Министерство образования и науки РФ
АМУРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
(ГОУВПО «АмГУ»)

Учебно-методический комплекс дисциплины

КОМПЬЮТЕРНЫЕ ТЕХНОЛОГИИ В ФУНДАМЕНТАЛЬНЫХ ИССЛЕДОВАНИЯХ

по направлению подготовки
010600.68 – «Прикладные математика и физика»

Утвержден на заседании кафедры теоретической и экспериментальной физики инженерно-физического факультета
«__» ___________ 20__г.,
(протокол № __ от «__»_____20__г.)
Зав. кафедрой __________Е.А. Ванина

Печатается по решению
редакционно-издательского совета
инженерно-физического факультета
Амурского государственного
университета

Красников И.В.

Компьютерные технологии в фундаментальных исследованиях. Учебно-методический комплекс дисциплины по направлению подготовки 010600.68 – «Прикладные математика и физика» – Благовещенск: Амурский гос. ун-т, 2011 – 84 с.

© Амурский государственный университет, 2011

1. ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

1.1. ЦЕЛИ ПРЕПОДАВАНИЯ ДИСЦИПЛИНЫ.

Основной целью дисциплины является формирование у будущих специалистов практических навыков по алгоритмизации вычислительных процессов для решения экономических и расчетных задач с применением современных методов и технологий программирования, обучение работе с научно-технической литературой и технической документацией по программному обеспечению ПЭВМ.

1.2. ЗАДАЧИ ИЗУЧЕНИЯ ДИСЦИПЛИНЫ

Задачей изучения дисциплины является реализация требований, установленных в квалификационной характеристике, в подготовке в области использования вычислительной техники и ее программного обеспечения в системах машинной обработки информации, проектирования и разработки этих систем.

Студент должен знать и уметь использовать:

* основы теории алгоритмов и ее применения, методы построения формальных языков, основные структуры данных, основы машинной графики, архитектурные особенности современных ЭВМ;
* синтаксис, семантику и формальные способы описания языков программирования, конструкции распределенного и параллельного программирования, методы и основные этапы трансляции; способы и механизмы управления данными;
* принципы организации, состав и схемы работы операционных систем, принципы управления ресурсами, методы организации файловых систем, принципы построения сетевого взаимодействия, основные методы разработки программного обеспечения;
* основные модели данных и их организацию, принципы построения языков запросов и манипулирования данными, методы построения баз знаний и принципы построения экспертных систем;

2.СОДЕРЖАНИЕ ДИСЦИПЛИНЫ

2.2. СОДЕРЖАНИЕ ЛЕКЦИЙ

9 семестр (54 часа)

2.2.1. Понятие информационных технологий (4 часа)

Информация. Информационная технология. Цель информационной технологии. Современная информационная технология. Телекоммуникации. Три основных принципа новой информационной технологии: интерактивный (диалоговый) режим работы с компьютером; интегрированность с другими программными продуктами; гибкость процесса изменения как данных, так и постановок задач. Информационные ресурсы и продукты. Рынок информационных продуктов и услуг. Понятие информационного общества. Информационная культура. Классификация информационных технологий по типу обрабатываемой информации, по типу пользовательского интерфейса, по степени взаимодействия. Командный интерфейс, WIMP-интерфейс, SILK-интерфейс, общественный интерфейс. Классификация ИТ по степени их взаимодействия.

2.2.2. Наиболее распространенные информационные технологии (4 часа)

Технология обработки текстовых, графических и табличных данных. Гипертекстовая технология. Технология мультимедиа. Технология автоматизации офиса. Интегрированные пакеты для офиса. Технология обмена данными в Microsoft Office. Буфер обмена (Clipboard). Обмен данными в сети (Microsoft Exchange). Динамический обмен данными (DDE). Связывание и внедрение объектов (OLE). Работа с OLE. Основные особенности современных проектов программного обеспечения (ПО), характеристики различных классов проектов. Проблема сложности больших систем. Место и роль CASE-технологии в жизненном цикле ПО. Жизненный цикл ПО. Понятие жизненного цикла (ЖЦ) ПО. Международные и отечественные стандарты, регламентирующие ЖЦ ПО. Стандарт ISO/IEC 12207 (Information Technology - Software Life Cycle Processes) и его практическое применение. Процессы ЖЦ ПО: основные, вспомогательные и организационные. Взаимосвязь между процессами ЖЦ ПО. Применение CASE-технологии в процессах ЖЦ ПО. Модели и стадии ЖЦ ПО. Взаимосвязь между процессами и стадиями. Каскадная и спиральная модели ЖЦ ПО, их сопоставление. Подход быстрого проектирования приложений (RAD).

2.2.3. Объектно-ориентированный подход к программированию (4 часа)

Анализ и проектирование ПО на основе объектно-ориентированного подхода. Сущность объектно-ориентированного подхода. Унифицированный язык моделирования UML. Основные средства языка. Описание требований к системе. Варианты использования (use case). Моделирование статической структуры системы. Диаграммы классов. Механизм пакетов. Моделирование

поведения системы. Диаграммы взаимодействия (диаграммы последовательности и кооперативные диаграммы). Диаграммы состояний. Диаграммы деятельности. Моделирование реализации системы. Диаграммы компонентов. Диаграммы размещения. Генерация кода программ и описаний баз данных. Реверсный инжиниринг. Пример использования объектно-ориентированного подхода. Анализ и проектирование ПО на основе структурного подхода. Сущность структурного подхода к разработке ПО. Сопоставление и взаимосвязь структурного и объектно-ориентированного подходов.

2.2.4. Технология OLE (6 часов)

Технология OLE (Objects Linked and Embedded - связывание и внедрение объектов) как технология интеграции программных продуктов, входящих в комплект Microsoft Office. Технология динамического обмена данными DDE (Dynamic Data Exchange) как предшественница OLE. История развития технологии OLE: OLE 1, OLE 2, OLE Automation. OLE-объекты и OLE-контейнеры. Понятие составного документа (например, документа Word). Механизмы OLE (OLE Automation, OLE Documents, OLE Controls,Объект). TOLEContainer. Приложение OLE. Техника инициализации OLE-объекта: либо в стандартном диалоге Windows “Insert Object”, либо с помощью Clipboard, либо с помощью техники “перенести и бросить” (drag-and-drop). Сохранение OLE - объекта в базе данных.

2.2.5. Технология COM (6 часов)

Технология COM (Component Object Model - модель многокомпонентных объектов) как основа OLE 2. Преимущества технологии COM. COM как конкретная реализация набора интерфейсов. Идентификация интерфейса. Спецификация интерфейса. Реализация интерфейса. Фундаментальный интерфейс IUnknown. Методы интерфейса объекта COM. Подсчет ссылок. COM - объект как экземпляр определенного класса. Библиотеки COM. COM и объектно-ориентированный подход. Инкапсуляция, полиморфизм и наследование объектов COM. COM и многокомпонентные программы. Хранилища (storage) и потоки (streams). Моникер (moniker, имя, кличка*)* как объект COM. Стандартный способ обмена информацией в мире COM — единообразная передача данных (Uniform Data Transfer). Серверы объектов COM. COM и многопоточность. Распределенная COM. Создание, инициализация и повторное применение объектов COM. Агрегирование. Распределенная COM (DCOM - Distributed COM). Поддержка удаленных объектов при помощи DCOM. Создание удаленного объекта. Доступ к удаленному объекту. Перспективы технологии COM.

2.2.6. Технология ActiveX (4 часа)

ActiveX как технология Microsoft, предназначенная для написания сетевых приложений. Клиентская и серверная части ActiveX. Клиентская технология ActiveX (Active Desktop). Программные компоненты ActiveX. Языки сценариев ActiveX (Jscript, Visual Basic или Visual Basic Scripting

Edition). Управляющие элементы ActiveX. Серверная технология ActiveX (Active Server). Выполнение на сервере языков сценариев (скриптов). Преимущества и перспективы технологии ActiveX.

2.2.7.Технология CORBA (6 часов)

Обобщенная Архитектура построения Брокеров Объектных Запросов (CORBA) как средство поддержки интеграции самых разнообразных объектных систем. Принципы создания Брокеров Объектных Запросов (ORB). Различные реализации ORB. Реализация и адаптеры объектов. Динамическая обработка запросов. Параметры и интерфейсы ORB-а. Интерфейс динамического выполнения вызовов. Обзор протокола GIOP. Транспорт для сообщений протокола IIOP- протокола обмена между Брокерами Объектных в Internet (Internet Inter-Orb Protocol - IIOP). Транспорт для протокола GIOP. Управление соединением. Язык описания интерфейсов (IDL). Базовые типы данных. Синтаксис Общего Представления Данных – CDR. Кодирование базовых типов, составных типов, инкапсуляции, псевдообъектов. Хранилище описаний. Сравнительный анализ технологий CORBA и COM.

2.2.8. Обзор современных Web-технологий (4 часа).

Средства публикации данных на web-сервере. Языки описания документов: HTML, XML, Dynamic HTML. Язык гипертекстовой развертки HTML, его возможности, достоинства и недостатки. Создание таблиц, списков. Стилевое оформление документов. Фреймы, таблицы, формы языка HTML. Каскадные таблицы стилей CSS. Язык XML.. Перспективы развития Web-технологий.

2.2.9. Языки web-программирования (2 часа).

Языки программирования на стороне клиента: JavaScript, VBScript, Java. Языки программирования серверов: технологии CGI, SSI и ISAPI, языки Perl, PHP, Python, ASP и ASP.Net.

2.2.10. Язык программирования PHP (6 часов).

Основные операторы языка PHP. Арифметические и логические функции языка. Функции и процедуры для работы с базами данных.

2.2.11. Разработка и создание базы данных MySQL на сервере Apache (6 часов).

Установка сервера Apache. Настройка и конфигурирование сервера. Файл httpd.conf. Установка базы данных MySQL. Настройка базы данных. Конфигурирование MySQL. Установка языка PHP. Конфигурирование связки PHP-MySQL-Apache. Создание таблиц в базе данных MySQL. Изменение и дополнение таблиц MySQL посредством языка PHP. Работа с базами данных через web-интерфейс.

3. ТЕМАТИКА ПРАКТИЧЕСКИХ ЗАНЯТИЙ

В данном курсе практические занятия не предусмотрены.

4. САМОСТОЯТЕЛЬНАЯ РАБОТА СТУДЕНТОВ (46 часов)

Создание сайта электронной коммерции (Интернет-магазин, туристическая фирма и т.п.) в рамках выполнения курсовой работы (30 часов). 16 часов самостоятельной работы студентов отводится на углубленное изучение языка написания web-страниц – HTML.

5. ВОПРОСЫ К ЗАЧЕТУ

1. Понятие информации и информационной технологии.
2. Информационные ресурсы и продукты.
3. Классификация информационных технологий по типу обрабатываемой информации.
4. Классификация информационных технологий по типу пользовательского интерфейса.
5. Классификация информационных технологий по степени взаимодействия.
6. Гипертекстовая технология.
7. Технология обработки текстовых, графических и табличных данных.
8. Технология обмена данными в Microsoft Office.
9. Динамический обмен данными (DDE).
10. Case-технология.
11. Технология OLE (связывания и внедрения объектов).
12. Сущность объектно-ориентированного подхода к программированию.
13. Унифицированный язык моделирования UML.
14. Моделирование статической структуры системы. Диаграммы классов. 15. Механизм пакетов. Моделирование поведения системы. Диаграммы взаимодействия (диаграммы последовательности и кооперативные диаграммы). Диаграммы состояний. Диаграммы деятельности.
16. Моделирование реализации системы. Диаграммы компонентов. Диаграммы размещения. Генерация кода программ и описаний баз данных.
17. Сущность структурного подхода к разработке ПО.
18. Общая характеристика и классификация CASE-средств. Состояние Российского рынка CASE-средств. Определение потребности в CASE-средствах. Анализ рынка CASE-средств.
19. Сравнительный анализ современных технологий проектирования.
20. Вспомогательные методы и средства, используемые в жизненном цикле ПО. Управление требованиями к системе. Оценка затрат на проектирование ПО (метод функциональных точек). Управление конфигурацией ПО. Документирование ПО. Тестирование ПО. Управление проектом ПО.

Требования к знаниям студентов, предъявляемые на зачете

На зачете студенту предлагается ответить на один вопрос из предлагаемого выше списка и ответить на дополнительные вопросы по теме.

Знания студента оцениваются на «зачтено» при полном ответе на вопрос и удовлетворительном ответе на дополнительные вопросы преподавателя.

Оценка «не зачтено» ставится при незнании вопроса, предлагаемого студенту на зачетном занятии.

ЛИТЕРАТУРА

Основная литература

1. Советов Б.Я. Информационные технологии: учеб.: доп. Мин.обр. РФ/ Б.Я. Советов, В.В. Цехановский. -4 –е изд.,стер.. -М.:Высш.шк.,2008.-264с.
2. Титоренко Г.А. Информационные системы и технологии управления: учеб.: рек. Мин. обр. РФ/ -3-е изд., перераб. и доп.. -М.:ЮНИТИ-ДАНА, 2010 -592 с.: а-рис.
3. Бахвалов Н.С. Численные методы: решение задач и упражнения: учеб. пособие: рек. УМО/ Н.С.Бахвалов, А.А.Корнев, Е.В.Чижонков. - М.: Дрофа, 2009.-396 с.

Дополнительная литература

1. Макарова Н.В. Информатика.: учебник, 3-е перераб. изд.- М.: Финансы и статистика, 2005.
2. Михеева Е.В. Практикум по информационным технологиям в профессиональной деятельности. «Академия», 2005.
3. Самарский А.А., Гулин А.В. Численные методы, - М.: Наука,1989.
4. Симонович С.В. Информатика. Базовый курс. - СПб: «Питер», 2000.
5. Сухомлин В.А. Введение в анализ информационных технологий. «Горячая линия- Телеком, Радио и связь»,2003.
6. Угринович Н.Д., Босова Л.Л. Практикум по информатике и информационным технологиям. «Лаборатория базовых знаний», 2002.
7. Смит Д.М. Математическое и цифровое моделирование для инженеров и исследователей/ пер.с анг. М: Машиностроение,1980.

Лекции

Тема 1. Информационные технологии.

Информация – сведения об объектах и явлениях окружающей среды, которые подлежат сбору, хранению, обработке и передаче.

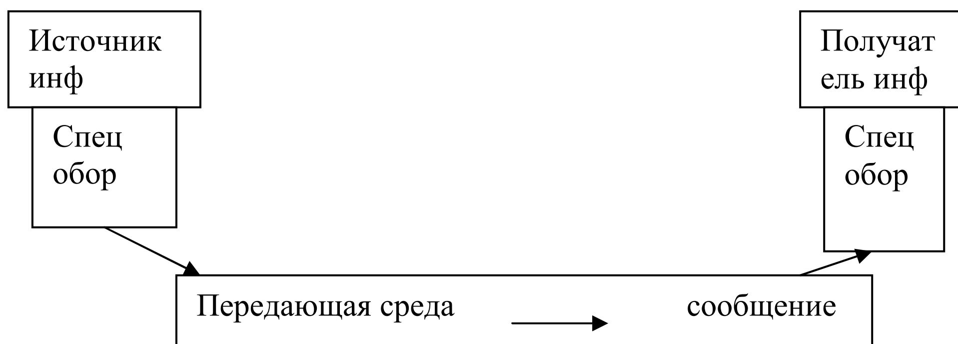
Инф. Процессы:

- сбор информации
- хранение информации. 2 носителя (бумажный и компьютерный как носитель инф, компьютерный диск позволяет удерживать больший объем инф на незначительном физическом пространстве).
- Обработка информации – основным средством обработки инф явл ПК, оснащенный пакетом прикладных программ (ППП). Обработка информации осуществляется, осуществляющаяся при помощи ПК наз автоматизированной.
- Передача инф- осуществляется в телекоммуникац системах.

Телекоммуникационная система – совокупность компонентов, использующиеся для передачи инф.

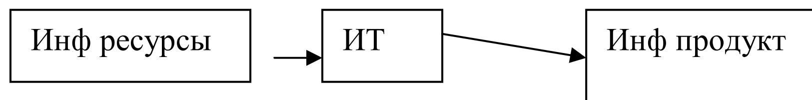
- источник инф –объект, генерирующий инф
- получатель инф –объект, потребляющий инф
- средства передачи-физически передающая среда + специализированный аппаратура для приема-отправки инф.
- Сообщение –передаваемая инф

Схема: компоненты телекомм системы



Инф технологии – процесс, использующий совокупность технич средств и методов осуществления инф процессов с целью получения инф продукта на базе имеющихся инф ресурсов.

Схема ИТ



Назначение ИТ- производство инф продуктов для их дальнейшего анализа и принятия управленческих решений.

В основе современных (новых, компьют) ИТ лежит использование ПК, комп дисков и современной телекоммуникаций, в первую очередь комп сетей. В большинстве случаев под соврем ИТ понимают пользовательную комп программу.

Современные ИТ строятся на 3 принципах

- принцип интерактивности, т.е. диалоговый режим работы с программой , удобный пользовательский интерфейс. Большинство совр ИТ использует WIMP – интерфейс (Windows Imidge Menu Pointer)
- принцип интегрированности, т.е. взаимосвязь между различными ИТ.
- Гибкость процесса изменения данных и постановок задач.

Инф ресурсы – информация, зафиксированная на каком-либо носителе и предназначается для широко соц использования.

Инф продукт –информация, сформированная ее производителями для дальнейшего распределения и исп-я.

В качестве инф рес-сов (продуктов) может выступать разл документы, архивы, комп программы, алгоритмы решения задач, патенты, лицензии, инженерно-технич изобретения, теле и радио реклама и тд.

В качестве производителя инф рес-са (продукта) могут выступать рядовые пользьватели, коммерческие орган-ции, гос центры по сбору и хранению инф, банки, биржи.

Инф услуга может пониматься в следующих аспектах: телеком услуга, предоставление доступа к инф, оказание инф поддержки, услуги образования.

<u>История развития ИТ</u>

Врем период	Этап ИТ	Инструментарий ИТ	Назначение ИТ
До сер 19в	Ручные Ит	Бумага, чернила, перо, счеты. Осн носитель инф –книга, телекоммуникация - почта	Зафиксировать инф на бум носителе
Сер 19в-нач20в	механические	Пищущая машина. Диктофон, механ вычислит устройства, носитель – книга, теле фото пленка, ср-ва для хранения аудио инф. Телекомм –почта, телефон, телеграфная связь	Зафиксировать инф более быстрым и удобным способом
30-70г 20в	электрич	Первые ЭВМ, электрич пишущая машина, ксероксы, диктофоны, носитель –книга, аудио видео фото пленка, Телеком – те же	Сформировать и зафиксировать инф продукт
С 80г 20в	Соврем (комп, новая)ИТ	ПК, внешнее устройство-оргтехника. Телекомм – разл виды телеф связи, факсим комп сети. Носители – комп диски	ИТ – часть системы инф поддержки принятия управленч решений пользователя.

ИТ первых двух этапов относят к традиционным ИТ.

<u>Классификация ИТ</u>

1 По типу обрабатываемой инф

- ИТ обработки данных (электрон таблицы, система управления БД)
- ИТ обработки текста (текст редактора, гипертекст технологии

Рассматривается ЭВМ как единая строка символов, читаемая в одном направлении. Гипертекст представляет собой сетевую модель взаимосвязи инф статей. Инф статья содержит ссылки на родственные статья, позволяющие осуществлять быстрый переход в них. Тезаурус гипертекста – автоматич словарь по поиску инф по ключевому слову. Гипертекст широко применяется в справочных программах и в поисковых системах глобальных сетей.

- графические редакторы
- экспертные системы, т.е. ИТ способные оказать проф экспертную поддержку пользователю
- технологии мультимедиа – интерактивные технологии, обеспечивающие работу с видео изображениями, звук и анимации.

В настоящее время ИТ различных классов объединяются в интегрированные пакеты, которые обладают следующими особенностями:

- общий интерфейс программ

- возможность обработки общих данных всеми программами пакета (интегрировать)
- одна программа может использовать функции других программ.

2. По обслуживаемой предметной области

- ИТ бух учета
- БанковскиеИТ
- Ит налоговой службы
- Ит страхования и тд.

Электронный офис

К офисным задачам традиционно относят делопроизводство, управление, контроль, формирование отчетности, поиск инф по запросам пользователя, обмен инф между разл офисами, взаимодействие с внешней средой и тд.

Для автоматич поддержки деятельности офиса организуют специализированный програмно-аппаратный комплекс – **эл офис**.

Схема: компоненты эл офиса

Эта инф обрабатывается техническим обеспечением офиса. В результате обработки получаем инф продукт.

Осн компонентами деятельности эл офиса явл:

- сбор и регистрация входящей инф офиса. Данная инф сохраняется в БД офиса. В крупных орг-циях, обрабатывающие большие объемы инф, м.б. сформированы банки данных (хранилище данных).
- Обработка входящих данных, их анализ, формирование инф продуктов. Входящая необработанная инф д.б. сохранена и передана по назначению.
- Передача сформированных инф продуктов внешн и внутр пользователям.

В состав эл офиса входят

- Технич обеспечение
- Програмное обеспечение

К программному обеспеч традиционно относят: технич ресурсы, СУБД, программы составления расписания, прогр по делопроизводству, прогр обслуживание факс модема, эл почта и тд.

К технич обесп офиса относят: ЭВМ в разл классах, внешние устройства, орг техника.

Совр офис широко использует возможности локальных комп сетей, которые позволяют перейти на полный электронный документооборот, т.е. инф получается, хранится и передается в виде эл документов (комп файлов).

Соврем российское законодательство придает эл документу юрид силу, равную бумажному документу. Юридическая сила эл документа устанавливается с помощью

кода, формы документа, № документа, дата создания документа и электронно-цифровая подпись.

Варианты внедрения ИТ в организацию

При внедрении комп программы орг-ция может придерживаться 1 из 2 осн концепций, которые различаются с т. зрения структуры орг-ции и роли в них инф технологий.

1 концепция внедрения

Существующая структура организации не изменяется. Программа приобретается с учетом необходимости перестраиваться под нее. Роль инф техн – это автоматизация 1 или нескольких раб мест.

+ незначит финансовые и временные затраты
  увеличение качества обработки инф на данном рабочем месте
  снижение затрат на обработку инф

- изменение программы может существенно снизить ее эффективность
  недостаточный уровень технич средств
  отсутствие комп сетей не позволяет пользователю выбрать наиболее эффективную и новую программу
  не решается проблема интегрированности ИТ.

2 концепция

Существующая структура организации значительно модернизируется , ориентируясь на максимальное эффективное использование ИТ. Роль ИТ –это создание единой системы инф поддержки, принятие управл решений всеми спец организациями.

При данном варианте изменяется внутр структура орг-ции, создаются новые орг взаимосвязи, обновляется техн обеспечение, создаются банковские хранилища данных. Таким путем внедряются ИТ, предстовляющие собой сложные програм комплексы. Каждая программа комплекса устанавливается на одном из рабочих мест и может работать как автономно, так и совместно с другими прогр-ми. В большинстве случаев такие ИТ явл индивид разработками, создающимися под конкретную организацию.

+ возможность перехода на полный эл документооборот в организации
  рационализация структуры организации, сокращение численности управленческого персонала
  возможность совместного решения задач всеми специализированными программами.

- существенные затраты связанные с разработкой и внедрением ИТ. Ит может внедряться последовательно, либо по всей орг-ции в целом.. Последовательное внедрение может происходить либо по отделам орг-ции, либо по решаемым задачам.
  Высока степень риска. Риск возрастает если орг-ция внедряет собств ИТ. При данном варианте рекомендуется заключать договор с крупной фирмой, широко известной на рынке.
  Психологическая напряженность в коллективе.

Методология использования ИТ

Выделяют 3 осн методологии:

1) **Централизованная обработка инф.** Представляет собой использование ИТ в следующих

- на рядовых рабочих местах регистрируется и сохраняются эл. Документы
- вся собранная инф передается в общий вычислительный цент оргции, в котором реализуется необход технич оборудование и осн модули инф технологии, производящие обработку инф
- вычислительный центр обрабатывает инф и формирует инф продукт

- инф продукты передаются их пользователям.

Положит :

Простота внедрения и использования сложн инф технологий

Решение проблемы интегрированности

Отрицат

Высокая стоимость содержания вычислительного центра

Разрыв во времени сбора и обработки инф

Данная методология широко исп-ся в банковских системах.

2) **Децентрализованная обработка ин**ф. При данной методологии каждое рабочее место оснащается персональным комп и необходимой ИТ. Каждый пользователь самостоятельно формирует инф продукты. Данная методология явл наиболее распространенной в современных офисах.

\+ отсутствие разрыва между сбором и обработкой инф

миним затраты на внедрение и использование ИТ.

\- сложность интегрированности ИТ

зависимость качества инф продуктов от подготовленности пользователя

3) Распределенная обработка инф. Включает в себя черты 2х выше перечисленных методологий.

1 Существует вычислительный центр, разрабатывающий общую стратегию внедрения ИТ, устанавливающий стандарты на технич ср-ва, приобретающий (разрабатывающий) Ит, обучающий пользователей, а также создающий единое правило работы в программе.

2 Стандартные технич ср-ва и ИТ устанавливаются на каждом рабочем месте. Пользователи работают в программе по строгим инструкциям.

Данная методология широко применяется в гос органах и крупных регионных и межрегионных предприятиях.

Тенденции развития ИТ

Выделяют 5 осн тенденций

1 **Усложнение инф продуктов и возрастание их роли**. В современной экономике финан состояние орг-ции во многом зависит от принимаемых в ней управленческих решений. Основой для принятия управляемого решения явл инф продукты орг-ции (отчеты. Рез-та исследований, планы развития, рез-ты анализа и тд.). Для формирования качественного инф продукта необх использовать наиболее современные ИТ. Совр программы оснащаются мощным математ аппаратом средствами анализа, планирования, оптимизации, а также ср-ми выхода в глобальные сети.

2 **Обеспечение совместимости (интегрированности) ИТ**. В последнее время большой популярностью пользуются интегрированные пакеты пользовательских программ.

3 **Ликвидация промежуточных звеньев**. Соврем ИТ, автоматизируя задачи управления, устраняет тем самым управленцев среднего и низшего звена, а также сокращение управл штатов.

4 **Глобализация** – это общая тенденция мировой экономики. Проявляется как развитие глобальных сетей, глобальных БД, как повсеместное использование единых программ

5 **Конвергенция** т.е. стирание различий между сферами материального и инф пространства, а также увеличение информационной составляющей в ?????? материальном продукте. Продукции развитых стран на долю материальных составляющих приходится 10-20% стоимости товаров.

1.

Информатизация общества – организационный социально-экономический и научно-технический процесс создания оптимальных условий для удовлетворения информацией потребителей и реализации информационных прав граждан, органов гос. власти, органов местного самоуправления, общественных объединений и использования информационных ресурсов и информационных технологий.
Процесс информатизации начался в 60-х г.г. 20 в. в США и Японии. В конце 70-х гг. в Зап. Европе и в кон. 80-х в России.
Характерные черты информатизации.

1) появление информационных кризисов.

   Информационный кризис проявляется как лавинообразное возрастание объемов информации, которое сопровождается недостаточным уровнем развития технических средств, ИТ и человеческих возможностей.
2) повсеместное внедрение технических средств, ИТ и телекоммуникаций в различных сферах человеческой деятельности.
3) Появление а активное развитие информационной индустрии.

   Информационная индустрия – это отрасль, связанная с производством технических средств, ИТ и телекоммуникаций.
   В развитых странах в информационной индустрии занято до 70 % трудоспособного населения, в России – 40 %.
4) развитая структура информационного рынка.

   В развитых странах процесс информатизации в последнее время замедлился, что говорит о полном удовлетворении существующих информационных потребностей. Данные страны во многом проявляют себя как живущие в стадии инф. общества.

2.

Информационная революция – это преобразование общественных отношений вследствие кардинальных изменений в сфере информационных процессов.
Следствием ИР является приобретение обществом новых качеств.

ИР

Временной период	Сущность революции	Изменения в информационных процессах	Изменения в обществе
	Изобретение письменности	1.возможность фиксирования информации на материале-носителе. 2.основной носитель информации - книга. 3.появление 1-х коммуникаций - почта. 4.средства для обработки инф - счета	Зарождение человеческих цивилизаций
Сер.16 в.	Появление книгопечатания	1.удешевление информации 2.массовость информации. 3.увеличение скорости производства информации	Начало индустриализации общества
Кон.19 в	электричество	1.появление новых телекоммуникаций: телеграф, телефон, радио. 2.появление новых носителей информации (фото, кинопленки)	Окончательный переход к индустриальному обществу
70-е гг. 20 в.	Появление ПК	1.ПК – основное устройство для обработки информации. 2.основной носитель инф. – компактный диск.	

		3.появление новых коммуникаций – компьютерных сетей.	

Информационное общество.

При изучении общества исследователи придерживаются различных подходов. Существует информационный подход, согласно которому движущей силой развития общества является информация, обеспечение информационных процессов и удовлетворение информационных потребностей. Кардинальные изменения в жизни общества происходят вследствие информационных революций.
С точки зрения информационного подхода выделяют 3 стадии в развитии общества:
1. Феодальное общество. Движущей силой развития общества является земля и ее с\х использование. Для данного общества характерна слабая степень информатизации; влияние информации на экономику и прочие виды деятельности минимально.
2. Индустриальное общество. Основной движущей силой общества является энергия и ее применение в промышленной деятельности. В данном обществе высока степень информатизации, влияние ИТ и промышленности и экономике достаточно высоко. В настоящее время информация рассматривается как стратегический ресурс и приравнивается к материальным ресурсам. Активно проявляется тенденция конвергенции в современной экономике. На данной стадии произошло 2 информационной революции: в конце 19 века – появление электричества и в 70-е 20 века.
После 4 революции процесс информатизации значительно усиливается, и в обществе начинают проявляться многие черты информационного общества.
3. Информационное общество. Основной движущей силой является информация и ИТ.
Характерные черты информационного общества.
- высокий уровень автоматизации всех сфер человеческой деятельности, глобальный характер ИТ.
- решение проблем информационного кризиса.
- приоритет информационных ресурсов и информационной индустрии над прочими отраслями
- формирование единого информационного пространства
- свободный доступ любым пользователям ко всем информационным ресурсам цивилизации
Кроме положительных моментов прогнозируются и опасные тенденции информационного общества.

- Большое влияние на информационное общество средств массовой информации
- Проблема отбора качественной и достоверной информации
- Возможность разрыва общества на информационную элиту и рядовых пользователей
- Проблема виртуальных людей.

Информационная культура.

Современный период является переходным к информационному обществу; от каждого пользователя требуется овладение определенными элементами информационной культуры.
Информационная культура – умение целенаправленно работать с информацией, умение использовать ИТ и технические средства для принятие управленческих решений. Информационная культура проявляется в следующих аспектах:
- умение использовать современные технические средства и телекоммуникации
- способность использовать ИТ в профессиональной и соц жизнедеятельности
- умение извлекать информацию из различных источников и с помощью различных технических средств.
- знание особенностей информационных потоков в своей области проф деятельности.

Информационная культура основывается на знаниях теории информатики, прикладной информатики, математики, математическом моделировании, теории проектирования информационных систем и др.

Классификация информационных систем по степени автоматизации
Ручные информационные системы характеризуются отсутствием современных технических средств переработки информации и выполнением всех операций человеком. Например, о деятельности менеджера в фирме, где отсутствуют компьютеры, можно говорить, что он работает с ручной ИС.
Автоматизированные информационные системы (АИС) — наиболее популярный класс ИС. Предполагают участие в процессе накопления, обработки информации баз данных, программного обеспечения, людей и технических средств.
Автоматические информационные системы выполняют все операции по переработке информации без участия человека, различные роботы. Примером автоматических информационных систем являются некоторые поисковые машины Интернет, например Google, где сбор информации о сайтах осуществляется автоматически поисковым роботом и человеческий фактор не влияет на ранжирование результатов поиска.

Обычно термином ИС в наше время называют автоматизированные информационные системы.

[править]
Классификация информационных систем по характеру использования информации
Информационно-поисковые системы — система для накопления, обработки, поиска и выдачи интересующей пользователя информации.
Информационно-аналитические системы — класс информационных систем, предназначенных для аналитической обработки данных с использованием баз знаний и экспертных систем.
Информационно-решающие системы — системы, осуществляющие накопления, обработки и переработку информации с использованием прикладного программного обеспечения.
управляющие информационные системы с использованием баз данных и прикладных пакетов программ.
советующие экспертные информационные системы, использующие прикладные базы знаний,
Ситуационные центры (информационно-аналитические комплексы)

[править]
Классификация информационных систем по архитектуре
Локальные ИС (работающие на одном электронном устройстве, не взаимодействующем с сервером или другими устройствами)
Клиент-серверные ИС (работающие в локальной или глобальной сети с единым сервером)
Распределенные ИС (децентрализованные системы в гетерогенной многосерверной сети)

[править]
Классификация информационных систем по сфере применения
Информационные системы организационного управления — обеспечение автоматизации функций управленческого персонала.
Информационные системы управления техническими процессами — обеспечение управления механизмами, технологическими режимами на автоматизированном производстве.

Автоматизированные системы научных исследований — программно-аппаратные комплексы, предназначенные для научных исследований и испытаний.
Информационные системы автоматизированного проектирования — программно-технические системы, предназначенные для выполнения проектных работ с применением математических методов.
Автоматизированные обучающие системы — комплексы программно-технических, учебно-методической литературы и электронные учебники, обеспечивающих учебную деятельность.
Интегрированные информационные системы - обеспечение автоматизации большинства функций предприятия.
Экономическая информационная система - обеспечение автоматизации сбора, хранения, обработки и выдачи необходимой информации, предназначенной для выполнения функций управления.

[править]
Классификация информационных систем по признаку структурированности решаемых задач
Модельные информационные системы позволяют установить диалог с моделью в процессе ее исследования (предоставляя при этом недостающую для принятия решения информацию), а также обеспечивает широкий спектр математических, статистических, финансовых и других моделей, использование которых облегчает выработку стратегии и объективную оценку альтернатив решения. Пользователь может получить недостающую ему для принятия решения информацию путем.
Использование экспертных информационных систем связано с обработкой знаний для выработки и оценки возможных альтернатив принятия решения пользователем. Реализуется на двух уровнях:

Первый уровень (концепция "типового набора альтернатив") - сведение проблемных ситуаций к некоторым однородным классам решений. Экспертная поддержка на этом уровне реализуется созданием информационного фонда хранения и анализа типовых альтернатив. Второй уровень - генерация альтернативы на основе правил преобразования и процедур оценки синтезированных альтернатив, используя базу имеющихся в информационном фонде данных.

Экспертные системы представляют совокупность фактов, сведений и данных с системой правил логического вывода информации на основании логической модели баз данных и баз знаний. Базы данных содержат совокупность конкретных данных, а базы знаний - совокупность конкретных и обобщенных сведений в рамках логической модели базы знаний.

Введение
Основные особенности и проблемы современных программных проектов

Накопленный к настоящему времени опыт создания систем ПО показывает, что это сложная и трудоемкая работа, требующая высокой квалификации участвующих в ней специалистов. Однако до настоящего времени создание таких систем нередко выполняется на интуитивном уровне с применением неформализованных методов, основанных на искусстве, практическом опыте, экспертных оценках и дорогостоящих экспериментальных проверках качества функционирования ПО. По данным Института программной инженерии (Software Engineering Institute, SEI) в последние годы до 80% всего эксплуатируемого ПО разрабатывалось вообще без использования какой-либо

дисциплины проектирования, методом "code and fix" (кодирования и исправления ошибок).

Проблемы создания ПО следуют из его свойств. Еще в 1975 г. Фредерик Брукс, проанализировав свой уникальный по тем временам опыт руководства крупнейшим проектом разработки операционной системы OS/360, определил перечень неотъемлемых свойств ПО: сложность, согласованность, изменяемость и незримость [1]. Что же касается современных крупномасштабных проектов ПО, то они характеризуются, как правило, следующими особенностями:

Характеристики объекта внедрения:

- структурная сложность (многоуровневая иерархическая структура организации) и территориальная распределенность;
- функциональная сложность (многоуровневая иерархия и большое количество функций, выполняемых организацией; сложные взаимосвязи между ними);
- информационная сложность (большое количество источников и потребителей информации (министерства и ведомства, местные органы власти, организации-партнеры), разнообразные формы и форматы представления информации, сложная информационная модель объекта - большое количество информационных сущностей и сложные взаимосвязи между ними), сложная технология прохождения документов;
- сложная динамика поведения, обусловленная высокой изменчивостью внешней среды (изменения в законодательных и нормативных актах, нестабильность экономики и политики) и внутренней среды (структурные реорганизации, текучесть кадров).

Технические характеристики проектов создания ПО:

- различная степень унифицированности проектных решений в рамках одного проекта;
- высокая техническая сложность, определяемая наличием совокупности тесно взаимодействующих компонентов (подсистем), имеющих свои локальные задачи и цели функционирования (транзакционных приложений, предъявляющих повышенные требования к надежности, безопасности и производительности, и приложений аналитической обработки (систем поддержки принятия решений), использующих нерегламентированные запросы к данным большого объема);
- отсутствие полных аналогов, ограничивающее возможность использования каких-либо типовых проектных решений и прикладных систем, высокая доля вновь разрабатываемого ПО;
- большое количество и высокая стоимость унаследованных приложений (существующего прикладного ПО), функционирующих в различной среде (персональные компьютеры, миникомпьютеры, мэйнфреймы), необходимость интеграции унаследованных и вновь разрабатываемых приложений;
- большое количество локальных объектов внедрения, территориально распределенная и неоднородная среда функционирования (СУБД, операционные системы, аппаратные платформы);
- большое количество внешних взаимодействующих систем различных организаций с различными форматами обмена информацией (налоговая служба, налоговая полиция, Госстандарт, Госкомстат, Министерство финансов, МВД, местная администрация).

Организационные характеристики проектов создания ПО:

- различные формы организации и управления проектом: централизованно управляемая разработка тиражируемого ПО, экспериментальные пилотные проекты, инициативные разработки, проекты с участием как собственных разработчиков, так и сторонних компаний на контрактной основе;
- большое количество участников проекта как со стороны заказчиков (с разнородными требованиями), так и со стороны разработчиков (более 100 человек), разобщенность и разнородность отдельных групп разработчиков по уровню квалификации, сложившимся традициям и опыту использования тех или иных инструментальных средств;
- значительная длительность жизненного цикла системы, в том числе значительная временная протяженность проекта, обусловленная масштабами организации-заказчика, различной степенью готовности отдельных ее подразделений к внедрению ПО и нестабильностью финансирования проекта;
- высокие требования со стороны заказчика к уровню технологической зрелости организаций-разработчиков (наличие сертификации в соответствии с международными и отечественными стандартами).

В конце 60-х годов прошлого века в США было отмечено явление под названием "software crisis" (кризис ПО). Это выражалось в том, что большие проекты стали выполняться с отставанием от графика или с превышением сметы расходов, разработанный продукт не обладал требуемыми функциональными возможностями, производительность его была низка, качество получаемого программного обеспечения не устраивало потребителей.

Аналитические исследования и обзоры, выполняемые в течение ряда последних лет ведущими зарубежными аналитиками, показывали не слишком обнадеживающие результаты. Так, например, результаты исследований, выполненных в 1995 году компанией Standish Group, которая проанализировала работу 364 американских корпораций и итоги выполнения более 23 тысяч проектов, связанных с разработкой ПО, выглядели следующим образом:

- только 16,2% завершились в срок, не превысили запланированный бюджет и реализовали все требуемые функции и возможности;
- 52,7% проектов завершились с опозданием, расходы превысили запланированный бюджет, требуемые функции не были реализованы в полном объеме;
- 31,1% проектов были аннулированы до завершения;
- для двух последних категорий проектов бюджет среднего проекта оказался превышенным на 89%, а срок выполнения - на 122%.

В 1998 году процентное соотношение трех перечисленных категорий проектов лишь немного изменилось в лучшую сторону (26%, 46% и 28% соответственно).

В последние годы процентное соотношение трех перечисленных категорий проектов также незначительно изменяется в лучшую сторону, однако, по оценкам ведущих аналитиков, это происходит в основном за счет снижения масштаба выполняемых проектов, а не за счет повышения управляемости и качества проектирования.

В числе причин возможных неудач, по мнению разработчиков, фигурируют:

- нечеткая и неполная формулировка требований к ПО;
- недостаточное вовлечение пользователей в работу над проектом;

- отсутствие необходимых ресурсов;
- неудовлетворительное планирование и отсутствие грамотного управления проектом;
- частое изменение требований и спецификаций;
- новизна и несовершенство используемой технологии;
- недостаточная поддержка со стороны высшего руководства;
- недостаточно высокая квалификация разработчиков, отсутствие необходимого опыта.

Объективная потребность контролировать процесс разработки сложных систем ПО, прогнозировать и гарантировать стоимость разработки, сроки и качество результатов привела в конце 60-х годов прошлого века к необходимости перехода от кустарных к индустриальным способам создания ПО и появлению совокупности инженерных методов и средств создания ПО, объединенных общим названием "программная инженерия" (software engineering). В основе программной инженерии лежит одна фундаментальная идея: проектирование ПО является формальным процессом, который можно изучать и совершенствовать. Освоение и правильное применение методов и средств создания ПО позволяет повысить его качество, обеспечить управляемость процесса проектирования ПО и увеличить срок его жизни.

В то же время, попытки чрезмерной формализации процесса, а также прямого заимствования идей и методов из других областей инженерной деятельности (строительства, производства) привели к ряду серьезных проблем. После двух десятилетий напрасных ожиданий повышения продуктивности процессов создания ПО, возлагаемых на новые методы и технологии, специалисты в индустрии ПО пришли к пониманию, что фундаментальная проблема в этой области - неспособность эффективного управления проектами создания ПО. Невозможно достичь удовлетворительных результатов от применения даже самых совершенных технологий и инструментальных средств, если они применяются бессистемно, разработчики не обладают необходимой квалификацией для работы с ними, и сам проект выполняется и управляется хаотически, в режиме "тушения пожара". Бессистемное применение технологий создания ПО (ТС ПО), в свою очередь, порождает разочарование в используемых методах и средствах (анализ мнений разработчиков показывает, что среди факторов, влияющих на эффективность создания ПО, используемым методам и средствам придается гораздо меньшее значение, чем квалификации и опыту разработчиков). Если в таких условиях отдельные проекты завершаются успешно, то этот успех достигается за счет героических усилий фанатично настроенного коллектива разработчиков. Постоянное повышение качества создаваемого ПО и снижение его стоимости может быть обеспечено только при условии достижения организацией необходимой технологической зрелости, создании эффективной инфраструктуры как в сфере разработки ПО, так и в управлении проектами. В соответствии с моделью SEI CMM (Capability Maturity Model), в хорошо подготовленной (зрелой) организации персонал обладает технологией и инструментарием оценки качества процессов создания ПО на протяжении всего жизненного цикла ПО и на уровне всей организации.

Одна из причин распространенности "хаотического" процесса создания ПО - стремление сэкономить на стадии разработки, не затрачивая времени и средств на обучение разработчиков и внедрение технологического процесса создания ПО. Эти затраты до недавнего времени были довольно значительными и составляли, по различным оценкам (в частности, Gartner Group), более $100 тыс. и около трех лет на внедрение развитой ТС ПО, охватывающей большинство процессов жизненного цикла ПО, в многочисленной команде

разработчиков (до 100 чел.). Причина - в "тяжести" технологических процессов. "Тяжелый" процесс обладает следующими особенностями:

- необходимость документировать каждое действие разработчиков;
- множество рабочих продуктов (в первую очередь - документов), создаваемых в бюрократической атмосфере;
- отсутствие гибкости;
- детерминированность (долгосрочное детальное планирование и предсказуемость всех видов деятельности, а также распределение человеческих ресурсов на длительный срок, охватывающий большую часть проекта.

Альтернативой "тяжелому" процессу является адаптивный (гибкий) процесс, основанный на принципах "быстрой разработки ПО", интенсивно развиваемых в последнее десятилетие.
Современные тенденции в программной инженерии

В начале 2001 года века ряд ведущих специалистов в области программной инженерии (Алистер Коберн, Мартин Фаулер, Джим Хайсмит, Кент Бек и другие) сформировали группу под названием Agile Alliance. Слово agile (быстрый, ловкий, стремительный) отражало в целом их подход к разработке ПО, основанный на богатом опыте участия в разнообразных проектах в течение многих лет. Этот подход под названием "Быстрая разработка ПО" (Agile software development) [10] базируется на четырех идеях, сформулированных ими в документе "Манифест быстрой разработки ПО" (Agile Alliance's Manifesto) и заключающихся в следующем:

- индивидуумы и взаимодействия между ними ценятся выше процессов и инструментов;
- работающее ПО ценится выше всеобъемлющей документации;
- сотрудничество с заказчиками ценится выше формальных договоров;
- реагирование на изменения ценится выше строгого следования плану.

При таком подходе технология занимает в процессе создания ПО вполне определенное место. Она повышает эффективность деятельности разработчиков при наличии любых из следующих четырех условий:

- когда она позволяет людям легче выразить свои мысли;
- когда она выполняет задачи, невыполнимые вручную;
- когда она автоматизирует утомительные и подверженные ошибкам действия.;
- когда она облегчает общение между людьми;

Технология не должна действовать против характера культурных ценностей и познавательной способности человека.

При этом следует четко понимать: при всех достоинствах быстрой разработки ПО этот подход не является универсальным и применим только в проектах определенного класса. Для характеристики таких проектов Алистер Коберн ввел два параметра - критичность и масштаб. Критичность определяется последствиями, вызываемыми дефектами в ПО, ее уровень может иметь одно из четырех значений:

- C - дефекты вызывают потерю удобства;
- D - дефекты вызывают потерю возместимых средств (материальных или финансовых);

- E - дефекты вызывают потерю невозместимых средств;
- L - дефекты создают угрозу человеческой жизни.

Масштаб определяется количеством разработчиков, участвующих в проекте:

- от 1 до 6 человек - малый масштаб;
- от 6 до 20 человек - средний масштаб;
- свыше 20 человек - большой масштаб.

По оценке Коберна, быстрая разработка ПО применима только в проектах малого и среднего масштаба с низкой критичностью (C или D). Общие принципы оценки технологий в таких проектах заключаются в следующем:

- интерактивное общение лицом к лицу - это самый дешевый и быстрый способ обмена информацией;
- избыточная "тяжесть" технологии стоит дорого;
- более многочисленные команды требуют более "тяжелых" и формальных технологий;
- большая формальность подходит для проектов с большей критичностью;
- возрастание обратной связи и коммуникации сокращает потребность в промежуточных и конечных продуктах;
- дисциплина, умение и понимание противостоят процессу, формальности и документированию;
- потеря эффективности в некритических видах деятельности вполне допустима.

Одним из наиболее известных примеров практической реализации подхода быстрой разработки ПО является "Экстремальное программирование" (Extreme Programming - XP) [10]. Этот метод предназначен для небольших компактных команд, нацеленных на получение как можно более высокого качества и продуктивности, и достигает этого посредством насыщенной, неформальной коммуникации, придания на персональном уровне особого значения умению и навыкам, дисциплине и пониманию, сводя к минимуму все промежуточные рабочие продукты.

Методические основы технологий создания ПО

Визуальное моделирование

Под моделью ПО в общем случае понимается формализованное описание системы ПО на определенном уровне абстракции. Каждая модель определяет конкретный аспект системы, использует набор диаграмм и документов заданного формата, а также отражает точку зрения и является объектом деятельности различных людей с конкретными интересами, ролями или задачами.

Графические (визуальные) модели представляют собой средства для визуализации, описания, проектирования и документирования архитектуры системы. Разработка модели системы ПО промышленного характера в такой же мере необходима, как и наличие проекта при строительстве большого здания. Это утверждение справедливо как в случае разработки новой системы, так и при адаптации типовых продуктов класса R/3 или BAAN, в составе которых также имеются собственные средства моделирования. Хорошие модели являются основой взаимодействия участников проекта и гарантируют корректность архитектуры. Поскольку сложность систем повышается, важно располагать хорошими методами моделирования. Хотя имеется много других факторов, от которых

зависит успех проекта, но наличие строгого стандарта языка моделирования является весьма существенным.

Состав моделей, используемых в каждом конкретном проекте, и степень их детальности в общем случае зависят от следующих факторов:

- сложности проектируемой системы;
- необходимой полноты ее описания;
- знаний и навыков участников проекта;
- времени, отведенного на проектирование.

Визуальное моделирование оказало большое влияние на развитие ТС ПО вообще и CASE-средств в частности. Понятие CASE (Computer Aided Software Engineering) используется в настоящее время в весьма широком смысле. Первоначальное значение этого понятия, ограниченное только задачами автоматизации разработки ПО, в настоящее время приобрело новый смысл, охватывающий большинство процессов жизненного цикла ПО.

CASE-технология представляет собой совокупность методов проектирования ПО, а также набор инструментальных средств, позволяющих в наглядной форме моделировать предметную область, анализировать эту модель на всех стадиях разработки и сопровождения ПО и разрабатывать приложения в соответствии с информационными потребностями пользователей. Большинство существующих CASE-средств основано на методах структурного или объектно-ориентированного анализа и проектирования, использующих спецификации в виде диаграмм или текстов для описания внешних требований, связей между моделями системы, динамики поведения системы и архитектуры программных средств.
Методы структурного анализа и проектирования ПО

В структурном анализе и проектировании используются различные модели, описывающие:

- Функциональную структуру системы;
- Последовательность выполняемых действий;
- Передачу информации между функциональными процессами;
- Отношения между данными.

Наиболее распространенными моделями первых трех групп являются:

функциональная модель SADT (Structured Analysis and Design Technique);
модель IDEF3;
DFD (Data Flow Diagrams) - диаграммы потоков данных.

Метод SADT [17] представляет собой совокупность правил и процедур, предназначенных для построения функциональной модели объекта какой-либо предметной области. Функциональная модель SADT отображает функциональную структуру объекта, т.е. производимые им действия и связи между этими действиями. Метод SADT разработан Дугласом Россом (SoftTech, Inc.) в 1969 г. для моделирования искусственных систем средней сложности. Данный метод успешно использовался в военных, промышленных и коммерческих организациях США для решения широкого круга задач, таких, как долгосрочное и стратегическое планирование, автоматизированное производство и проектирование, разработка ПО для оборонных систем, управление финансами и материально-техническим снабжением и др. Метод SADT поддерживается Министерством обороны США, которое было инициатором разработки семейства

стандартов IDEF (Icam DEFinition), являющегося основной частью программы ICAM (интегрированная компьютеризация производства), проводимой по инициативе ВВС США. Метод SADT реализован в одном стандартов этого семейства - IDEF0, который был утвержден в качестве федерального стандарта США в 1993 г., его подробные спецификации можно найти на сайте http://www.idef.com.

Модели SADT (IDEF0) традиционно используются для моделирования организационных систем (бизнес-процессов). Следует отметить, что метод SADT успешно работает только при описании хорошо специфицированных и стандартизованных бизнес-процессов в зарубежных корпорациях, поэтому он и принят в США в качестве типового. Достоинствами применения моделей SADT для описания бизнес-процессов являются:

полнота описания бизнес-процесса (управление, информационные и материальные потоки, обратные связи);
жесткие требования метода, обеспечивающих получение моделей стандартного вида;
соответствие подхода к описанию процессов стандартам ISO 9000.

В большинстве российских организаций бизнес-процессы начали формироваться и развиваться сравнительно недавно, они слабо типизированы, поэтому разумнее ориентироваться на менее жесткие модели.

Метод моделирования IDEF3 [22], являющийся частью семейства стандартов IDEF, был разработан в конце 1980-х годов для закрытого проекта ВВС США. Этот метод предназначен для таких моделей процессов, в которых важно понять последовательность выполнения действий и взаимозависимости между ними. Хотя IDEF3 и не достиг статуса федерального стандарта США, он приобрел широкое распространение среди системных аналитиков как дополнение к методу функционального моделирования IDEF0 (модели IDEF3 могут использоваться для детализации функциональных блоков IDEF0, не имеющих диаграмм декомпозиции). Основой модели IDEF3 служит так называемый сценарий процесса, который выделяет последовательность действий и подпроцессов анализируемой системы.

Диаграммы потоков данных (Data Flow Diagrams - DFD) [6] представляют собой иерархию функциональных процессов, связанных потоками данных. Цель такого представления - продемонстрировать, как каждый процесс преобразует свои входные данные в выходные, а также выявить отношения между этими процессами.

Для построения DFD традиционно используются две различные нотации, соответствующие методам Йордона-ДеМарко и Гейна-Сэрсона. Эти нотации незначительно отличаются друг от друга графическим изображением символов. В соответствии с данными методами модель системы определяется как иерархия диаграмм потоков данных, описывающих асинхронный процесс преобразования информации от ее ввода в систему до выдачи потребителю. Практически любой класс систем успешно моделируется при помощи DFD-ориентированных методов. Они с самого начала создавались как средство проектирования информационных систем (тогда как SADT - как средство моделирования систем вообще) и имеют более богатый набор элементов, адекватно отражающих специфику таких систем (например, хранилища данных являются прообразами файлов или баз данных, внешние сущности отражают взаимодействие моделируемой системы с внешним миром).

С другой стороны, эти разновидности средств структурного анализа примерно одинаковы с точки зрения возможностей изобразительных средств моделирования. При этом одним

из основных критериев выбора того или иного метода является степень владения им со стороны консультанта или аналитика, грамотность выражения своих мыслей на языке моделирования. В противном случае в моделях, построенных с использованием любого метода, будет невозможно разобраться.

Наиболее распространенным средством моделирования данных (предметной области) является модель "сущность-связь" (Entity-Relationship Model - ERM) [12]. Она была впервые введена Питером Ченом в 1976 г. Эта модель традиционно используется в структурном анализе и проектировании, однако, по существу, представляет собой подмножество объектной модели предметной области. Одна из разновидностей модели "сущность-связь" используется в методе IDEF1X, входящем в семейство стандартов IDEF и реализованном в ряде распространенных CASE-средств (в частности, AllFusion ERwin Data Modeler).

Методы объектно-ориентированного анализа и проектирования ПО. Язык UML

Концептуальной основой объектно-ориентированного анализа и проектирования ПО (ООАП) является объектная модель. Ее основные принципы (абстрагирование, инкапсуляция, модульность и иерархия) и понятия (объект, класс, атрибут, операция, интерфейс и др.) наиболее четко сформулированы Гради Бучем в его фундаментальной книге [2] и последующих работах.

Большинство современных методов ООАП [5], [9], [14], [20] основаны на использовании языка UML. Унифицированный язык моделирования UML (Unified Modeling Language) [3], [19], [21] представляет собой язык для определения, представления, проектирования и документирования программных систем, организационно-экономических систем, технических систем и других систем различной природы. UML содержит стандартный набор диаграмм и нотаций самых разнообразных видов.

UML - это преемник того поколения методов ООАП, которые появились в конце 1980-х и начале 1990-х годов. Создание UML фактически началось в конце 1994 г., когда Гради Буч и Джеймс Рамбо начали работу по объединению их методов Booch и OMT (Object Modeling Technique) под эгидой компании Rational Software. К концу 1995 г. они создали первую спецификацию объединенного метода, названного ими Unified Method, версия 0.8. Тогда же в 1995 г. к ним присоединился создатель метода OOSE (Object-Oriented Software Engineering) Ивар Якобсон. Таким образом, UML является прямым объединением и унификацией методов Буча, Рамбо и Якобсона, однако дополняет их новыми возможностями. Главными в разработке UML были следующие цели:

- предоставить пользователям готовый к использованию выразительный язык визуального моделирования, позволяющий им разрабатывать осмысленные модели и обмениваться ими;
- предусмотреть механизмы расширяемости и специализации для расширения базовых концепций;
- обеспечить независимость от конкретных языков программирования и процессов разработки.
- обеспечить формальную основу для понимания этого языка моделирования (язык должен быть одновременно точным и доступным для понимания, без лишнего формализма);
- стимулировать рост рынка объектно-ориентированных инструментальных средств;
- интегрировать лучший практический опыт.

UML находится в процессе стандартизации, проводимом OMG (Object Management Group) - организацией по стандартизации в области объектно-ориентированных методов и технологий, в настоящее время принят в качестве стандартного языка моделирования и получил широкую поддержку в индустрии ПО. UML принят на вооружение практически всеми крупнейшими компаниями - производителями ПО (Microsoft, Oracle, IBM, Hewlett-Packard, Sybase и др.). Кроме того, практически все мировые производители CASE-средств, помимо IBM Rational Software, поддерживают UML в своих продуктах (Oracle Designer, Together Control Center (Borland), AllFusion Component Modeler (Computer Associates), Microsoft Visual Modeler и др.). Полное описание UML можно найти на сайтах http://www.omg.org и http://www.rational.com.

Стандарт UML версии 1.1, принятый OMG в 1997 г., содержит следующий набор диаграмм:

Структурные (structural) модели:

- диаграммы классов (class diagrams) - для моделирования статической структуры классов системы и связей между ними;
- диаграммы компонентов (component diagrams) - для моделирования иерархии компонентов (подсистем) системы;
- диаграммы размещения (deployment diagrams) - для моделирования физической архитектуры системы.

Модели поведения (behavioral):

- диаграммы вариантов использования (use case diagrams) - для моделирования функциональных требований к системе (в виде сценариев взаимодействия пользователей с системой);
- диаграммы взаимодействия (interaction diagrams):

- диаграммы последовательности (sequence diagrams) и кооперативные диаграммы (collaboration diagrams) - для моделирования процесса обмена сообщениями между объектами;
- диаграммы состояний (statechart diagrams) - для моделирования поведения объектов системы при переходе из одного состояния в другое;
- диаграммы деятельности (activity diagrams) - для моделирования поведения системы в рамках различных вариантов использования, или потоков управления.

Диаграммы вариантов использования показывают взаимодействия между вариантами использования и действующими лицами, отражая функциональные требования к системе с точки зрения пользователя. Цель построения диаграмм вариантов использования - это документирование функциональных требований в самом общем виде, поэтому они должны быть предельно простыми.

Вариант использования представляет собой последовательность действий (транзакций), выполняемых системой в ответ на событие, инициируемое некоторым внешним объектом (действующим лицом). Вариант использования описывает типичное взаимодействие между пользователем и системой и отражает представление о поведении системы с точки зрения пользователя. В простейшем случае вариант использования определяется в процессе обсуждения с пользователем тех функций, которые он хотел бы реализовать, или целей, которые он преследует по отношению к разрабатываемой системе.

Диаграмма вариантов использования является самым общим представлением функциональных требований к системе. Для последующего проектирования системы требуются более конкретные детали, которые описываются в документе, называемом "сценарием варианта использования" или "потоком событий" (flow of events). Сценарий подробно документирует процесс взаимодействия действующего лица с системой, реализуемого в рамках варианта использования [11]. Основной поток событий описывает нормальный ход событий (при отсутствии ошибок). Альтернативные потоки описывают отклонения от нормального хода событий (ошибочные ситуации) и их обработку.

Достоинства модели вариантов использования заключаются в том, что она:

- определяет пользователей и границы системы;
- определяет системный интерфейс;
- удобна для общения пользователей с разработчиками;
- используется для написания тестов;
- является основой для написания пользовательской документации;
- хорошо вписывается в любые методы проектирования (как объектно-ориентированные, так и структурные).

Диаграммы взаимодействия описывают поведение взаимодействующих групп объектов (в рамках варианта использования или некоторой операции класса). Как правило, диаграмма взаимодействия охватывает поведение объектов в рамках только одного потока событий варианта использования. На такой диаграмме отображается ряд объектов и те сообщения, которыми они обмениваются между собой. Существует два вида диаграмм взаимодействия: диаграммы последовательности и кооперативные диаграммы.

Диаграммы последовательности отражают временную последовательность событий, происходящих в рамках варианта использования, а кооперативные диаграммы концентрируют внимание на связях между объектами.

Диаграмма классов определяет типы классов системы и различного рода статические связи, которые существуют между ними. На диаграммах классов изображаются также атрибуты классов, операции классов и ограничения, которые накладываются на связи между классами. Вид и интерпретация диаграммы классов существенно зависит от точки зрения (уровня абстракции): классы могут представлять сущности предметной области (в процессе анализа) или элементы программной системы (в процессах проектирования и реализации).

Диаграммы состояний определяют все возможные состояния, в которых может находиться конкретный объект, а также процесс смены состояний объекта в результате наступления некоторых событий. Диаграммы состояний не надо создавать для каждого класса, они применяются только в сложных случаях. Если объект класса может существовать в нескольких состояниях и в каждом из них ведет себя по-разному, для него может потребоваться такая диаграмма.

Диаграммы деятельности, в отличие от большинства других средств UML, заимствуют идеи из нескольких различных методов, в частности, метода моделирования состояний SDL и сетей Петри. Эти диаграммы особенно полезны в описании поведения, включающего большое количество параллельных процессов. Диаграммы деятельности являются также полезными при параллельном программировании, поскольку можно графически изобразить все ветви и определить, когда их необходимо синхронизировать.

Диаграммы деятельности можно применять для описания потоков событий в вариантах использования. С помощью текстового описания можно достаточно подробно рассказать о потоке событий, но в сложных и запутанных потоках с множеством альтернативных ветвей будет трудно понять логику событий. Диаграммы деятельности предоставляют ту же информацию, что и текстовое описание потока событий, но в наглядной графической форме.

Диаграммы компонентов моделируют физический уровень системы. На них изображаются компоненты ПО и связи между ними. На такой диаграмме обычно выделяют два типа компонентов: исполняемые компоненты и библиотеки кода.

Каждый класс модели (или подсистема) преобразуется в компонент исходного кода. Между отдельными компонентами изображают зависимости, соответствующие зависимостям на этапе компиляции или выполнения программы.

Диаграммы компонентов применяются теми участниками проекта, кто отвечает за компиляцию и сборку системы. Они нужны там, где начинается генерация кода.

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

Диаграмма размещения показывает физическое расположение сети и местонахождение в ней различных компонентов. Ее основными элементами являются узел (вычислительный ресурс) и соединение - канал взаимодействия узлов (сеть).

Диаграмма размещения используется менеджером проекта, пользователями, архитектором системы и эксплуатационным персоналом, чтобы понять физическое размещение системы и расположение ее отдельных подсистем.

UML обладает механизмами расширения, предназначенными для того, чтобы разработчики могли адаптировать язык моделирования к своим конкретным нуждам, не меняя при этом его метамодель. Наличие механизмов расширения принципиально отличает UML от таких средств моделирования, как IDEF0, IDEF1X, IDEF3, DFD и ERM. Перечисленные языки моделирования можно определить как сильно типизированные (по аналогии с языками программирования), поскольку они не допускают произвольной интерпретации семантики элементов моделей. UML, допуская такую интерпретацию (в основном за счет стереотипов), является слабо типизированным языком. К его механизмам расширения относятся:

- стереотипы;
- тегированные (именованные) значения;
- ограничения.

Стереотип - это новый тип элемента модели, который определяется на основе уже существующего элемента. Стереотипы расширяют нотацию модели и могут применяться к любым элементам модели. Стереотипы классов - это механизм, позволяющий разделять классы на категории. Разработчики ПО могут создавать свои собственные наборы стереотипов, формируя тем самым специализированные подмножества UML (например, для описания бизнес-процессов, Web-приложений, баз данных и т.д.). Такие подмножества (наборы стереотипов) в стандарте языка UML носят название профилей языка.

Именованное значение - это пара строк "тег = значение", или "имя = содержимое", в которых хранится дополнительная информация о каком-либо элементе системы, например, время создания, статус разработки или тестирования, время окончания работы над ним и т.п.

Ограничение - это семантическое ограничение, имеющее вид текстового выражения на естественном или формальном языке (OCL - Object Constraint Language), которое невозможно выразить с помощью нотации UML.

4.1. Понятие гипертекста

В 1945 г. Ваневар Буш - научный советник президента США Г. Трумена, проанализировал способы представления информации в виде отчетов, докладов, проектов, графиков, планов и, поняв неэффективность такого представления, предложил способ размещения информации по принципу ассоциативного мышления. На основе этого принципа была разработана модель гипотетической машины "МЕМЕКС". Через 20 лет Теодор Нельсон реализовал этот принцип на ЭВМ и назвал его гипертекстом.

Гипертекст обладает нелинейной сетевой формой организации материала, разделенного на фрагменты, для каждого из которых указан переход к другим фрагментам по определенным типам связей. При установлении связей можно опираться на разные основания (ключи), но в любом случае речь идет о смысловой, семантической близости связываемых фрагментов. Следуя указанным связям, можно читать или осваивать материал в любом порядке. Текст теряет свою замкнутость, становится принципиально открытым, в него можно вставлять новые фрагменты, указывая для них связи с имеющимися фрагментами. Структура текста не нарушается, и вообще у гипертекста нет априорно заданной структуры. Таким образом, гипертекст - это технология представления неструктурного свободно наращиваемого знания.

Под гипертекстом понимают систему информационных объектов, объединенных между собой направленными семантическими связями, образующими сеть. Каждый объект связывается с информационной панелью экрана, на которой пользователь может ассоциативно выбирать одну из связей.

Гипертекстовая технология предполагает перемещение от одних объектов к другим с учетом их смысловой, семантической связанности. Обработке информации по правилам формального вывода в гипертекстовой технологии соответствует запоминание пути перемещения по гипертекстовой сети. Пользователь сам определяет подход к изучению материала, учитывая свои индивидуальные способности, знания, уровень квалификации и подготовки.

Гипертекст содержит не только информацию, но и аппарат ее эффективного поиска. Структурно гипертекст состоит из информационного материала, тезауруса гипертекста, списка главных тем и алфавитного словаря.

Информационный материал подразделяется на информационные статьи, состоящие из заголовка статьи и текста. Заголовок содержит тему или наименование описываемого объекта. Информационная статья содержит традиционные определения и понятия, должна занимать одну панель и быть легко обозримой, чтобы пользователь мог понять, стоит ли ее внимательно читать или перейти к другим, близким по смыслу статьям. Текст,

включаемый в информационную статью, может сопровождаться пояснениями, примерами, графиками, документами и видеоизображениями объектов реального мира. Ключевые слова для связи с другими информационными статьями должны визуально различаться.

Тезаурус гипертекста - это автоматизированный словарь, отображающий семантические отношения между лексическими единицами информационно-поискового языка и предназначенный для поиска слов по их смысловому содержанию.

Термин "тезаурус" был введен в XIII в. флорентийцем Брунетто Лотики для названия энциклопедии. С греческого языка этот термин переводится как сокровище, запас, богатство. Тезаурус гипертекста состоит из тезаурусных статей, каждая из которых имеет заголовок и список заголовков родственных тезаурусных статей, где указаны тип родства и заголовки тезаурусных статей. Заголовок тезаурусной статьи совпадает с заголовком информационной статьи и является наименованием объекта, описание которого содержится в информационной статье. Формирование тезаурусной статьи гипертекста означает индексирование текста.

Список главных тем содержит заголовки всех справочных статей, для которых нет ссылок с отношениями "род - вид", "часть - целое". Желательно, чтобы список занимал не более одной панели экрана.

Алфавитный словарь содержит перечень наименований всех информационных статей в алфавитном порядке.

Изучая информацию, представленную в виде гипертекста, пользователь может знакомиться с последовательностями блоков данных. Процесс выбора последовательностей этих блоков, т. е. методику вождения пользователя от одного объекта к другому, называют навигацией. При этом выделяют терминологическую навигацию - последовательное движение по терминам, друг из друга вытекающим, и тематическую навигацию, с помощью которой пользователь должен получить для чтения все статьи, необходимые для изучения нужной ему темы.

4.2. Системы мультимедиа

Мультимедиа - это интерактивная технология, обеспечивающая работу с графическими изображениями, видеоизображением, анимацией, текстом и звуковым рядом.

Появлению систем мультимедиа способствовал технический прогресс в вычислительных системах: возросла оперативная память у ЭВМ, появились широкие графические возможности (изображение стало цветным и объемным); улучшилось качество аудио- и видеотехники, появились лазерные компакт-диски и др.

Стив Джобс в 1988 г. создал принципиально новый тип персонального компьютера - NeXT, у которого базовые средства систем мультимедиа заложены в архитектуру и программные средства. NeXT дает возможность работать с интерфейсом SILK (речь, образ, язык, знания). В состав NeXT входит система электронной мультимедиа-почты, позволяющая обмениваться сообщениями типа речи, текста, графической информации.

Интерфейсы

Как любое техническое устройство, компьютер обменивается информацией с человеком посредством набора определенных правил, обязательных как для машины, так и для человека. Эти правила в компьютерной литературе называются интерфейсом. От интерфейса зависит технология общения человека с компьютером. Можно выделить следующие виды интерфейса: командный интерфейс, графический WIMP-интерфейс, SILK-интерфейс.

1. Командный интерфейс. Этот интерфейс называется так потому, что в этом виде интерфейса человек подает команды компьютеру, а компьютер их выполняет и выдает результат человеку. Командный интерфейс реализован в виде пакетной технологии и технологии командной строки.

2. Графический WIMP-интерфейс (Window- окно, Image - образ, Menu - меню, Pointer - указатель). Характерной особенностью этого вида интерфейса является то, что диалог с пользователем ведется не с помощью команд, а с помощью графических образов - меню, окон, других элементов.

3. SILK-интерфейс (Speech - речь, Image - образ, Lanquage - язык, Knowlege - знание). Этот вид интерфейса наиболее приближен к обычной, человеческой форме общения. В рамках этого интерфейса идет речевое общение человека и компьютера. При этом компьютер находит для себя команды, анализируя человеческую речь и находя в ней ключевые фразы. Результат выполнения команд он также преобразует в понятную человеку форму.

Графический интерфейс
Отличительные особенности этого интерфейса.

- Выделение областей экрана.
- Переопределение клавиш клавиатуры в зависимости от контекста.
- Использование манипуляторов и серых клавиш клавиатуры для управления курсором.
- Широкое использование цветных мониторов.

Появление этого типа интерфейса совпало с широким распространением операционной системы MS DOS. Типичным примером использования этого вида интерфейса является файловая оболочка Norton Commander и текстовый процессор Microsoft Word for Dos.

Вторым этапом в развитии графического интерфейса стал «чистый» интерфейс WIMP. Он характеризуется следующими особенностями.

- Вся работа с программами, файлами и документами происходит в окнах - определенных очерченных рамкой частях экрана.
- Все программы, файлы, документы, устройства и другие объекты представляются в виде значков - иконок. При открытии иконки превращаются в окна.
- Все действия с объектами осуществляются с помощью меню. Хотя меню появилось на первом этапе становления графического интерфейса, оно не имело в нем главенствующего значения, а служило лишь дополнением к командной строке. В чистом WIMP- интерфейсе меню становится основным элементом управления.
- Широкое использование манипуляторов для указания на объекты. С помощью манипулятора указывают на любую область экрана, окна или иконки, выделяют ее, а уже потом через меню или с использованием других технологий осуществляют управление ими.

Важнейшей особенностью этого интерфейса является его понятность и простота в усвоении. Поэтому сейчас WIMP-интерфейс стал стандартом де-факто. Ярким примером программ с графическим интерфейсом является операционная система Microsoft Windows. Графический интерфейс

SILK-интерфейс

С середины 90-х годов XX в связи с появлением звуковых карт и широкого распространения технологий распознавания речи начинается активное развитие и применение «речевой технологии» SILK -интерфейса. При этой технологии команды подаются голосом путем произнесения специальных зарезервированных слов - команд. Такими основными командами (по правилам системы речевого ввода «Горыныч») являются:
«Проснись» - включение голосового интерфейса.
«Отдыхай» - выключение речевого интерфейса.
«Открыть» - переход в режим вызова той или иной программы. Имя программы называется в следующем слове
«Буду диктовать» - переход из режима команд в режим набора текста голосом.
«Режим команд» - возврат в режим подачи команд голосом
и некоторые другие.

Слова должны выговариваться четко, в одном темпе. Между словами обязательна пауза. Из-за неразвитости алгоритма распознавания речи такие системы требуют индивидуальной предварительной настройки на каждого конкретного пользователя. В состав Office XP уже вошла система распознавания речи, правда, она пока понимает лишь английский, китайский и японский языки.

Мультимедиа-акселератор - программно-аппаратные средства, которые объединяют базовые возможности графических акселераторов (см. ниже) с одной или несколькими мультимедийными функциями, обычно требующими установки в компьютер дополнительных устройств.

К мультимедийным функциям относятся следующие:

- цифровая фильтрация;
- масштабированное видео;
- аппаратно-цифровое сжатие и развертка видео;
- ускорение графических операций, связанных с трехмерной графикой;
- поддержка "живого" видео на мониторе;
- наличие композитного видеовыхода;
- вывод телевизионного сигнала на монитор.

Графический акселератор - представляет собой программно-аппаратные средства ускорения графических операций.

Технология мультимедиа позволила заменить техноцентрический подход (планирование индустрии зависит от прогноза возможных технологий) на антропоцентрический подход (индустрия управляется рынком); дает возможность динамически отслеживать индивидуальные запросы мирового рынка, что отражается в тенденции перехода к мелкосерийному производству.

Широкое применение технология мультимедиа получает в сфере образования. Созданы видеоэнциклопедии по многим школьным предметам, музеям, городам, маршрутам путешествий. Созданы игровые ситуационные тренажеры, позволяющие сокращать время обучения.

Термин "виртуальная реальность" был введен в 1989 г. для обозначения искусственного трехмерного мира (киберпространства), создаваемого мультимедийными технологиями и воспринимаемого человеком посредством специальных устройств: шлемов, очков, перчаток и т. д. Киберпространство отличается от обычной компьютерной анимации более точным воспроизведением деталей и работает в режиме реального времени. Человек видит не изображение на плоском экране дисплея, а воспринимает объект объемно, как в реальном мире. Создается диалоговое кино, где пользователь может управлять ходом зрелища с клавиатуры, дисплея или посредством реплик, если к компьютеру подключена плата распознавания речи.

СОВРЕМЕННЫЕ ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ АВТОМАТИЗАЦИИ ОФИСА

В настоящее время существует множество программных продуктов, обеспечивающих информационные технологии автоматизации офиса. К ним относятся текстовые процессоры, табличные процессоры, системы управления базами данных, электронная почта, электронный календарь, компьютерные конференции, видеотекст, а также специализированные программы управленческой деятельности: ведение документов, контроль исполнения приказов и т. д.

Текстовые процессоры предназначены для создания и обработки текстовых документов. Подготовленные текстовые документы могут быть распечатаны, а также переданы по компьютерной сети. Таким образом, в распоряжении менеджера оказывается эффективный вид письменной коммуникации.

Табличные процессоры позволяют выполнять многочисленные операции над данными, представленными в табличной форме. Пользователь имеет возможность вводить табличные данные, обрабатывать их, проводить необходимые вычисления, автоматически формировать итоги, выводить информацию в печатном виде и в виде импортируемых в другие системы файлов, качественно оформлять табличные данные, в том числе в виде графиков и диаграмм, проводить инженерные, финансовые, статистические расчеты, проводить математическое моделирование и т. д.

Системы управления базами данных предназначены для создания и поддержания в актуальном состоянии баз данных, содержащих различные сведения о системе управления и производственной деятельности фирмы.

Электронная почта позволяет пользователю получать, хранить и отправлять сообщения своим партнерам по сети. Возможности, предоставляемые пользователю электронной почтой, различны, и зависят от применяемого программного обеспечения.

Электронный календарь предоставляет средства для хранения и управления рабочим расписанием менеджеров и других работников предприятия. Система позволяет установить дату, время и место (аудиторию) встречи или другого мероприятия, согласовав их с расписанием всех участвующих в нем менеджеров.

Компьютерные конференции используют компьютерные сети для обмена информацией между участниками группы, решающей определенную задачу.

Видеотекст основан на использовании компьютера для получения,отображений текстовых и графических данных на экране монитора. В настоящее время все более широкое распространение получает обмен между компаниями каталогами и прайс-листами а также заказ газет, журналов и другой печатной продукции в форме видеотекста.

Динамический обмен данными (DDE)

DDE — давний и прижившийся протокол обмена данными между разными приложениями, появившийся еще на заре эры Windows. С тех пор на его базе был создан интерфейс OLE, а в 32-разрядном API Windows появились и другие методы межпрограммного взаимодействия. Но ниша, занимаемая DDE, осталась неизменной — это оперативная передача и синхронизация данных в приложениях.

Приложения, использующие DDE, разделяются на две категории — клиенты и серверы (не путать с одноименной архитектурой СУБД). Оба участника процесса осуществляют контакты (conversations) по определенным темам (topic), при этом в рамках темы производится обмен элементами данных (items). Устанавливает контакт клиент, который посылает запрос, содержащий имена контакта и темы. После установления контакта всякое изменение элемента данных на сервере передается данным клиента. Подробно функции DDE описаны в [4].

Первоначально программирование DDE было чрезвычайно сложным делом — оно требовало взаимосвязанной обработки более чем десяти сообщений Windows. В версии Windows 3.1 появилась библиотека DDEML, которая перевела управление DDE на уровень вызова процедур. Разработчики подсистемы DDE в Delphi, верные идеологии создания VCL, свели интерфейс этого протокола к четырем компонентам — двум для сервера и двум для клиента.

Связывание и внедрение объектов

Итак, OLE — это протокол, позволяющий создавать составные документы, которые включают в себя документы, созданные другими приложениями. Документ, который включает в себя другие документы, называется документом-контейнером OLE. В данном случае документами-контейнерами являются формы и отчеты Access. Документы, которые включаются в форму или отчет, называются документами-источниками или объектами OLE. Объектами OLE могут быть документы Word, Excel, рисунки, созданные в одном из графических редакторов, например Paint, видеоролики (файлы с расширением avi), звуковые файлы с расширением wav. Объекты OLE отличаются от объектов Automation, о которых мы будем говорить ниже, тем, что они являются документами, получаемыми с помощью приложения, а не частью его модели объектов.

Объекты OLE могут быть либо внедрены в документ-контейнер, либо связаны с ним. Приложение, которое поставляет объекты для внедрения и связывания, называется сервером OLE. Внедренный объект представляет собой копию документа-источника, который сохраняется вместе с формой или отчетом. Связанный объект хранится в отдельном файле, и документ-контейнер содержит только указатель на исходный файл объекта. Если кто-либо обновляет исходный файл объекта, то обновляется и

представление объекта в составном документе. Коварство связанных объектов заключается в том, что при изменении местоположения исходного файла относительно составного документа, либо при изменении местоположения составного документа таким образом, что исходный файл становится недоступным, связь разрывается. Внедренные объекты всегда доступны, однако частое их использование приводит к непомерному увеличению файла составного документа.

Выполнив внедрение или связывание объекта OLE, можно легко активизировать из документа Access приложение, которому этот объект принадлежит. Для этого достаточно дважды щелкнуть левой кнопкой мыши по внедренному объекту, после чего объект может быть изменен. Когда же активизированное приложение будет закрыто, в документе-контейнере отразится (а в случае внедрения — сохранится) внесенное изменение.

Понятие жизненного цикла ПО ИС.

Процессы жизненного цикла: основные, вспомогательные, организационные. Содержание и взаимосвязь процессов жизненного цикла ПО ИС. Модели жизненного цикла: каскадная, модель с промежуточным контролем, спиральная. Стадии жизненного цикла ПО ИС. Регламентация процессов проектирования в отечественных и международных стандартах.

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

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

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

В настоящее время известны и используются следующие модели жизненного цикла:
Каскадная модель (рис. 2.1) предусматривает последовательное выполнение всех этапов проекта в строго фиксированном порядке. Переход на следующий этап означает полное завершение работ на предыдущем этапе.
Поэтапная модель с промежуточным контролем (рис. 2.2). Разработка ИС ведется итерациями с циклами обратной связи между этапами. Межэтапные корректировки позволяют учитывать реально существующее взаимовлияние результатов разработки на различных этапах; время жизни каждого из этапов растягивается на весь период разработки.
Спиральная модель (рис. 2.3). На каждом витке спирали выполняется создание очередной версии продукта, уточняются требования проекта, определяется его качество и планируются работы следующего витка.Особое внимание уделяется начальным этапам разработки - анализу и проектированию, где реализуемость тех или иных технических

решений проверяется и обосновывается посредством создания прототипов (макетирования).

Рис. 2.1. Каскадная модель ЖЦ ИС

Рис. 2.2. Поэтапная модель с промежуточным контролем

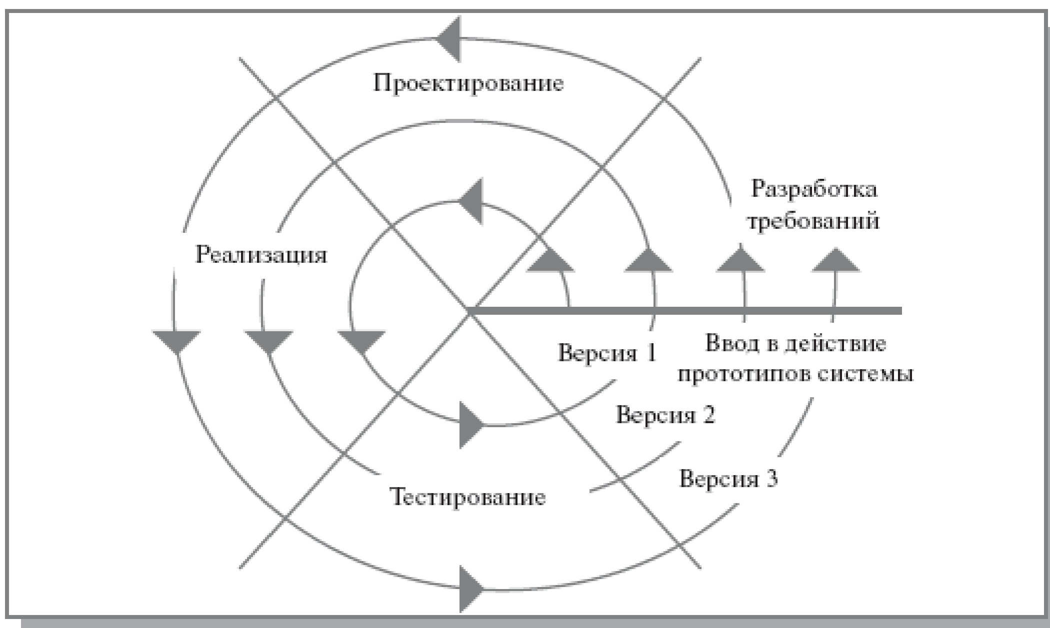


Рис. 2.3. Спиральная модель ЖЦ ИС

На практике наибольшее распространение получили две основные модели жизненного цикла:
каскадная модель (характерна для периода 1970-1985 гг.);
спиральная модель (характерна для периода после 1986.г.).

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

Можно выделить следующие положительные стороны применения каскадного подхода:
на каждом этапе формируется законченный набор проектной документации, отвечающий критериям полноты и согласованности;
выполняемые в логической последовательности этапы работ позволяют планировать сроки завершения всех работ и соответствующие затраты.

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

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

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

Итеративная разработка отражает объективно существующий спиральный цикл создания сложных систем. Она позволяет переходить на следующий этап, не дожидаясь полного завершения работы на текущем и решить главную задачу - как можно быстрее показать пользователям системы работоспособный продукт, тем самым активизируя процесс уточнения и дополнения требований.

Основная проблема спирального цикла - определение момента перехода на следующий этап. Для ее решения вводятся временные ограничения на каждый из этапов жизненного цикла, и переход осуществляется в соответствии с планом, даже если не вся

запланированная работа закончена. Планирование производится на основе статистических данных, полученных в предыдущих проектах, и личного опыта разработчиков.

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

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

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

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

Среди наиболее известных стандартов можно выделить следующие:

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

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

Основные процессы:
- приобретение;
- поставка;
- разработка;
- эксплуатация;
- сопровождение.

Вспомогательные процессы:
- документирование;
- управление конфигурацией;
- обеспечение качества;
- разрешение проблем;
- аудит;
- аттестация;
- совместная оценка;
- верификация.

Организационные процессы:
- создание инфраструктуры;
- управление;
- обучение;
- усовершенствование.

РАЗРАБОТКА СТРУКТУРЫ ПРОГРАММЫ И МОДУЛЬНОЕ ПРОГРАММИРОВАНИЕ

7.1. Цель модульного программирования.

Приступая к разработке каждой программы ПС, следует иметь ввиду, что она, как правило, является большой системой, поэтому мы должны принять меры для ее упрощения. Для этого такую программу разрабатывают по частям, которые называются программными модулями [7.1, 7.2]. А сам такой метод разработки программ называют модульным программированием [7.3].

Программный модуль - это любой фрагмент описания процесса, оформляемый как самостоятельный программный продукт, пригодный для использования в описаниях процесса.

Это означает, что каждый программный модуль программируется, компилируется и отлаживается отдельно от других модулей программы, и тем самым, физически разделен с другими модулями программы. Более того, каждый разработанный программный модуль может включаться в состав разных программ, если выполнены условия его использования, декларированные в документации по этому модулю.

Модульное программирование является воплощением в процессе разработки программ обоих общих методов борьбы со сложностью (см. лекцию 3, п. 3.5): и обеспечение независимости компонент системы, и использование иерархических структур. Для воплощения первого метода формулируются определенные требования, которым должен удовлетворять программный модуль, т.е. выявляются основные характеристики "хорошего" программного модуля. Для воплощения второго метода используют древовидные модульные структуры программ (включая деревья со сросшимися ветвями).

7.2. Основные характеристики программного модуля.

Не всякий программный модуль способствует упрощению программы [7.2]. Выделить хороший с этой точки зрения модуль является серьезной творческой задачей. Для оценки приемлемости выделенного модуля используются некоторые критерии. Так, Хольт [7.4] предложил следующие два общих таких критерия:

- хороший модуль снаружи проще, чем внутри;
- хороший модуль проще использовать, чем построить.

Майерс [7.5] предлагает использовать более конструктивные характеристики программного модуля для оценки его приемлемости:

- размер модуля;
- прочность модуля;
- сцепление с другими модулями;
- рутинность модуля (независимость от предыстории обращений к не-му).

Размер модуля измеряется числом содержащихся в нем операторов (строк). Модуль не должен быть слишком маленьким или слишком большим. Маленькие модули приводят к громоздкой модульной структуре программы и могут не окупать накладных расходов, связанных с их оформлением. Большие модули неудобны для изучения и изменений, они могут существенно увеличить суммарное время повторных трансляций программы при отладке программы. Обычно рекомендуются программные модули размером от нескольких десятков до нескольких сотен операторов.

Прочность модуля - это мера его внутренних связей. Чем выше прочность модуля, тем больше связей он может спрятать от внешнейпо отношению к нему части программы и, следовательно, тем больший вклад в упрощение программы он может внести.

- Функционально прочный модуль - это модуль, выполняющий (реализующий) одну какую-либо определенную функцию. При реализации этой функции такой модуль может использовать и другие модули. Такой класс программных модулей рекомендуется для использования.
- Информационно прочный модуль - это модуль, выполняющий (реализующий) несколько операций (функций) над одной и той же структурой данных (информационным объектом), которая считается неизвестной вне этого модуля. Для каждой из этих операций в таком модуле имеется свой вход со своей формой обращения к нему. Информационно-прочный модуль может реализовывать, например, абстрактный тип данных.

Сцепление модуля - это мера его зависимости по данным от других модулей. Характеризуется способом передачи данных. Чем слабее сцепление модуля с другими модулями, тем сильнее его независимость от других модулей.
Единственным видом сцепления модулей, который рекомендуется для использования современной технологией программирования, является параметрическое сцепление (сцепление по данным по Майерсу [7.5]) - это случай, когда данные передаются модулю либо при обращении к нему как значения его параметров, либо как результат его обращения к другому модулю для вычисления некоторой функции. Такой вид сцепления модулей реализуется на языках программирования при использовании обращений к процедурам (функциям).

Рутинность модуля - это его независимость от предыстории обращений к нему. Модуль будем называть рутинным, если результат (эффект) обращения к нему зависит только от значений его параметров (и не зависит от предыстории обращений к нему). Модуль будем

называть зависящим от предыстории, если результат (эффект) обращения к нему зависит от внутреннего состояния этого модуля, храняшего следы предыдущих обращений к нему.

- всегда следует использовать рутинный модуль, если это не приводит к плохим (не рекомендуемым) сцеплениям модулей;
- зависящие от предыстории модули следует использовать только в случае, когда это необходимо для обеспечения параметрического сцепления;
- в спецификации зависящего от предыстории модуля должна быть четко сформулирована эта зависимость таким образом, чтобы было возможно прогнозировать поведение (эффект выполнения) данного модуля при разных последующих обращениях к нему.

7.3. Методы разработки структуры программы.

Как уже отмечалось выше, в качестве модульной структуры программы принято использовать древовидную структуру, включая деревья со сросшимися ветвями. В узлах такого дерева размещаются программные модули, а направленные дуги (стрелки) показывают статическую подчиненность модулей, т.е. каждая дуга показывает, что в тексте модуля, из которого она исходит, имеется ссылка на модуль, в который она входит. Другими словами, каждый модуль может обращаться к подчиненным ему модулям, т.е. выражается через эти модули.

В процессе разработки программы ее модульная структура может по-разному формироваться и использоваться для определения порядка программирования и отладки модулей, указанных в этой структуре. Обычно в литературе обсуждаются два метода [7.1, 7.7]: метод восходящей разработки и метод нисходящей разработки.

Метод восходящей разработки заключается в следующем. Сначала строится модульная структура программы в виде дерева. Затем поочередно программируются модули программы, начиная с модулей самого нижнего уровня (листья дерева модульной структуры программы), в таком порядке, чтобы для каждого программируемого модуля были уже запрограммированы все модули, к которым он может обращаться. После того, как все модули программы запрограммированы, производится их поочередное тестирование и отладка в принципе в таком же (восходящем) порядке, в каком велось их программирование. На первый взгляд такой порядок разработки программы кажется вполне естественным: каждый модуль при программировании выражается через уже запрограммированные непосредственно подчиненные модули, а при тестировании использует уже отлаженные модули. Однако, современная технология не рекомендует такой порядок разработки программы.
Во-первых, для программирования какого-либо модуля совсем не требуется текстов используемых им модулей - для этого достаточно, чтобы каждый используемый модуль был лишь специфицирован (в объеме, позволяющем построить правильное обращение к нему), а для тестирования его возможно (и даже, как мы покажем ниже, полезно) используемые модули заменять их имитаторами (заглушками).
Во-вторых, каждая программа в какой-то степени подчиняется некоторым внутренним для нее, но глобальным для ее модулей соображениям (принципам реализации, предположениям, структурам данных и т.п.), что определяет ее концептуальную целостность и формируется в процессе ее разработки.
В-третьих, при восходящем тестировании для каждого модуля (кроме головного) приходится создавать ведущую программу (модуль), которая должна подготовить для тестируемого модуля необходимое состояние информационной среды и произвести требуемое обращение к нему. Это приводит к большому объему "отладочного" программирования и в то же время не дает никакой гарантии, что тестирование модулей

производилось именно в тех условиях, в которых они будут выполняться в рабочей программе.

Метод нисходящей разработки заключается в следующем. Как и в предыдущем методе сначала строится модульная структура программы в виде дерева. Затем поочередно программируются модули программы, начиная с модуля самого верхнего уровня (головного), переходя к программированию какого-либо другого модуля только в том случае, если уже запрограммирован модуль, который к нему обращается. После того, как все модули программы запрограммированы, производится их поочередное тестирование и отладка в таком же (нисходящем) порядке. При таком порядке разработки программы вся необходимая глобальная информация формируется своевременно, т.е. ликвидируется весьма неприятный источник просчетов при программировании модулей. Существенно облегчается и тестирование модулей, производимое при нисходящем тестировании программы. Первым тестируется головной модуль программы, который представляет всю тестируемую программу и поэтому тестируется при "естественном" состоянии информационной среды, при котором начинает выполняться эта программа. При этом все модули, к которым может обращаться головной, заменяются на их имитаторы (так называемые заглушки [7.5]).

Некоторым недостатком нисходящей разработки, приводящим к определенным затруднениям при ее применении, является необходимость абстрагироваться от базовых возможностей используемого языка программирования, выдумывая абстрактные операции, которые позже нужно будет реализовать с помощью выделенных в программе модулей. Однако способность к таким абстракциям представляется необходимым условием разработки больших программных средств, поэтому ее нужно развивать.

В рассмотренных методах восходящей и нисходящей разработок (которые мы будем называть классическими) модульная древовидная структуру программы должна разрабатываться до начала программирования модулей. Однако такой подход вызывает ряд возражений: представляется сомнительным, чтобы до программирования модулей можно было разработать структуру программы достаточно точно и содержательно. На самом деле это делать не обязательно: так при конструктивном и архитектурном подходах к разработке программ [7.3] модульная структура формируется в процессе программирования модулей.

Конструктивный подход к разработке программы представляет собой модификацию нисходящей разработки, при которой модульная древовидная структура программы формируется в процессе программирования модуля. Сначала программируется головной модуль, исходя из спецификации программы в целом, причем спецификация программы является одновременно и спецификацией ее головного модуля, так как последний полностью берет на себя ответственность за выполнение функций программы. В процессе программирования головного модуля, в случае, если эта программа достаточно большая, выделяются подзадачи (внутренние функции), в терминах которых программируется головной модуль. Это означает, что для каждой выделяемой подзадачи (функции) создается спецификация реализующего ее фрагмента программы, который в дальнейшем может быть представлен некоторым поддеревом модулей. Важно заметить, что здесь также ответственность за выполнение выделенной функции берет головной (может быть, и единственный) модуль этого поддерева, так что спецификация выделенной функции является одновременно и спецификацией головного модуля этого поддерева. В головном модуле программы для обращения к выделенной функции строится обращение к головному модулю указанного поддерева в соответствии с созданной его спецификацией. Таким образом, на первом шаге разработки программы (при программировании ее

головного модуля) формируется верхняя начальная часть дерева, например, такая, которая показана на рис. 7.1.

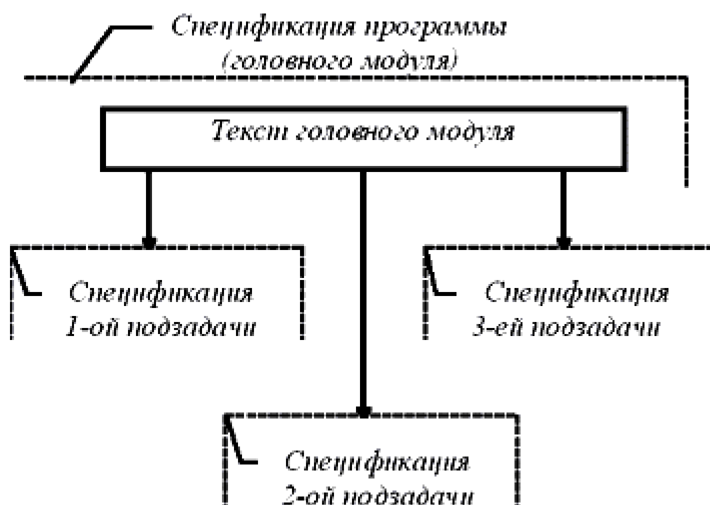


Рис. 7.1. Первый шаг формирования модульной структуры программы при конструктивном подходе.

Аналогичные действия производятся при программировании любого другого модуля, который выбирается из текущего состояния дерева программы из числа специфицированных, но пока еще не запрограммированных модулей. В результате этого производится очередное доформирование дерева программы, например, такое, которое показано на рис. 7.2.

Архитектурный подход к разработке программы представляет собой модификацию восходящей разработки, при которой модульная структура программы формируется в процессе программирования модуля. Но при этом ставится существенно другая цель разработки: повышение уровня используемого языка программирования, а не разработка конкретной программы.

Это означает, что для заданной предметной области выделяются типичные функции, каждая из которых может использоваться при решении разных задач в этой области, и специфицируются, а затем и программируются отдельные программные модули, выполняющие эти функции.

Так как процесс выделения таких функций связан с накоплением и обобщением опыта решения задач в заданной предметной области, то обычно сначала выделяются и реализуются отдельными модулями более простые функции, а затем постепенно появляются модули, использующие ранее выделенные функции.

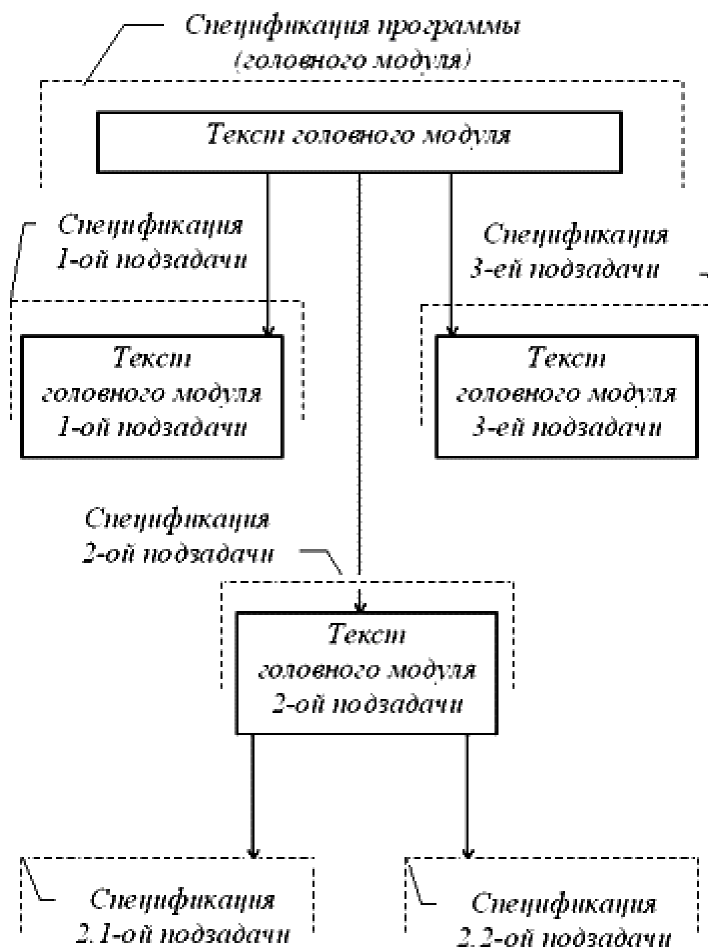


Рис. 7.2. Второй шаг формирования модульной структуры программы при конструктивном подходе.

В классическом методе нисходящей разработки рекомендуется сначала все модули разрабатываемой программы запрограммировать, а уж затем начинать нисходящее их тестирование [7.5]. Однако такой порядок разработки не представляется достаточно обоснованным: тестирование и отладка модулей может привести к изменению спецификации подчиненных модулей и даже к изменению самой модульной структуры программы, так что в этом случае программирование некоторых модулей может оказаться бесполезно проделанной работой.

Все эти методы имеют еще различные разновидности в зависимости от того, в какой последовательности обходятся узлы (модули) древовидной структуры программы в процессе ее разработки [7.1]. Это можно делать, например, по слоям (разрабатывая все модули одного уровня, прежде чем переходить к следующему уровню).
При нисходящей разработке дерево можно обходить также в лексикографическом порядке (сверху-вниз, слева-направо). Возможны и другие варианты обхода дерева. Так, при конструктивной реализации для обхода дерева программы целесообразно следовать идеям Фуксмана, которые он использовал в предложенном им методе вертикального слоения [7.8]. Сущность такого обхода заключается в следующем. В рамках конструктивного подхода сначала реализуются только те модули, которые необходимы для самого простейшего варианта программы, которая может нормально выполняться только для весьма ограниченного множества наборов входных данных, но для таких данных эта задача будет решаться до конца. Вместо других модулей, на которые в такой программе имеются ссылки, в эту программу вставляются лишь их имитаторы, обеспечивающие, в основном, контроль за выходом за пределы этого частного случая. Затем к этой программе

добавляются реализации некоторых других модулей (в частности, вместо некоторых из имеющихся имитаторов), обеспечивающих нормальное выполнение для некоторых других наборов входных данных. И этот процесс продолжается поэтапно до полной реализации требуемой программы. Таким образом, обход дерева программы производится с целью кратчайшим путем реализовать тот или иной вариант (сначала самый простейший) нормально действующей программы. В связи с этим такая разновидность конструктивной реализации получила название метода целенаправленной конструктивной реализации. Достоинством этого метода является то, что уже на достаточно ранней стадии создается работающий вариант разрабатываемой программы. Психологически это играет роль допинга, резко повышающего эффективность разработчика. Поэтому этот метод является весьма привлекательным.

Рис. 7.3. Классификация методов разработки структуры программ.

Подводя итог сказанному, на рис. 7.3 представлена общая схема классификации рассмотренных методов разработки структуры программы.

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ ПОДХОД К ПРОЕКТИРОВАНИЮ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Принципиальное различие между структурным и объектно-ориентированным подходом заключается в способе декомпозиции системы. Объектно-ориентированный подход использует объектную декомпозицию, при этом статическая структура системы описывается в терминах объектов и связей между ними, а поведение системы описывается в терминах обмена сообщениями между объектами. Каждый объект системы обладает своим собственным поведением, моделирующим поведение объекта реального мира.

Понятие "объект" впервые было использовано около 30 лет назад в технических средствах, при попытках отойти от традиционной архитектуры фон Неймана.

По определению признанного авторитета в области объектно-ориентированных методов разработки программ Гради Буча "объектно-ориентированное программирование (ООП) — это методология программирования, которая основана на представлении программы в виде совокупности объектов, каждый из которых является реализацией определенного класса (типа особого вида), а классы образуют иерархию на принципах наследуемости".

Объектно-ориентированная методология (ООМ) так же, как и структурная методология, была создана с целью, дисциплинировать процесс разработки больших программных комплексов и тем самым снизить их сложность и стоимость.

ООМ преследует те же цели, что и структурная, но решает их с другой отправной точки и в большинстве случаев позволяет управлять более сложными проектами, чем структурная методология.

Как отмечалось выше, одним из принципов управления сложностью проекта является декомпозиция. Буч выделяет две разновидности декомпозиции: алгоритмическую (так он называет декомпозицию, поддерживаемую структурными методами) и объектно-ориентированную, отличие которых состоит, по его мнению, в следующем: "Разделение по алгоритмам концентрирует внимание на порядке происходящих событий, а разделение по объектам придает особое значение факторам, либо вызывающим действия, либо являющимся объектами приложения этих действий".

Другими словами, алгоритмическая декомпозиция учитывает в большей степени структуру взаимосвязей между частями сложной проблемы, а объектно-ориентированная декомпозиция уделяет больше внимания характеру взаимосвязей [1,4,7].

На практике рекомендуется применять обе разновидности декомпозиции: при создании крупных проектов целесообразно сначала применять объектно-ориентированный подход для создания общей иерархии объектов, отражающих сущность программируемой задачи, а затем использовать алгоритмическую декомпозицию на модули для упрощения разработки и сопровождения разрабатываемого программного комплекса.

Следует заметить, что определений понятия "объект" существует несколько. Дадим определение объекта, придерживаясь мнения Гради Буча: " Объект — осязаемая сущность, которая четко проявляет свое поведение" [1,7].

Объект характеризуется как совокупностью всех своих свойств (например, для животных — это наличие головы, ушей, глаз и т.д.) и их текущих значений (голова — большая, уши — длинные, глаза — желтые и т.д.), так и совокупностью допустимых для данного объекта действий (умение принимать пищу, стоять, сидеть, идти, бежать и т.д.).

Указанное объединение в едином объекте как "материальных" составных частей (голова, лапы, хвост), так и действий, манипулирующих этими частями (действие "бежать" – быстро перемещает лапы) называется инкапсуляцией. Объекты постоянно взаимодействуют, оказывая влияние на другие объекты, и, в свою очередь, подвергаясь влиянию с их стороны.

Другим принципом управления сложностью проекта является иерархическое упорядочение объектов (подзадач), получаемых в процессе декомпозиции.

И структурная, и ООМ преследуют цель построения иерархического дерева взаимосвязей между объектами (подзадачами). Но если структурная иерархия строится по простому принципу разделения целого на составные части, то при создании объектно-ориентированной иерархии (ОО-иерархии) принимается другой взгляд на тот же исходный объект и в иерархии непременно отражается наследование свойств родительских (вышележащих) типов объектов дочерними (нижележащими) типами объектов.

Родительские типы также называют просто родителями или предками, а дочерние — потомками. Если тип А непосредственно (без промежуточных типов) связан с нижележащим типом В, то тип А называется непосредственным предком (родителем) типа В, а тип В - непосредственным потомком типа А.

По Бучу "наследование — это такое отношение между объектами, когда один объект повторяет структуру и поведение другого ".

Принцип наследования действует в жизни повсеместно и повседневно. Млекопитающие и птицы наследуют признаки живых организмов, в отличие от растений, орел и воробей наследуют общее свойство птиц — умение летать. С другой стороны, львы, тигры, леопарды наследуют "структуру" и поведение, характерное для представителей отряда кошачьих и так далее [1, 4, 7, 13, 16].

Типы верхних уровней ОО - иерархии, как правило, не имеют конкретных экземпляров объектов. Не существует, например, конкретного живого организма (объекта), который бы сам по себе назывался "Млекопитающее" или "Птица". Такие типы называются абстрактными. Конкретные экземпляры объектов имеют, как правило, типы самых нижних уровней ОО - иерархии: "крокодил Гена" — конкретный экземпляр объекта типа "Крокодил", "кот Матроскин" — конкретный экземпляр объекта типа "Кошка домашняя" [1,4,7, 18, 19,20].

Еще одним основополагающим понятием ООП является полиморфизм. Полиморфизм представляет собой свойство различных объектов выполнять одно и то же действие по-своему. Например, действие "бежать" свойственно большинству животных. Однако каждое из них (лев, слон, крокодил, черепаха) выполняет это действие различным образом.

При традиционном (не объектно-ориентированном) подходе к программированию, животных перемещать будет программист, вызывая отдельную для конкретного животного и конкретного действия подпрограмму. При объектно-ориентированном подходе программист только указывает, какому объекту какое из присущих ему действий требуется выполнить, и (для рассматриваемого примера) однажды описанные объекты-животные сами будут себя передвигать характерным для них способом, используя входящие в его состав методы.

Таким образом, в данном случае действие "бежать" будет называться полиморфическим действием, а многообразие форм проявления этого действия— полиморфизмом.

С объектно-ориентированной архитектурой также тесно связаны объектно-ориентированные операционные системы. Однако наиболее значительный вклад в объектный подход был внесен объектно-ориентированными языками программирования: Simula, Smalltalk, C++, Object Pascal. На объектный подход оказали влияние также развивавшиеся достаточно независимо методы моделирования баз данных, в особенности подход "сущность-связь".

Концептуальной основой объектно-ориентированного подхода является объектная модель. Основными ее элементами являются [1,4,7]:

- абстрагирование (abstraction);

- инкапсуляция (encapsulation);

- модульность (modularity);

- иерархия (hierarchy).

Имеются еще три дополнительных элемента:

- типизация (typing);

- параллелизм (concurrency);

- устойчивость (persistence).

Абстрагирование — это выделение существенных характеристик некоторого объекта, которые отличают его от всех других видов объектов и, таким образом, четко определяют его концептуальные границы относительно дальнейшего рассмотрения и анализа. Абстрагирование концентрирует внимание на внешних особенностях объекта и позволяет отделить самые существенные особенности его поведения от деталей их реализации. Выбор правильного набора абстракций для заданной предметной области представляет собой главную задачу объектно-ориентированного проектирования.

Инкапсуляция - это процесс отделения друг от друга отдельных элементов объекта, определяющих его устройство и поведение. Инкапсуляция служит для того, чтобы изолировать интерфейс объекта, отражающий его внешнее поведение, от внутренней реализации объекта. Объектный подход предполагает, что собственные ресурсы, которыми могут манипулировать только методы самого класса, скрыты от внешней среды. Абстрагирование и инкапсуляция являются взаимодополняющими операциями: абстрагирование фокусирует внимание на внешних особенностях объекта, а инкапсуляция (или, иначе, ограничение доступа) не позволяет объектам-пользователям различать внутреннее устройство объекта.

Модульность — это свойство системы, связанное с возможностью ее декомпозиции на ряд внутренне связных, но слабо связанных между собой модулей. Инкапсуляция и модульность создают барьеры между абстракциями.

Иерархия — это ранжированная или упорядоченная система абстракций, располагающая их по уровням. Основными видами иерархических структур применительно к сложным системам являются структура классов (иерархия по номенклатуре) и структура объектов (иерархия по составу). Примерами иерархии классов являются простое и множественное наследование (один класс использует структурную или функциональную часть соответственно одного или нескольких других классов), а иерархии объектов — агрегация.

Типизация - это ограничение, накладываемое на класс объектов и препятствующее взаимозаменяемости различных классов (или сильно сужающее ее возможность). Типизация позволяет защититься от использования объектов одного класса вместо другого или, по крайней мере, управлять таким использованием.

Параллелизм — свойство объектов находиться в активном или пассивном состоянии и различать активные и пассивные объекты между собой.

Устойчивость — свойство объекта существовать во времени (вне зависимости от процесса, породившего данный объект) и/или в пространстве (при перемещении объекта из адресного пространства, в котором он был создан).

Основные понятия объектно-ориентированного подхода — объект и класс.

Объект определяется как осязаемая реальность (tangible entity) — предмет или явление, обладающая четко определяемым поведением. Объект обладает состоянием, поведением и индивидуальностью; структура и поведение схожих объектов определяют общий для них класс. Термины "экземпляр класса" и "объект" являются эквивалентными. Состояние объекта характеризуется перечнем всех возможных (статических) свойств данного объекта и текущими значениями (динамическими) каждого из этих свойств. Поведение характеризует воздействие объекта на другие объекты и наоборот относительно изменения состояния этих объектов и передачи сообщений. Иначе говоря, поведение объекта полностью определяется его действиями. Индивидуальность — это свойства объекта, отличающие его от всех других объектов.

Определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию называется операцией. Как правило, в объектных и объектно-ориентированных языках операции, выполняемые над данным объектом, называются методами и являются составной частью определения класса.

Класс — это множество объектов, связанных общностью структуры и поведения. Любой объект является экземпляром класса. Определение классов и объектов - одна из самых сложных задач объектно-ориентированного проектирования.

Следующую группу важных понятий объектного подхода составляют наследование и полиморфизм. Понятие полиморфизма может быть интерпретирован как способность класса принадлежать более чем одному типу. Наследование означает построение новых классов на основе существующих с возможностью добавления или переопределения данных и методов.

Объектно-ориентированная система изначально строится с учетом ее эволюции. Наследование и полиморфизм обеспечивают возможность определения новой функциональности классов с помощью создания производных классов — потомков базовых классов. Потомки наследуют характеристики родительских классов без

изменения их первоначального описания и добавляют при необходимости собственные структуры данных и методы. Определение производных классов, при котором задаются только различия или уточнения, в огромной степени экономит время и усилия при производстве и использовании спецификаций и программного кода.

Важным качеством объектного подхода является согласованность моделей деятельности организации и моделей проектируемой системы от стадии формирования требований до стадии реализации. Требование согласованности моделей выполняется благодаря возможности применения абстрагирования, модульности, полиморфизма на всех стадиях разработки. Модели ранних стадий могут быть непосредственно подвергнуты сравнению с моделями реализации. По объектным моделям может быть прослежено отображение реальных сущностей моделируемой предметной области (организации) в объекты и классы информационной системы.

Первоначально OLE была задумана как технология интеграции программных продуктов, входящих в комплект Microsoft Office. Предшественницей OLE является реализованная в Windows технология динамического обмена данными DDE (Dynamic Data Exchange), до сих пор широко применяемая в данной среде. Однако многие разработчики не без оснований считают, что DDE трудно использовать, поскольку это технология низкого уровня. По существу, DDE представляет собой модель взаимодействия процессов - протокол, с помощью которого приложение может организовать канал обмена данными с DDE-сервером, находящимся на той же машине. DDE - это асинхронный протокол. Иными словами, после установления связи вызывающая сторона передает запрос и ожидает возврата результатов. Такой механизм более сложен, чем синхронный вызов функции, так как нужно учитывать вероятность нарушения связи, тайм- ауты и другие ошибки, которые приложение должно распознавать и исправлять. Низкая популярность DDE вынуждала Microsoft искать различные способы его усовершенствования. Для упрощения наиболее сложных аспектов протокола была предложена спецификация DDEML, но этого оказалось недостаточно.

Несмотря на различия между низкоуровневой технологией системных объектов и средствами интеграции компонентов высокого уровня, Microsoft попыталась предоставить разработчикам объединенное решение. В качестве технологии более высокого уровня была реализована OLE 1.0 OLE 1 (Object Linking and Embedding — связывание и внедрение объектов). Она расширила возможности протокола DDE и, используя его как базовый механизм коммуникаций, позволила активизировать встроенный объект в документе, т. е. получить составной документ. Таким образом, OLE 1.0 унаследовала многие проблемы асинхронного протокола. Эта технология имела множество недостатков, а ее компоновка была слишком сложна для пользователей среднего уровня. Кроме того, установленные связи легко нарушались, например, в результате изменения маршрута доступа к файлу связанного объекта.

Первое воплощение OLE — OLE 1 — представляло собой механизм создания и работы с составными документами (compound documents). С точки зрения пользователя, составной документ выглядит единым набором информации, но фактически содержит элементы, созданные двумя или несколькими разными приложениями. С помощью OLE 1 пользователь мог, например, объединить электронную таблицу, созданную Microsoft Excel, с текстовым документом “производства” Microsoft Word. Идея состояла в том, чтобы документо-ориентированная (document-centric) модель работы с компьютером позволила бы пользователю больше думать об информации и меньше — о приложениях, ее обрабатывающих. Как следует из слов “связывание и внедрение”, составные документы

можно создать, либо связав два разных документа, либо полностью внедрив один документ в другой.

OLE 1, как и большинство первых версий программных продуктов, была несовершенна. Архитекторам следующей версии предстояло улучшить первоначальный проект. Вскоре они поняли, что составные документы — лишь частный случай более общей проблемы: как разные программные компоненты должны предоставлять друг другу сервисы? Для решения этой проблемы архитекторы OLE создали группу технологий, область применения которых гораздо шире составных документов. Основу OLE 2 составляет важнейшая из этих технологий — Модель многокомпонентных объектов (Component Object Model — COM). Новая версия OLE не только обеспечивает поддержку составных документов лучше, чем первая, но и несомненно идет куда дальше простого объединения документов, созданных в разных приложениях. OLE 2 позволяет по-новому взглянуть на взаимодействие любых типов программ.

Новые возможности многим обязаны COM, предоставившей общую парадигму взаимодействия программ любых типов: библиотек, приложений, системного программного обеспечения и др. Вот почему подход, предложенный COM, можно использовать при реализации практически любой программной технологии, и его применение дает немало существенных преимуществ.

Благодаря этим преимуществам, COM скоро стал частью технологий, не имеющих никакого отношения к составным документам. Однако в Microsoft хотели сохранить общее имя для всей группы технологий, в основе которых лежит COM. Компания решила сократить название Object Linking and Embedding до OLE — эта комбинация более не рассматривалась как аббревиатура — и опустить номер версии.

В начале 1996 года Microsoft ввела в оборот новый термин — ActiveX. Сначала он относился к технологиям, связанным с Интернетом, и приложениям, выросшим из него, вроде WWW (World Wide Web). Поскольку большинство разработок Microsoft в данной области было основано на COM, то и ActiveX была непосредственно связана с OLE. Однако очень скоро новый термин стал захватывать территории, традиционно принадлежавшие OLE, и вот теперь все вернулось на круги своя: OLE, как встарь, обозначает только технологию создания составных документов связыванием и внедрением, а разнообразные технологии на основе COM, ранее объединенные под именем OLE, собраны под знаменем ActiveX. А некоторые технологии, название которых содержало слово "OLE" даже перекрестили: теперь это технологии ActiveX. Новые технологии на основе COM, которым раньше полагался ярлык "OLE", теперь частенько получают пометку "ActiveX".

Динамический обмен данными (DDE)

DDE — давний и прижившийся протокол обмена данными между разными приложениями, появившийся еще на заре эры Windows. С тех пор на его базе был создан интерфейс OLE, а в 32-разрядном API Windows появились и другие методы межпрограммного взаимодействия. Но ниша, занимаемая DDE, осталась неизменной — это оперативная передача и синхронизация данных в приложениях.

Приложения, использующие DDE, разделяются на две категории — клиенты и серверы (не путать с одноименной архитектурой СУБД). Оба участника процесса осуществляют контакты (conversations) по определенным темам (topic), при этом в рамках темы

производится обмен элементами данных (items). Устанавливает контакт клиент, который посылает запрос, содержащий имена контакта и темы. После установления контакта всякое изменение элемента данных на сервере передается данным клиента. Подробно функции DDE описаны в [4].

Первоначально программирование DDE было чрезвычайно сложным делом — оно требовало взаимосвязанной обработки более чем десяти сообщений Windows. В версии Windows 3.1 появилась библиотека DDEML, которая перевела управление DDE на уровень вызова процедур. Разработчики подсистемы DDE в Delphi, верные идеологии создания VCL, свели интерфейс этого протокола к четырем компонентам — двум для сервера и двум для клиента.

История OLE

OLE позволяет передавать часть работы от одной программы редактирования к другой и возвращать результаты назад. Например, установленная на персональном компьютере издательская система может послать некий текст на обработку в текстовый редактор, либо некоторое изображение в редактор изображений с помощью OLE-технологии.

Основное преимущество использования OLE (кроме уменьшения размера файла) в том, что она позволяет создать главный файл, картотеку функций, к которой обращается программа. Этот файл может оперировать данными из исходной программы, которые после обработки возвращаются в исходный документ.

OLE используется при обработке составных документов (англ. compound documents), может быть использована при передаче данных между различными несвязанными между собой системами посредством интерфейса переноса (англ. drag-and-drop), а также при выполнении операций с буфером обмена. Идея внедрения широко используется при работе с мультимедийным содержанием на веб-страницах (пример — Веб-ТВ), где используется передача изображение звука, видео, анимации в страницах HTML (язык гипертекстовой разметки) либо в других файлах, также использующих текстовую разметку (например, XML и SGML). Однако, технология OLE использует архитектуру «толстого клиента», то есть сетевой ПК с избыточными вычислительными ресурсами. Это означает, что тип файла либо программа, которую пытаются внедрить, должна присутствовать на машине клиента. Например, если OLE оперирует таблицами Microsoft Excel, то программа Excel должна быть инсталлирована на машине пользователя.

OLE 1.*

OLE 1.0 был выпущен в 1990 году на основе технологии DDE (Dynamic Data Exchange), использовавшейся в более ранних версиях операционной системы Microsoft Windows. В то время как технология DDE была сильно ограничена в количестве и методах передачи данных между двумя работающими программами, OLE имел возможность оперировать активными соединениями между двумя документами либо даже внедрить документ одного типа в документ другого типа.

OLE сервера и клиенты взаимодействуют с системными библиотеками при помощи таблиц виртуальных функций (англ. virtual function tables, VTBL). Эти таблицы содержат указатели на функции, которые системная библиотека может использовать для взаимодействия с сервером или клиентом. Библиотеки OLESVR.DLL (на сервере) и OLECLI.DLL (на клиенте) первоначально были разработаны для взаимодействия между

собой с помощью сообщения WM_DDE_EXECUTE, разработанного операционной системой.

OLE 1.1 позднее развился в архитектуру COM (component object model) для работы с компонентами программного обеспечения. Позднее архитектура COM была преобразована и стала называться DCOM.

Когда объект OLE помещен в буфер обмена информацией, он сохраняется в оригинальных форматах Windows (таких как bitmap или metafile), а также сохраняется в своём собственном формате. Собственный формат позволяет поддерживающей OLE программе внедрить порцию другого документа, скопированного в буфер, и сохранить её в документе пользователя.

OLE 2.0

Следующим эволюционным шагом стал OLE 2.0, сохранивший те же цели и задачи, что и предыдущая версия. Но OLE 2.0 стал надстройкой над архитектурой COM вместо использования VTBL. Новыми особенностями стали автоматизация технологии drag-and-drop, in-place activation и structured storage.

ActiveX

В 1996 году Microsoft переименовала технологию OLE 2.0 в ActiveX. Были представлены элементы управления ActiveX, ActiveX документы и технология Active Scripting. Эта версия OLE в основном используется веб-дизайнерами для вставки в страницы мультимедийных данных.

Верификация программного обеспечения

1. Лекция: Место верификации среди процессов разработки программного обеспечения: версия для печати и PDA
Лекция посвящена рассмотрению различных видов жизненного цикла разработки программного обеспечения и современных технологий разработки. Показано место процесса верификации в жизненном цикле, определена его цель и задачи. Рассматриваются различные типы процессов верификации, определяется разница между тестированием, верификацией и валидацией. Цель данной лекции: дать представление о процессе верификации как о четко определенном виде деятельности в рамках жизненного цикла разработки программной системы, определить современные подходы к верификации

1.1. Понятие верификации

Верификация - это процесс определения, выполняют ли программные средства и их компоненты требования, наложенные на них в последовательных этапах жизненного цикла разрабатываемой программной системы.

Основная цель верификации состоит в подтверждении того, что программное обеспечение соответствует требованиям. Дополнительной целью является выявление и регистрация дефектов и ошибок, которые внесены во время разработки или модификации программы.

Верификация является неотъемлемой частью работ при коллективной разработке программных систем. При этом в задачи верификации включается контроль результатов

одних разработчиков при передаче их в качестве исходных данных другим разработчикам.

Для повышения эффективности использования человеческих ресурсов при разработке верификация должна быть тесно интегрирована с процессами проектирования, разработки и сопровождения программной системы.

Заранее разграничим понятия верификации и отладки. Оба этих процесса направлены на уменьшение ошибок в конечном программном продукте, однако отладка - процесс, направленный на локализацию и устранение ошибок в системе, а верификация - процесс, направленный на демонстрацию наличия ошибок и условий их возникновения.

Кроме того, верификация, в отличие от отладки - контролируемый и управляемый процесс. Верификация включает в себя анализ причин возникновения ошибок и последствий, которые вызовет их исправление, планирование процессов поиска ошибок и их исправления, оценку полученных результатов. Все это позволяет говорить о верификации как о процессе обеспечения заранее заданного уровня качества создаваемой программной системы.

1.2. Жизненный цикл разработки программного обеспечения

Коллективная разработка, в отличие от индивидуальной, требует четкого планирования работ и их распределения во время создания программной системы. Один из способов организации работ состоит в разбиении процесса разработки на отдельные последовательные стадии, после полного прохождения которых получается конечный продукт или его часть. Такие стадии называют жизненным циклом разработки программной системы. Как правило, жизненный цикл начинается с формирования общего представления о разрабатываемой системе и его формализации в виде требований верхнего уровня. Завершается жизненный цикл разработки вводом системы в эксплуатацию. Однако, нужно понимать, что разработка - только один из процессов, связанных с программной системой, которая также имеет свой жизненный цикл. В отличие от жизненного цикла разработки системы, жизненный цикл самой системы заканчивается выводом ее из эксплуатации и прекращением ее использования.

Жизненный цикл программного обеспечения - совокупность итерационных процедур, связанных с последовательным изменением состояния программного обеспечения от формирования исходных требований к нему до окончания его эксплуатации конечным пользователем.

В контексте данного курса практически не будут затрагиваться такие этапы жизненного цикла, как системная интеграция и сопровождение. Для целей курса достаточно ограничиться упрощенным представлением, что после реализации кода и доказательства его соответствия требованиям разработка ПО завершается.

1.3. Модели жизненного цикла

Любой этап жизненного цикла имеет четко определенные критерии начала и окончания. Состав этапов жизненного цикла, а также критерии, в конечном итоге определяющие последовательность этапов жизненного цикла, определяется коллективом разработчиков и/или заказчиком. В настоящее время существует несколько основных моделей жизненного цикла, которые могут быть адаптированы под реальную разработку.

1.3.1. Каскадный жизненный цикл

Каскадный жизненный цикл (иногда называемый водопадным) основан на постепенном увеличении степени детализации описания всей разрабатываемой системы. Каждое повышение степени детализации определяет переход к следующему состоянию разработки (Рис. 1.1).

увеличить изображение
Рис. 1.1. Каскадная модель жизненного цикла

На первом этапе составляется концептуальная структура системы, описываются общие принципы ее построения, правила взаимодействия с окружающим миром, - определяются системные требования.

На втором этапе по системным требованиям составляются требования к программному обеспечению - здесь основное внимание уделяется функциональности программной компоненты, программным интерфейсам. Естественно, все программные комплексы выполняются на какой-либо аппаратной платформе. Если в ходе проекта требуется также разработка аппаратной компоненты, параллельно с требованиями к программному обеспечению идет подготовка требований к аппаратному обеспечению.

На третьем этапе на основе требований к программному обеспечению составляется детальная спецификация архитектуры системы - описываются разбиение системы по конкретным модулям, интерфейсы между ними, заголовки отдельных функций и т.п.

На четвертом этапе пишется программный код, соответствующий детальной спецификации, на пятом этапе выполняется тестирование - проверка соответствия программного кода требованиям, определенным на предыдущих этапах.

Особенность каскадного жизненного цикла состоит в том, что переход к следующему этапу происходит только тогда, когда полностью завершены все работы предыдущего этапа. То есть сначала полностью готовятся все требования к системе, затем по ним полностью готовятся все требования к программному обеспечению, полностью разрабатывается архитектура системы и так далее до тестирования.

Естественно, что в случае достаточно больших систем такой подход себя не оправдывает. Работа на каждом этапе занимает значительное время, а внесение изменений в первичные документы либо невозможно, либо вызывает лавинообразное изменения на всех других этапах.

Как правило, используется модификация каскадной модели, допускающая возврат на любой из ранее выполненных этапов. При этом фактически возникает дополнительная процедура принятия решения.

Действительно если тесты обнаружили несоответствие реализации требованиям, то причина может крыться: (а) в неправильном тесте, (б) в ошибке кодирования (реализации), (в) в неверной архитектуре системы, (г) в некорректности требований к программному обеспечению и т.д. Все эти случаи требуют анализа, для того чтобы принять решение о том, на какой этап жизненного цикла надо возвратиться для

устранения обнаруженного несоответствия.

1.3.2. V-образный жизненный цикл

В качестве своеобразной "работы над ошибками" классической каскадной модели стала применяться модель жизненного цикла, содержащая процессы двух видов - основные процессы разработки, аналогичные процессам каскадной модели, и процессы верификации, представляющие собой цепь обратной связи по отношению к основным процессам (Рис. 1.2).

Рис. 1.2. V-образный жизненный цикл

Таким образом, в конце каждого этапа жизненного цикла разработки, а зачастую и в процессе выполнения этапа, осуществляется проверка взаимной корректности требований различных уровней. Данная модель позволяет более оперативно проверять корректность разработки, однако, как и в каскадной модели, предполагается, что на каждом этапе разрабатываются документы, описывающие поведение всей системы в целом.

1.3.3. Спиральный жизненный цикл

Оба рассмотренных типа жизненных циклов предполагают, что заранее известны все требования пользователей или, по крайней мере, предполагаемые пользователи системы настолько квалифицированы, что могут высказывать свои требования к будущей системе, не видя ее перед глазами.

Естественно, такая картина достаточно утопична, поэтому постепенно появилось решение, исправляющее основной недостаток V-образного жизненного цикла - предположение о том, что на каждом этапе разрабатывается очередное полное описание системы. Этим решением стала спиральная модель жизненного цикла (Рис. 1.3).

Рис. 1.3. Спиральный жизненный цикл

В спиральной модели разработка системы происходит повторяющимися этапами - витками спирали. Каждый виток спирали - один каскадный или V-образный жизненный цикл. В конце каждого витка получается законченная версия системы, реализующая некоторый набор функций. Затем она предъявляется пользователю, на следующий виток переносится вся документация, разработанная на предыдущем витке, и процесс повторяется.

Таким образом, система разрабатывается постепенно, проходя постоянные согласования с заказчиком. На каждом витке спирали функциональность системы расширяется, постепенно дорастая до полной.

1.3.4. Экстремальное программирование

Реалии последних лет показали, что для систем, требования к которым изменяются достаточно часто, необходимо еще больше уменьшить длительность витка спирального жизненного цикла. В связи с этим сейчас стали весьма популярными быстрые жизненные

циклы разработки, например, жизненный цикл в методологии eXtreme Programming (XP).

Основная идея жизненного цикла экстремального подхода - максимальное укорачивание длительности одного этапа жизненного цикла и тесное взаимодействие с заказчиком. По сути, на каждом этапе происходит реализация и тестирование одной функции системы, после завершения которых система сразу передается заказчику на проверку или эксплуатацию.

Основная проблема данного подхода - интерфейсы между модулями, реализующими эту функцию. Если во всех предыдущих типах жизненного цикла интерфейсы достаточно четко определяются в самом начале разработки, поскольку заранее известны все модули, то при экстремальном подходе интерфейсы проектируются "на лету", вместе с разрабатываемыми модулями.

1.3.5. Сравнение различных типов жизненного цикла и вспомогательные процессы

Особенности рассмотренных выше типов жизненного цикла сведены в таблицу 1.1. Из нее можно видеть, что различные типы жизненных циклов применяются в зависимости от планируемой частоты внесения изменений в систему, сроков разработки и ее сложности. Жизненные циклы с более короткими фазами больше подходят для разработки систем, требования к которым еще не устоялись и вырабатываются во взаимодействии с заказчиком системы во время ее разработки.

Таблица 1.1. Сравнение различных типов жизненного цикла

Тип жизненного цикла	Длина цикла	Верификация и внесение изменений	Интеграция отдельных компонент системы
Каскадный	Все этапы разработки системы. Длинный	В конце разработки всей системы. Изменения вносятся редко	Четко определенные до начала кодирования интерфейсы
V-образный	Все этапы разработки системы. Длинный	В конце полной разработки каждого из этапов системы. Изменения вносятся со средней частотой	Редко изменяемые интерфейсы
Спиральный	Разработка одной версии системы. Средний	В конце разработки каждого из этапов версии системы. Изменения вносятся со средней частотой	Периодически изменяемые интерфейсы, редко меняемые в пределах версии
XP	Разработка одной истории. Короткий	В конце разработки каждой истории. Изменения вносятся очень часто	Часто изменяемые интерфейсы

В приведенном выше описании различных моделей жизненного цикла по сути затрагивался только один процесс - процесс разработки системы. На самом деле в любой модели жизненного цикла можно увидеть четыре вида процессов:

1. Основной процесс разработки
2. Процесс верификации
3. Процесс управления разработкой

4. Вспомогательные (поддерживающие) процессы

Процесс верификации - процесс, направленный на проверку корректности разрабатываемой системы и определения степени ее соответствия требованиям заказчика. Подробному рассмотрению этого процесса и посвящен данный учебный курс.

Процесс управления разработкой - отдельная дисциплина. На управление очень сильно влияет тип жизненного цикла основного процесса разработки. По сути, чем короче один этап жизненного цикла, тем активнее управление и тем больше задач стоит перед менеджером проекта. При классических схемах достаточно просто построить иерархическую пирамиду подчиненности, в которой каждый нижестоящий менеджер отвечает за разработку определенной части системы. В XP-подходе нет жесткого разделения системы на части, и менеджер должен охватывать все истории. При этом процесс управления активен на протяжении всего жизненного цикла основного процесса разработки.

Вспомогательные (поддерживающие) процессы обеспечивают своевременное создание всего, что может понадобиться разработчику или конечному пользователю. Сюда входит подготовка пользовательской документации, подготовка приемо-сдаточных тестов, управление конфигурациями и изменениями, взаимодействие с заказчиком и т.д. Вообще говоря, вспомогательные процессы могут существовать в течение всего жизненного цикла разработки, а могут быть своеобразными стыкующими звеньями между процессом разработки и процессом эксплуатации.

В конце данного курса особое внимание будет уделено наиболее значимым поддерживающим процессам - процессу управления конфигурациями и процессу обеспечения гарантии качества.

Основная цель процесса управления конфигурациями - обеспечение целостности всех данных, возникающих в процессе коллективной разработки. Под целостностью понимается, прежде всего, идентифицируемость, доступность этих данных в любой момент времени и недопущение несанкционированных изменений. Важным аспектом при этом становится процесс управления изменениями данных, т.е. планирование и утверждение любых изменений в проектную документацию или программный код, а также определение области влияния этих изменений.

Процесс гарантии качества обеспечивает проведение проверок, гарантирующих, что процесс разработки удовлетворяет набору определенных требований (стандартов), необходимых для выпуска качественной продукции. Фактически он проверяет, что все предусмотренные стандартами разработки процедуры выполняются и при выполнении соблюдаются декларированные для них правила.

Нужно особо отметить, что процесс гарантии качества не гарантирует разработку качественной программной системы. Он гарантирует только лишь, что процессы разработки построены и выполняются таким образом, чтобы не снижать качество продукции.

Требования, которые предъявляются к организации работы, необходимой для выпуска качественной продукции, оформлены в виде стандартов качества. Наиболее часто цитируемая и известная группа стандартов качества - серия стандартов ISO 9000. В дополнение к ним существует стандарт, содержащий требования к жизненному циклу

разработки ПО - ISO 12207.

В реальной практике сейчас наиболее широко применяется стандарт ISO 12207, в отечественных госструктурах используются стандарты серии ГОСТ 34.

Стандарты комплекса ГОСТ 34 на создание и развитие АС - обобщенные, но воспринимаемые как весьма жесткие по структуре ЖЦ и проектной документации.

Международный стандарт ISO/IEC 12207 на организацию жизненного цикла продуктов программного обеспечения (ПО) содержит общие рекомендации по организации жизненного цикла, не постулируя при этом его жесткой структуры.

1.4. Современные технологии разработки программного обеспечения

1.4.1. Microsoft Solutions Framework

Microsoft Solutions Framework (MSF) - это методология ведения проектов и разработки решений, базирующаяся на принципах работы над продуктами самой фирмы Microsoft и предназначенная для использования в организациях, нуждающихся в концептуальной схеме для построения современных решений [35].

Microsoft Solutions Framework является схемой для принятия решений по планированию и реализации новых технологий в организациях. MSF включает обучение, информацию, рекомендации и инструменты для идентификации и структуризации информационных потоков бизнес-процессов и всей информационной инфраструктуры новых технологий.

Microsoft Solutions Framework представляет собой хорошо сбалансированный набор методик организации процесса разработки, который может быть адаптирован под потребности практически любого коллектива разработчиков. MSF содержит не только рекомендации общего характера, но и предлагает адаптируемую модель коллектива разработчиков, определяющую взаимоотношения внутри коллектива, гибкую модель проектного планирования, основанного на управлении проектными группами, а также набор методик для оценки рисков.

В момент подготовки данного учебного курса последней версией MSF была 3.1, при этом существовали документы, относящиеся к версии 4.0 beta.

MSF состоит из двух моделей:

- модель проектной группы;
- модель процессов,

и трех дисциплин:

- управление проектами;
- управление рисками;
- управление подготовкой.

В MSF нет роли "менеджер проекта" и иерархии руководства, управление разработкой распределено между руководителями отдельных проектных групп внутри коллектива, выполняющих следующие задачи:

- Управление программой
- Разработка
- Тестирование
- Управление выпуском
- Удовлетворение потребителя
- Управление продуктом

Жизненный цикл процессов в MSF сочетает водопадную и спиральную модели разработки: проект реализуется поэтапно, с наличием соответствующих контрольных точек, а сама последовательность этапов может повторяться по спирали (Рис. 1.4).

Рис. 1.4. Жизненный цикл в MSF

При таком подходе от водопадной модели берется простота планирования, от классической спиральной - легкость модификаций. Благодаря промежуточным контрольным точкам и обратной спирали верификации облегчается взаимодействие с заказчиком.

При управлении проектом четко ставится цель, которую необходимо достичь в результате, и учитываются ограничения, накладываемые на проект. Все виды ограничений могут быть отнесены к одному из трех видов: ограничения ресурсов, ограничения времени и ограничения возможностей. Эти три вида ограничений и приоритетность задач по их преодолению образуют треугольник приоритетов в MSF (Рис. 1.5).

Рис. 1.5. Треугольник приоритетов в MSF

Треугольник приоритетов является основой для матрицы компромиссов - заранее утвержденных представлений о том, какие аспекты процесса разработки будут четко заданы, а какие будут согласовываться или приниматься как есть.

Microsoft выпустила среду разработки, в полной мере поддерживающей основные идеи MSF - Microsoft Visual Studio 2005 Team Edition. Это первый программный комплекс, представляющий собой не среду разработки для индивидуальных членов коллектива, а комплексное средство поддержки коллективной работы.

В состав Visual Studio Team Edition входит специальная редакция для тестировщиков - Team Edition for Software Testers (Рис. 1.6). Материалы семинарских занятий по данному курсу ориентированы на эту среду разработки, в то время как лекционные материалы ориентированы на изложение общих принципов и методик тестирования.

увеличить изображение
Рис. 1.6. Структура Microsoft Visual Studio 2005 Team System

1.4.2. Rational Unified Process

Rational Unified Process - это методология создания программного обеспечения,

оформленная в виде размещаемой на Web базы знаний, которая снабжена поисковой системой.

Продукт Rational Unified Process (RUP) разработан и поддерживается Rational Software. Он регулярно обновляется с целью учета передового опыта и улучшается за счет проверенных на практике результатов.

RUP обеспечивает строгий подход к распределению задач и ответственности внутри организации-разработчика. Его предназначение заключается в том, чтобы гарантировать создание точно в срок и в рамках установленного бюджета качественного ПО, отвечающего нуждам конечных пользователей.

RUP способствует повышению производительности коллективной разработки и предоставляет лучшее из накопленного опыта по созданию ПО, посредством руководств, шаблонов и наставлений по пользованию инструментальными средствами для всех критически важных работ, в течение жизненного цикла создания и сопровождения ПО. Обеспечивая каждому члену группы доступ к той же самой базе знаний, вне зависимости от того, разрабатывает ли он требования, проектирует, выполняет тестирование или управляет проектом - RUP гарантирует, что все члены группы используют общий язык моделирования и процесс, имеют согласованное видение того, как создавать ПО. В качестве языка моделирования в общей базе знаний используется Unified Modeling Language (UML), являющийся международным стандартом.

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.

RUP поддерживается инструментальными средствами, которые автоматизируют многие элементы процесса разработки. Они используются для создания и совершенствования различных промежуточных продуктов на различных этапах процесса создания ПО, например, при визуальном моделировании, программировании, тестировании и т.д.

RUP - это конфигурируемый процесс, поскольку вполне понятно, что невозможно создать единого руководства на все случаи разработки ПО. RUP пригоден как для маленьких групп разработчиков, так и для больших организаций, занимающихся созданием ПО. В основе RUP лежит простая и понятная архитектура процесса, которая обеспечивает общность для целого семейства процессов. Более того, RUP может конфигурироваться для учета различных ситуаций. В его состав входит Development Kit, который обеспечивает поддержку процесса конфигурирования под нужды конкретных организаций.

RUP описывает, как эффективно применять коммерчески обоснованные и практически опробованные подходы к разработке ПО для коллективов разработчиков, где каждый из членов получает преимущества от использования передового опыта в:

- итерационной разработке ПО;
- управлении требованиями;
- использовании компонентной архитектуры;
- визуальном моделировании;

- тестировании качества ПО;
- контроле за изменениями в ПО.

RUP организует работу над проектом в терминах последовательности действий (workflows), продуктов деятельности, исполнителей и других статических аспектов процесса, с одной стороны, и в терминах циклов, фаз, итераций и временных отметок завершения определенных этапов в создании ПО (milestones), т.е. в терминах динамических аспектов процесса - с другой. [29]

1.4.3. eXtreme Programming

Экстремальное программирование [36] - сравнительно молодая методология разработки программных систем, основанная на постепенном улучшении системы и разработки ее очень короткими итерациями. По своей сути экстремальное программирование (XP) - это одна из так называемых "гибких" методологий разработки ПО, которая представляет собой небольшой набор конкретных правил, позволяющих максимально эффективно выполнять требования современной теории управления программными проектами.

XP ориентирована на:

- командную работу с тесными связями внутри команды и с заказчиком;
- разработку наиболее простых работающих решений;
- гибкое адаптивное планирование;
- оперативную обратную связь (путем модульного и функционального тестирования).

Основными принципами XP является разработка небольшими итерациями на основании порции требований заказчика (т.н. пользовательских историй), написание функциональных тестов до написания программного кода, постоянное общение и постоянный рефакторинг кода.

Основными практиками XP являются:

- Планирование процесса
- Частые релизы
- Метафора системы
- Простая архитектура
- Тестирование
- Рефакторинг
- Парное программирование
- Коллективное владение кодом
- Частая интеграция
- 40-часовая рабочая неделя
- Стандарты кодирования
- Тесное взаимодействие с заказчиком

1.4.4. Сравнение технологий MSF, RUP и XP

Основные особенности MSF, RUP и XP сведены в таблицу 1.2 [1]. По ней можно судить, что Rational Unified Process является хорошо сбалансированным решением для средних по размерам коллективов разработчиков, работающих с применением продуктов и технологий компании Rational. Сопровождение разработки системы и самой системы

регламентируется методологией RUP, однако данная технология достаточно сильно ориентирована на внутрифирменные инструментальные средства.

Extreme Programming хорошо подходит для проектных групп малого размера и для небольших систем с часто изменяемыми требованиями. Основная проблема XP - сопровождаемость. В случае текучки кадров в коллективе разработчиков значительная часть проектной информации может быть утеряна из-за практически отсутствующей документации.

Таблица 1.2. Технологии MSF, RUP и XP

Технология	Оптимальная команда	Соответствие стандартам	Допустимые технологии и инструменты	Удобство модификации и сопровождения
Rational Unified Process	10 - 40 чел.	стандарты Rational	UML и продукты Rational	Удобно (RUP)
Microsoft Solutions Framework	3 - 20 чел.	адаптируема	любые	Удобно (MSF+MOF)
XP	2 - 10 чел.	стандарты отсутствуют	любые	Сложно (зависимость от конкретных участников коллектива)

Microsoft Solutions Framework является наиболее сбалансированной технологией, ориентированной на проектные группы малых и средних размеров. MSF не накладывает никаких ограничений на используемый инструментарий и содержит рекомендации весьма общего характера. Однако, эти рекомендации могут быть использованы для построения конкретного процесса, соответствующего потребностям коллектива разработчиков.

1.5. Ролевой состав коллектива разработчиков, взаимодействие между ролями в различных технологических процессах

Когда проектная команда включает более двух человек неизбежно встает вопрос о распределении ролей, прав и ответственности в команде. Конкретный набор ролей определяется многими факторами - количеством участников разработки и их личными предпочтениями, принятой методологией разработки, особенностями проекта и другими факторами. Практически в любом коллективе разработчиков можно выделить перечисленные ниже роли. Некоторые из них могут вовсе отсутствовать, при этом отдельные люди могут выполнять сразу несколько ролей, однако общий состав меняется мало.

Заказчик (заявитель). Эта роль принадлежит представителю организации, заказавшей разрабатываемую систему. Обычно заявитель ограничен в своем взаимодействии и общается только с менеджерами проекта и специалистом по сертификации или внедрению. Обычно заказчик имеет право изменять требования к продукту (только во взаимодействии с менеджерами), читать проектную и сертификационную документацию, затрагивающую нетехнические особенности разрабатываемой системы.

Менеджер проекта. Эта роль обеспечивает коммуникационный канал между заказчиком и проектной группой. Менеджер продукта управляет ожиданиями заказчика, разрабатывает

и поддерживает бизнес-контекст проекта. Его работа не связана напрямую с продажей, он сфокусирован на продукте, его задача - определить и обеспечить требования заказчика. Менеджер проекта имеет право изменять требования к продукту и финальную документацию на продукт.

Менеджер программы. Эта роль управляет коммуникациями и взаимоотношениями в проектной группе, является в некотором роде координатором, разрабатывает функциональные спецификации и управляет ими, ведет график проекта и отчитывается по состоянию проекта, инициирует принятие критичных для хода проекта решений.

Менеджер программы имеет право изменять функциональные спецификации верхнего уровня, план-график проекта, распределение ресурсов по задачам. Часто на практике роль менеджера проекта и менеджера программы выполняет один человек.

Разработчик. Разработчик принимает технические решения, которые могут быть реализованы и использованы, создает продукт, удовлетворяющий спецификациям и ожиданиям заказчика, консультирует другие роли в ходе проекта. Он участвует в обзорах, реализует возможности продукта, участвует в создании функциональных спецификаций, отслеживает и исправляет ошибки за приемлемое время. В контексте конкретного проекта роль разработчика может подразумевать, например, инсталляцию программного обеспечения, настройку продукта или услуги. Разработчик имеет доступ ко всей проектной документации, включая документацию по тестированию, имеет право на изменение программного кода системы в рамках своих служебных обязанностей.

Специалист по тестированию. Специалист по тестированию определяет стратегию тестирования, тест-требования и тест-планы для каждой из фаз проекта, выполняет тестирование системы, собирает и анализирует отчеты о прохождении тестирования. Тест-требования должны покрывать системные требования, функциональные спецификации, требования к надежности и нагрузочной способности, пользовательские интерфейсы и собственно программный код. В реальности роль специалиста по тестированию часто разбивается на две - разработчика тестов и тестировщика. Тестировщик выполняет все работы по выполнению тестов и сбору информации, разработчик тестов - всю остальные работы.

Специалист по контролю качества. Эта роль принадлежит члену проектной группы, который осуществляет взаимодействие с разработчиком, менеджером программы и специалистами по безопасности и сертификации с целью отслеживания целостной картины качества продукта, его соответствия стандартам и спецификациям, предусмотренным проектной документацией. Следует различать специалиста по тестированию и специалиста по контролю качества. Последний не является членом технического персонала проекта, ответственным за детали и технику работы. Контроль качества подразумевает в первую очередь контроль самих процессов разработки и проверку их соответствия определенным в стандартах качества критериям.

Специалист по сертификации. При разработке систем, к надежности которых предъявляются повышенные требования, перед вводом системы в эксплуатацию требуется подтверждение со стороны уполномоченного органа (обычно государственного) соответствия ее эксплуатационных характеристик заданным критериям. Такое соответствие определяется в ходе сертификации системы. Специалист по сертификации может либо быть представителем сертифицирующих органов, включенным в состав коллектива разработчиков, либо наоборот - представлять интересы разработчиков в сертифицирующем органе. Специалист по сертификации приводит документацию на

программную систему в соответствие требованиям сертифицирующего органа либо участвует в процессе создания документации с учетом этих требований. Также специалист по сертификации ответственен за все взаимодействие между коллективом разработчиков и сертифицирующим органом. Важной особенностью роли является независимость специалиста от проектной группы на всех этапах создания продукта. Взаимодействие специалиста с членами проектной группы ограничивается менеджерами по проекту и по программе.

Специалист по внедрению и сопровождению. Участвует в анализе особенностей площадки заказчика, на которой планируется проводить внедрение разрабатываемой системы, выполняет весь спектр работ по установке и настройке системы, проводит обучение пользователей.

Специалист по безопасности. Данный специалист ответственен за весь спектр вопросов безопасности создаваемого продукта. Его работа начинается с участия в написании требований к продукту и заканчивается финальной стадией сертификации продукта.

Инструктор. Эта роль отвечает за снижение затрат на дальнейшее сопровождение продукта, обеспечение максимальной эффективности работы пользователя. Важно, что речь идет о производительности пользователя, а не системы. Для обеспечения оптимальной продуктивности инструктор собирает статистику по производительности пользователей и создает решения для повышения производительности, в том числе с использованием различных аудиовизуальных средств. Инструктор принимает участие во всех обсуждениях пользовательского интерфейса и архитектуры продукта.

Технический писатель. Лицо, осуществляющее эту роль, несет обязанности по подготовке документации к разработанному продукту, финального описания функциональных возможностей. Также он участвует в написании сопроводительных документов (системы помощи, руководство пользователя).

1.6. Задачи и цели процесса верификации

Сначала рассмотрим цели верификации. Основная цель процесса - доказательство того, что результат разработки соответствует предъявленным к нему требованиям. Обычно процесс верификации проводится сверху вниз, начиная от общих требований, заданных в техническом задании и/или спецификации на всю информационную систему, и заканчивая детальными требованиями к программным модулям и их взаимодействию. В состав задач процесса входит последовательная проверка того, что в программной системе:

- общие требования к информационной системе, предназначенные для программной реализации, корректно переработаны в спецификацию требований высокого уровня к комплексу программ, удовлетворяющих исходным системным требованиям;
- требования высокого уровня правильно переработаны в архитектуру ПО и в спецификации требований к функциональным компонентам низкого уровня, которые удовлетворяют требованиям высокого уровня;
- спецификации требований к функциональным компонентам ПО, расположенным между компонентами высокого и низкого уровня, удовлетворяют требованиям более высокого уровня;
- архитектура ПО и требования к компонентам низкого уровня корректно переработаны в удовлетворяющие им исходные тексты программных и информационных модулей;
- исходные тексты программ и соответствующий им исполняемый код не содержат

ошибок.

Кроме того, верификации на соответствие спецификации требований на конкретный проект программного средства подлежат требования к технологическому обеспечению жизненного цикла ПО, а также требования к эксплуатационной и технологической документации.

Цели верификации ПО достигаются посредством последовательного выполнения комбинации из инспекций проектной документации и анализа их результатов, разработки тестовых планов тестирования и тест-требований, тестовых сценариев и процедур и последующего выполнения этих процедур. Тестовые сценарии предназначены для проверки внутренней непротиворечивости и полноты реализации требований. Выполнение тестовых процедур должно обеспечивать демонстрацию соответствия испытываемых программ исходным требованиям.

На выбор эффективных методов верификации и последовательность их применения в наибольшей степени влияют основные характеристики тестируемых объектов:

- класс комплекса программ, определяющийся глубиной связи его функционирования с реальным временем и случайными воздействиями из внешней среды, а также требования к качеству обработки информации и надежности функционирования;
- сложность или масштаб (объем, размеры) комплекса программ и его функциональных компонентов, являющихся конечными результатами разработки;
- преобладающие элементы в программах: осуществляющие вычисления сложных выражений и преобразования измеряемых величин или обрабатывающие логические и символьные данные для подготовки и отображения решений.

Определим некоторые понятия и определения, связанные с процессом тестирования как составной части верификации. Майерс дает следующие определения основных терминов [2].

Тестирование - процесс выполнения программы с целью обнаружения ошибки.

Тестовые данные - входы, которые используются для проверки системы.

Тестовая ситуация (test case) - входы для проверки системы и предполагаемые выходы в зависимости от входов, если система работает в соответствии со спецификацией требований.

Хорошая тестовая ситуация - та ситуация, которая обладает большой вероятностью обнаружения пока еще необнаруженной ошибки.

Удачный тест - тест, который обнаруживает пока еще необнаруженную ошибку.

Ошибка - действие программиста на этапе разработки, приводящее к тому, что в программном обеспечении содержится внутренний дефект, который в процессе работы программы может привести к неправильному результату.

Отказ - непредсказуемое поведение системы, приводящее к неожидаемому результату, которое могло быть вызвано дефектами, содержащимся в ней.

Таким образом, в процессе тестирования программного обеспечения, как правило, проверяют следующее:

- программное обеспечение соответствует требованиям на него;
- в ситуациях, не отраженных в требованиях, программное обеспечение ведет себя адекватно, то есть не происходит отказ системы;
- наличие типичных ошибок, которые делают программисты.

1.7. Тестирование, верификация и валидация - различия в понятиях

Несмотря на кажущуюся схожесть, термины "тестирование", "верификация" и "валидация" означают разные уровни проверки корректности работы программной системы. Дабы избежать дальнейшей путаницы, четко определим эти понятия (Рис 1.7).

Тестирование программного обеспечения - вид деятельности в процессе разработки, который связан с выполнением процедур, направленных на обнаружение (доказательство наличия) ошибок (несоответствий, неполноты, двусмысленностей и т.д.) в текущем определении разрабатываемой программной системы. Процесс тестирования относится в первую очередь к проверке корректности программной реализации системы, соответствия реализации требованиям, т.е. тестирование - это управляемое выполнение программы с целью обнаружения несоответствий ее поведения и требований.

Рис. 1.7. Тестирование, верификация и валидация

Верификация программного обеспечения - более общее понятие, чем тестирование. Целью верификации является достижение гарантии того, что верифицируемый объект (требования или программный код) соответствует требованиям, реализован без непредусмотренных функций и удовлетворяет проектным спецификациям и стандартам. Процесс верификации включает в себя инспекции, тестирование кода, анализ результатов тестирования, формирование и анализ отчетов о проблемах. Таким образом, принято считать, что процесс тестирования является составной частью процесса верификации, такое же допущение сделано и в данном учебном курсе.

Валидация программной системы - целью этого процесса является доказательство того, что в результате разработки системы мы достигли тех целей, которые планировали достичь благодаря ее использованию. Иными словами, валидация - это проверка соответствия системы ожиданиям заказчика. Вопросы, связанные с валидацией, выходят за рамки данного учебного курса и представляют собой отдельную интересную тему для изучения.

Если посмотреть на эти три процесса с точки зрения вопроса, на который они дают ответ, то тестирование отвечает на вопрос "Как это сделано?" или "Соответсвует ли поведение разработанной программы требованиям?", верификация - "Что сделано?" или ""Соответствует ли разработанная система требованиям?", а валидация - "Сделано ли то, что нужно?" или "Соответствует ли разработанная система ожиданиям заказчика?".

1.8. Документация, создаваемая на различных этапах жизненного цикла

Синхронизация всех этапов разработки происходит при помощи документов, которые создаются на каждом из этапов. Документация при этом создается и на прямом отрезке

жизненного цикла - при разработке программной системы, и на обратном - при ее верификации. Попробуем на примере V-образного жизненного цикла проследить, какие типы документов создаются на каждом из отрезков и какие взаимосвязи между ними существуют (Рис 1.8).

Результатом этапа разработки требований к системе являются сформулированные требования к системе: документ, описывающие общие принципы работы системы, ее взаимодействие с "окружающей средой" - пользователями системы, а также, программными и аппаратными средствами, обеспечивающими ее работу. Обычно параллельно с требованиями к системе создается план верификации и определяется стратегия верификации. Эти документы определяют общий подход к тому, как будет выполняться тестирование, какие методики будут применяться, какие аспекты будущей системы должны быть подвергнуты тщательной проверке. Еще одна задача, решаемая при помощи определения стратегии верификации - определение места различных верификационных процессов и их связей с процессами разработки.

Верификационный процесс, работающий с системными требованиями - это процесс валидации требований, сопоставления их реальным ожиданиям заказчика. Забегая вперед, скажем, что процесс валидации отличается от приемо-сдаточных испытаний, выполняемых при передаче готовой системы заказчику, хотя может считаться частью таких испытаний. Валидация является средством доказать не только корректность реализации системы с точки зрения заказчика, но и корректность принципов, положенных в основу ее разработки.

Рис. 1.8. Процессы и документы при разработке программных систем

Требования к системе являются основой для процесса разработки функциональных требований и архитектуры проекта. В ходе этого процесса разрабатываются общие требования к программному обеспечению системы, к функциям, которые она должна выполнять. Функциональные требования часто включают в себя определение моделей поведения системы в штатных и нештатных ситуациях, правила обработки данных и определения интерфейса пользователя. Текст требования, как правило, включает в себя слова "должна, должен" и имеет структуру вида "В случае, если значение температуры на датчике ABC достигает 30 и выше градусов Цельсия, система должна прекращать выдачу звукового сигнала". Функциональные требования являются основой для разработки архитектуры системы - описания ее структуры в терминах подсистем и структурных единиц языка, на котором производится реализация - областей, классов, модулей, функций и т.п.

На базе функциональных требований пишутся тест-требования - документы, содержащие определение ключевых точек, которые должны быть проверены для того, чтобы убедиться в корректности реализации функциональных требований. Часто тест-требования начинаются словами "Проверить, что" и содержат ссылки на соответствующие им функциональные требования. Примером тест-требований для приведенного выше функционального требования могут служить "Проверить, что в случае падения температуры на датчике ABC ниже 30 градусов Цельсия система выдает предупреждающий звуковой сигнал" и "Проверить, что в случае, когда значение температуры на датчике ABC выше 30 градусов Цельсия, система не выдает звуковой сигнал".

Одна из проблем, возникающих при написании тест-требований - принципиальная нетестируемость некоторых требований: например, требование "Интерфейс пользователя должен быть интуитивно понятным" невозможно проверить без четкого определения того, что является интуитивно понятным интерфейсом. Такие неконкретные функциональные требования обычно впоследствии видоизменяют.

Архитектурные особенности системы также могут служить источником для создания тест-требований, учитывающих особенности программной реализации системы. Примером такого требования является, например, "Проверить, что значение температуры на датчике ABC не выходит за 255".

На основе функциональных требований и архитектуры пишется программный код системы, для его проверки на основе тест-требований готовится тест-план - описание последовательности тестовых примеров, выполняющих проверку соответствия реализации системы требованиям. Каждый тестовый пример содержит конкретное описание значений, подаваемых на вход системы, значений, которые ожидаются на выходе, и описание сценария выполнения теста.

В зависимости от объекта тестирования тест-план может готовиться либо в виде программы на каком-либо языке программирования, либо в виде входного файла данных для инструментария, который выполняет тестируемую систему и передает ей значения, указанные в тест-плане, либо в виде инструкций для пользователя системы, описывающей необходимые действия, которые нужно выполнить для проверки различных функций системы.

В результате выполнения всех тестовых примеров собирается статистика об успешности прохождения тестирования - процент тестовых примеров, для которых реальные выходные значения совпали с ожидаемыми, так называемых пройденных тестов. Непройденные тесты являются исходными данными для анализа причин ошибок и последующего их исправления.

На этапе интеграции осуществляется сборка отдельных модулей системы в единое целое и выполнение тестовых примеров, проверяющих всю функциональность системы.

На последнем этапе осуществляется поставка готовой системы заказчику. Перед внедрением специалисты заказчика совместно с разработчиками проводят приемо-сдаточные испытания - выполняют проверку критичных для пользователя функций согласно заранее утвержденной программе испытаний. При успешном прохождении испытаний система передается заказчику, в противном случае отправляется на доработку.

1.9. Типы процессов тестирования и верификации и их место в различных моделях жизненного цикла

1.9.1. Модульное тестирование

Модульное тестирование предназначено для небольших модулей (процедур, классов и т.п.). В ходе тестирования одного модуля, размер которого редко превышает 1000 строк, возможно проверить большую часть логических ветвей алгоритма, типичные граничные условия и т.п. В качестве критерия полноты тестирования используется полнота покрытия тестами ключевых элементов модуля (покрыты все требования, все операторы, все ветви логических условий, все компоненты логических условий и т.п.) [3]. Модульное тестирование обычно выполняется для каждого независимого программного модуля и

является, пожалуй, наиболее распространенным видом тестирования, особенно для систем малых и средних размеров.

1.9.2. Интеграционное тестирование

При проверке каждого модуля системы по отдельности невозможно дать гарантии того, что эти модули будут работать вместе. Как правило, могут возникать (и возникают) проблемы, связанные с интеграцией модулей, с их взаимодействием. Для выявления таких проблем на ранних этапах разработки применяют интеграционное тестирование, т.е. тестирование модулей, объединенных в совместно работающие комплексы. Интеграция модулей и интеграционное тестирование, как правило, проводится в течение всего жизненного цикла разработки. Это позволяет облегчить процесс локализации проблем и дефектов. При откладывании интеграции на последние этапы жизненного цикла локализовать дефекты практически невозможно [3].

1.9.3. Системное тестирование

Логическим завершением интеграционного тестирования является системное тестирование. На этом этапе все модули системы объединены и работают вместе. Системное тестирование предназначено не для выявления проблем отдельных модулей - все они должны были быть устранены ранее, а для выявления проблем системы в целом, проблем использования системы в реальном окружении. Системные тесты учитывают такие аспекты системы, как устойчивость в работе, производительность, соответствие системы ожиданиям пользователя и т.п. Для определения полноты системного тестирования также используются иные способы - оценивается полнота выполнения всех возможных сценариев работы (как штатных, так и нештатных), полнота различных методов взаимодействия системы с внешним миром и т.п. [3]

1.9.4. Нагрузочное тестирование

Из системного тестирования часто выделяют как самостоятельную процедуру нагрузочное тестирование. В результате нагрузочного тестирования можно оценить, как будет изменяться производительность системы под различной нагрузкой. Данная информация позволяет принимать решения об области применения системы, ее масштабируемости. В результате нагрузочного тестирования зачастую пересматривается архитектура системы (если она не обеспечивает достаточного уровня производительности при заданной нагрузке) или отдельные архитектурные решения. С точки зрения заказчика системы нагрузочное тестирование является одним из способов проверки работы системы в условиях, приближеным к реальным.

1.9.5. Формальные инспекции

Формальная инспекция является одним из способов верификации документов и программного кода, создаваемых в процессе разработки программного обеспечения. В ходе формальной инспекции группой специалистов осуществляется независимая проверка соответствия инспектируемых документов исходным документам. Независимость проверки обеспечивается тем, что она осуществляется инспекторами, не участвовавшими в разработке инспектируемого документа.

1.10. Верификация сертифицируемого программного обеспечения

Дадим несколько определений, определяющих общую структуру процесса сертификации

программного обеспечения.

Сертификация ПО - процесс установления и официального признания того, что разработка ПО проводилась в соответствии с определенными требованиями. В процессе сертификации происходит взаимодействие Заявителя, Сертифицирующего органа и Наблюдательного органа.

Заявитель - организация, подающая заявку в соответствующий Сертифицирующий орган на получения сертификата (соответствия, качества, годности и т.п.) изделия.

Сертифицирующий орган - организация, рассматривающая заявку Заявителя о проведении Сертификации ПО и либо самостоятельно, либо путем формирования специальной комиссии производящая набор процедур направленных на проведение процесса Сертификации ПО Заявителя.

Наблюдательный орган - комиссия специалистов, наблюдающих за процессами разработки Заявителем сертифицируемой информационной системы и дающих заключение о соответствии данного процесса определенным требованиям, которое передается на рассмотрение в Сертифицирующий орган.

Сертификация может быть направлена на получение сертификата соответствия либо сертификата качества.

В первом случае результатом сертификации является признание соответствия процессов разработки определенным критериям, а функциональности системы - определенным требованиям. Примером таких требований могут служить руководящие документы Федеральной службы по техническому и экспортному контролю в области безопасности программных систем [4, 5].

Во втором случае результатом является признание соответствия процессов разработки определенным критериям, гарантирующим соответствующий уровень качества выпускаемой продукции и его пригодности для эксплуатации в определенных условиях. Примером таких стандартов может служить серия международных стандартов качества ISO 9000:2000 (ГОСТ Р ИСО 9000-2001) [6] или авиационные стандарты DO-178B [7], AS9100 [8], AS9006 [9].

Тестирование сертифицируемого программного обеспечения имеет две взаимодополняющие цели.

- Первая - продемонстрировать, что программное обеспечение удовлетворяет требованиям на него.
- Вторая - продемонстрировать с высоким уровнем доверительности, что ошибки, которые могут привести к неприемлемым отказным ситуациям, как они определены процессом, оценки отказобезопасности системы, выявлены в процессе тестирования.

Например, согласно требованиям стандарта DO-178B, для того, чтобы удовлетворить целям тестирования программного обеспечения, необходимо следующее:

- тесты, в первую очередь, должны основываться на требованиях к программному обеспечению;
- тесты должны разрабатываться для проверки правильности функционирования и

создания условий для выявления потенциальных ошибок.
- анализ полноты тестов, основанных на требованиях на программное обеспечение, должен определить, какие требования не протестированы.
- анализ полноты тестов, основанных на структуре программного кода, должен определить, какие структуры не исполнялись при тестировании.

Также в этом стандарте говорится о тестировании, основанном на требованиях. Установлено, что эта стратегия наиболее эффективна при выявлении ошибок. Руководящие указания для выбора тестовых примеров, основанных на требованиях, включают следующее:

- для достижения целей тестирования программного обеспечения должны быть проведены две категории тестов: тесты для нормальных ситуаций и тесты для ненормальных (не отраженных в требованиях, робастных) ситуаций;
- должны быть разработаны специальные тестовые примеры для требований на программное обеспечение и источников ошибок, присущих процессу разработки программного обеспечения.

Целью тестов для нормальных ситуаций является демонстрация способности программного обеспечения давать отклик на нормальные входы и условия в соответствии с требованиями.

Целью тестов для ненормальных ситуаций является демонстрация способности программного обеспечения адекватно реагировать на ненормальные входы и условия, иными словами, это не должно вызывать отказ системы.

Категории отказных ситуаций для системы устанавливаются путем определения опасности отказной ситуации, например, для самолета и тех, кто в нем находится. Любая ошибка в программном обеспечении может вызвать отказ, который внесет свой вклад в отказную ситуацию. Таким образом, уровень целостности программного обеспечения, необходимый для безопасной эксплуатации, связан с отказными ситуациями для системы.

Существует 5 уровней отказных ситуаций от несущественной до критически опасной. Согласно этим уровням вводится понятие уровня критичности программного обеспечения. От уровня критичности зависит состав документации, предоставляемой в сертифицирующий орган, а значит и глубина процессов разработки и верификации системы. Например, количество типов документов и объем работ по разработке системы, необходимых для сертификации по самому низкому уровню критичности DO-178B могут отличаться на один-два порядка от количества и объемов, необходимых для сертификации по самому высокому уровню. Конкретные требования определяет стандарт, по которому планируется вести сертификацию.

Верификация программного обеспечения 3. Лекция: Тестирование программного кода (методы+окружение): версия для печати и PDA
Лекция посвящена процессу тестирования программного кода. Определяются его задачи и цели, перечисляются основные методы и подходы к тестированию программного кода. Вводится понятие тестового окружения, рассматриваются его компоненты и различные виды окружения. Цель данной лекции: дать представление о процессе тестирования программного кода, его видах. Определить методы построения тестового окружения, необходимого для выполнения тестирования

3.1. Задачи и цели тестирования программного кода

Тестирование программного кода - процесс выполнения программного кода, направленный на выявление существующих в нем дефектов. Под дефектом здесь понимается участок программного кода, выполнение которого при определенных условиях приводит к неожиданному поведению системы (т.е. поведению, не соответствующему требованиям). Неожиданное поведение системы может приводить к сбоям в ее работе и отказам, в этом случае говорят о существенных дефектах программного кода. Некоторые дефекты вызывают незначительные проблемы, не нарушающие процесс функционирования системы, но несколько затрудняющие работу с ней. В этом случае говорят о средних или малозначительных дефектах.

Задача тестирования при таком подходе - определение условий, при которых проявляются дефекты системы, и протоколирование этих условий. В задачи тестирования обычно не входит выявление конкретных дефектных участков программного кода и никогда не входит исправление дефектов - это задача отладки, которая выполняется по результатам тестирования системы.

Цель применения процедуры тестирования программного кода - минимизация количества дефектов (в особенности существенных) в конечном продукте. Тестирование само по себе не может гарантировать полного отсутствия дефектов в программном коде системы. Однако, в сочетании с процессами верификации и валидации, направленными на устранение противоречивости и неполноты проектной документации (в частности - требований на систему), грамотно организованное тестирование дает гарантию того, что система удовлетворяет требованиям и ведет себя в соответствии с ними во всех предусмотренных ситуациях.

При разработке систем повышенной надежности, например, авиационных, гарантии надежности достигаются с помощью четкой организации процесса тестирования, определения его связи с остальными процессами жизненного цикла, введения количественных характеристик, позволяющих оценивать успешность тестирования. При этом чем выше требования к надежности системы (ее уровень критичности), тем более жесткие требования предъявляются.

Таким образом, в первую очередь мы рассматриваем не конкретные результаты тестирования конкретной системы, а общую организацию процесса тестирования, используя подход "хорошо организованный процесс дает качественный результат". Такой подход является общим для многих международных и отраслевых стандартов качества, о которых более подробно будет рассказано в конце данного курса. Качество разрабатываемой системы при таком подходе является следствием организованного процесса разработки и тестирования, а не самостоятельным неуправляемым результатом.

Поскольку современные программные системы имеют весьма значительные размеры, при тестировании их программного кода используется метод функциональной декомпозиции. Система разбивается на отдельные модули (классы, пространства имен и т.п.), имеющие определенную требованиями функциональность и интерфейсы. После этого по отдельности тестируется каждый модуль - выполняется модульное тестирование. Затем происходит сборка отдельных модулей в более крупные конфигурации - выполняется интеграционное тестирование, и наконец, тестируется система в целом - выполняется системное тестирование.

С точки зрения программного кода, модульное, интеграционное и системное тестирование имеют много общего, поэтому пока основное внимание будет уделено модульному

тестированию, особенности интеграционного и системного тестирования будут рассмотрены позднее.

В ходе модульного тестирования каждый модуль тестируется как на соответствие требованиям, так и на отсутствие проблемных участков программного кода, которые могут вызвать отказы и сбои в работе системы. Как правило, модули не работают вне системы - они принимают данные от других модулей, перерабатывают их и передают дальше. Для того, чтобы с одной стороны, изолировать модуль от системы и исключить влияние потенциальных ошибок системы, а с другой стороны - обеспечить модуль всеми необходимыми данными, используется тестовое окружение.

Задача тестового окружения - создать среду выполнения для модуля, эмулировать все внешние интерфейсы, к которым обращается модуль. Об особенностях организации тестового окружения пойдет речь далее в данной лекции.

Типичная процедура тестирования состоит в подготовке и выполнении тестовых примеров (также называемых просто тестами). Каждый тестовый пример проверяет одну "ситуацию" в поведении модуля и состоит из списка значений, передаваемых на вход модуля, описания запуска и выполнения переработки данных - тестового сценария и списка значений, которые ожидаются на выходе модуля в случае его корректного поведения. Тестовые сценарии составляются таким образом, чтобы исключить обращения к внутренним данным модуля, все взаимодействие должно происходить только через его внешние интерфейсы.

Выполнение тестового примера поддерживается тестовым окружением, которое включает в себя программную реализацию тестового сценария. Выполнение начинается с передачи модулю входных данных и запуска сценария. Реальные выходные данные, полученные от модуля в результате выполнения сценария, сохраняются и сравниваются с ожидаемыми. В случае их совпадения тест считается пройденным, в противном случае - не пройденным. Каждый не пройденный тест указывает на дефект либо в тестируемом модуле, либо в тестовом окружении, либо в описании теста.

Совокупность описаний тестовых примеров составляет тест-план - основной документ, определяющий процедуру тестирования программного модуля. Тест-план задает не только сами тестовые примеры, но и порядок их следования, который также может быть важен. Структура и особенности тест-планов, а также проблемы, связанные с порядком следования тестовых примеров, будут рассмотрены в следующих лекциях.

При тестировании часто бывает необходимо учитывать не только требования к системе, но и структуру программного кода тестируемого модуля. В этом случае тесты составляются таким образом, чтобы детектировать типичные ошибки программистов, вызванные неверной интерпретацией требований. Применяются проверки граничных условий, проверки классов эквивалентности. Отсутствие в системе возможностей, не заданных требованиями, гарантировано различными оценками покрытия программного кода тестами, т.е. оценками того, какой процент тех или иных языковых конструкций выполнен в результате выполнения всех тестовых примеров. Обо всем этом пойдет речь в завершение рассмотрения процесса тестирования программного кода.

3.2. Методы тестирования

3.2.1. Черный ящик

Основная идея в тестировании системы как черного ящика состоит в том, что все материалы, которые доступны тестировщику, - требования на систему, описывающие ее поведение, и сама система, работать с которой он может, только подавая на ее входы некоторые внешние воздействия и наблюдая на выходах некоторый результат. Все внутренние особенности реализации системы скрыты от тестировщика, - таким образом, система представляет собой "черный ящик", правильность поведения которого по отношению к требованиям и предстоит проверить.

С точки зрения программного кода черный ящик может представлять с собой набор классов (или модулей) с известными внешними интерфейсами, но недоступными исходными текстами.

Основная задача тестировщика для данного метода тестирования состоит в последовательной проверке соответствия поведения системы требованиям. Кроме того, тестировщик должен проверить работу системы в критических ситуациях - что происходит в случае подачи неверных входных значений. В идеальной ситуации все варианты критических ситуаций должны быть описаны в требованиях на систему и тестировщику остается только придумывать конкретные проверки этих требований. Однако в реальности в результате тестирования обычно выявляется два типа проблем системы.

1. Несоответствие поведения системы требованиям
2. Неадекватное поведение системы в ситуациях, не предусмотренных требованиями.

Отчеты об обоих типах проблем документируются и передаются разработчикам. При этом проблемы первого типа обычно вызывают изменение программного кода, гораздо реже - изменение требований. Изменение требований в данном случае может потребоваться из-за их противоречивости (несколько разных требований описывают разные модели поведения системы в одной и той же самой ситуации) или некорректности (требования не соответствуют действительности).

Проблемы второго типа однозначно требуют изменения требований ввиду их неполноты - в требованиях явно пропущена ситуация, приводящая к неадекватному поведению системы. При этом под неадекватным поведением может пониматься как полный крах системы, так и вообще любое поведение, не описанное в требованиях.

Тестирование черного ящика называют также тестированием по требованиям, т.к. это единственный источник информации для построения тест-плана.

3.2.2. Стеклянный (белый) ящик

При тестировании системы как стеклянного ящика тестировщик имеет доступ не только к требованиям к системе, ее входам и выходам, но и к ее внутренней структуре - видит ее программный код.

Доступность программного кода расширяет возможности тестировщика тем, что он может видеть соответствие требований участкам программного кода и определять тем самым, на весь ли программный код существуют требования. Программный код, для которого отсутствуют требования, называют кодом, не покрытым требованиями. Такой код является потенциальным источником неадекватного поведения системы. Кроме того, прозрачность системы позволяет углубить анализ ее участков, вызывающих проблемы - часто одна проблема нейтрализует другую, и они никогда не возникают одновременно.

3.2.3. Тестирование моделей

Тестирование моделей находится несколько в стороне от классических методов верификации программного обеспечения. Причина прежде всего в том, что объект тестирования - не сама система, а ее модель, спроектированная формальными средствами. Если оставить в стороне вопросы проверки корректности и применимости самой модели (считается, что ее корректность и соответствие исходной системе могут быть доказаны формальными средствами), то тестировщик получает в свое распоряжение достаточно мощный инструмент анализа общей целостности системы. На модели можно создать такие ситуации, которые невозможно создать в тестовой лаборатории для реальной системы. Работая с моделью программного кода системы, можно анализировать его свойства и такие параметры системы, как оптимальность алгоритмов или ее устойчивость.

Однако тестирование моделей не получило широкого распространения именно из-за трудностей, возникающих при разработке формального описания поведения системы. Одно из немногих исключений - системы связи, алгоритмический и математический аппарат которых достаточно хорошо проработан.

3.2.4. Анализ программного кода (инспекции)

Во многих ситуациях тестирование поведения системы в целом невозможно - отдельные участки программного кода могут никогда не выполняться, при этом они будут покрыты требованиями. Примером таких участков кода могут служить обработчики исключительных ситуаций. Если, например, два модуля передают друг другу числовые значения и функции проверки корректности значений работают в обоих модулях, то функция проверки модуля-приемника никогда не будет активизирована, т.к. все ошибочные значения будут отсечены еще в передатчике.

В этом случае выполняется ручной анализ программного кода на корректность, называемый также просмотрами или инспекциями кода. Если в результате инспекции выявляются проблемные участки, то информация об этом передается разработчикам для исправления наравне с результатами обычных тестов.

3.3. Тестовое окружение

Основной объем тестирования практически любой сложной системы обычно выполняется в автоматическом режиме. Кроме того, тестируемая система обычно разбивается на отдельные модули, каждый из которых тестируется вначале отдельно от других, затем в комплексе.

Это означает, что для выполнения тестирования необходимо создать некоторую среду, которая обеспечит запуск и выполнение тестируемого модуля, передаст ему входные данные, соберет реальные выходные данные, полученные в результате работы системы на заданных входных данных. После этого среда должна сравнить реальные выходные данные с ожидаемыми и на основании данного сравнения сделать вывод о соответствии поведения модуля заданному (Рис 3.1).

увеличить изображение

Рис. 3.1. Обобщенная схема среды тестирования

Тестовое окружение также может использоваться для отчуждения отдельных модулей системы от всей системы. Разделение модулей системы на ранних этапах тестирования позволяет более точно локализовать проблемы, возникающие в их программном коде. Для поддержки работы модуля в отрыве от системы тестовое окружение должно моделировать поведение всех модулей, к функциям или данным которых обращается тестируемый модуль.

Поскольку тестовое окружение само является программой (причем зачастую реализованной не на том языке программирования, на котором написана система), оно само должно быть протестировано. Целью тестирования тестового окружения является доказательство того, что тестовое окружение никаким образом не искажает выполнение тестируемого модуля и адекватно моделирует поведение системы.

3.3.1. Драйверы и заглушки

Тестовое окружение для программного кода на структурных языках программирования состоит из двух компонентов - драйвера, который обеспечивает запуск и выполнение тестируемого модуля, и заглушек, которые моделируют функции, вызываемые из данного модуля. Разработка тестового драйвера представляет собой отдельную задачу тестирования, сам драйвер должен быть протестирован, дабы исключить неверное тестирование. Драйвер и заглушки могут иметь различные уровни сложности, требуемый уровень сложности выбирается в зависимости от сложности тестируемого модуля и уровня тестирования. Так, драйвер может выполнять следующие функции:

1. Вызов тестируемого модуля
2. 1 + передача в тестируемый модуль входных значений и прием результатов
3. 2 + вывод выходных значений
4. 3 + протоколирование процесса тестирования и ключевых точек программы

Заглушки могут выполнять следующие функции:

1. Не производить никаких действий (такие заглушки нужны для корректной сборки тестируемого модуля)
2. Выводить сообщения о том, что заглушка была вызвана
3. 1 + выводить сообщения со значениями параметров, переданных в функцию
4. 2 + возвращать значение, заранее заданное во входных параметрах теста
5. 3 + выводить значение, заранее заданное во входных параметрах теста
6. 3 + принимать от тестируемого ПО значения и передавать их в драйвер [10].

Для тестирования программного кода, написанного на процедурном языке программирования, используются драйверы, представляющие собой программу с точкой входа (например, функцией main()), функциями запуска тестируемого модуля и функциями сбора результатов. Обычно драйвер имеет как минимум одну функцию - точку входа, которой передается управление при его вызове.

Функции-заглушки могут помещаться в тот же файл исходного кода, что и основной текст драйвера. Имена и параметры заглушек должны совпадать с именами и параметрами "заглушаемых" функций реальной системы. Это требование важно не столько с точки зрения корректной сборки системы (при сборке тестового драйвера и тестируемого ПО может использоваться приведение типов), сколько для того, чтобы максимально точно моделировать поведение реальной системы по передаче данных. Так, например, если в реальной системе присутствует функция вычисления квадратного корня

```
double sqrt(double value);
```

то, с точки зрения сборки системы, вместо типа double может использоваться и float, но снижение точности может вызвать непредсказуемые результаты в тестируемом модуле.

В качестве примера драйвера и заглушек рассмотрим реализацию стека на языке С, причем значения, помещаемые в стек, хранятся не в оперативной памяти, а помещаются в ППЗУ при помощи отдельного модуля, содержащего две функции - записи данных в ППЗУ по адресу и чтения данных по адресу.

Формат этих функций следующий:

```
void NV_Read(char *destination, long length, long offset);
void NV_Write(char *source, long length, long offset);
```

Здесь destination - адрес области памяти, в которую записывается значение, считанное из ППЗУ, source - адрес области памяти, из которой записывается значение в ППЗУ, length - длина записываемой области памяти, offset - смещение относительно начального адреса ППЗУ.

Реализация стека с использованием этих функций выглядит следующим образом:

```
long currentOffset;

void initStack()
{
  currentOffset=0;
}

void push(int value)
{
  NV_Write((int*)&value,sizeof(int),currentOffset);
  currentOffset+=sizeof(int);
}

int pop()
{
  int value;
  if (currentOffset>0)
  {
    NV_Read((int*)&value,sizeof(int),currentOffset;
    currentOffset-=sizeof(int);
  }
}
```

При выполнении этого кода на реальной системе происходит запись в ППЗУ, однако, если мы хотим протестировать только реализацию стека, изолировав ее от реализации модуля работы с ППЗУ, необходимо использовать заглушки вместо реальных функций. Для имитации работы ППЗУ можно выделить достаточно большой участок оперативной памяти, в которой и будет производиться запись данных, получаемых заглушкой.

Заглушки для функций могут выглядеть следующим образом:

```
char nvrom[1024];
```

```
void NV_Read(char *destination, long length, long offset)
{
  printf("NV_Read called\n");
  memcpy(destination, nvrom+offset, length);
}
void NV_Write(char *source, long length, long offset);
{
  printf("NV_Write called\n");
  memcpy(nvrom+offset, source, length);
}
```

Каждая из заглушек выводит трассировочное сообщение и перемещает переданное значение в память, эмулирующую ППЗУ (функция NV_Write), или возвращает по ссылке значение, которое хранится в памяти, эмулирующей ППЗУ (функция NV_Read).

Схема взаимодействия тестируемого ПО (функций работы со стеком) с реальным окружением (основной частью системы и модулем работы с ППЗУ) и тестовым окружением (драйвером и заглушками функций работы с ППЗУ) показана на Рис 3.2 и Рис 3.3.

увеличить изображение

Рис. 3.2. Схема взаимодействия частей реальной системы

Рис. 3.3. Схема взаимодействия тестового окружения и тестируемого ПО

3.3.2. Тестовые классы

Тестовое окружение для объектно-ориентированного ПО выполняет те же самые функции, что и для структурных программ (на процедурных языках). Однако, оно имеет некоторые особенности, связанные с применением наследования и инкапсуляции.

Если при тестировании структурных программ минимальным тестируемым объектом является функция, то в объектно-ориентированным ПО минимальным объектом является класс. При применении принципа инкапсуляции все внутренние данные класса и некоторая часть его методов недоступна извне. В этом случае тестировщик лишен возможности обращаться в своих тестах к данным класса и произвольным образом вызывать методы; единственное, что ему доступно - вызывать методы внешнего интерфейса класса.

Существует несколько подходов к тестированию классов, каждый из них накладывает свои ограничения на структуру драйвера и заглушек.

1. Драйвер создает один или больше объектов тестируемого класса, все обращения к объектам происходят только с использованием их внешнего интерфейса. Текст драйвера в этом случае представляет собой т.н. тестирующий класс, который содержит по одному методу для каждого тестового примера. Процесс тестирования заключается в последовательном вызове этих методов. Вместо заглушек в состав тестового окружения входит программный код реальной системы, соответственно, отсутствует изоляция тестируемого класса. Однако, именно такой подход к тестированию принят сейчас в большинстве методологий и сред разработки. Его

классическое название - unit testing (тестирование модулей), более подробно он будет рассматриваться позднее.
2. Аналогично предыдущему подходу, но для всех классов, которые использует тестируемый класс, создаются заглушки
3. Программный код тестируемого класса модифицируется таким образом, чтобы открыть доступ ко всем его свойствам и методам. Строение тестового окружения в этом случае полностью аналогично окружению для тестирования структурных программ.
4. Используются специальные средства доступа к закрытым данным и методам класса на уровне объектного или исполняемого кода - скрипты отладчика или accessors в Visual Studio.

Основное достоинство первых двух методов: при их использовании класс работает точно таким же образом, как в реальной системе. Однако в этом случае нельзя гарантировать, что в процессе тестирования будет выполнен весь программный код класса и не останется непротестированных методов.

Основной недостаток 3-го метода: после изменения исходных текстов тестируемого модуля нельзя дать гарантии того, что класс будет вести себя таким же образом, как и исходный. В частности это связано с тем, что изменение защиты данных класса влияет на наследование данных и методов другими классами.

Тестирование наследования - отдельная сложная задача в объектно-ориентированных системах. После того, как протестирован базовый класс, необходимо тестировать классы-потомки. Однако, для базового класса нельзя создавать заглушки, т.к. в этом случае можно пропустить возможные проблемы полиморфизма. Если класс-потомок использует методы базового класса для обработки собственных данных, необходимо убедиться в том, что эти методы работают.

Таким образом, иерархия классов может тестироваться сверху вниз, начиная от базового класса. Тестовое окружение при этом может меняться для каждой тестируемой конфигурации классов.

3.3.3. Генераторы сигналов (событийно-управляемый код)

Значительная часть сложных программ в настоящее время использует ту или иную форму межпроцессного взаимодействия. Это обусловлено естественной эволюцией подходов к проектированию программных систем, которая последовательно прошла следующие этапы [11].

1. Монолитные программы, содержащие в своем коде все необходимые для своей работы инструкции. Обмен данными внутри таких программ производится при помощи передачи параметров функций и использования глобальных переменных. При запуске таких программ образуется один процесс, который выполняет всю необходимую работу.
2. Модульные программы, которые состоят из отдельных программных модулей с четко определенными интерфейсами вызовов. Объединение модулей в программу может происходить либо на этапе сборки исполняемого файла (статическая сборка или static linking), либо на этапе выполнения программы (динамическая сборка или dynamic linking). Преимущество модульных программ заключается в достижении некоторого уровня универсальности - один модуль может быть заменен другим. Однако, модульная программа все равно представляет собой один процесс, а

данные, необходимые для решения задачи, передаются внутри процесса как параметры функций.

3. Программы, использующие межпроцессное взаимодействие. Такие программы образуют программный комплекс, предназначенный для решения общей задачи. Каждая запущенная программа образует один или более процессов. Каждый из процессов либо использует для решения задачи свои собственные данные и обменивается с другими процессами только результатом своей работы, либо работает с общей областью данных, разделяемых между разными процессами. Для решения особо сложных задач процессы могут быть запущены на разных физических компьютерах и взаимодействовать через сеть. Преимущество использования межпроцессного взаимодействия заключается в еще большей универсальности - взаимодействующие процессы могут быть заменены независимо друг от друга при сохранении интерфейса взаимодействия. Другое преимущество состоит в том, что вычислительная нагрузка распределяется между процессами. Это позволяет операционной системе управлять приоритетами выполнения отдельных частей программного комплекса, выделяя большее или меньшее количество ресурсов ресурсоемким процессам.

При выполнении многих процессов, решающих общую задачу, используются несколько типичных механизмов взаимодействия между ними, направленных на решение следующих задач:

- передача данных от одного процесса к другому;
- совместное использование одних и тех же данных несколькими процессами;
- извещения об изменении состояния процессов.

Во всех этих случаях типичная структура каждого процесса представляет собой конечный автомат с набором состояний и переходов между ними. Находясь в определенном состоянии, процесс выполняет обработку данных, при переходе между состояниями - пересылает данные другим процессам или принимает данные от них [12].

Для моделирования конечных автоматов используются stateflow [13] или SDL-диаграммы [13], акцент в которых делается соответственно на условиях перехода между состояниями и пересылаемыми данными.

Так, на Рис 3.4 показана схема процесса приема/передачи данных. Закругленными прямоугольниками указаны состояния процесса, тонкими стрелками - переходы между состояниями, большими стрелками - пересылаемые данные. Находясь в состоянии "Старт", процесс посылает во внешний мир (или процессу, с которым он обменивается данными) сообщение о своей готовности к началу сеанса передачи данных. После получения от второго процесса подтверждения о готовности начинается сеанс обмена данными. В случае поступления сообщения о конце данных происходит завершение сеанса и переход в стартовое состояние. В случае поступления неверных данных (например, неправильного формата или с неверной контрольной суммой) процесс переходит в состояние "Ошибка", выйти из которого возможно только завершением и перезапуском процесса.

Рис. 3.4. Пример конечного автомата процесса приема-передачи данных

Тестовое окружение для такого процесса также должно иметь структуру конечного автомата и пересылать данные в том же формате, что и тестируемый процесс. Целью тестирования в данном случае будет показать, что процесс обрабатывает принимаемые данные в соответствии с требованиями, форматы передаваемых данных корректны, а также что процесс во время своей работы действительно проходит все состояния конечного автомата, моделирующего его поведение.