СОДЕРЖАНИЕ

Лекция 1. Инструментальное программное обеспечение. История развития ИСРПО. 4

Лекция 2. Методологии разработки ПО. 18

Лекция 3. Методология RUP. 28

Лекция 4. Этап логического проектирования ИС. Основные подходы при создании концептуальной модели. 35

Лекция 5. Описание функциональности разработки: нотация IDEF0. 44

Лекция 6. Описание функциональности разработки: диаграммы потоков данных. 50

Лекция 7. Описание функциональности разработки: нотация IDEF3. 59

Лекция 8. Проектирование с использованием метода «сущность-связь». 67

Лекция 9. Определение языка разработки, среды реализации, инструментов

разработки. 76

Лекция 10. Инструментальные средства и технологии Windows. SDK. 90

Лекция 11. Процедура физического проектирования – порядок, инструменты,

ресурсы, документы. 100

Лекция 12. Управление версиями ПП. 119

Лекция 13. Open TOOLs API. 143

Лекция 14. Использование и создание DLL. 153

Лекция 15. Разработка собственных компонентов. 165

Лекция 16. Инструментальные средства и методы расширения функциональности

среды разработки. 185

Лекция 17. Отладка программ. Инструменты. Методика отладки. 196

Лекция 18. Тестирование ПО. Средства автоматизированного тестирования. 219

Лекция 19. Документирование кода в Delphi. 230

Лекция 20. Создание системы помощи в программе. 238

Лекция 21. Защита приложения после компиляции. 277

Лекция 22. Автоматизация процесса сборки проекта. 283

Лекция 23. Создание инсталляции программы. 287

Лекция 1

**ТЕМА: Инструментальное программное обеспечение. История развития ИСРПО**.

Инструменты разработки программ делятся на аппаратные и программные.

**Предметом** междисциплинарного курса «Инструментальные средства разработки программного обеспечения» являются программные инструментальные средства, используемые для разработки и установки программ на компьютер, а также принципы их разработки и эксплуатации.

Разработка программного продукта (ПП) представляет множество связанных действий - таких как:

- создание модели данных и методики вычислений;

- описание функциональности;

- определение структуры данных; определение и описание способа реализации задачи (алгоритма решения);

- определение и описание интерфейса пользователя;

- определение средств поддержки ПП;

- спецификация задачи;

- написание текста программы;

- трансляция и отладка программы;

- связывание и подключение библиотек поддержки;

- создание среды выполнения; размещения исходного модуля и загрузка;

- создание встроенной помощи и документирование разработки;

- создание устанавливаемого (инсталляционного) пакета ПП.

В рамках Rational Unified Process (RUP) набор действий по разработке программ сконцентрирован в следующих этапах:

- определение требований;

- проектирование;

- программирование;

- тестирование;

- внедрение.

Для выполнения указанных работ разработан и постоянно пополняется огромный набор программ - инструментов, позволяющих формализовать и автоматизировать процесс разработки программ. Использование этих средств существенно сокращает сроки разработки и внедрения программных продуктов.

Рассмотрим, что представляет собой программа и программное обеспечение.

**Программа = задача + модель + алгоритм + структура данных**

Программа создается для того, чтобы решить определенную задачу: распознать преступника по фотографии в потоке людей, принять SMS-сообщение с одного мобильного телефона и передать его на другой и т.д.

Модель описывает то, что должна сделать программа для решения поставленной задачи, но не как она это должна сделать. Созданные в мышление человека модели предметной области могут относиться как к явлениям реального мира, например, движение космического объекта в гравитационном поле и атмосфере Земли, так и к идеальным понятиям, таким как интернет-магазин. Модель – это образ будущего инструмента, который позволит нам решить поставленную задачу.

Ключевым понятием в определении программы является задача. Как неразумно обсуждать техническую систему в отрыве от задачи, которую она решает, так бессмысленно рассматривать программу вне целей, для которых она создается. Для решения разных задач даже в одной предметной области будут созданы разные модели и разработаны разные программы.

Любой алгоритм, как мы знаем, есть последовательность предписаний, выполнив которые можно за конечное число шагов перейти от исходных данных к результату.

Итак,

**Определение 1:** Программа – это записанное на понятном некоторому вычислителю языке решение стоящей перед нами задачи.

**Определение 2:** Компьютерная программа - набор определенных команд, выполняющихся по заданному алгоритму.

**Определение 3:** Программа - данные, предназначенные для управления конкретными компонентами системы обработки информации в целях реализации определённого алгоритма. (ГОСТ 19781—90. ЕСПД. Термины и определения)

**Определение 4:**Программное обеспечение - совокупность программ системы обработки информации и программных документов, необходимых для их эксплуатации. (ГОСТ 19781—90. ЕСПД. Термины и определения)

Существенно, что ПО - это программы, предназначенные для многократного использования и применения разными пользователями. В связи с этим следует обратить внимание на ряд необходимых свойств ПО.

1. Необходимость документирования. По определению программы становятся ПО только при наличии документации. Конечный пользователь не может работать, не имея документации. Документация делает возможным тиражирование ПО и продажу его без его разработчика. По Бруксу ошибкой в ПО является ситуация, когда программное изделие функционирует не в соответствии со своим описанием, следовательно, ошибка в документации также является ошибкой в программном изделии.

2. Эффективность. ПО, рассчитанное на многократное использование (например, ОС, текстовый редактор и т.п.) пишется и отлаживается один раз, а выполняется многократно. Таким образом, выгодно переносить затраты на этап производства ПО и освобождать от затрат этап выполнения, чтобы избежать тиражирования затрат.

3. Надежность. В том числе:

- Тестирование программы при всех допустимых спецификациях входных данных
- Защита от неправильных действий пользователя
- Защита от взлома - пользователи должны иметь возможность взаимодействия с ПО только через легальные интерфейсы.

Готье: "Ошибки в системе возможны из-за сбоев аппаратуры, ошибок ПО, неправильных действий пользователя. Первые - неизбежны, вторые - вероятны, третьи - гарантированы". Появление ошибок любого уровня не должно приводить к краху системы. Ошибки должны вылавливаться диагностироваться и (если их невозможно исправить) превращаться в корректные отказы.

Системные структуры данных должны сохраняться безусловно.

Сохранение целостности пользовательских данных желательно.

4. Возможность сопровождения. Возможные цели сопровождения - адаптация ПО к конкретным условиям применения, устранение ошибок, модификация.

Во всех случаях требуется тщательное структурирование ПО и носителем информации о структуре ПО должна быть программная документация.

Адаптация во многих случаях м.б. передоверена пользователю - при тщательной отработке и описании сценариев инсталляции и настройки.

Исправление ошибок требует развитой сервисной службы, собирающей информацию об ошибках и формирующей исправляющие пакеты.

Модификация предполагает изменение спецификаций на ПО. При этом, как правило, должны поддерживаться и старые спецификации. Эволюционное развитие ПО экономит вложения пользователей.

Существуют различные направления программирования:

- искусственный интеллект;

- распознавание образов (изображения, звуки, анализ движения, ..);

- автоматное проектирование/программирование;

- теория алгоритмов ( например, методы кодирования, сжатия, шифрования, параллельная обработка информации);

- компьютерная лингвистика;

- объектно-ориентированное проектирование/программирование;

- математическое программирование;

- информационная безопасность (в том числе компьютерная вирусология, взлом/защита вычислительных систем)

- исследования в области обеспечения надёжности и эффективности разрабатываемого программного обеспечения;

- системы массового обслуживания (электронные платёжные системы, массовое распространение информации - электронная почта, медиа-сервисы, онлайн –клиники и магазины, поисковые системы, переводчики и т.д.);

- разработка приложений и игр на мобильные платформы (Android/ IPhone и т.д.)

- системное программирование;

- разработка игр и приложений для lin/win;

- разработка баз данных;

- низкоуровневое программирование.

При разработке программ применяются различные языки программирования. Выделяют:

- **низкоуровневый**язык программирования - близкий к программированию непосредственно в машинных кодах используемого реального или виртуального (например, Java, Microsoft .NET) процессора. (ассемблер)

- **высокоуровневый** язык программирования - разработанный для быстроты и удобства использования программистом. Они имитируют естественные языки, используя некоторые слова разговорного языка и общепринятые математические символы. Основная черта высокоуровневых языков - это абстракция, то есть введение смысловых конструкций, кратко описывающих такие структуры данных и операции над ними, описания которых на машинном коде (или другом низкоуровневом языке программирования) очень длинны и сложны для понимания. Языкам высокого уровня свойственно умение работать с комплексными структурами данных. В большинстве из них интегрирована поддержка строковых типов, объектов, операций файлового ввода-вывода и т. п. (C++, C#, Java, JavaScript, Python, PHP, Ruby, Perl, Паскаль, Delphi, Лисп)

Языки высокого уровня делятся на:

· **процедурные (алгоритмические)** (Basic, Pascal, C и др.), которые предназначены для однозначного описания алгоритмов; для решения задачи процедурные языки требуют в той или иной форме явно записать процедуру ее решения;

· **логические** (Prolog, Lisp и др.), которые ориентированы не на разработку алгоритма решения задачи, а на систематическое и

формализованное описание задачи с тем, чтобы решение следовало из составленного описания;

·**объектно-ориентированные** (Object Pascal, C++, Java и др.), в основе которых лежит понятие объекта, сочетающего в себе данные и действия над нами. Программа на объектно-ориентированном языке, решая некоторую задачу, по сути описывает часть мира, относящуюся к этой задаче. Описание действительности в форме системы взаимодействующих объектов естественнее, чем в форме взаимодействующих процедур.

- **ультра-высокоуровневый** язык программирования - характеризуется наличием дополнительных структур и объектов, ориентированных на прикладное использование. Использование его снижает временные затраты на разработку программного обеспечения и повышает качество конечного продукта за счет уменьшения объёма исходных кодов.

- **сверхвысокоуровневый** язык программирования - с очень высоким уровнем абстракции. В отличие от языков программирования высокого уровня, где описывается принцип «как нужно сделать», в сверхвысокоуровневых языках программирования описывается лишь принцип «что нужно сделать». Термин впервые появился в середине 1990-х годов для идентификации группы языков, используемых для быстрого прототипирования, написания одноразовых скриптов и подобных задач (пример Icon). К языкам сверхвысокого уровня также часто относят такие современные сценарные и декларативные (в частности функциональные) языки как Ruby и Haskell, а также Perl.

- **предметно-ориентированный язык** - большой класс языков сверхвысокого уровня, используемые для специфических приложений и задач.

Примеры:

· TeX/LaTeX для подготовки (компьютерной вёрстки) текстовых документов;

· Perl для манипулирования текстами;

· SQL для СУБД;

· Tcl/Tk для графического интерфейса пользователя;

· HTML и SGML для разметки документов;

· Verilog и VHDL для описания аппаратного обеспечения;

· Mathematica и Maple для символьных вычислений;

· AutoLisp для компьютерного моделирования (САПР);

· Prolog для задач, сформулированных в терминах исчисления предикатов;

· ML и Haskell для задач, сформулированных в терминах функций.

По мнению Валида Тахи, с позиции языково-ориентированного програмирования Microsoft Excel оказывается едва ли не наиболее широко применяемым в мире языком программирования.

Другими примерами предметно-ориентированных языков служат FoxPro, командные языки операционных систем (языки пакетных заданий, такие как JCL, языки интерактивной командной оболочки, такие как bash и batch), языки структурирования данных (XML, .ini, .conf), язык Вики-разметки, языки моделирования (UML, GPSS), Erlang для многопользовательских серверов, функционирующих в бесперебойном режиме. Следует отметить, что примеры не всегда являются показательными, некоторые предметно-ориентированные языки подвергаются критике.

При разработке программного продукта мы пользуемся различными инструментами, т.е. средствами, которые обеспечивают и облегчают выполнение поставленных задач.

Инструмента́льное програ́ммное обеспе́чение - программное обеспечение, предназначенное для использования в ходе проектирования, разработки и сопровождения программ, в отличие от прикладного и системного программного обеспечения.

Инструментальное ПО условно можно разбить на четыре группы:

1) **необходимое** – те, без которых невозможно в принципе получить исполняемый код;

К необходимым можно отнести:

- редакторы текстов (Word, WordPad, Блокнот);

- компиляторы и ассемблеры;

- компоновщики или редакторы связей (linkers);

2) **часто используемое** – средства, использования которых, в отличие от необходимых, можно избежать. Но без них процесс разработки весьма затрудняется и удлиняется;

Из часто используемых средств стоит назвать:

- утилиты автоматической сборки проекта;

- отладчики;

- программы создания инсталляторов;

- редакторы ресурсов;

- профилировщики;

- программы поддержки версий;

- программы создания файлов помощи (документации).

3) **специализированное** – используются в исключительных случаях, решают довольно специфичные задачи:

- программы отслеживания зависимостей;

- парсеры;

- дизассемблеры;

- декомпиляторы;

- hex-редакторы;

- программы отслеживания активности системы и изменений, происходящих в системе;

- программы-вериферы и контейнеры;

- программы для защиты разрабатываемого программного обеспечения (протекторы);

- CASE-средства для моделирования и проектирования ПО (BP Win, MS Visio, MS Net и др. средства логического проектирования (Ration Rose или любой UML-редактор и RBin, Coad)

- и т.д.

- специфическое– используются при разработке определенных видов программного обеспечения:

1) SDK, DDK, PDK, JDK;

2) API;

3) различные dll-библиотеки;

4) пользовательские компоненты.

5) технологические стандарты (Microsoft - OLE,ODBC, MAPI)

4) **интегрированные среды** – включают в себя большую часть выше перечисленных средств и обеспечивают их взаимосвязь.

Представители: Borland Delphi, Borland C++ Builder, Kylix (Borland Software Corporation), Power Builder(SY Base), Designer, Developer(Oracle), Visual Basic, Visual C++, Microsoft Visual Studio (.Net) (Microsoft Corp.), NuMega Driver Studio (NuMega), Eclipse (IBM).

Отдельно стоит отметить такие инструменты как методологии разработки ПО (например, RUP) и языки моделирования (например, ER-диаграммы, IDEF, DFD, UML), которые применяются в различных CASE-средствах, но могут использоваться и без них.

В каждом классе существуют огромное число продуктов, каждый со своими особенностями, достоинствами и недостатками.

Дадим краткую характеристику названным классам программ и приведем некоторые критерии оценки, по которым можно сравнивать программы из одного класса.

Но сначала укажем на характеристики, универсальные для всех программ:

- фирма-производитель, автор (зачастую имя производителя значит больше, чем все остальное).

- название продукта;

- номер последней версии;

- класс продукта, который установил для него производитель (например, HackersViewer, который включает в себя неплохой дизассемблер и редактор PE-файлов, поставляется просто как hex-редактор);

- тип дистрибьюции программы (с открытыми кодами/бесплатная (freeware)/условно-бесплатная (shareware)/платная) и стоимость;

- наличие и тип поддержки, ее стоимость;

- доступность и качество документации;

- простота и понятность интерфейса;

- наличие пробных версий (для платных программ);

- сайт программы и возможность ее скачки;

- размер дистрибутива и его состав;

- дополнительные (не основные) возможности, предоставляемые программой;

Теперь рассмотрим отдельно основные классы инструментов.

**Отладчики.**

Предназначены для поиска ошибок в других программах, ядрах операционных систем, SQL-запросах и других видах кода. Отладчик позволяет выполнять пошаговую трассировку, отслеживать, устанавливать или изменять значения переменных в процессе выполнения кода, устанавливать и удалять контрольные точки или условия остановки, отслеживать изменение состояния процессора во время работы программы и т.д.

Различают два основных типа отладчиков:

- отладчики пользовательского режима;

- отладчики режима ядра.

Первые могут лишь следить за работой программ пользовательского режима и не способны ни отслеживать системные вызовы, ни следить за работой ядра. Кроме того, для использования таких отладчиков программа должна быть соответствующим образом подготовлена (скомпилирована).

Отладчики же режима ядра, напротив, позволяют полностью контролировать работу системы, а, следовательно, и всех программ.

Характеристики:

- тип (режима ядра/пользовательский);

- поддержка символьной отладки (способность читать исходные коды программы и работать с ними). Набор поддерживаемых языков (сред/диалектов);

- набор отображаемой информации: регистры процессора, стек, память (режимы отображения содержимого памяти);

- поддерживаемые режимы отладки: пошаговый, с точками останова, с реакцией на события в системе;

- состав отслеживаемых событий в системе: аппаратные прерывания, обращения к драйверу (другому модулю ядра), вызов функции и т.д.

- (обычно для отладчиков режима ядра) требования к аппаратной поддержке, возможность работы на «живой» системе;

- возможность анализа файлов дампа.

Представители:

Отладчики пользовательского режима: Turbo Debugger (Borland Software Corporation), Cool Debugger (Wei Bao), W32Dasm, AQtime, FlexTracer, GNU Debugger.

Отладчики режима ядра: i386kd/alphakd/ia64kd и WinDbg (Microsoft Corporation) (для работы в “живую” требуют 2 машины. Для обхода этого ограничения существует надстройка LiveKd (MarkE. Russinovich)), SoftIce (NuMega).

**Программы создания инсталляторов**

Предназначены для создания дистрибутивов программ и пакетов программ.

Задачи, выполняемые подобными программами для различных платформ, могут сильно различаться. Мало того, с выходом WindowsInstaller и опубликования его API для платформы Win32 началось разделение программ на поддерживающие WindowsInstaller и использующие свои средств.

Как правило, все дистрибутивы имеют интерфейс программ-мастеров (т.е. пошаговое уточнение настроек). Кроме того, почти всегда имеется возможность удаления установленной программы.

Характеристики:

- ориентированны на использование Windows Installer или используют свои средства;

- возможность автоматического отслеживания зависимостей исполнимых файлов и разделяемых библиотек;

- наличие встроенного языка сценариев;

- возможность и пределы, в которых можно изменять поведения мастера инсталляции;

- возможность использования и поддержка национальных языков;

- функции, поддерживаемые в процессе установки (кроме копирования файлов):

- создание ключей реестра;

- регистрация COM-объектов;

- перезагрузка системы после или в процессе установки;

- возможность удаления установленной программы;

- возможность контроля версий устанавливаемой программы (перезапись, если необходимо) и разделяемых библиотек;

- возможность и степень сжатия дистрибутива;

- возможность создания дистрибутива, состоящего из одного, или заданного количества файлов;

Представители:

InstallShield (Install Shield Corp.), Wise InstallMaster Setup (Wise Solutions), Factory (Indigo Rose Corp.), Ghost Installer Studio; GkSetup (Gero Kuehn), Nullsoft Install System (Nullsoft), GP-Install (Quality Software Components), Little Setup Builder (http://www.ammasw.eboard.com), Inno Setup (http://www.gentee.com), Setup Generator (http://www.jrsoftware.org), Ghost Installer (http://www.ginstall.com).

**Дизассемблеры и декомпиляторы.**

Предназначены для получения исходного кода на языке программирования из исполняемого модуля.

Характеристики:

- поддерживаемые языки (компиляторы);

- возможность использования символьной информации о файле (отладочной и др.);

- возможность интерактивной работы с листингом (замены имен переменных и функций, отслеживания вызовов, модификация кода).

Представители: Interactive DisAssembler (Data Resource), Sourcer, Decafe Pro, DeDe.

**Программы отслеживания активности системы и изменений, происходящих в системе.**

Позволяют отслеживать действия программ по изменению реестра, файловой системы, вызовов системных сервисов и т.д. Следят за загруженностью системы в целом.

Характеристики:

- тип отслеживаемых изменений/активности;

- возможность протоколирования (логирования);

- возможность фильтрации получаемой информации;

- возможность уведомления.

Представители:

Microsoft: Spy++, Process Viewer, ApiMon, SysMon (для Win2000/XP – ActiveX компонентдля mmc).

Winternals Systems (Mark E. Russinovich): RegMon, FileMon, HandleEx.

Программы-вериферы и контейнеры.

Создают виртуальную среду для отдельных классов программ, в которой можно исследовать поведение программы).

Представители: Driver Verifier, ActiveX Control Test Container (Microsoft Corp.)

**Программы для защиты разрабатываемого программного обеспечения (протекторы).**

Позволяют создавать систему защиты ПО от несанкционированного копирования исполняемых файлов, непрофессионального реверс-инжиниринга, а также создавать регистрационные ключи, оценочные или демо-версии приложений (к примеру, программу, работающую 30 дней), шифровать и сжимать данные и т.п.

Представители: ASProtect, Obsidium, Armadillo, VMProtect,ORiEN

SDK.

SDK (от англ. software development kit) - комплект средств разработки, который позволяет специалистам по программному обеспечению создавать приложения для определённого пакета программ, программного обеспечения базовых средств разработки, аппаратной платформы, компьютерной системы, игровых консолей, операционных систем и прочих платформ.

Программист, как правило, получает SDK непосредственно от разработчика целевой технологии или системы. Часто SDK распространяется через Интернет. Многие SDK распространяются бесплатно для того, чтобы побудить разработчиков использовать данную технологию или платформу.

Поставщики SDK иногда подменяют слово «software» в словосочетании «software development kit» на более точное слово. Например, Microsoft и Apple предоставляют Driver Development Kit (DDK) для разработки драйверов устройств, PalmSource называет свой инструментарий для разработки PalmOS Development Kit (PDK), а Oracle -Java Development Kit (JDK).

Примеры SDK: Windows Phone SDK, Adobe Flex, DirectX, Eclipse, iPhone SDK, Java Development Kit, Opera Devices SDK, Source SDK.

**Краткий исторический обзор развития инструментальные средства разработки ПО**

**Этап 1: до середины 50-х.**

Основные затраты связаны с кодированием (в машинных кодах). Появляются автокоды (языки с использованием мнемонических обозначений команд) и трансляторы с них (ассемблеры).

Реализуются возможности раздельной компиляции и перемещаемости программ. Появляются загрузчики и компоновщики программ.

**Этап 2: середина 50-х – середина 60-х гг.**

Увеличиваются размеры программ, выявляется разрыв между понятиями проблемных областей и машинно-ориентированных языков. Появляются различные языки высокого уровня (алгоритмические, универсальные):

- Fortran (1954-1957);

- Algol-60 (1958-1960);

- Cobol (1959-1961);

- Lisp (1959);

- Basic (1964);

- PL/1 (1964).

и трансляторы с них (компиляторы). Изобретаются и опробуются почти все основные типы данных, операции над ними, управляющие структуры и способы изображения их в программах, различные варианты параметризации подпрограмм.

**Этап 3: середина 60-х – начало 70-х гг.**

Резко увеличиваются размеры ПО, происходит переход к коллективному характеру работ. Повышаются требования к ПО вследствие перехода к товарному производству.

Изменяется соотношение затрат на разработку ПО (40% и более тратится на отладку, проектирование и документирование), кодирование – один из самых простых видов работ. Используются и создаются "большие" языки программирования – ПЛ/1, АЛГОЛ-68, СИМУЛА-67, обобщающие и интегрирующие ранее найденные решения.

Появляются развитые системы программирования с оптимизирующими и отладочными трансляторами, макробиблиотеками, библиотеками стандартных программ, специализированных текстовыми редакторами, средствами анализа и диалоговой отладки в терминах входного

языка. Разрабатываются развитые операционные системы, первые СУБД, многочисленные системы автоматизации документирования, системы управления программной конфигурацией (отслеживания модификаций и сборки версий ПО).

**Этап 4 (“этап кризиса в развитии ПО”): начало 70-х–середина 70-х гг.**

Несмотря на развитие инструментальных средств, производительность труда программистов не растёт. Более того, вследствие повышения требований к ПО и нелинейного роста его сложности, производительность труда падает. Срываются сроки разработки ПО, растёт его стоимость, непредсказуемо его качество, не срабатывают традиционные методы (предоставление дополнительных человеческих и материальных ресурсов), что характеризуется как "кризис ПО".

Получают признание методологии структурного программирования (Дейкстра, 1968г.), формируются основы технологии программирования (язык Паскаль (Н.Вирт), 1971г.).

**Этап 5:1976г.– наше время. Этап посткризисного развития инструментальных средств.**

1976г. – публикация работы Боэма, где вводится понятие жизненного цикла ПО и указывается, что основные затраты приходятся не на разработку, а на сопровождение программ.

Языки программирования:

- C (начало 1970-х, впервые достаточно полно описан в 1978 г.);

- Modula-2 (1978 г., развитие – язык Oberon (1988));

- Ada (1980);

- Prolog (1972 г., распространение получил с 1980 г.);

- Smalltalk (1970-е годы, в 1980 был представлен как Smalltalk-80);

- C++ (начало 1980-х гг., название – 1983, в привычном сегодня виде существует с 1990 г.);

- Java (версия Java 1.0 – 1996 г., Java 2.0 – 1998, Java 5 – 2004...);

- C# (1998–2001, версия 1.0 – 2000–2002, версия 2.0 – 2003-2005, версия 3.0 – 2004–2008, версия 4.0 – 2008–2010).

Развиваются интегрированные инструментальные среды разработки программ. Получает признание объектно-ориентированный подход к проектированию и программированию. Разрабатываются программы, поддерживающие создание ПО на каждом этапе.

Лекция 2

**ТЕМА: Методологии разработки ПО.**

Рассмотрим понятия методологии, метода и средства.

Определение 1: **Метод** (от греч. methodos - способ исследования или познания, теория или учение) - прием или система приемов практического осуществления чего-нибудь в какой-либо предметной области, совокупность приемов или операций практического или теоретического освоения действительности, подчиненных решению конкретных задач.

Метод включает **средства**- с помощью чего осуществляется действие и **способы**- каким образом осуществляется действие.

Определение 2: **Методология**- это система принципов, а также совокупность идей, понятий, методов, способов и средств, определяющих стиль разработки программного обеспечения.

Методология - это реализация стандарта. Сами стандарты лишь говорят о том, что должно быть, оставляя свободу выбора и адаптации.

Конкретные вещи реализуются через выбранную методологию. Именно она определяет, как будет выполняться разработка. Существует много успешных методологий создания программного обеспечения. Выбор конкретной методологии зависит от размера команды, от специфики и сложности проекта, от стабильности и зрелости процессов в компании и от личных качеств сотрудников.

Методологии представляют собой ядро теории управления разработкой программного обеспечения.

В зависимости от используемой модели жизненного цикла методологии делятся на:

- водопадные (каскадные);

- итерационные (спиральные).

Также существует и более общая классификация на:

- прогнозируемые;

- адаптивные.

**Прогнозируемые методологии** фокусируются на детальном планировании будущего. Известны запланированные задачи и ресурсы на весь срок проекта. Команда с трудом реагирует на возможные изменения. План оптимизирован исходя из состава работ и существующих требований. Изменение требований может привести к существенному изменению плана, а также дизайна проекта. Часто создается специальный комитет по «управлению изменениями», чтобы в проекте учитывались только самые важные требования.

**Адаптивные методологии** нацелены на преодоление ожидаемой неполноты требований и их постоянного изменения. Когда меняются требования, команда разработчиков тоже меняется. Команда, участвующая в адаптивной разработке, с трудом может предсказать будущее проекта. Существует точный план лишь на ближайшее время. Более удаленные во времени планы существуют лишь как декларации о целях проекта, ожидаемых затратах и результатах.

**Каскадная разработка** или модель водопада (англ. waterfall model) - модель процесса разработки программного обеспечения, в которой процесс разработки выглядит как поток, последовательно проходящий фазы анализа требований, проектирования, реализации, тестирования, интеграции и поддержки.

Принципиальной особенностью каскадного подхода является: ***переход на следующую стадию осуществляется только после того, как будет полностью завершена работа на текущей стадии, и возвратов на пройденные стадии не предусматривается***. Каждая стадия заканчивается получением некоторых результатов, которые служат в качестве исходных данных для следующей стадии (рис. 1).

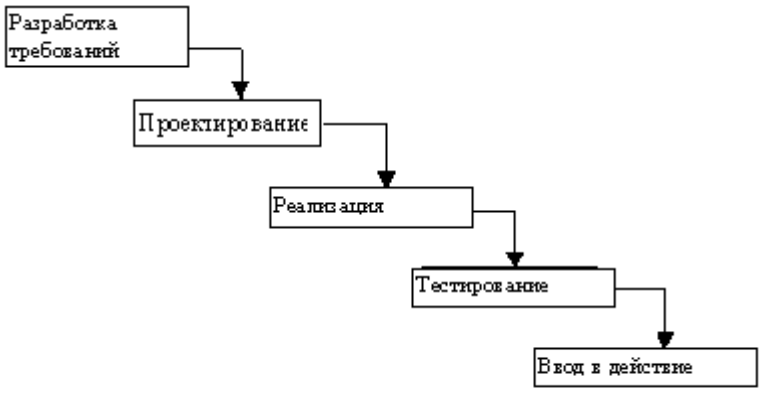



Рис. 1. Каскадная модель жизненного цикла.

Каждая стадия завершается выпуском комплекта документации, достаточной для того, чтобы разработка могла быть продолжена другой командой разработчиков. Критерием качества разработки при таком подходе является точность выполнения спецификаций технического задания.

Преимущества применения каскадного способа:

- на каждой стадии формируется законченный набор проектной документации, отвечающий требованиям полноты и согласованности;

- выполняемые в логической последовательности стадии работ позволяют планировать сроки завершения всех работ и соответствующие затраты.

Каскадный подход хорошо зарекомендовал себя при построении электронных информационных систем, для которых в самом начале разработки можно достаточно точно и полно сформулировать все требования, с тем, чтобы предоставить разработчикам свободу реализовать их технически как можно лучше.

В то же время этот подход обладает рядом недостатков, вызванных, прежде всего тем, что реальный процесс создания программного обеспечения никогда полностью не укладывается в такую жесткую схему. Процесс создания ПО носит, как правило, итерационный характер: результаты очередной стадии часто вызывают изменения в проектных решениях, выработанных на предыдущих стадиях. Таким образом, постоянно возникает потребность в возврате к предыдущим стадиям и уточнении или пересмотре ранее принятых решений (рис. 2). Изображенную схему можно отнести к отдельной модели - модели с промежуточным контролем, в которой межстадийные корректировки обеспечивают большую надежность по сравнению с каскадной моделью, хотя увеличивают весь период разработки.

Основным недостатком каскадной модели является существенное запаздывание с получением результатов и, как следствие, высокий риск создания системы, не удовлетворяющей изменившимся потребностям пользователей. Это объяснятся двумя причинами:

- пользователи не в состоянии сразу изложить все свои требования и не могут предвидеть, как они изменятся в ходе разработки;

- за время разработки могут произойти изменения во внешней среде, которые повлияют на требования к системе.


Рис. 2. Каскадная модель ЖЦ на практике.

В рамках каскадного подхода требования к разрабатываемому продукту фиксируются в виде технического задания на все время его создания, а согласование получаемых результатов с пользователями производится только в точках, планируемых после завершения каждой стадии (при этом возможна корректировка результатов по замечаниям пользователей, если они не затрагивают требования, изложенные в техническом задании). Таким образом, пользователи могут внести существенные замечания только после того, как работа над системой будет полностью завершена. Пользователи могут получить систему, не удовлетворяющую их потребностям. В результате приходится начинать новый проект, который может постигнуть та же участь.

Для преодоления перечисленных проблем в середине 80-х годов была предложена спиральная модель жизненного цикла (рис.3).

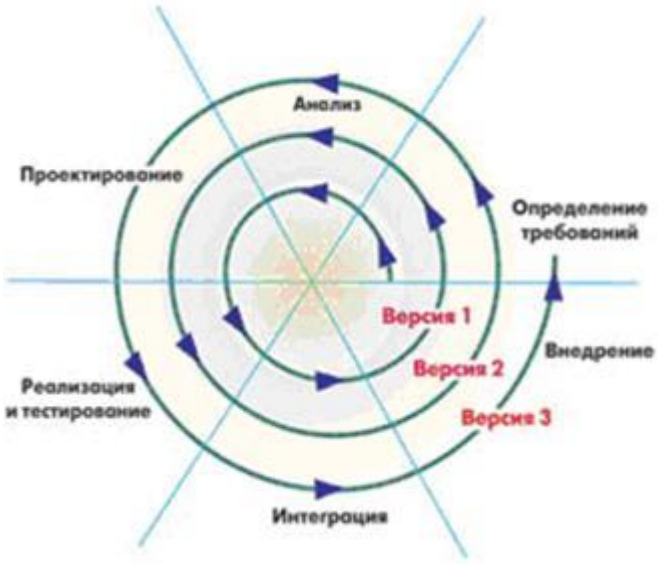



Рис. 3. Спиральная (итерационная) модель ЖЦ.

Ее принципиальной особенностью является следующее: ***прикладное ПО создается не сразу, как в случае каскадного подхода, а по частям с использованием метода прототипирования***.

Под **прототипом** понимается действующий программный компонент, реализующий отдельные функции и внешние интерфейсы разрабатываемого ПО. Создание прототипов осуществляется в несколько итераций, или витков спирали. Каждая итерация соответствует созданию фрагмента или версии ПО, на ней уточняются цели и характеристики проекта, оценивается качество полученных результатов и планируются работы следующей итерации. На каждой итерации производится тщательная оценка риска превышения сроков и стоимости проекта, чтобы определить необходимость выполнения еще одной итерации, степень полноты и точности понимания требований к системе, а также целесообразность прекращения проекта.

Спиральная модель избавляет пользователей и разработчиков от необходимости точного и полного формулирования требований к системе на начальной стадии, поскольку они уточняются на каждой итерации. Таким образом, углубляются и последовательно конкретизируются детали проекта, и в результате выбирается обоснованный вариант, который доводится до реализации.

Спиральная модель - классический пример применения эволюционной стратегии конструирования. Спиральная модель (автор Барри Боэм, 1988) базируется на лучших свойствах классического жизненного цикла и макетирования, к которым добавляется новый элемент - анализ риска, отсутствующий ранее.

Спиральная модель определяет четыре действия, представляемые отдельными секторами 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 проектирования). Для дальнейшего определения проблемных и уточненных требований может быть использовано моделирование. Заказчик оценивает инженерную (конструкторскую) работу и вносит предложения по модификации. Следующая фаза планирования и анализа риска базируется на предложениях заказчика. В каждом цикле по спирали результаты анализа риска формируются 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.

При итеративном способе недостающую часть работы можно выполнять на следующей итерации. Главная же задача - как можно быстрее показать пользователям системы работоспособный продукт, тем самым активизируя процесс уточнения и дополнения требований.

Спиральная модель не исключает каскадного подхода на завершающих стадиях проекта в тех случаях, когда требования к системе оказываются полностью определенными.

Основная проблема спирального цикла - определение момента перехода на следующую стадию. Для ее решения необходимо ввести временные ограничения на каждую из стадий ЖЦ. Переход осуществляется в соответствии с планом, даже если не вся запланированная работа закончена. План составляется на основе статистических данных, полученных на предыдущих проектах, и личного опыта разработчиков.

Достоинства спиральной модели:

- наиболее реально (в виде эволюции) отображает разработку программного обеспечения;

- позволяет явно учитывать риск на каждом витке эволюции разработки;

- включает шаг системного подхода в итерационную структуру разработки;

- использует моделирование для уменьшения риска и совершенствования программного изделия.

Недостатки спиральной модели:

- новизна (отсутствует достаточная статистика эффективности модели);

- повышенные требования к заказчику;

- трудности контроля и управления временем разработки.

На сегодняшний день можно выделить следующие итеративные методологии разработки программного обеспечения:

- Rational Unified Process (RUP)

- Гибкие методологии разработки (SCRUM, KANBAN, DSDM, MSF, ALM, XP)

**Гибкая методология разработки** (англ. Agile software development).

Большинство гибких методологий нацелены на минимизацию рисков, путём сведения разработки к серии коротких циклов, называемых *итерациями*, которые обычно длятся одну-две недели. Каждая итерация сама по себе выглядит как программный проект в миниатюре, и включает все задачи, необходимые для выдачи мини-прироста по функциональности: планирование, анализ требований, проектирование, кодирование, тестирование и документирование. Хотя отдельная итерация, как правило, недостаточна для выпуска новой версии продукта, подразумевается, что гибкий программный проект готов к выпуску в конце каждой итерации. По окончании каждой итерации, команда выполняет переоценку приоритетов разработки.

Agile-методы делают упор на непосредственное общение лицом к лицу. Большинство agile-команд расположены в одном офисе. Как минимум она включает и «заказчиков» (заказчики которые определяют продукт, также это могут быть менеджеры продукта, бизнес-аналитики или клиенты). Офис может также включать тестировщиков, дизайнеров интерфейса, технических писателей и менеджеров.

Одной из наиболее известных и передовых гибких методик является методология SCRUM.

**SCRUM**- методология, предназначенная для небольших команд (до 10 человек). Весь проект делится на итерации (спринты) продолжительностью 30 дней каждый. Выбирается список функций системы, которые планируется реализовать в течение следующего спринта. Самые важные условия - неизменность выбранных функций во время выполнения одной итерации и строгое соблюдение сроков выпуска очередного релиза, даже если к его

выпуску не удастся реализовать весь запланированный функционал. Руководитель разработки проводит ежедневные 20 минутные совещания, которые так и называют — scrum, результатом которых является определение функции системы, реализованных за предыдущий день, возникшие сложности и план на следующий день. Такие совещания позволяют постоянно отслеживать ход проекта, быстро выявлять возникшие проблемы и оперативно на них реагировать.

**KANBAN** – гибкая методология разработки программного обеспечения, ориентированная на задачи.

Основные правила:

- визуализация разработки:

о разделение работы на задачи;

о использование отметок о положение задачи в разработке;

- ограничение работ, выполняющихся одновременно, на каждом этапе разработки;

- измерение времени цикла (среднее время на выполнение одной задачи) и оптимизация процесса.

Преимущества KANBAN:

- уменьшение числа параллельно выполняемых задач значительно уменьшает время выполнения каждой отдельной задачи;

- быстрое выявление проблемных задач;

- вычисление времени на выполнение усредненной задачи.

**DYNAMIC SYSTEM DEVELOPMENT METHOD** (DSDM) появился в результате работы консорциума из 17 английских компаний. Целая организация занимается разработкой пособий по этой методологии, организацией учебных курсов, программ аккредитации и т.п. Кроме того, ценность DSDM имеет денежный эквивалент.

Все начинается с изучения осуществимости программы и области ее применения. В первом случае, вы пытаетесь понять, подходит ли DSDM для данного проекта. Изучать область применения программы предполагается на короткой серии семинаров, где программисты узнают о той сфере бизнеса, для которой им предстоит работать. Здесь же обсуждаются основные положения, касающиеся архитектуры будущей системы и план проекта.

Далее процесс делится на три взаимосвязанных цикла: цикл функциональной модели отвечает за создание аналитической документации и прототипов, цикл проектирования и конструирования — за приведение системы в рабочее состояние, и наконец, последний цикл — цикл реализации — обеспечивает развертывание программной системы.

Базовые принципы, на которых строится DSDM:

- активное взаимодействие с пользователями;

- частые выпуски версий;

- самостоятельность разработчиков в принятии решений;

- тестирование в течение всего цикла работ.

Как и большинство других гибких методологий, DSDM использует короткие итерации, продолжительностью от двух до шести недель каждая. Особый упор делается на высоком качестве работы и адаптируемости к изменениям в требованиях.

**MICROSOFT SOLUTIONS FRAMEWORK** (MSF) - методология разработки программного обеспечения, предложенная корпорацией Microsoft. MSF опирается на практический опыт Microsoft и описывает управление людьми и рабочими процессами в процессе разработки решения.

Базовые концепции и принципы модели процессов MSF:

- единое видение проекта - все заинтересованные лица и просто участники проекта должны чётко представлять конечный результат, всем должна быть понятна цель проекта;

- управление компромиссами - поиск компромиссов между ресурсами проекта, календарным графиком и реализуемыми возможностями;

- гибкость – готовность к изменяющимся проектным условиям;

- концентрация на бизнес-приоритетах - сосредоточенность на той отдаче и выгоде, которую ожидает получить потребитель решения;

- поощрение свободного общения внутри проекта;

- создание базовых версии — фиксация состояния любого проектного артефакта, в том числе программного кода, плана проекта, руководства пользователя, настройки серверов и последующее эффективное управление изменениями, аналитика проекта.

MSF предлагает проверенные методики для планирования, проектирования, разработки и внедрения успешных IT-решений. Благодаря своей гибкости, масштабируемости и отсутствию жестких инструкций MSF способен удовлетворить нужды организации или проектной группы любого размера. Методология MSF состоит из принципов, моделей и дисциплин по управлению персоналом, процессами, технологическими элементами и связанными со всеми этими факторами вопросами, характерными для большинства проектов.

**Application Lifecycle Management** (ALM) - разработанная и поддерживаемая компанией Borland.

**Extreme Programming** (XP) -экстремальное программирование, поддерживаемое открытым сообществом независимых разработчиков.

**Подход RAD**

Одним из возможных подходов к разработке прикладного ПО в рамках спиральной модели ЖЦ является получивший широкое распространение способ так называемой ***быстрой разработки приложений, или RAD*** (Rapid Application Development). RAD предусматривает наличие трех составляющих:

- небольших групп разработчиков (3-7 чел.), выполняющих работы по проектированию отдельных подсистем ПО. Это обусловлено требованием максимальной управляемости коллектива;

- короткого, но тщательного проработанного производственного графика (до 3 месяцев);

- повторяющегося цикла, при котором разработчики по мере того, как приложение начинает приобретать форму, запрашивают и реализуют в продукте требования, полученные в результате взаимодействия с заказчиком.

Команда разработчиков должна представлять собой группу профессионалов, имеющих опыт в проектировании, программировании и тестировании ПО, способных хорошо взаимодействовать с конечным пользователем и трансформировать их предложения в рабочие прототипы.

ЖЦ ПО в соответствии с подходом RAD включает 4 стадии:

**1. Анализ и планирование требований** предусматривает действия:

- определение функций, которые должна выполнять система;

- выделение наиболее приоритетных функций, требующих проработки в первую очередь;

- описание информационных потребностей. Формулирование требований к системе осуществляется в основном силами пользователей под руководством специалистов-разработчиков.

Кроме того, на данной стадии реализуются следующие задачи:

- ограничивается масштаб времени;

- устанавливаются временные рамки для каждой из последующих стадий. Определяется сама возможность реализации проекта в заданных рамках финансирования, на имеющихся аппаратных средствах.

Результатом стадии должны быть список расставленных по приоритету функций будущего ПО ЭИС и предварительные модели ПО.

**2. Проектирование.**На этой стадии часть пользователей принимает участие в техническом проектировании системы под руководством специалистов-разработчиков. Для быстрого получения работающих прототипов приложений используются соответствующие инструментальные средства (CASE - средства). Пользователи, непосредственно взаимодействуя с разработчиками, уточняют и дополняют требования к системе, которые не были выявлены на предыдущей стадии:

- более детально рассматриваются процессы системы;

- при необходимости для каждого элементарного процесса создается частичный прототип: экранная форма, диалог, отчет, устраняющий неясности и неоднозначности;

- устанавливаются требования разграничения доступа к данным;

- определяется состав необходимой документации.

После детального определения состава процессов оценивается количество так называемых функциональных точек разрабатываемой системы и принимается решение о разделении ЭИС на подсистемы, поддающиеся реализации одной командой разработчиков за приемлемое время (до 3 месяцев). Под ***функциональной точкой*** понимается любой из следующих элементов разрабатываемой системы:

- входной элемент приложения (входной документ или экранная форма);

- выходной элемент приложения (отчет, документ, экранная форма)

- запрос (пара «вопрос/ответ»);

- логический файл (совокупность записей данных, используемых внутри приложения);

- интерфейс приложения (совокупность записей данных, передаваемых другому приложению или получаемых от него).

Далее проект распределяется между различными командами разработчиков. Опыт показывает, что для повышения эффективности работ необходимо разбить проект на отдельные слабо связанные по данным и функциям подсистемы. Реализация подсистем должна выполняться отдельными группами специалистов. При этом необходимо обеспечить координацию ведения общего проекта и исключить дублирование результатов работ каждой проектной группы, которое может возникнуть в силу наличия общих данных и функций.

Результатом данной стадии должно быть:

- общая информационная модель системы;

- функциональные модели системы в целом и подсистем, реализуемых отдельными командами разработчиков;

- точно определенные интерфейсы между автономно разрабатываемыми подсистемами;

- построенные прототипы экранных форм, отчетов, диалогов.

**3. Реализация.**На этой стадиивыполняется непосредственно сама быстрая разработка приложения.

Разработчики производят итеративное построение реальной системы на основе полученных на предыдущей стадии моделей, а также требований нефункционального характера (требования к надежности, производительности и т.д.).

Пользователи оценивают получаемые результаты и вносят коррективы, если в процессе разработки система перестает удовлетворять определенным ранее требованиям. Тестирование системы осуществляется в процессе разработки.

После окончания работ каждой отдельной команды разработчиков производится постепенная интеграция данной части системы с остальными, формируется полный программный код, выполняется тестирование совместной работы данной части приложения, а затем тестирование системы в целом. Реализация системы завершается выполнением следующих работ:

- осуществляется анализ использования данных и определяется необходимость их распределения;

- Производится физическое проектирование базы данных;

- формируются требования к аппаратным ресурсам;

- устанавливаются способы увеличения производительности;

- завершается разработка документации проекта.

Результатом стадии является готовая система, удовлетворяющая всем согласованным требованиям.

**4. Внедрение.**На этой стадиипроизводится обучение пользователей, организационные изменения и параллельно с внедрением новой системы продолжается эксплуатация существующей системы (до полного внедрения новой). Так как стадия реализации достаточно продолжительна, планирование и подготовка к внедрению должны начинаться заранее, как правило, на стадии проектирования системы.

Приведенная схема разработки ЭИС не является абсолютной. Возможны различные варианты, зависящие, например, от начальных условий, в которых ведется разработка:

- разрабатывается совершенно новая система;

- уже было проведено обследование организации и существует модель ее деятельности;

- в организации уже существует некоторая ЭИС, которая может быть использована в качестве начального прототипа или должна быть интегрирована с разрабатываемой системой.

Следует отметить, что подход RAD, как и любой другой подход, не может претендовать на универсальность. Он хорош в первую очередь для относительно небольших проектов, разрабатываемых для конкретного заказчика. Если же разрабатывается крупномасштабная система (например, масштаба отрасли), которая не является законченным продуктом, а представляет собой комплекс программных компонентов, адаптируемых к программно-аппаратным платформам, системам управления базами данных (СУБД), средствам телекоммуникаций, то на первый план выступают такие показатели проекта как управляемость и качество, которые могут войти в противоречие с простотой и скоростью разработки. Для таких проектов необходимы высокий уровень планирования и жесткая дисциплина

проектирования, строгое следование заранее разработанным протоколам и интерфейсам, что снижает скорость разработки.

Подход RAD не применим для построения сложных расчетных программ, операционных систем.

Не годится этот подход и для приложений, в которых отсутствует ярко выраженная интерфейсная часть, наглядно определяющая логику работы системы и приложений, от которых зависит безопасность людей (например программа управления самолетом или атомной станцией), так как итеративный подход предполагает, что первые несколько версий наверняка не будут полностью работоспособны, что в данном случае исключается.

Итак, перечислим основные принципы подхода RAD.

- Разработка приложений итерациями.

- Необязательность полного завершения работ на каждой стадии ЖЦ ПО.

- Обязательность вовлечения пользователей в процесс разработки ЭИС.

- Целесообразность применения CASE - средств, обеспечивающих целостность проекта и генерацию кода приложений.

- Целесообразность применения средств управления конфигурацией, облегчающих внесение изменений в проект и сопровождение готовой системы.

- Использование прототипирования, позволяющее полнее выяснить и удовлетворить потребности пользователей.

- Тестирование и развитие проекта, осуществляемые одновременно с разработкой.

- Ведение разработки немногочисленной хорошо управляемой командой профессионалов.

- Грамотное руководство разработкой системы, четкое планирование и контроль выполнения работ.

Таким образом, существует множество различных методологий и подходов при разработке программного обеспечения, они не универсальны и описываются различными принципами. Выбор методологии и подхода при разработке конкретного проекта зависит от предъявляемых требований.

Лекция 3

**ТЕМА:Методология RUP.**

**Rational Unified Process** (RUP) - методология разработки программного обеспечения, созданная и поддерживаемая компанией Rational Software. Это:

- новый подход к разработке ПС, основанный на использовании лучших практических методов, успешно зарекомендовавших себя во многих проектах разработки ПС по всему миру;

- четко определенный процесс (технологическая процедура), описывающий структуру жизненного цикла проекта, роли и ответственности отдельных исполнителей, выполняемые ими задачи и используемые в процессе разработки модели, отчеты и т.д.;

- готовый продукт, предоставляемый в виде веб-сайта, содержащего все необходимые модели и документы с описанием процесса.

В основе методологии лежат 6 основных принципов:

- Ранняя идентификация и непрерывное (до окончания проекта) устранение основных рисков.

- Концентрация на выполнении требований заказчиков к исполняемой программе (анализ и построение модели прецедентов).

- Ожидание изменений в требованиях, проектных решениях и реализации в процессе разработки.

- Компонентная архитектура, реализуемая и тестируемая на ранних стадиях проекта.

- Постоянное обеспечение качества на всех этапах разработки проекта (продукта).

- Работа над проектом в сплочённой команде, ключевая роль в которой принадлежит архитекторам.

Использование методологии RUP направлено на итеративную модель разработки. Особенность методологии состоит в том, что степень формализации может меняться в зависимости от потребностей проекта. Можно по окончании каждого этапа и каждой итерации создавать все требуемые документы и достигнуть максимального уровня формализации, а

можно создавать только необходимые для работы документы, вплоть до полного их отсутствия. За счет такого подхода к формализации процессов методология является достаточно гибкой и широко популярной. Данная методология применима как в небольших и быстрых проектах, где за счет отсутствия формализации требуется сократить время выполнения проекта и расходы, так и в больших и сложных проектах, где требуется высокий уровень формализма, например, с целью дальнейшей сертификации продукта. Это преимущество дает возможность использовать одну и ту же команду разработчиков для реализации различных по объему и требованиям проектов.

В конце каждой итерации (в идеале продолжающейся от 2 до 6 недель) проектная команда должна достичь запланированных на данную итерацию целей, создать или доработать проектные артефакты и получить промежуточную, но функциональную версию конечного продукта. Итеративная разработка позволяет быстро реагировать на меняющиеся требования, обнаруживать и устранять риски на ранних стадиях проекта, а также эффективно контролировать качество создаваемого продукта.

**IBM Rational Unified Process** — процесс, управляемый на основе прецедентов. Это означает, что в качестве метода описания функциональных требований к системе, а также в качестве естественной единицы для дальнейшего планирования и оценки выполнения работ используются сценарии использования. Сценарии использования позволяют легко выявлять реальные потребности будущих пользователей системы и отслеживать полноту описания этих требований. Они гарантируют выполнения требований заказчика к ПС. Кроме того, использование завершенных сценариев в качестве единицы измерения прогресса помогает избежать неадекватной оценки степени выполнения проекта исполнителем.

**IBM Rational Unified Process** предполагает разработку, реализацию и тестирование архитектуры на самых ранних стадиях выполнения проекта. Такой подход позволяет устранять самые опасные риски, связанные с архитектурой, на ранних стадиях разработки. Благодаря ему удается избежать существенных переработок в последний момент, если вдруг выяснится, что выбранное решение не обеспечивает, к примеру, выполнение требований к производительности системы.

**RUP- Четко определенный процесс**

RUP создавался по методике, используемой при проектировании ПС. В частности, моделирование производилось с помощью Software Process Engineering Metamodel (SPEM) -стандартамоделированияпроцессов, основанногона Unified Modeling Language (UML).

Особенностью RUP является то, что в результате работы над проектом создаются и совершенствуются модели. Вместо создания громадного

количества бумажных документов, RUP опирается на разработку и развитие семантически обогащенных моделей, всесторонне представляющих разрабатываемую систему. RUP – это руководство по тому, как эффективно использовать UML. Стандартный язык моделирования, используемый всеми членами группы, делает понятными для всех описания требований, проектирование и архитектуру системы.

У процесса есть два измерения:

**Динамическая структура**. Горизонтальное измерение представляет собой динамическую структуру или временное измерение процесса. Оно показывает, как процесс, выраженный в форме циклов, фаз, итераций и вех, развертывается в ходе жизненного цикла проекта.

**Статическая структура.** Вертикальное измерение представляет собой статическую структуру процесса. Оно описывает, как элементы процесса - задачи, дисциплины, артефакты и роли - логически группируются в дисциплины или рабочие процессы.

**Моделирование бизнес-процессов** применяется с тем, чтобы разобраться в структуре исследуемой предметной области, обеспечить единство понимания основных автоматизируемых процессов среди всех участников проекта и определить высокоуровневые требования, которые должны быть реализованы в ходе проекта.

**Управление требованиями** позволяет прийти к соглашению с заказчиками и конечными пользователями, определить, что должна уметь делать создаваемая система, предоставить более четкие инструкции участникам проекта о возможностях системы, создать базу для успешного планирования работ в проекте и оценки его статуса в любой момент жизненного цикла.

**Анализ и проектирование** служат для последовательного преобразования выявленных требований к системе в спецификации особого вида, которые описывают, как следует конкретно реализовать конечный продукт. Следует при этом делать различия между анализом и проектированием. Основное из них состоит в том, что спецификации анализа не зависят от конкретной платформы и технологии, для которой осуществляется создание ИС. А спецификации проектирования являются точным представлением проектируемой системы, часто позволяя автоматизировать процесс генерации на их основе программного кода.

**Реализация** необходима для выявления порядка организации программного кода в терминах отдельных подсистем, преобразования исходного кода в выполняемые компоненты, тестирования созданных компонентов и интеграции отдельных компонентов в подсистемы и системы.

**Тестирование** позволяет определять и контролировать качество создаваемых продуктов, следить за тем, насколько качественно осуществлена интеграция компонентов и подсистем, все ли требования к системе реализованы и все ли выявленные ошибки устранены до того, как система будет развернута на оборудовании конечного пользователя.

**Развертывание** является процессом, в ходе которого осуществляется доставка разрабатываемого продукта к конечному пользователю. В ходе данного процесса производится новый выпуск системы, распространение ПО, его установка на стороне конечного пользователя, обучение последнего навыкам эффективной работы с поставленным ПО, предоставление услуг по технической поддержке, бета-тестирование и т. п.

**Конфигурационное управление и управление изменениями** позволяет организовать эффективную работу с артефактами проекта, контролировать и управлять доступом к ним, вести историю изменений, обеспечить эффективное взаимодействие участников проекта, как в простых командах, так и в распределенных, находящихся на большом удалении друг от друга.

**Управление проектом** включает в себя непосредственное формирование условий для эффективного хода всего проекта, определение руководств и руководящих принципов для планирования, формирования команды и мониторинга проекта, выявление и управление рисками, организацию работы участников проекта, формирование бюджета, планирование фаз и итераций.

**Управление средой** позволяет осуществить поддержку всех участников проекта. В эту поддержку входят выбор инструментария и его приобретение, настройка и установка, конфигурирование процесса, доработка и адаптация методологии, используемой для ведения проекта, обучение

Полный жизненный цикл разработки продукта состоит из четырех фаз, каждая из которых включает в себя одну или несколько итераций:

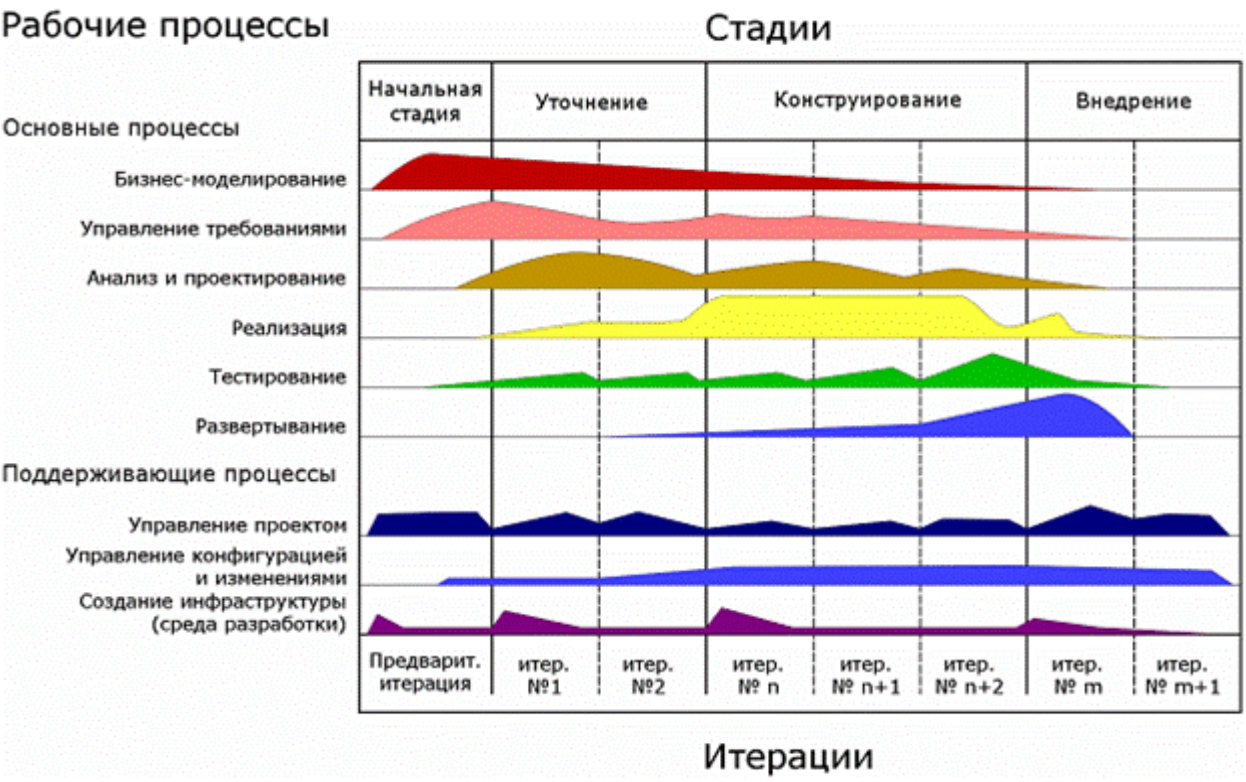



Рис. 4. Графическое представление процесса разработки по RUP

Переход с фазы на фазу возможен только после выполнения задач фазы ипредставляет собой контрольную точку процесса.

**Внедрение (Transition)**

В фазе «Внедрение» создается финальная версия продукта и передается от разработчика к заказчику. Это включает в себя программу бета-тестирования, обучение пользователей, а также определение качества продукта. В случае, если качество не соответствует ожиданиям пользователей или критериям, установленным в фазе Начало, фаза Внедрение повторяется снова. Выполнение всех целей означает достижение вехи готового продукта и завершение полного цикла разработки.

**IBM Rational Unified Process** представляет собой готовый продукт. Он состоит из нескольких частей. Это:

- размещаемая на Web база знаний, которая состоит из руководств, шаблонов, наставлений по использованию инструментальных средств, и которая может быть разбита на:

· обширные руководства для всех членов коллектива разработчиков, для каждого временного интервала жизненного цикла ПО. Руководства представлены в двух видах: для осмысления процесса на верхнем уровне, и в

виде подробных наставлений по повседневной деятельности. Руководства опубликованы в HTML формате;

· наставления по пользованию инструментальными средствами, которые автоматизируют большие разделы процесса создания ПО. Наставления опубликованы в HTML формате.

· примеры и шаблоны для Rational Rose, которые служат руководствами по тому, как структурировать информацию в Rational Rose при следовании указаниям RUP;

· шаблоны для SoDa – более десятка шаблонов для SoDa, которые помогают автоматизировать документирование ПО;

· Microsoft Word шаблоны – более 30 шаблонов, которые предназначены для поддержки документации по всем последовательностям действий и интервалам жизненного цикла ПО;

· планы в формате Microsoft Project – для тех, кому трудно сразу перейти к созданию планов - отражают итерационную разработку. Данные документы помогают произвести такой переход;

· Книга Ph. Kruchten - Rational Unified Process-An Introduction. Книга содержит 277 страниц и является хорошим вступлением и обзором к процессу и базе знаний.

- веб сайт, содержащий описание процесса и интегрированный со многими инструментальными средствами;

- Development Kit – описывает то, каким образом можно конфигурировать и расширить RUP для специфических нужд проекта, и обеспечивает инструменты и шаблоны, помогающие это выполнить;

- средства кастомизации, позволяющие создавать собственные процессы (IBM Rational Workbench);

- доступ к Resource Center, который содержит последние публикации, обновления, подсказки, методики, а также ссылки на add-on и сервисы.

- сообщество пользователей RUP, участие в котором поможет вам обмениваться опытом (в том числе и готовыми описаниями процессов) с другими разработчиками.
Пользователи RUP могут либо выбрать одно из типовых представлений процесса, либо создать свое собственное.

Продукт RUP позволяет настраивать процесс под нужды конкретной организации-разработчика и конкретного проекта, включая в него различные готовые компоненты (plug-in), а также разрабатывать и включать в состав процесса собственные компоненты. Продукт содержит также представления (view), которые позволяют участникам разработки получать доступ к необходимой им информации в зависимости от ролевых или персональных настроек.

**Использование инструментальных средств**

Для обеспечения инструментальной поддержки всех процессов жизненного цикла разработки и сопровождения ПС RUP рекомендует использование специализированных инструментальных средств IBM Rational:
— **управление требованиями** – IBM Rational RequisitePro;
— **визуальное моделирование и генерация объектного кода** – IBM Rational Rose, IBM Rational XDE;
— **разработка** — IBM Rational RapidDeveloper
— **конфигурационное управление** – IBM Rational ClearCase;
— **управление изменениями** – IBM Rational ClearQuest;
— **автоматизированное документирование** – IBM Rational SoDA;
— **автоматизированное тестирование** – IBM Rational TeamTest, IBM Rational TestFactory, IBM Rational Robot, IBM Rational PurifyPlus, Rational Visual Quantify, Rational Visual PureCoverage, Rational PerformanceStudio, IBM Rational SiteCheck и IBM Rational SiteLoad.

· Rational Requisite Pro - поддерживает обновления и отслеживает изменения в требованиях для всего коллектива разработчиков, представляя их в удобном виде для чтения, обсуждения и изменений.

- Rational ClearQuest - Windows и Web-размещаемый продукт, который помогает коллективу разработчиков отслеживать и управлять всеми действиями по изменению ПО в течение его жизненного цикла.
- Rational Rose - мировой лидер среди средств визуального моделирования для бизнес процессов, анализа требований, и проектирования на основе архитектуры компонентов.
- Rational SoDA - автоматизирует создание документации для всего процесса разработки ПО, значительно сокращая стоимость документации и время на ее создание.
- Rational Purify - средство поиска ошибок на run-time для разработчиков приложений и компонентов, программирующих на C/C++; помогает находить ошибки утечки памяти.
- Rational Visual Quantify - средство измерения характеристик для разработчиков приложений и компонентов, программирующих на C/C++, Visual Basic и Java; помогает определять и устранять узкие места в производительности ПО.

- Rational Visual PureCoverage - автоматически определяет области кода, которые не подвергаются тестированию; разработчики могут учесть это и более тщательно выполнять проверку.
- SQA TeamTest - создает, обслуживает и выполняет автоматизированные функциональные тесты, позволяя тщательно протестировать код и проверить, соответствует ли ПО предъявляемым к нему требованиям.
- Rational PerformanceStudio - простое в использовании, точное и масштабируемое средство, которое измеряет и предсказывает характеристики клиент/серверных и Web систем.
- Rational ClearCase - лидирующее на рынке средство конфигурационного управления, позволяющее менеджерам проекта отслеживать эволюцию каждого разрабатываемого проекта.

Лекция 4

**ТЕМА:Этап логического проектирования ИС. Основные подходы при создании концептуальной модели.**

Рассмотрим такие понятии, как «Предметная область» и «Бизнес-моделирование».

**Предметная область** - часть реального мира, подлежащая изучению с целью организации управления и, в конечном счете, автоматизации. Предметная область представляется множеством *фрагментов*, например, предприятие - цехами, дирекцией, бухгалтерией и т.д. Каждый фрагмент предметной области характеризуется множеством *объектов* и *процессов*, использующих объекты, а также множеством *пользователей*, характеризуемых различными взглядами на предметную область.

**Бизнес-моделирование (деловое моделирование)** - деятельность по формированию моделей организаций, включающая описание деловых объектов (подразделений, должностей, ресурсов, ролей, процессов, операций, информационных систем, носителей информации и т. д.) и указание связей между ними. Требования к формируемым моделям и их соответствующее содержание определяются целями моделирования.

Бизнес-моделирование является отдельным подпроцессом в процессе разработки программного обеспечения, в котором описывается деятельность компании и определяются требования к системе, т.е. те подпроцессы и операции, которые подлежат автоматизации в разрабатываемой информационной системе.

**Моделью бизнес-процесса** называется его формализованное (графическое, табличное, текстовое, символьное) описание, отражающее реально существующую или предполагаемую деятельность предприятия.

Модель, как правило, содержит следующие сведения о бизнес-процессе:

· набор составляющих процесс шагов - бизнес-функций;

· порядок выполнения бизнес-функций;

· механизмы контроля и управления в рамках бизнес-процесса;

· исполнителей каждой бизнес-функции;

· входящие документы/информацию, исходящие документы/информацию;

· ресурсы, необходимые для выполнения каждой бизнес-функции;

· документацию/условия, регламентирующие выполнение каждой бизнес-функции;

· параметры, характеризующие выполнение бизнес-функций и процесса в целом.

Модели бизнес-процессов применяются предприятиями для различных целей, что определяет тип разрабатываемой модели.

**Графическая** модель бизнес-процесса в виде наглядной, общепонятной диаграммы может служить для обучения новых сотрудников их должностным обязанностям, согласования действий между структурными единицами компании, подбора или разработки компонентов информационной системы и т. д. Описание с помощью моделей такого типа существующих и целевых бизнес-процессов используется для оптимизации и совершенствования деятельности компании путем устранения узких мест, дублирования функций и прочего.

**Имитационные** модели бизнес-процессов позволяют оценить их эффективность и посмотреть, как будет выполняться процесс с входными данными, не встречавшимися до сих пор в реальной работе предприятия.

**Исполняемые** модели бизнес-процессов могут быть запущены на специальном программном обеспечении для автоматизации процесса непосредственно по модели.

Поскольку модели бизнес-процессов предназначены для широкого круга пользователей (бизнес-аналитиков, рядовых сотрудников и руководства компании), а их построением часто занимаются неспециалисты в области информационных технологий, наиболее широко используются модели графического типа, в которых в соответствии с определенной методологией бизнес-процесс представляется в виде наглядного графического изображения - диаграммы, состоящей в основном из прямоугольников и стрелок. Такое представление обладает высокой, многомерной информативностью, которая выражается в различных свойствах (цвет, фон, начертание и т.д.) и атрибутах (вес, размер, стоимость, время и т.д.) каждого объекта и связи.

В последние годы разработчики программных средств моделирования бизнес-процессов уделяют большое внимание преобразованию графических моделей в модели других видов, в частности в исполняемые, назначением которых является обеспечение автоматизации бизнес-процесса и интеграция работы задействованных в его исполнении информационных систем.

Согласно еще одной классификации, пришедшей из моделирования сложных систем, выделяют следующие виды моделей бизнес-процессов:

· **функциональные**, описывающие совокупность выполняемых системой функций и их входы и выходы;

· **поведенческие**, показывающие, когда и/или при каких условиях выполняются бизнес- функции, с помощью таких категорий, как состояние системы, событие, переход из одного состояния в другое, условия перехода, последовательность событий;

· **структурные**, характеризующие морфологию системы - состав подсистем, их взаимосвязи;

· **информационные**, отражающие структуры данных - их состав и взаимосвязи.

Проблема сложности является главной проблемой, которую приходится решать при создании больших систем любой природы, в том числе и ЭИС. Ни один разработчик не в состоянии выйти за пределы человеческих возможностей и понять все систему в целом. Единственно эффективный подход к решению этой проблемы заключается в построении сложной системы из небольшого количества крупных частей, каждая из которых, в свою очередь, строится из частей меньшего размера и т.д., до тех

пор, пока самые небольшие части можно будет строить из имеющегося материала. Этот подход известен под самыми разными названиями, среди них такие, как «разделяй и властвуй», иерархическая декомпозиция и др. По отношению к проектированию сложной программной системы это означает, что ее необходимо разделять (декомпозировать) на небольшие подсистемы, каждую из которых можно разрабатывать независимо от других. Это позволяет при разработке подсистемы любого уровня держать в уме информацию только о ней, а не обо всех остальных частях системы. Правильная декомпозиция является главным способом преодоления сложности разработки больших систем. Понятие «правильная» по отношению к декомпозиции означает следующее:

1. Количество связей между отдельными подсистемами должно быть минимальным.

2. Связность отдельных частей внутри каждой подсистемы должна быть максимальной.

Структура системы должна быть таковой, чтобы все взаимодействия между ее подсистемами укладывались в ограниченные, стандартные рамки:

1. Каждая подсистема должна инкапсулировать свою содержимое (скрывать его от других подсистем).

2. Каждая подсистема должна иметь четко определенный интерфейс с другими подсистемами.

На сегодняшний день в программной инженерии существуют два основных подхода к разработке ПО ЭИС, принципиальное различие которых обусловлено разными способами декомпозиции систем:

1. Функционально-модульный или структурный

2. Объектно-ориентированный.

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. При этом структура системы описывается в терминах объектов и связей между ними, а поведение системы описывается в терминах обмена сообщениями между объектами.

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.

**Методы моделирования бизнес процессов**

На сегодняшний день существует достаточно большое количество методов моделирования бизнес процессов. Эти методы относятся к разным видам моделирования и позволяют сфокусировать внимание на различных аспектах. Они содержат как графические, так и текстовые средства, за счет которых можно наглядно представить основные компоненты процесса и дать точные определения параметров и связей элементов.

Наиболее часто моделирование бизнес-процессов выполняют с помощью следующих методов:

· **Flow Chart Diagram** (диаграмма потока работ) – это графический метод представления процесса в котором операции, данные, оборудование процесса и пр. изображаются специальными символами. Метод применяется для отображения логической последовательности действий процесса. Главным достоинством метода является его гибкость. Процесс может быть представлен множеством способов.

· **Data Flow Diagram** (диаграмма потока данных). Диаграмма потока данных или DFD применяется для отображения передачи информации (данных) от одной операции процесса к другой. DFD описывает взаимосвязь операций за счет информации и данных. Этот метод является основой структурного анализа процессов, т.к. позволяет разложить процесс на логические уровни. Каждый процесс может быть разбит на подпроцессы с более высоким уровнем детализации. Применение DFD позволяет отразить только поток информации, но не поток материалов. Диаграмма потока данных показывает, как информация входит и выходит из процесса, какие действия изменяют информацию, где информация хранится в процессе и прочее.

· **RoleActivityDiagram** (диаграммаролей). Она применяется для моделирования процесса с точки зрения отдельных ролей, групп ролей и взаимодействия ролей в процессе. Роль представляет собой абстрактный элемент процесса, выполняющий какую-либо организационную функцию. Диаграмма ролей показывает степень «ответственности» за процесс и его операции, а также взаимодействие ролей.

· **IDEF** (Integrated Definition for Function Modeling) – представляет собой целый набор методов для описания различных аспектов бизнес-процессов (IDEF0, IDEF1, IDEF1X , IDEF2, IDEF3, IDEF4, IDEF5). ЭтиметодыстроятсянабаземетодологииSADT (StructuredAnalysisandDesignTechnique). Для моделирования бизнес процессов наиболее часто применяют методы IDEF0 и IDEF3.

- IDEF0 – позволяет создать модель функций процесса. На диаграмме IDEF0 отображаются основные функции процесса, входы, выходы, управляющие воздействия и устройства, взаимосвязанные с основными функциями. Процесс может быть декомпозирован на более низкий уровень.

- IDEF3 – этот метод позволяет создать «поведенческую» модель процесса. IDEF3 состоит из двух видов моделей. Первый вид представляет описание потока работ. Второй – описание состояний перехода объектов.

·**ERD** (Entity - Relationship Diagrams) - диаграммы «сущность-связь». С помощью ERD выполняется описание используемых в организации данных на концептуальном уровне, не зависимо от средств реализации базы данных (СУБД). ER-диаграммы моделируют данные и их отношения. Ключевыми понятиями в них являются ***сущность***, ***атрибуты*** сущности и ***связи*** между сущностями.

· **Цветные сети Петри** – этот метод представляет модель процесса в виде графа, где вершинами являются действия процесса, а дугами события, за счет которых осуществляется переход процесса из одного состояния в другое. Сети Петри применяют для динамического моделирования поведения процесса.

· **UnifiedModeling Language** (UML) - представляет собой объектно-ориентированный метод моделирования процессов. Он состоит из 8-ти различных диаграмм, каждая из которых позволяет моделировать отдельные статические или динамические аспекты процесса.

Большинство из указанных методов реализованы в виде программного обеспечения. Оно позволяет осуществлять поддержку бизнес-процессов или проводить их анализ. Примерами такого ПО являются различные CASE средства моделирования процессов.

**Структурный подход**

Итак, сущность структурного подхода к разработке ПО ЭИС заключается в ее декомпозиции (разбиении) на автоматизируемые функции: система разбивается на функциональные подсистемы, которые, в свою очередь, делятся на подфункции, те - на задачи и так далее до конкретных процедур. При этом система сохраняет целостное представление, в котором все составляющие компоненты взаимоувязаны. При разработке системы «снизу-вверх», от отдельных задач ко всей системе, целостность теряется, возникают проблемы при описании информационного взаимодействия отдельных компонентов.

Все наиболее распространенные методы структурного подхода базируются на ряде общих принципов:

1. Принцип «разделяй и властвуй»;

2. Принцип иерархического упорядочения- принцип организации составных частей системы в иерархические древовидные структуры с добавлением новых деталей на каждом уровне.

Выделение двух базовых принципов не означает, что остальные принципы являются второстепенными, т.к. игнорирование любого из них может привести к непредсказуемым последствиям (в том числе и к провалу всего проекта»). Основными из этих принципов являются:

1. Принцип абстрагирования- выделение существенных аспектов системы и отвлечение от несущественных.

2. Принцип непротиворечивости,обоснованность и согласованность элементов системы.

3. Принцип структурирования данных- данные должны быть структурированы и иерархически организованы.

В структурном подходе в основном две группы средств, описывающих функциональную структуру системы и отношения между данными. Каждой группе средств соответствуют определенные виды моделей (диаграмм), наиболее распространенными среди них являются:

·DFD (Data Flow Diagrams) - диаграммы потоков данных;

· SADT (Structured Analysis and Design Technique - методология структурного анализа и проектирования) - модели и соответствующие функциональные диаграммы: нотации IDEF0 (функциональное моделирование систем), IDEF1x (концептуальное моделирование баз данных),

IDEF3x (построение систем оценки качества работы объекта; графическое описание потока процессов, взаимодействия процессов и объектов, которые изменяются этими процессами);

·ERD (Entity - Relationship Diagrams) - диаграммы «сущность-связь».

Практически во всех методах структурного подхода (структурного анализа) на стадии формирования требований к ПО используются две группы средств моделирования:

1. Диаграммы, иллюстрирующие функции, которые система должна выполнять, и связи между этими функциями - DFD или SADT (IDEF0).

2. Диаграммы, моделирующие данные и их отношения (ERD).

Конкретный вид перечисленных диаграмм и интерпретация их конструкций зависят от стадии ЖЦ ПО.

На стадии формирования требований к ПО SADT-модели и DFD используются для построения модели “AS-IS” и модели “TO-BE”, отражая таким образом существующую и предлагаемую структуру бизнес-процессов организации и взаимодействие между ними (использование SADT-моделей , как правило, ограничивается только данной стадией, поскольку они изначально не предназначались для проектирования ПО). С помощью ERD выполняется описание используемых в организации данных на концептуальном уровне, не зависимо от средств реализации базы данных (СУБД).

На стадии проектирования DFD используются для описания структуры проектируемой системы.

Перечисленные модели в совокупности дают полное описание ПО ЭИС независимо от того, является ли система существующей или вновь разрабатываемой.

**Объектно-ориентированный подход**

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.

**Определение: Объект** -предмет или явление, имеющее четко определенное поведение и обладающие состоянием, поведением и индивидуальностью.

Структура и поведение схожих объектов определяют общий для них класс.

**Определение: 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.

В нотации языка UML определены следующие виды канонических диаграмм:

· вариантов использования (use case diagram)

- классов (class diagram)
- кооперации (collaboration diagram)
- последовательности (sequence diagram)
- состояний (statechart diagram)
- деятельности (activity diagram)
- компонентов (component diagram)

· развертывания (deployment diagram)

Перечень этих диаграмм и их названия являются каноническими в том смысле, что представляют собой неотъемлемую часть графической нотации языка UML. Более того, процесс ООАП неразрывно связан с процессом построения этих диаграмм. При этом совокупность построенных таким образом диаграмм является самодостаточной в том смысле, что в них содержится вся информация, которая необходима для реализации проекта сложной системы.

Каждая из этих диаграмм детализирует и конкретизирует различные представления о модели сложной системы в терминах языка UML. При этом диаграмма вариантов использования представляет собой наиболее общую концептуальную модель сложной системы, которая является исходной для построения всех остальных диаграмм. Диаграмма классов, по своей сути, логическая модель, отражающая статические аспекты структурного построения сложной системы.

Диаграммы кооперации и последовательностей представляют собой разновидности логической модели, которые отражают динамические аспекты функционирования сложной системы. Диаграммы состояний и деятельности предназначены для моделирования поведения системы. И, наконец, диаграммы компонентов и развертывания служат для представления физических компонентов сложной системы и поэтому относятся к ее физической модели.

В целом интегрированная модель сложной системы в нотации UML может быть представлена в виде совокупности указанных выше диаграмм (рис. 2).


**Рис. 2.** Интегрированная модель сложной системы в нотации UML

На этапе постановки задачи и анализа требований к системе используют диаграммы прецедентов, диаграммы деятельностей для расшифровки содержания прецедентов, диаграммы состояний для моделирования поведения объектов со сложным состоянием, диаграммы классов для выделения концептуальных сущностей предметной области задачи и диаграммы последовательностей действий.

Спецификация разрабатываемого программного обеспечения при использовании UML объединяет несколько моделей: логическую, использования, реализации, процессов, развертывания.

**Модель использования** содержит описание функций программного обеспечения с точки зрения пользователя.

**Логическая модель** описывает ключевые понятия моделируемого программного обеспечения (классы, интерфейсы и т. п.), т. е. средства, обеспечивающие его функциональность.

**Модель реализации** определяет реальную организацию программных модулей в среде разработки.

**Модель процессов** отображает организацию вычислений и позволяет оценить производительность, масштабируемость и надежность программного обеспечения.

**Модель развертывания** показывает, каким образом программные компоненты размещаются на конкретном оборудовании.

Все вместе указанные модели, каждая из которых характеризует определенную сторону проектируемого продукта, составляют относительно полную модель разрабатываемого программного обеспечения.

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 ориентирована 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.

Лекция 5

**ТЕМА: Описание функциональности разработки: нотация IDEF0.**

**Функциональная методика IDEF0**

Методологию IDEF0 можно считать следующим этапом развития хорошо известного графического языка описания функциональных систем SADT (Structured Analysis and Design Technique). Исторически IDEF0 как стандарт был разработан в 1981 году в рамках обширной программы автоматизации промышленных предприятий, которая носила обозначение ICAM (Integrated Computer Aided Manufacturing). Семейство стандартов IDEF унаследовало свое обозначение от названия этой программы (IDEF=Icam DEFinition), и последняя его редакция была выпущена в декабре 1993 года Национальным Институтом по Стандартам и Технологиям США (NIST).

Целью методики является построение функциональной схемы исследуемой системы, описывающей все необходимые процессы с точностью, достаточной для однозначного моделирования деятельности системы.

В основе методологии лежат четыре основных понятия: **функциональный блок, интерфейсная дуга, декомпозиция, глоссарий**.

**Функциональный блок** (Activity Box) представляет собой некоторую конкретную функцию в рамках рассматриваемой системы. По требованиям стандарта название каждого функционального блока должно быть сформулировано в глагольном наклонении (например, "производить услуги"). На диаграмме функциональный блок изображается прямоугольником (рис. 1). Каждая из четырех сторон функционального блока имеет свое определенное значение (роль), при этом:

· верхняя сторона имеет значение "Управление" (Control);

· левая сторона имеет значение "Вход" (Input);

· правая сторона имеет значение "Выход" (Output);

· нижняя сторона имеет значение "Механизм" (Mechanism).


**Рис. 1.**Функциональный блок

**Интерфейсная дуга** (Arrow) отображает элемент системы, который обрабатывается функциональным блоком или оказывает иное влияние на

функцию, представленную данным функциональным блоком. Интерфейсные дуги часто называют потоками или стрелками.

С помощью интерфейсных дуг отображают различные объекты, в той или иной степени определяющие процессы, происходящие в системе. Такими объектами могут быть элементы реального мира (детали, вагоны, сотрудники и т.д.) или потоки данных и информации (документы, данные, инструкции и т.д.).

В зависимости от того, к какой из сторон функционального блока подходит данная интерфейсная дуга, она носит название "входящей", "исходящей" или "управляющей". Функциональный блок (или Функция) преобразует Входы в Выходы (т.е. входную информацию в выходную), Управление определяет, когда и как это преобразование может или должно произойти, Механизмы непосредственно осуществляют это преобразование. Механизмы (дуги снизу) показывают средства, с помощью которых осуществляется выполнение функций. Механизм может быть человеком, компьютером или любым другим устройством, которое помогает выполнять данную функцию.

Дуги показывают, как функции между собой взаимосвязаны, как они обмениваются данными и осуществляют управление друг другом. Дуги могут разветвляться и соединяться. Выходы одной функции могут быть Входами, Управлением или Механизмами для другой.

Необходимо отметить, что любой функциональный блок по требованиям стандарта должен иметь, по крайней мере, одну управляющую интерфейсную дугу и одну исходящую. Это и понятно – каждый процесс должен происходить по каким-то правилам (отображаемым управляющей дугой) и должен выдавать некоторый результат (выходящая дуга), иначе его рассмотрение не имеет никакого смысла.

Обязательное наличие управляющих интерфейсных дуг является одним из главных отличий стандарта IDEF0 от других методологий классов DFD (Data Flow Diagram) и WFD (Work Flow Diagram).

**Декомпозиция** (Decomposition) является основным понятием стандарта IDEF0. Принцип декомпозиции применяется при разбиении сложного процесса на составляющие его функции. При этом уровень детализации процесса определяется непосредственно разработчиком модели.

Декомпозиция позволяет постепенно и структурировано представлять модель системы в виде иерархической структуры отдельных диаграмм, что делает ее менее перегруженной и легко усваиваемой.

Последним из понятий IDEF0 является **глоссарий (Glossary)**. Для каждого из элементов IDEF0-диаграмм, функциональных блоков, интерфейсных дуг - существующий стандарт подразумевает создание и поддержание набора соответствующих определений, ключевых слов, повествовательных изложений и т.д., которые характеризуют объект, отображенный данным элементом. Этот набор называется глоссарием и является описанием сущности данного элемента. Глоссарий гармонично дополняет наглядный графический язык, снабжая диаграммы необходимой дополнительной информацией.

Прежде чем приступать к описанию модели, необходимо определить **субъект модели**, **цель** и **точку зрения** модели. В качестве субъекта модели выступает описываемая система. Субъект определяет, что включить в модель, а что исключить из нее.

Модель IDEF0 всегда начинается с представления системы как единого целого – одного функционального блока с интерфейсными дугами, простирающимися за пределы рассматриваемой области. Такая диаграмма с одним функциональным блоком называется **контекстной диаграммой**.

В пояснительном тексте к контекстной диаграмме должна быть указана **цель** (Purpose) построения диаграммы в виде краткого описания и зафиксирована **точка зрения** (Viewpoint).

Определение и формализация цели разработки IDEF0-модели является крайне важным моментом. Фактически цель определяет соответствующие области в исследуемой системе, на которых необходимо фокусироваться в первую очередь. Цель становится критерием окончания моделирования.

**Точка зрения определяет основное направление развития модели и уровень необходимой детализации**. Четкое фиксирование точки зрения позволяет разгрузить модель, отказавшись от детализации и исследования отдельных элементов, не являющихся необходимыми, исходя из выбранной точки зрения на систему. Правильный выбор точки зрения существенно сокращает временные затраты на построение конечной модели.

**Выделение подпроцессов.** В процессе декомпозиции функциональный блок, который в контекстной диаграмме отображает систему как единое целое, подвергается детализации на другой диаграмме. Получившаяся диаграмма второго уровня содержит функциональные блоки, отображающие главные подфункции функционального блока контекстной диаграммы, и называется дочерней (Child Diagram) по отношению к нему (каждый из функциональных блоков, принадлежащих дочерней диаграмме, соответственно называется дочерним блоком – Child Box). В свою очередь, функциональный блок-предок называется родительским блоком по отношению к дочерней диаграмме (Parent Box), а диаграмма, к которой он

принадлежит – родительской диаграммой (Parent Diagram). В правом верхнем углу диаграммы располагается номер блока, чью декомпозицию представляет диаграмма, он же рассматривается как номер диаграммы. Внизу блоков располагается номер, который будет присваиваться диаграмме, на которой будет представлена декомпозиция данного блока.

Каждая из подфункций дочерней диаграммы может быть далее детализирована путем аналогичной декомпозиции соответствующего ей функционального блока. В каждом случае декомпозиции функционального блока все интерфейсные дуги, входящие в данный блок или исходящие из него, фиксируются на дочерней диаграмме. Этим достигается структурная целостность IDEF0–модели.

Блоки SADT никогда не размещаются на диаграмме случайным образом. Они размещаются по степени важности, как ее понимает автор диаграммы. Этот относительный порядок называется доминированием. Доминирование понимается как влияние, которое один блок оказывает на другие блоки диаграммы. Например, самым доминирующим блоком диаграммы может быть либо первый из требуемой последовательности функций, либо планирующая или контролирующая функция, влияющая на все остальные функции. Наиболее доминирующий блок обычно располагается в верхнем левом углу диаграммы, а наименее доминирующий – в правом нижнем углу. В результате получается «ступенчатая» схема.

После завершения диаграммы ее внешние дуги стыкуются с родительской диаграммой для обеспечения согласованности. Одним из способов такой стыковки может служить присваивание кодов ICOM внешним дугам новой диаграммы согласно следующим правилам:

1) присваивается код каждой зрительной связи. Используется I для входных дуг, C - для связей между дугами управления, O - для связей между выходными дугами, M - для связей между дугами механизма.

2) после каждой буквы добавляется цифра, соответствующая положению данной дуги среди других дуг того же типа, касающихся родительскогоблока. Причем входные и выходные дуги пересчитываются сверху вниз, а дуги управлений и механизмов пересчитываются слева направо.

Данный принцип используется на контекстной диаграмме. Далее полученные там ICOM-коды используются вместо имен данных для сокращения записи.

Дуги, входящие в блок и выходящие из него на диаграмме верхнего уровня, являются точно теми же самыми, что и дуги, входящие в диаграмму

нижнего уровня и выходящие из нее, потому что блок и диаграмма изображают одну и ту же часть системы.

Иногда отдельные интерфейсные дуги высшего уровня не имеет смысла продолжать рассматривать на диаграммах нижнего уровня, или наоборот — отдельные дуги нижнего отражать на диаграммах более высоких уровней – это будет только перегружать диаграммы и делать их сложными для восприятия. Для решения подобных задач в стандарте IDEF0 предусмотрено понятие туннелирования. Обозначение "туннеля" (Arrow Tunnel) в виде двух круглых скобок вокруг начала интерфейсной дуги обозначает, что эта дуга не была унаследована от функционального родительского блока и появилась (из "туннеля") только на этой диаграмме. В свою очередь, такое же обозначение вокруг конца (стрелки) интерфейсной дуги в непосредственной близи от блока–приемника означает тот факт, что в дочерней по отношению к этому блоку диаграмме эта дуга отображаться и рассматриваться не будет. Чаще всего бывает, что отдельные объекты и соответствующие им интерфейсные дуги не рассматриваются на некоторых промежуточных уровнях иерархии, – в таком случае они сначала "погружаются в туннель", а затем при необходимости "возвращаются из туннеля".

Обычно IDEF0-модели несут в себе сложную и концентрированную информацию, и для того, чтобы ограничить их перегруженность и сделать удобочитаемыми, в стандарте приняты соответствующие ограничения сложности.

Рекомендуется представлять на диаграмме от трех до шести функциональных блоков, при этом количество подходящих к одному функциональному блоку (выходящих из одного функционального блока) интерфейсных дуг предполагается не более четырех.

Стандарт IDEF0 содержит набор процедур, позволяющих разрабатывать и согласовывать модель большой группой людей, принадлежащих к разным областям деятельности моделируемой системы. Обычно процесс разработки является итеративным и состоит из следующих условных этапов:

· Создание модели группой специалистов, относящихся к различным сферам деятельности предприятия. Эта группа в терминах IDEF0 называется авторами (Authors). Построение первоначальной модели является динамическим процессом, в течение которого авторы опрашивают компетентных лиц о структуре различных процессов, создавая модели деятельности подразделений. При этом их интересуют ответы на следующие вопросы:

- Что поступает в подразделение "на входе"?

- Какие функции и в какой последовательности выполняются в рамках подразделения?

- Кто является ответственным за выполнение каждой из функций?

- Чем руководствуется исполнитель при выполнении каждой из функций?

- Что является результатом работы подразделения (на выходе)?

На основе имеющихся положений, документов и результатов опросов создается черновик (Model Draft) модели.

· Распространение черновика для рассмотрения, согласований и комментариев. На этой стадии происходит обсуждение черновика модели с широким кругом компетентных лиц (в терминах IDEF0 — читателей) на предприятии. При этом каждая из диаграмм черновой модели письменно критикуется и комментируется, а затем передается автору. Автор, в свою очередь, также письменно соглашается с критикой или отвергает ее с изложением логики принятия решения и вновь возвращает откорректированный черновик для дальнейшего рассмотрения. Этот цикл продолжается до тех пор, пока авторы и читатели не придут к единому мнению.

· Официальное утверждение модели. Утверждение согласованной модели происходит руководителем рабочей группы в том случае, если у авторов модели и читателей отсутствуют разногласия по поводу ее адекватности. Окончательная модель представляет собой согласованное представление о предприятии (системе) с заданной точки зрения и для заданной цели.

Наглядность графического языка IDEF0 делает модель вполне читаемой и для лиц, которые не принимали участия в проекте ее создания, а также эффективной для проведения показов и презентаций. В дальнейшем на базе построенной модели могут быть организованы новые проекты, нацеленные на производство изменений в модели.

Недостатки IDEF0-диаграмм:

- явно не указаны ни последовательность, ни время;

- большое количество дуг на диаграммах и большое количество уровней декомпозиции увеличивают сложность восприятия;

- трудно увязать нескольких процессов.

На рис. 2 приведена диаграмма IDEF0 верхнего уровня бизнес-процесса "Увольнение сотрудника".

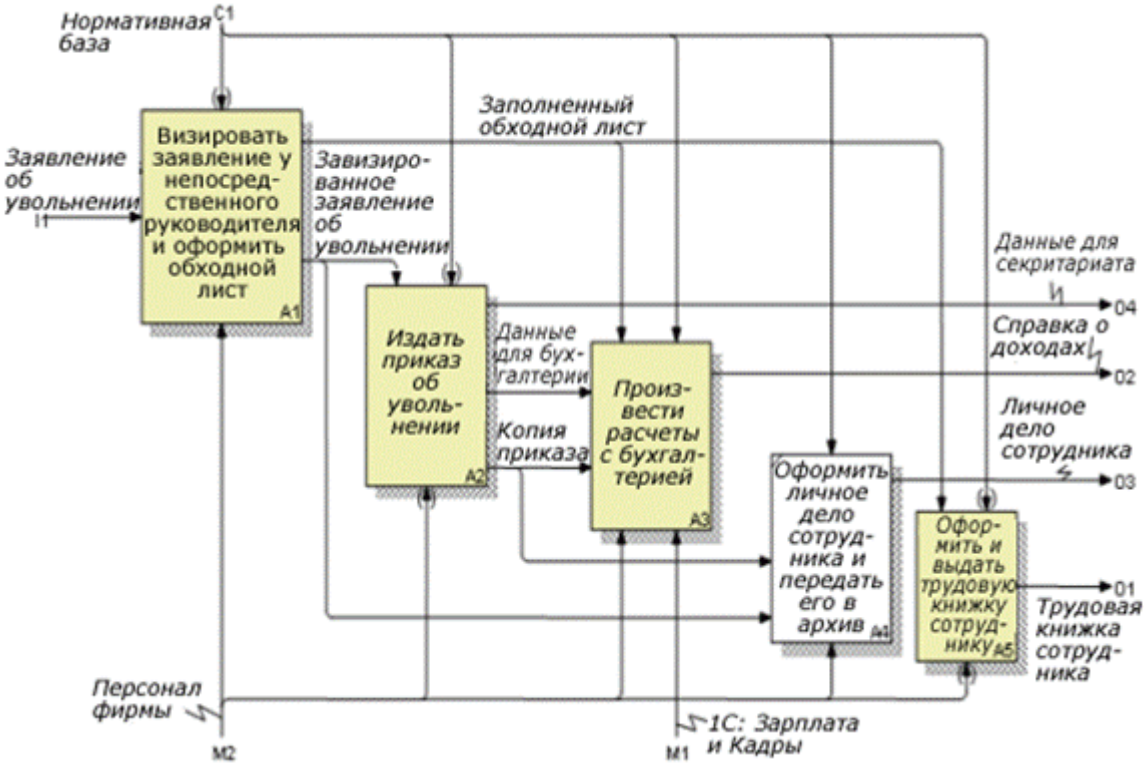



Рис. 2. Диаграмма IDEF0 верхнего уровня бизнес-процесса "Увольнение сотрудника".

Лекция 6

**ТЕМА:Описание функциональности разработки: диаграммы потоков данных.**

Нотация DFD – моделирование потоков данных (процессов) – основа методологии Gane/Sarson, в соответствии с которой модель системы определяется как иерархия диаграмм потоков данных, описывающих асинхронный процесс преобразования информации от ее ввода в систему до

выдачи объекту или субъекту. Контекстные диаграммы иерархии определяют основные процессы или подсистемы системы с внешними входами и выходами. Они детализируются при помощи диаграмм-потомков. Декомпозиция ведется до тех пор, пока не будет достигнут такой уровень декомпозиции, на котором процессы становятся элементарными и детализировать их далее невозможно. Главная цель такого представления - продемонстрировать, как каждый процесс преобразует свои входные данные в выходные, а также выявить отношения между этими процессами.

Диаграммы DFD обычно строятся для наглядного изображения текущей работы системы документооборота организации. Как правило, диаграммы DFD используют в качестве дополнения модели бизнес-процессов, выполненной в IDEF0.

Источники информации (внешние сущности) порождают информационные потоки (потоки данных), переносящие информацию к подсистемам или процессам. Те в свою очередь преобразуют информацию и порождают новые потоки, которые переносят информацию к другим процессам или подсистемам, накопителям данных или внешним сущностям - потребителям информации. Таким образом, основными компонентами диаграмм потоков данных являются:

·внешние сущности;

·системы/подсистемы;

·процессы;

·накопители данных;

·потоки данных.

Данный стандарт представлен двумя немного различающихся вариантами, которые называют нотациями. Первая из них называется нотацией Гейна-Сарсана, вторая нотацией Йордона-Де Марко.

**Таблица 1. Элементы методологии DFD в нотациях Гейна-Сарсана и Йордона-Де Марко.**

| Элемент | Описание | Нотация Йордона-Де Марко | Нотация Гейна-Сарсона |
|---|---|---|---|
| **Функция** | Работа. | Имя функции<br>Номер | Имя функции<br>Номер |
| **Поток данных** | Объект, над которым выполняется работа. Может быть логическим или управляющим. (Управляющие потоки обозначаются пунктирной линией со стрелкой). | Имя объекта | Имя объекта<br>(Понятие управляющего потока отсутствует) |
| **Хранилище данных** | Структура для хранения информационных объектов. | Имя объекта | Имя объекта |
| **Внешняя сущность** | Внешний по отношению к системе объект, обменивающийся с нею потоками. | Имя внешнего объекта | Имя внешнего объекта |

**Внешние сущности**

Внешняя сущность представляет собой материальный предмет или физическое лицо, представляющее собой источник или приемник информации, например, заказчики, персонал, поставщики, клиенты, склад. Определение некоторого объекта или системы в качестве внешней сущности указывает на то, что она находится за пределами границ анализируемой ИС. В процессе анализа некоторые внешние сущности могут быть перенесены внутрь диаграммы анализируемой ИС, если это необходимо, или, наоборот, часть процессов ИС может быть вынесена за пределы диаграммы и представлена как внешняя сущность.

Внешняя сущность обозначается квадратом (рисунок 3), расположенным как бы "над" диаграммой и бросающим на нее тень, для того, чтобы можно было выделить этот символ среди других обозначений:


Рис. 3. Внешняя сущность.

**Системы и подсистемы**

При построении модели сложной ИС она может быть представлена в самом общем виде на так называемой контекстной диаграмме в виде одной системы как единого целого, либо может быть декомпозирована на ряд подсистем.

Подсистема (или система) на контекстной диаграмме изображается следующим образом (рисунок 4).

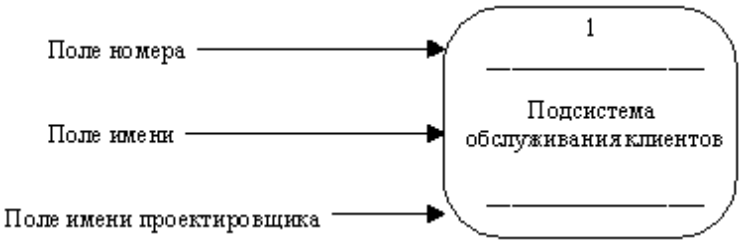



Рис. 4. Подсистема.

Номер подсистемы служит для ее идентификации. В поле имени вводится наименование подсистемы в виде предложения с подлежащим и соответствующими определениями и дополнениями.

**Процессы**

Процесс представляет собой преобразование входных потоков данных в выходные в соответствии с определенным алгоритмом. Физически процесс может быть реализован различными способами: это может быть подразделение организации (отдел), выполняющее обработку входных документов и выпуск отчетов, программа, аппаратно реализованное логическое устройство и т.д.

Процесс на диаграмме потоков данных изображается, как показано на рисунке 5.

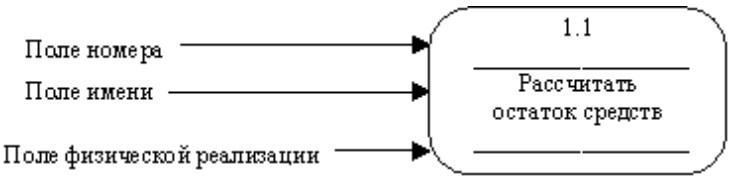



Рис. 5. Процесс.

Номер процесса служит для его идентификации. В поле имени вводится наименование процесса в виде предложения с активным недвусмысленным глаголом в неопределенной форме (вычислить, рассчитать, проверить, определить, создать, получить), за которым следуют существительные в винительном падеже, например: "Ввести сведения о

клиентах"; "Выдать информацию о текущих расходах"; "Проверить кредитоспособность клиента".

Использование таких глаголов, как "обработать", "модернизировать" или "отредактировать" означает, как правило, недостаточно глубокое понимание данного процесса и требует дальнейшего анализа.

Информация в поле физической реализации показывает, какое подразделение организации, программа или аппаратное устройство выполняет данный процесс.

**Накопители данных**

Накопитель данных представляет собой абстрактное устройство для хранения информации, которую можно в любой момент поместить в накопитель и через некоторое время извлечь, причем способы помещения и извлечения могут быть любыми.

Накопитель данных может быть реализован физически в виде ящика в картотеке, таблицы в оперативной памяти, файла на магнитном носителе и т.д. Накопитель данных на диаграмме потоков данных изображается, как показано на рисунке 6.

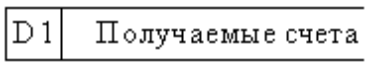



Рис. 6. Накопитель данных.

Накопитель данных идентифицируется буквой "D" и произвольным числом. Имя накопителя выбирается из соображения наибольшей информативности для проектировщика.

Накопитель данных в общем случае является прообразом будущей базы данных и описание хранящихся в нем данных должно быть увязано с информационной моделью (ERD).

**Потоки данных**

Поток данных определяет информацию, передаваемую через некоторое соединение от источника к приемнику. Реальный поток данных может быть информацией, передаваемой по кабелю между двумя устройствами, пересылаемыми по почте письмами, магнитными лентами или дискетами, переносимыми с одного компьютера на другой и т.д.

Поток данных на диаграмме изображается линией, оканчивающейся стрелкой, которая показывает направление потока (рисунок 7). Каждый поток данных имеет имя, отражающее его содержание.

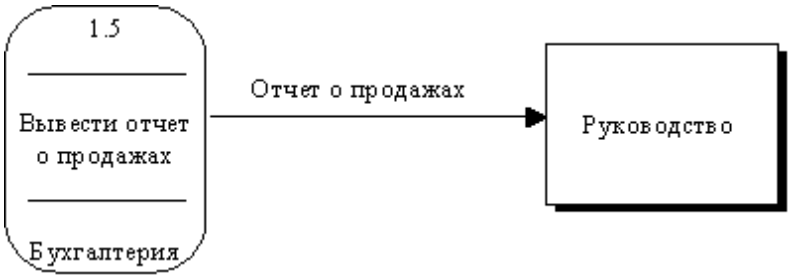



Рис. 7. Поток данных.

**Построение иерархии диаграмм потоков данных**

При построении иерархии потоков данных целесообразно пользоваться следующими рекомендациями:

- Размещать на каждой диаграмме от 3 до 6-7 процессов. Верхняя граница соответствует человеческим возможностям одновременного восприятия и понимания структуры сложной системы с множеством внутренних связей, нижняя граница выбрана по соображениям здравого смысла: нет необходимости детализировать процесс диаграммой, содержащей всего один или два процесса.

- Не загромождать диаграммы не существенными на данном уровне деталями.

- Декомпозицию потоков данных осуществлять параллельно с декомпозицией процессов. Эти две работы должны выполняться одновременно, а не после завершения другой.

- Выбирать ясные, отражающие суть дела имена процессов и потоков, при этом стараться не использовать аббревиатуры.

Первым шагом при построении иерархии DFD является построение контекстных диаграмм. Обычно при проектировании относительно простых ИС строится единственная контекстная диаграмма со звездообразной топологией, в центре которой находится так называемый главный процесс, соединенный с приемниками и источниками информации, посредством которых с системой взаимодействуют пользователи и другие внешние системы. Количество потоков на контекстной диаграмме должно быть по возможности небольшим, поскольку каждый из них может быть в дальнейшем разбит на несколько потоков на следующих уровнях диаграммы.

Внешние сущности выделяются по отношению к основному процессу. Для их определения необходимо выделить поставщиков и потребителей основного процесса, т.е. все объекты, которые взаимодействуют с основным процессом. На этом этапе описание взаимодействия заключается в выборе глагола, дающего представление о том, как внешняя сущность использует

основной процесс или используется им. Например, основной процесс – "учет обращений граждан", внешняя сущность – "граждане", описание взаимодействия – "подает заявления и получает ответы". Этот этап является принципиально важным, поскольку именно он определяет границы моделируемой системы.

Для всех внешних сущностей строится таблица событий, описывающая их взаимодействие с основным потоком. Таблица событий включает в себя наименование внешней сущности, событие, его тип (типичный для системы или исключительный, реализующийся при определенных условиях) и реакцию системы.

Если же для сложной системы ограничиться единственной контекстной диаграммой, то она будет содержать слишком большое количество источников и приемников информации, которые трудно расположить на листе бумаги нормального формата, и кроме того, единственный главный процесс не раскрывает структуры распределенной системы. Признаками сложности (в смысле контекста) могут быть:

·наличие большого количества внешних сущностей (десять и более);

·распределенная природа системы;

·многофункциональность системы с уже сложившейся или выявленной группировкой функций в отдельные подсистемы.

Для сложных ИС строится иерархия контекстных диаграмм. При этом контекстная диаграмма верхнего уровня содержит не единственный главный процесс, а набор подсистем, соединенных потоками данных. Контекстные диаграммы следующего уровня детализируют контекст и структуру подсистем.

Иерархия контекстных диаграмм определяет взаимодействие основных функциональных подсистем проектируемой ИС как между собой, так и с внешними входными и выходными потоками данных и внешними объектами (источниками и приемниками информации), с которыми взаимодействует ИС.

Разработка контекстных диаграмм решает проблему строгого определения функциональной структуры ИС на самой ранней стадии ее проектирования, что особенно важно для сложных многофункциональных систем, в разработке которых участвуют разные организации и коллективы разработчиков.

После построения контекстных диаграмм полученную модель следует проверить на полноту исходных данных об объектах системы и

изолированность объектов (отсутствие информационных связей с другими объектами).

Для каждой подсистемы, присутствующей на контекстных диаграммах, выполняется ее детализация при помощи DFD. Каждый процесс DFD, в свою очередь, может быть детализирован при помощи DFD или миниспецификации. При детализации должны выполняться следующие правила:

·правило балансировки - означает, что при детализации подсистемы или процесса детализирующая диаграмма в качестве внешних источников/приемников данных может иметь только те компоненты (подсистемы, процессы, внешние сущности, накопители данных), с которыми имеет информационную связь детализируемая подсистема или процесс на родительской диаграмме;

·правило нумерации - означает, что при детализации процессов должна поддерживаться их иерархическая нумерация. Например, процессы, детализирующие процесс с номером 12, получают номера 12.1, 12.2, 12.3 и т.д.

Миниспецификация (описание логики процесса) должна формулировать его основные функции таким образом, чтобы в дальнейшем специалист, выполняющий реализацию проекта, смог выполнить их или разработать соответствующую программу.

Миниспецификация является конечной вершиной иерархии DFD. Решение о завершении детализации процесса и использовании миниспецификации принимается аналитиком исходя из следующих критериев:

·наличия у процесса относительно небольшого количества входных и выходных потоков данных (2-3 потока);

·возможности описания преобразования данных процессом в виде последовательного алгоритма;

·выполнения процессом единственной логической функции преобразования входной информации в выходную;

·возможности описания логики процесса при помощи миниспецификации небольшого объема (не более 20-30 строк).

Спецификации должны удовлетворять следующим требованиям:

·Для каждого процесса нижнего уровня должна существовать одна и только одна спецификация.

·Спецификация должна определять способ преобразования входных потоков в выходные.

·Нет необходимости (по крайней мере на стадии формирования требований) определять метод реализации этого преобразования.

·Спецификация должна стремиться к ограничению избыточности - не следует переопределять то, что уже было определено на диаграмме.

·Набор конструкций для построения спецификаций должен быть простым и понятным.

Фактически спецификации представляют собой описания алгоритмов задач, выполняемых процессами. Спецификации содержат:

·Номер и/или имя процесса.

·Списки входных и выходных данных.

·Тело (описание процесса), являющееся спецификацией алгоритма или операции, трансформирующей входные потоки данных в выходные.

При построении иерархии диаграмм потоков данных переходить к детализации процессов следует только после определения содержания всех потоков и накопителей данных, которое описывается при помощи структур данных. Для каждого потока данных формируется список всех его элементов данных, затем элементы данных объединяются в структуры данных, соответствующие более крупным объектам данных (например, строкам документов или объектам предметной области). Каждый объект должен состоять из элементов, являющихся его атрибутами. Структуры данных могут содержать **альтернативы**, **условные вхождения** и **итерации**.

Условное вхождение означает, что данный компонент может отсутствовать в структуре (например, структура «данные о страховании» для объекта «служащий»).

Альтернатива означает, что в структуру может входить один из перечисленных элементов.

Итерация означает вхождение любого числа элементов в указанном диапазоне (например, элемент «имя ребенка» для объекта «служащий»).

Для каждого элемента данных может указываться его тип (непрерывные или дискретные данные). Для непрерывных данных может указываться единица измерения (кг, см и т.п.), диапазон значений, точность представления и форма физического кодирования. Для дискретных данных может указываться таблица допустимых значений.

После декомпозиции основного процесса для каждого подпроцесса строится аналогичная таблица внутренних событий.

Следующим шагом после определения полной таблицы событий выделяются **потоки данных**, которыми обмениваются процессы и внешние сущности. Простейший способ их выделения заключается в анализе таблиц событий. События преобразуются в потоки данных от инициатора события к запрашиваемому процессу, а реакции – в обратный поток событий. После построения входных и выходных потоков аналогичным образом строятся внутренние потоки. Для их выделения для каждого из внутренних процессов выделяются поставщики и потребители информации. Если поставщик или потребитель информации представляет процесс сохранения или запроса информации, то вводится хранилище данных, для которого данный процесс является интерфейсом.

После построения законченной модели системы ее необходимо верифицировать (проверить на полноту и согласованность). В полной модели все ее объекты (подсистемы, процессы, потоки данных) должны быть подробно описаны и детализированы. Выявленные недетализированные объекты следует детализировать, вернувшись на предыдущие шаги разработки. В согласованной модели для всех потоков данных и накопителей данных должно выполняться правило сохранения информации: все поступающие куда-либо данные должны быть считаны, а все считываемые данные должны быть записаны. Непротиворечивость системы обеспечивается выполнением наборов формальных правил о возможных типах процессов: на диаграмме не может быть потока, связывающего две внешние сущности – это взаимодействие удаляется из рассмотрения; ни одна сущность не может непосредственно получать или отдавать информацию в хранилище данных – хранилище данных является пассивным элементом, управляемым с помощью интерфейсного процесса; два хранилища данных не могут непосредственно обмениваться информацией – эти хранилища должны быть объединены.

Кроме основных элементов, в состав DFD входят **словари данных**, которые являются каталогами всех элементов данных, присутствующих в DFD, включая групповые и индивидуальные потоки данных, хранилища и процессы, а также все их атрибуты.

К преимуществам методики DFD относятся:

·возможность однозначно определить внешние сущности, анализируя потоки информации внутри и вне системы;

·возможность проектирования сверху вниз, что облегчает построение модели "как должно быть";

·наличие спецификаций процессов нижнего уровня, что позволяет преодолеть логическую незавершенность функциональной модели и построить полную функциональную спецификацию разрабатываемой системы.

К недостаткам модели отнесем: необходимость искусственного ввода управляющих процессов, поскольку управляющие воздействия (потоки) и управляющие процессы с точки зрения DFD ничем не отличаются от обычных; отсутствие понятия времени, т.е. отсутствие анализа временных промежутков при преобразовании данных (все ограничения по времени должны быть введены в спецификациях процессов).

На рис. 8 приведен пример DFD-схемы бизнес-процесса "Оформлении и выдача трудовой книжки сотруднику при увольнении", разработанной в нотации Гейна-Сарсона, а на **рис. 9-в нотации Йордона-Де Марко.**


**Рис. 8. DFD-схема бизнес-процесса "Оформлении и выдача трудовой книжки сотруднику при увольнении" в нотации Гейна-Сарсона.**

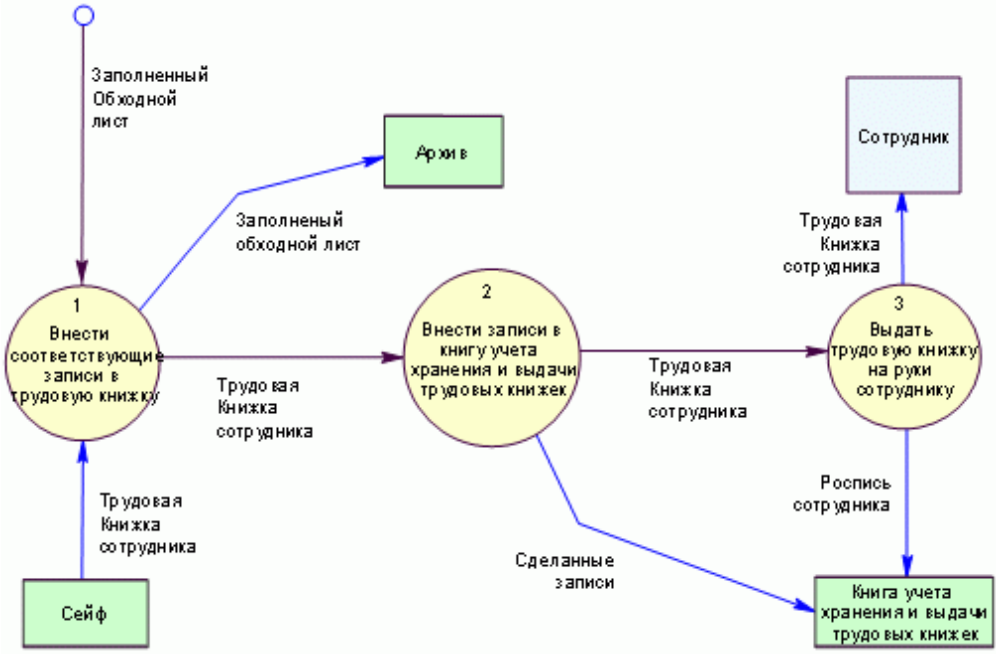



**Рис. 9. DFD-схема бизнес-процесса "Оформлении и выдача трудовой книжки сотруднику при увольнении" в нотации Йордона-Де Марко.**

Лекция 7

**ТЕМА:Описание функциональности разработки: нотация IDEF3.**

Этот метод предназначен для моделирования **последовательности выполнения действий** и взаимозависимости между ними в рамках процессов. Модели IDEF3 могут использоваться для детализации функциональных блоков IDEF0, не имеющих диаграмм декомпозиции.

Нотация IDEF3 использует категорию **Сценариев** (Scenario) для упрощения структуры описаний сложного многоэтапного процесса. IDEF3 осуществляет реализацию следующей информации о процессе:

·объекты, участвующие в описании операции;

·функции, которые выполняют эти объекты;

·взаимосвязь между процессами;

·состояния и изменения, которым подвергаются объекты;

·время выполнения и контрольные точки синхронизации работ;

·ресурсы, необходимые для выполнения работ.

Существует два типа диаграмм в стандарте IDEF3:

1. PFDD - диаграммы описания последовательности этапов процесса.

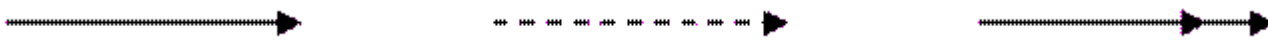
2. OSTN - диаграммы состояния объекта и его изменений в процессе.

На рис. 10 изображена диаграмма PFDD, показывающая процессы создания программного обеспечения. Прямоугольники на диаграмме PFDD называются функциональными элементами или элементами поведения (UOB) и обозначают событие, стадию процесса или принятие решения (рис. 11). Каждый UOB имеет конкретное имя (функция, процесс, действие, акт, событие, сценарий, процедура, операция, решение), отображаемое в глагольном наклонении и уникальный номер (номер действия обычно предваряется номером его родителя, например, 1.1.). В правом нижнем углу UOB элемента располагается ссылка на какие-либо элементы функциональной модели IDEF0 или на отделы, конкретных исполнителей, выполняющие конкретный процесс.

Рис. 10. PFDD-диаграмма создания электронной программы.


Рис. 11. Функциональный элемент (UOB).

Стрелки или линии являются отображением хода выполнения операций между UOB-блоками в ходе процесса (рис. 12).



а) б) в)

Рис. 12. Стрелки для отображения хода выполнения операции

Линии в нотации IDEF3 бывают следующих видов:

1. **Временное предшествование** или **старшая** (Temporal precedence, рис. 12, а) - сплошная линия, связывающая UOB. Рисуется слева направо или сверху вниз. Исходное действие должно завершиться, прежде чем конечное действие сможет начаться.

2. **Нечеткое отношение** (Relationship link, рис. 12, б) - пунктирная линия, использующаяся для изображения связей между UOB в том случае, если конечное действие сможет начаться и даже завершиться до того момента, когда завершится исходное действие.

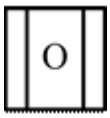
3. **Объектный поток** (Object flow, рис. 12, в) - стрелка с двумя наконечниками используется для описания того факта, что объект (деталь) используется в двух или более единицах работы, например, когда объект порождается в одной работе и используется в другой (т.е. выход исходного действия является входом конечного действия). Исходное действие должно завершиться, прежде чем конечное действие сможет начаться. Наименования потоковых связей должны чётко идентифицировать объект, который передается с их помощью.

Все **связи** в IDEF3 являются однонаправленными.

Завершение одного действия может инициировать начало выполнения сразу нескольких других действий, или наоборот, определенное действие может требовать завершения нескольких других действий до начала своего выполнения (ветвление процесса). Ветвление процесса отражается с помощью специальных блоков, называемых перекрестками. Каждый перекресток (Junction) имеет свой определенный идентификационный номер (на рис. 10 J1 - перекресток). Перекресток не может использоваться одновременно для слияния и для разветвления. При вводе перекрестка в диаграмму необходимо указать тип перекрестка. Типы перекрестков представлены в таблице 2.

Таблица 2. Описание типов перекрестков.

| Обозначение | Наименование | Смысл для стрелок слияния | Смыл для стрелок разветвления |
|---|---|---|---|

| | | | |
|---|---|---|---|
|  | Асинхронное И (asynchronous AND) | Все предшествующие процессы должны быть завершены | Все следующие процессы должны быть запущены |
|  | Синхронное И (synchronous AND) | Все предшествующие процессы завершены одновременно | Все следующие процессы запускаются одновременно |
|  | Асинхронное ИЛИ (asynchronous OR) | Один или несколько предшествующих процессов должны быть завершены | Один или несколько следующих процессов должны быть запущены |
|  | Синхронное ИЛИ (synchronous OR) | Один или несколько предшествующих процессов завершаются одновременно | Один или несколько следующих процессов запускаются одновременно |
|  | Исключающее ИЛИ(XOR, exclusive OR) | Только один предшествующий процесс завершен | Только один следующий процесс запускается |

**Пояснение:** *Перекресток "Исключающий ИЛИ" обозначает, что после завершения работы "A" (рис. 13), начинает выполняться только одна из трех расположенных параллельно работ B, C или D в зависимости от условий 1, 2 и 3. Перекресток "И" обозначает, что после завершения работы "A", начинают выполняться одновременно три параллельно расположенные работы B, C и D. Перекресток "ИЛИ" обозначает, что после завершения работы "A", может запуститься любая комбинация трех параллельно расположенных работ B, C и D. Например может запуститься только одна из них, могут запуститься три работы, а также могут запуститься двойные комбинации B и C, либо C и D, либо B и D. Перекресток "Исключающий ИЛИ" является самым неопределенным, так как предполагает несколько возможных сценариев реализации бизнес-процесса и применяется для описания слабо формализованных ситуаций.*

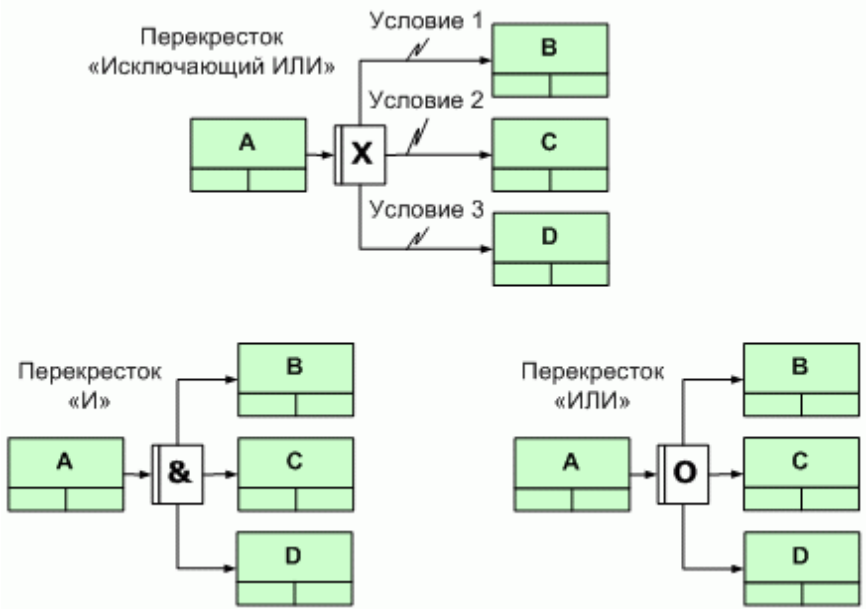



*Рис. 13. Применение перекрестков "Исключающий ИЛИ", "И" и "ИЛИ" - схемы расхождения.*

**4**

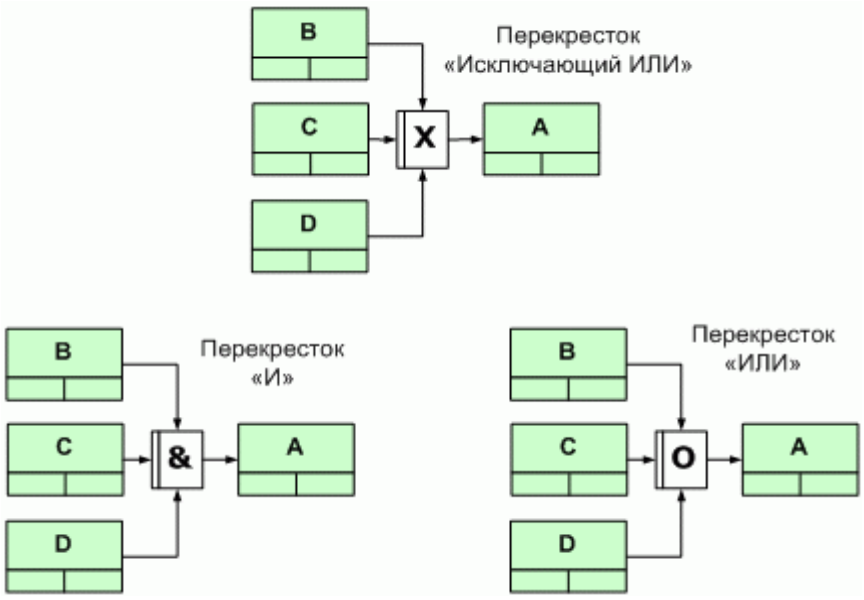



*Рис. 14. Применение перекрестков "Исключающий ИЛИ", "И" и "ИЛИ" - схемы схождения.*

Сценарий, отображаемый на диаграмме (рис. 10), можно описать в следующем виде. Программный код, подготовленный к компиляции, компилируется в компиляторе программ. В процессе компиляции создается исполнительный файл программы. После этого, производится тестирование программы, после которой начинается этап проверки программного продукта. Если тест подтверждает недостаточное качество программы, то она заново

пропускается через этап создания программного кода. Если программа успешно проходит контроль качества, то она отправляется пользователю.

Метод IDEF3 позволяет декомпозировать действие несколько раз, что обеспечивает документирование альтернативных потоков процесса в одной модели.

Каждый функциональный блок UOB может иметь последовательность декомпозиций. Номера UOB дочерних диаграмм имеют сквозную нумерацию, т.е., если родительский UOB имеет номер "1", то блоки UOB на его декомпозиции будут соответственно иметь номера "1.1", "1.2" и т.д.

Если диаграммы PFDD представляют технологический процесс "С точки зрения наблюдателя", то другой класс диаграмм IDEF3 - OSTN позволяет рассматривать тот же самый процесс "С точки зрения объекта". На рис. 15 представлено отображение процесса создания электронной программы с точки зрения OSTN диаграммы. Состояния объекта (в нашем случае электронной программы) и изменение состояния являются ключевыми понятиями OSTN диаграммы. Состояния объекта отображаются окружностями, а их изменения направленными линиями. Каждая линия имеет ссылку на соответствующий функциональный блок UOB, в результате которого произошло отображаемое ей изменение состояния объекта.

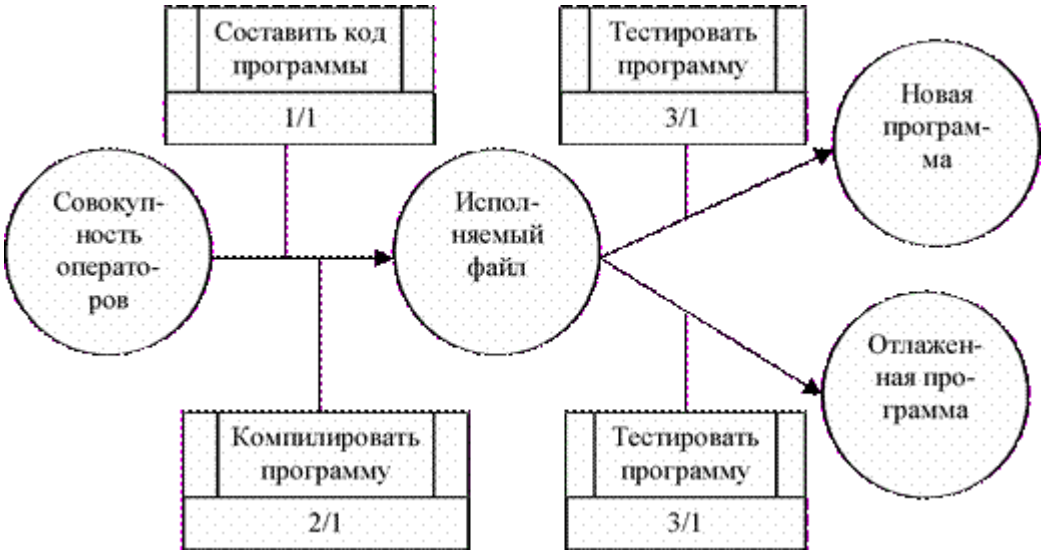



Рис. 15. Пример OSTN-диаграммы создания электронной программы.

На схеме бизнес-процесса также можно использовать такой элемент как "объект- ссылка", который связывается с работами и перекрестками. Ссылки обеспечивают более полное понимание, дополнительный смысл и упрощение описания процесса. Ссылки позволяют:

·обращаться к ранее определенному действию;

·организовывать циклы;

·уточнять работу перекрестков;

·связывать элементы диаграммы с каким-либо внешним объектом;

·комментировать различные элементы диаграммы.

Ссылка изображается в виде прямоугольника. В верхней его части указывается тип ссылки и ее имя.

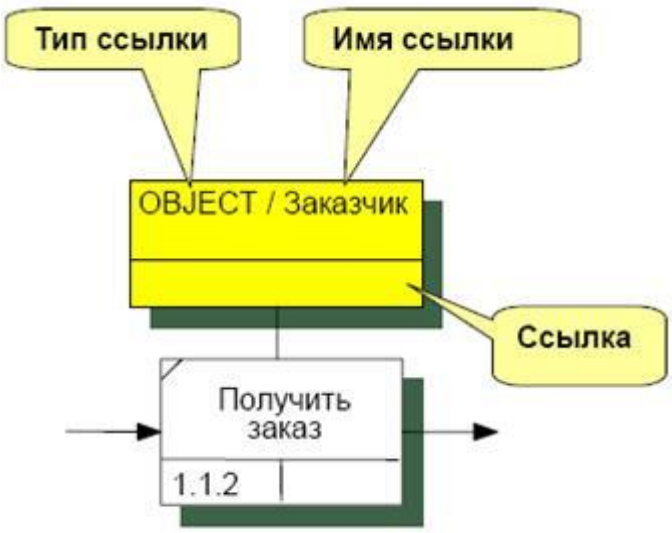



Рис. 16. Ссылки

Ссылки могут быть различного типа. Список типов ссылок приведен в таблице 3.

Таблица 3. Типы ссылок.

| Тип ссылки | Назначение |
|---|---|
| OBJECT | Описывает участие важного объекта в действии. |
| GOTO | Позволяет применять на диаграмме циклический переход. В том случае, когда все действия цикла находятся в рамках одной диаграммы, цикл можно изобразить стрелкой, которая будет указывать на начало цикла. Тогда ссылка будет связана с перекрестком, управляющим циклом. |
| UOB | Предназначена для многократного вызова какого-либо действия в рамках одной модели |

| | |
|---|---|
| NOTE | Позволяет прокомментировать присутствие какого-либо элемента на диаграмме. |
| ELAB (elaboration) | Применяется для уточнения использования ветвления стрелок на перекрестках. |

Примеры использования ссылок различного типа приведены на рис. 17.

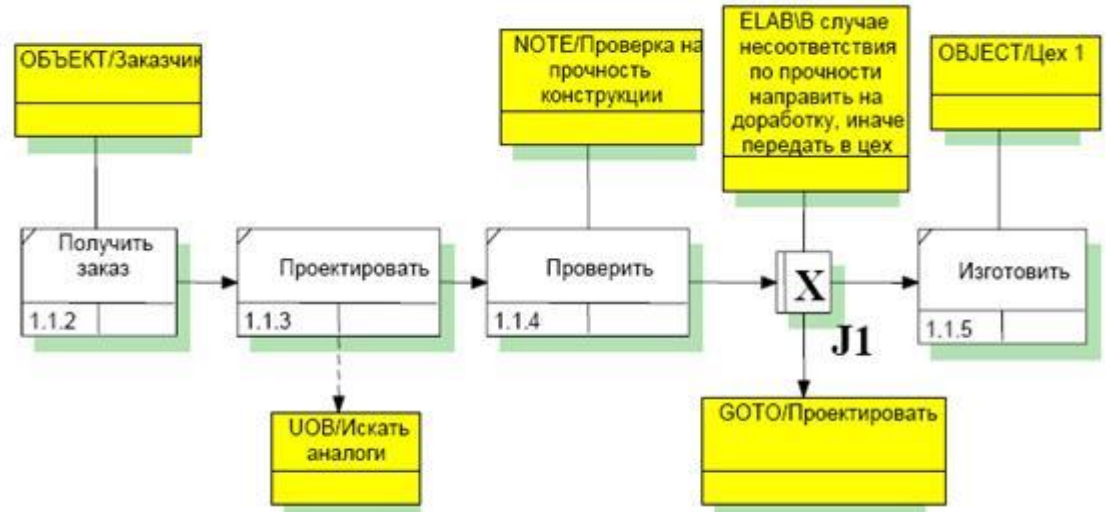



Рис. 17. Примеры использования ссылок различного типа

Таким образом, можно сделать следующие выводы по практическому использованию: применение универсальных графических языков моделирования IDEF0, IDEF3 и DFD обеспечивает логическую целостность и полноту описания, необходимую для достижения точных и непротиворечивых результатов на этапе анализа.

По диаграммам делаем следующий вывод: наиболее существенное различие между разновидностями структурного анализа заключается в их функциональности.

Модели SADT (IDEF0) наиболее удобны при построении функциональных моделей. Они наглядно отражают функциональную структуру объекта: производимые действия, связи между этими действиями. Таким образом, четко прослеживается логика и взаимодействие процессов организации. Главным достоинством нотации является возможность получить полную информацию о каждой работе, благодаря ее жестко регламентированной структуре. С ее помощью можно выявить все недостатки, касающиеся как самого процесса, так и то, с помощью чего он реализуется: дублирование функций, отсутствие механизмов, регламентирующих данный процесс, отсутствие контрольных переходов и т.д.

DFD позволяет проанализировать информационное пространство системы и используется для описания документооборота и обработки информации. Поэтому, диаграммы DFD применяют в качестве дополнения модели бизнес-процессов, выполненной в IDEF0.

IDEF3 хорошо приспособлен для сбора данных, требующихся для проведения анализа системы с точки зрения рассогласования/согласования процессов во времени.

Нельзя говорить о достоинствах и недостатках отдельных нотаций. Возможны ситуации, при которых анализ IDEF0 не обнаружил недостатков в деятельности организации с точки зрения технологического или производственного процесса, однако это не является гарантией отсутствия ошибок. Поэтому в следующем этапе анализа необходимо перейти к исследованию информационных потоков с помощью DFD и затем объединить эти пространства с помощью последней нотации - IDEF3.

Сегодня для описания функциональности разработки предлагаются различные инструменты. Например:

· CASE-средство**AllFusion Process Modeler** (BPwin) компании Computer Associates. AllFusion Process Modeler нарядус ERwin Data Modeler (ERwin), входитвсоставпакетапрограммныхсредств AllFusion Modeling Suite. Основным преимуществом данного инструмента является присутствие нотаций IDEF и DFD, которые распространены в российских компаниях и принята в России на уровне стандарта, а также связь с продуктом Erwin, используемым для проектирования структур данных.

· CASE-средство**Silverrun**американскойфирмыComputer Systems Advisers, Inc. (CSA) используется для анализа и проектирования ИС бизнес-класса [22] и ориентировано в большей степени на спиральную модель ЖЦ. Оно применимо для поддержки любой методологии, основанной на раздельном построении функциональной и информационной моделей (диаграмм потоков данных и диаграмм "сущность-связь").

· CASE.Аналитик 1.1 является практически единственным в настоящее время конкурентоспособным отечественным CASE-средством функционального моделирования. Его основные функции:

о построение и редактирование DFD;

о анализ диаграмм и проектных спецификаций на полноту и непротиворечивость;

о получение разнообразных отчетов по проекту;

о генерация макетов документов в соответствии с требованиями ГОСТ 19.XXX и 34.XXX.

· Также существует множество программ для рисования различных диаграмм, например, MS Visio или Dia.

Лекция 8

**ТЕМА:Проектирование с использованием метода «сущность-связь».**

В реальном проектировании структуры базы данных применяется так называемое, ***семантическое моделирование***. Семантическое моделирование представляет собой моделирование структуры данных, опираясь на смысл этих данных. В качестве инструмента семантического моделирования используются различные варианты ***диаграмм сущность-связь*** (***ER - Entity-Relationship***), которые могут быть относительно легко отображены в любую систему баз данных.

Первый вариант модели сущность-связь был предложен в 1976 г. Питером Пин-Шэн Ченом. В дальнейшем многими авторами были разработаны свои варианты подобных моделей (нотация Мартина, нотация IDEF1X, нотация Баркера и др.). Кроме того, различные CASE-средства используют несколько отличающиеся друг от друга нотации ERD. Одна из наиболее распространенных нотаций предложена Баркером и используется в Oracle Designer. В CASE-средстве SilverRun используется один из вариантов нотации Чена. CASE-средства ERwin, ER / Studio, Design / IDEF используют методологию IDEF1X. По сути, все варианты диаграмм сущность-связь исходят из одной идеи - рисунок всегда нагляднее текстового описания. Все такие диаграммы используют графическое изображение сущностей предметной области, их свойств (атрибутов), и взаимосвязей между сущностями.

Мы опишем работу с ER-диаграммами близко к нотации Баркера, как довольно легкой в понимании основных идей.

**Основные понятия ER-диаграмм**

**Определение 1.Сущность (Entity)**- это реальный или представляемый объект, имеющий существенное значение для рассматриваемой предметной области, информация о котором должна сохраняться и быть доступна.

Каждая сущность должна иметь уникальное наименование, выраженное существительным в единственном числе. Примерами сущностей могут быть такие классы объектов как "Поставщик", "Сотрудник", "Накладная". Каждая сущность в модели изображается в виде прямоугольника с наименованием (рис.1). При этом имя сущности – это имя типа, а не некоторого конкретного экземпляра этого типа.

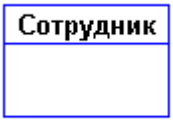



Рис. 1. Обозначение сущности.

Для сущностей различают **тип** сущности и **экземпляр**. Тип характеризуется именем и списком свойств, а экземпляр – конкретными значениями свойств.

**Определение 2. Экземпляр сущности-** это конкретный представитель данной сущности.

Например, представителем сущности "Сотрудник" может быть "Сотрудник Иванов".

Каждый экземпляр сущности должен быть отличим от любого другого экземпляра той же сущности, т.е. сущности должны иметь некоторые свойства, уникальные для каждого экземпляра этой сущности (это требование в некотором роде аналогично требованию отсутствия кортежей-дубликатов в реляционных таблицах).

Типы сущностей можно классифицировать как сильные и слабые. Сильные сущности существуют сами по себе, а существование слабых сущностей зависит от существования сильных. Например, читатель библиотеки – сильная сущность, а абонемент этого читателя – слабая, которая зависит от наличия соответствующего читателя. Слабые сущности называют подчинёнными (дочерними), а сильные – базовыми (основными, родительскими).

Также сущности разделяют на **независимые**и**зависимые.**Сущность является **независимой**, если каждый экземпляр ее может быть однозначно идентифицирован без определения его отношений с другими сущностями. Независимая сущность изображается прямоугольником с четко выраженными углами. Сущность является **зависимой**, если однозначная идентификация экземпляра сущности зависит от его отношения к другой сущности. Зависимая сущность изображается прямоугольником со скругленными углами.

Сущности бывают как физически существующие (например, *СОТРУДНИК* или *АВТОМОБИЛЬ*), так и абстрактные (например, *ЭКЗАМЕН* или *ДИАГНОЗ*).

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

**Определение 3. Атрибут сущности-** это именованная характеристика, являющаяся некоторым значимым для рассматриваемой предметной области свойством сущности.

Наименование атрибута должно быть выражено существительным в единственном числе (возможно, с характеризующими прилагательными).

Примерами атрибутов сущности "Сотрудник" могут быть такие атрибуты как "Табельный номер", "Фамилия", "Имя", "Отчество", "Должность", "Зарплата" и т.п.

Атрибуты изображаются в пределах прямоугольника определяющего сущность, причем каждый атрибут занимает отдельную строку, и отделяются от названия сущности линией (рис. 2).

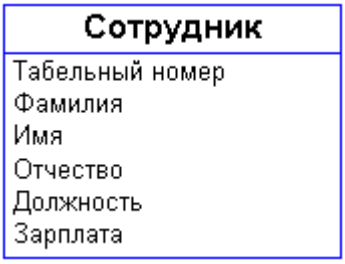



Рис. 2. Указание атрибутов сущности.

Рядом с именем атрибута можно приводить примеры значений данного атрибута.

Различают:

1. ***Идентифицирующие и описательные атрибуты***. Идентифицирующие атрибуты имеют уникальное значение для сущностей данного типа и являются ***потенциальными ключами***. Они позволяют однозначно распознавать экземпляры сущности. Из потенциальных ключей выбирается один первичный ключ. В качестве первичного ключа обычно выбирается потенциальный ключ, по которому чаще происходит обращение к экземплярам сущности. Кроме того, ПК должен включать в свой состав минимально необходимое для идентификации количество атрибутов. Остальные атрибуты называются описательными и заключают в себе интересующие свойства сущности.

2. ***Составные и простые атрибуты***. Простой атрибут состоит из одного компонента, его значение неделимо. Составной атрибут является комбинацией нескольких компонентов, возможно, принадлежащих разным типам данных (например, ФИО или адрес). Решение о том, использовать составной атрибут или разбивать его на компоненты, зависит от характера его обработки и формата пользовательского представления этого атрибута.

3. ***Однозначные и многозначные атрибуты.***Могут иметь соответственно одно или много значений для каждого экземпляра сущности. На диаграмме изображаются с двойным подчеркиванием.

4. ***Основные и производные атрибуты.***Значение основного атрибута не зависит от других атрибутов. Значение производного атрибута вычисляется на основе значений других атрибутов (например, возраст студента вычисляется на основе даты его рождения и текущей даты).

Спецификация атрибута состоит из его названия, указания типа данных и описания ограничений целостности – множества значений (или домена), которые может принимать данный атрибут.

Атрибут может быть либо обязательным, либо необязательным. Обязательность означает, что атрибут не может принимать неопределенных значений (Null). Обязательный атрибут помечается звездочкой, а необязательный - кружком (рис. 3).


Рис. 3. Указание обязательных и необязательных атрибутов сущности.

**Определение 4.Ключ сущности-** это неизбыточный набор атрибутов, значения которых в совокупности являются уникальными для каждого экземпляра сущности. Неизбыточность заключается в том, что при удалении любого атрибута из ключа нарушается его уникальность.

Сущность может иметь несколько различных ключей. Различают **первичный, альтернативный и внешний**ключи.

**Первичный ключ (Primary Key)** -это атрибут или группа атрибутов, используемых для однозначной идентификации экземпляра сущности. На диаграмме атрибуты первичного ключа размещаются первыми в списке атрибутов и подчеркиваются или предваряются # (рис. 4):

Рис. 4. Указание ключевого атрибута.

**Альтернативный ключ (Alternate Key)**-потенциальный ключ, не ставший первичным. На диаграмме альтернативный ключ обозначается **AK n. m,** где n - порядковый номер ключа, m - порядковый номер атрибута в ключе.

**Внешние ключи (Foreign Key)**создаются, когда сущности соединяются связью при построениифизических ER-диаграмм. Происходит миграция атрибутов первичного ключа родительской сущности в дочернюю сущность. Появившийся таким образом в дочерней сущности атрибут будет являться внешним ключом. (Из родительской сущности атрибут не исчезает, а просто копируется в дочерней сущности). Внешний ключ обозначается ВК.

Первичный ключ может быть**абсолютный или относительный**. Если все атрибуты, составляющие первичный ключ, принадлежат сущности, то он является абсолютным. Если один или более атрибутов первичного ключа принадлежат другой сущности, то он является относительным. Когда первичный ключ является относительным, сущность определяется как **зависимая** сущность, поскольку ее идентификатор зависит от другой сущности. В примере на рисунке 5 первичный ключ сущности Компания является относительным. Он включает первичный ключ сущности Список компаний.


Рис. 5. Относительный первичный ключ.

**Определение 5. Связь-** это графически изображаемая ассоциация, устанавливаемая между двумя сущностями, значимая для рассматриваемой предметной области.

Эта ассоциация всегда является бинарной и может существовать между двумя разными сущностями или между сущностью и ей же самой (рекурсивная связь).

Определение связи в методе Баркера несколько отличается от данного Ченом.

**Определение 5.1. Связь-** это ассоциация между сущностями, при которой, как правило, каждый экземпляр одной сущности, называемой ***родительской сущностью***, ассоциирован с произвольным (в том числе и нулевым) количеством экземпляров второй сущности, называемой ***сущностью-потомком***, а каждый экземпляр сущности-потомка ассоциирован в точности с одним экземпляром сущности-родителя. Таким образом, экземпляр сущности-потомка может существовать только при существовании сущности-родителя.

Связи позволяют по одной сущности находить другие сущности, связанные с нею.

В любой связи выделяются два конца (в соответствии с существующей парой связываемых сущностей), на каждом из которых указывается:

· имя конца связи,

· тип конца связи (сколько экземпляров данной сущности связывается),

· обязательность связи (т.е. любой ли экземпляр данной сущности должен участвовать в данной связи).

Связь представляется в виде линии, связывающей две сущности или ведущей от сущности к ней же самой. При этом в месте "стыковки" связи с сущностью используется трехточечный вход в прямоугольник сущности, если для этой сущности в связи могут использоваться много (many) экземпляров сущности, и одноточечный вход, если в связи может участвовать только один экземпляр сущности. Обязательный конец связи изображается сплошной линией, а необязательный – прерывистой линией (рис. 6).

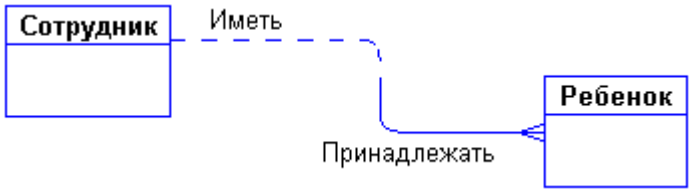



Рис. 6. Изображение связи между сущностями.

Наименование связи обычно выражается в неопределенной глагольной форме: "иметь", "принадлежать" и т.п. Каждое из наименований относится к своему концу связи. Иногда наименования не пишутся ввиду их очевидности. Имя каждой связи между двумя сущностями должно быть уникальным, но имена связей в модели не обязаны быть уникальными. Имя связи всегда формируется с точки зрения родителя, так что может быть образовано

предложение соединением имени сущности-родителя, имени связи, выражения степени и имени сущности-потомка.

Каждая связь может иметь один из следующих **типов** связи (рис. 7):

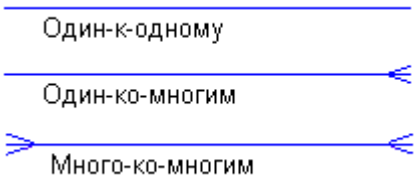



Рис. 7. Типы связей и их изображение на диаграмме.

Связь типа **один-к-одному** означает, что один экземпляр первой сущности (левой) связан с одним экземпляром второй сущности (правой). Связь один-к-одному чаще всего свидетельствует о том, что на самом деле мы имеем всего одну сущность, неправильно разделенную на две.

Связь типа **один-ко-многим** означает, что один экземпляр первой сущности (левой) связан с несколькими экземплярами второй сущности (правой). Это наиболее часто используемый тип связи. Левая сущность (со стороны "один") называется **родительской**, правая (со стороны "много") - **дочерней**. Характерный пример такой связи приведен на рис. 6.

Связь типа **много-ко-многим** означает, что каждый экземпляр первой сущности может быть связан с несколькими экземплярами второй сущности, и каждый экземпляр второй сущности может быть связан с несколькими экземплярами первой сущности. Тип связи много-ко-многим является **временным** типом связи, допустимым на ранних этапах разработки модели. В дальнейшем этот тип связи должен быть заменен двумя связями типа **один-ко-многим** путем создания промежуточной сущности.

Каждая связь может иметь одну из двух **модальностей**связи (рис. 8):

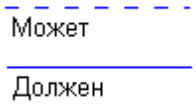



Рис. 8. Модальность связи и ее изображение на диаграмме.

Модальность "**может**" означает, что экземпляр одной сущности ***может быть связан*** с одним или несколькими экземплярами другой сущности, ***а может быть и не связан*** ни с одним экземпляром.

Модальность "**должен**" означает, что экземпляр одной сущности ***обязан быть связан не менее чем с одним***экземпляром другой сущности.

Связь может иметь разную модальность с разных концов (как на рис. 6).

Тип связи в совокупности с модальностью называют еще **кардинальностью** связи (табл. 1):

Таблица 1. Кардинальность связи.

| Обозначение | Кардинальность |
|---|---|
|  | 0,1 |
|  | 1,1 |
|  | 0,N |
|  | 1,N |

Описанный графический синтаксис позволяет ***однозначно*** читать диаграммы, пользуясь следующей схемой построения фраз:

<Каждый экземпляр СУЩНОСТИ 1><МОДАЛЬНОСТЬ СВЯЗИ><НАИМЕНОВАНИЕ СВЯЗИ><ТИП СВЯЗИ><экземпляр СУЩНОСТИ 2>.

Каждая связь может быть прочитана как слева направо, так и справа налево. Связь на рис. 6 читается так:

Слева направо: "каждый сотрудник может иметь несколько детей".

Справа налево: "Каждый ребенок обязан принадлежать ровно одному сотруднику".

На следующем примере (рис. 9) изображена рекурсивная связь, связывающая сущность ЧЕЛОВЕК с ней же самой. Конец связи с именем "являться сыном" определяет тот факт, что один человек может быть сыном не более чем одному отцу. Конец связи с именем "являться отцом" означает, что один человек может являться отцом для одного или более ЛЮДЕЙ (т.е не каждый человек имеет детей).

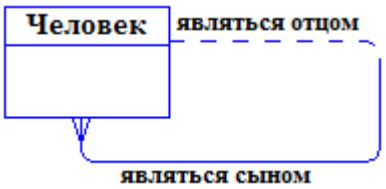



Рисунок 9. Пример рекурсивной связи.

**Более сложные элементы ER-модели**

Ранее были представлены только основные и наиболее очевидные понятия ER-модели данных. К числу более сложных элементов модели относятся следующие:

· **Подтипы и супертипы сущностей**. Как в языках программирования с развитыми типовыми системами (например, в языках объектно-ориентированного программирования), вводится возможность наследования типа сущности, исходя из одного или нескольких супертипов. Интересные нюансы связаны с необходимостью графического изображения этого механизма.

· **Уточняемые степени связи**. Иногда бывает полезно определить возможное количество экземпляров сущности, участвующих в данной связи (например, служащему разрешается участвовать не более, чем в трех проектах одновременно). Для выражения этого семантического ограничения разрешается указывать на конце связи ее максимальную или обязательную степень.

· **Каскадные удаления экземпляров сущностей**. Некоторые связи бывают настолько сильными (конечно, в случае связи "один-ко-многим"), что при удалении опорного экземпляра сущности (соответствующего концу связи "один") нужно удалить и все экземпляры сущности, соответствующие концу связи "многие". Соответствующее требование "каскадного удаления" можно сформулировать при определении сущности.

· **Домены.** Как и в случае реляционной модели данных бывает полезна возможность определения потенциально допустимого множества значений атрибута сущности (домена).

Эти и другие более сложные элементы модели данных "Сущность-Связи" делают ее существенно более мощной, но одновременно несколько усложняют ее использование.

**Разработка ER- моделей.**

Семантическое моделирование начинается с разбивки предметной области на ряд локальных областей, каждая из которых (в идеале) включает в

себя информацию, достаточную для обеспечения запросов отдельной группы будущих пользователей или решения отдельной задачи (подзадачи). Каждое локальное представление моделируется отдельно, затем они объединяются.

Выбор локального представления зависит от масштабов предметной области. Обычно она разбивается на локальные области таким образом, чтобы каждая из них соответствовала отдельному внешнему приложению и содержала 6-7 сущностей.

Разработка ERD включает следующие основные этапы:

· Идентификация сущностей, их атрибутов, а также первичных и альтернативных ключей.

· Идентификация отношений между сущностями и указание типов отношений.

· Разрешение неспецифических отношений (отношений многие-ко-многим).

Процесс выделения сущностей, атрибутов и связей является итерационным. Разработав первый приближенный вариант диаграмм, мы уточняем их, опрашивая экспертов предметной области. При этом документацией, в которой фиксируются результаты бесед, являются сами ER-диаграммы.

Как и в реляционных схемах баз данных, в ER -диаграммах вводится понятие нормальных форм, причем их смысл очень близко соответствует смыслу реляционных нормальных форм. Приведем только краткие и неформальные определения трех первых нормальных форм.

В **первой нормальной форме ER-схемы** устраняются повторяющиеся атрибуты или группы атрибутов, т.е. производится выявление неявных сущностей, "замаскированных" под атрибуты.

Во **второй нормальной форме**устраняются атрибуты, зависящие только от части уникального идентификатора. Эта часть уникального идентификатора определяет отдельную сущность.

В **третьей нормальной форме** устраняются атрибуты, зависящие от атрибутов, не входящих в уникальный идентификатор. Эти атрибуты являются основой отдельной сущности.

После того, как созданы локальные представления, выполняется их объединение. При небольшом количестве локальных областей (не более пяти)

они объединяются за один шаг. В противном случае обычно выполняют бинарное объединение в несколько этапов.

При объединении проектировщик может формировать конструкции, производные по отношению к тем, которые были использованы в локальных представлениях. Такой подход может преследовать следующие цели:

· объединение в единое целое фрагментарных представлений о различных свойствах одного и того же объекта;

· введение абстрактных понятий, удобных для решения задач системы, установление их связи с конкретными понятиями, использованными в модели;

· образование классов и подклассов подобных объектов (например, класс "изделие" и подклассы типов изделий, производимых на предприятии).

На этапе объединения необходимо выявить и устранить все противоречия. Например, одинаковые названия семантически различных объектов или связей или несогласованные ограничения целостности на одни и те же атрибуты в разных приложениях. Устранение противоречий вызывает необходимость возврата к этапу моделирования локальных представлений с целью внесения в них соответствующих изменений.

По завершении объединения результаты проектирования являют собой концептуальную инфологическую модель предметной области. Модели локальных представлений – это внешние инфологические модели.

**Концептуальные и физические ER-диаграммы.**

Различают **концептуальные** и **физические** ER-диаграммы. Концептуальные диаграммы не учитывают особенностей конкретных СУБД. Физические диаграммы строятся по концептуальным и представляют собой прообраз конкретной базы данных. Сущности, определенные в концептуальной диаграмме становятся таблицами, атрибуты становятся колонками таблиц (при этом учитываются допустимые для данной СУБД типы данных и наименования столбцов), связи реализуются путем **миграции** ключевых атрибутов родительских сущностей и создания внешних ключей.

При правильном определении сущностей, полученные таблицы будут сразу находиться в 3НФ.

Лекция 9.

**ТЕМА:Определение языка разработки, среды реализации, инструментов разработки.**

***Программная среда разработки пользовательской программы***

Программную среду (программное окружение) разработки пользовательской программы составляет совокупность программных средств (системных программ), используемых при создании и исполнении программы в данной аппаратно-операционной среде.

В понятие аппаратно-операционной среды входит набор устройств компьютера и средств операционной системы. Основные устройства персонального компьютера:

· ***Процессор***- выполняет выборку команд программы, выборку аргументов команды, ее исполнение и отсылку на запоминание полученных результатов.

· ***Оперативная память***- служит для хранения кода программы и ее данных. Представляет собой последовательность перенумерованных элементов (слов, байтов, битов), номер - адрес элемента. Оперативная память выделяется программе и ее данным только на время исполнения программы.

· ***Внешняя память***- предназначена для долговременного хранения большого объема информации. Информация, хранящаяся во внешней памяти, используется процессором только через оперативную память; для обмена информацией между этими двумя видами памяти имеются специальные команды.

· ***Внешние устройства*** (клавиатура, дисплей, принтер и др.) - служат для взаимодействия компьютера с пользователем и другими устройствами.

Управлением всеми устройствами компьютера занимается операционная система (например, DOS, Windows и т.п.). Как и всякая программная система, она состоит из набора компонент (программ и данных). Основное отличие операционной системы от других программных систем - исполнение ее программ инициируется сигналами (прерываниями), поступающими от устройств компьютера. В свою очередь, программы операционной системы вырабатывают сигналы, заставляющие эти

устройства выполнять "пользовательские" программы в соответствии с определенными правилами, определяемыми в данной операционной системе.

В целом аппаратные устройства и операционные средства создают ту операционную среду, в которой работают системные и пользовательские программы.

**Среда разработки программного обеспечения**- совокупность программных средств, используемая программистами для разработки программного обеспечения. Системы программирования представляют собой единство средств статической (инструментальной) и динамической (исполнительной) поддержки. Простая среда разработки включает в себя:

·редактор текста с подсветкой синтаксиса конкретного языка программирования - в нем программист пишет текст программы, так называемый программный код;

·компилятор и/или интерпретатор - транслирует программу, написанную на высокоуровневом языке программирования в машинный язык (машинный код), непосредственно понятный компьютеру. Язык C++ относится к компилируемым языкам, поэтому для обработки текстов его программ служит компилятор, иногда вместо компилятора (либо вместе с ним) используется интерпретатор, для программ, написанных на интерпретируемых языках программирования;

·отладчик - служит для отладки программ. Ошибки в программах могут быть синтаксическими (обычно они выявляются еще на стадии компиляции) и логическими. Для тестирования программы и выявления в ней логических ошибок служит отладчик;

·средства автоматизации сборки.

Когда эти компоненты собраны в единый программный комплекс, говорят об **интегрированной среде разработки**(Integrated development environment -IDE). Такая среда представлена одной программой, не выходя из которой можно производить весь цикл разработки. В состав комплекса кроме перечисленных выше компонент могут входить средства управления проектами, система управления версиями, разнообразные инструменты для упрощения разработки интерфейса пользователя, стандартные заготовки («мастера»), упрощающие разработку стандартных задач, и др. Современные среды разработки, поддерживающие объектно-ориентированную разработку ПО, также включают браузер классов, инспектор объектов и диаграмму иерархии классов.

Обычно среда разработки предназначается для одного определённого языка программирования, хотя существуют среды разработки,

предназначенные для нескольких языков - такие как Eclipse или Microsoft Visual Studio.

Системы программирования по типу предоставляемого программного интерфейса можно классифицировать на:

·Имеющие интерфейс командной строки (Command Line Interface - CLI). Это традиционный интерфейс систем программирования в операционной системе Unix. В современных диалектах Unix практически все инструменты имеют и надстройку с графическим пользовательским интерфейсом.

·Имеющие графический пользовательский интерфейс (Graphic User Interface - GUI). Этот интерфейс традиционен для систем программирования в Windows.

Если IDE включает в себя возможность визуального редактирования интерфейса программы, она называется **визуальной средой**.

**Системы визуальной разработки приложений**

Системы визуальной разработки приложений объединяют в себе возможности систем программирования и систем разработки интерфейсов.

Системы разработки интерфейсов в начале 90-х годов XX века составляли большую долю в инструментарии. Сейчас такие системы входят составной частью в CASE-средства. Самые известные из них:

C++ Visual Studio (компании Microsoft);

C++ WorkShop Visual (компании Sun Microsystems);

Delphi Suite (компании Borland Inc.);

·Средства построения графического интерфейса в Java (компоненты и контейнеры).

Общая **схема работы** в среде визуального программирования предполагает:

**Выбор типа** разрабатываемого приложения из имеющегося набора прототипов.

**Создание**в визуальной манере **интерфейса**приложения.

**Настройку свойств** интерфейсных элементов.

**Написание кода обработчиков событий** для использованных интерфейсных элементов, который позволил бы объединить их в единую систему. Именно эта фаза разработки приложения является самой ответственной и требует квалификации и наибольших усилий со стороны программиста.

Структуру визуальной среды программирования рассмотрим на примере системы Delphi.

Визуальная среда программирования Delphi- наиболее распространенный инструмент для быстрого создания эффективных Windows-приложений. Она проста в освоении, так как большинство средств программирования в ней визуализированы, а в основе лежит достаточно простой для изучения язык ObjectPascal.

Основным достоинства данной среды программирования является то, что Delphi - это **комбинация нескольких важнейших технологий:**

1. Высокопроизводительный компилятор;

2. Объектно-ориентированная модель компонент;

3. Визуальное построение приложений из программных прототипов;

4. Быстрая разработка работающего приложения из прототипов.

Среда Delphi включает в себя полный набор визуальных инструментов для быстрой разработки приложений (RAD - rapid application development), поддерживающий разработку пользовательского интерфейса и подключение к корпоративным базам данных. К их числу относятся:

·**Визуальный построитель интерфейсов** (Visual User-interface builder) - дает возможность быстро создавать приложения визуально, просто выбирая компоненты из соответствующей палитры.

· **Библиотека визуальных компонентов** (VCL – Visual ComponentLibrary) - эта библиотека объектов включает в себя стандартные объекты построения пользовательского интерфейса, графические объекты, объекты мультимедиа, диалоги, объекты управления файлами и управление DDE.

· Delphi обладает удобным **графическим отладчиком**, позволяющим находить и устранять ошибки в коде. Можно устанавливать точки останова, проверять и изменять переменные, при помощи пошагового выполнения. Если же требуются возможности более тонкой отладки, то можно использовать отдельно доступный Turbo Debugger.

Среда Delphi следует спецификации, называемой Single Document Interface (SDI), и состоит из нескольких отдельно расположенных окон. Основные составные части интерфейса Delphi:

1. Дизайнер Форм (Form Designer);

2. Окно Редактора Исходного Текста (Editor Window);

3. Палитра Компонент (Component Palette);

4. Инспектор Объектов (Object Inspector);

5. Интерактивный Справочник (On-line help).

Дизайнер Форм в Delphi настолько интуитивно понятен и прост в использовании, что создание визуального интерфейса превращается в игру. Дизайнер Форм первоначально состоит из одного пустого окна, которое вы заполняете всевозможными объектами, выбранными на Палитре Компонент. Информация о формах хранится в двух типах файлов - .dfm и .pas, причем первый тип файла (двоичный) хранит образ формы и ее свойства, второй тип описывает функционирование обработчиков событий и поведение компонент. Оба файла автоматически синхронизируются Delphi, так что если добавить новую форму в проект, связанный с ним файл pas автоматически будет создан, и его имя будет добавлено в проект.

В дополнение к инструментам, обсуждавшимся выше, существует набор **инструментальных средств**, поставляемых вместе с Delphi:

· Встроенный отладчик;

· Внешний отладчик (поставляется отдельно);

· Компилятор командной строки;

·ReportSmith - генератор отчетов для баз данных;

· Team Development Support: предоставляет контроль версий при помощи PVCS компании Intersolve (приобретается отдельно) или при помощи других программных продуктов контроля версий;

·Visual Query Builder - средство визуального построения SQL-запросов;

· и ряд других продуктов.

Иногда достаточно использовать только одну интегрированную среду разработки, но для больших проектов в среду разработки включаются разнородные продукты разных фирм, разных версий. Пример такого набора: файловый менеджер, набор вспомогательных утилит и пакетных файлов, C++Builder – как IDE, PLSQLDeveloper – для работы с СУБД Oracle, CristalReports – для создания отчетов , StarTeam – для ведения версий и поддержки коллективной работы.

**Выбор среды разработки**

Технология программирования во многом определяется языком программирования, на котором пишутся программы. В языке могут быть заложены средства, влияющие на технологичность и архитектуру разрабатываемой системы (например, объектно-ориентированность, модульность и т.п.). Обычно выбирают ту модель разработки и те языки программирования, которые хорошо знают члены коллектива разработчиков. Выбирать новую технологию, которую предстоит осваивать в процессе разработки – риск провалить проект.

У каждого программиста есть свой взгляд на модель разработки, определяющийся его прошлым опытом, степенью освоения тех или иных инструментальных средств.

Любая среда позволяет производить настройку и адаптацию под те или иные требования: изменение интерфейса, режимов работы, назначения горячих клавиш, установка дополнительных средств («плагинов») и т.п. В арсенале каждого опытного программиста есть свои приемы разработки, собственные вспомогательные средства. Он имеет собственные вкусы и предпочтения. Используя настройки, программист может сделать работу в среде более удобной для себя и, тем самым, более эффективной. Он как бы проецирует свою модель разработки на модель среды. Это особенно важно, когда среди инструментов есть программы с отличающимся интерфейсом (например, разное назначение горячих клавиш).

Создатели инструментальных средств закладывают, как правило, избыточный набор возможностей и программисты практически никогда не используют инструментальное средство на все 100%.