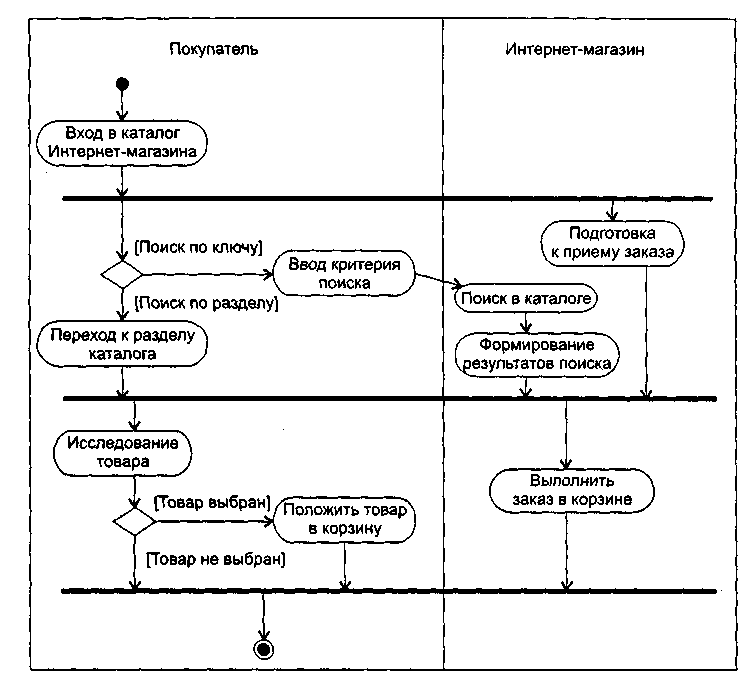
* решение (ромбик с одной входящей и несколькими исходящими стрелками);
* объединение (ромбик с несколькими входящими и одной исходящей стрелкой);
* линейка синхронизации — разделение (жирная горизонтальная линия с одной входящей и несколькими исходящими стрелками);
* линейка синхронизации — слияние (жирная горизонтальная линия с несколькими входящими и одной исходящей стрелкой);
* начальное состояние (черный кружок);
* конечное состояние (незакрашенный кружок, в котором размещен черный кружок меньшего размера).

Вершина «решение» позволяет отобразить разветвление вычислительного процесса, исходящие из него стрелки помечаются сторожевыми условиями ветвления.

Вершина «объединение» отмечает точку слияния альтернативных потоков действий.

Линейки синхронизации позволяют показать параллельные потоки действий, отмечая точки их синхронизации при запуске (момент разделения) и при завершении (момент слияния).

Пример диаграммы деятельности приведен на рис. 12.15. Эта диаграмма описывает деятельность покупателя в Интернет-магазине. Здесь представлены две точки ветвления — для выбора способа поиска товара и для принятия решения о покупке. Присутствуют три линейки синхронизации: верхняя отражает разделение на два параллельных процесса, средняя отражает и разделение, и слияние процессов, а нижняя — только слияние процессов.



**Рис. 12.15.** Диаграмма деятельности покупателя в Интернет-магазине

Дополнительно на этой диаграмме показаны две плавательные дорожки — дорожка покупателя и дорожка магазина, которые разделены вертикальной линией. Каждая дорожка имеет имя и фиксирует область деятельности конкретного лица, обозначая зону его ответственности.

## Диаграммы взаимодействия

Диаграммы взаимодействия предназначены для моделирования динамических аспектов системы. Диаграмма взаимодействия показывает взаимодействие, включающее набор объектов и их отношений, а также пересылаемые между объектами сообщения. Существуют две разновидности диаграммы взаимодействия — диаграмма последовательности и диаграмма сотрудничества. Диаграмма последовательности — это диаграмма взаимодействия, которая выделяет упорядочение сообщений по времени. Диаграмма сотрудничества — это диаграмма взаимодействия, которая выделяет структурную организацию объектов, посылающих и принимающих сообщения. Элементами диаграмм взаимодействия являются участники взаимодействия — объекты, связи, сообщения.

## Диаграммы сотрудничества

Диаграммы сотрудничества отображают взаимодействие объектов в процессе функционирования системы. Такие диаграммы моделируют сценарии поведения системы. В русской литературе диаграммы сотрудничества часто называют диаграммами кооперации.

Обозначение объекта показано на рис. 12.16.



**Рис. 12.16.** Обозначение объекта

Имя объекта подчеркивается и указывается всегда, свойства указываются выборочно. Синтаксис представления имени имеет вид

ИмяОбъекта : ИмяКласса

*Примеры записи имени:*

|  |  |
| --- | --- |
| Адам : Человек  : Пользователь  мойКомпьютер  агент : | Имя объекта и класса  Только имя класса (анонимный объект)  Только имя объекта (подразумевается, что имя класса известно)  Объект — сирота (подразумевается, что имя класса неизвестно) |

Синтаксис представления свойства имеет вид

Имя : Тип = Значение

*Примеры записи свойства:*

|  |  |
| --- | --- |
| номер:Телефон = "7350-420"  активен = True | Имя, тип, значение  Имя и значение |

Объекты взаимодействуют друг с другом с помощью связей — каналов для передачи сообщений. Связь между парой объектов рассматривается как экземпляр ассоциации между их классами. Иными словами, связь между двумя объектами существует только тогда, когда имеется ассоциация между их классами. Неявно все классы имеют ассоциацию сами с собой, следовательно, объект может послать сообщение самому себе.

Итак, связь — это путь для пересылки сообщения. Путь может быть снабжен характеристикой видимости. Характеристика видимости проставляется как стандартный стереотип над дальним концом связи. В языке предусмотрены следующие стандартные стереотипы видимости:

|  |  |
| --- | --- |
| «global»  «local»  «parameter»  «self» | Объект-поставщик находится в глобальной области определения  Объект-поставщик находится в локальной области определения объекта-клиента  Объект-поставщик является параметром операции объекта-клиента  Один и тот же объект является и клиентом, и поставщиком |

Сообщение — это спецификация передачи информации между объектами в ожидании того, что будет обеспечена требуемая деятельность. Прием сообщения рассматривается как событие.

Результатом обработки сообщения обычно является действие. В языке UML моделируются следующие разновидности действий:

|  |  |
| --- | --- |
| Вызов  Возврат  Посылка(Send)  Создание  Уничтожение | В объекте запускается операция  Возврат значения в вызывающий объект  В объект посылается сигнал  Создание объекта, выполняется по стандартному сообщению «create»  Уничтожение объекта, выполняется по стандартному сообщению «destroy» |

Для записи сообщений в языке UML принят следующий синтаксис:

ВозврВеличина := ИмяСообщения (Аргументы),

где ВозврВеличина задает величину, возвращаемую как результат обработки сообщения.

*Примеры записи сообщений:*

|  |  |
| --- | --- |
| Коорд := ТекущПоложение(самолетТ1)  оповещение( )  УстановитьМаршрут(х)  «create» | Вызов операции, возврат значения  Посылка сигнала  Вызов операции с действительным параметром  Стандартное сообщение для создания объекта |

Когда объект посылает сообщение в другой объект (делегируя некоторое действие получателю), объект-получатель, в свою очередь, может послать сообщение в третий объект, и т. д. Так формируется поток сообщений — последовательность управления. Очевидно, что сообщения в последовательности должны быть пронумерованы. Номера записываются перед именами сообщений, направления сообщений указываются стрелками (размещаются над линиями связей).

Наиболее общую форму управления задает процедурный или вложенный поток (поток синхронных сообщений). Как показано на рис. 12.17, процедурный поток рисуется стрелками с заполненными наконечниками.



**Рис. 12.17.** Поток синхронных сообщений

Здесь сообщение 2.1 : Напиток : = Изготовить(Смесь№3) определено как первое сообщение, вложенное во второе сообщение 2 : Заказать(Смесь№3) последовательности, а сообщение 2.2 : Принести(Напиток) — как второе вложенное сообщение. Все сообщения процедурной последовательности считаются синхронными. Работа с синхронным сообщением подчиняется следующему правилу: передатчик ждет до тех пор, пока получатель не примет и не обработает сообщение. В нашем примере это означает, что третье сообщение будет послано только после обработки сообщений 2.1 и 2.2. Отметим, что степень вложенности сообщений может быть любой. Главное, чтобы соблюдалось правило: последовательность сообщений внешнего уровня возобновляется только после завершения вложенной последовательности.

Менее общую форму управления задает асинхронный поток управления. Как показано на рис. 12.18, асинхронный поток рисуется обычными стрелками. Здесь все сообщения считаются асинхронными, при которых передатчик не ждет реакции от получателя сообщения. Такой вид коммуникации имеет семантику почтового ящика — получатель принимает сообщение по мере готовности. Иными словами, передатчик и получатель не синхронизируют свою работу, скорее, один объект «избавляется» от сообщения для другого объекта. В нашем примере сообщение ПодтверждениеВызова определено как второе сообщение в последовательности.



**Рис. 12.18.** Поток асинхронных сообщений

Помимо рассмотренных линейных потоков управления, можно моделировать и более сложные формы — итерации и ветвления.

Итерация представляет повторяющуюся последовательность сообщений. После номера сообщения итерации добавляется выражение

\*[i := 1 .. n].

Оно означает, что сообщение итерации будет повторяться заданное количество раз. Например, четырехкратное повторение первого сообщения РисоватьСторонуПрямоугольника можно задать выражением

1\*[1 := 1 .. 4] : РисоватьСторонуПрямоугольника(i)

Для моделирования ветвления после номера сообщения добавляется выражение условия, например: [х>0]. Сообщение альтернативной ветви помечается таким же номером, но с другим условием: [х<=0]. Пример итерационного и разветвляющегося потока сообщений приведен на рис. 12.19.

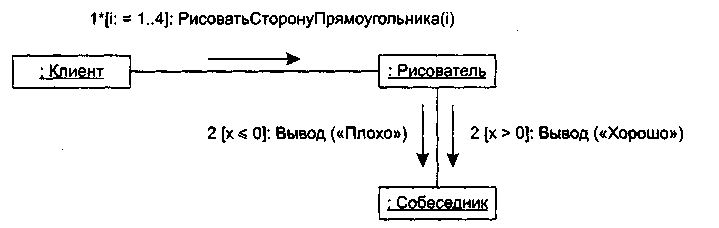
Здесь первое сообщение повторяется 4 раза, а в качестве второго выбирается одно из двух сообщений (в зависимости от значения переменной х). В итоге экземпляр рисователя нарисует на экране прямоугольное окно, а экземпляр собеседника выведет в него соответствующее донесение.

Таким образом, для формирования диаграммы сотрудничества выполняются следующие действия:

1) отображаются объекты, которые участвуют во взаимодействии;

2) рисуются связи, соединяющие эти объекты;

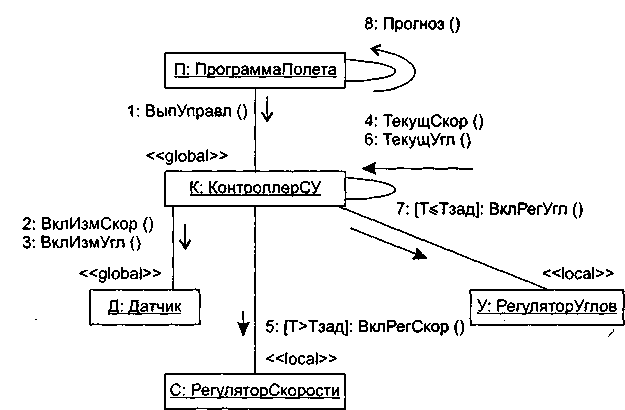
3) связи помечаются сообщениями, которые посылают и получают выделенные объекты.



**Рис. 12.19.** Итерация и ветвление

В итоге формируется ясное визуальное представление потока управления (в контексте структурной организации сотрудничающих объектов).

В качестве примера на рис. 12.20 приведена диаграмма сотрудничества системы управления полетом летательного аппарата.



**Рис. 12.20.** Диаграмма сотрудничества системы управления полетом

На данной диаграмме представлены пять объектов, явно показаны характеристики видимости всех связей системы. Поток управления в системе включает восемь сообщений: четыре асинхронных и четыре синхронных сообщения. Экземпляр Контроллера СУ ждет приема и обработки сообщений:

* ВклРегСкор( );
* ВрРегУгл();
* ТекущСкор();
* ТекущУгл( ).

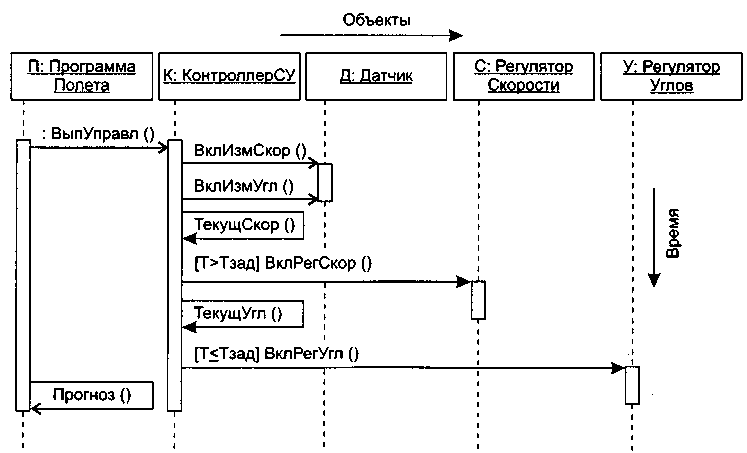
Порядок следования сообщений задан их номерами. Для пятого и седьмого сообщений указаны условия:

* включение Регулятора Скорости происходит, если относительное время полета *Т* больше заданного периода *Тзад*;
* включение Регулятора Углов обеспечивается, если относительное время поле-; та меньше или равно заданному периоду.

## Диаграммы последовательности

Диаграмма последовательности — вторая разновидность диаграмм взаимодействия. Отражая сценарий поведения в системе, эта диаграмма обеспечивает более наглядное представление порядка передачи сообщений. Правда, она не позволяет показать такие детали, которые видны на диаграмме сотрудничества (структурные характеристики объектов и связей).

Графически диаграмма последовательности — разновидность таблицы, которая показывает объекты, размещенные вдоль оси *X,* и сообщения, упорядоченные по времени вдоль оси *Y.*



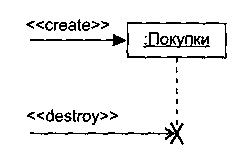
**Рис. 12.21.** Диаграмма последовательности системы управления полетом

Как показано на рис. 12.21, объекты, участвующие во взаимодействии, помещаются на вершине диаграммы, вдоль оси *X.* Обычно слева размещается объект, инициирующий взаимодействие, а справа — объекты по возрастанию подчиненности. Сообщения, посылаемые и принимаемые объектами, помещаются вдоль оси *Y* в порядке возрастания времени от вершины к основанию диаграммы. Используются те же синтаксис и обозначения синхронизации, что и в диаграммах сотрудничества. Таким образом, обеспечивается простое визуальное представление потока управления во времени.

От диаграмм сотрудничества диаграммы последовательности отличают две важные характеристики.

Первая характеристика — *линия жизни* объекта.

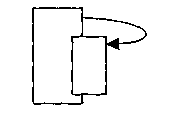
Линия жизни объекта — это вертикальная пунктирная линия, которая обозначает период существования объекта. Большинство объектов существуют на протяжении всего взаимодействия, их линии жизни тянутся от вершины до основания диаграммы. Впрочем, объекты могут создаваться в ходе взаимодействия. Их линии жизни начинаются с момента приема сообщения «create». Кроме того, объекты могут уничтожаться в ходе взаимодействия. Их линии жизни заканчиваются с момента приема сообщения «destroy». Как представлено на рис. 12.22, уничтожение линии жизни отмечается пометкой X в конце линии:



**Рис. 12.22.** Создание и уничтожение объекта

Вторая характеристика — *фокус управления.*

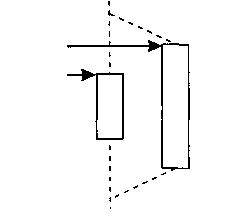
Фокус управления — это высокий тонкий прямоугольник, отображающий период времени, в течение которого объект выполняет действие (свою или подчиненную процедуру). Вершина прямоугольника отмечает начало действия, а основание — его завершение. Момент завершения может маркироваться сообщением возврата, которое показывается пунктирной стрелкой. Можно показать вложение фокуса управления (например, рекурсивный вызов собственной операции). Для этого второй фокус управления рисуется немного правее первого (рис. 12.23).



**Рис. 12.23.** Вложение фокусов управления

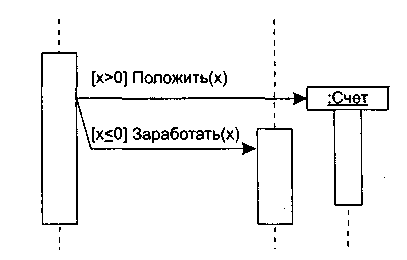
*Замечания.*

1. Для отображения «условности» линия жизни может быть разделена на несколько параллельных линий жизни. Каждая отдельная линия соответствует условному ветвлению во взаимодействии. Далее в некоторой точке линии жизни могут быть снова слиты (рис. 12.24).



**Рис. 12.24.** Параллельные линии жизни

1. Ветвление показывается множеством стрелок, идущих из одной точки. Каждая стрелка отмечается сторожевым условием (рис. 12.25).



**Рис. 12.25.** Ветвление

## Диаграммы Use Case

Диаграмма Use Case определяет поведение системы с точки зрения пользователя. Диаграмма Use Case рассматривается как главное средство для первичного моделирования динамики системы, используется для выяснения требований к разрабатываемой системе, фиксации этих требований в форме, которая позволит проводить дальнейшую разработку. В русской литературе диаграммы Use Case часто называют диаграммами прецедентов, или диаграммами вариантов использования.

В состав диаграмм Use Case входят элементы Use Case, актеры, а также отношения зависимости, обобщения и ассоциации. Как и другие диаграммы, диаграммы Use Case могут включать примечания и ограничения. Кроме того, диаграммы Use Case могут содержать пакеты, используемые для группировки элементов модели в крупные фрагменты.

### Актеры и элементы Use Case

Вершинами в диаграмме Use Case являются актеры и элементы Use Case. Их обозначения показаны на рис. 12.26.

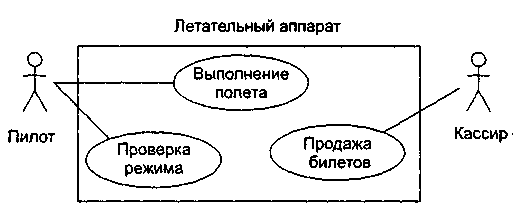
Актеры представляют внешний мир, нуждающийся в работе системы. Элементы Use Case представляют действия, выполняемые системой в интересах актеров.



**Рис. 12.26.** Обозначения актера и элемента Use Case

*Актер —* это роль объекта вне системы, который прямо взаимодействует с ее частью — конкретным элементом (элементом Use Case). Различают актеров и пользователей. Пользователь — это физический объект, который использует систему. Он может играть несколько ролей и поэтому может моделироваться несколькими актерами. Справедливо и обратное — актером могут быть разные пользователи.

Например, для коммерческого летательного аппарата можно выделить двух актеров: пилота и кассира. Сидоров — пользователь, который иногда действует как пилот, а иногда — как кассир. Как изображено на рис. 12.27, в зависимости от роли Сидоров взаимодействует с разными элементами Use Case.



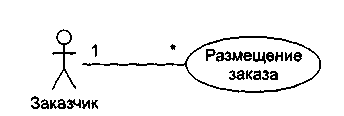
**Рис. 12.27.** Модель Use Case

*Элемент Use Case* — это описание последовательности действий (или нескольких последовательностей), которые выполняются системой и производят для отдельного актера видимый результат.

Один актер может использовать несколько элементов Use Case, и наоборот, один элемент Use Case может иметь несколько актеров, использующих его. Каждый элемент Use Case задает определенный путь использования системы. Набор всех элементов Use Case определяет полные функциональные возможности системы.

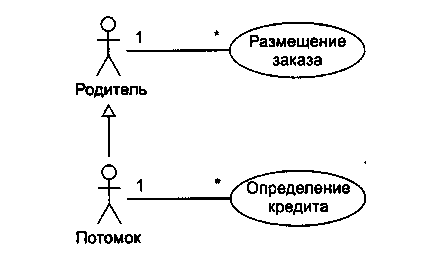
### Отношения в диаграммах Use Case

Между актером и элементом Use Case возможен только один вид отношения — ассоциация, отображающая их взаимодействие (рис. 12.28). Как и любая другая ассоциация, она может быть помечена именем, ролями, мощностью.



**Рис. 12.28.** Отношение ассоциации

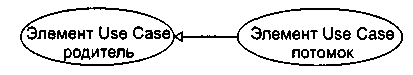
Между актерами допустимо отношение обобщения (рис. 12.29), означающее, что экземпляр потомка может взаимодействовать с такими же разновидностями экземпляров элементов Use Case, что и экземпляр родителя.



**Рис. 12.29.** Отношение обобщения между актерами

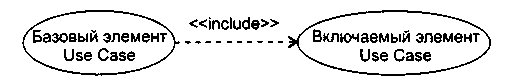
Между элементами Use Case определены отношение обобщения и две разновидности отношения зависимости — включения и расширения.

Отношение обобщения (рис. 12.30) фиксирует, что потомок наследует поведение родителя. Кроме того, потомок может дополнить или переопределить поведение родителя. Элемент Use Case, являющийся потомком, может замещать элемент Use Case, являющийся родителем, в любом месте диаграммы.



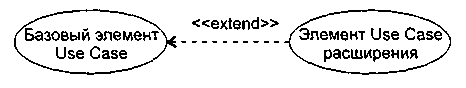
**Рис. 12.30.** Отношение обобщения между элементами Use Case

Отношение включения (рис. 12.31) между элементами Use Case означает, что базовый элемент Use Case *явно* включает поведение другого элемента Use Case в точке, которая определена в базе. Включаемый элемент Use Case никогда не используется самостоятельно — его конкретизация может быть только частью другого, большего элемента Use Case. Отношение включения является примером отношения делегации. При этом в отдельное место (включаемый элемент Use Case) помещается определенный набор обязанностей системы. Далее остальные части системы могут агрегировать в себя эти обязанности (при необходимости).

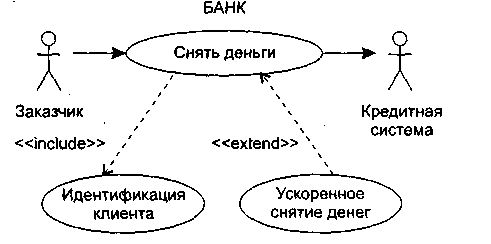


**Рис. 12.31.** Отношение включения между элементами Use Case

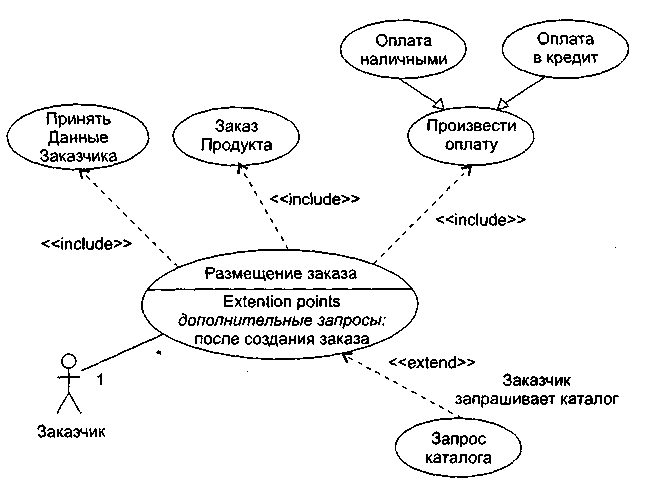
Отношение расширения (рис. 12.32) между элементами Use Case означает, что базовый элемент Use Case *неявно* включает поведение другого элемента Use Case в точке, которая определяется косвенно расширяющим элементом Use Case. Базовый элемент Use Case может быть автономен, но при определенных условиях его поведение может расширяться поведением из другого элемента Use Case. Базовый элемент Use Case может расширяться только в определенных точках — точках расширения. Отношение расширения применяется для моделирования выбираемого поведения системы. Таким способом можно отделить обязательное поведение от необязательного поведения. Например, можно использовать отношение расширения для отдельного подпотока, который выполняется только при определенных условиях, находящихся вне поля зрения базового элемента Use Case. Наконец, можно моделировать отдельные потоки, вставка которых в определенную точку управляется актером.



**Рис. 12.32.** Отношение расширения между элементами Use Case



**Рис. 12.33.** Простейшая диаграмма Use Case для банка



**Рис. 12.34.** Диаграмма Use Case для обслуживания заказчика

Пример простейшей диаграммы Use Case, в которой использованы отношения включения и расширения, приведен на рис. 12.33.

Как показано на рис. 12.34, внутри элемента Use Case может быть дополнительная секция с заголовком Extention points. В этой области перечисляются точки расширения. В указанную здесь точку *дополнительные запросы* вставляется последовательность действий от расширяющего элемента Use Case Запрос каталога. Для справки отмечено, что точка расширения размещена после действий, обеспечивающих создание заказа. На этом же рисунке отображены отношения наследования между элементами Use Case. Видно, что элементы Use Case Оплата наличными и Оплата в кредит наследуют поведение элемента Use Case Произвести оплату и являются его специализациями.

### Работа с элементами Use Case

Элемент Use Case описывает, *что* должна делать система, но не определяет, *как* она должна это делать. При моделировании это позволяет отделять внешнее представление системы от ее внутреннего представления.

Поведение элемента Use Case описывается потоком событий. Начальное описание выполняется в текстовой форме, прозрачной для пользователя системы. В потоке событий выделяют:

* основной поток и альтернативные потоки поведения;
* как и когда стартует и заканчивается элемент Use Case;
* когда элемент Use Case взаимодействует с актерами;
* какими данными обмениваются актер и система.

Для уточнения и формализации потоков событий используют диаграммы последовательности. Обычно одна диаграмма последовательности определяет главный поток в элементе Use Case, а другие диаграммы — потоки исключений.

В общем случае один элемент Use Case описывает набор последовательностей, в котором каждая последовательность представляет возможный поток событий. Каждая последовательность называется сценарием. Сценарий — конкретная последовательность действий, которая иллюстрирует поведение. Сценарии являются для элемента Use Case почти тем же, чем являются экземпляры для класса. Говорят, что сценарий — это экземпляр элемента Use Case.

## Спецификация элементов Use Case

Спецификация элемента Use Case — основной источник информации для выполнения анализа и проектирования системы. Очень важно, чтобы содержание спецификации было представлено в полной и конструктивной форме. В общем случае спецификация включает главный поток, подпотоки и альтернативные потоки поведения. В качестве шаблона спецификации представим описание элемента Use Case *«Покупать авиабилет»* для модели информационной системы авиакассы.

**Предусловие:** перед началом этого элемента Use Case должен быть выполнен элемент Use Case «Заполнить базу данных авиарейсов».

### Главный поток

Этот элемент Use Case начинается, когда покупатель регистрируется в системе и вводит свой пароль. Система проверяет, правилен ли пароль (Е-1), и предлагает покупателю выбрать одно из действий: СОЗДАТЬ, УДАЛИТЬ, ПРОВЕРИТЬ, ВЫПОЛНИТЬ, ВЫХОД.

1. Если выбрано действие СОЗДАТЬ, выполняется подпоток S-1: создать заказ авиабилета.

2. Если выбрано действие УДАЛИТЬ, выполняется подпоток S-2: удалить заказ авиабилета.

3. Если выбрано действие ПРОВЕРИТЬ, выполняется подпоток S-3: проверить заказ авиабилета.

4. Если выбрано действие ВЫПОЛНИТЬ, выполняется подпоток S-4: реализовать заказ авиабилета.

5. Если выбрано действие ВЫХОД, элемент Use Case заканчивается.

### Подпотоки

**S-1: создать заказ авиабилета.** Система отображает диалоговое окно, содержащее поля для пункта назначения и даты полета. Покупатель вводит пункт назначения и дату полета (Е-2). Система отображает параметры авиарейсов (Е-3). Покупатель выбирает авиарейс. Система связывает покупателя с выбранным авиарейсом (Е-4). Возврат к началу элемента Use Case.

**S-2: удалить заказ авиабилета.** Система отображает параметры заказа. Покупатель подтверждает решение о ликвидации заказа (Е-5). Система удаляет связь с покупателем (Е-6). Возврат к началу элемента Use Case.

**S-3: проверить заказ авиабилета.** Система выводит (Е-7) и отображает параметры заказа авиабилета: номер рейса, пункт назначения, дата, время, место, цену. Когда покупатель указывает, что он закончил проверку, выполняется возврат к началу элемента Use Case.

**S-4: реализовать заказ авиабилета.** Система запрашивает параметры кредитной карты покупателя. Покупатель вводит параметры своей кредитной карты (Е-8). Возврат к началу элемента Use Case.

### Альтернативные потоки

**Е-1: введен неправильный ID-номер покупателя.** Покупатель может повторить ввод ID-номера или прекратить элемент Use Case.

**Е-2: введены неправильные пункт назначения/дата полета.** Покупатель может повторить ввод пункта назначения/даты полета или прекратить элемент Use Case.

**Е-3: нет подходящих авиарейсов.** Покупатель информируется, что в данное время такой полет невозможен. Возврат к началу элемента Use Case.

**Е-4: не может быть создана связь между покупателем и авиарейсом.** Информация сохраняется, система создаст эту связь позже. Элемент Use Case продолжается.

**Е-5: введен неправильный номер заказа.** Покупатель может повторить ввод правильного номера заказа или прекратить элемент Use Case.

**Е-6: не может быть удалена связь между покупателем и авиарейсом.** Информация сохраняется, система будет удалять эту связь позже. Элемент Use Case продолжается.

**Е-7: система не может вывести информацию заказа.** Возврат к началу элемента Use Case.

**Е-8: некорректные параметры кредитной карты.** Покупатель может повторить ввод параметров карты или прекратить элемент Use Case.

Таким образом, в данной спецификации зафиксировано, что внутри элемента Use Case находится один основной поток и двенадцать вспомогательных потоков действий. В дальнейшем разработчик может принять решение о выделении из этого элемента Use Case самостоятельных элементов Use Case. Очевидно, что если самостоятельный элемент Use Case содержит подпоток, то его следует подключать к базовому элементу Use Case отношением include. В свою очередь, самостоятельный элемент Use Case с альтернативным потоком подключается к базовому элементу Use Case отношением extend.

## Пример диаграммы Use Case

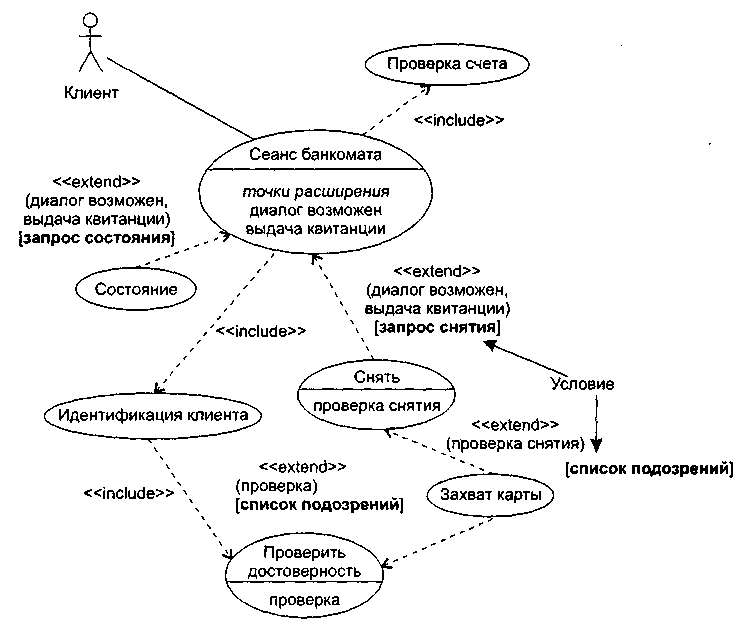
Наибольшие трудности при построении диаграмм Use Case вызывает применение отношений включения и расширения. Очень важно разобраться в отличительных особенностях этих отношений, специфике взаимодействия элементов Use Case, соединяемых с их помощью.

Пример диаграммы Use Case, в которой использованы отношения включения и расширения, приведен на рис. 12.35.

В этой диаграмме один базовый элемент Use Case Сеанс банкомата, который взаимодействует с актером Клиент. К базовому элементу Use Case подключены два расширяющих элемента Use Case (Состояние, Снять) и два включаемых элемента Use Case (Идентификация клиента, Проверка счета). В свою очередь, к элементу Use Case Идентификация клиента подключен включаемый элемент Use Case Проверить достоверность, а к элементу Use Case Снять — расширяющий элемент Use Case Захват карты (он же расширяет элемент Use Case Проверить достоверность).

Видим, что элемент Use Case Сеанс банкомата имеет две точки расширения (диалог возможен, выдача квитанции), а элементы Use Case Снять и Проверить достоверность — по одной точке расширения (проверка снятия и проверка соответственно). В точки расширения возможна вставка поведения из расширяющего элемента Use Case. Вставка происходит, если выполняется условие расширения:

* для расширяющего элемента Use Case Состояние — это условие запрос состояния;
* для расширяющего элемента Use Case Снять — это условие запрос снятия;
* для расширяющего элемента Use Case Захват карты — это условие список подозрений.



**Рис. 12.35.** Использование включения и расширения

Для расширяемого (базового) элемента Use Case эти условия являются внешними, то есть формируемыми вне его. Иными словами, элементу Use Case Сеанс банкомата ничего не известно об условиях запрос состояния и запрос снятия, а элементам Use Case Снять и Проверить достоверность — об условии список подозрений. Условия расширения являются следствиями событий, происходящих во внешней среде.

Стрелки расширения в диаграмме подписаны. Помимо стереотипа, здесь указаны:

* в круглых скобках — имена точек расширения;
* в квадратных скобках — условие расширения.

Описание расширяющего элемента Use Case разделено на сегменты, каждый сегмент обслуживает одну точку расширения базового элемента Use Case.

Количество сегментов расширяющего элемента Use Case равно количеству точек расширения базового элемента Use Case. Первый сегмент расширяющего элемента Use Case начинается с условия расширения, условие записывается только один раз, его действие распространяется и на все остальные сегменты.

Поведение базового элемента Use Case задается внутренним потоком событий, вплоть до точки расширения. В точке расширения возможно выполнение расширяющего элемента Use Case, после чего возобновляется работа внутреннего потока.

Спецификации элементов Use Case рассматриваемой диаграммы имеют следующий вид:

*Элемент Use Case* Сеанс банкомата

|  |  |
| --- | --- |
| include (Идентификация клиента)  include (Проверка счета)  (диалог возможен)  напечатать заголовок квитанции  (выдача квитанции)  конец сеанса | //включение  //включение  //первая точка расширения  //вторая точка расширения |

*Расширяющий элемент Use Case* Состояние

|  |  |
| --- | --- |
| сегмент | //начало первого сегмента |
| принять запрос состояния | //условие расширения |
| отобразить информацию о состоянии счета |  |
| сегмент | //вторая точка расширения |
| конец сеанса |  |

*Расширяющий элемент Use Case* Снять

|  |  |
| --- | --- |
| сегмент | //начало первого сегмента |
| принять запрос снятия | //условие расширения |
| определить сумму |  |
| (проверка снятия) | //точка расширения |
| сегмент | //начало второго сегмента |
| напечатать снимаемую сумму |  |
| выдать наличные деньги |  |

*Расширяющий элемент Use Case* Захват карты

|  |  |
| --- | --- |
| сегмент  принять список подозрений  проглотить карту  конец сеанса | //начало единственного сегмента  //условие расширения |

*Включаемый элемент Use Case* Идентификация клиента

|  |  |
| --- | --- |
| получить имя клиента  include (Проверить достоверность)  получить номер счета клиента | //включение |

*Включаемый элемент Use Case* Проверка счета

|  |
| --- |
| установить соединение с базойданных счетов  получить состояние и ограничения счета |

*Включаемый элемент Use Case* Проверить достоверность

|  |
| --- |
| установить соединение с базой данных клиентов  получить параметры клиента  (проверка) //точка расширения |

Опишем возможное поведение модели, задаваемое этой диаграммой.

Актер Клиент инициирует действия базового элемента Use Case Сеанс банкомата. На первом шаге запускается включаемый элемент Use Case Идентификация клиента. Этот элемент Use Case получает имя клиента и запускает элемент Use Case Проверить достоверность, в результате чего устанавливается соединение с базой данных клиентов и получаются параметры клиента.

Если к этому моменту исполняется условие расширения список подозрений, то «срабатывает» расширяющий элемент Use Case Захват карты, карта арестовывается и работа системы прекращается.

В противном случае происходит возврат к элементу Use Case Идентификация клиента, который получает номер счета клиента и возвращает управление базовому элементу Use Case.

Базовый элемент Use Case переходит ко второму шагу работы — вызывает включаемый элемент Use Case Проверка счета, который устанавливает соединение с базой данных счетов и получает состояние и ограничения счета.

Управление опять возвращается к базовому элементу Use Case. Базовый элемент Use Case переходит к первой точке расширения диалог возможен. В этой точке возможно подключение одного из двух расширяющих элементов Use Case.

Положим, что к этому моменту выполняется условие расширения запрос состояния, поэтому запускается первый сегмент элемента Use Case Состояние. В результате отображается информация о состоянии счета и управление передается базовому элементу Use Case. В базовом элементе Use Case печатается заголовок квитанции и обеспечивается переход ко второй точке расширения выдача квитанции.

Поскольку в активном состоянии продолжает находиться расширяющий элемент Use Case Состояние, запускается его второй сегмент — в квитанции печатается информация о состоянии счета.

В последний раз управление возвращается к базовому элементу Use Case — завершается сеанс работы банкомата.

## Построение модели требований

Напомним, что основное назначение диаграмм Use Case — определение требований заказчика к будущему программному приложению. Обсудим разработку ПО для машины утилизации, которая принимает использованные бутылки, банки, ящики. Для определения элементов Use Case, которые должны выполняться в системе, вначале определяют актеров.

**Выбор актеров**

Поиск актеров — большая работа. Сначала выделяют первичных актеров, использующих систему по прямому назначению. Каждый из первичных актеров участвует в выполнении одной или нескольких главных задач системы. В нашем примере первичным актером является Потребитель. Потребитель кладет в машину бутылки, получает квитанцию от машины.

Кроме первичных, существуют и вторичные актеры. Они наблюдают и обслуживают систему. Вторичные актеры существуют только для того, чтобы первичные актеры могли использовать систему. В нашем примере вторичным актером является Оператор. Оператор обслуживает машину и получает дневные отчеты о ее работе. Мы не будем нуждаться в операторе, если не будет потребителей.

Таким образом, внешняя среда машины утилизации имеет вид, представленный на рис. 12.36.



**Рис. 12.36.** Внешняя среда машины утилизации

Деление актеров на первичных и вторичных облегчает выбор системной архитектуры в терминах основного функционального назначения. Системную структуру определяют в основном первичные актеры. Именно от них в систему приходят главные изменения. Поэтому полное выделение первичных актеров гарантирует, что архитектура системы будет настроена на большинство важных пользователей.

**Определение элементов Use Case**

После выбора внешней среды можно выявить внутренние функциональные возможности системы. Для этого определяются элементы Use Case.

Каждый элемент Use Case задает некоторый путь использования системы, выполнение некоторой части функциональных возможностей. Полная совокупность элементов Use Case определяет все существующие пути использования системы.

Элемент Use Case — это последовательность взаимодействий в диалоге, выполняемом актером и системой. Запускается элемент Use Case актером, поэтому удобно выявлять элементы Use Case с помощью актеров.

Рассматривая каждого актера, мы решаем, какие элементы Use Case он может выполнять. Для этого изучается описание системы (с точки зрения актера) или проводится обсуждение с теми, кто будет действовать как актер.

Перейдем к примеру. Потребитель — первичный актер, поэтому начнем с этой роли. Этот актер должен выполнять возврат утилизируемых элементов. Так формируется элемент Use Case Возврат элемента. Приведем его текстовое описание:

Начинается, когда потребитель начинает возвращать банки, бутылки, ящики. Для каждого элемента, помещенного в машину утилизации, система увеличивает количество элементов, принятых от Потребителя, и общее количество элементов этого типа за день.

После сдачи всех элементов Потребитель нажимает кнопку квитанции, чтобы получить квитанцию, на которой напечатаны названия возвращенных элементов и общая сумма возврата.

Следующий актер — Оператор. Он получает дневной отчет об элементах, сданных за день. Это образует элемент Use Case Создание дневного отчета. Его описание:

Начинается оператором, когда он хочет получить информацию об элементах, сданных за день.

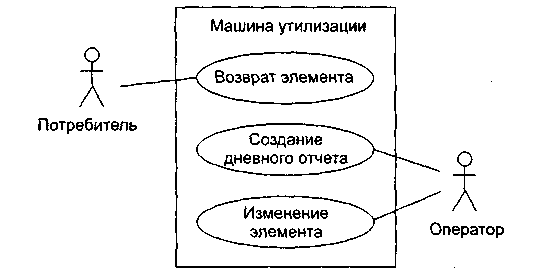
Система печатает количество элементов каждого типа и общее количество элементов, полученных за день.

Доя подготовки к созданию нового дневного отчета сбрасывается в ноль параметр Общее количество.

Кроме того, актер Оператор может изменять параметры сдаваемых элементов. Назовем соответствующий элемент Use Case Изменение элемента. Его описание:

Могут изменяться цена и размер каждого возвращаемого элемента. Могут добавляться новые типы элементов.

После выявления всех элементов диаграмма Use Case для системы принимает вид, показанный на рис. 12.37.



**Рис. 12.37.** Диаграмма Use Case для машины утилизации

Чаще всего полные описания элементов Use Case формируются за несколько итераций. На каждом шаге в описание вводятся дополнительные детали. Например, окончательное описание Возврата элемента может иметь следующий вид:

Когда потребитель возвращает сдаваемый элемент, элемент измеряется системой. Измерения позволяют определить тип элемента. Если тип допустим, то увеличивается количество элементов этого типа, принятых от Потребителя, и общее количество элементов этого типа за день.

Если тип недопустим, то на панели машины высвечивается «недействительно».

Когда Потребитель нажимает кнопку квитанции, принтер печатает дату. Производятся вычисления. По каждому типу принятых элементов печатается информация: название, принятое количество, цена, итого для типа. В конце печатается сумма, которую должен получить потребитель.

Не всегда очевидно, как распределить функциональные возможности по отдельным элементам Use Case и что является вариантом одного и того же элемента Use Case. Основной критерий выбора — сложность элемента Use Case. При анализе вариантов поведения рассматривают их различия. Если различия малы, варианты встраивают в один элемент Use Case. Если различия велики, то варианты описываются как отдельные элементы Use Case.

Обычно элемент Use Case задает одну основную и несколько альтернативных последовательностей событий.

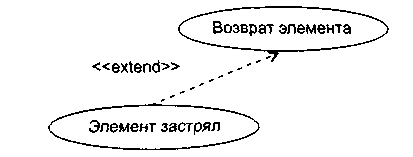
Каждый элемент Use Case выделяет частный аспект функциональных возможностей системы. Поэтому элементы Use Case обеспечивают инкрементную схему анализа функций системы. Можно независимо разрабатывать элементы Use Case для разных функциональных областей, а позднее соединить их вместе (для формирования полной модели требований).

**Вывод**: на основе элементов Use Case в каждый момент времени можно концентрировать внимание на одной частной проблеме, что позволяет вести параллельную разработку.

Расширение функциональных возможностей

Для добавления в элемент Use Case новых действий удобно применять отношение расширения. С его помощью базовый элемент Use Case может быть расширен новым элементом Use Case.

В нашем примере поведение системы не определено для случая, когда сдаваемый элемент застрял. Введем элемент Use Case Элемент Застрял, который будет расширять базовый элемент Use Case Возврат Элемента (рис. 12.38).



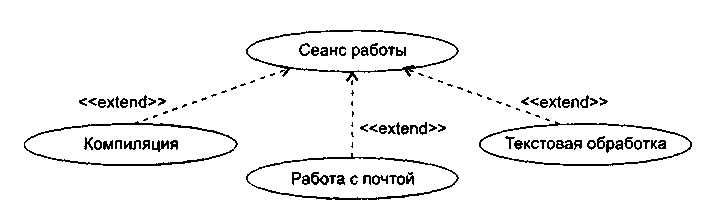
**Рис 12.38.** Расширение элемента Use Case возврат элемента

Описание элемента Use Case Элемент застрял может иметь следующий вид:

Если элемент застрял, для вызова Оператора вырабатывается сигнал тревоги. После удаления застрявшего элемента Оператор сбрасывает сигнал тревоги. В результате Потребитель может продолжить сдачу элементов. Величина ИТОГО сохраняет правильное значение. Цена застрявшего элемента не засчитывается.

Таким образом, описание базового элемента остается прежним, простым. Еще один пример приведен на рис. 12.39.

Здесь мы видим только один базовый элемент Use Case Сеанс работы. Все остальные элементы Use Case могут добавляться как расширения. Базовый элемент Use Case при этом остается почти без изменений.



**Рис. 12.39.** Применение отношения расширения

Отношение расширения определяет прерывание базового элемента Use Case, которое происходит для вставки другого элемента Use Case. Базовый элемент Use Case не знает, будет выполняться прерывание или нет. Вычисление условий прерывания находится вне компетенции базового элемента Use Case.

В расширяющем элементе Use Case указывается ссылка на то место базового элемента Use Case, куда он будет вставляться (при прерывании). После выполнения расширяющего элемента Use Case продолжается выполнение базового элемента Use Case.

Обычно расширения используют:

* для моделирования вариантных частей элементов Use Case;
* для моделирования сложных и редко выполняемых альтернативных последовательностей;
* для моделирования подчиненных последовательностей, которые выполняются только в определенных случаях;
* для моделирования систем с выбором на основе меню.

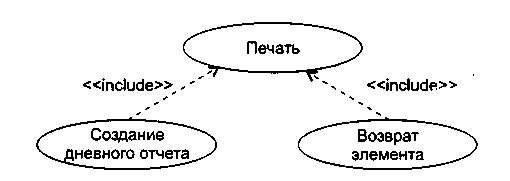
Главное, что следует помнить: решение о выборе, подключении варианта на основе расширения принимается вне базового элемента Use Case. Если же вы вводите в базовый элемент Use Case условную конструкцию, конструкцию выбора, то придется применять отношение включения. Это случай, когда «штурвал управления» находится в руках базового элемента Use Case.

**Уточнение модели требований**

Уточнение модели сводится к выявлению одинаковых частей в элементах Use Case и извлечению этих частей. Любые изменения в такой части, выделенной в отдельный элемент Use Case, будут автоматически влиять на все элементы Use Case, которые используют ее совместно.

Извлеченные элементы Use Case называют абстрактными. Они не могут быть конкретизированы сами по себе, применяются для описания одинаковых частей в других, конкретных элементах Use Case. Таким образом, описания абстрактных элементов Use Case используются в описаниях конкретных элементов Use Case. Говорят, что конкретный элемент Use Case находится в отношении «включает» с абстрактным элементом Use Case.

Вернемся к нашему примеру. В этом примере два конкретных элемента Use Case Возврат элемента и Создание дневного отчета имеют общую часть — действия, обеспечивающие печать квитанции. Поэтому, как показано на рис. 12.40, можно выделить абстрактный элемент Use Case Печать. Этот элемент Use Case будет специализироваться на выполнении распечаток.

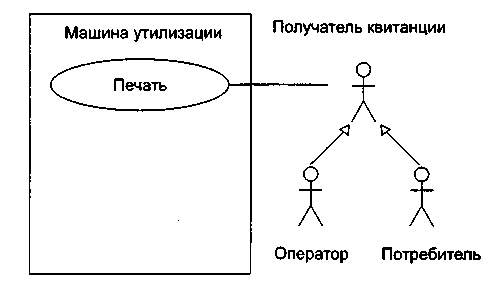


**Рис. 12.40.** Применение отношения включения

В свою очередь, абстрактные элементы Use Case могут использоваться другими абстрактными элементами Use Case. Так образуется иерархия. При построении иерархии абстрактных элементов Use Case руководствуются правилом: выделение элементов Use Case прекращается при достижении уровня отдельных операций над объектами.

Выделение абстрактных элементов Use Case можно упростить с помощью абстрактных актеров.

Абстрактный актер — это общий фрагмент роли в нескольких конкретных актерах. Абстрактный актер выражает подобия в элементах Use Case. Конкретные актеры находятся в отношении наследования с абстрактным актером. Так, в машине утилизации конкретные актеры имеют одно общее поведение: они могут получать квитанцию. Поэтому можно определить одного абстрактного актера — Получателя квитанции. Как показано на рис. 12.41, наследниками этого актера являются Потребитель и Оператор.



**Рис. 12.41.** Выделение абстрактного актера

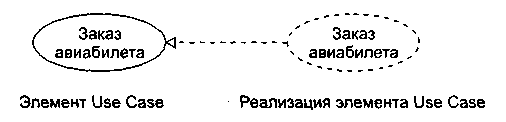
**Выводы:**

1. Абстрактные элементы Use Case находят извлечением общих последовательностей из различных элементов Use Case.
2. Отношение «включает» применяется, если несколько элементов Use Case имеют общее поведение. Цель: устранить повторения, ликвидировать избыточность.
3. Кроме того, это отношение часто используют для ограничения сложности большого элемента Use Case.
4. Отношение «расширяет» применяется, когда описывается вариация, дополняющая нормальное поведение.

## Кооперации и паттерны

Кооперации (сотрудничества) являются средством представления комплексных решений в разработке ПО на высшем, архитектурном уровне. С одной стороны, *^* кооперации обеспечивают компактность цельной спецификации программного продукта, с другой стороны — несут в себе реализации потоков управления и данных, а также структур данных.

В терминологии фирмы Rational (вдохновителя и организатора побед языка UML) кооперации называют реализациями элементов Use Case, да и обозначения их весьма схожи (рис. 12.42).



**Рис. 12.42.** Элемент Use Case и его реализация

Обратите внимание на то, что и связаны эти элементы отношением реализации: кооперация реализует конкретный элемент Use Case.

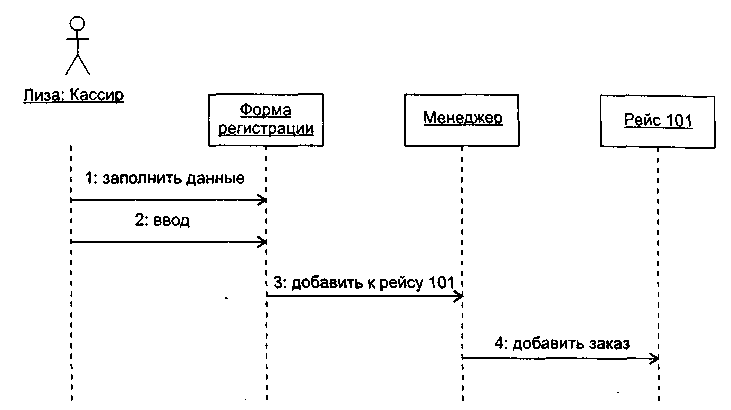
Кооперации содержат две составляющие — статическую (структурную) и динамическую (поведенческую).

Статическая составляющая кооперации задает структуру совместно работающих классов и других элементов (интерфейсов, компонентов, узлов). Чаще всего для этого используют одну или несколько диаграмм классов. Динамическая составляющая кооперации определяет поведение совместно работающих элементов. Обычно для определения применяют одну или несколько диаграмм последовательности.

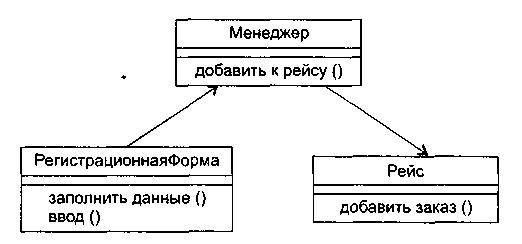
Таким образом, если заглянуть под «обложку» кооперации, мы увидим набор разнообразных диаграмм. Например, требования к информационной системе авиакассы задаются множеством элементов Use Case, каждый из которых реализуется отдельной кооперацией. Все эти кооперации применяют одни и те же классы, но все же имеют разную функциональную организацию. В частности, поведение кооперации для заказа авиабилета может описываться диаграммой последовательности, показанной на рис. 12.43.

Соответственно, структура кооперации для заказа авиабилета может иметь вид, представленный на рис. 12.44.

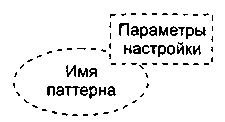
Важно понимать, что кооперации отражают понятийный аспект архитектуры системы. Один и тот же элемент может участвовать в различных кооперациях. Ведь речь здесь идет не о владении элементом, а только о его применении.



**Рис. 12.43.** Динамическая составляющая кооперации Заказ авиабилета



**Рис. 12.44.** Статическая составляющая кооперации Заказ авиабилета



**Рис. 12.45.** Обозначение паттерна

Параметризованные, то есть настраиваемые кооперации называют паттернами (образцами). Паттерн является решением типичной проблемы в определенном контексте. Обозначение паттерна имеет вид, представленный на рис. 12.45.

На место параметров настройки паттерна подставляются различные фактические параметры, в результате создаются разные кооперации.

Паттерны рассматриваются как крупные строительные блоки. Их использование приводит к существенному сокращению затрат на анализ и проектирование ПО. повышению качества и правильности разработки на логическом уровне, ведь паттерны создаются опытными профессионалами и отражают проверенные и оптимизированные решения [26], [31], [68].

Итак, паттерны — это наборы готовых решений, рецепты, предлагающие к повторному использованию самое ценное для разработчика — сплав мирового опыта по созданию ПО.

## Бизнес-модели

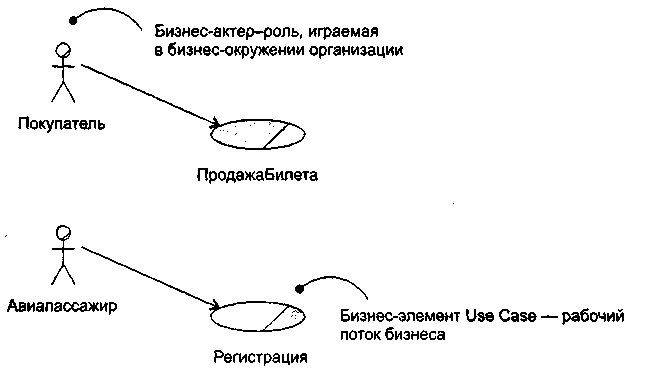
Достаточно часто перед тем, как решиться на заказ ПО, организация проводит бизнес-моделирование. Цели бизнес-моделирования:

* отобразить структуру и процессы деятельности организации;
* обеспечить ясное, комплексное и, главное, одинаковое понимание нужд организации как сотрудниками, так и будущими разработчиками ПО;
* сформировать реальные требования к программному обеспечению деятельности организации.

Для достижения этих целей разрабатываются две модели: Q бизнес-модель Use Case; а бизнес-объектная модель.

Бизнес-модель Use Case задает внешнее представление бизнес-процессов организации (с точки зрения внешней среды — клиентов и партнеров).

Как показано на рис. 12.57, бизнес-модель Use Case строится с помощью бизнес-актеров и бизнес-элементов Use Case — простого расширения средств, используемых в обычных диаграммах Use Case.



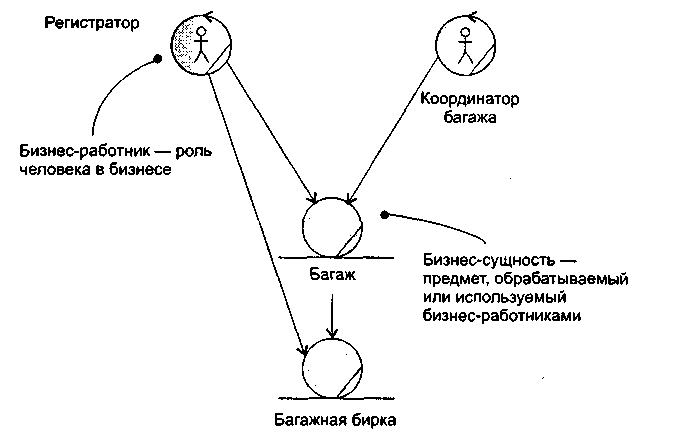
**Рис. 12.57.** Фрагмент бизнес-модели Use Case для аэропорта

Бизнес-актеры определяют внешние сущности и людей, с которыми взаимодействует бизнес. Бизнес-актер представляет собой человека, но информационная система, взаимодействующая с бизнесом, также может играть роль такого актера.

Бизнес-элементы Use Case изображают различные рабочие потоки бизнеса. Последовательности действий в бизнес-элементах Use Case обычно описываются диаграммами деятельности.

Бизнес-объектная модель отражает внутреннее представление бизнес-процессов организации (с точки зрения ее сотрудников).

Как показано на рис. 12.58, бизнес-объектная модель строится с помощью бизнес-работников и бизнес-сущностей — классов со специальными стереотипами. Эти классы имеют специальные графические обозначения.



**Рис. 12.58.** Фрагмент бизнес-объектной модели аэропорта

Бизнес-работник — абстракция человека, действующего в бизнесе. Бизнес-сущности являются «предметами», обрабатываемыми или используемыми бизнес-работниками по мере выполнения бизнес-элемента Use Case. Например, бизнес-сущность представляет собой документ или существенную часть продукта. Фактически бизнес-объектная модель отображается с помощью диаграмм классов.

## Контрольные вопросы

1. Поясните два подхода к моделированию поведения системы. Объясните достоинства и недостатки каждого из этих подходов.
2. Охарактеризуйте вершины и дуги диаграммы схем состояний. В чем состоит назначение этой диаграммы?
3. Как отображаются действия в состояниях диаграммы схем состояний?
4. Как показываются условные переходы между состояниями?
5. Как задаются вложенные состояния в диаграммах схем состояний?
6. Поясните понятие исторического подсостояния.
7. Охарактеризуйте средства и возможности диаграммы деятельности.
8. Когда не следует применять диаграмму деятельности?
9. Какие средства диаграммы деятельности позволяют отобразить параллельные действия?
10. Зачем в диаграмму деятельности введены плавательные дорожки?
11. Как представляется имя объекта в диаграмме сотрудничества?
12. Поясните синтаксис представления свойства в диаграмме сотрудничества.
13. Какие стереотипы видимости используются в диаграмме сотрудничества? Поясните их смысл.
14. В какой форме записываются сообщения в языке UML? Поясните смысл сообщения.
15. В каком отношении находятся сообщения и действия? Перечислите разновидности действий.
16. Чем отличается процедурный поток от асинхронного потока сообщений?
17. Как указывается повторение сообщений?
18. Как показать ветвление сообщений?
19. Что общего в диаграмме последовательности и диаграмме сотрудничества? Чем они отличаются друг от друга?
20. Как отображается порядок передачи сообщений в диаграмме последовательности?
21. Когда удобнее применять диаграммы последовательности?
22. Из каких элементов состоит диаграмма Use Case?
23. Какие отношения разрешены между элементами диаграммы Use Case?
24. Для чего применяют диаграммы Use Case?
25. Чем отличаются друг от друга отношения включения и расширения с точки зрения управления?
26. Каково назначение спецификации элемента Use Case и как она оформляется?
27. Что такое сценарий элемента Use Case?
28. Как документируется отношение включения?
29. Как документируется отношение расширения?
30. Каков порядок построения модели требований?
31. Каково назначение кооперации? Какие составляющие ее образуют?
32. Могут ли разные кооперации использовать одинаковые классы? Обоснуйте ответ.
33. Что такое паттерн?
34. Чем паттерн отличается от кооперации? Чем они схожи?
35. Как описывается паттерн?
36. Что нужно сделать для применения паттерна?
37. Каковы цели бизнес-моделирования?
38. Из каких частей состоит бизнес-модель? На что похожи эти части? В чем их своеобразие?

# 10. Модели реализации объектно-ориентированных программных систем

Статические и динамические модели описывают логическую организацию системы, отражают логический мир программного приложения. Модели реализации обеспечивают представление системы в физическом мире, рассматривая вопросы упаковки логических элементов в компоненты и размещения компонентов в аппаратных узлах [8], [23], [53], [67].

## Компонентные диаграммы

Компонентная диаграмма — первая из двух разновидностей диаграмм реализации, моделирующих физические аспекты объектно-ориентированных систем. Компонентная диаграмма показывает организацию набора компонентов и зависимости между компонентами.

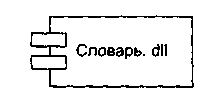
Элементами компонентных диаграмм являются компоненты и интерфейсы, а также отношения зависимости и реализации. Как и другие диаграммы, компонентные диаграммы могут включать примечания и ограничения. Кроме того, компонентные диаграммы могут содержать пакеты или подсистемы, используемые для группировки элементов модели в крупные фрагменты.

### Компоненты

По своей сути компонент является физическим фрагментом реализации системы, который заключает в себе программный код (исходный, двоичный, исполняемый), сценарные описания или наборы команд операционной системз (имеются в виду командные файлы). Язык UML дает следующее определение.

*Компонент —* физическая и заменяемая часть системы, которая соответствует набору интерфейсов и обеспечивает реализацию этого набора интерфейсов.

Интерфейс — очень важная часть понятия «компонент», его мы обсудим в следующем подразделе. Графически компонент изображается как прямоугольник с вкладками, обычно включающий имя (рис. 13.1).



**Рис. 13.1.** Обозначение компонента

Компонент — базисный строительный блок физического представления ПО, поэтому интересно сравнить его с базисным строительным блоком логического представления ПО — классом.

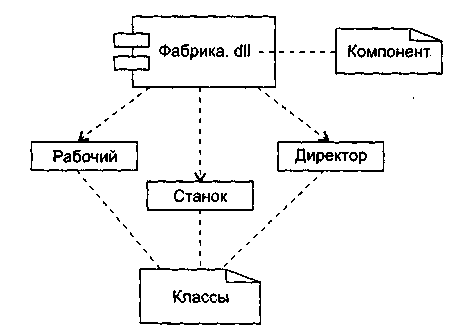
Сходные характеристики компонента и класса:

* наличие имени;
* реализация набора интерфейсов;
* участие в отношениях зависимости;
* возможность быть вложенным;
* наличие экземпляров (экземпляры компонентов можно использовать только в диаграммах размещения).

Вы скажете — много общего. И тем не менее между компонентами и классами есть существенная разница, ее характеризует табл. 13.1.

**Таблица 13.1.** Различия компонентов и классов

|  |  |
| --- | --- |
| **№** | **Описание** |
| 1  2  3 | Классы — логические абстракции, компоненты — физические предметы, которые живут в мире битов. В частности, компоненты могут «жить» в физических узлах, а классы лишены такой возможности  Компоненты являются физическими упаковками, контейнерами, инкапсулирующими в себе различные логические элементы. Они — элементы абстракций другого уровня  Классы имеют свойства и операции. Компоненты имеют только операции, которые доступны через их интерфейсы |



**Рис.** 13.2. Классы в компоненте

О чем говорят эти различия? Во-первых, класс не может «дышать» воздухом физического мира реализации. Ему нужен скафандр. Таким скафандром является компонент.

Во-вторых, им не жить друг без друга — пустые скафандры никому не нужны. Причем в скафандре-компоненте может находиться несколько классов и коопераций. Итак, в скафандре — физической реализации — располагается набор логики. Как показано на рис. 13.2, с помощью отношения зависимости можно явно отобразить отношение между компонентом и классами, которые он реализует. Правда, чаще всего такие отношения не отображаются. Их удобно представлять в компонентной спецификации.

В-третьих, класс — душа нараспашку (он может даже показать свои свойства). Компонент всегда застегнут на все пуговицы (правда, из него торчат интерфейсные разъемы операций).

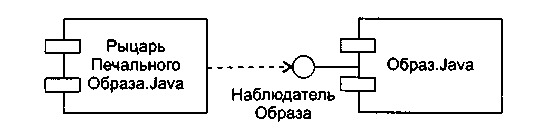
Теперь уместно перейти к обсуждению интерфейсов.

### Интерфейсы

Интерфейс — список операций, которые определяют услуги класса или компонента. Образно говоря, интерфейс — это разъем, который торчит из ящичка компонента. С помощью интерфейсных разъемов компоненты стыкуются друг с другом, объединяясь в систему.

Еще одна аналогия. Интерфейс подобен абстрактному классу, у которого отсутствуют свойства и работающие операции, а есть только абстрактные операции (не имеющие тел). Если хотите, интерфейс похож на улыбку чеширского кота из правдивой истории об Алисе, где кот отдельно и улыбка отдельно. Все операции интерфейса открыты и видимы клиенту (в противном случае они потеряли бы всякий смысл). Итак, операции интерфейса только именуют предлагаемые услуги, не более того.

Очень важна взаимосвязь между компонентом и интерфейсом. Возможны два способа отображения взаимосвязи между компонентом и его интерфейсами. В первом, свернутом способе, как показано на рис. 13.3, интерфейс изображается в форме пиктограммы. Компонент Образ.java, который реализует интерфейс, соединяется со значком интерфейса (кружком) НаблюдательОбраза простой линией. Компонент РыцарьПечальногоОбраза.jауа, который использует интерфейс, связан с ним отношением зависимости.



**Рис. 13.3.** Представление интерфейса в форме пиктограммы

Второй способ представления интерфейса иллюстрирует рис. 13.4. Здесь используется развернутая форма изображения интерфейса, в которой могут показываться его операции. Компонент, который реализует интерфейс, подключается к нему отношением реализации. Компонент, который получает доступ к услугам другого компонента через интерфейс, по-прежнему подключается к интерфейсу отношением зависимости.



**Рис. 13.4.** Развернутая форма представления интерфейса

По способу связи компонента с интерфейсом различают:

* экспортируемый интерфейс — тот, который компонент реализует и предлагает как услугу клиентам;
* импортируемый интерфейс — тот, который компонент использует как услугу другого компонента.

У одного компонента может быть несколько экспортируемых и несколько импортируемых интерфейсов.

Тот факт, что между двумя компонентами всегда находится интерфейс, устраняет их прямую зависимость. Компонент, использующий интерфейс, будет функционировать правильно вне зависимости от того, какой компонент реализует этот интерфейс. Это очень важно и обеспечивает гибкую замену компонентов в интересах развития системы.

### Компоновка системы

За последние полвека разработчики аппаратуры прошли путь от компьютеров размером с комнату до крошечных «ноутбуков», обеспечивших возросшие функциональные возможности. За те же полвека разработчики программного обеспечения прошли путь от больших систем на Ассемблере и Фортране до еще больших систем на C++ и Java. Увы, но программный инструментарий развивается медленнее, чем аппаратный инструментарий. В чем главный секрет аппаратчиков? — спросят у аппаратчика-мальчиша программеры-буржуины.

Этот секрет — компоненты. Разработчик аппаратуры создает систему из готовых аппаратных компонентов (микросхем), выполняющих определенные функции и предоставляющих набор услуг через ясные интерфейсы. Задача конструкторов упрощается за счет повторного использования результатов, полученных другими.

Повторное использование — магистральный путь развития программного инструментария. Создание нового ПО из существующих, работоспособных программных компонентов приводит к более надежному и дешевому коду. При этом сроки разработки существенно сокращаются.

Основная цель программных компонентов — допускать сборку системы из двоичных заменяемых частей. Они должны обеспечить начальное создание системы из компонентов, а затем и ее развитие — добавление новых компонентов и замену некоторых старых компонентов без перестройки системы в целом. Ключ к воплощению такой возможности — интерфейсы. После того как интерфейс определен, к выполняемой системе можно подключить любой компонент, который удовлетворяет ему или обеспечивает этот интерфейс. Для расширения системы производятся компоненты, которые обеспечивают дополнительные услуги через новые интерфейсы. Такой подход основывается на особенностях компонента, перечисленных в табл. 13.2.

**Таблица 13.2.** Особенности компонента

|  |
| --- |
| Компонент физичен. Он живет в мире битов, а не логических понятий и не зависит от языка программирования |
| Компонент — заменяемый элемент. Свойство заменяемости позволяет заменить один компонент другим компонентом, который удовлетворяет тем же интерфейсам. Механизм замены оговорен современными компонентными моделями (COM, COM+, CORBA, Java Beans), требующими незначительных преобразований или предоставляющими утилиты, которые автоматизируют механизм |
| Компонент является частью системы, он редко автономен. Чаще компонент сотрудничает с другими компонентами и существует в архитектурной или технологической среде, предназначенной для его использования. Компонент связан и физически, и логически, он обозначает фрагмент большой системы |
| Компонент соответствует набору интерфейсов и обеспечивает реализацию этого набора интерфейсов |

**Вывод**: компоненты — базисные строительные блоки, из которых может проектироваться и составляться система. Компонент может появляться на различных уровнях иерархии представления сложной системы. Система на одном уровне абстракции может стать простым компонентом на более высоком уровне абстракции.

### Разновидности компонентов

Мир современных компонентов достаточно широк и разнообразен. В языке UML для обозначения новых разновидностей компонентов используют механизм стереотипов. Стандартные стереотипы, предусмотренные в UML для компонентов, представлены в табл. 13.3.

**Таблица 13.3.** Разновидности компонентов

|  |  |
| --- | --- |
| **Стереотип** | **Описание** |
| «executable»  «library»  «file»  «table»  «document» | Компонент, который может выполняться в физическом узле (имеет расширение .ехе)  Статическая или динамическая объектная библиотека (имеет расширение .dll)  Компонент, который представляет файл, содержащий исходный код или данные (имеет расширение .ini)  Компонент, который представляет таблицу базы данных (имеет расширение .tbl)  Компонент, который представляет документ (имеет расширение .hip) |

В языке UML не определены пиктограммы для перечисленных стереотипов, применяемые на практике пиктограммы компонентов показаны на рис. 13.5-13.9.



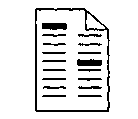
**Рис. 13.5.** Пиктограмма исполняемого **Рис. 13.6.** Пиктограмма объектной

элемента библиотеки



**Рис. 13.7.** Пиктограмма документа **Рис. 13.8.** Пиктограмма таблицы

с исходным кодом или данными базы данных



**Рис. 13.9.** Пиктограмма документа

## Использование компонентных диаграмм

Компонентные диаграммы используют для моделирования статического представления реализации системы. Это представление поддерживает управление конфигурацией системы, составляемой из компонентов. Подразумевается, что для получения работающей системы существуют различные способы сборки компонентов.

Компонентные диаграммы показывают отношения:

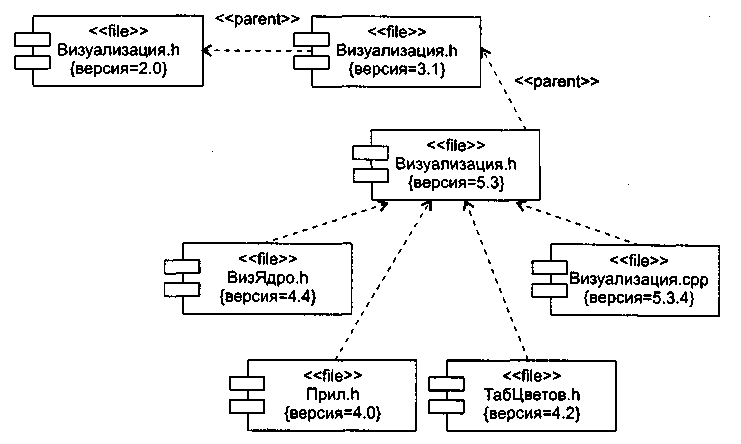
* периода компиляции (среди текстовых компонентов);
* периода сборки, линковки (среди объектных двоичных компонентов);
* периода выполнения (среди машинных компонентов).

Рассмотрим типовые варианты применения компонентных диаграмм.

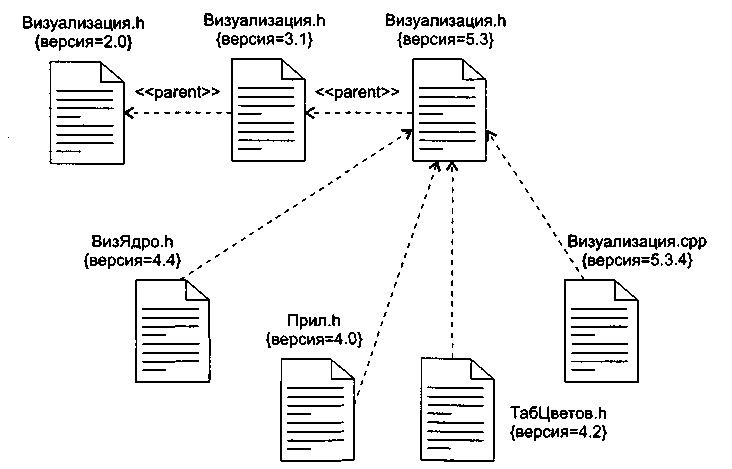
### Моделирование программного текста системы

При разработке сложных систем программный текст (исходный код) разбросан по многим файлам исходного кода. При использовании Java исходный код сохраняется в .java-файлах, при использовании C++ — в заголовочных файлах (.h-фай-лах) и телах (.срр-файлах), при использовании Ada 95 — в спецификациях (.ads-файлах) и реализациях (.adb-файлах).

Между файлами существуют многочисленные зависимости компиляции. Если к этому добавить, что по мере разработки рождаются новые версии файлов, то становится очевидной необходимость управления конфигурацией системы, визуализации компиляционных зависимостей.



**Рис. 13.10.** Моделирование исходного кода



**Рис. 13.11.** Моделирование исходного кода с использованием пиктограмм

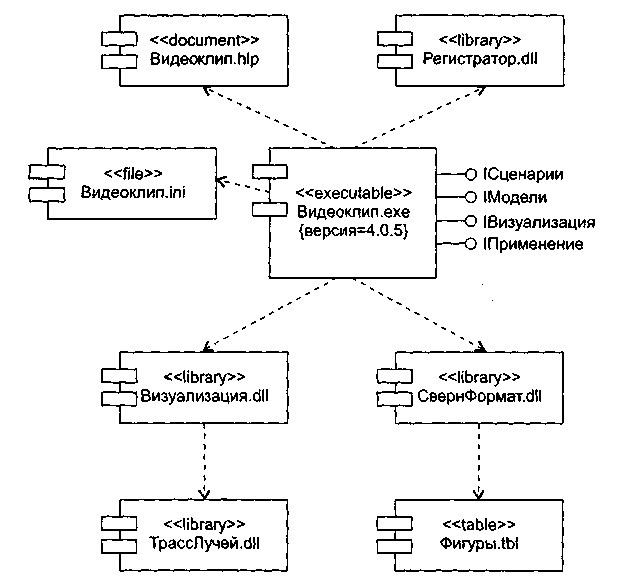
В качестве примера на рис. 13.10 приведена компонентная диаграмма, где изображены файлы исходного кода, используемые для построения библиотеки Визуализация.dll. Имеются четыре заголовочных файла (Визуализация.h, ВизЯдро.h, Прил.h, ТабЦветов.h), которые представляют исходный код для спецификации определенных классов. Файл реализации здесь один (Визуализация.срр), он является реализацией одного из заголовков. Отметим, что для каждого файла явно указана его версия, причем для файла Визуализация.h показаны три версии и история их появления. На рис. 13.11 повторяется та же диаграмма, но здесь для обозначения компонентов использованы специальные пиктограммы.

### Моделирование реализации системы

Реализация системы может включать большое количество разнообразных компонентов:

* исполняемых элементов;
* динамических библиотек;
* файлов данных;
* справочных документов;
* файлов инициализации;
* файлов регистрации;
* сценариев;
* файлов установки.

Моделирование этих компонентов, отношений между ними — важная часть управления конфигурацией системы.

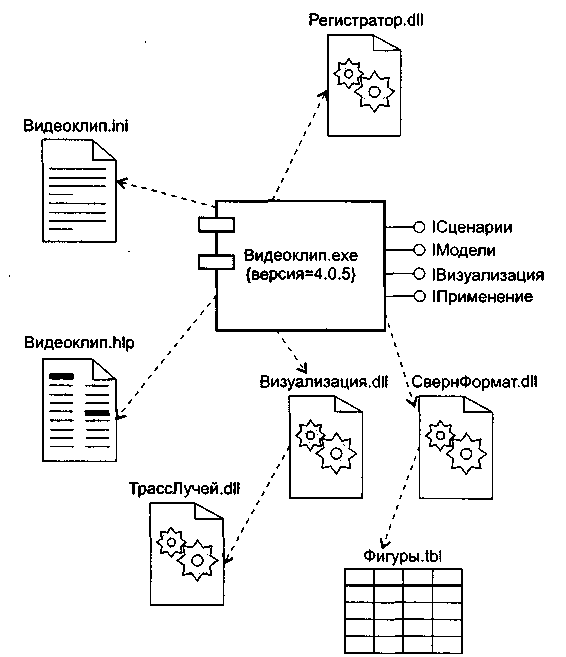


**Рис. 13.12.** Моделирование реализации системы

Например, на рис. 13.12 показана часть реализации системы, группируемая вокруг исполняемого элемента Видеоклип.ехе. Здесь изображены четыре библиотеки (Регистратор.dll, СвернФормат.dll, Визуализация.dll, ТрассЛучей.dll), один документ (Видеоклип.hlp), один простой файл (Видеоклип.ini),атакже таблица базы данных (Фигуры.tbl). Вдиаграмме указаны отношения зависимости, существующие между компонентами.

Для исполняемого компонента Видеоклип.ехе указан номер версии (с помощью пгеговой величины), представлены его экспортируемые интерфейсы (IСценарии, IВизуализация, IМодели, IПрименение). Эти интерфейсы образуют API компонента «интерфейс прикладного программирования).

На рис. 13.13 повторяется та же диаграмма, моделирующая реализацию, но здесь для обозначения компонентов использованы специальные пиктограммы.



**Рис. 13.13.** Моделирование реализации с использованием пиктограмм

# 11. Унифицированный процесс разработки объектно-ориентированных ПС

В первой главе рассматривались основы организации процессов разработки ПО. В данной главе внимание сосредоточено на детальном обсуждении унифицированного процесса разработки объектно-ориентированного ПО, на базе которого возможно построение самых разнообразных схем конструирования программных приложений. Далее описывается содержание ХР-процесса экстремальной разработки, являющегося носителем адаптивной технологии, применяемой в условиях частого изменения требований заказчика.

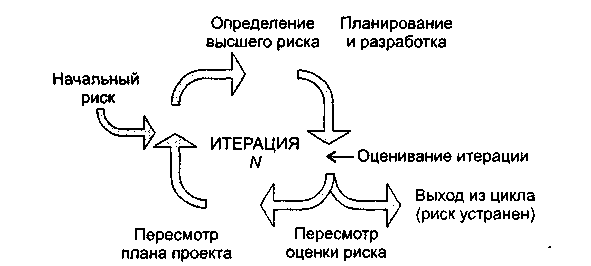
## Эволюционно-инкрементная организация жизненного цикла разработки

Рассматриваемый подход является развитием спиральной модели Боэма [8], [40], [44], [57]. В этом случае процесс разработки программной системы организуется в виде эволюционно-инкрементного жизненного цикла. Эволюционная составляющая цикла основывается на доопределении требований в ходе работы, инкрементная составляющая — на планомерном приращении реализации требований.

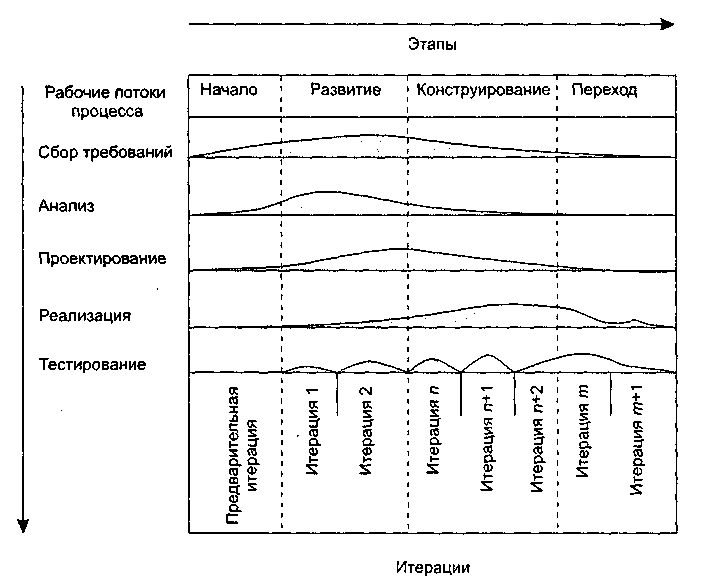
В этом цикле разработка представляется как серия итераций, результаты которых развиваются от начального макета до конечной системы. Каждая итерация включает сбор требований, анализ, проектирование, реализацию и тестирование. Предполагается, что вначале известны не все требования, их дополнение и изменение осуществляется на всех итерациях жизненного цикла. Структура типовой итерации показана на рис. 15.1.

Видно, что критерием управления этим жизненным циклом является уменьшение риска. Работа начинается с оценки начального риска. В ходе выполнения каждой итерации риск пересматривается. Риск связывается с каждой итерацией так, что ее успешное завершение уменьшает риск. План последовательности реализаций гарантирует, что наибольший риск устраняется в первую очередь.

Такая методика построения системы нацелена на выявление и уменьшение риска в самом начале жизненного цикла. В итоге минимизируются затраты на уменьшение риска.



**Рис. 15.1.** Типовая итерация эволюционно-инкрементного жизненного цикла



**Рис. 15.2.** Два измерения унифицированного процесса разработки

Как показано на рис. 15.2, в структуре унифицированного процесса разработки выделяют два измерения:

* горизонтальная ось представляет время и демонстрирует характеристики жизненного цикла процесса;
* вертикальная ось представляет рабочие потоки процесса, которые являются логическими группировками действий.

Первое измерение задает динамический аспект развития процесса в терминах циклов, этапов, итераций и контрольных вех. Второе измерение задает статический аспект процесса в терминах компонентов процесса, рабочих потоков, приводящих к выработке искусственных объектов (артефактов), и участников.

### Этапы и итерации

По времени в жизненном цикле процесса выделяют четыре этапа:

* начало (Inception) — спецификация представления продукта;
* развитие (Elaboration) — планирование необходимых действий и требуемых ресурсов;
* конструирование (Construction) — построение программного продукта в виде серии инкрементных итераций;
* переход (Transition) — внедрение программного продукта в среду пользователя (промышленное производство, доставка и применение).

В свою очередь, каждый этап процесса разделяется на итерации. Итерация — это полный цикл разработки, вырабатывающий промежуточный продукт. По мере перехода от итерации к итерации промежуточный продукт инкрементно усложняется, постепенно превращаясь в конечную систему. В состав каждой итерации входят все рабочие потоки — от сбора требований до тестирования. От итерации к итерации меняется лишь удельный вес каждого рабочего потока — он зависит от этапа. На этапе Начало основное внимание уделяется сбору требований, на этапе Развитие — анализу и проектированию, на этапе Конструирование — реализации, на этапе Переход — тестированию. Каждый этап и итерация уменьшают некоторый риск и завершается контрольной вехой. К вехе привязывается техническая проверка степени достижения ключевых целей. По результатам проверки возможна модификация дальнейших действий.

### Рабочие потоки процесса

Рабочие потоки процесса имеют следующее содержание:

* Сбор требований — описание того, что система должна делать;
* Анализ — преобразование требований к системе в классы и объекты, выявляемые в предметной области;
* Проектирование — создание статического и динамического представления системы, выполняющего выявленные требования и являющегося эскизом реализации;
* Реализация — производство программного кода, который превращается в исполняемую систему;
* Тестирование — проверка всей системы в целом.

Каждый рабочий поток определяет набор связанных артефактов и действий. Артефакт — это документ, отчет или выполняемый элемент. Артефакт может вырабатываться, обрабатываться или потребляться. Действие описывает задачи — шаги обдумывания, шаги исполнения и шаги проверки. Шаги выполняются участниками процесса (для создания или модификации артефактов).

Между артефактами потоков существуют зависимости. Например, модель Use Case, генерируемая в ходе сбора требований, уточняется моделью анализа из процесса анализа, обеспечивается проектной моделью из процесса проектирования, реализуется моделью реализации из процесса реализации и проверяется тестовой моделью из процесса тестирования.

### Модели

Модель — наиболее важная разновидность артефакта. Модель упрощает реальность, создается для лучшего понимания разрабатываемой системы. Предусмотрены девять моделей, вместе они покрывают все решения по визуализации, спецификации, конструированию и документированию программных систем:

* бизнес-модель. Определяет абстракцию организации, для которой создается система;
* модель области определения. Фиксирует контекстное окружение системы;
* модель Use Case. Определяет функциональные требования к системе;
* модель анализа. Интерпретирует требования к системе в терминах проектной модели;
* проектная модель. Определяет словарь проблемы и ее решение;
* модель размещения. Определяет аппаратную топологию, в которой исполняется система;
* модель реализации. Определяет части, которые используются для сборки и реализации физической системы;
* тестовая модель. Определяет тестовые варианты для проверки системы;
* модель процессов. Определяет параллелизм в системе и механизмы синхронизации.

### Технические артефакты

Технические артефакты подразделяются на четыре основных набора:

* набор требований. Описывает, что должна делать система;
* набор проектирования. Описывает, как должна быть сконструирована система;
* набор реализации. Описывает сборку разработанных программных компонентов;
* набор размещения. Обеспечивает всю информацию о поставляемой конфигурации.

Набор требований группирует всю информацию о том, что система должна делать. Он может включать модель Use Case, модель нефункциональных требований, модель области определения, модель анализа, а также другие формы выражения нужд пользователя.

Набор проектирования группирует всю информацию о том, как будет конструироваться система при учете всех ограничений (времени, бюджета, традиций, повторного использования, качества и т.д.).

Он может включать проектную модель, тестовую модель и другие формы выражения сущности системы (например, макеты).

Набор реализации группирует все данные о программных элементах, образующих систему (программный код, файлы конфигурации, файлы данных, программные компоненты, информацию о сборке системы).

Набор размещения группирует всю информацию об упаковке, отправке, установке и запуске системы.

## Управление риском

Словарь русского языка С. И. Ожегова и Н. Ю. Шведовой определяет риск как *«возможность опасности, неудачи».* Влияние риска вычисляют по выражению

RE = *P*(UO) x *L*(UO),

где:

* RE — показатель риска (Risk Exposure — подверженность риску);
* *P*(UO) — вероятность неудовлетворительного результата (Unsatisfactory Outcome);
* *L*(UO) — потеря при неудовлетворительном результате.

При разработке программного продукта неудовлетворительным результатом может быть: превышение бюджета, низкая надежность, неправильное функционирование и т. д. Управление риском включает шесть действий:

1. Идентификация риска — выявление элементов риска в проекте.
2. Анализ риска — оценка вероятности и величины потери по каждому элементу риска.
3. Ранжирование риска — упорядочение элементов риска по степени их влияния.
4. Планирование управления риском — подготовка к работе с каждым элементом риска.
5. Разрешение риска — устранение или разрешение элементов риска.
6. Наблюдение риска — отслеживание динамики элементов риска, выполнениекорректирующих действий.

Первые три действия относят к этапу оценивания риска, последние три действия — к этапу контроля риска [20].

### Идентификация риска

В результате идентификации формируется список элементов риска, специфичных для данного проекта.

Выделяют три категории источников риска: проектный риск, технический риск, коммерческий риск.

Источниками проектного риска являются:

* выбор бюджета, плана, человеческих ресурсов программного проекта;
* формирование требований к программному продукту;
* сложность, размер и структура программного проекта;
* методика взаимодействия с заказчиком.

К источникам технического риска относят:

* трудности проектирования, реализации, формирования интерфейса, тестирования и сопровождения;
* неточность спецификаций;
* техническая неопределенность или отсталость принятого решения.

Главная причина технического риска — реальная сложность проблем выше предполагаемой сложности.

Источники коммерческого риска включают:

* создание продукта, не требующегося на рынке;
* создание продукта, опережающего требования рынка (отстающего от них);
* потерю финансирования.

Лучший способ идентификации — использование проверочных списков риска, которые помогают выявить возможный риск. Например, проверочный список десяти главных элементов программного риска может иметь представленный ниже вид.

1. Дефицит персонала.

2. Нереальные расписание и бюджет.

3. Разработка неправильных функций и характеристик.

4. Разработка неправильного пользовательского интерфейса.

5. Слишком дорогое обрамление.

6. Интенсивный поток изменения требований.

7. Дефицит поставляемых компонентов.

8. Недостатки в задачах, разрабатываемых смежниками.

9. Дефицит производительности при работе в реальном времени.

10. Деформирование научных возможностей.

На практике каждый элемент списка снабжается комментарием — набором методик для предотвращения источника риска.

После идентификации элементов риска следует количественно оценить их влияние на программный проект, решить вопросы о возможных потерях. Эти вопросы решаются на шаге анализа риска.

### Анализ риска

В ходе анализа оценивается вероятность возникновения *Рi* и величина потери *Li* для каждого выявленного *i*-го элемента риска. В результате вычисляется влияние *REi i*-го элемента риска на проект.

Вероятности определяются с помощью экспертных оценок или на основе статистики, накопленной за предыдущие разработки. Итоги анализа, как показано в табл. 15.1, сводятся в таблицу.

**Таблица 15.1.** Оценка влияния элементов риска

|  |  |  |  |
| --- | --- | --- | --- |
| **Элемент риска** | **Вероятность, %** | **Потери** | **Влияние риска** |
| 1. Критическая программная ошибка | 3-5 | 10 | 30-50 |
| 2. Ошибка потери ключевых данных | 3-5 | 8 | 24-40 |
| 3. Отказоустойчивость недопустимо снижает производительность | 4-8 | 7 | 28-56 |
| 4. Отслеживание опасного условия как безопасного | 5 | 9 | 45 |
| 5. Отслеживание безопасного условия как опасного | 5 | 3 | 15 |
| 6. Аппаратные задержки срывают планирование | 6 | 4 | 24 |
| 7. Ошибки преобразования данных приводят к избыточным вычислениям | 8 | 1 | 8 |
| 8. Слабый интерфейс пользователя снижает эффективность работы | 6 | 5 | 30 |
| 9. Дефицит процессорной памяти | 1 | 7 | 7 |
| 10. СУБД теряет данные | 2 | 2 | 4 |

### Ранжирование риска

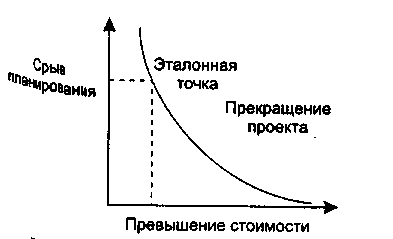
Ранжирование заключается в назначении каждому элементу риска приоритета, который пропорционален влиянию элемента на проект. Это позволяет выделить категории элементов риска и определить наиболее важные из них. Например, представленные в табл. 15.1 элементы риска упорядочены по их приоритету.

Для больших проектов количество элементов риска может быть очень велико (30-40 элементов). В этом случае управление риском затруднено. Поэтому к элементам риска применяют принцип Парето 80/20. Опыт показывает, что 80% всего проектного риска приходятся на долю 20% от общего количества элементов риска. В ходе ранжирования определяют эти 20% элементов риска (их называют существенными элементами). В дальнейшем учитывается влияние только существенных элементов риска.

### Планирование управления риском

Цель планирования — сформировать набор функций управления каждым элементом риска. Введем необходимые определения.

В планировании используют понятие эталонного уровня риска. Обычно выбирают три эталонных уровня риска: превышение стоимости, срыв планирования, упадок производительности. Они могут быть причиной прекращения проекта. Если комбинация проблем, создающих риск, станет причиной превышения любого из этих уровней, работа будет остановлена. В фазовом пространстве риска эталонному уровню риска соответствует эталонная точка. В эталонной точке решения « продолжать проект» и «прекратить проект» имеют одинаковую силу. На рис. 15.3 показана кривая останова, составленная из эталонных точек.



**Рис. 15.3.** Кривая останова проекта

Ниже кривой располагается рабочая область проекта, выше кривой — запретная область (при попадании в эту область проект должен быть прекращен).

Реально эталонный уровень редко представляется как кривая, чаще это сфера, в которой есть области неопределенности (в них принять решение невозможно).

Теперь рассмотрим последовательность шагов планирования.

1. Исходными данными для планирования является набор четверок *[Ri Pi, Li, REi],* где *Ri* — 2-й элемент риска, *Pi* — вероятность *i*-го элемента риска, *Li* — потеря по *i*-му элементу риска, *REi* — влияние *i*-го элемента риска.
2. Определяются эталонные уровни риска в проекте.
3. Разрабатываются зависимости между каждой четверкой *[Ri Pi, Li, REi]* и каждым эталонным уровнем.
4. Формируется набор эталонных точек, образующих сферу останова. В сфере останова предсказываются области неопределенности.
5. Для каждого элемента риска разрабатывается план управления. Предложения плана составляются в виде ответов на вопросы «зачем, что, когда, кто, где, как и сколько».
6. План управления каждым элементом риска интегрируется в общий план программного проекта.

### Разрешение и наблюдение риска

Основанием для разрешения и наблюдения является план управления риском. Работы по разрешению и наблюдению производятся с начала и до конца процесса разработки.

Разрешение риска состоит в плановом применении действий по уменьшению риска.

Наблюдение риска гарантирует:

* цикличность процесса слежения за риском;
* вызов необходимых корректирующих воздействий.

Для управления риском используется эффективная методика *«Отслеживание 10 верхних элементов риска».* Эта методика концентрирует внимание на факторах повышенного риска, экономит много времени, минимизирует «сюрпризы» разработки.

Рассмотрим шаги методики *«Отслеживания 10 верхних элементов риска».*

1. Выполняется выделение и ранжирование наиболее существенных элементов риска в проекте.
2. Производится планирование регулярных просмотров (проверок) процесса разработки. В больших проектах (в группе больше 20 человек) просмотр должен проводиться ежемесячно, в остальных проектах — чаще.
3. Каждый просмотр начинается с обсуждения изменений в 10 верхних элементах риска (их количество может изменяться от 7 до 12). В обсуждении фиксируется текущий приоритет каждого из 10 верхних элементов риска, его приоритет в предыдущем просмотре, частота попадания элемента в список верхних элементов. Если элемент в списке опустился, он по-прежнему нуждается в наблюдении, но не требует управляющего воздействия. Если элемент поднялся в списке, или только появился в нем, то элемент требует повышенного внимания. Кроме того, в обзоре обсуждается прогресс в разрешении элемента риска (по сравнению с предыдущим просмотром).
4. Внимание участников просмотра концентрируется на любых проблемах в разрешении элементов риска.

## Этапы унифицированного процесса разработки

Обсудим назначение, цели, содержание и основные итоги каждого этапа унифицированного процесса разработки.

### Этап НАЧАЛО (Inception)

Главное назначение этапа — запустить проект.

Цели этапа НАЧАЛО:

* определить область применения проектируемой системы (ее предназначение, границы, интерфейсы с внешней средой, критерий признания — приемки);
* определить элементы Use Case, критические для системы (основные сценарии поведения, задающие ее функциональность и покрывающие главные проектные решения);
* определить общие черты архитектуры, обеспечивающей основные сценарии, создать демонстрационный макет;
* определить общую стоимость и план всего проекта и обеспечить детализированные оценки для этапа развития;
* идентифицировать основные элементы риска. Основные действия этапа НАЧАЛО:
* формулировка области применения проекта — выявление требований и ограничений, рассматриваемых как критерий признания конечного продукта;
* планирование и подготовка бизнес-варианта и альтернатив развития для управления риском, определение персонала, проектного плана, а также выявление зависимостей между стоимостью, планированием и полезностью;
* синтезирование предварительной архитектуры, развитие компромиссных решений проектирования; определение решений разработки, покупки и повторного использования, для которых можно оценить стоимость, планирование и ресурсы.

В итоге этапа НАЧАЛО создаются следующие артефакты:

* спецификация представления основных проектных требований, ключевых характеристик и главных ограничений;
* начальная модель Use Case (20% от полного представления); а начальный словарь проекта;
* начальный бизнес-вариант (содержание бизнеса, критерий успеха — прогноз дохода, прогноз рынка, финансовый прогноз);
* начальное оценивание риска;
* проектный план, в котором показаны этапы и итерации.

### Этап РАЗВИТИЕ (Elaboration)

Главное назначение этапа — создать архитектурный базис системы.

Цели этапа РАЗВИТИЕ:

* определить оставшиеся требования, функциональные требования формулировать как элементы Use Case;
* определить архитектурную платформу системы;
* отслеживать риск, устранить источники наибольшего риска;
* разработать план итераций этапа КОНСТРУИРОВАНИЕ.

Основные действия этапа РАЗВИТИЕ:

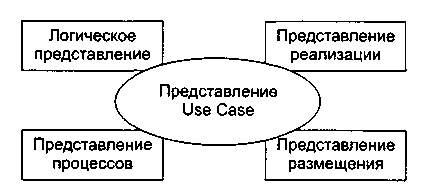
* развитие спецификации представления, полное формирование критических элементов Use Case, задающих дальнейшие решения;
* развитие архитектуры, выделение ее компонентов.

В итоге этапа РАЗВИТИЕ создаются следующие артефакты:

* модель Use Case (80% от полного представления);
* дополнительные требования (нефункциональные требования, а также другие требования, которые не связаны с конкретным элементом Use Case);
* описание программной архитектуры;
* выполняемый архитектурный макет;
* пересмотренный список элементов риска и пересмотренный бизнес-вариант;
* план разработки для всего проекта, включающий крупноблочный проектный план и показывающий итерации и критерий эволюции для каждой итерации.

Обсудим более подробно главную цель этапа РАЗВИТИЕ — создание архитектурного базиса.

Архитектура объектно-ориентированной системы многомерна — она описывается множеством параллельных представлений. Как показано на рис. 15.4, обычно используется «4+1»-представление [44].



**Рис. 15.4.** «4+1»-представление архитектуры

Представление Use Case описывает систему как множество взаимодействий с точки зрения внешних актеров. Это представление создается на этапе НАЧАЛО жизненного цикла и управляет оставшейся частью процесса разработки.

Логическое представление содержит набор пакетов, классов и отношений. Изначально создается на этапе развития и усовершенствуется на этапе конструирования.

Представление процессов создается для параллельных программных систем, содержит процессы, потоки управления, межпроцессорные коммуникации и механизмы синхронизации. Представление изначально создается на этапе развития, усовершенствуется на этапе конструирования.

Представление реализации содержит модули и подсистемы. Представление изначально создается на этапе развития и усовершенствуется на этапе конструирования.

Представление размещения содержит физические узлы системы и соединения между узлами. Создается на этапе развития.

В качестве примера рассмотрим порядок создания логического представления архитектуры. Для решения этой задачи исследуются элементы Use Case, разработанные на этапе НАЧАЛО. Рассматриваются экземпляры элементов Use Case — сценарии. Каждый сценарий преобразуется в диаграмму последовательности. Далее в диаграммах последовательности выделяются объекты. Объекты группируются в классы. Классы могут группироваться в пакеты.

Согласно взаимодействиям между объектами в диаграммах последовательности устанавливаются отношения между классами. Для обеспечения функциональности в классы добавляются свойства (они определяют их структуру) и операторы (они определяют поведение). Для размещения общей структуры и поведения создаются суперклассы.

В качестве другого примера рассмотрим разработку плана итераций для этапа КОНСТРУИРОВАНИЕ. Такой план должен задавать управляемую серию архитектурных реализаций, каждая из которых увеличивает свои функциональные возможности, а конечная — покрывает все требования к полной системе. Главным источником информации являются элементы Use Case и диаграммы последовательности. Будем называть их обобщенно — сценариями. Сценарии группируются так, чтобы обеспечивать реализацию определенной функциональности системы. Кроме того, группировки должны устранять наибольший (в данный момент) риск в проекте.

План итераций включает в себя следующие шаги:

1. Определяются все элементы риска в проекте. Устанавливаются их приоритеты.
2. Выбирается группа сценариев, которым соответствуют элемент риска с наибольшим приоритетом. Сценарии исследуются. Порядок исследования определяется не только степенью риска, но и важностью для заказчика, а также потребностью ранней разработки базовых сценариев.
3. В результате анализа сценариев формируются классы и отношения, которые их реализуют.
4. Программируются сформированные классы и отношения.
5. Разрабатываются тестовые варианты.
6. Тестируются классы и отношения. Цель — проверить выполнение функционального назначения сценария.
7. Результаты объединяются с результатами предыдущих итераций, проводится тестирование интеграции.
8. Оценивается итерация. Выделяется необходимая повторная работа. Она назначается на будущую итерацию.

### Этап КОНСТРУИРОВАНИЕ (Construction)

Главное назначение этапа — создать программный продукт, который обеспечивает начальные операционные возможности.

Цели этапа КОНСТРУИРОВАНИЕ:

* минимизировать стоимость разработки путем оптимизации ресурсов и устранения необходимости доработок;
* добиться быстрого получения приемлемого качества;
* добиться быстрого получения контрольных версий (альфа, бета и т. д.).

Основные действия этапа КОНСТРУИРОВАНИЕ:

* управление ресурсами, контроль ресурсов, оптимизация процессов;
* полная разработка компонентов и их тестирование (по сформулированному критерию эволюции);
* оценивание реализаций продукта (по критерию признания из спецификации представления).

В итоге этапа КОНСТРУИРОВАНИЕ создаются следующие артефакты:

* программный продукт, готовый для передачи в руки конечных пользователей;
* описание текущей реализации;
* руководство пользователя.

Реализации продукта создаются в серии итераций. Каждая итерация выделяет конкретный набор элементов риска, выявленных на этапе развития. Обычно в итерации реализуется один или несколько элементов Use Case. Типовая итерация включает следующие действия:

1. Идентификация реализуемых классов и отношений.
2. Определение в классах типов данных (для свойств) и сигнатур (для операций). Добавление сервисных операций, например операций доступа и управления. Добавление сервисных классов (классов-контейнеров, классов-контроллеров). Реализация отношений ассоциации, агрегации и наследования.
3. Создание текста на языке программирования.
4. Создание(обновление) документации.
5. Тестирование функций реализации продукта.
6. Объединение текущей и предыдущей реализаций. Тестирование итерации.

### Этап ПЕРЕХОД (Transition)

Главное назначение этапа — применить программный продукт в среде пользователей и завершить реализацию продукта.

Этап начинается с предъявления пользователям бета-реализации продукта. В ней обнаруживаются ошибки, они корректируются в последующих бета-реализациях. Параллельно решаются вопросы размещения, упаковки и сопровождения продукта. После завершения бета-периода тестирования продукт считается реализованным.

### Оценка качества проектирования

Качество проектирования оценивают с помощью объектно-ориентированных метрик, введенных в главе 14.

**Этап РАЗВИТИЕ**

Качество логического представления архитектуры оценивают по метрикам:

* WMC — взвешенные методы на класс;
* NOC — количество детей;
* DIT — высота дерева наследования;
* NOM — суммарное количество методов, определенных во всех классах системы;
* NC — общее количество классов в системе.

Метрики WMC, NOC вычисляются для каждого класса, кроме того, формируются их средние значения в системе. Метрики DIT, NOM, NC вычисляются для всей системы.

**Этап КОНСТРУИРОВАНИЕ**

На каждой итерации конструирования продукта вычисляются метрики:

* WMC — взвешенные методы на класс;
* NOC — количество детей;
* СВО — сцепление между классами объектов;
* RFC — отклик для класса;
* LCOM — недостаток связности в методах;
* CS — размер класса;
* NOO — количество операций, переопределяемых подклассом;
* NOA — количество операций, добавленных подклассом;
* SI — индекс специализации;
* OSavg — средний размер операции;
* NPavg — среднее количество параметров на операцию;
* NC — общее количество классов в системе;
* LOC — суммарная LOC-оценка всех методов системы;
* DIT — высота дерева наследования;
* NOM — суммарное количество методов в системе.

Метрики WMC, NOC, СВО, RFC, LCOM, CS, NOO, NOA, SI, OSAVG, NPAVG вычисляются для каждого класса, кроме того, формируются их средние значения в системе. Метрики DIT, NOM, NC, *LOCS* вычисляются для всей системы.

На последней итерации дополнительно вычисляется набор метрик MOOD, предложенный Абреу:

* МНF — фактор закрытости метода;
* AHF — фактор закрытости свойства;
* MIF — фактор наследования метода;
* AIF — фактор наследования свойства;
* POF — фактор полиморфизма;
* СОF — фактор сцепления.

## Пример объектно-ориентированной разработки

Для иллюстрации унифицированного процесса рассмотрим фрагмент разработки, выполненной автором совместно с Ольвией Комашиловой. Поставим задачу *—* разработать оконный интерфейс пользователя, который будет использоваться прикладными программами.

### Этап НАЧАЛО

Оконный интерфейс пользователя(WUI) — среда, управляемая событиями. Действия в среде инициируются функциями обратного вызова, которые вызываются в ответ на событие — пользовательский ввод. Ядром WUI является цикл обработки событий, который организуется менеджером ввода.

WUI должен обеспечивать следующие типы неперекрывающихся окон:

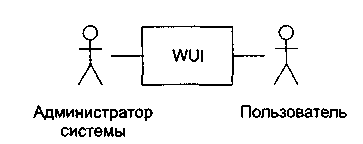
* простое окно, в которое может быть выведен текст;
* окно меню, в котором пользователь может задать вариант действий — выбор подменю или функции обратного вызова.

**Идентификация актеров**

Актерами для *WUI* являются:

* пользователь прикладной программы, использующей *WUI;*
* администратор системы, управляющий работой *WUI.*

Внешнее окружение *WUI* имеет вид, представленный на рис. 15.5.



**Рис. 15.5.** Внешнее окружение WUI

**Идентификация элементов Use Case**

В WUI могут быть выделены два элемента Use Case:

* управление окнами;
* использование окон.

Диаграмма Use Case для среды WUI представлена на рис. 15.6.



**Рис. 15.6.** Диаграмма Use Case для среды WUI

**Описания элементов Use Case**

Описание элемента Use Case Управление окнами.

Действия начинаются администратором системы. Администратор может создать, удалить или модифицировать окно.

Описание элемента Use Case Использование окон.

Действия начинаются пользователем прикладной программы. Обеспечивается возможность работы с меню и простыми окнами.

### Этап РАЗВИТИЕ

На этом этапе создаются сценарии для элементов Use Case, разрабатываются диаграммы последовательности (формализующие текстовые представления сценариев), проектируются диаграммы классов и планируется содержание следующего этапа разработки.

**Сценарии для элемента Use Case Управление окнами**

В элементе Use Case Управление окнами заданы три потока событий — три сценария.

1. Сценарий Создание окна.

Устанавливаются координаты окна на экране, стиль рамки окна. Образ окна сохраняется в памяти. Окно выводится на экран. Если создается окно меню, содержащее обращение к функции обратного вызова, то происходит установка этой функции. В конце менеджер окон добавляет окно в список управляемых окон WUI.

2. Сценарий Изменение стиля рамки.

Указывается символ, с помощью которого будет изображаться рамка. Образ окна сохраняется в памяти. Окно перерисовывается на экране.

3. Сценарий Уничтожение окна.

Менеджер окон получает указание удалить окно. Менеджер окон снимает окно с регистрации (в массиве управляемых окон WUI). Окно снимает отображение с экрана.

**Развитие описания элемента Use Case Использование окон**

Действия начинаются с ввода пользователем символа. Символ воспринимается менеджером ввода. В зависимости от значения введенного символа выполняется один из следующих вариантов:

при значении ENTER - вариант ОКОНЧАНИЯ ВВОДА;

при переключающем значении - вариант ПЕРЕКЛЮЧЕНИЯ;

при обычном значении - символ выводится в активное окно.

Вариант ОКОНЧАНИЯ ВВОДА:

при активном окне меню выбирается пункт меню. В ответ либо выполняется функция обратного вызова (закрепленная за этим пунктом меню), либо вызывается подменю (соответствующее данному пункту меню);

при активном простом окне выполняется переход на новую строку окна.

Вариант ПЕРЕКЛЮЧЕНИЯ.

При вводе переключающего символа:

ESC - активным становится окно меню;

TAB - активным становится следующее простое окно;

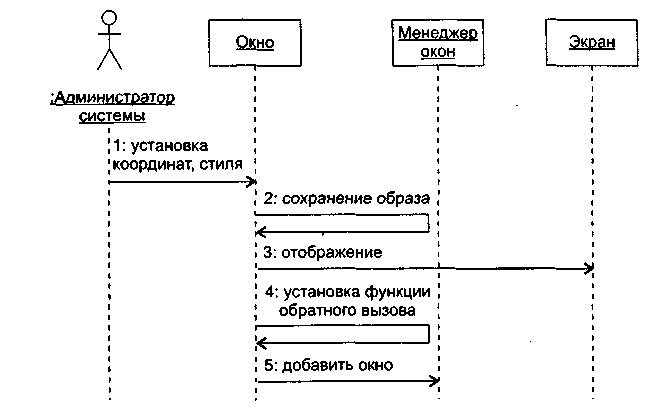
Ctrl-E - все окна закрываются и сеанс работы заканчивается.

Далее из описания элемента Use Case Использование окон выделяются два сценария: Использование простого окна и Использование окна меню.

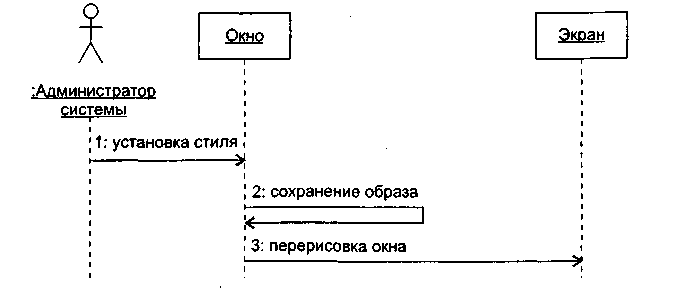
На следующем шаге сценарии элементов Use Case преобразуются в диаграммы последовательности — за счет этого достигается формализация описаний, требуемая для построения диаграмм классов. Для построения диаграмм последовательности проводится грамматический разбор каждого сценария элемента Use Case: значащие существительные превращаются в объекты, а значащие глаголы — в сообщения, пересылаемые между объектами.

**Диаграммы последовательности**

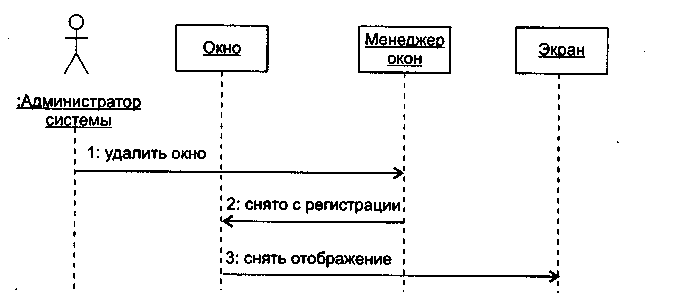
Диаграммы изображены на рис. 15.7-15.11.



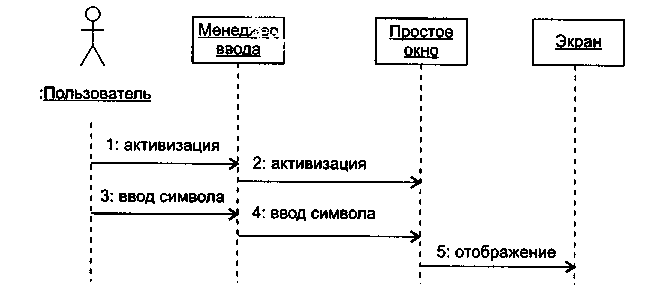
**Рис. 15.7.** Диаграмма последовательности Создание окна



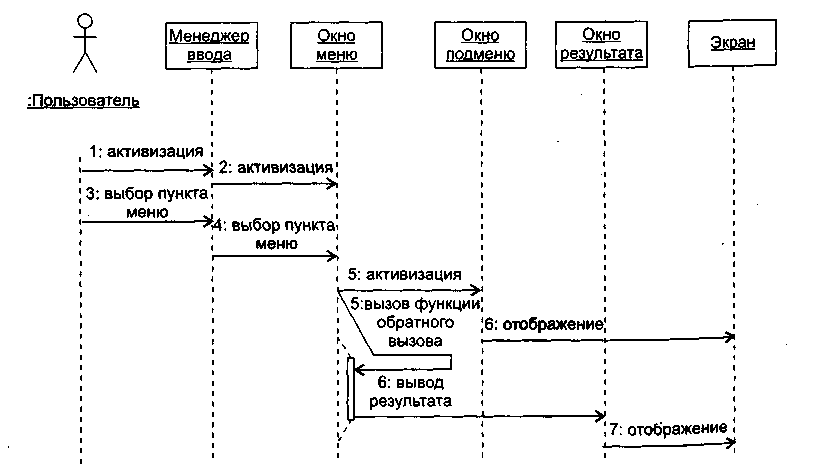
**Рис. 15.8.** Диаграмма последовательности Изменение стиля рамки



**15.9.** Диаграмма последовательности Уничтожение окна



**Рис. 15.10.** Диаграмма последовательности Использование простого окна



**Рис. 15.11.** Диаграмма последовательности Использование окна меню

**Создание классов**

Работа по созданию классов (и включению их в диаграмму классов) требует изучения содержания всех диаграмм последовательности. Проводится она в три этапа.

На первом этапе выявляются и именуются классы. Для этого просматривается каждая диаграмма последовательности. Любой объект в этой диаграмме должен принадлежать конкретному классу, для которого надо придумать имя. Например, резонно предположить, что объекту Менеджер окон должен соответствовать класс Window\_Manager, поэтому класс Window\_Manager следует ввести в диаграмму. Конечно, если в другой диаграмме последовательности опять появится подобный объект, то дополнительный класс не образуется.

На втором этапе выявляются операции классов. На диаграмме последовательности такая операция соответствует стрелке (и имени) сообщения, указывающей на линию жизни объекта класса. Например, если к линии жизни объекта Менеджер окон подходит стрелка сообщения добавить окно, то в класс Window\_Manager нужно ввести си операцию add\_to\_list().

На третьем этапе определяются отношения ассоциации между классами — они обеспечивают пересылки сообщений между соответствующими объектами.

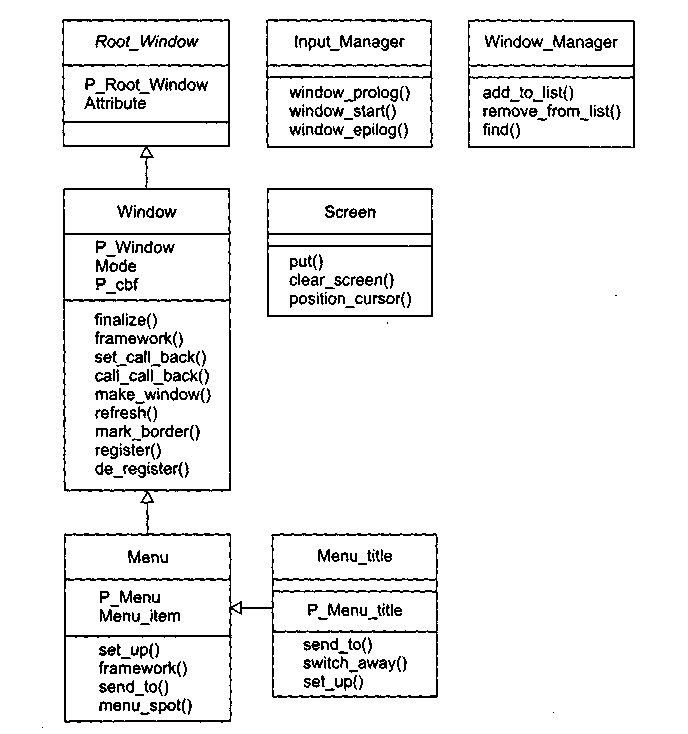
В нашем примере анализ диаграмм последовательности позволяет выделить следующие классы:

* Window — класс, объектами которого являются простые окна;
* Menu — класс, объектами которого являются окна меню. Этот класс является потомком класса Window;
* Menu\_title — класс, объектом которого является окно главного меню. Класс является потомком класса Menu;
* Screen — класс, объектом которого является экран. Этот класс обеспечивает позиционирование курсора, вывод изображения на экран дисплея, очистку экрана;
* Input\_Manager — объект этого класса управляет взаимодействием между пользователем и окнами интерфейса. Его обязанности: начальные установки среды WUI, запуск цикла обработки событий, закрытие среды WUI;
* Window\_Manager — осуществляет общее управление окнами, отображаемыми на экране. Используется менеджером ввода для получения доступа к конкретному окну.

Для оптимизации ресурсов создается абстрактный суперкласс Root\_Window. Он определяет минимальные обязанности, которые должен реализовать любой тип окна (а (посылка символа в окно, перевод окна в активное/пассивное состояние, перерисовка окна, возврат информации об окне). Все остальные классы окон являются его потомками.

Для реализации функций, определенных в сценариях, в классы добавляются свойства и операции. По результатам формирования свойств и операций классов обновляется содержание диаграмм последовательности.

Начальное представление иерархии классов WUI показано на рис. 15.12. Результаты начальной оценки качества проекта сведены в табл. 15.2.



**Рис. 15.12.** Начальная диаграмма классов WUI

Отметим, что для упрощения рисунка на этой диаграмме не показаны существующие между классами отношения ассоциации. В реальной диаграмме они обязательно отображаются — без них экземпляры классов не смогут взаимодействовать друг с другом.

**Планирование итераций конструирования**

На данном шаге составляется план итераций, который определяет порядок действий на этапе конструирования. Цель каждой итерации — уменьшить риск разработки конечного продукта. Для создания начального плана анализируются элементы Us Case, их сценарии и диаграммы последовательности. Устанавливается приоритет их реализации. При завершении каждой итерации будет повторно вычисляться риск. Оценка риска может привести к необходимости обновления плана итераций.

Положим, что максимальный риск связан с реализацией элемента Use Case Управление окнами, причем наиболее опасна разработка сценария Создание окна, среднюю опасность несет сценарий Уничтожение окна и малую опасность — Изменение стиля рамки.

В связи с этими соображениями начальный план итераций принимает вид:

**Итерация 1** — реализация сценариев элемента Use Case Управление окнами:

1. Создание окна.

2. Уничтожение окна.

3. Изменение стиля рамки.

**Итерация 2** — реализация сценариев элемента Use Case Использование окон:

4. Использование простого окна.

5. Использование окна меню.

### Этап КОНСТРУИРОВАНИЕ

Рассмотрим содержание итераций на этапе конструирования.

**Итерация 1 — реализация сценариев элемента Use Case Управление окнами**

Для реализации сценария Создание окна программируются следующие операции класса Window:

* framework — создание каркаса окна;
* register — регистрация окна;
* set\_call\_back — установка функции обратного вызова;
* make\_window — задание видимости окна.

Далее реализуются операции общего управления окнами, методы класса Window\_Manager:

* add\_to\_list — добавление нового окна в массив управляемых окон;
* find — поиск окна с заданным переключающим символом.

Программируются операции класса Input-Manager:

* window\_prolog — инициализация WUI;
* window\_start — запуск цикла обработки событий;
* window\_epilog — закрытие WUI.

В ходе реализации перечисленных операций выясняется необходимость и программируется содержание вспомогательных операций.

1. В классе Window\_Manager:

* write\_to — форматный вывод сообщения в указанное окно;
* hide\_win — удаление окна с экрана;
* switchAwayFromTop — подготовка окна к переходу в пассивное состояние;
* switch\_to\_top — подготовка окна к переходу в активное состояние;
* window\_fatal — формирование донесения об ошибке;
* top — переключение окна в активное состояние;
* send\_to\_top — посылка символа в активное окно.

2. В классе Window:

* put — три реализации для записи в окно символьной, строковой и числовой информации;
* create — создание макета окна (используется операцией framework);
* position — изменение позиции курсора в окне;
* about — возврат информации об окне;
* switch\_to — пометка активного окна;
* switch\_away — пометка пассивного окна;
* send\_to — посылка символа в окно для обработки.

Второй шаг первой итерации ориентирован на реализацию сценария Уничтожение окна. Основная операция — finalize (метод класса Window), она выполняет разрушение окна. Для ее обеспечения создаются вспомогательные операции:

* de\_register — удаление окна из массива управляемых окон;
* remove\_from\_list (метод класса Window\_Manager) — вычеркивание окна из регистра.

Для реализации сценария Изменение стиля рамки создаются операции в классе Window:

* mark\_border — построение новой рамки окна;
* refresh — перерисовка окна на экране.

В конце итерации создаются операции класса Screen:

* dear\_screen — очистка экрана;
* position\_cursor — позиционирование курсора;
* put — вывод на экран дисплея строк, символов и чисел.

**Итерация 2 — реализация сценариев элемента Use Case Использование окон**

На этой итерации реализуем методы классов Menu и Menu\_title, а также добавим необходимые вспомогательные методы в класс Window.

Отметим, что операции, обеспечивающие сценарий Использование простого окна, в основном уже реализованы (на первой итерации). Осталось запрограммировать следующие операции — методы класса Window:

* call\_call\_back — вызов функции обратного вызова;
* initialize — управляемая инициализация окна;
* clear — очистка окна с помощью пробелов;
* new\_line — перемещение на следующую строку окна.

Для обеспечения сценария Использование окна меню создаются следующие операции.

1. В классе Menu:

* framework — создание каркаса окна-меню;
* send\_to — обработка пользовательского ввода в окно-меню;
* menu\_spot — выделение выбранного элемента меню;
* set\_up — заполнение окна-меню именами элементов;
* get\_menu\_name — возврат имени выбранного элемента меню;
* get\_cur\_selected\_detaits — возврат указателя на выбранное окно и функцию обратного вызова.

2. В классе Menu\_title:

* send\_to — выделение новой строки меню или вызов функции обратного вызова;
* switch\_away — возврат в базовое окно-меню более высокого уровня;
* set\_up — установки окна меню-заголовка.

В практике проектирования достаточно типичны случаи, когда в процессе разработки меняются исходные требования или появляются дополнительные требования к продукту. Предположим, что в конце второй итерации появилось дополнительное требование — ввести в WUI новый тип окна — диалоговое окно. Диалоговое окно должно обеспечивать не только вывод, но и ввод данных, а также их обработку.

Для реализации этого требования вводится третья итерация конструирования.

**Итерация 3 — разработка диалогового окна**

*Шаг 1: Спецификация представления диалогового окна.*

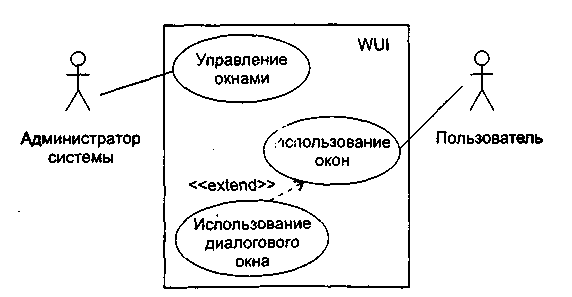
На этом шаге фиксируется представление заказчика об обязанностях диалогового окна. Положим, что оно имеет следующий вид:

1. Диалоговое окно накапливает посылаемые в него символы, отображая их по мере получения.
2. При получении символа конца сообщения (ENTER) полная строка текста принимается в функцию обратного вызова, связанную с диалоговым окном.
3. Функция обратного вызова реализует обслуживание, требуемое пользователю.
4. Функция обратного вызова обеспечивается прикладным программистом.

*Шаг 2: Модификация диаграммы Use Case для WUI.*

Очевидно, что дополнительное требование приводит к появлению дополнительного элемента Use Case, который находится в отношении «расширяет» с базовым г элементом Use Case Использование окон.

Диаграмма Use Case принимает вид, представленный на рис. 15.13.



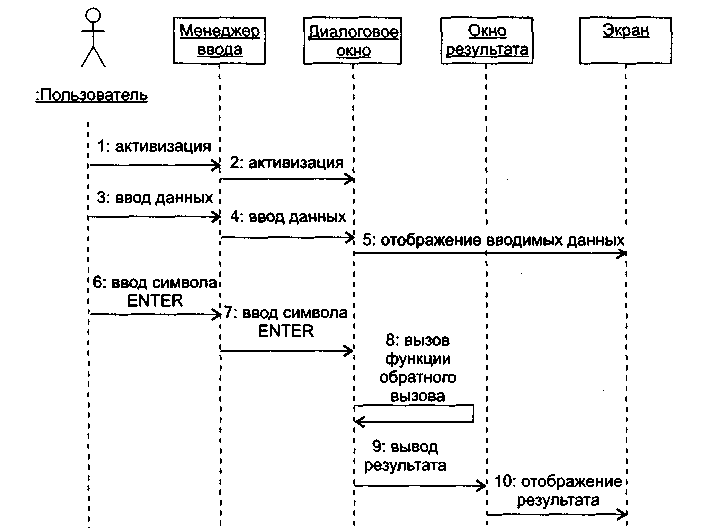
**Рис. 15.13.** Модифицированная диаграмма Use Case для WUI

*Шаг 3: Описание элемента Use Case Использование диалогового окна.*

Действия начинаются с ввода пользователем переключающего символа, активизирующего данный тип окна. Символ воспринимается менеджером ввода. Далее пользователь вводит данные, которые по мере поступления отображаются в диалоговом окне. После нажатия пользователем символа окончания ввода (ENTER) данные передаются в функцию обратного вызова как параметр. Выполняется функция обратного вызова, результат выводится в простое окно результата.

*Шаг 4: Диаграмма последовательности Использование диалогового окна.*

Диаграмма последовательности для сценария Использование диалогового окна показана на рис. 15.14.



**Рис. 15.14.** Диаграмма последовательности Использование диалогового окна

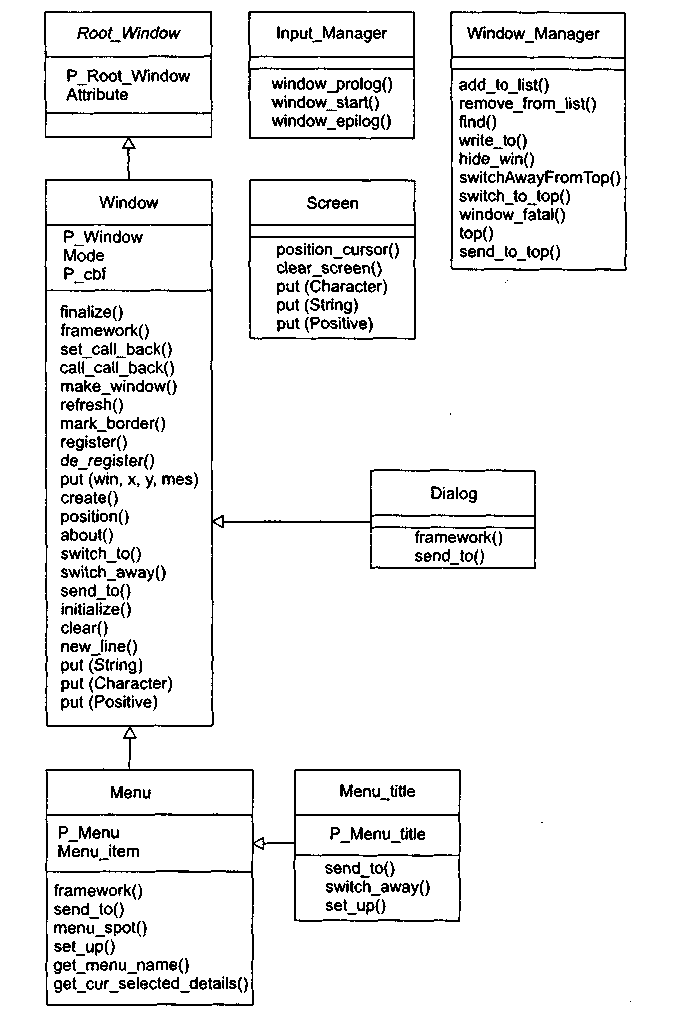
*Шаг 5: Создание класса.*

Для реализации сценария Использование диалогового окна создается новый класс Dialog, который является наследником класса Window. Объекты класса Dialog образуют диалоговые окна.

Класс Dialog переопределяет следующие операции, унаследованные от класса Window:

* framework — формирование диалогового окна. Параметры операции: имя диалогового окна, координаты, ширина окна, заголовок окна и ссылка на функцию обратного вызова. Операция создает каркас окна, устанавливает для него функцию обратного вызова, делает окно видимым и регистрирует его в массиве управляемых окон;
* send\_to — обрабатывает пользовательский ввод, посылаемый в диалоговое окно. Окно запоминает символы, вводимые пользователем, а после нажатия пользователем клавиши ENTER вызывает функцию обратного вызова, обрабатывающую эти данные.

Конечное представление иерархии классов WUI показано на рис. 15.15.



**Рис. 15.15.** Конечная диаграмма классов WUI

**10.Технология проектирования MSF**

В [1994 году](http://ru.wikipedia.org/wiki/1994_%D0%B3%D0%BE%D0%B4), стремясь достичь максимальной отдачи от [IT](http://ru.wikipedia.org/wiki/IT)-проектов, Microsoft выпустила в свет пакет руководств по эффективному проектированию, разработке, внедрению и сопровождению решений, построенных на основе своих технологий. Эти знания базируются на опыте, полученном Microsoft при работе над большими проектами по разработке и сопровождению программного обеспечения, опыте консультантов Microsoft и лучшем из того, что накопила на данный момент IT-индустрия. Всё это представлено в виде двух взаимосвязанных и хорошо дополняющих друг друга областей знаний: *Microsoft Solutions Framework* ([MSF](http://ru.wikipedia.org/wiki/MSF)) и [*Microsoft Operations Framework*](http://en.wikipedia.org/wiki/Microsoft_Operations_Framework) ([MOF](http://ru.wikipedia.org/w/index.php?title=MOF&action=edit&redlink=1)).

Следует отметить, что Microsoft разработала на базе общих методов MSF методики для прикладного и специализированного применения. Причём, Microsoft сертифицирует экспертов именно по прикладным знаниям в применении MSF (например, сертификация MCTS 74-131 по экспертизе в методике управления проектами). Перед тем, как изучать методы MSF, следует сначала определить, какой прикладной вариант MSF имеется ввиду.

Наиболее популярные прикладные варианты MSF, разработанные Microsoft:

* [методика внедрения решений в области Управления проектами](http://www.microsoftproject.ru/articles.phtml?aid=69);
* [методика управления IT-проектами на базе методологий MSF и Agile](http://www.microsoftproject.ru/articles.phtml?aid=180).

Важность прикладных вариантов MSF подчёркивает тот факт, что в «чистом варианте» саму методику MSF в своих IT-проектах компания Microsoft не использует [[1]](http://www.microsoftproject.ru/articles.phtml?aid=180). В проектах *Microsoft Consulting Services* используется гибридная методология MSF и Agile. Несмотря на внешние существенные различия прикладных вариантов MSF, разработанных экспертами Microsoft, общая база методов MSF для них остается общая и отражает общие методологические подходы к итеративному ведению проектов [[2]](http://www.microsoftproject.ru/articles.phtml?aid=158#agile).

[MOF](http://ru.wikipedia.org/w/index.php?title=MOF&action=edit&redlink=1) призван обеспечить организации, создающие критически важные ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *mission-critical*) [IT решения](http://ru.wikipedia.org/w/index.php?title=IT_%D1%80%D0%B5%D1%88%D0%B5%D0%BD%D0%B8%D1%8F&action=edit&redlink=1) на базе продуктов и технологий Microsoft, техническим руководством по достижению их надёжности ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *reliability*), доступности ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *availability*), удобства сопровождения ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *supportability*) и управляемости ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *manageability*). [MOF](http://ru.wikipedia.org/w/index.php?title=MOF&action=edit&redlink=1) затрагивает вопросы, связанные с организацией персонала и процессов, технологиями и менеджментом в условиях сложных ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *complex*), распределённых ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *distributed*) и разнородных ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *heterogeneous*) IT-сред. [MOF](http://ru.wikipedia.org/w/index.php?title=MOF&action=edit&redlink=1) основан на лучших производственных методиках, собранных в *IT Infrastructure Library* ([ITIL](http://ru.wikipedia.org/wiki/ITIL)), составленных *Central Computer and Telecommunications Agency* — Агентством правительства Великобритании.

Создание бизнес-решения в рамках отведенных времени и бюджета требует наличия испытанной методологической основы. [MSF](http://ru.wikipedia.org/wiki/MSF) предлагает проверенные методики для планирования, проектирования, разработки и внедрения успешных IT-решений. Благодаря своей гибкости, масштабируемости и отсутствию жестких инструкций [MSF](http://ru.wikipedia.org/wiki/MSF) способен удовлетворить нужды организации или проектной группы любого размера. Методология [MSF](http://ru.wikipedia.org/wiki/MSF) состоит из принципов, моделей и дисциплин по управлению персоналом, процессами, технологическими элементами и связанными со всеми этими факторами вопросами, характерными для большинства проектов.

MSF состоит из двух моделей и трех дисциплин. Они подробно описаны в 5 whitepapers. Начинать изучение MSF лучше с моделей, а затем перейти к дисциплинам.

MSF содержит:

* **модели**:
  + модель проектной группы
  + модель процессов
* **дисциплины**:
  + дисциплина [*управление проектами*](http://ru.wikipedia.org/wiki/%D0%A3%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B0%D0%BC%D0%B8)
  + дисциплина [*управление рисками*](http://ru.wikipedia.org/wiki/%D0%A3%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_%D1%80%D0%B8%D1%81%D0%BA%D0%B0%D0%BC%D0%B8)
  + дисциплина [*управление подготовкой*](http://ru.wikipedia.org/w/index.php?title=%D0%A3%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BF%D0%BE%D0%B4%D0%B3%D0%BE%D1%82%D0%BE%D0%B2%D0%BA%D0%BE%D0%B9&action=edit&redlink=1)

## Модель проектной группы MSF

**Модель проектной группы** MSF (MSF Team Model) описывает подход Майкрософт к организации работающего над проектом персонала и его деятельности в целях максимизации успешности проекта. Данная модель определяет ролевые кластеры, их области компетенции и зоны ответственности, а также рекомендации членам проектной группы, позволяющие им успешно осуществить свою миссию по воплощению проекта в жизнь.

Модель проектной группы MSF разрабатывалась в течение нескольких лет и возникла в результате осмысления недостатков пирамидальной, иерархической структуры традиционных проектных групп.

В соответствии с моделью MSF проектные группы строятся как небольшие многопрофильные команды, члены которых распределяют между собой ответственность и дополняют области компетенций друг друга. Это дает возможность четко сфокусировать внимание на нуждах проекта. Проектную группу объединяет единое видение проекта, стремление к воплощению его в жизнь, высокие требования к качеству работы и желание самосовершенствоваться.

Ниже описываются основные принципы, ключевые идеи и испытанные методики MSF в применении к модели проектной группы.

MSF включает в себя ряд **основных принципов**. Вот те из них, которые имеют отношение к успешной работе команды:

1. Распределение ответственности при фиксации отчетности
2. Наделяйте членов команды полномочиями
3. Концентрируйтесь на бизнес-приоритетах
4. Единое видение проекта
5. Проявляйте гибкость — будьте готовы к переменам
6. Поощряйте свободное общение

Успешное использование модели проектной группы MSF основывается на ряде **ключевых концепций** (key concepts):

1. Команда соратников
2. Сфокусированность на нуждах заказчика
3. Нацеленность на конечный результат
4. Установка на отсутствие дефектов
5. Стремление к самосовершенствованию
6. Заинтересованные команды работают эффективно

MSF основан на постулате о шести качественных целях, достижение которых определяет успешность проекта. Эти цели обуславливают модель проектной группы. В то время как за успех проекта ответственна вся команда, каждый из её ролевых кластеров, определяемых моделью, ассоциирован с одной из упомянутых шести целей и работает над её достижением.

**Посредством пакета руководств MSF (Microsoft Solutions Framework) гигант IT индустрии решил поделиться опытом и накопленной информацией в области проектирования, разработки, внедрения и сопровождения IT -проектов**. База знаний MSF состоит из оригинальных моделей, методов и взглядов на такие области знаний как управление проектами, персоналом, планирование, анализ рисков и другие смежные дисциплины. Нельзя сказать, что MSF очередной чисто теоретический взгляд на управление. В руководствах по MSF кроме методов, моделей и концепций встречаются практические советы и приёмы, отнюдь не лишенные рациональности. Структурно пакет руководств MSF разделён на пять документов, так называемых «белых книг», каждый из которых охватывает определенную дисциплину или модель MSF : «Модель процессов MSF», «Модель проектной группы MSF», «Дисциплина управления проектами MSF», «Дисциплина управления рисками MSF» и «Дисциплина управления подготовкой MSF». Предметной областью рассматриваемой методологии является IT -проект (разработка ПО, внедрение/адаптация уже разработанных систем, разворачивание сетевой инфраструктуры или всё вышеперечисленное в комплексе).

**2. Базовые концепции и принципы модели процессов MSF**

«Информационная работа – это работа мысли»  
Билл Гейтс. «Бизнес со скоростью мысли»

**2.1. Единое видение проекта**

Любой коллективный труд, а тем более такой нетривиальный как разработка и внедрения программного обеспечения, имеет мало шансов на успех, если все участвующие стороны не представляют (или еще хуже не правильно представляют) конечной цели проекта, иначе говоря, не имеют единого видения (shared vision) проекта. Все заинтересованные лица и просто участники проекта должны чётко представлять конечный результат. Всем должна быть понятна цель проекта. Подход MSF не зря акцентирует внимание на этом, казалось бы, умозрительном принципе. На практике не всегда так просто обеспечить единое понимание целей и задач проекта. Зачастую эта единая цель вступает в противоречие с интересами некоторых участников проекта. Или эти участники видят выгоды только от одной подсистемы проекта, которая касается их непосредственно. Например, бывает очень тяжело на последних этапах проекта объяснить сотрудникам склада, зачем им вводить номер накладной на отпущенный товар. Им он не нужен. Зато он нужен бухгалтерии. Конфликт возникает, потому что кладовщики думали, что автоматизируют склад, а не предприятие в целом. Поэтому MSF выделяет в жизненном цикле проекта целую фазу для обеспечения единого видения проекта.

**2.2. Управляйте компромиссами**

Наверное, не было еще ни одного менеджера проекта, которому не приходилось бы бороться с разрастанием рамок проекта и, как следствие, с перерасходом средств и затягиванием сдачи. Настоящим искусством менеджера является элегантное нахождение компромисса между ресурсами проекта (людскими, финансовыми и пр.), календарным графиком (временем) и реализуемыми возможностями (рамками).

**2.2.1. Треугольник компромиссов**

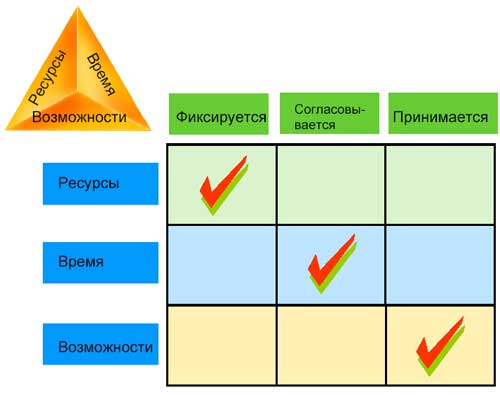
Часто бывает очень полезно изобразить эту зависимость заказчику в виде треугольника компромиссов



После достижения утвержденного равновесия с заказчиком, т.е. на запрашиваемые возможности вы назвали и зафиксировали сроки и смету, любое изменение на одной из сторон треугольника влечет изменение на двух оставшихся. Такой подход служит удобным инструментом для нахождения компромиссов с заказчиком и поможет объяснить суть имеющихся ограничений. Иногда имеет смысл добавлять еще четвертое измерение – качество, за счёт которого можно сэкономить время и ресурсы при неизменных возможностях, хотя, на мой взгляд, это последнее на чём стоит экономить. Однако отечественные реалии диктуют свои законы и не редки случаи, когда важен сам факт внедрения и заниженная планка качества решения (quality bar) всех устраивает. В таких случаях участники проекта должны, по крайней мере, знать об этом, и этот факт должен найти адекватное отражение в проектной документации (например, отсутствие дизайнерского оформления интерфейса или ведения журнала операций пользователя в системе).

**2.2.2. Матрица компромиссов**

Для эффективного достижения компромиссов в течение всего жизненного цикла проекта, MSF рекомендует на начальных этапах зафиксировать приоритеты факторов проекта (ресурсы, время, возможности). На один из факторов влиять в течение проекта будет практически невозможно (Фиксируется), - другой будет обладать некоторым приоритетом при разрешении компромиссов (Согласовывается) и оставшийся будет принят в соответствии с первыми двумя (Принимается).



Зафиксировать приоритеты в проектной документации можно с помощью простых текстовых оборотов: «Зафиксировав ресурсы, мы согласовываем календарный график и принимаем результирующий объем функциональности решения» или «Зафиксировав объем функциональности решения, мы согласовываем затрачиваемые ресурсы и принимаем результирующие сроки» и т.п. В дальнейшем возврат к приоритетам может сильно помочь при нахождении компромисса внутри проектной группы (обращаю внимание на то, что представитель заказчика тоже является членом проектной группы).

**2.3. Проявляйте гибкость – будьте готовы к переменам**

В отличие от традиционной модели управления проектами, в которой подразумевается четкая формулировка требований на начальном этапе проекта и разработка на основании ТЗ, подход MSF основывается на принципе изменяющихся проектных условий. Проектная группа должна быть готова к переменам и методология MSF предоставляет эффективный инструментарий для адекватной и своевременной реакции на изменения в проектной среде.

**2.4. Концентрация на бизнес-приоритетах**

При создании любого решения необходимо сосредоточиться на той отдаче и выгоде, которую ожидает получить потребитель решения. Если проект нацелен на предприятие, то точкой концентрации будет бизнес-отдача (business - value). В отношении персональных программ, как, например, компьютерная игра, отдачу можно выразить в виде эмоционального удовлетворения потребителя. Можно сказать уверенно, что MSF не подходит для волонтёрских проектов, удовлетворяющих чьи-то амбиции или являющихся творческой отдушиной для разработчиков – творцов с рождения.

**2.5. Поощряйте свободное общение**

До сих пор большинство IT организаций ограничивает доступность к информации участников проекта исключительно необходимой для выполнения их работы или хотя бы пытается это делать, порождая недоверие внутри рабочей группы, но те, кому интересно всё равно узнают рано или поздно то, что им интересно. Методология MSF разрушает этот стереотип, поощряя свободное общение внутри проекта. Это заметно сокращает риск недопонимания, возникновения недоразумений и стимулирует максимальный вклад всех участников проекта в общее дело. Стоит заметить, что здесь не идет речь о финансовой или юридической информации, которая априори не может быть общедоступной. Зачастую разработчики не знают о проблемах возникающих у внедренцев, а тестирование не в курсе, что идет уже вторая неделя срыва сроков. Во избежание этого MSF предлагает проведение анализа хода работы над проектом в определённых временных точках, документирование результатов пройденных фаз проекта и обеспечение свободного общения в течение жизненного цикла проекта. Благодаря такому подходу обеспечивается выявление рисков всеми участниками проекта на ранних стадиях, что без сомнения вносит ощутимый вклад в успех решения.

**2.6. Создавайте базовые версии**

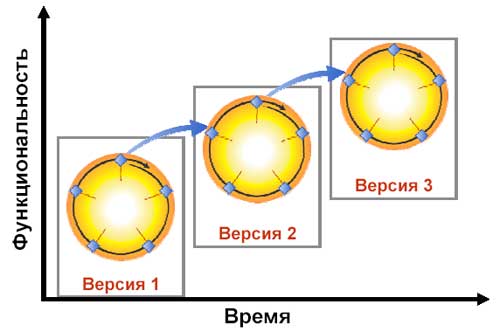
Под базовой версией в MSF подразумевается зафиксированное состояние любого проектного артефакта (а лучше всех одновременно) в том числе программный код, план проекта, руководства пользователя, настройки серверов и т.п. В последствии, имея базовые версии, вы сможете эффективно управлять изменениями, вернуться на шаг назад, если это необходимо и выполнять аналитику по завершению проекта.

**3. MSF – симбиоз итеративного и фазового подхода**

«Развитие, как бы повторяющее пройденные уже ступени,  
но повторяющее их иначе, на более высокой базе   
(“отрицание отрицания”), развитие, так сказать,  
по спирали, а не по прямой линии; — развитие скачкообразное,  
катастрофическое, революционное…»

**3.1. Итеративный подход**

Весь жизненный цикл проекта протекает в виде последовательности итераций выпуска версий. MSF рекомендует вкладывать в первую версию продукта только базовую функциональность и затем наращивать ее в следующих версиях. Для малых проектов иногда бывает достаточно одной итерации, однако все равно рекомендуется не упускать возможности версионирования т.к. это эффективный инструмент достижения успеха проекта.



Пересмотр функциональности, сетевых графиков работ, планов, спецификаций, требований и других проектных артефактов не прекращается до конца проекта и производится после каждой итерации. Такой подход позволяет планировать возможности для последующих версий, делать учёт опыта предыдущих циклов, обеспечивая гибкость и устойчивость к переменам требований. Таким образом, обеспечивается ведение «живой» документации, которая изменяется по мере эволюции проекта. В рамках одной итерации все документы (равно как и программный код) тоже развиваются итеративно. Например, план тестирования или концепция проекта (shared vision document) распространяется среди всех ролей проекта в виде «подходов» (approaches) еще до утверждения и затем итеративно эволюционируют благодаря обратной связи участников проекта до формы, подлежащей утверждению. Все изменения в уже принятые документы в рамках одной итерации (например, утверждённое ТЗ) вносятся посредством компромиссов проектной группы и согласно технологии управления изменениями, и часто имеет смысл отложить эти изменения до следующей версии, дабы не допускать разрастания рамок проекта и срыва сроков сдачи текущей версии. Создание базовой версии всех проектных документов на самых ранних этапах дает возможность всем членам проектной группы осмыслить свои задачи и цели проекта, а так же приступить к работе с минимальными задержками.

Методология MSF рекомендует делать текущие сборки программных компонент решения. Это особенно важно в сложных комплексах, где необходимо учитывать и тестировать совместимость компонент. Для этих целей Майкрософт при разработке своих продуктов выполняет ежедневные билды (daily builds). Разработка и тестирование ведутся практически одновременно, что увеличивает надежность результирующего кода. Для осуществления такого подхода к разработке необходимо вести управление конфигурациями проекта. Необходим строгий мониторинг и контроль версий программного кода, документации, аппаратных настроек и других артефактов проекта. Управление конфигурациями в свою очередь служит эффективным инструментом управления изменениями в проекте, которое регламентируют процесс внесения изменений в артефакты проекта от запроса на изменение, до включения его в базовую версию. Вот основные рекомендации MSF для выпуска версий решения:

* Создавая планы, предусматривайте версионирование.
* Прежде всего, поставляйте базовую функциональность.
* Выбирайте приоритеты, учитывая риски.
* Осуществляйте частые итерации разработки.
* Институцируйте процедуры контроля изменений в проекте.
* Не создавайте новых версий, если они не увеличивают ценность решения.

**3.2. Подход, основанный на фазах и вехах**

Что может быть ближе отечественной АСУшной душе, чем такой подход. Десятилетиями наши системы внедрялись поэтапно. Не годиться – на доработку. Подправили – совещание. Майкрософт не предлагает то же самое, только с перламутровыми пуговицами. В рамках одной итерации, жизненный цикл выпуска версии разбивается на пять фаз (выработка концепции (единого видения), планирование, разработка, стабилизация (тестирование), внедрение). Каждая фаза цикла заканчивается главной вехой (контрольной точкой). Соответственно главные вехи будут иметь названия: концепция продукта утверждена, планы продукта утверждены, разработка завершена, готовность решения утверждена, внедрение завершено. Веха является точкой синхронизации достигнутых результатов и ожиданий заказчика, а также анализа проектной среды. В решении о закрытии очередной фазы должны принимать участие ответственные представители всех ролевых кластеров (разработка, тестирование, внедрение, управление проектом и пр.).



В этой контрольной точке всплывают все противоречия и коллизии, возникшие за период фазы проекта. Хорошей практикой является не откладывание проблем до конца фазы, дабы не тратить время и нервы всех участников совещания в главной вехе (хотя на самом деле совещания может и не быть, а возможно просто рассылка и утверждение документов по электронной почте). В рамках фазы должны присутствовать промежуточные вехи, обозначивающие достигнутые промежуточные результаты. Например, на фазе разработки такими промежуточными вехами могут быть: билд n завершен, билд n +1 завершен и т.д. MSF дает определенные рекомендации (рассмотрены ниже) относительно промежуточных вех на каждой фазе, однако проектная команда может сформировать свои специфические для проекта и фазы промежуточные вехи.

**3.3. Модель проектной группы MSF**

Вот здесь начинается самое интересное и, на мой взгляд, новаторское в подходе MSF. Этой теме уделена целая «белая книга» из пакета руководств MSF и здесь мы рассмотрим только основные подходы к построению проектной группы. Существуют основные принципы и концепции модели проектной группы, которые во многом являются чуждыми большинству IT компаний. Здесь разрушаются некоторые стереотипы (например, диктаторские полномочия и соответствующая ответственность менеджера проекта) и предлагается несколько иная, более органичная структура организации рабочих групп. Подход, по меньшей мере, весьма интересен.

Проектная группа разделяется на шесть ролевых кластеров, соответствующих шести качественно различным задачам проекта.

* Управление продуктом
* Управление программой
* Разработка
* Тестирование
* Удовлетворение потребителя
* Управление выпуском

Между этими кластерами образовывается устойчивый баланс ответственности и полномочий, позволяющий команде эффективно функционировать. Как я уже упоминал, существуют основные принципы модели проектной группы.

* Распределяйте ответственности при фиксации отчетности
* Наделяйте соответствующими полномочиями ролевые кластеры.

Таким образом, все ролевые кластеры в команде **равноправны** . Однако иерархия отчётности при этом не нарушается, а остается прежней. Это путь к качественному продукту. Составление календарных планов работ осуществляется руководителем соответствующего ролевого кластера. Это побуждает к большей ответственности за свои сроки и обязательства, чем за те, которые на тебя спустили сверху. Каждый ролевой кластер принимает участие в принятии решений о завершении фаз проекта. Противоречия разрешаются методом компромиссов. Кроме того хорошей практикой является вход в состав группы представителей заказчика, что увеличивает заинтересованность и активное содействие внедрению со стороны заказчика. Кроме того, заказчик становится более информированным о ходе проекта т.к. информация по проекту между кластерами доступна членам команды. Например, если разрабатывается система автоматизации предприятия, хорошей, на мой взгляд, практикой будет назначить руководителем кластера Управление выпуском (по сути, это внедрение и материальное обеспечение внедрения) представителя заказчика, а исполнителями же внедрения - сотрудников компании-исполнителя. В случае успеха слава за удачу (речь идет не о финансовом поощрении) разделится поровну между всеми участниками проекта. А вот ответственность распределена в соответствии с ролевым кластером. Менеджер проекта (ролевой кластер Управление программой) будет отвечать, если проект не уложится в срок и в смету, разработка – если не выполнит разработку в названный ей же срок, тестирование – если в выпущенной версии будут возникать проблемы и т.д. Этап планирования не завершиться пока все ролевые кластеры не будут согласны с планом проекта. Это, конечно, несколько увеличит срок принятия решений, зато многократно уменьшит вероятность ошибочных решений. Что важнее решать вам – о управленец! Однако, по мнению Майкрософт, такой подход намного сильнее стимулирует творчество, ответственность, коммуникации и взаимопонимание в проектной группе, чем привычный нам, вид управления самодержца. Перечислю основные области ответственности ролевых кластеров.

|  |  |
| --- | --- |
| Управление продуктом | Представление интересов заказчика; аналитика; обоснование бизнес-отдачи; формирование единого видения и рамок проекта; управление требованиями заказчика; определение приоритетов факторов (время/рамки/ресурсы); PR продукта; план коммуникаций. |
| Управление программой | Управление процессом разработки с целью получение продукта в рамках проектных ограничений; формулировка спецификации и разработка архитектуры; управление совещаниями и коммуникациями; формирует сводный план; управление рисками; сетевой график работ; отчётность. |
| Разработка | Определяет дизайн решения; оценка времени разработки; разработка; консультирование. |
| Тестирование | Планирование тестирования; тестирование. |
| Удовлетворение потребителя | Представляет интересы потребителя (на заказчика, а конечного пользователя системы); сбор требований потребителя; формирует требования к эргономике, справочной системе и учебным материалам. |
| Управление выпуском | Внедрение; обеспечение сопровождения; материальное обеспечение и логистика; инфраструктура поставок. |

Для малых проектов возможно совмещение ролевых кластеров, например Тестирование и Удовлетворение Потребителей. Однако MSF настоятельно рекомендует не объединять кластер Разработка, ни с каким из других потому же, почему нельзя отвлекать хирурга во время операции. И Управление Продуктом с Управлением Программой, потому что, как известно, единство и борьба противоположностей рождает качественное диалектическое развитие.

Рассмотрим каждую из фаз жизненного цикла проекта подробнее.

**3.4. Фаза выработки концепции**

Это первая фаза жизненного цикла проекта. Начинается она с формирования ядра проектной группы (если конечно это первая итерация продукта). Затем закладывается основа, фундамент, будущего решения на основе единого видения проекта (ничем не ограниченное представление о целях и задачах, стоящих перед проектной группой). После этого очерчиваются рамки (чётко описанные задачи, которые предстоит решить), однозначно описывающие то, что предстоит сделать в рамках проектных ограничений, и оцениваются риски. Нельзя смешивать эти два понятия, но и нельзя рассматривать одно в отрыве от другого. Если программист будет создавать продукт, не владея документом единого видения (shared vision document) только на основании описания рамок (scope document), он вероятнее всего не сможет создать то, что ожидает заказчик, ведь основные цели, представления и ожидания заказчика сконцентрированы именно в документе единого видения. Оба эти документа должны создаваться итеративно и тщательно, минимизируя дальнейшие отклонения от них.

Главной вехой этой фазы будет событие «Концепция утверждена». К этому моменту у заказчика и проектной группы уже сформированы устойчивые представления о задачах, функциональности и ограничениях проекта. К моменту этой вехи должны быть готовы следующие документы: общее описание и рамки проекта (vision \ scope document), документ оценки рисков, описание структуры проекта. Задачи проектной группы на этой фазе распределяются следующим образом:

|  |  |
| --- | --- |
| Ролевой кластер | Фокус |
| Управление продуктом | Общие цели проекта; выявление нужд и требований заказчика; документ общего описания и рамок проекта. |
| Управление программой | Цели дизайна; концепция решения; структура проекта. |
| Разработка | Прототипирование; анализ технологических возможностей; анализ осуществимости. |
| Удовлетворение потребителя | Необходимые эксплуатационные характеристики решения и их влияние на его разработку. |
| Тестирование | Стратегии тестирования; критерии приемлемости, их влияние на разработку решения. |
| Управление выпуском | Требования внедрения и их влияние на разработку решения; требования сопровождения. |

MSF рекомендует обозначить следующие промежуточные вехи в течение фазы:

* Ядро проектной группы сформировано
* Черновой вариант концепции проекта составлен

**3.5. Фаза планирования**

Наступает важнейший этап разработки решения. Фаза планирования включает в себя подготовку проектной группой функциональной спецификации, разработку дизайнов, подготовку рабочих планов, оценку проектных затрат и сроков разработки различных составляющих проекта. Всё начинается с анализа и документирования проектных требований с разделением их на категории: бизнес-требования, потребительские требования, эксплуатационные требования и системные требования. Затем при создании функциональной спецификации нужно следить за соответствием (traceability) функциональности и существующих требований. Детализация требований происходит по известному всем аналитикам сценарию, прибегая к моделированию вариантов использования, (use - case modeling) и стори-боардингу (story-boarding), хотя у каждого опытного аналитика наверняка есть свой арсенал методов. Затем работа переходит в область проектирования (дизайна), где разрабатываются концептуальный, логический и физический дизайны, служащие уже инструментами для разработчиков. Результаты проектирования документируются в функциональную спецификацию, которая детально описывает вид и поведение всех составляющих решения. На основании спецификации работает команда разработчиков (с учётом синхронизации действий между собой), производится оценивание работ, достигается чёткое соглашение с заказчиком о том, что должно быть сделано. Как только создана спецификация, руководители ролевых кластеров смогут создать детальный план, относящийся к его роли. Примерами планов могут стать: план внедрения, план тестирования, план обучения, план мер безопасности и т.п. Затем проектная группа коллективно анализирует планы и выявляет зависимости между ними. Эти планы в последствии объединяются в сводный план проекта и сводный сетевой график работ. На этом же этапе создается также план управления рисками, к составлению которого есть смысл привлекать всех участвующих в проекте лиц.

Завершается этап вехой «Планы проекта утверждены», на момент которой уже существуют документы: функциональная спецификация, план управления рисками, сводный план и сводный календарный график работ.

Основные задачи проектной группы на фазе планирования:

|  |  |
| --- | --- |
| Ролевой кластер | Фокус |
| Управление продуктом | Концептуальный дизайн; анализ бизнес-требований; коммуникационный план. |
| Управление программой | Концептуальный и логический дизайн; функциональная спецификация; сводный план и сводный календарный график проекта; бюджет. |
| Разработка | Оценка технологий; логический и физический дизайн; план и календарный график разработки; смета разработки (development estimates). |
| Удовлетворение потребителя | Сценарии/примеры использования, пользовательские требования, требования локализации и общедоступности (accessibility); пользовательская документация/план обучения/график тестирования удобства эксплуатации; обучение. |
| Тестирование | Оценка дизайна; требования тестирования; план и календарный график тестирования. |
| Управление выпуском | Оценка дизайна; эксплуатационные требования; план и календарный график пилотного и окончательного внедрения. |

MSF рекомендует создавать следующие промежуточные вехи:

* Верификация технологий
* Базовая версия функциональной спецификации создана
* Базовая версия сводного плана проекта создана
* Базовая версия сводного календарного графика работ создана
* Среды разработки и тестирования развернуты

**3.6. Фаза разработки**

Название фазы говорит само за себя. Однако не стоит думать, что все остальные ролевые кластеры проекта на этой фазе могут пить кофе. Все роли принимают активное и деятельно участие в тестировании, выявлении дефектов, анализе удовлетворения нужд заказчика и других задачах в соответствии со своим ролевым кластером. Равно как и сама разработка, очень редко заканчивается этой фазой (да и начинается тоже редко только с этого момента), обычно разработка продолжается и на фазе стабилизации и даже на фазе внедрения иногда приходится что-то править. К моменту завершения этой фазы разработка всех компонент завершена и решение готово к комплексному тестированию.

Задачи ролей во время этой фазы:

|  |  |
| --- | --- |
| Ролевой кластер | Фокус |
| Управление продуктом | Ожидания заказчика. |
| Управление программой | Управление функциональной спецификацией; мониторинг проекта; доработка планов. |
| Разработка | Разработка программного кода и инфраструктуры; документирование конфигураций. |
| Удовлетворение потребителя | Обучение; доработка плана обучения; тестирование удобства эксплуатации (usability testing); графический дизайн. |
| Тестирование | Функциональное тестирование; выявление проблем; тестирование документации; доработка плана тестирования. |
| Управление выпуском | Чеклисты развертывания (rollout checklists); доработка планов внедрения (включая пилотное внедрение); чеклисты подготовки к внедрению (site preparation checklists). |

Рекомендуемые промежуточные вехи:

* Концепция подтверждена
* Билд n завершен, билд n +1 завершен…

**3.7. Фаза стабилизации**

На фазе стабилизации фокус смещается в область тестирования и документирования решения. А также создание пилотного внедрения. Результатами этой фазы являются: окончательный продукт (golden release), документация выпуска, материалы поддержки решения, результаты тестирования, проектная документация и анализ пройденной фазы.

Задачи проектной группы на этой фазе:

|  |  |
| --- | --- |
| Ролевой кластер | Фокус |
| Управление продуктом | Исполнение коммуникационного плана; планирование премьеры продукта. |
| Управление программой | Мониторинг проекта; приоритезация ошибок. |
| Разработка | Устранение ошибок; оптимизация программного кода. |
| Удовлетворение потребителя | Доработка эксплуатационных руководств; учебные материалы. |
| Тестирование | Тестирование; сообщение об ошибках и их статусе; тестирование конфигурации. |
| Управление выпуском | Развертывание и поддержка пилотного внедрения; планирование внедрения; обучение персонала сопровождения. |

**3.8. Фаза внедрения**

Наконец-то. Если проект докатился до этой фазы, то можно считать, что треть пути пройдена... Это конечно шутка, так быть не должно, хотя так бывает довольно часто, особенно если были допущены ошибки на начальных этапах проекта. На этом этапе группа внедряет решение, стабилизирует его и, пробиваясь сквозь противоборство службы сопровождения и поддержки, таки передает решение им, получив одобрение заказчика. И, наконец, получает вожделенный акт. Однако согласно MSF и вопреки реалиям работа группы на этом не заканчивается. Группа работает над сбором метрик и анализом проекта, дабы бесценный опыт, полученный в результате проекта, не канул в дебрях короткой памяти человеческой.

Результатами фазы будут: информационные системы эксплуатации и поддержки, процедуры и процессы, базы знаний, отчёты, журналы протоколов, версии проектных документов, отчёт о завершении проекта, окончательные версии всех проектных документов, описание последующих шагов.

Задачи группы на фазе внедрения:

|  |  |
| --- | --- |
| Ролевой кластер | Фокус |
| Управление продуктом | Получение отзывов и оценок заказчика; акт о приеме выполненной работы. |
| Управление программой | Сопоставление рамок проекта с поставленным решением; управление стабилизацией. |
| Разработка | Разрешение проблем; поддержка эскалации. |
| Удовлетворение потребителя | Обучение; управление календарным графиком обучения. |
| Тестирование | Тестирование производительности. |
| Управление выпуском | Управление внедрением; одобрение изменений. |

Не нужно также забывать, что после внедрения первой версии продукта, может последовать следующая итерация жизненного цикла.

**3.9. Замечания**

Существуют проекты, в которых может отсутствовать фаза внедрения или разработки (например, разработка игры, или адаптация 1С).

В первую очередь реализовуйте рискованные нововведения, минимизируя влияние риска на весь проект.

Деятельность каждой фазы не ограничивается ее рамками, как было уже сказано, разработка продолжается почти всегда.

Длительность фаз может быть совсем не одинакова, как изображено на картинках (суть это замечания взята с текста MSF, я бы в жизни не догадался его написать).

**4. Резюме**

Подытожим. Методология очень интересна, если не сказать больше. Ее можно использовать как эффективный инструмент, подходя к нему с творческой изобретательностью и осмысленностью. Нет, наверное, такой методологии управления, строгое соответствие которой, обеспечит успешность проекта. Не нужно воспринимать MSF как догму или панацею. Можно её адаптировать под свои нужды или использовать только фрагменты или концепции применимые в Вашем проекте.

Например, в качестве дополнительного фактора проекта при поиске компромисса (наравне с ресурсами, временем и возможностями) может выступать безопасность или можно добавить дополнительную промежуточную веху «Контроль безопасности» в конце каждой фазы проекта, если того, конечно, требует ваша проектная среда.

**11. Scrum: гибкое управление разработкой (XP-процесс)**

В большинстве случаев программирование — сложный, слабо определенный процесс, требующий от разработчиков творческого подхода. Различные agile-технологии позволяют организовать процесс постепенного приближения к цели проекта путем проведения циклов испытаний с корректировкой последующих, основанных на анализе результатов предыдущих. Scrum — одна из первых методологий циклического наращивания функциональности и корректировки хода проекта на основе анализа обратной связи от пользователей.

Термин Scrum был впервые озвучен в работе Хиротаки Такеучи и Икуджиро Нонака, опубликованной в Harvard Business Review [[1](http://www.osp.ru/os/2007/04/4220063/#1)], где авторы отметили успешность проектов, использующих небольшие команды без жесткой специализации. Такие команды чем-то напоминают конструкцию схватки (scrum) в регби, которая назначается при нарушении правил или остановке игры. Джеф Сазерленд использовал эту работу при создании методологии для корпорации Easel в 1993 году, которую по аналогии и назвал Scrum, а Кен Швабер формализовал процесс для использования в индустрии разработки программного обеспечения, представив в 1995 году работу [2] на конференции Object-Oriented Programming Systems, Languages & Applications.

Методология Scrum устанавливает правила управления процессом разработки и позволяет использовать уже существующие практики кодирования, корректируя требования или внося тактические изменения. Использование этой методологии дает возможность выявлять и устранять отклонения от желаемого результата на более ранних этапах разработки программного продукта.

Основа Scrum — итеративная разработка. Scrum определяет правила, по которым должен планироваться и управляться список требований к продукту для достижения максимальной прибыльности от реализованной функциональности; правила планирования итераций для максимальной заинтересованности команды в результате; основные правила взаимодействия участников команды для максимально быстрой реакции на существующую ситуацию; правила анализа и корректировки процесса разработки для совершенствования взаимодействия внутри команды. Каждую итерацию можно описать так: планируем — фиксируем — реализуем — анализируем. За счет фиксирования требований на время одной итерации и изменения длины итерации можно управлять балансом между гибкостью и планируемостью разработки.

**Концепция Scrum**

Scrum — простой каркас, который можно использовать для организации команды и достижения результата более продуктивно и с более высоким качеством за счет анализа сделанной работы и корректировки направления развития между итерациями. Методология позволяет команде выбрать задачи, которые должны быть выполнены, учитывая бизнес-приоритеты и технические возможности, а также решить, как их эффективно реализовать. Это позволяет создать условия, при которых команда работает с удовольствием и максимально продуктивно. К примеру, возможность самостоятельного выбора объема и пути решения задач без внешнего давления позволяет всем участникам команды почувствовать себя активными игроками, вовлеченными в процесс, а не простыми исполнителями, от которых требуется лишь четкая реализация поручений.

Scrum фокусируется на постоянном определении приоритетных задач, основываясь на бизнес целях, что увеличивает полезность и доходность проекта на его ранних стадиях. Так как при инициации проекта его доходность определить почти невозможно, Scrum предлагает концентрироваться на качестве разработки и к концу каждой итерации иметь промежуточный продукт, который можно использовать, пусть и с минимальными возможностями. Например, результатом итерации может быть каркас сайта, который можно показать на презентации.

Методология Scrum ориентирована на то, чтобы оперативно приспосабливаться к изменениям в требованиях, что позволяет команде быстро адаптировать продукт к нуждам заказчика. Такая адаптация достигается за счет получения обратной связи по результатам итерации: имея после каждой итерации продукт, который уже можно использовать, показывать и обсуждать, легче собирать информацию и делать правильные корректировки и изменять приоритеты требований. Например, если каркас сайта показать потенциальным пользователям, то появится много вопросов, на основании которых можно скорректировать то, что уже написано или еще не реализовано, понять что более важно пользователю.

Девиз Scrum — «анализируй и адаптируй»: анализируй то, что получил, адаптируй то, что есть, к реальной ситуации, а потом анализируй снова. Чем меньше формализма, тем более гибко и эффективно можно работать, — это основной принцип данной методологии. Но это не означает, что формальных процессов не должно быть совсем, их должно быть достаточно для организации эффективного взаимодействия и управления проектом. Формальная часть Scrum состоит из трех ролей, трех практик и трех основных документов.

**Роли**

*Владелец продукта* (Product Owner) — человек, поставляющий требования программистам. От того, как четко написаны требования, зависит, насколько часто команде придется переключаться с задачи на задачу в связи с отсутствием нужной информации, как много нужно задавать вопросов, на которые уходит дополнительное время, как сильно придется изменять уже написанную функциональность от итерации к итерации и, соответственно, эффективность разработки в целом. Обычно владелец продукта является представителем или доверенным лицом заказчика, а для компаний, выпускающих коробочные продукты, он представляет рынок, на котором реализуется продукт. Владелец продукта должен составить бизнес план, показывающий ожидаемую доходность и план развития с требованиями, отсортированными по коэффициенту окупаемости инвестиций. Исходя из имеющейся информации, владелец продукта подготавливает список требований, отсортированный по значимости. Чем лучше владелец продукта описывает требования, управляет приоритетами и чем быстрее выдает информацию, тем больший финансовый эффект получит компания от методологии. В обязанности этого сотрудника входит своевременное предоставление требований к продукту, определение дат и содержания релизов, эффективное управление приоритетами и корректировка требований для достижения максимальной окупаемости инвестиций в продукт.

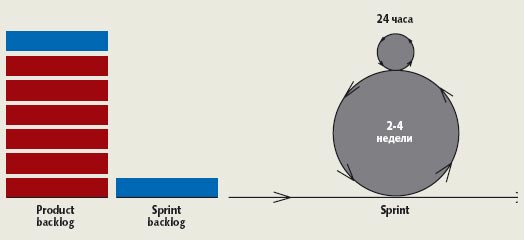
От человека, исполняющего роль *Scrum-мастера* (Scrum Master), во многом зависит самостоятельность, инициативность программистов, удовлетворенность сделанной работой, атмосфера в команде и результат всей работы. Этот человек должен быть одним из членов команды разработки и участвовать в проекте как разработчик. Он отвечает за своевременное решение текущих проблем, от ремонта сломанного стула до обеспечения необходимой информацией членов команды для продолжения их работы и загруженности, за поддержание нужных технических практик, используемых на проекте. В обязанности Scrum-мастера входит обеспечение максимальной работоспособности и продуктивности команды, четкого взаимодействия между всеми участниками проекта, своевременное решение всех проблем, тормозящих или останавливающих работу любого члена команды, ограждение команды от всех воздействий извне во время итерации и обеспечение следования процессу всех участников проекта.

*Scrum-команда* (Scrum Team) — группа, состоящая из пяти–девяти самостоятельных, инициативных программистов. Первая задача этой команды — поставить реально достижимую, прогнозируемую, интересную и значимую цель для итерации. Вторая задача — сделать все для того, чтобы эта цель была достигнута в отведенные сроки и с заявленным качеством. Цель итерации считается достигнутой только в том случае, если все поставленные задачи реализованы, весь код написан по определенным проектом «стандартам кодирования» (coding guidelines), программа протестирована полностью, а все найденные дефекты устранены. Программисты этой команды должны уметь оценивать и планировать свою работу, работать в команде, постоянно анализировать и улучшать качество взаимодействия и работы. В обязанности всех членов Scrum-команды входит участие в выборе цели итерации и определение результата работы. Они должны делать все возможное и невозможное для достижения цели итерации в рамках, определенных проектом, эффективно взаимодействовать со всеми участниками команды, самостоятельно организовывать свою работу, предоставлять владельцу рабочий продукт в конце каждого цикла.

**Практики**

Подготовка к первой итерации, называемой *спринт* (Sprint), начинается после того, как владелец продукта разработал план проекта, определил требования и отсортировал их в количестве, достаточном для наполнения одной итерации. Такой список требований называется *журналом продукта* (Product Backlog). При планировании итерации происходит детальная разработка сессий планирования спринта (Sprint Planning Meeting), который начинается с того, что владелец продукта, Scrum-команда и Scrum-мастер проверяют план развития продукта, план релизов и список требований. Scrum-команда проверяет оценки требований, убеждается, что они достаточно точны, чтобы начать работать, решает, какой объем работы она может успешно выполнить за спринт, основываясь на размере команды, доступном времени и производительности. Важно, чтобы Scrum-команда выбирала первые по приоритету требования из журнала продукта. После того как Scrum-команда обязуется реализовать выбранные требования, Scrum-мастер начинает планирование спринта. Scrum-команда разбивает выбранные требования на задачи, необходимые для его реализации. Эта активность в идеале не должна занимать больше четырех часов, и ее результатом служит список требований, разбитый на задачи, — *журнал спринта* (Sprint Backlog). Необходимо, чтобы все участники команды приняли на себя обязательство по реализации выбранной цели.

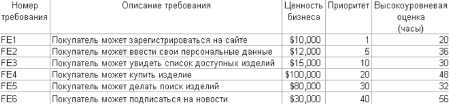
После окончания планирования начинается итерация. Каждый день Scrum-мастер проводит «скрам» (Daily Scrum Meeting) — пятнадцатиминутное совещание, цель которого — достичь понимания того, что произошло со времени предыдущего совещания, скорректировать рабочий план к реалиям сегодняшнего дня и обозначить пути решения существующих проблем. Каждый участник Scrum-команды отвечает на три вопроса: что я сделал со времени предыдущего скрама, что меня тормозит или останавливает в работе, что я буду делать до следующего скрама? В этом митинге может принимать участие любое заинтересованное лицо, но только участники Scrum-команды имеют право принимать решения. Правило обосновано тем, что они давали обязательство реализовать цель итерации, и только это дает уверенность в том, что она будет достигнута. На них лежит ответственность за их собственные слова, и, если кто-то со стороны вмешивается и принимает решения за них, тем самым он снимает ответственность за результат с участников команды.



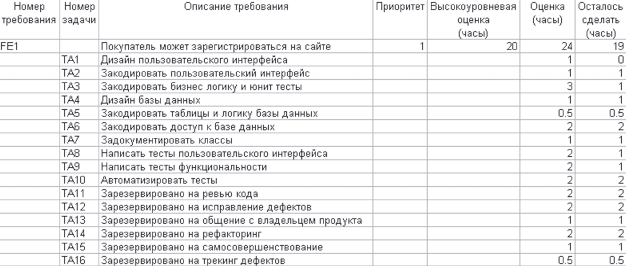
В конце каждого спринта проводится *демонстрационный митинг* (Sprint Review Meeting) продолжительностью не более четырех часов. Сначала Scrum-команда демонстрирует владельцу продукта сделанную в течение спринта работу, а тот в свою очередь ведет эту часть митинга и может пригласить к участию всех заинтересованных заказчиков. Владелец продукта определяет, какие требования из журнала спринта были выполнены, и обсуждает с командой и заказчиками, как лучше расставить приоритеты в журнале продукта для следующей итерации. Во второй части митинга производится анализ прошедшего спринта, который ведет Scrum-мастер. Scrum-команда ищет использованные в последнем спринте положительные и отрицательные способы совместной работы, анализирует их, делает выводы и принимает важные для дальнейшей работы решения. Scrum-команда также определяет программы, которые могут работать лучше, и ищет пути для увеличения эффективности дальнейшей работы. Затем цикл замыкается, и начинается планирование следующего спринта (рис. 1).

**Документы**

В начале проекта владелец продукта готовит журнал продукта (рис. 2) — список требований, отсортированный по значимости, а Scrum-команда дополняет этот журнал оценками стоимости реализации требований. Список должен включать функциональные и технические требования, необходимые для реализации продукта. Самые приоритетные из них должны быть достаточно детально прописаны, чтобы их можно было оценить и протестировать. О своевременной детализации требований должен заботиться владелец продукта и предоставлять необходимый объем в нужное время. В этом смысле программисты являются заказчиками требований для владельца продукта. И отношение владельца продукта должно быть соответствующее — каковы требования, такова и их реализация. В дальнейшем остальные требования должны постепенно уточняться и детализироваться до такого же уровня. Главное, чтобы у команды всегда был достаточный объем подготовленных к реализации требований.



После того как команда во время сессии планирования выбрала и обязалась реализовать набор требований из журнала продукта, эти требования разбиваются на более мелкие задачи, составляющие детализированный список требований — журнал спринта ([*рис. 3*](http://www.osp.ru/data/696/787/1238/057_r3.gif)). Разбивка на задачи должна быть сделана таким образом, чтобы выполнение одной задачи занимало не больше двух дней (считается, что менее детальная, например, полдня, разбивка приводит к более грубой оценке, чем приемлема в большинстве проектов, использующих методологию Scrum). Разбивка на задачи поможет так спланировать итерацию, чтобы в конце не осталось ни одной невыполненной задачи и, соответственно, достичь ее цели. После завершения детализации оценка журнала спринта сравнивается с первичной оценкой в журнале продукта. Если существует значительное расхождение, команда договаривается с владельцем продукта об объеме работ, который должен быть выполнен в течение итерации, и о том, какой объем будет перенесен на следующую. Менее важные и мало влияющие на цель итерации задачи выносятся из журнала спринта.



*График спринта* (Burndown Chart) показывает ежедневное изменение общего объема работ, оставшегося до окончания итерации. Это график позволяет команде разработчиков делать анализ текущей ситуации и своевременно реагировать на отклонения. График спринта позволяет также владельцу продукта наблюдать за ходом итерации — если общий объем работ не уменьшается каждый день, значит, что-то идет не так. Во время сессии планирования команда находит и оценивает задачи, которые надо выполнить для успешного завершения итерации. Сумма оценок всех задач в журнале спринта является общим объемом работы, который надо выполнить за итерацию. После завершения каждой задачи Scrum-мастер пересчитывает объем оставшейся работы и отмечает это на графике спринта. Только в том случае, если объем работ по окончании итерации закончился (в журнале спринта не осталось незавершенных задач), итерация считается успешной. График спринта используется как вспомогательный инструмент, позволяющий корректировать работу для завершения итерации вовремя, с работающим кодом и требуемым качеством.

Время между итерациями — это время принятия основополагающих решений, влияющих на ход всего проекта. Во время итерации никакие изменения извне не могут быть сделаны. После того как команда дала обязательство реализовать журнал спринта, он фиксируется, и изменения в нем могут быть сделаны только по следующим причинам:

* Scrum-команда в течение итерации получила лучшее представление о требованиях и нуждается в дополнительных задачах для успешного завершения итерации;
* найдены дефекты, которые нужно обязательно исправить для успешного завершения итерации;
* Scrum-мастер и Scrum-команда могут решить, что небольшие изменения, не влияющие на общий объем работ, могут быть реализованы в связи с возникшей у владельца продукта необходимостью.

Исходя из того что журнал спринта не может быть изменен извне во время итерации, нужно выбирать ее длину, основываясь на стабильности требований. Если требования стабильны, меняются или дополняются редко, то можно выбрать шестинедельный цикл. В этом случае экономится время на переключение команды с активной разработки на планирование и демонстрационные митинги. Если требования часто меняются и дополняются, нужно отталкиваться от двухнедельного цикла, в любом случае длина итерации — это величина экспериментальная.

Основной упор методология Scrum делает на управление проектами и не задает никаких технических практик, что дает возможность использовать весь технический багаж, накопленный компанией. При внедрении Scrum чаще всего возникает две трудности. Первая — добиться активного участия от каждого разработчика и слаженной коллективной работы в команде. Похожую задачу решает тренер спортивной команды. Вторая — вовлечь поставщика требований в активное участие в проекте, заинтересовать его динамикой развития продукта, дать возможность быть активным болельщиком и спонсором команды. Несмотря на это, использование методологии Scrum в проекте MediaFACEonline (см. врезку) позволило за одиннадцать месяцев, с высоким качеством и в рамках бюджета реализовать большой объем функциональности, спецификации на которую отсутствовали на момент начала проекта. Согласно статистике использования сайта MediaFACEonline, накопленной за несколько месяцев работы, практически все реализованные функции активно используются посетителями, что в очередной раз подтверждает эффективность использования методологии Scrum в отношении обеспечения максимальной бизнес-ценности производимого продукта.

Благодаря Scrum была достигнута высокая сопровождаемость кода (возможность внесения изменений с минимальными трудозатратами) — стоимость изменений, вносимых в продукт, практически эквивалентна стоимости разработки аналогичных функций продукта в начале проекта, что редко достигается в так называемой «водопадной» модели производства, для которой характерен экспоненциальный рост стоимости изменений по мере выполнения проекта.