ВВЕДЕНИЕ

Данное пособие составлено в соответствии с требованиями Государственного образовательного стандарта специальности «Прикладная информатика в экономике» и отражает следующие разделы дисциплины «Разработка и стандартизация программного обеспечения и информационных систем» согласно учебной программы:

- классификация программного обеспечения;
- жизненный цикл ПО;
- этапы проектирования ПО;
- структурный подход проектирования ПО;
- объектно-ориентированный подход проектирования ПО;
- разработка интерфейса системы;
- тестирование системы;
- управление разработкой ПО;
- обеспечение качества ПО;
- документирование разработки ПО;

Описываются процессы жизненного цикла программного обеспечения (ПО) согласно международному стандарту ISO/IEC 12207, а также модели и стадии разработки ПО. Особое внимание уделяется процессу проектирования, подробно рассмотрены надежность и качество программных средств, принципы организации и методики тестирования при испытании надежности программных средств.

Рассматриваются современные методы и средства программной инженерии: структурный и объектно-ориентированный подходы. Рассмотрены функции и компоненты CASE-средств и их практическое воплощение в программных продуктах.

Для успешного освоения материала пособия студенту необходимы знания основ программирования, архитектуры современных вычислительных сетей, а также современных информационных технологий и теории информационных систем и баз данных. Эти знания студент получает при изучении следующих дисциплин: «Информатика и программирование», «Программирование», «Информационные технологии», «Информационные системы», «Базы данных».

Знания и навыки, получаемые студентами при изучения дисциплины, необходимы для подготовки к изучению следующих дисциплин: «Проектирование баз данных», «Разработка информационных систем», а также для дисциплин, связанных с проектированием проблемно-ориентированных информационных систем.

1. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ЭВМ

Программное обеспечение (ПО) – это совокупность программ и сопровождающей их документации, позволяющую использовать вычислительную машину для решения задач.

Классификация программного обеспечения показана на рис. 1.

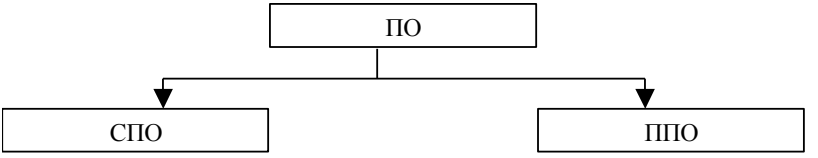


Рис. 1. Классификация ПО

Системное программное обеспечение (СПО) – это комплекс программ, предназначенных для управления работой персонального компьютера, распределение его ресурсов, поддержание диалога с пользователем, оказание ему помощи в разработке новых программ и выполнение работ связанных с обслуживанием компьютера.

Прикладное программное обеспечение (ППО) – это совокупность программ для решения прикладных задач, в определенной области (в промышленности, математике, бухгалтерии и т.д.).

По функциональному назначению системное ПО подразделяется на операционную систему (ОС), систему программирования (СП), системные обслуживающие программы (СОП), средства контроля и диагностики (СКД) (рис. 2).

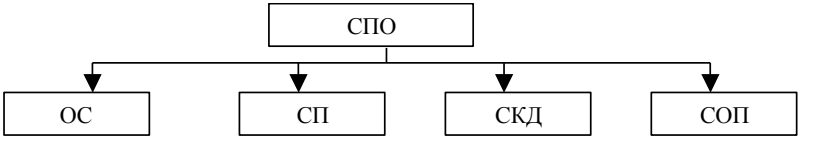


Рис. 2. Классификация СПО

Операционная система – комплекс управляющих программ, обеспечивающих функционирование вычислительной машины, включая планирование и управление ресурсами ЭВМ, решение задач (выполнение прикладных и обслуживающих программ) по запросам пользователей, управление вводом-выводом данных.

Система программирования – комплекс средств для разработки и отладки программ. В систему программирования включают языки про-

граммирования, трансляторы, различные обслуживающие программы для редактирования текстов и отладки программ.

Системные обслуживающие программы. Предназначены для выполнения типовых действий по подготовке носителей информации к записи на них данных, копирования, переименования и удаления файлов и т.п.

Средства контроля и диагностики устройств ЭВМ служат для проверки исправности отдельных устройств машины и локализации выявленных неисправностей.

Контрольные вопросы

1. Дайте определение термину программное обеспечение
2. На какие виды делится программное обеспечение ЭВМ.
3. Перечислите основные компоненты системного программного обеспечения и укажите их назначение.
4. Определите основные функции ОС.
5. Каковы функции прикладного программного обеспечения?
6. Как классифицируется прикладное программное обеспечение?
7. Укажите назначение и функции основных групп прикладного ПО.
8. Дайте определение пакету прикладных программ (ППП).
9. Чем прикладная программа отличается от ППП?

2. ПАКЕТЫ ПРИКЛАДНЫХ ПРОГРАММ

Определение пакетов прикладных программ (ППП). Классификация ППП. Составные части ППП. Модульный принцип формирования пакета. Функции отдельных модулей пакета.

Модель предметной области пакета. Статическая и динамическая модели предметной области.

Пакеты прикладных программ (ППП)– комплекс программ, предназначенных для решения задач определенного класса.

ППП должно обладать следующими свойствами:

1) пакет должен состоять из нескольких программных единиц;

2) пакет предназначен для решения определенного класса задач;

3) в пределах своего класса пакет обладает определенной универсальностью, т.е. позволяет решать все или почти все задачи этого класса;

4) в пакете предусмотрены средства управления, позволяющие выбирать конкретные возможности из числа предусмотренных в пакете, пакет допускает настройку на конкретные условия применения;

5) пакет разработан с учетом возможности его использования за пределами той организации, в которой он создан и удовлетворяет общим требованиям к ПИ;

6) документация и способы применения пакета ориентированы на пользователя, имеющего определенный уровень квалификации в той области знаний, к которой относятся решаемые пакетом задачи.

2.1. Классификация ППП

ППП можно классифицировать по функциональному признаку на универсальные, методо-ориентированные и проблемно-ориентированные. Схема классификации представлена на рис. 3.

ППП общего назначения – это универсальные программные продукты, предназначенные для автоматизации разработки и эксплуатации функциональных задач пользователя и информационных систем в целом. Типичными примерами этого класса являются пакеты интегрированной системы MS Office.

Рис. 3. Классификация СПО

Методо-ориентированные ППП отличаются тем, что в их алгоритмической основе реализован какой-либо экономико-математический метод.

Проблемно-ориентированные ППП предназначены для решения в конкретной функциональной области. Это могут быть комплексные ППП, которые автоматизируют все процессы предметной области (предприятия, организации) или применяемые для отдельных предметных областей (бухгалтерия, кадры). Кроме того выделяют комплексные ППП для непроизводственных областей (банковская деятельность, сфера услуг)

2.2. Составные части ППП

Класс задач, для решения которых предназначается пакет, называют предметной областью пакета. Для решения задач предметной области определяют некоторую структуру данных (входные, промежуточные, выходные). Эту структуру данных называют *информационной базой пакета*.

Для реализации функций пакета он должен воспринимать от пользователя управляющую информацию. Эта управляющая информация

представляется на формальном языке, который называется *входным языком пакета* (ПВЯ).

Программные модули пакета, реализующие алгоритмы задач, называют *обрабатывающими модулями*. Они выполняют преобразование данных, составляющих информационную базу.

Управляющие модули служат для преобразования задания пользователя в последовательность вызовов обрабатывающих модулей. *Обслуживающие модули* обеспечивают взаимодействие пакета с пользователем и управляющих модулей пакета с информационной базой и обрабатывающими модулями.

Совокупность обрабатывающих модулей называют *функциональной частью пакета*.

Совокупность управляющих и обрабатывающих модулей – *системная часть пакета*.

Взаимодействие составных частей пакета схематически показано на рис. 4.

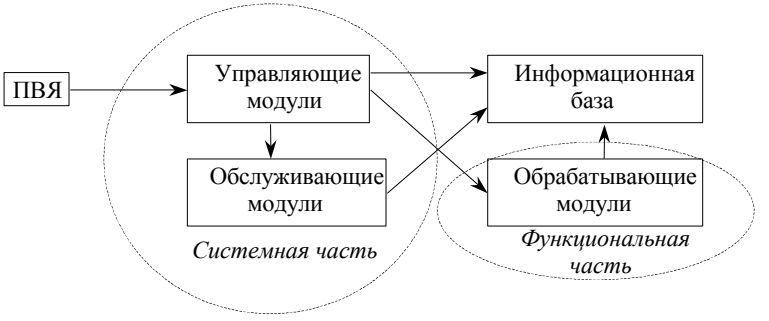


Рис. 4. Классификация СПО

2.3. Модель предметной области

Область науки или деятельности, к которой относятся задачи, решаемые с применением ППП, называют предметной областью пакета.

Математическая модель – это совокупность некоторых объектов (переменных) и связей между этими объектами.

Модель предметной области ППП (МПО) можно представить как совокупность данных, используемых при решении задачи и связи между этими данными. Характер связей определяется при разработке информационной базы пакета. Такие связи называются связями по определению.

Иной характер носят связи, реализуемые обрабатывающими модулями. Такие связи определены, но они реализуются только по прямому или косвенному указанию пользователя. Это функциональные связи.

Таким образом модель ПО пакета можно представить как объединение множества данных, связей по определению и функциональных связей.

$$МПО=\{X, R, F\},$$

где X – данные,

R – связи по определению,

F – функциональные связи.

Если в процессе выполнения пакета множество {X, R, F} остаются неизменными, а изменяются только значение данных, то такую МПО называют *статической*.

Если в процессе работы с пакетом пользователь имеет возможность изменять хотя бы одно из этих множеств, включая или удаляя из него компоненты, то это *динамическая МПО*.

Контрольные вопросы

1. Какими свойствами должен обладать ППП?
2. Как можно классифицировать ППП?
3. Дайте определение методо-ориентированным ППП.
4. Перечислите функции текстовых процессоров.
5. Перечислите функции систем управления базами данных.
6. Перечислите функции табличных процессоров.
7. Что такое интегрированные ППП?
8. Какие возможности табличных процессоров используют при проведении экономических расчетов?
9. Какие ППП можно использовать при проведении экономических расчетов?
10. Какие ППП относятся к классу универсальных?
11. Какие ППП относятся к классу проблемно-ориентированных?
12. Какие ППП относятся к классу методо-ориентированных?
13. Из каких основных частей состоит ППП?
14. Перечислите основные функции управляющих модулей пакета.
15. Перечислите основные функции обслуживающих модулей пакета.
16. Перечислите основные функции обрабатывающих модулей пакета.
17. Что такое модель предметной области пакета?
18. Из каких компонентов состоит модель предметной области пакета.

3. ПРОГРАММНЫЕ СРЕДСТВА

Программное средство (ПС), предназначенное для продажи, называют *программным изделием* (ПИ).

ПИ – программа на носителе данных, являющаяся продуктом промышленного производства. ПИ разрабатывается для обработки множества набора данных с учетом конкретных условий разных потребителей.

Отличием ПИ от любого продукта производства является то, что оно не имеет физического износа, происходит износ только физических носителей информации. Но происходит моральный износ ПИ, ограничивающий их жизненный цикл (разработка, внедрение, сопровождение).

ПИ требует постоянного сопровождения, т.к. требования пользователей в процессе эксплуатации постоянно дорабатываются. Дополнительных затрат требует и адаптация ПИ.

ПИ должно отвечать ряду требований:

1) ПИ должно создаваться в соответствии с государственными отраслевыми стандартами;

2) ПИ должно иметь установленную цену;

3) При реализации ПИ должны быть оговорены вопросы совершенствования (модернизации) ПИ организациями-поставщиками;

4) ПИ должно иметь документацию, которая обеспечивает возможность их применения пользователями различной квалификации. Состав и количество документации, сопровождающей ПИ, определяются в соответствии с ГОСТами на ПИ.

Контрольные вопросы

1. Какое определение стандарт дает программному средству?
2. Что такое программное изделие?
3. Что производит отрасль производства – информатика?
4. В чем особенность программного изделия, как продукта производства?
5. Что такое моральный износ?
6. Почему ПИ не подвержено физическому износу?

4. ЖИЗНЕННЫЙ ЦИКЛ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Понятие жизненного цикла программного обеспечения (ЖЦ ПО) является одним из базовых в программной инженерии. *Жизненный цикл программного обеспечения* определяется как период времени, который начинается с момента принятия решения о необходимости создания ПО и заканчивается в момент его полного изъятия из эксплуатации.

Основным нормативным документом, регламентирующим состав процессов ЖЦ ПО, является международный стандарт ISO/IEC 12207: 1995 «Information Technology -Software Life Cycle Processes» (ISO – International Organization for Standardization – Международная организация по стандартизации, 1EC – International Electrotechnical Commission – Международная комиссия по электротехнике). Он определяет структуру ЖЦ, содержащую процессы, действия и задачи, которые должны быть выполнены во время создания ПО. В данном стандарте *ПО (или программный продукт)* определяется как набор компьютерных программ, процедур и, возможно, связанной с ними документации и данных. *Процесс* определяется как совокупность взаимосвязанных действий, преобразующих некоторые входные данные в выходные. Каждый процесс характеризуется определенными задачами и методами их решения, исходными данными, полученными от других процессов, и результатами.

Каждый процесс разделен на набор действий, каждое действие – на набор задач. Каждый процесс, действие или задача инициируется и выполняется другим процессом по мере необходимости, причем не существует заранее определенных последовательностей выполнения (естественно, при сохранении связей по входным данным).

Следует отметить, что в России создание ПО первоначально, в 70-е гг., регламентировалось стандартами ГОСТ ЕСПД (Единой системы программной документации – серия ГОСТ 19.XXX), которые были ориентированы на класс относительно простых программ небольшого объема, создаваемых отдельными программистами. В настоящее время эти стандарты устарели концептуально и по форме, их сроки действия закончились и использование нецелесообразно. Процессы создания автоматизированных систем (АС), в состав которых входит и ПО, регламентированы стандартами ГОСТ 34.601-90 «Информационная технология. Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Стадии создания», ГОСТ 34.602-89 «Информационная технология. Комплекс стандартов на автоматизированные системы. Техническое задание на создание автоматизированной системы» и ГОСТ 34.603-92 «Информационная технология. Виды испытаний автоматизированных систем». Однако процессы создания ПО для современных распределенных ЭИС, функционирующих в неоднородной среде, в

этих стандартах отражены недостаточно, а отдельные их положения явно устарели. В результате для каждого серьезного проекта ЭИС приходится создавать комплекты нормативных и методических документов, регламентирующих процессы создания конкретного прикладного ПО, поэтому в отечественных разработках целесообразно использовать современные международные стандарты.

В соответствии со стандартом ISO/IEC 12207 все процессы ЖЦ ПО разделены на три группы (рис 5).

- пять основных процессов (приобретение, поставка, разработка, эксплуатация, сопровождение);
- восемь вспомогательных процессов, обеспечивающих выполнение основных процессов {документирование, управление конфигурацией, обеспечение качества, верификация, аттестация, совместная оценка, аудит, разрешение проблем);
- четыре организационных процесса (управление, создание инфраструктуры, усовершенствование, обучение).

4.1. Основные процессы ЖЦ ПО

Процесс приобретения (acquisition process). Он состоит из действий и задач заказчика, приобретающего ПО. Данный процесс охватывает следующие действия:

1) инициирование приобретения;
2) подготовку заявочных предложений;
3) подготовку и корректировку договора;
4) надзор за деятельностью поставщика;
5) приемку и завершение работ.

Инициирование приобретения включает следующие задачи:

- определение заказчиком своих потребностей в приобретении, разработке или усовершенствовании системы, программных продуктов или услуг;
- анализ требований к системе;
- принятие решения относительно приобретения, разработки или совершенствования существующего ПО;
- проверку наличия необходимой документации, гарантий, сертификатов,
- лицензий и поддержки в случае приобретения программного продукта; подготовку и утверждение плана приобретения, включающего требования к системе, тип договора, ответственность сторон и т.д.

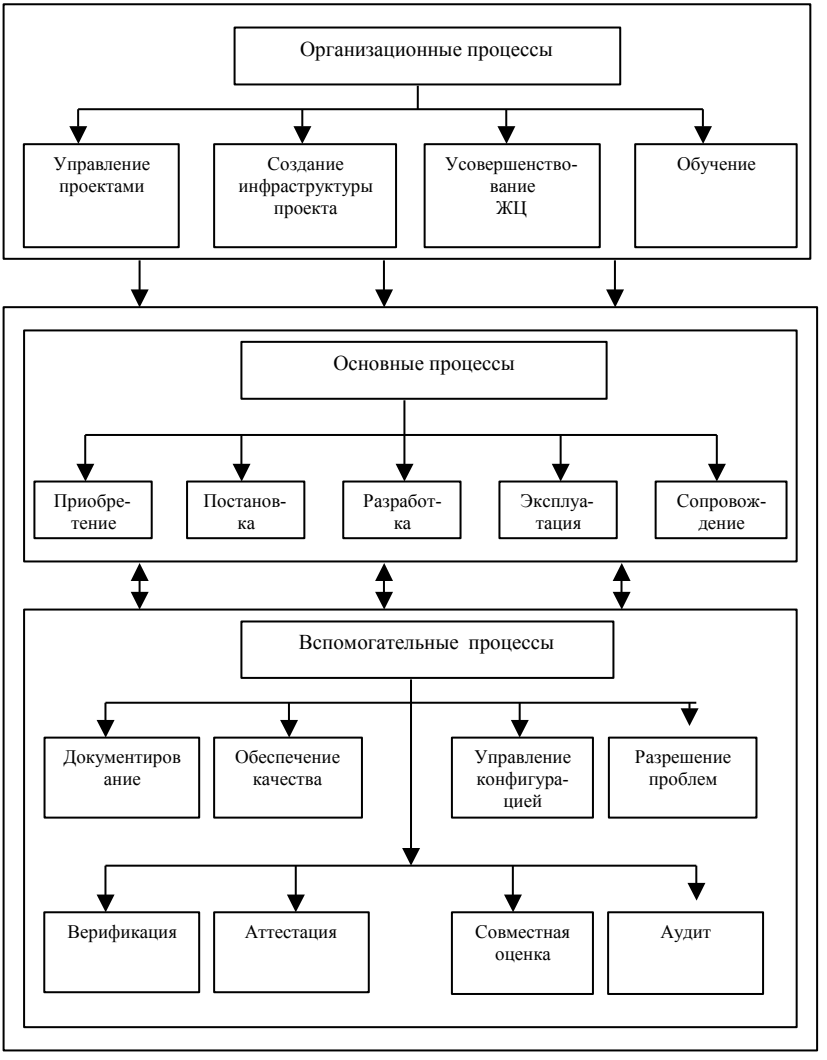


Рис. 5. Процессы ЖЦ ПО

Заявочные предложения должны содержать:

- требования к системе;
- перечень программных продуктов;
- условия и соглашения;

• технические ограничения (например, среда функционирования системы).

Заявочные предложения направляются выбранному поставщику (или нескольким поставщикам в случае проведения тендера). *Поставщик* -это организация, которая заключает договор с заказчиком на поставку системы, ПО или программной услуги на условиях, оговоренных в договоре.

Подготовка и корректировка договора включают следующие задачи:

• определение заказчиком процедуры выбора поставщика, включающей критерии оценки предложений возможных поставщиков;

• выбор конкретного поставщика на основе анализа предложений;

• подготовку и заключение договора с поставщиком;

• внесение изменений (при необходимости) в договор в процессе его выполнения.

Надзор за деятельностью поставщика осуществляется в соответствии с действиями, предусмотренными в процессах *совместной оценки* и *аудита*.

В процессе *приемки* подготавливаются и выполняются необходимые тесты. *Завершение работ* по договору осуществляется в случае удовлетворения всех условий приемки.

Процесс поставки (supply process). Он охватывает действия и задачи, выполняемые поставщиком, который снабжает заказчика программным продуктом или услугой. Данный процесс включает следующие действия:

1) инициирование поставки;
2) подготовку ответа на заявочные предложения;
3) подготовку договора;
4) планирование;
5) выполнение и контроль;
6) проверку и оценку;
7) поставку и завершение работ.

Инициирование поставки заключается в рассмотрении поставщиком заявочных предложений и принятии решения согласиться с выставленными требованиями и условиями или предложить свои,

Планирование включает следующие задачи:

• принятие решения поставщиком относительно выполнения работ своими силами или с привлечением субподрядчика;

• разработку поставщиком плана управления проектом, содержащего организационную структуру проекта, разграничение ответственности, технические требования к среде разработки и ресурсам, управление субподрядчиками и др.

Процесс разработки (development process). Он предусматривает действия и задачи, выполняемые разработчиком, и охватывает работы по созданию ПО и его компонентов в соответствии с заданными требо-

ваниями, включая оформление проектной и эксплуатационной документации, подготовку материалов, необходимых для проверки работоспособности и соответствующего качества программных продуктов, материалов, необходимых для организации обучения персонала, и т.д.

Процесс разработки включает следующие действия:

1) подготовительную работу;
2) анализ требований к системе;
3) проектирование архитектуры системы;
4) анализ требований к ПО;
5) проектирование архитектуры ПО;
6) детальное проектирование ПО;
7) кодирование и тестирование ПО;
8) интеграция ПО;
9) квалификационное тестирование ПО;
10) интеграцию системы;
11) квалификационное тестирование системы;
12) установку ПО
13) приемку ПО.

Подготовительная работа начинается с выбора модели ЖЦ ПО. Соответствующей масштабу, масштабу, значимости и сложности проекта. Действия и задачи процесса разработки должны соответствовать выбранной модели. Разработчик должен выбрать, адаптировать к условиям проекта и использовать согласованные с заказчиком стандарты, методы и средства разработки, а также составить план выполнения работ.

Анализ требований к системе подразумевает определение ее функциональных возможностей, пользовательских требований, требований к надежности и безопасности, требований к внешним интерфейсам и т.д. требования к системе оцениваются исходя из критериев реализуемости и возможности проверки при тестировании.

Проектирование архитектуры системы на высоком уровне заключается в определении компонентов ее оборудования, ПО и операций, выполняемых эксплуатирующим систему персоналом. Архитектура системы должна соответствовать требованиям, предъявляемым к системе, а также принятым стандартам и методам.

Анализ требований к ПО предполагает определение следующих характеристик для каждого компонента ПО:

- функциональных возможностей, включая характеристики производительности и среды функционирования компонента;
- внешних интерфейсов;
- спецификаций надежности и безопасности;
- эргономических требований;
- требований к используемым данным;
- требований к установке и приемке;

- требований к пользовательской документации;
- требований к эксплуатации и сопровождению.

Требования к ПО оцениваются исходя из критериев соответствия требованиям к системе, реализуемости и возможности проверки при тестировании.

Проектирование архитектуры ПО включает следующие задачи (для каждого компонента ПО):

- трансформацию требований к ПО в архитектуру, определяющую на высоком уровне структуру ПО и состав его компонентов;
- разработку и документирование программных интерфейсов ПО и баз данных;
- разработку предварительной версии пользовательской документации;
- разработку и документирование предварительных требований к текстам и планам интеграции ПО.
- Архитектура компонентов ПО должна соответствовать требованиям, предъявляемых к ним, а также принятым проектным стандартам и методам.

Детальное проектирование ПО *включает следующие задачи:*

- описание компонентов ПО и интерфейсов между ними на более низком уровне, достаточном для их последующего самостоятельного кодирования и тестирования;
- разработку и документирование детального проекта базы данных;
- обновление (при необходимости) пользовательской документации;
- разработку и документирование требований к тестам и планам тестирования компонентов По;
- обновление плана интеграции ПО.

Кодирование и тестирование ПО охватывают следующие задачи:

- разработку (кодирование) и документирование каждого компонента ПО и базы данных, а также совокупности тестовых процедур и данных для их тестирования;
- тестирование каждого компонента ПО и базы данных на соответствие предъявляемым к ним требованиям. Результаты тестирования компонентов должны быть документированы;
- обновление (при необходимости) пользовательской документации;
- обновление плана интеграции ПО.

Интеграция ПО предусматривает сборку разработанных компонентов ПО в соответствии с планом интеграции и тестирование агрегированных компонентов. Для каждого из агрегированных компонентов разрабатываются наборы тестов и тестовые процедуры, предназначенные для проверки каждого из квалификационных требований при последующем квалификационном тестировании. *Квалификационное требование* – это

набор критериев или условий, которые необходимо выполнить, чтобы квалифицировать программный продукт как соответствующий своим спецификациям и готовый к использованию в условиях эксплуатации.

Квалификационное тестирование ПО проводится разработчиком в присутствии заказчика (по возможности) для демонстрации того, что ПО удовлетворяет своим спецификациям и готово к использованию в условиях эксплуатации. Квалификационное тестирование выполняется для каждого компонента ПО по всем разделам требований при широком варьировании тестов. При этом также проверяются полнота технической и пользовательской документации и ее адекватность самим компонентам ПО.

Интеграция системы заключается в сборке всех ее компонентов, включая ПО и оборудование. После интеграции система, в свою очередь, подвергается *квалификационному тестированию* на соответствие совокупности требований к ней. При этом также производятся оформление и проверка полного комплекта документации на систему.

Установка ПО осуществляется разработчиком в соответствии с планом в той среде и на том оборудовании, которые предусмотрены договором. В процессе установки проверяется работоспособность ПО и баз данных. Если устанавливаемое ПО заменяет существующую систему, разработчик должен обеспечить их параллельное функционирование в соответствии с договором.

Приемка ПО предусматривает оценку результатов квалификационного тестирования ПО и системы и документирование результатов оценки, которые проводятся заказчиком с помощью разработчика. Разработчик выполняет окончательную передачу ПО заказчику в соответствии с договором, обеспечивая при этом необходимое обучение и поддержку.

Процесс эксплуатации (operation process). Он охватывает действия и задачи оператора – организации, эксплуатирующей систему. Данный процесс включает следующие действия:

1) подготовительную работу;
2) эксплуатационное тестирование;
3) эксплуатацию системы;
4) поддержку пользователей.

Подготовительная работа включает проведение оператором следующих задач:

– планирование действий и работ, выполняемых в процессе эксплуатации, и установку эксплуатационных стандартов;

– определение процедур локализации и разрешения проблем, возникающих в процессе эксплуатации.

Эксплуатационное тестирование осуществляется для каждой очередной редакции программного продукта, после чего она передается в эксплуатацию.

Эксплуатация системы выполняется в предназначенной для этого среде в соответствии с пользовательской документацией.

Поддержка пользователей заключается в оказании помощи и консультаций при обнаружении ошибок в процессе эксплуатации ПО.

Процесс сопровождения (maintenance process). Он предусматривает действия и задачи, выполняемые сопровождающей организацией (службой сопровождения). Данный процесс активизируется при изменениях (модификациях) программного продукта и соответствующей документации, вызванных возникшими проблемами или потребностями в модернизации либо адаптации ПО. В соответствии со стандартом IEEE-90 под *сопровождением* понимается внесение изменений в ПО в целях исправления ошибок, повышения производительности или адаптации к изменившимся условиям работы или требованиям.

Изменения, вносимые в существующее ПО, не должны нарушать его целостность. Процесс сопровождения включает перенос ПО в другую среду (миграцию) и заканчивается снятием ПО с эксплуатации.

Процесс сопровождения охватывает следующие действия:

1) подготовительную работу;
2) анализ проблем и запросов на модификацию ПО;
3) модификацию ПО;
4) проверку и приемку;
5) перенос ПО в другую среду;
6) снятие ПО с эксплуатации.

Подготовительная работа службы сопровождения включает следующие задачи:

- планирование действий и работ, выполняемых в процессе сопровождения;
- определение процедур локализации и разрешения проблем, возникающих в процессе сопровождения.

Анализ проблем и запросов на модификацию ПО, выполняемый службой сопровождения, включает следующие задачи:

- анализ сообщения о возникшей проблеме или запроса на модификацию ПО относительно его влияния на организацию, существующую систему и интерфейсы с другими системами. При этом определяются следующие характеристики возможной модификации: тип (корректирующая, улучшающая, профилактическая или адаптирующая к новой среде); масштаб (размеры модификации, стоимость и время ее реализации); критичность (воздействие на производительность, надежность или безопасность);
- оценка целесообразности проведения модификации и возможных вариантов ее проведения;
- утверждение выбранного варианта модификации.

Модификация ПО предусматривает определение компонентов ПО, их версий и документации, подлежащих модификации, и внесение необходимых изменений в соответствии с правилами процесса разработки. Подготовленные изменения тестируются и проверяются по критериям, определенным в документации. При подтверждении корректности изменений в программах производится корректировка документации.

Проверка и приемка заключаются в проверке целостности модифицированной системы и утверждении внесенных изменений.

При *переносе ПО в другую среду* используются имеющиеся или разрабатываются новые средства переноса, затем выполняется конвертирование программ и данных в новую среду. С целью облегчить переход предусматривается параллельная эксплуатация ПО в старой и новой среде в течение некоторого периода, когда проводится необходимое обучение пользователей работе в новой среде.

Снятие ПО с эксплуатации осуществляется по решению заказчика при участии эксплуатирующей организации, службы сопровождения и пользователей. При этом программные продукты и соответствующая документация подлежат архивированию в соответствии с договором. Аналогично переносу ПО в другую среду с целью облепить переход к новой системе предусматривается параллельная эксплуатация старого и нового ПО в течение некоторого периода, когда выполняется необходимое обучение пользователей работе с новой системой.

4.2. Вспомогательные процессы ЖЦ ПО

Процесс документирования (documentation process). Он предусматривает формализованное описание информации, созданной в течение ЖЦ ПО. Данный процесс состоит из набора действий, с помощью которых планируют, проектируют, разрабатывают, выпускают, редактируют, распространяют и сопровождают документы, необходимые для всех заинтересованных лиц, таких, как руководство, технические специалисты и пользователи системы.

Процесс документирования включает следующие действия:

1) подготовительную работу;
2) проектирование и разработку;
3) выпуск документации;
1) сопровождение.

Процесс управления конфигурацией (configuration management process). Он предполагает применение административных и технических процедур на всем протяжении ЖЦ ПО для определения состояния компонентов ПО в системе, управления модификациями ПО, описания и подготовки отчетов о состоянии компонентов ПО и запросов на модификацию, обеспечения полноты, совместимости и корректности ком-

понентов ПО, управления хранением и поставкой ПО. Согласно стандарту IEEE-90 под *конфигурацией ПО* понимается совокупность его функциональных и физических характеристик, установленных в технической документации и реализованных в ПО.

Управление конфигурацией позволяет организовать, систематически учитывать и контролировать внесение изменений в ПО на всех стадиях ЖЦ. Общие принципы и рекомендации по управлению конфигурацией ПО отражены в проекте стандарта ISO/IEC CD 12207-2: 1995 «Information Technology – Software Life Cycle Processes. Part 2. Configuration Management for Software».

Процесс управления конфигурацией включает следующие действия:

1) подготовительную работу;
2) идентификацию конфигурации;
3) контроль конфигурации;
4) учет состояния конфигурации;
5) оценку конфигурации;
6) управление выпуском и поставку.

Подготовительная работа заключается в планировании управления конфигурацией.

Идентификация конфигурации устанавливает правила, с помощью которых можно однозначно идентифицировать и различать компоненты ПО и их версии. Кроме того, каждому компоненту и его версиям соответствует однозначно обозначаемый комплект документации. В результате создается база для однозначного выбора и манипулирования версиями компонентов ПО, использующая ограниченную и упорядоченную систему символов, идентифицирующих различные версии ПО. *Контроль конфигурации* предназначен для систематической оценки предполагаемых модификаций ПО и координированной их реализации с учетом эффективности каждой модификации и затрат на ее выполнение. Он обеспечивает контроль состояния и развития компонентов ПО и их версий, а также адекватность реально изменяющихся компонентов и их комплектной документации.

Учет состояния конфигурации представляет собой регистрацию состояния компонентов ПО, подготовку отчетов Обо всех реализованных и отвергнутых модификациях версий компонентов ПО, Совокупность отчетов обеспечивает однозначное отражение текущего состояния системы и ее компонентов, а также ведение истории модификаций.

Оценка конфигурации заключается в оценке функциональной полноты компонентов ПО, а также соответствия их физического состояния текущему техническому описанию.

Управление выпуском и поставка охватывают изготовление эталонных копий программ и документации, их хранение и поставку пользователям в соответствии с порядком, принятым в организации.

Процесс обеспечения качества (quality assurance process). Он обеспечивает соответствующие гарантии того, что ПО и процессы его ЖЦ соответствуют заданным требованиям и утвержденным планам. Под *качеством ПО* понимается совокупность свойств, которые характеризуют способность ПО удовлетворять заданным требованиям.

Для получения достоверных оценок создаваемого ПО процесс обеспечения его качества должен происходить независимо от субъектов, непосредственно связанных с разработкой ПО. При этом могут использоваться результаты других вспомогательных процессов, таких, как верификация , аттестация, совместная оценка, аудит и разрешение проблем.

Процесс обеспечения качества включает следующие действия:

1) подготовительную работу;
2) обеспечение качества продукта;
3) обеспечение качества процесса;
4) обеспечение прочих показателей качества системы.

Подготовительная работа заключается в координации с другими вспомогательными процессами и планировании самого процесса обеспечения качества с учетом используемых стандартов, методов, процедур и средств.

Обеспечение качества продукта подразумевает гарантирование полного соответствия программных продуктов и их документации требованиям заказчика, предусмотренным в договоре.

Обеспечение качества процесса предполагает гарантирование соответствия процессов ЖЦ ПО, методов разработки, среды разработки и квалификации персонала условиям договора, установленным стандартам и процедурам.

Обеспечение прочих показателей качества системы осуществляется в соответствии с условиями договора и стандартом качества ISO 9001.

Процесс верификации (verification process). Он состоит в определении того, что программные продукты, являющиеся результатами некоторого действия, полностью удовлетворяют требованиям или условиям, обусловленным предшествующими действиями *(верификация* в узком смысле означает формальное доказательство правильности ПО). Для повышения эффективности верификация должна как можно раньше интегрироваться с использующими ее процессами (такими, как поставка, разработка, эксплуатация или сопровождение). Данный процесс может включать анализ, оценку и тестирование,

Верификация может проводиться с различными степенями независимости. Степень независимости может варьироваться от выполнения верификации самим исполнителем или другим специалистом данной организации до ее выполнения специалистом другой организации с различными вариациями. Если процесс верификации осуществляется

организацией, не зависящей от поставщика, разработчика, оператора или службы сопровождения, то он называется *процессом независимой верификации*.

Процесс верификации включает следующие действия:

1) подготовительную работу;
2) верификацию.

В процессе верификации проверяются следующие условия:

- непротиворечивость требований к системе и степень учета потребностей пользователей;
- возможности поставщика выполнить заданные требования;
- соответствие выбранных процессов ЖЦ ПО условиям договора;
- адекватность стандартов, процедур и среды разработки процессам ЖЦ ПО;
- соответствие проектных спецификаций ПО заданным требованиям;
- корректность описания в проектных спецификациях входных и выходных данных, последовательности событий, интерфейсов, логики и т.д.;
- соответствие кода проектным спецификациям и требованиям;
- тестируемость и корректность кода, его соответствие принятым стандартам кодирования;
- корректность интеграции компонентов ПО в систему;
- адекватность, полнота и непротиворечивость документации.

Процесс аттестации (validation process). Он предусматривает определение полноты соответствия заданных требований и созданной системы или программного продукта их конкретному функциональному назначению. Под *аттестацией* обычно понимается подтверждение и оценка достоверности проведенного тестирования ПО. Аттестация должна гарантировать полное соответствие ПО спецификациям, требованиям и документации, а также возможность его безопасного и надежного применения пользователем. Аттестацию рекомендуется выполнять путем тестирования во всех возможных ситуациях и использовать при этом независимых специалистов. Аттестация может проводиться на начальных стадиях ЖЦ ПО или как часть работы по приемке ПО.

Аттестация, так же как и верификация, может осуществляться с различными степенями независимости. Если процесс аттестации выполняется организацией, не зависящей от поставщика, разработчика, оператора или службы сопровождения, то он называется *процессом независимой аттестации*.

Процесс аттестации включает следующие действия:

1) подготовительную работу;
2) аттестацию.

Процесс совместной оценки (joint review process). Он предназначен для оценки состояния работ по проекту и ПО, создаваемого при выполнении данных работ (действий). Он сосредоточен в основном на контроле планирования и управления ресурсами, персоналом, аппаратурой и инструментальными средствами проекта.

Оценка применяется как на уровне управления проектом, так и на уровне технической реализации проекта и проводится в течение всего срока действия договора. Данный процесс может выполняться двумя любыми сторонами, участвующими в договоре, при этом одна сторона проверяет другую.

Процесс совместной оценки включает следующие действия:

1) подготовительную работу;
2) оценку управления проектом;
3) техническую оценку.

Процесс аудита (audit process). Он представляет собой определение соответствия требованиям, планам и условиям договора. Аудит может выполняться двумя любыми сторонами, участвующими в договоре, когда одна сторона проверяет другую.

Аудит – это ревизия (проверка), проводимая компетентным органом (лицом) в целях обеспечения независимой оценки степени соответствия ПО или процессов установленным требованиям. Аудит служит для установления соответствия реальных работ и отчетов требованиям, планам и контракту. Аудиторы (ревизоры) не должны иметь прямой зависимости от разработчиков ПО. Они определяют состояние работ, использование ресурсов, соответствие документации спецификациям и стандартам, корректность тестирования.

Процесс аудита включает следующие действия:

1) подготовительную работу;
2) аудит.

Процесс разрешения проблем (problem resolution process). Он предусматривает анализ и решение проблем (включая обнаруженные несоответствия) независимо от их происхождения или источника, которые обнаружены в ходе разработки, эксплуатации, сопровождения или других процессов. Каждая обнаруженная проблема должна быть идентифицирована, описана, проанализирована и разрешена.

Процесс разрешения проблем включает следующие действия:

1) подготовительную работу;
2) разрешение проблем.

4.3. Организационные процессы ЖЦ ПО

Процесс управления (management process). Он состоит из действий и задач, которые могут выполняться любой стороной, управляющей

своими процессами. Данная сторона (менеджер) отвечает за управление выпуском продукта, управление проектом и управление задачами соответствующих процессов, таких, как приобретение, поставка, разработка, эксплуатация, сопровождение и др.

Процесс управления включает следующие действия:

1) инициирование и определение области управления;
2) планирование;
3) выполнение и контроль;
4) проверку и оценку;
5) завершение.

При *инициировании* менеджер должен убедиться, что необходимые для управления ресурсы (персонал, оборудование и технология) имеются в его распоряжении в достаточном количестве.

Планирование подразумевает выполнение, как минимум, следующих задач:

- составление графиков выполнения работ;
- оценку затрат;
- выделение требуемых ресурсов;
- распределение ответственности;
- оценку рисков, связанных с конкретными задачами;
- создание инфраструктуры управления.

Процесс создания инфраструктуры (infrastructure process). Он охватывает выбор и поддержку (сопровождение) технологии, стандартов и инструментальных средств, выбор и установку аппаратных и программных средств, используемых для разработки, эксплуатации или сопровождения ПО. Инфраструктура должна модифицироваться и сопровождаться в соответствии с изменениями требований к соответствующим процессам. Инфраструктура, в свою очередь, является одним из объектов управления конфигурацией.

Процесс создания инфраструктуры включает следующие действия:

1) подготовительную работу;
2) создание инфраструктуры;
3) сопровождение инфраструктуры.

Процесс усовершенствования (improvement process). Он предусматривает оценку, измерение, контроль и усовершенствование процессов ЖЦ ПО. Данный процесс включает следующие действия:

1) создание процесса;
2) оценку процесса;
3) усовершенствование процесса.

Усовершенствование процессов ЖЦ ПО направлено на повышение производительности труда всех участвующих в них специалистов за счет совершенствования используемой технологии, методов управления, выбора инструментальных средств и обучения персонала.

Усовершенствование основано на анализе достоинств и недостатков каждого процесса. Такому анализу в большой степени способствует накопление в организации исторической, технической, экономической и иной информации по реализованным проектам.

Процесс обучения (training process). Он охватывает первоначальное обучение и последующее постоянное повышение квалификации персонала. Приобретение, поставка, разработка, эксплуатация и сопровождение ПО в значительной степени зависят от уровня знаний и квалификации персонала. Например, разработчики ПО должны пройти необходимое обучение методам и средствам программной инженерии. Содержание процесса обучения определяется требованиями к проекту. Оно должно учитывать необходимые ресурсы и технические средства обучения. Должны быть разработаны и представлены методические материалы, необходимые для обучения пользователей в соответствии с учебным планом. Процесс обучения включает следующие действия:

1) подготовительную работу;
2) разработку учебных материалов;
3) реализацию плана обучения

Контрольные вопросы

1. Что такое жизненный цикл программного обеспечения?
2. Чем регламентируется ЖЦ ПО?
3. Какие группы процессов входят в состав ЖЦ ПО и какие процессы входят в состав каждой группы?
4. Какие процессы, по вашему мнению, наиболее часто используются в реальных процессах и почему?
5. Что понимается под стадией ЖЦ ПО и какие стадии входят в его состав?
6. Каково соотношение между стадиями и процессами ЖЦ ПО?
7. Какие процессы ЖЦ используются для получения достоверных оценок качества ПО?
8. Когда ЖЦ ПС заканчивается?

5. МОДЕЛИ ЖИЗНЕННОГО ЦИКЛА ПО

Под *моделью ЖЦ ПО* понимается структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач на протяжении ЖЦ. Модель ЖЦ зависит от специфики, масштаба и сложности проекта и специфики условий, в которых система создается и функционирует.

Стандарт ISO/IEC I2207 не предлагает конкретную модель ЖЦ и методы разработки ПО. Его положения являются общими для любых моделей ЖЦ, методов и технологий разработки ПО. Стандарт

описывает структуру процессов ЖЦ ПО, но не конкретизирует в деталях, как реализовать или выполнить действия и задачи, включенные в эти процессы.

Модель ЖЦ любого конкретного ПО ЭИС определяет характер *процесса его создания,* который представляет собой совокупность упорядоченных во времени, взаимосвязанных и объединенных в стадии работ, выполнение которых необходимо и достаточно для создания ПО, соответствующего заданным требованиям. Под *стадией создания ПО* понимается часть процесса создания ПО, ограниченная некоторыми временными рамками и заканчивающаяся выпуском конкретного продукта (моделей ПО, программных компонентов, документации), определяемого заданными для данной стадии требованиями. Стадии создания ПО выделяются по соображениям рационального планирования и организации работ, заканчивающихся заданными результатами. В состав жизненного цикла ПО обычно включаются следующие стадии:

- Формирование требований к ПО.
- Проектирование.
- Реализация.
- Тестирование.
- Ввод в действие.
- Эксплуатация и сопровождение.
- Снятие с эксплуатации.

Стадия формирования требований к ПО. Она является одной из важнейших, поскольку определяет успех всего проекта. Данная стадия включает следующие этапы:

- *планирование работ,* предваряющее работы над проектом. Основными задачами этапа являются: определение целей разработки, предварительная экономическая оценка проекта, построение плана-графика выполнения работ, создание и обучение совместной рабочей группы;
- *проведение обследования деятельности автоматизируемого объекта (организации),* в рамках которого осуществляются: предварительное выявление требований к будущей системе: определение структуры организации;

определение перечня целевых функций организации; анализ распределения функций по подразделениям и сотрудникам; выявление функциональных взаимодействий между подразделениями, информационных потоков внутри подразделений и между ними, внешних по отношению к организации объектов и внешних информационных взаимодействий; анализ существующих средств автоматизации деятельности организации;

• *построение моделей деятельности организации,* предусматривающее обработку материалов обследования и построение двух видов моделей:

– модели *«AS-IS» («как есть»),* отражающей существующее на момент обследования положение пел в организации и позволяющей понять, каким образом функционирует данная организация, а также выявить узкие места и сформулировать предложения по улучшению ситуации;

– модели *«TO-BE» («как должно быть»),* отражающей представление о новых технологиях работы организации.

Каждая из моделей включает в себя полную функциональную и информационную модель деятельности организации, а также, в случае необходимости, модель, описывающую динамику поведения организации.

Переход от модели «AS-IS» к модели «TO-BE» может выполняться двумя способами:

1) совершенствованием существующих технологий на основе оценки их эффективности;

2) радикальным изменением технологий и перепроектирование бизнес-процессов (реинжиниринг бизнес-процессов).

Построенные модели имеют самостоятельное практическое значение. Например, модель «AS-IS» позволяет выявлять узкие места в существующих технологиях и предлагать рекомендации по решению проблем независимо от того, предполагается на данном этапе дальнейшая разработка ЭИС или нет. Кроме того, модель облегчает обучение сотрудников конкретным направлениям деятельности организации за счет использования наглядных диаграмм.

Стадия проектирования. Она, как правило, включает следующие этапы:

• *разработка системного проекта.* На этом этапе дается ответ на вопрос: «Что должна делать будущая система?», а именно: определяются архитектура системы, ее функции, внешние условия функционирования, интерфейсы и распределение функций между пользователями и системой, требования к программным и информационным компонентам, состав исполнителей и сроки разработки. Основу системного проекта составляют модели проектируемой ЭИС, которые строятся на основе модели «TO-BE». Документальным результатом этапа является техническое задание;

• *разработка технического проекта.* На этом этапе на основе системного проекта осуществляется собственно проектирование системы, включающее проектирование архитектуры системы и детальное проек-

тирование. Таким образом, дается ответ на вопрос: «Как построить систему, чтобы она удовлетворяла предъявленным к ней требованиям?».

Содержание последующих стадий совпадает в основном с соответствующими процессами ЖЦ ПО.

К настоящему времени наибольшее распространение получили следующие две основные модели ЖЦ ПО: каскадная модель (]970 -I985 гг.) и спиральная модель (1986 – 1990 гг.).

В однородных ЭИС 70-х и 80-х гг. прикладное ПО представляло собой единое целое. Для разработки такого типа ПО применялся каскадный подход (рис. 6). Принципиальной особенностью каскадного подхода является следующее: *переход на следующую стадию осуществляется только после того, как будет полностью завершена работа на текущей стадии, и возвратов на пройденные стадии не предусматривается*. Каждая стадия заканчивается получением некоторых результатов, которые служат в качестве исходных данных для следующей стадии. Требования к разрабатываемому ПО, определенные на стадии формирования требований, строго документируются в виде технического задания и фиксируются на все время разработки проекта. Каждая стадия завершается выпуском полного комплекта документации, достаточной для того, чтобы разработка могла быть продолжена другой командой разработчиков. Критерием качества разработки при таком подходе является точность выполнения спецификаций технического задания.

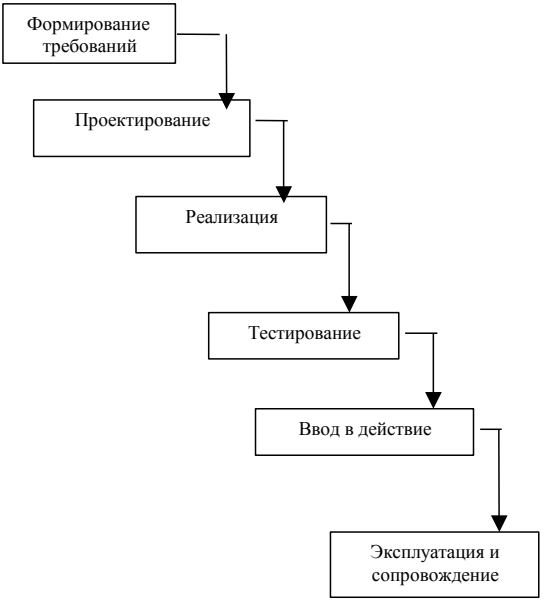


Рис. 6. Каскадная схема разработки ПО

При этом основное внимание разработчиков сосредоточивается на достижении оптимальных значений технических характеристик разрабатываемого ПО: производительности, объема занимаемой памяти и др.

Преимущества применения каскадного способа заключаются в следующем:

- на каждой стадии формируется законченный набор проектной документации, отвечающий критериям полноты и согласованности;
- выполняемые в логичной последовательности стадии работ позволяют планировать сроки завершения всех работ и соответствующие затраты.

Каскадный подход хорошо зарекомендовал себя при построении ЭИС, для которых в самом начале разработки можно достаточно точно и полно сформулировать все требования, с тем чтобы предоставить разработчикам свободу реализовать их технически как можно лучше. В эту категорию попадают сложные системы с большим количеством задач вычислительного характера, системы реального времени и др.

В то же время этот подход обладает рядом недостатков, вызванных прежде всего тем, что реальный процесс создания ПО никогда полностью не укладывался в такую жесткую схему. Процесс создания ПО носит, как правило, *итерационный* характер: результаты очередной стадии часто вызывают изменения в проектных решениях, выработанных на более ранних стадиях. Таким образом, постоянно
возникает потребность в возврате к предыдущим стадиям и уточнении или пересмотре ранее принятых решений. В результате реальный процесс создания ПО принимает иной вид (рис. 7).

Изображенную на рис. 7 схему часто относят к отдельной модели, так называемой *модели с промежуточным контролем*, в которой межстадийные корректировки обеспечивают большую надежность по сравнению с каскадной моделью, хотя и увеличивают весь период разработки.

Основным недостатком каскадного подхода являются существенное запаздывание с получением результатов и, как следствие, достаточно высокий риск создания системы, не удовлетворяющей изменившимся потребностям пользователей. Практика показывает, что на начальной стадии проекта полностью и точно сформулировать все требования к будущей системе не удается. Это объясняется двумя причинами: 1) пользователи не в состоянии сразу изложить все свои требования и не могут предвидеть, как они изменятся в ходе разработки; 2) за время разработки могут произойти изменения во внешней среде, которые повлияют на требования к системе. В рамках каскадного подхода требования к ЭИС фиксируются в виде технического задания на все время ее создания, а согласование получаемых результатов с пользователями производится только в точках, планируемых после завершения каждой стадии (при этом возможна корректировка результатов по замечаниям

пользователей, если они не затрагивают требования, изложенные в техническом задании). Таким образом, пользователи могут внести существенные замечания только после того, как работа над системой будет полностью завершена. В случае неточного изложения требований или их изменения в течение длительного периода создания ПО пользователи получают систему, не удовлетворяющую их потребностям. В результате приходится начинать новый проект, который может постигнуть та же участь.

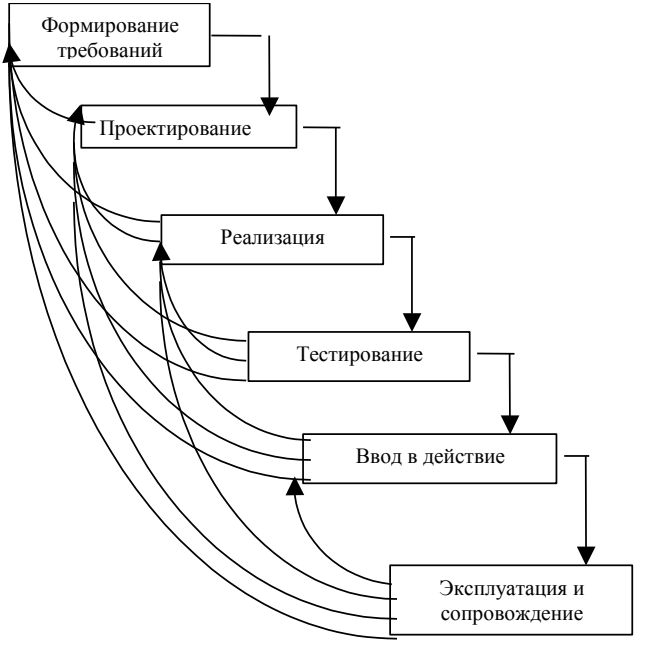


Рис. 7. Реальный процесс разработки ПО

Для преодоления перечисленных проблем в середине 80-х гг. была предложена *спиральная модель ЖЦ* (рис. 8). Ее принципиальной особенностью является следующее: прикладное ПО создается не сразу, как в случае каскадного подхода, а по частям с использованием метода прототипирования. Под прототипом понимается действующий программный компонент, реализующий отдельные функции и внешние интерфейсы разрабатываемого ПО. Создание прототипов осуществляется в несколько итераций, или витков спирали. Каждая итерация соответствует созданию фрагмента или версии ПО, на ней уточняются цели и характеристики проекта, оценивается качество полученных результатов и планируются работы следующей итерации. На каждой итерации произво-

дится тщательная оценка риска превышения сроков и стоимости проекта, чтобы определить необходимость выполнения еще одной итерации, степень полноты и точности понимания требований к системе, а также целесообразность прекращения проекта. Спиральная модель избавляет пользователей и разработчиков ПО от необходимости полного и точного формулирования требований к системе на начальной стадии, поскольку они уточняются на каждой итерации. Таким образом, углубляются и последовательно конкретизируются детали проекта, и в результате выбирается обоснованный вариант, который доводится до реализации.

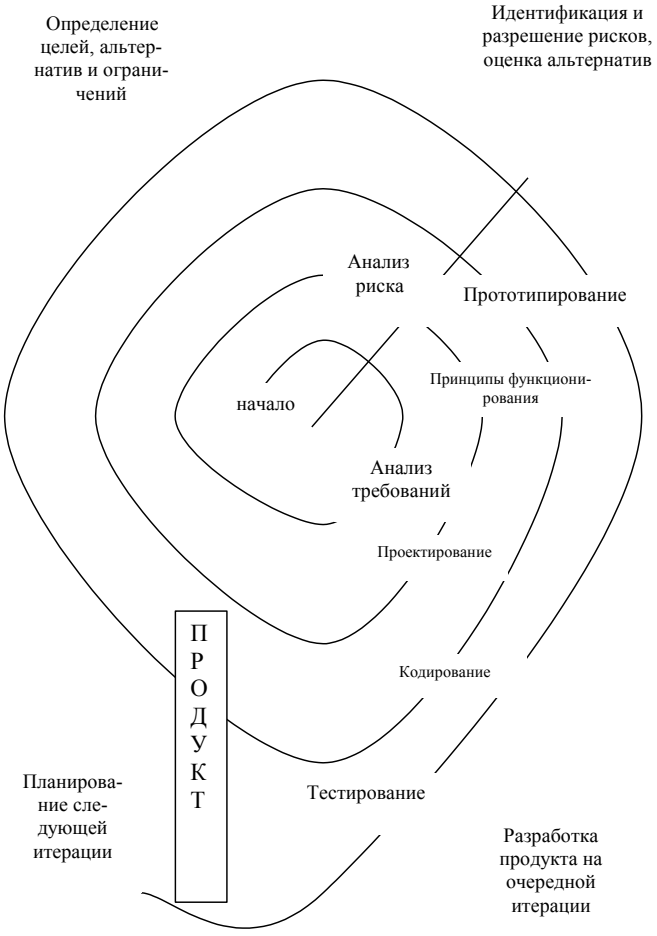


Рис. 8. Спиральная модель

Разработка итерациями отражает объективно существующий спиральный цикл создания системы. Неполное завершение работ на каждой стадии позволяет переходить на следующую стадию, не дожидаясь полного завершения работы на текущей. При итеративном способе разработки недостающую работу можно будет выполнить на следующей итерации. Главная же задача – как можно быстрее показать пользователям системы работоспособный продукт, тем самым активизируя процесс уточнения и дополнения требований. Спиральная модель не исключает использования каскадного подхода на завершающих стадиях проекта в тех случаях, когда требования к системе оказываются полностью определенными. Основная проблема спирального цикла – определение момента перехода на следующую стадию. Для ее решения необходимо ввести временные ограничения на каждую из стадий жизненного цикла. Переход осуществляется в соответствии с планом, даже если не вся запланированная работа закончена. План составляется на основе статистических данных, полученных в предыдущих проектах, и личного опыта разработчиков.

5.1. Подход RAD

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

- небольших групп разработчиков (от 3 до 7 человек), выполняющих работы по проектированию отдельных подсистем ПО. Это обусловлено требованием максимальной управляемости коллектива;
- короткого, но тщательно проработанного производственного графика (до 3 месяцев);
- повторяющегося цикла, при котором разработчики по мере того, как приложение начинает обретать форму, запрашивают и реализуют в продукте требования, полученные в результате взаимодействия с заказчиком.

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

Жизненный цикл ПО в соответствии с подходом RAD включает четыре стадии:

- анализ и планирование требований;
- проектирование;
- реализация;
- внедрение.

На стадии анализа и планирования требований пользователи осуществляют следующие действия:

- определяют функции, которые должна выполнять система;
- выделяют наиболее приоритетные функции, требующие проработки в первую очередь;
- описывают информационные потребности.

Формулирование требований к системе осуществляется в основном силами пользователей под руководством специалистов-разработчиков. Кроме того, на данной стадии решаются следующие задачи:

- ограничивается масштаб проекта;
- устанавливаются временные рамки для каждой из последующих стадий;
- определяется сама возможность реализации проекта в заданных размерах финансирования, на имеющихся аппаратных средствах и т.п.

Результатом стадии должны быть:

- список расставленных по приоритету функций будущего ПО ЭИС;
- предварительные модели ПО.

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

- более детально рассматриваются процессы системы;
- при необходимости для каждого элементарного процесса создается частичный прототип: экранная форма, диалог, отчет, устраняющий неясности пли неоднозначности;
- устанавливаются требования разграничения доступа к данным;
- определяется состав необходимой документации.

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

- входной элемент приложения (входной документ или экранная форма);
- выходной элемент приложения (отчет, документ, экранная форма);
- запрос (пара «вопрос/ответ»);

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

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

Далее проект распределяется между различными командами разработчиков. Реализация подсистем должна выполняться отдельными группами специалистов. При этом необходимо обеспечить координацию ведения общего проекта и исключить дублирование результатов работ каждой проектной группы, которое может возникнуть в силу наличия общих данных и функций. Результатом данной стадии должны быть:

• общая информационная модель системы;

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

• точно определенные интерфейсы между автономно разрабатываемыми подсистемами;

• построенные прототипы экранных форм, отчетов, диалогов.

В отличие от имевшего место ранее подхода, при котором использовались специфические средства прототипирования, не предназначенные для построения реальных приложений, а прототипы выбрасывались после того, как выполняли задачу устранения неясностей в проекте, в подходе RAD каждый прототип развивается в часть будущей системы. Таким образом, на следующую стадию передается более полная и полезная информация.

На стадии реализации выполняется непосредственно сама быстрая разработка приложения:

• разработчики производят итеративное построение реальной системы на основе полученных на предыдущей стадии моделей, а также требований нефункционального характера (требований к надежности, производительности и т.п.);

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

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

• осуществляется анализ использования данных и определяется необходимость их распределения;

• производится физическое проектирование базы данных;

• формулируются требования к аппаратным ресурсам;

• устанавливаются способы увеличения производительности;

• завершается разработка документации проекта.

Результатом стадии является готовая система, удовлетворяющая всем согласованным требованиям.

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

Следует, однако, отметить, что подход RAD, как и любой другой, не может претендовать на универсальность. Он хорош в первую очередь для относительно небольших проектов, разрабатываемых для конкретного заказчика.

Подход RAD не применим для построения сложных расчетных программ, операционных систем или программ управления сложными объектами в реальном масштабе времени, т.е. программ, содержащих большой объем (сотни тысяч строк) уникального кода.

Контрольные вопросы

1. Что называют моделью ЖЦ ПО?
2. Как выбирается модель ЖЦ ПО?
3. Какие стадии ЖЦ чаще всего присутствуют в модели ЖЦ?
4. Каковы принципиальные особенности каскадной модели?
5. В чем заключаются преимущества и недостатки каскадной модели?
7. Каковы принципиальные особенности спиральной модели?
8. В чем заключаются преимущества и недостатки спиральной модели?
9. Каковы особенности RAD-подхода при разработке прикладного ПО?
10. Какая модель используется при RAD-подходе?
11. Почему RAD-подход не используют при проектировании сложных систем?
12. В чем отличие между моделью ЖЦ и жизненным циклом ПС?

6. РАЗРАБОТКА ТРЕБОВАНИЙ И ВНЕШНЕЕ ПРОЕКТИРОВАНИЕ ПО

6.1. Понятии метода и технологии проектирования ПО

Методы и инструментальные средства проектирования (CASE-средства) составляют центральную часть формализованной дисциплины выполнения проекта любого ПО ЭИС. *Метод проектирования ПО* представляет собой организованную совокупность процессов создания ряда моделей, которые описывают различные аспекты разрабатываемой системы с использованием четко определенной нотации.

На более формальном уровне метод определяется как совокупность следующих составляющих:

- *концепций и теоретических основ.* В качестве таких основ могут выступать структурный или объектно-ориентированный подход;
- *нотаций,* используемых для построения моделей статической структуры и динамики поведения проектируемой системы. В качестве таких нотаций обычно используются графические диаграммы, поскольку они наиболее наглядны и просты в восприятии (диаграммы потоков данных и диаграммы «сущность-связь» для структурного подхода, диаграммы вариантов использования, диаграммы классов и др. – для объектно-ориентированного подхода);
- *процедуры,* определяющей практическое применение метода (последовательность и правила построения моделей, критерии, используемые для оценки результатов).

Методы реализуются через конкретные технологии и поддерживающие их методики, стандарты и инструментальные средства, которые обеспечивают выполнение процессов ЖЦ ПО.

Технология проектирования ПО определяется как совокупность технологических операций проектирования в их последовательности и взаимосвязи, приводящая к разработке проекта ПО.

6.1.1. Требования к технологии

Современная технология проектирования ПО ЭИС должна обеспечивать:

- соответствие стандарту ISO/IEC 12207 (поддержка всех процессов ЖЦ ПО);
- гарантированное достижение целей разработки ЭИС в рамках установленного бюджета, с заданным качеством и в установленное время;
- возможность декомпозиции проекта на составные части, разрабатываемые группами исполнителей ограниченной численности (3–7 человек), с последующей интеграцией составных частей;

• минимальное время получения работоспособного ПО ЭИС. Речь идет не о сроках готовности всей ЭИС, а о сроках реализации отдельных подсистем. Реализация ПО ЭИС в целом в короткие сроки может потребовать привлечения большого числа разработчиков. При этом эффект может оказаться ниже, чем при реализации в более короткие сроки отдельных подсистем меньшим числом разработчиков. Практика показывает, что даже при наличии полностью завершенного проекта внедрение ЭИС зачастую идет последовательно по отдельным подсистемам;

• независимость получаемых проектных решений от средств реализации ЭИС (СУБД, операционных систем, языков и систем программирования);

• поддержка комплексом согласованных CASE-средств, обеспечивающих автоматизацию процессов, выполняемых на всех стадиях ЖЦ.

Современные технологии поставляются, как правило, в электронном, виде вместе с CASE-средствами и включают библиотеки процессов, шаблонов, методов, моделей и других компонентов, предназначенных для построения ПО того класса систем, на который ориентирована технология. Электронные технологии включают также средства, которые должны обеспечивать их адаптацию для конкретных пользователей и развитие по результатам выполнения конкретных проектов.

Процесс адаптации заключается в удалении ненужных процессов и действий ЖЦ, компонентов методов, в изменении неподходящих или в добавлении собственных процессов и действий, а также методов, методик, стандартов и руководств. Настройка технологии может осуществляться также по следующим параметрам: стадии ЖЦ, участники проекта, используемые модели ЖЦ и др.

Электронные технологии (и поддерживающие их CASE-средства) составляют ядро комплекса согласованных инструментальных средств среды разработки ЭИС.

Реальное применение любой технологии проектирования ПО ЭИС в конкретной организации и конкретном проекте невозможно без выработки ряда стандартов (правил, соглашений), которые должны соблюдаться всеми участниками проекта (это особенно актуально при коллективной разработке ПО большим количеством групп специалистов). К таким стандартам относятся следующие:

• стандарт проектирования;

• стандарт оформления проектной документации;

• стандарт интерфейса конечного пользователя с системой.

Стандарт проектирования. Он должен устанавливать:

• набор необходимых моделей (диаграмм) на каждой стадии проектирования и степень их детализации;

• правила фиксации проектных решений на диаграммах, в том числе правила именования объектов (включая соглашения по терминологии), набор атрибутов для всех объектов и правила их заполнения на каждой стадии, правила оформления диаграмм (включая требования к форме и размерам объектов) и т.д.;

• требования к конфигурации рабочих мест разработчиков, включая настройки операционной системы, настройки CASE-средств и т.д.;

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

Стандарт оформления проектной документации. Он должен устанавливать:

• комплектность, состав и структуру документации на каждой стадии проектирования (в соответствии со стандартом ГОСТ Р ИСО9127-94 «Системы обработки информации. Документация пользователя и информация на упаковке потребительских программных пакетов»);

• требования к оформлению документации (включая требования к содержанию разделов, подразделов, пунктов, таблиц и т.д.);

• правила подготовки, рассмотрения, согласования и утверждения документации с указанием предельных сроков для каждой стадии;

• требования к настройке издательской системы, используемой в качестве встроенного средства подготовки документации;

• требования к настройке CASE-средств для обеспечения подготовки документации в соответствии с установленными правилами.

Стандарт интерфейса конечного пользователя с системой. Он должен регламентировать:

• правила оформления экранов (шрифты и цветовая палитра), состав и расположение окон и элементов управления;

• правила использования клавиатуры и мыши;

• правила оформления текстов помощи;

• перечень стандартных сообщений;

• правила обработки реакции пользователя.

6.2. Выработка требований при разработке ПС

При проектировании можно выделить 3 группы программных проектов:

1) управляемые пользователем – все требования выдвигаются самим пользователем;

2) утверждаемые пользователем – совокупность требований разрабатывается проектировщиками или проектировщиками совместно с организацией-пользователем, которая утверждает эти требования;

3) независимые от пользователя – требования полностью определяются и утверждаются организацией-разработчиком, которая несет полную ответственность за работу.

ПИ должно разрабатываться небольшой группой лиц, которые могут и правомочны принимать окончательные решения. В проекте должны участвовать представитель от организации-заказчика, наделенный правом принятия решения и представитель, который непосредственно будет применять ПИ. От организации-разработчика участвует специалист, который будет играть главную роль в процессе внешнего проектирования ПИ, и специалист, который будет принимать участие во внутреннем проектировании.

6.2.1. Определение требований к ПС

Требования к разрабатываемому программному средству должны однозначно определять конечный продукт разработки и методы разработки.

В процессе разработки требований необходимо:

1. выявить наличие информации, необходимой для выполнения планируемых функций;

2. определить трудоемкость и стоимость предстоящей работы;

3. обеспечить полноту и точность определения функций, подлежащих выполнению ПИ, и их взаимосвязь;

4. выявить пространственно-временные ограничения, налагаемые на систему, а также средства системы, которые в будущем могут претерпеть изменения.

Можно установить две фазы в выработке требований:

1) Планирования. На этой фазе в результате взаимодействия пользователей и разработчиков определяется реализуемость, устанавливаются цели, оцениваются затраты и обеспечивается ориентация для разработки проекта.

2) Фаза выработки требований пользователя. Вырабатываются требования к входным данным, информационным потокам, выходным данным, документации, среде и вычислительным ресурсам.

Результатом этого этапа является документ, включающий в себя цели ПИ. По этому документу можно определить преимущества и недостатки ПИ для пользователя, состав и конфигурации ресурсов для его работы. Этот документ должен быть достаточно полным. За требования к ПИ отвечает пользователь. Проектировщик отвечает за качество описания требований и за их реализацию.

6.2.2. Определение целей создания ПС

Вторым процессом начального этапа проектирования ПИ являются разработка и описание целей. На этом этапе устанавливаются взаимосогласованные цели создания ПИ.

Все цели могут быть сгруппированы в 10 категорий:

1) универсальность;
2) человеческий фактор;
3) адаптивность;
4) сопровождение;
5) безопасность;
6) документация;
7) стоимость;
8) календарный план;
9) эффективность;
10) надежность.

Одни цели согласуются между собой (человеческий фактор и надежность), другие вступают в противоречие (надежность и стоимость).

Целью является принять компромиссное решение, т.е. выявить какие цели наиболее важны, а какими можно пренебречь.

Цели ПИ с точки зрения пользователя включают следующую информацию:

1) краткое описание. Определяется общее назначение разрабатываемого ПИ и его функции;

2) определение пользователя. Описывается круг возможных пользователей, характеризуются специфические особенности отдельных групп пользователей;

3) подробное описание функциональных задач. Согласовывает задачи между пользователем и проектировщиком;

4) документация. Определяются типы документации и круг пользователей документации;

5) эффективность. Описываются цели, касающиеся производительности: временные характеристики, пропускная способность, использование ресурсов и т.д.;

6) совместимость. Указываются стандарты, которым необходимо следовать в процессе разработки, а также другие программные продукты, с которыми разрабатываемое ПИ должно быть совместимо;

7) конфигурация. Определяются различные конфигурации технических и программных средств, в среде которых может работать проектируемое ПИ;

8) безопасность. Формируются цели в отношении обеспечения безопасности ПИ;

9) обслуживание. Описываются цели по затратам и времени исправления ошибок, а также функции для достижения этих целей;

10) установка. Описываются методы и средства настройки ПИ на конкретные условия эксплуатации;

11) надежность. Процесс восстановления и последствий от сбоя.

Однако можно определить некоторые общие вопросы, которые должны быть рассмотрены.

Цели проекта – цели, которые должны быть достигнуты в процессе проектирования.

Цели проекта должны содержать следующую информацию:

1) стоимостные ограничения;
2) календарный план выполнения работ;
3) задачи каждого этапа тестирования;
4) цели в области адаптируемости. Указывают степень адаптируемости или расширяемости, которая должна быть достигнута;
5) цели в области сопровождаемости;
6) уровни надежности каждого этапа;
7) документирование;
8) критерий завершенности.

Цели проекта должны быть ясными, обоснованными и измеримыми, а также известными как пользователям, так и разработчикам. Они должны быть реальными и по возможности выражаться в количественном измерении для последующей проверки достижения поставленных целей.

Контрольные вопросы

1. Какие задачи должны быть решены в процессе разработки требований к проектируемому ПО?
2. Какие этапы включает стадия формирования требований к ПО?
3. Какие модели строятся на стадии выработки требований?
4. Каким образом определяются метод и технология проектирования ПО?
5. Какие стандарты необходимы для выполнения конкретного проекта?
6. Каковы цели проектирования ПС и как их нужно ставить?
7. Как соотносятся цели проектирования ПС со стороны проектировщика и заказчика?

7. СТРУКТУРНЫЙ ПОДХОД К РАЗРАБОТКЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

7.1. Проблема сложности больших систем

Проблема сложности является главной проблемой, которую приходится решать при создании больших и сложных систем любой природы, в том числе и ЭИС. Ни один разработчик не в состоянии выйти за пределы человеческих возможностей и понять всю систему в целом. Единственный эффективней подход к решению этой проблемы, который выработало человечество за всю свою историю, заключается в построении сложной системы из небольшого количества крупных частей, каждая из которых, в свою очередь, строится из частей меньшего размера и т.д., до тех пор, пока самые небольшие части можно будет строить из имеющегося материала. Этот подход известен под самыми разными названиями, среди них такие, как «разделяй и властвуй», иерархическая декомпозиция и др. По отношению к проектированию сложной программной системы это означает, что ее необходимо разделять (декомпозировать) на небольшие подсистемы, каждую из которых можно разрабатывать независимо от других. Это позволяет при разработке подсистемы любого уровня держать в уме информацию только о ней, а не обо всех остальных частях системы. Правильная декомпозиция является главным способом преодоления сложности разработки больших систем ПО. Понятие «правильная» по отношению к декомпозиции означает следующее:

- количество связей между отдельными подсистемами должно быть минимальным;
- связность отдельных частей внутри каждой подсистемы должна быть максимальной.

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

- каждая подсистема должна инкапсулировать свое содержимое (скрывать его от других подсистем);
- каждая подсистема должна иметь четко определенный и интерфейс с другими подсистемами.

Инкапсуляция позволяет рассматривать структуру каждой подсистемы независимо от других подсистем. Интерфейсы позволяют строить систему более высокого уровня, рассматривая каждую подсистему как единое целое и игнорируя ее внутреннее устройство.

На сегодняшний день в программной инженерии существуют два основных подхода к разработке ПО ЭИС, принципиальное различие между которыми обусловлено разными способами декомпозиции сис-

тем. Первый подход называют функционально-модульным или структурным. В его основу положен принцип функциональней декомпозиции, при которой структура системы описывается в терминах иерархии ее функций к передачи информации между отдельными функциональными элементами. Второй, объектно-ориентированный подход использует объектную декомпозицию. При этом структура системы описывается в терминах объектов и связей между ними, а поведение системы описывается в терминах обмена сообщениями между объектами.

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

Все наиболее распространенные методы структурного подхода базируются на ряде общих принципов. Базовыми принципами являются:

- принцип «разделяй и властвуй»;
- принцип иерархического упорядочения – принцип организации составных частей системы в иерархические древовидные структуры с добавлением новых деталей на каждом уровне.

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

- принцип абстрагирования выделение существенных аспектов системы и отвлечение от несущественных;
- принцип непротиворечивости – обоснованности согласованность элементов системы;
- принцип структурирования данных – данные должны быть структурированы и иерархически организованы.

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

- DFD(Data Flow Diagrams) – диаграммы потоков данных;
- SADT(Structured Analysis and Design technique – метод структурного анализа и проектирования), – модели и соответствующие функциональные диаграммы;

• ERD (Entity-Relationship Diagrams) диаграммы «сущность–связь». Диаграммы потоков данных и диаграммы «сущность–связь» – наиболее часто используемые в CASE-средствах виды моделей.

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

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

На стадии проектирования DFD используются для описания структуры проектируемой системы ПО, при этом они могут уточняться, расширяться и дополняться новыми конструкциями. Аналогично ERD уточняются и дополняются новыми конструкциями, описывающими представление данных на логическом уровне, пригодном для последующей генерации схемы базы данных. Данные модели могут дополняться диаграммами, отражающими системную архитектуру ПО, структурные схемы программ, иерархию экранных форм и меню и др.

Перечисленные модели в совокупности дают полное описание ПО ЭИС независимо от того, является ли система существующей или вновь разрабатываемой. Состав диаграмм в каждом конкретном случае зависит от сложности системы и необходимой полноты ее описания.

7.2. Метод функционального моделирования SADT

Метод SADT разработан Дугласом Россом (SoftTech, Inc.) в 1973т. 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 стандарта IDEFO (Icam DEFinitioq) – подмножества SADT, являющегося основной частью программы ICAM (Integrated Computer Aided Manufacturing – интегрированная компьютеризация производства), проводимой по инициативе ВВС США. IDEFO был утвержден в качестве федерального стандарта США.

Метод SADT 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 действиям. Основные элементы этого метода основываются на следующих концепциях:

- графическое представление блочного моделирования. Графика блоков и дуг SADT-диаграммы отображает функцию в виде блока, а интерфейсы входа-выхода представляются дугами, соответственно входящими в блок и выходящими из него. Взаимодействие блоков друг с другом описывается посредством интерфейсных дуг, выражающих «ограничения», которые, в свою очередь, определяют, когда и каким образом функции выполняются и управляются;
- строгость и точность. Выполнение правил SADT требует достаточной строгости и точности в то же время чрезмерных ограничений на действия аналитика. Правила SADT включают:

– ограничение количества блоков на каждом уровне декомпозиции (правило 3-6 блоков),

– связность диаграмм (номера блоков),

– уникальность меток и наименований (отсутствие повторяющихся имен),

– синтаксические правила для графики (блоков и дуг)

– разделение входов и управлений (правило определения роли данных);

- отделение организации от функции, т.е. исключение влияния административной структуры организации на функциональную модель.

7.2.1. Состав функциональной модели

Результатом применения метода SADT является модель, которая состоит из диаграмм, фрагментов текстов и глоссария; имеющих ссылки друг на друга. Диаграммы – главные компоненты модели, все функции организации и интерфейсы на них представлены как блоки и дуги соответственно. Место соединения дуги с блоком определяет тип интерфейса. Управляющая информация входит в блок сверху, в то время как входная информация, которая подвергается обработке, показана с левой стороны блокада результаты (выход) показаны с правой стороны. Механизм (человек ила автоматизированная система), который осуществляет операцию, представляется дугой, входящей в блок снизу (рис. 9).

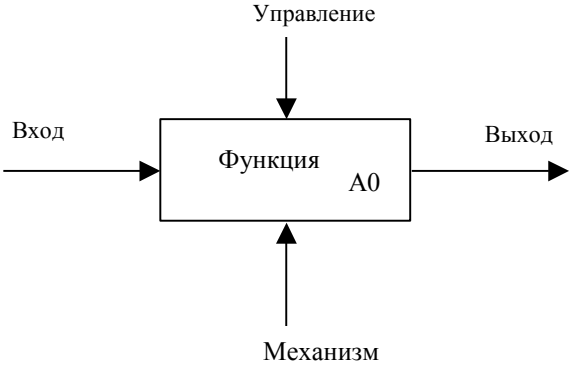


Рис. 9. Функциональный блок и интерфейсные дуги

Одной из наиболее важных особенностей метода SADT является постепенное введение все больших уровней детализации по мере создания диаграмм, отображающих модель.

7.2.2. Построение иерархии диаграмм

Построение SADT-модели начинается с представления всей системы в виде простейшего компонента – одного блока и дуг, изображающих интерфейсы с функциями вне системы. Поскольку единственный блок отражает систему как единое целое, имя, указанной в блоке, является общим. Это верно и для интерфейсных дуг они также соответствуют полному набору внешних интерфейсов системы в целом.

Затем блок, который представляет систему в качестве единого модуля, детализируется на другой диаграмме с помощью нескольких блоков, соединенных интерфейсными дугами. Эти блоки определяют основные подфункции исходной функции. Данная декомпозиция выявляет полный набор подфункций, каждая из которых показана как блок, границы которого определены интерфейсными дугами. Каждая из этих подфункций может быть декомпозирована подобным образом в целях большей детализации.

Во всех случаях каждая подфункция; может содержать только те элементы, которые входят в исходную функцию. Кроме того модель не может опустить, какие-либо элементы т.е. как уже отмечалось, родительский блок и его интерфейсы обеспечивают контекст. К нему нельзя ничего добавить, и из него не может быть ничего удалено.

Модель SADT представляет собой серию диаграмм с сопроводительной документацией, разбивающих сложный объект на составные части, которые изображены в виде блоков. Детали каждого из основных блоков показаны в виде блоков на других диаграммах. Каждая деталь-

ная диаграмма является декомпозицией блока из диаграммы предыдущего уровня. На каждом шаге декомпозиции диаграмма предыдущего уровня называется родительской для более детальной диаграммы.

Синтаксис диаграмм определяется следующими правилами:

- диаграммы содержат блоки и дуги;
- блоки представляют функции:
- блоки имеют доминирование (выражающееся в их ступенчатом расположении, причем доминирующий блок располагается в верхнем левом углу диаграммы);
- дуги изображают наборы объектов, передаваемых между блоками;
- дуги изображают взаимосвязи между блоками:

– выход-управление;
– выход-вход;
– обратная связь по управлению;
– обратная связь по входу;
– выход-механизм

Дуги, входящие в блок и выходящие из него на диаграмме верхнего уровня, являются точно теми же самыми, что и дуги, входящие в диаграмму нижнего уровня и выходящие из нее, потому что блок и диаграммы изображают одну и ту же часть системы.

Некоторые дуги присоединены к блокам диаграммы обоими концами, у других же один конец остается неприсоединенным. Неприсоединенным дуги соответствуют входам, управлениям и выходам родительского блока. Источник или получатель этих пограничных дуг может быть обнаружен только на родительской диаграмме. Неприсоединенным концы должны соответствовать дугам на исходной диаграмме. Все граничные дуги должны продолжаться на родительской диаграмме, чтобы она была полной и непротиворечивой.

На SADT-диаграммах не указаны явно ни последовательность, ни время. Обратные связи, итерации, продолжающиеся процессы и перекрывающиеся (по времени) функции могут быть изображены с помощью дуг. Обратные связи могут выступать в виде комментариев, замечаний, исправлений и т.д.

Как было отмечено, механизмы (дуги с нижней стороны) показывают средства, с помощью которых осуществляется выполнение функций. Механизм может быть человеком, компьютером или любым другим устройством, которое помогает выполнять данную функцию.

Каждый блок на диаграмме имеет свой номер. Блок любой диаграммы может быть описан диаграммой нижнего уровня, которая, в свою очередь может быть далее детализирована с помощью необходимого числа диаграмм. Таким образом формируется иерархия диаграмм.

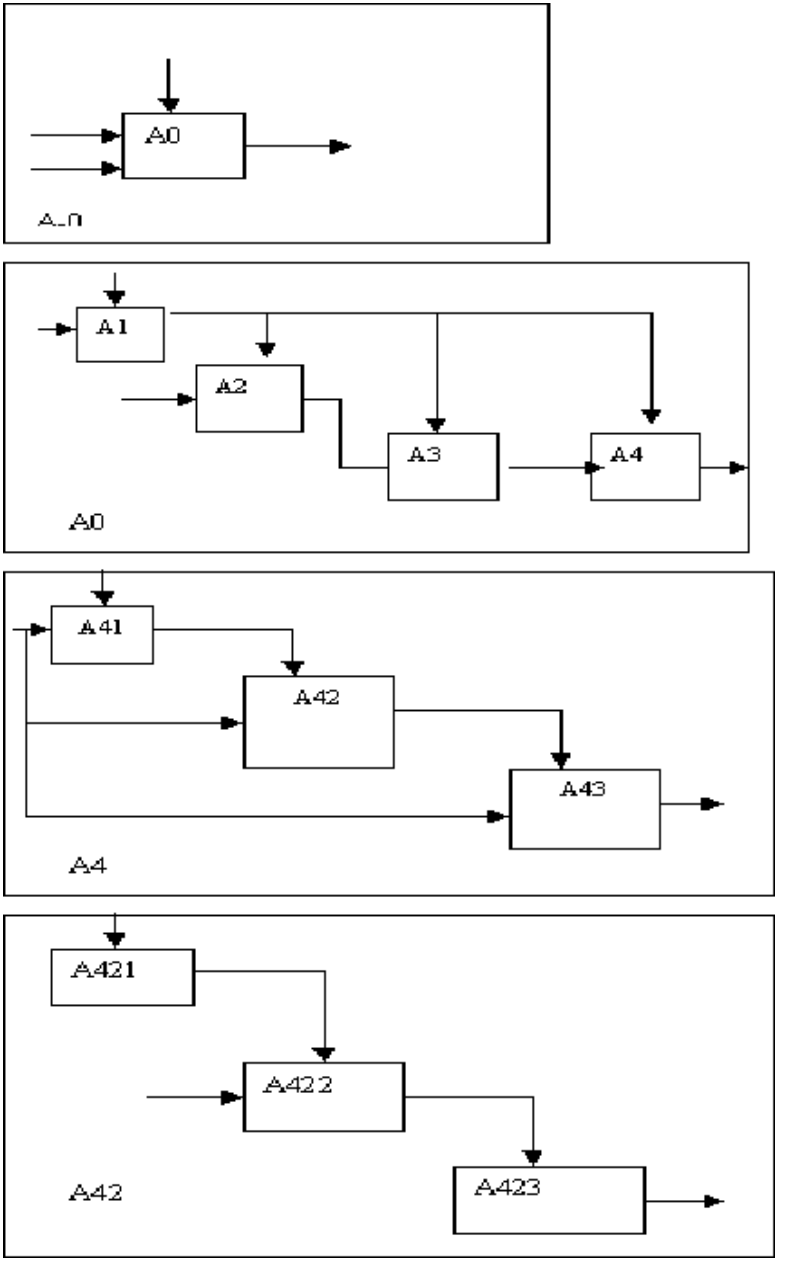


Рис. 10. Структура SADT модели. Декомпозиция диаграмм

Для того чтобы указать положение любой диаграммы или блока и иерархии, используются номера диаграмм, Например А21 является диаграммой, которая детализирует блок А21 на диаграмме А2. Аналогично диаграмма А2 детализирует блок А2 на диаграмме АО, которая является самой верхней диаграммой модели.

7.2.3. Типы связей между функциями

Одним из важных моментов при моделировании бизнес-процессов организации с помощью метода SADT является точная согласованность типов связей между функциями. Различают по крайней мере связи семи типов (в порядке возрастания их относительной значимости):

- случайная;
- логическая;
- временная;
- процедурная;
- коммуникационная;
- последовательная;
- функциональная.

Случайная связь – показывает, что конкретная связь между функциями незначительна или полностью отсутствует. Это относится к ситуации, когда имена данных на SADT-дугах в одной диаграмме имеют слабую связь друг с другом (рис. 11).

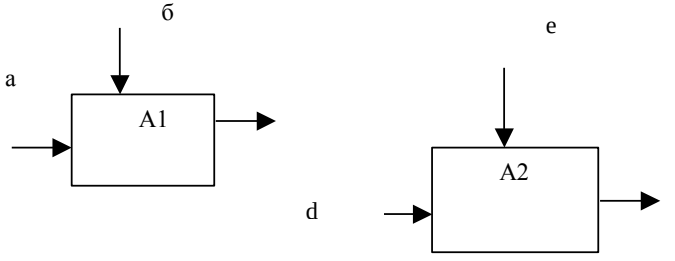


Рис. 11. Случайная связь

Логическая связь – данные и функции собираются вместе благодаря тому, что они попадают в общий класс или набор элементов, но необходимых функциональных отношений между ними не обнаруживается.

Временная связь – представляет функции, связанные во времени, когда данные используются одновременно или функции включаются параллельно, а не последовательно.

Процедурная связь (рис. 12) – функции, сгруппированы вместе благодаря тому, что они выполняются в течение одной и той же части цикла или процесса.

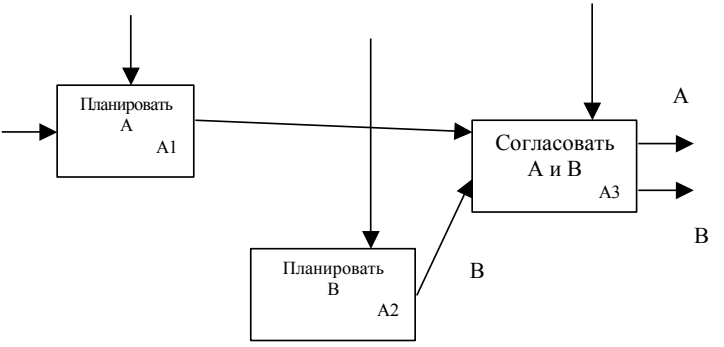


Рис. 12. Процедурная связь

Коммуникационная – функции группируются благодаря тому, что они используют одни и те же входные данные и/или производят одни и те же выходные данные (рис. 13).

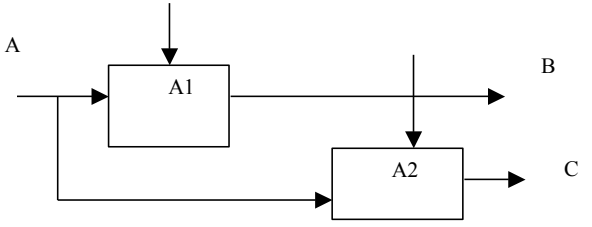


Рис. 13. Коммуникационная связь

Последовательная связь – выход одной функции служит входными данными для следующей функции. Связь между элементами на диаграмме является более тесной, чем в рассмотренных выше случаях, поскольку моделируются причинно-следственные зависимости (рис. 14).

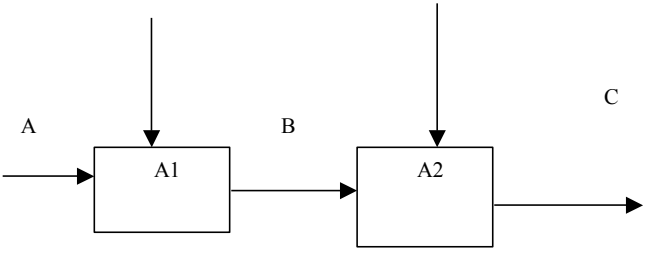


Рис. 14. Последовательная связь

Функциональная связь – все элементы функции влияют на выполнение одной и только одной функции. Диаграмма являющаяся чисто функциональной, не содержит чужеродных элементов, относящихся к последовательному или более слабому типу связи. Одним из способов определения функционально-связанных диаграмм является рассмотрение двух блоков, связанных через управляющие дуги, как показано на рис. 15.

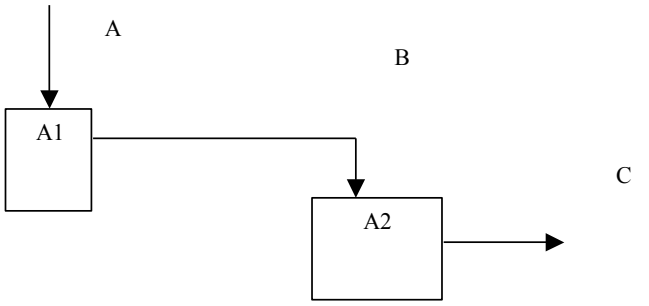


Рис. 15. Функциональная связь

В математических терминах необходимое условие для простейшего типа функциональной связи (рис. 15) имеет следующий вид:

$$C=g(B)=g(f(A))$$

В табл.1 представлены все типы связей, рассмотренные выше. Важно отметить, что уровни 4–6 устанавливают типы связей, которые разработчики считают важнейшими для получения диаграмм хорошего качества.

Таблица 1

Описание типов связей

Уровень значимости	Тип связи	Характеристика типа связи	
		Для функций	Для данных
1	2	3	4
0	Случайная	Случайная	Случайная
1	Логическая	Функции одного и того же множества или типа (например, «редактировать все входы»)	Данные одного и того же множества или типа.

Окончание табл. 1

1	2	3	4
2	Временная	Функции одного и того же периода времени (например, «операции инициализации»)	Данные, используемые в каком-либо временном интервале
3	Процедурная	Функции, работающие в одной и той же фазе или итераций (например, «первый проход компилятора»)	Данные, используемые во время одной и той же фазы или итерации
4	Коммуникационная	Функции, использующие одни и те же данные	Данные, на которые воздействует одна и та же деятельность
5	Последовательная	Функции, выполняющие последовательные преобразования одних и тех же данных	Данные, преобразуемые последовательными функциями
6	Функциональная	Функции, объединяемые для выполнения одной функции	Данные, связанные с одной функцией

7.2. Моделирование потоков данных

Диаграммы потоков данных (DFD) являются основным средством моделирования функциональных требований проектируемой системе. С их помощью эти требования представляются в виде иерархии функциональных компонентов (процессов), связанных потоками данных. Главная цель такого представления – продемонстрировать, как каждый процесс преобразует свои входные данные в выходные, а также выявить отношения между этими процессами.

Диаграммы потоков данных известны очень давно. В фольклоре упоминается следующий пример использования DFD для реорганизации переполненного клерками офиса, относящийся к 20-м гг. Осуществлявший реорганизацию консультант обозначил кружком каждого клерка, а стрелкой – каждый документ, передаваемый между ними. Используя такую диаграмму он предложил схему реорганизации, в соот-

ветствий с которой два клерка, обменивающихся множеством документов, были посажены рядом, а клерки с малым взаимодействием были посажены на большом расстоянии друг от друга. Так появилась первая модель, представляющая собой потоковую диаграмму т предвестника DFD.

Для построения DFD традиционно используются две различные нотации, соответствующие методам Йордана-Сэрсона. Эти нотации незначительно отличаются друг от друга графическим изображением символов. Далее при построении примеров будет использоваться нотация Геина – Сэрсона.

В соответствии с данными методами модель системы определяется как иерархия диаграмм потоков данных, описывающих асинхронный процесс преобразования информации от ее входа в систему до выдачи пользователю. Диаграммы верхних уровней иерархии (контекстные диаграммы) определяют основные процессы или подсистемы с внешними входами и выходами. Они детализируются при помощи диаграмм нижнего уровня. Такая декомпозиция продолжается, создавая многоуровневую иерархию диаграмм, до тех пор, пока не будет достигнут уровень декомпозиции, на котором процессы становятся элементарными и детализировать их далее невозможно.

Источники информации (внешние сущности) порождает информационные потоки (потоки данных), переносящие информацию к подсистемам или процессам. Те, в свою очередь, преобразуют информацию и порождают новые потоки, которые переносят информацию к другим процессам или подсистемам, накопителям данных или {внешним сущностям – потребителям информации.

7.2.1. Состав диаграмм потоков данных

Основными компонентами диаграмм потоков данных являются:

- внешние сущности;
- системы и подсистемы;
- процессы;
- накопители данных;
- потоки данных
- канал данных.

Внешняя сущность представляет собой материальный объект или физическое лицо, представляющие собой источник или приемник информации, например заказчики, персонал, поставщики, клиенты, склад. Определение некоторого объекта или системы в качестве внешней сущности указывает на то, что они находятся за пределами границ анализируемой системы. В процессе анализа некоторые внешние сущности могут быть перенесены внутрь диаграммы анализируемой системы если

эго необходимо, или, наоборот, часть процессов может быть вынесена за пределы диаграммы и представлена как внешняя сущность.

Внешняя сущность обозначается квадратом (рис. 16), расположенным как бы над диаграммой и бросающим на нее тень для того, чтобы можно было выделить этот символ среди других обозначений.

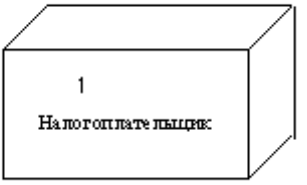
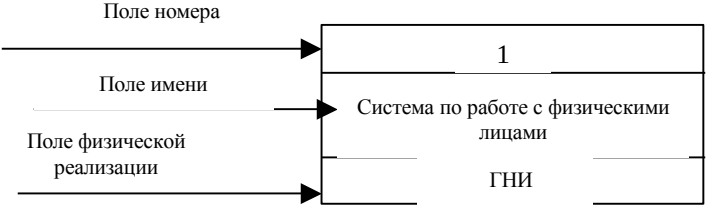


Рис. 16. Графическое изображение внешней сущности

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



Подсистема по работе с физическими лицами
(ГНИ – Государственная налоговая инспекция)

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

Процесс представляет собой преобразование входных потоков данных в выходные в соответствии с определенным алгоритмом. Физически процесс может быть реализован различными способами: это может быть подразделение организации (отдел), выполняющий обработку входных документов и выпуск отчетов, программа, аппаратно реализованное логическое устройство и т.д. Процесс на диаграмме потеков данных изображен на рис. 17.

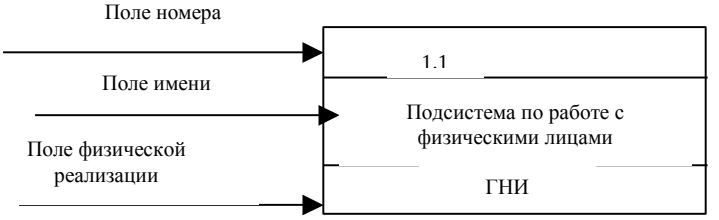


Рис. 17. Графическое изображение процесса

Номер процесса служит дли его идентификации; В поле имени вводится наименование процесса в виде предложения с активным недвусмысленным глаголом в неопределенной форме (вычислить, рассчитать, проверить, определить, создать, получить), за которым следуют существительные в винительном падеже, например: «Ввести сведения о налогоплательщиках», «Выдать информацию о текущих расходах», «Проверить поступление денег».

Использование таких глаголов, как «обротать», «модернизировать» или «отредактировать», означает недостаточно глубокое понимание данного процесса и требует дальнейшего анализа.

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

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

Накопитель данных может быть реализован физически в виде микрофиши, ящика в картотеке, таблицы в оперативной памяти, файла на магнитном носителе и т.д. Накопитель данных на диаграмме потоков данных (рис. 18) идентифицируется буквой «D» и произвольным числом. Имя накопителя выбирается из соображения наибольшей информативности для проектировщика.

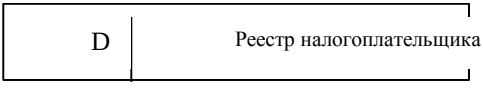


Рис. 18. Графическое изображение накопителя данных

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

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

Поток данных на диаграмме изображается линией, оканчивающейся стрелкой, которая показывает направление, потока (рис. 19). Каждый поток данных имеет имя, отражающее его содержание.

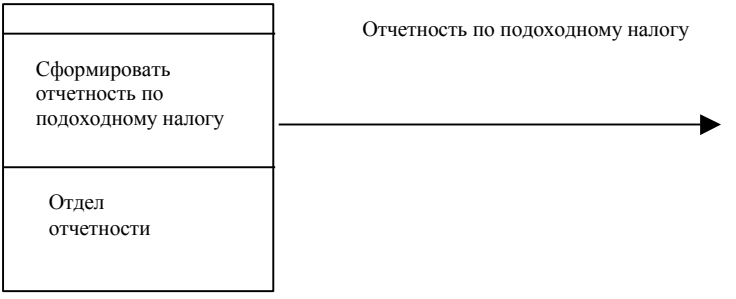


Рис. 19. Поток данных

7.2.2. Построение иерархии диаграмм потоков данных

Главная цель построения иерархии DFD заключается в том, чтобы сделать требования к системе ясными и понятными на каждом уровне детализации, а также разбить эти требования на части с точно определенными отношениями между ними. Для достижения этого целесообразно пользоваться следующими рекомендациями:

- размещать на каждой диаграмме от 3 до 7 процессов. Верхняя граница соответствует человеческим возможностям одновременного восприятия и понимания структуры сложной системы с множеством внутренних связей, нижняя граница выбрана по соображениям здравого смысла: нет необходимости детализировать процесс диаграммой, содержащей всего один процесс или два;
- не загромождать диаграммы не существенными на данном уровне деталями;
- декомпозицию потоков данных осуществлять параллельно с декомпозицией процессов. Эти две работы должны выполняться одновременно, а не одна после завершения другой;
- выбирать ясные, отражающие суть дела имена процессов и потоков, при этом стараться не использовать аббревиатуры.

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

Для проверки контекстной диаграммы можно составить список событий. Список событий должен состоять из описаний действий внешних сущностей (событий) и соответствующих реакций системы на события. Каждое событие должно соответствовать одному (или более) потоку данных: входные потоки интерпретируются как воздействия, а выходные потоки – как реакции системы на входные потоки.

Если для сложной системы ограничиться единственной контекстной диаграммой, то она будет содержать слишком большое количество источников и приемников информации, которые трудно расположить на листе бумаги нормального формата, и, кроме того, единственный главный процесс не раскрывает структуры такой системы.

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

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

После построения контекстных диаграмм полученную модель следует проверить на полноту исходных данных об объектах системы и изолированность объектов (отсутствие информационных связей с другими объектами).

Для каждой подсистемы, присутствующей на контекстных диаграммах, выполняется ее детализация при помощи DFD. Это можно сделать путем построения диаграммы для каждого события. Каждое, событие представляется в виде процесса с соответствующими входными и выходными потоками, накопителями данных, внешними сущностями и ссылки на другие процессы для описания связей между этим

процессом его окружением. Затем все построенные диаграммы сводятся в одну диаграмму нулевого уровня.

Каждый процесс на DFP, в свою очередь, может быть детализирован при помощи DFD или (если процесс элементарный) спецификации. При детализации должны выполняться следующие правила:

- правило балансировки – при детализации подсистемы или процесса детализирующая диаграмма в качестве внешних источников или приемников данных может иметь только те компоненты (подсистемы, процессы, внешние сущности, накопители данных), с которыми имеют информационную связь детализируемые подсистема или процесс на родительской диаграмме;
- правило нумерации – при детализации процессов должна поддерживаться их иерархическая нумерация. Например, процессы, детализирующие процесс с номером 12, получают номера 12.1, 12.2, 12.3 и т.д.

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

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

- наличия у процесса относительно небольшого количества входных и выходных потоков данных (2–3 потока);
- возможности описания преобразования данных процессом в виде последовательного алгоритма;
- выполнения процессом единственной логической функции преобразования входной информации в выходную;
- возможности описания логики процесса при помощи спецификации небольшого объема (не более 20–30 строк).

Спецификации должны удовлетворять следующим требованиям:

- для каждого процесса нижнего уровня должна существовать одна и только одна спецификация;
- спецификация должна определять способ преобразования входных потоков в выходные;
- нет необходимости (по крайней мере на стадии формирования требований) определять метод реализации этого преобразования;
- спецификация должна стремиться к ограничению избыточности не следует переопределять то, что уже было определено на диаграмме;
- набор конструкций для построения спецификаций должен быть простым и понятным.

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

и/или имя процесса, списки входных и выходных данных и тело (описание) процесса, являющееся спецификацией алгоритма или операции, трансформирующей входные потоки данных 8 выходные. Известно большое количество разнообразных методов, позволяющих описать тело процесса. Соответствующие этим методам языки могут варьироваться от структурированного естественного языка или псевдокода до визуальных языков проектирования.

Структурированный естественный язык применяется для читабельного, достаточно строгого описания спецификаций процессов. Он представляет собой разумное сочетание строгости языка программирования и читабельности естественного языка и состоит из подмножества слов, организованных в определенные логические структуры, арифметических выражений и диаграмм.

В состав языка входят следующие основные символы:

- глаголы, ориентированные на действие и применяемые к объектам;
- термины, определенные на любой стадии проекта ПО (например, задачи, процедуры, символы данных и т.п.);
- предлоги и союзы, используемые в логических отношениях;
- общеупотребительные математические, физические и технические термины;
- арифметические уравнения;
- таблицы, диаграммы, графы и т.п. комментарии.

К управляющим структурам языка относятся последовательная конструкция, конструкция выбора и итерация (цикл).

При использовании структурированного естественного языка приняты следующие соглашения:

- логика процесса выражается в виде комбинации последовательных конструкций, конструкций выбора и итераций;
- глаголы должны быть активными, недвусмысленными и ориентированными на целевое действие (заполнить, вычислить, извлечь, а не модернизировать, обработать);
- логика процесса должна быть выражена четко и недвусмысленно.

При построении иерархии DFD переходить к детализации процессов следует только после определения содержания всех потоков и накопителей данных, которое описывается с помощью структур данных. Для каждого потока данных формируется список всех его элементов данных, затем элементы данных объединяются в структуры данных, соответствующие более крупным объектам данных (например строкам документов или объектам предметной области).

Каждый объект должен состоять из элементов, являющихся его атрибутами. Структуры данных могут содержать альтернативные, условные вхождения и итерации. Условное вхождение показывает, что дан-

ный компонент может отсутствовать в структуре (например, структура «данные о страховании» для объекта «служащий»). Альтернатива означает, что в структуру может входить один из перечисленных элементов. Итерация предусматривает вхождение любого числа элементов в указанном диапазоне (например, элемент «имя ребенка» для объекта «служащий»). Для каждого элемента данных может указываться его тип (непрерывные или дискретные данные). Для непрерывных данных могут указываться единица измерения (кг, см и т.п.), диапазон значений, точность представления и форма физического кодирования. Для дискретных данных может указываться таблица допустимых значений.

После построения законченной модели системы ее необходимо верифицировать (проверить на полноту и согласованность). В полной модели все ее объекты (подсистемы, процессы, потоки данных) должны быть подробно описаны и детализированы. Выявленные недетализированные объекты следует детализировать, вернувшись на предыдущие шаги разработки. В согласованной модели для всех потоков данных и накопителей данных должно выполняться правило сохранения информации: все поступающие куда-либо данные должны быть считаны, а все считываемые данные должны быть записаны.

7.3. Сравнительный анализ SADT-моделей и диаграмм потоков данных

Как уже отмечалось, практически во всех методах структурного подхода (структурного анализа)| на стадии формирования требований к ПО используются две группы средств моделирования:

- диаграммы, итерирующие функции, которые система должна выполнять, и связи между этими функциями – DFD или SADT (IDEFO);
- диаграммы, моделирующие данные и их отношения (ERD).

Таким образом, наиболее существенное различие между разновидностями структурного анализа заключается в средствах функционального моделирования. С этой точки зрения все разновидности структурного анализа могут быть разбиты на две группы – использующие DFD (в различных нотациях) и использующие SADT-модели. Соотношение применения этих двух разновидностей структурного составляет 90% для DFD и 10% для SADT. Вероятно, соотношение такого же порядка справедливо и для распространенности рассматриваемых моделей на практике.

Сравнительный анализ этих двух разновидностей методов структурного анализа проводится по следующим параметрам:

- адекватность средств решаемым задачам;
- согласованность с другими средствами структурного анализа;
- интеграция с последующими стадиями ЖЦ ПО (прежде всего со стадией проектирования).

Адекватность средств решаемым задачам. Модели SADT (IDEFQ) традиционно используются для моделирования организационных систем. С другой стороны, не существует никаких принципиальных ограничении на использование DFD в качестве средства построения статических моделей деятельности организаций. 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. Например, в Министерстве обороны США десятки лет существуют четкие должностные инструкции и методики, которые жестко регламентируют деятельность подразделений, делают ее высокотехнологичной и ориентированной на бизнес-процесс. В российской действительности с ее слабой типизацией бизнес-процессов, их стихийным появлением и развитием разумнее ориентироваться на модели, основанные на потоковых диаграммах.

Если же речь идет не о системах вообще, а о ЭИС, то здесь DFD вне конкуренции. Практически любой класс систем успешно моделируется при помощи DFD-ориентированных методов. SADT-диаграммы оказываются значительно менее выразительными и, удобными при моделировании ЭИС, Так, дуги в SADT жестко типизированы (вход, выход, управление, механизм). В то же время применительно к ЭИС стирается смысловое различие между входами и выходами, с одной стороны, и управлениями и механизмами, с другой – входы, выходы и управления являются потоками данных и правилами их преобразования. Анализ системы с помощью потоков данных и процессов, их преобразующих, является более прозрачным и недвусмысленным.

Более того, в SADT вообще отсутствуют выразительные средства для моделирования особенностей ЭИС. DFD же с самого начала создавались как средство проектирования информационных систем (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).

Наличие в DFD спецификаций процессов нижнего уровня позволяет преодолеть логическую незавершенность SADT (а именно обрыв модели на некотором достаточно низком уровне, когда дальнейшая ее детализация становится бессмысленной) и построить полную функциональную спецификацию разрабатываемой системы,

Согласованность с другими средствами структурного анализа. Главным достоинством любых моделей является возможность их интеграции с моделями других типов. В данном случае речь идет о согласованности функциональных моделей со средствами моделирования дан-

ных. Согласование SADT-модели с ERD практически невозможно или носит искусственный характер, В свою очередь, DFD и ERD взаимно дополняют друг друга и являются согласованными, поскольку в DFD присутствует описание структур данных, непосредственно используемое для построения ERD.

Интеграция с последующими стадиями ЖЦ ПО. Важная характеристика модели – ее совместимость с моделями последующих стадий ЖЦ (прежде всего стадии проектирования, непосредственно следующей за стадией формирования требований и опирающейся на её результаты).

В заключение необходимо отметить, что одним из основных критериев выбора того или иного метода являемся 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.

Контрольные вопросы

1. В чем заключаются основные принципы структурного подхода?
2. Какой стандарт на основе метода SADT был принят как федеральный стандарт США?
3. Чем определяются интерфейсы между функциями в модели SADT?
4. Что общего и в чем различия между методом SADT и моделированием потоков данных?
5. В чем заключаются достоинства и недостатки структурного подхода?

8. ПРОЕКТИРОВАНИЕ И ПРОГРАММИРОВАНИЕ ПС

Технологический цикл проектирования ПС включает три процесса анализ, синтез и сопровождение. В процесс анализа вырабатываются требования к системе. Здесь закладывается фундамент всего проекта, ведь из-за неполноты и неточности в определении требований потерпели неудачи множество проектов. В процессе синтеза формируется ответ на вопрос: «Каким образом система будет реализовывать предъявленные к ней требования. Выделяют три этапа синтеза: проектирование, кодирование (программирование) и тестирование.

Рассмотрим информационные потоки процесса разработки программных средств (рис. 20). В процессе анализа формируются информационная, функциональная и поведенческая модели. Они поставляю этапу проектирования исходные данные для работы.

Информационная модель описывает информацию, которую должна обрабатывать система. Функциональная модель определяет перечень функций обработки. Поведенческая модель фиксирует желаемую динамику системы (режимы ее работы). В ходе этапа проектирования происходит разработка данных, разработка архитектуры и процедурная разработка ПС.

Разработка данных – это результат преобразования информационной модели анализа в структуры данных, которые потребуются для реализации ПС

Разработка архитектуры выделяет основные структурные компоненты и фиксирует связь между ними.

Процедурная разработка описывает последовательность действий в структурных компонентах, то есть определяет их содержание

Следующими этапами синтеза являются этапы кодирования ПС и его тестирования. Но следует отметить, что решения принимаемые в ходе проектирования, делают его стержневым этапом процесса синтеза, здесь закладывается качество будущей системы. Проектирование – единственный путь, обеспечивающий правильную трансляцию требований заказчика в конечный программный продукт.

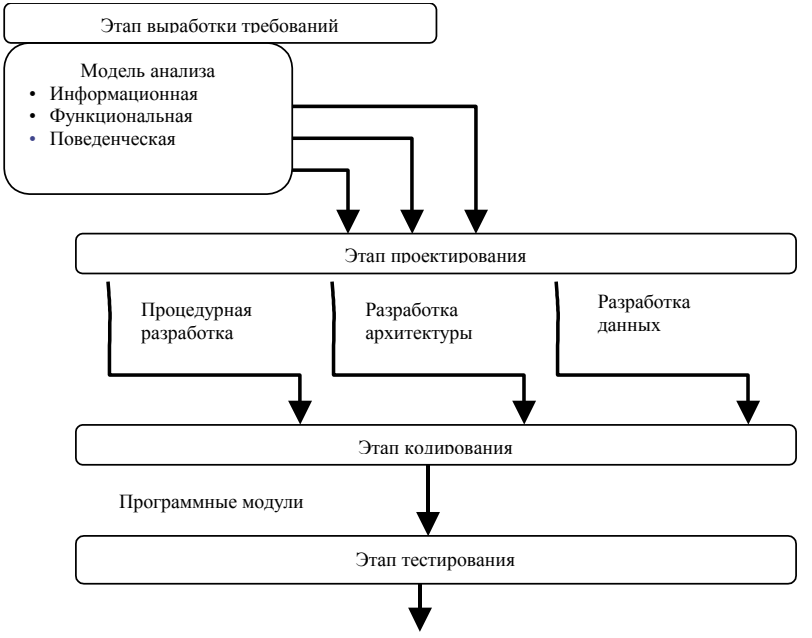


Рис. 20. Схема этапов разработки ПС

8.1. Проектирование ПС

Проектирование – итерационный процесс, при помощи которого требования к ПС преобразуются в инженерные представления ПС.Процесс проектирования включает в себя анализ и декомпозицию задач и данных в соответствии с принятым методом проектирования и завершается построением иерархической схемы, отражающей структурные взаимосвязи между всеми модулями, описанием функций каждого модуля и интерфейса между ними. Таким образом выделяют предварительное проектирование, когда строится архитектура системы, затем детальное проектирование уточняет архитектурные абстракции и параллельно происходит интерфейсное проектирование, где формируется графический интерфейс системы (рис. 21).

Предварительное проектирование включает три типа деятельности:

- Структурирование системы. Система структурируется на несколько подсистем, где под подсистемой понимается независимый программный компонент. Определяются взаимодействия подсистем.
- Моделирование управления. Определяется модель связей управления между частями системы.

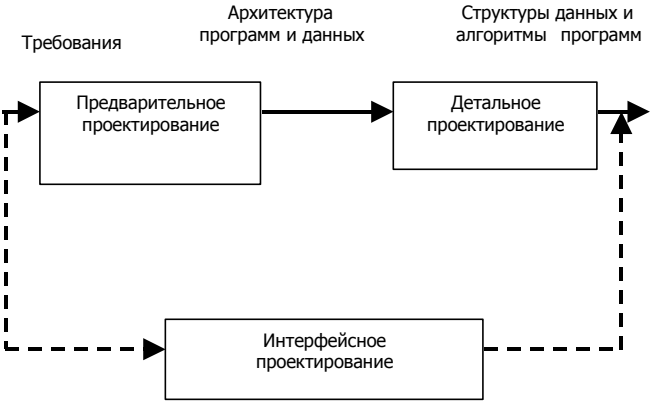


Рис. 21. Информационные потоки процесса разработки ПС

• Декомпозиция подсистем на модули. Каждая подсистема разбивается на модули. Определяются типы модулей и межмодульные соединения.

8.1.1. Модульность программ

ПС создается на основе модульно-иерархической структуры, состоящей из отдельных модулей.

Модуль – отдельная, функционально законченная программная единица, которая может применяться самостоятельно либо быть частью программы.

По определению Г. Майерса, модульность – свойство ПО, обеспечивающее интеллектуальную возможность создания сколь угодно сложной системы.Модуль обладает тремя основными признаками:

• реализует одну или несколько функций, т.е. выполняет какое-то действие;

• имеет определенную логическую структуру, т.е. определяет его внутренний алгоритм (то, как модуль выполняет функцию);

• используется в одном или нескольких контекстах – описывает конкретное использование модуля.

Принцип информационной закрытости (автор – Д. Парнас, 1972) утверждает: содержание модулей должно быть скрыто друг от друга. Модуль должен определяться и проектироваться так, чтобы его содержимое было недоступно тем модулям, которые не нуждаются в такой информации.

Информационная закрытость означает следующее:

• все модули независимы, обмениваются только информацией, необходимой для работы;

• доступ к операциям и структурам данных модуля ограничен.

Достоинства информационной закрытости:обеспечивается возможность разработки модулей различными независимыми коллективами;

- обеспечивается легкая модификация системы.

Идеальный модуль играет роль «черного ящика», содержимое которого невидимо другим модулям, его легко развивать и корректировать в процессе сопровождения программы. Для обеспечения таких возможностей система внутренних и внешних связей модуля должна отвечать особым требованиям. Рассмотрим характеристики внутренних и внешних связей.

Связанность модуля – мера зависимости его частей. Чем выше связанность модуля, тем лучше результат проектирования. Для измерения связности используют понятие силы связности (СС). *Существуют* 7 типов связности.

1. Связность по совпадению (СС=0). В модуле отсутствуют явно выраженные связи. Например повторяющиеся элементы программы вводятся в модуль.

2. Логическая связность (СС=1). Части модуля объединены по принципу функционального подобия. Например, модуль состоит из разных программ обработки ошибок.

3. Временная связанность (СС=3). Части модуля не связаны, но необходимы в один и тот же момент времени. Например, при входе в систему.

4. Процедурная связность (СС=5). Части модуля связаны порядком выполняемых действий, реализующих сценарий поведения. Например, есть функциональная зависимость между переменными, описывается алгоритм выполнения какой-либо задачи.

5. Коммуникативная связность (СС=7). Модуль обладает процедурной связанностью, все его функции связаны через используемые данные. Общая структура данных является основой.

6. Информационная связность (СС=9). Выходные данные одной части модуля используются как входные данные другой части модуля. Модуль разбивается на подмодули (последовательные части), выполняющие независимые функции, но вместе реализующие единую функцию.

7. Функциональная связность (СС=10). Модуль выполняет определенную функцию, но не может быть разбит на два или более других модулей.

Второй путь достижения независимости модулей состоит в минимизации связи между ними. Сцепление модулей – мера независимости модулей по данным, характеризуется как способом передачи данных, так и свойствами самих этих данных. Слабое сцепление более желательно, так как это означает высокий уровень их независимости. Модули являются полностью независимыми, если каждый из них не содержит о другом никакой информации. Чем больше информации о других модулях используется в них, тем более они независимы и тем теснее сцепление.

Таблица 2

Характеристика связности модуля

Связности	Сопровождаемость
Функциональная	Лучшая сопровождаемость
Информационная	
Коммуникативная	
Процедурная	Худшая сопровождаемость
Временная	
Логическая	
Связность по определению	

Для измерения сцепления используют понятие силы сцепления (СЦ). Существуют 7 типов сцепления.

1. Сцепление по кодам (СЦ=10). Если коды их команд перемещаются друг с другом. Это сцепление возникает тогда, когда для одного из модулей доступны внутренние области другого без обращения к его точкам входа или когда два модуля используют общий участок памяти с командами;

2. Сцепление по общей области (СЦ=7). Если модули разделяют одну и туже глобальную структуру данных;

3. Сцепление по внешним ссылкам (СЦ=5). Если есть доступ к данным в другом модуле через внешнюю точку входа. Модули ссылаются на один и тот же глобальный элемент данных.

4. Сцепление по управлению (СЦ=4). Один модуль управляет решением внутри другого модуля, с помощью различных переключателей и т.д.;

5. Сцепление по образцу (СЦ=3). Если параметры, передаваемые от модуля к модулю, содержат структуру данных;

6. Сцепление по данным (СЦ=1). Параметры одного модуля передаются в другой в виде простых элементов данных.

7. Независимое сцепление (СЦ=0). Модули не вызывают друг друга и не обрабатывают общую информацию.

Хороший модуль обладает наивысшей связностью и наименьшим сцеплением.

Добиться независимости модуля можно, учитывая следующие факторы:

- размер модуля. От него зависит читаемость, сложность, тестируемость (идеал: от 10–100 операторов);
- предсказуемость модуля. Работа модуля должна быть независима от предшествующих вызовов, т.е. не должен хранить остаточные данные;

• структура принятия решения. Модуль, принимающий решение должен вызываться модулем, содержащим такое решение;

• минимизация доступа к данным. Данных, на который модуль может ссылаться, должен быть сведен к минимуму;

• внутренние процедуры. Избегать применения в модуле внутренних процедур или оформлять их как отдельные модули.

8.2. Программирование модулей

Программирование модуля включает**:**

1. *Выбор языка программирования.* Существенное влияние на выбор языка оказывают его возможности обеспечивать надежный процесс получения программ, наличие и специфические особенности компилятора и т.д.

2. *Проектирование внешних спецификаций модуля.* Внешние спецификации модуля должны содержать следующую информацию:

а) имя модуля. Указывается имя, с помощью которого можно обратиться к модулю.

б) функция. Определяется, что делает модуль, когда он вызван, а также его назначение. Этот элемент спецификации не должен содержать сведения о том, как функция реализуется.

в) список параметров. Определяются число и порядок параметров, передаваемых модулю.

г) входные параметры. Подробно описываются все входные параметры (указываются атрибуты, формат, размер, единицы измерения, а также допустимые диапазоны возможных значений всех входных параметров).

д) выходные параметры. Описываются все данные, возвращаемые модулем.

е) связь между входными и выходными данными. Описывается взаимосвязь между входными и выходными данными, т.е. какие выходные данные на основе каких входных данных получаются. Определяются выходные данные, возвращаемые в вызывающий модуль в случае ошибочных входных данных.

ж) описание внешних эффектов. Дается описание всех внешних для программы или системы событий, происходящих при работе модуля, таких, как прием запроса, выдача сообщений об ошибках и т. п.

3. *Проверка правильности внешних спецификаций модуля.* Правильность спецификаций каждого модуля должна быть проверена сравнением их с информацией о взаимосвязях, полученной при проектировании структуры программы и в результате последующего обсуждения всеми программистами, разрабатывающими вызывающие модули.

4. Выбор алгоритма и структуры данных.

5. *Оформление начала и конца будущего модуля.* Предусматривается оформление модуля в соответствии с требованиями принятого языка программирования.

6. *Объявление всех данных*, используемых в качестве параметров. Записываются соответствующие операторы объявления.

7. *Объявление оставшихся данных.* Записываются операторы объявления всех оставшихся данных, которые должны быть использованы в модуле.

8. *Детализация текста программы.* На этом шаге используются методы пошаговой детализации и структурного программирования. *Пошаговая детализация* – процесс разложения (производится сверху вниз) функции модуля на подфункции. В конечном итоге подфункции превращаются в шаги требуемой программы.

9. Окончательное описание текста программы.

10. *Проверка правильности программы.* Проверка может осуществляться как в форме статического чтения программы (текст программы просто почитается от начала до конца), так и в форме динамического чтения (используются динамические тексты при отслеживании программы).

11. *Компиляция модуля*. Этот шаг отмечает переход от проектирования к тестированию модуля.

8.3. Функциональные модели используемые на стадии проектирования

Функциональные модели, используемые на стадии проектирования ПО, предназначены для описания функциональной структуры проектируемой системы. Построенные ранее DFD при этом уточняются, расширяются и дополняются новыми конструкциями. Помимо DFD могут использоваться и другие диаграммы, отражающие системную архитектуру ПО, иерархию экранных форм и меню, структурные схемы программ (структурные карты) и т. д. Состав диаграмм и степень их детализации определяются необходимой полнотой описания системы для непосредственного перехода к ее последующей реализации (программированию).

Так, например, для DFD переход от модели бизнес-процессов организации к модели системных процессов может происходить следующим образом:

- внешние сущности на контекстной диаграмме заменяются или дополняются техническими устройствами (например, рабочими станциями, принтерами и т.д.);
- для каждого потока данных определяется, посредством каких технических устройств информация передается или производится;
- процессы на диаграмме нулевого уровня заменяются соответствующими процессорами – обрабатывающими устройствами (процессорами

могут быть как технические устройства – ПК конечных пользователей, рабочие станции, серверы баз данных, так и служащие – исполнители);

- определяется и изображается на диаграмме тип связи между процессорами (например, локальная сеть – LAN – Local Area Network)
- определяются задачи для каждого процессора (приложения, необходимые для работы системы), для них строятся соответствующие диаграммы. Определяется тип связи между задачами;
- устанавливаются ссылки между задачами и процессами диаграмм потоков данных следующих уровней.

Иерархия экранных форм моделируется с помощью диаграмм последовательностей экранных форм. Совокупность таких диаграмм представляет собой абстрактную модель пользовательского интерфейса системы, отражающую последовательность появления экранных форм в приложении. Построение диаграмм последовательностей экранных форм выполняется следующим образом:

На DFD выбираются интерактивные процессы нижнего уровня. Интерактивные процессы нуждаются в пользовательском интерфейсе, поэтому можно определить экранную форму для каждого такого процесса;

Форма диаграммы изображается в виде прямоугольника для каждого интерактивного процесса на нижнем уровне диаграммы;

- определяется структура меню. Для этого интерактивные процессы группируются в меню (либо так же как и модели процессов, либо другим способом – по функциональным признакам или в зависимости от принадлежности к определенным объектам);
- формы с меню изображаются над формами, соответствующими интерактивным процессам, и соединяются с ними переходами в виде стрелок, направленным от меню к формам;
- определяется верхняя форма (главная форма приложения), связывающая все формы с меню.

Контрольные вопросы

1. Что такое модуль, какими он обладает признаками?
2. Что такое функциональная связность?
3. Какие виды связности модулей вы знаете?
4. Что такое сцепление модулей?
5. Какие виды сцепления модулей вы знаете?
6. Что понимают под стилем программирования?
7. Что такое детальное кодирование?

9. ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ ПОДХОД К ПРОЕКТИРОВАНИЮ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

9.1. Сущность Объектно-ориентированного подхода

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.

Концептуальной основой объектно-ориентированного подхода является *объектная модель. Она имеет четыре главных свойства*:

- *абстрагирование (abstraction);*
- *инкапсуляция (encapsulation);*
- *модульность (modularity);*
- *иерархия (hierarchy).*

Кроме основных имеются еще три дополнительных свойства, не являющихся в отличие от основных строго обязательными:

- *типизация (typing);*
- *параллелизм (concurrency);*
- *устойчивость (persistence).*

Абстрагирование – это выделение существенных характеристик некоторого объекта, которые отличают его от всех других видов объектов.

Инкапсуляция – скрытие внутренней реализации объекта за предоставляемым этим объектом интерфейсом.

Модульность – это свойство системы, связанное с возможностью ее декомпозиции на ряд внутренне связных, но слабо связанных между собой модулей.

Иерархия – это упорядочивание абстракций, расположение их по уровням.

Типизация – это ограничение, накладываемое на класс объектов и препятствующее взаимозаменяемости различных классов.

Параллелизм – способность системы обрабатывать несколько сообщений или задач параллельно.

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, имеющие четко определяемое поведение. 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 интерпретировано как способность класса принадлежать более чем одному типу. *Наследование* означает построение новых классов на основе существующих с возможностью добавления или переопределения данных и методов.

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.

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.

9.2. Унифицированный язык моделирования UML

Большинство существующих методов объектно-ориентированного анализа и проектирования (ООАП) включают как язык моделирования, так и описа-

ние процесса моделирования. Язык моделирования – это нотация (в основном графическая), которая используется методом для описания проектов. *Нотация* представляет собой совокупность графических объектов, которые используются в моделях; она является синтаксисом языка моделирования. Например, нотация диаграммы классов определяет, каким образом представляются такие элементы и понятия, как класс, ассоциация и множественность. *Процесс* — это описание шагов, которые необходимо выполнить при разработке проекта.

Унифицированный язык моделирования UML (Unified Modeling Language) – 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 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.

Язык UML принят на вооружение практически всеми крупнейшими компаниями – производителями ПО (Microsoft, IBM, Hewlett-Packard, Oracle, Sybase и др.). Кроме того, практически все мировые производители CASE-средств, помимо Rational Software (Rational Rose), поддерживают UML в своих продуктах (Paradigm Plus 3.6, System Architec, Microsoft Visual Modeler for Visual Basic, Delphi, PowerBuilder и др)

Создатели UML представляют его как язык для определения, представления, проектирования и документирования программных систем, организационно-экономических, технических и др. UML содержит стандартный набор диаграмм и нотаций самых разнообразных видов. Стандарт UML версии 1.1, принятый OMG в 1997 г., предлагает следующий набор диаграмм для моделирования:

- *диаграммы вариантов использования (use case diagrams)* – для моделирования бизнес-процессов организации (требований к системе);
- *диаграммы классов (class diagrams)* – для моделирования статической структуры классов системы и связей между ними;
- *диаграммы поведения системы (behavior diagrams);*

– *диаграммы состояний (statechart diagrams)* – для моделирования поведения объектов системы при переходе из одного состояния в другое;

– *диаграммы деятельностей (activity diagrams)* – для моделирования поведения системы в рамках различных вариантов использования или моделирования деятельностей;

– *диаграммы взаимодействия (interaction diagrams)* – для моделирования процесса обмена сообщениями между объектами. Существуют два вида диаграмм взаимодействия:

диаграммы последовательности (sequence diagrams);

кооперативные диаграммы (collaboration diagrams).

- *диаграммы реализации (implementation diagrams):*

– *диаграммы компонентов (component diagrams)* – для моделирования иерархии компонентов (подсистем) системы;

– *диаграммы размещения (deployment diagrams)* – для моделирования физической архитектуры системы.

9.2.1. Пакеты в языке UML

Пакет – основной способ организации элементов модели в языке UML. Каждый пакет владеет всеми своими элементами, т.е. теми элементами, которые включены в него. Про соответствующие элементы пакета говорят, что они принадлежат пакету или входят в него. При этом каждый элемент может принадлежать только *одному* пакету. В свою очередь, одни пакеты могут быть вложены в другие пакеты. В этом случае первые называются *подпакетами***,** поскольку все элементы подпакета будут принадлежать более общему пакету. Тем самым для элементов модели задается отношение вложенности пакетов, которое представляет собой иерархию.

В языке UML для визуализации пакетов разработана специальная символика или графическая нотация. Для графического изображения пакетов на диаграммах применяется специальный графический символ – большой прямоугольник с небольшим прямоугольником. присоединенным к левой части верхней стороны первого (рис. 22 а, б). Можно сказать, что визуально символ пакета напоминает пиктограмму папки в популярном графическом интерфейсе. Внутри большого прямоугольника может записываться информация, относящаяся к данному пакету. Если такой информации нет, то внутри большого прямоугольника записывается имя пакета, которое должно быть уникальным в пределах рассматриваемой модели (рис. 22 а). Если же такая информация имеется, то имя пакета записывается в верхнем маленьком прямоугольнике (рис. 22, б).

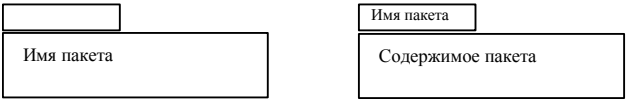


Рис. 22. Графические изображения пакета в языке UML

Говоря об имени пакета, следует остановиться на общем соглашении об именах в языке UML. В данном случае именем пакета может быть строка (или несколько строк) текста, содержащее любое число букв, цифр и некоторых специальных знаков. С целью удобства спецификации пакетов принято в качестве их имен использовать одно или несколько существительных, например, контроллер, графический интерфейс, форма ввода данных.

Перед именем пакета может помещаться строка текста, содержащая некоторое ключевое слово. Подобными ключевыми словами являются заранее определенные в языке UML слова, которые получили название *стереотипов.* Такими стереотипами для пакетов являются слова facade, framework, stub и topLevel. В качестве содержимого пакета могут выступать имена его отдельных элементов и их свойства, такие как видимость элементов за пределами пакета.

Конечно, сами по себе пакеты могут найти ограниченное применение, поскольку содержат лишь информацию о входящих в их состав элементах модели. Не менее важно представить графически отношения, которые могут иметь место между отдельными пакетами. Как и в теории графов, для визуализации отношений в языке UML применяются отрезки линий, внешний вид которых имеет смысловое содержание.

Одним из типов отношений между пакетами является отношение вложенности или включения пакетов друг в друга. С одной стороны, в языке UML это отношение может быть изображено без использования линий простым размещением одного пакета-прямоугольника внутри другого пакета-прямоугольника (рис. 23). Так, в данном случае пакет с именем Пакет1 содержит в себе два подпакета: Пакет_2 и Пакет3.

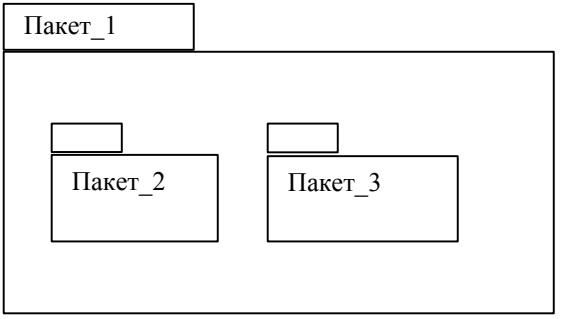


Рис. 23. Графические изображение вложенности пакетов друг в друга

9.2.2. Варианты использования

Проектирование системы начинается с описания ее функционального назначения и определения требования к ней. С этой целью Ивар Якобсон впервые ввел понятие «вариант использования» (use case) и придал ему такую значимость, что он превратился в основной элемент разработки и планирования проекта.

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. Например, два типичных варианта использования обычного текстового процессора – «сделать некоторый текст полужирным» и «создать индекс». Даже на таком простом примере можно выделить ряд свойств варианта использования: он охватывает некоторую очевидную для пользователей функцию, может быть как небольшим, так и достаточно крупным и решает для пользователя некоторую дискретную задачу. В простейшем случае вариант использования определяется в процессе обсуждения с пользователем тех функций, которые он хотел бы реализовать.

Для их наглядного представления вариантов использования применяются диаграммы вариантов использования. На рис. 24 показаны некоторые варианты использования для системы торговой организации; человеческие фигурки здесь обозначают действующих лиц, овалы – варианты использования, а линии и стрелки – различные связи между действующими лицами и вариантами использования.

Действующее лицо (actor) – это роль, которую пользователь играет по отношению к системе. На рис. четыре действующих лица: Менеджер по продажам, Оптовый торговец, Продавец и Система учета. Действующие лица представляют собой роли, а не конкретных людей или наименования работ. Несмотря на то, что на диаграммах вариантов использования они изображаются в виде стилизованных человеческих фигурок, действующее лицо может также быть внешней системой, которой необходима некоторая информация от данной системы (например, Система учета). Показывать на диаграмме действующих лиц системы следует только в том случае, когда им действительно необходимы некоторые варианты использования.

Все варианты использования так или иначе связаны с внешними требованиями к функциональности системы. Если Системе учета требуется файл, то это требование должно быть удовлетворено. Варианты использования всегда следует анализировать вместе с действующими лицами системы, определяя при этом реальные задачи пользователей и рассматривая альтернативные способы решения этих задач.

Рис. 24. Диаграмма вариантов использования

Действующие лица могут играть различные роли по отношению к варианту использования. Они могут пользоваться его результатами или могут сами непосредственно в нем участвовать. Значимость различных ролей действующего лица зависит от того, каким образом используются его связи.

Хорошим источником для идентификации вариантов использования служат внешние события. Следует начать с перечисления всех событий, происходящих во внешнем мире, на которые система должна каким-то образом реагировать. Идентификация событий, на которые необходимо реагировать, помогает выделить варианты использования.

В дополнение к связям между действующими лицами и вариантами использования существуют два других типа связей: «использование» (uses) и «расширение» (extends) между вариантами использования. Связь типа «рас-

ширение» применяется тогда, когда один вариант использования подобен другому, но несет несколько большую нагрузку.

Связь «использование» применяется в тех ситуациях, когда имеется какой-либо фрагмент поведения системы, который повторяется более чем в одном варианте использования, и нет необходимости копировать его описание в каждом из этих вариантов. Например, варианты Проанализировать риск и Договориться о цене требуют оценки стоимости сделки. Таким образом, создается отдельный вариант использования под названием Оценка стоимости, и предыдущие два варианта будут на него ссылаться.

Выбор применяемой связи определяется следующими правилами:

связь «расширение» следует применять при описании изменений в нормальном поведении системы;

связь «использование» следует применять для избежания повторов в двух (или более) вариантах использования.

Варианты использования являются необходимым средством на стадии формирования требований к ПО. Каждый вариант использования – это потенциальное требование к системе, и пока оно не выявлено, невозможно запланировать его реализацию.

9.2.3. Диаграммы классов

Диаграммы классов являются центральным звеном объектно-ориентированных методов. *Диаграмма классов* определяет типы объектов системы и различного рода статические связи, которые существуют между ними.

Класс (class) служит для обозначения множества объектов, которые обладают одинаковой структурой, поведением и отношениями с объектами из других классов. Графически класс изображается в виде прямоугольника, который может быть разделен на секции, где указывают имя класса, атрибуты, операции (рис. 25).

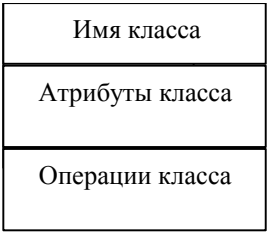


Рис. 25. Обозначение класса

Имя класса должно быть уникальным в пределах пакета, который описывается некоторой совокупностью диаграмм классов (возможно одной диаграммой).

Атрибуты описывают свойства класса.

В зависимости от степени детальности диаграммы обозначь атрибута может включать имя атрибута, тип и значение, присваиваемое по умолчанию (в синтаксисе UML это выглядит следуют образом:

<признак видимости> <имя атрибута>: <тип атрибута> = <значение по умолчанию>, где признак видимости может принимать одно из трех значений:

«+» (общий), «#» (защищенный) или «–» (секретный);

Операции представляют собой процессы, реализуемые классом. Наиболее очевидное соответствие существует между операциями и методами над классом.

Полный синтаксис UML для операций выглядит следующим образом:

<признак видимости> <имя операции> (<список параметров>):

<тип выражения возвращающего значение> {<строка свойств>}

где признак видимости может принимать одно из трех значений:

«+» (общий), «#» (защищенный) или «–» (секретный);

Пример записи операции: кредитныйРейтинг(): Строка.

Кроме внутреннего устройства или структуры классов на соответствующей диаграмме указываются различные отношения между классами. Базовыми отношениями или классами в языке UML являются:

- Отношение зависимости
- Отношение ассоциации
- Отношение обобщения
- Отношение агрегации

Каждое из этих отношений имеет собственное графическое представление на диаграмме, которое отражает взаимосвязи между объектами соответствующих классов.

Отношение зависимости используется в такой ситуации, когда некоторое изменение одного элемента модели может потребовать изменения другого зависимого от него элемента модели. Отношение зависимости графически изображается пунктирной линией между соответствующими элементами со стрелкой на одном из ее концов: от зависимого класса к независимому. На рис. 26 класс Б является источником зависимости, а класс А клиентом.

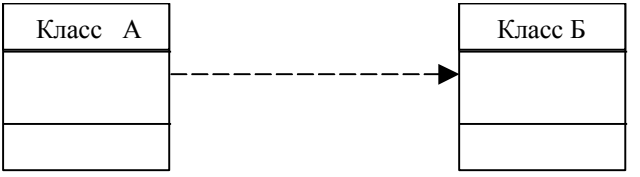


Рис. 26. Отношение между классами

Отношение ассоциации соответствует наличию некоторого отношения между классами. Данное отношение обозначается сплошной линией с дополнительными специальными символами, которые обозначают имя ассоциации, и имя и кратность классов-ролей ассоциации. Каждая ассоциация обладает двумя ролями; каждая роль представляет собой направление ассоциации. Пример ассоциации приведен на рис. 27, ассоциация имеет имя «Работа» и кратность один ко многим (в одной компании работает много сотрудников).

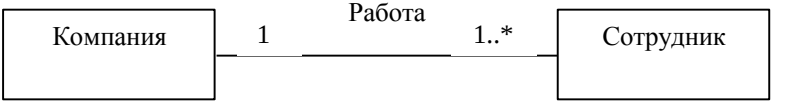


Рис. 27. Ассоциация между классами

Ассоциации представляют собой связи между экземплярами классов (личность работает в компании, компания имеет ряд офисов).

В общем случае множественность указывает нижнюю и верхнюю границы количества объектов, которые могут участвовать в связи. Символ «*» в действительности выражает диапазон «ноль-бесконечность».

На практике наиболее распространенными вариантами множественности являются «1», «*» и «0..1» (либо ноль, либо единица). В общем случае может использоваться единственное число (например, 11 для количества игроков в команде), диапазон (например, 2..4 для игроков в карты) или дискретная комбинация из чисел и диапазонов (например, 2,4 для количества дверей в автомобиле).

Отношение обобщения является обычным таксономическим отношением между более общим элементом (родителем или предком) и более частным или специальным элементом (дочерним или потомком.

Применительно к диаграмме классов данное отношение описывает иерархическое строение классов и наследование их свойств и поведения. При этом предполагается, что класс-потомок обладает всеми

свойствами и поведением класса предка, а также имеет свои собственные свойства и поведение, которые отсутствуют у класса-предка. На диаграмме отношение обобщения обозначается сплошной линией с треугольной стрелкой. Стрелка указывает на более общий класс (рис. 28). Типичный пример обобщения включает частного и корпоративного клиентов.

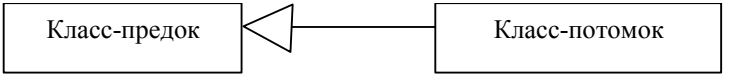


Рис. 28. Отношение обобщения

Этот пример описан несколькими типами, которые не обязательно должны быть связаны наследованием.

Отметим, что множественная классификация отличается от множественного наследования. При множественном наследовании тип может иметь много супертипов, но для каждого объекта должен быть определен только один тип. Множественная классификация допускает принадлежность объекта многим типам без определения для этих целей какого-либо специального типа.

Отношение агрегация имеет место между несколькими классами в том случае, если один из классов представляет собой некоторую сущность, включающую в себя в качестве составных частей другие сущности. Агрегация представляет собой связь «часть-целое» и является одним из наиболее часто используемых приемов моделирования (например, можно сказать, что двигатель и колеса являются частями автомобиля).

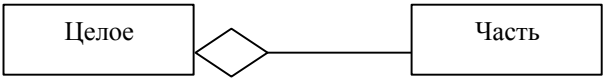


Рис. 29. Отношение агрегации

В дополнение к простой агрегации UML вводит более сильную разновидность агрегации, называемую *композицией* (рис. 30). Согласно композиции объект-часть может принадлежать только единственному целому и, кроме того, как правило, жизненный цикл частей совпадает с циклом целого: они живут и умирают вместе с ним. Любое удаление целого распространяется на его части.

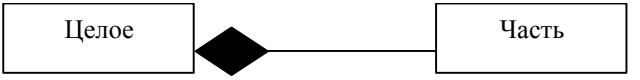


Рис. 30. Отношение композиция

Такое каскадное удаление нередко рассматривается как часть определения агрегации, однако оно всегда подразумевается в том случае, когда множественность роли составляет 1..1; например, если необходимо удалить Клиента, то это удаление должно распространиться и на Заказы (и, в свою очередь, на Строки заказа).

9.2.4. Диаграммы взаимодействия

Диаграммы взаимодействия (interaction diagrams) являются моделями, описывающими поведение взаимодействующих групп объектов.

Как правило, диаграмма взаимодействия охватывает поведение объектов в рамках только одного варианта использования. На такой диаграмме отображаются ряд объектов и те сообщения, которыми они обмениваются между собой.

Существуют два вида диаграмм взаимодействия: диаграммы последовательности (sequence diagrams) и кооперативные диаграммы (collaboration diagrams).

9.2.4.1. Диаграммы последовательности

На диаграмме последовательности изображаются исключительно те объекты, которые непосредственно участвуют во взаимодействии и не показываются возможные статические ассоциации с другими объектами. Для диаграммы последовательности ключевым моментом является именно динамика взаимодействия объектов во времени. При этом диаграмма последовательности имеет как бы два измерения. Одно – слева направо в виде вертикальных линий, каждая из которых изображает линию жизни отдельного объекта, участвующего во взаимодействии. Графически каждый объект изображается прямоугольником и располагается в верхней части своей линии жизни (рис. 31). Внутри прямоугольника записываются имя объекта и имя класса, разделенные двоеточием. При этом вся запись подчеркивается, что является признаком объекта, который, как известно, представляет собой экземпляр класса**.**

Крайним слева на диаграмме изображается объект, который является инициатором взаимодействия (объект 1 на рис. 31). Правее изображается другой объект который непосредственно взаимодействует с первым. Таким образом, все объекты на диаграмме последовательности образуют некоторый порядок, определяемый степенью активности этих объектов при взаимодействии друг с другом.

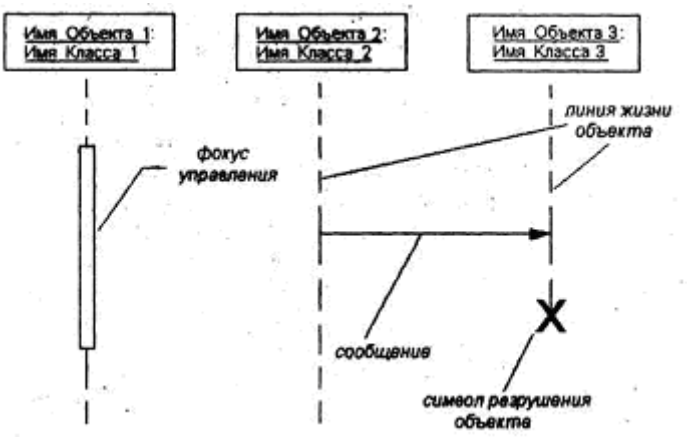


Рис. 31. Различные графические примитивы
диаграммы последовательности

Второе измерение диаграммы последовательности – вертикальная временная ось, направленная сверху вниз. Начальному моменту времени соответствует самая верхняя часть диаграммы. При этом взаимодействия объектов реализуются посредством сообщений, которые посылаются одними объектами другим. Сообщения изображаются в виде горизонтальных стрелок с именем сообщения и также образуют порядок по времени своего возникновения. Другими словами, сообщения, расположенные на диаграмме последовательности выше, инициируются раньше тех, которые расположены ниже.

Линия жизни объекта (object lifeline) изображается пунктирной вертикальной линией, ассоциированной с единственным объектом на диаграмме последовательности. Линия жизни служит для обозначения периода времени, в течение которого объект существует в системе. Если объект существует в системе постоянно, то и его линия жизни должна продолжаться по всей плоскости диаграммы последовательности от самой верхней ее части до самой нижней (объекты 1 и 2 на рис. 31).

Отдельные объекты, выполнив свою роль в системе, могут быть уничтожены (разрушены), чтобы освободить занимаемые ими ресурсы. Для таких объектов линия жизни обрывается в момент его уничтожения. Для обозначения момента уничтожения объекта в языке UML используется специальный символ в форме латинской буквы «X» (объект 3 на рис. 31). Ниже этого символа пунктирная линия не изображается, поскольку соответствующего объекта в системе уже нет, и этот объект должен быть исключен из всех последующих взаимодействий.

Вовсе не обязательно создавать все объекты в начальный момент времени. Отдельные объекты в системе могут создаваться по мере необходимости существенно экономя ресурсы системы и повышая ее производительность. В этом случае прямоугольник такого объекта изображается не в верхней части диаграммы последовательности, а в той ее части, которая соответствует моменту создания объекта (объект 6 на рис. 32). При этом прямоугольник объекта вертикально располагается в том месте диаграммы, которое по оси времени совпадает с моментом его возникновения в системе. Очевидно, объект обязательно создается со своей линией жизни и возможно с фокусом управления.

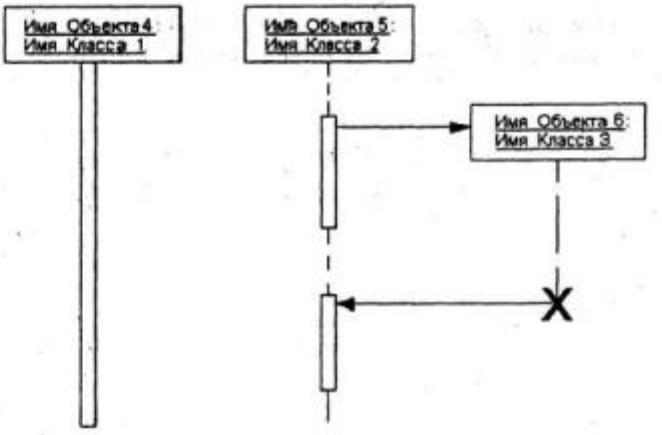


Рис. 32. Графическое изображение различных вариантов линий жизни и фокусов управления объектов

В процессе функционирования объектно-ориентированных систем одни объекты могут находиться в активном состоянии, непосредственно выполняя определенные действия или в состоянии пассивного ожидания сообщений от других объектов. Чтобы явно выделить подобную активность объектов, в языке UML применяется специальное понятие, получившее название *фокуса управления* (focus of control). Фокус управления изображается в форме вытянутого узкого прямоугольника (см. объект 1 на рис. 31), верхняя сторона которого обозначает начало получения фокуса управления объекта (начало активности), а ее нижняя сторона – окончание фокуса управления (окончание активности). Этот прямоугольник располагается ниже обозначения соответствующего объекта и может заменять его линию жизни (объект 4 на рис. 33), если на всем ее протяжении он является активным.

С другой стороны, периоды активности объекта могут чередоваться с периодами его пассивности или ожидания. В этом случае у такого объекта имеются несколько фокусов управления (объект 5 на рис. 33). Важно понимать, что получить фокус управления может только существующий объект, у которого в этот момент имеется линия жизни. Если же некоторый объект был уничтожен, то вновь возникнуть в системе он уже не может. Вместо него лишь может быть создан другой экземпляр этого же класса, который, строго говоря, будет являться другим объектом.

В отдельных случаях инициатором взаимодействия в системе может быть актер или внешний пользователь. В этом случае актер изображается на диаграмме последовательности самым первым объектом слева со своим фокусом управления (рис. 33). Чаще всего актер и его фокус управления будут существовать в системе постоянно, отмечая характерную для пользователя активность в инициировании взаимодействий с системой. При этом сам актер может иметь собственное имя либо оставаться анонимным.

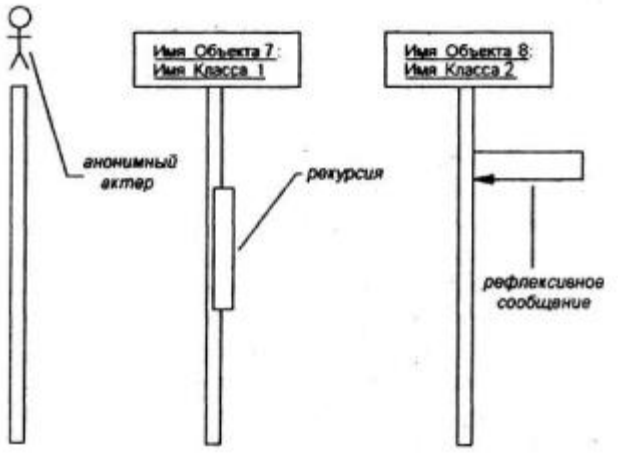


Рис. 33. Графическое изображение актера, рекурсии и рефлексивного сообщения

Иногда некоторый объект может инициировать рекурсивное взаимодействие с самим собой. Речь идет о том, что наличие во многих языках программирования специальных средств построения рекурсивных процедур требует визуализации соответствующих понятий в форме графических примитивов. На диаграмме последовательности *рекурсия* обозначается небольшим прямоугольником, присоединенным к правой

стороне фокуса управления того объекта, для которого изображается это рекурсивное взаимодействие (объект 7 на рис. 33).

Таким образом, сообщения не только передают некоторую информацию, но и требуют или предполагают от принимающего объекта выполнения ожидаемых действий. На диаграмме последовательности все сообщения упорядочены по времени своего возникновения в моделируемой системе.

В таком контексте каждое сообщение имеет направление от объекта, который инициирует и отправляет сообщение, к объекту, который его получает. Иногда отправителя сообщения называют клиентом, а получателя – сервером. При этом сообщение от клиента имеет форму запроса некоторого сервиса, а реакция сервера на запрос после получения сообщения может быть связана с выполнением определенных действий или передачи клиенту необходимой информации тоже в форме сообщения.

В языке UML могут встречаться несколько разновидностей сообщений, каждое из которых имеет свое графическое изображение (рис. 34).

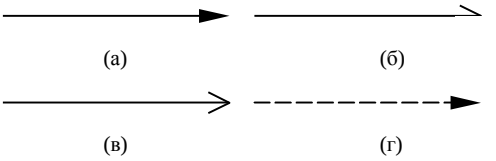


Рис. 34. Графическое изображение различных видов сообщений между объектами на диаграмме последовательности

Первая разновидность сообщения (рис. 34, а) является наиболее распространенной и используется для вызова процедур, выполнения операций или обозначения отдельных вложенных потоков управления. Начало этой стрелки всегда соприкасается с фокусом управления или линией жизни того объекта-клиента, который инициирует это сообщение. Конец стрелки соприкасается с линией жизни того объекта, который принимает это сообщение и выполняет в ответ определенные действия.

Вторая разновидность сообщения (рис. 34, б) используется для обозначения простого (не вложенного) потока управления. Каждая такая стрелка указывает на прогресс одного шага потока. При этом соответствующие сообщения обычно являются асинхронными, т.е. могут возникать в произвольные моменты времени. Передача такого сообщения обычно сопровождается получением фокуса управления объектом, его принявшим.

Третья разновидность (рис. 34, в) явно обозначает асинхронное сообщение между двумя объектами в некоторой процедурной последовательности. Примером такого сообщения может служить прерывание операции при возникновении исключительной ситуации. В этом случае информация о такой ситуации передается вызывающему объекту для продолжения процесса дальнейшего взаимодействия.

Наконец, последняя разновидность сообщения (рис. 34, г) используется для возврата из вызова процедуры. Примером может служить простое сообщение о завершении некоторых вычислений без предоставления результата расчетов объекту-клиенту. В процедурных потоках управления эта стрелка может быть опушена, поскольку ее наличие неявно предполагается в конце активации объекта. В то же время считается, что каждый вызов процедуры имеет свою пару – возврат вызова. Для непроцедурных потоков управления, включая параллельные и асинхронные сообщения, стрелка возврата должна указываться явным образом.

Обычно сообщения изображаются горизонтальными стрелками, соединяющими линии жизни или фокусы управления двух объектов на диаграмме последовательности. При этом неявно предполагается, что время передачи сообщения достаточно мало по сравнению с процессами выполнения действий объектами. Считается также, что за время передачи сообщения с соответствующими объектами не может произойти никаких событий. Другими словами, состояния объектов остаются без изменения. Если же это предположение не может быть признано справедливым, то стрелка сообщения изображается под некоторым наклоном, так чтобы конец стрелки располагался ниже ее начала.

В отдельных случаях объект может посылать сообщения самому себе, инициируя так называемые рефлексивные сообщения (объект 8 на рис. 33). Такие сообщения изображаются прямоугольником со стрелкой, начало и конец которой совпадают. Подобные ситуации возникают, например, при обработке нажатий на клавиши клавиатуры при вводе текста в редактируемый документ, при наборе цифр номера телефона абонента.

Таким образом, в языке UML каждое сообщение ассоциируется с некоторым действием, которое должно быть выполнено принявшим его объектом. При этом действие может иметь некоторые аргументы или параметры, в зависимости от конкретных значений которых может быть получен различный результат. Соответствующие параметры будет иметь и вызывающее это действие сообщение. Более тога, значения параметров отдельных сообщений могут содержать условные выражения, образуя ветвление или альтернативные пути основного потока управления.

9.2.4.2. Кооперативные диаграммы

Вторым видом диаграммы взаимодействия является кооперативная диаграмма, на которой экземпляры объектов показаны в виде пиктограмм. Как

и на диаграмме последовательности, здесь стрелки обозначают сообщения, обмен которыми осуществляется в рамках данного варианта использования. Их временная последовательность, однако, указывается путем нумерации сообщений.

Нумерация сообщений делает восприятие их последовательности более трудным, чем в случае расположения линий на странице сверху вниз. С другой стороны, такое пространственное расположение позволяет более легко отразить некоторые другие моменты, например можно показать взаимосвязь объектов, перекрывающиеся компоненты или другую информацию.

Независимо от используемой схемы нумерации на диаграмме можно разместить такого же рода управляющую информацию, как и на диаграмме последовательности.

9.2.5. Диаграммы состояний

Диаграммы состояний – хорошо известное средство описания поведения систем. Они определяют все возможные состояния, в которых может находиться конкретный объект, а также процесс смены состояний объекта в результате наступления некоторых событий. В большинстве объектно-ориентированных методов диаграммы состояний строятся для единственного класса и отражают динамику поведения единственного объекта.

Существует много форм диаграмм состояний, незначительно отличающихся друг от друга семантикой. Наиболее популярная форма, используемая в объектно-ориентированных методах, впервые применялась в методе OMT и впоследствии была адаптирована Гради Бучем.

Состояние может быть задано в виде набора конкретных значений атрибутов класса или объекта, при этом изменение их отдельных значений будет отражать изменение состояния моделируемого класса или объекта.

Следует заметить, что не каждый атрибут класса может характеризовать его состояние. Как правило, имеют значение только такие свойства элементов системы, которые отражают динамический или функциональный аспект ее поведения.

Состояние на диаграмме изображается прямоугольником со скругленными вершинами (рис. 35). Этот прямоугольник, в свою очередь, может быть разделен на две секции горизонтальной линией. Если указана лишь одна секция, то в ней записывается только имя состояния (рис. 35, а). В противном случае в первой из них записывается имя состояния, а во второй список некоторых внутренних действий или переходов в данном состоянии (рис. 35, б). При этом под действием в языке UML понимают некоторую атомарную операцию, выполнение которой приводит к изменению состояния или возврату некоторого значения (например, «истина» или «ложь»).

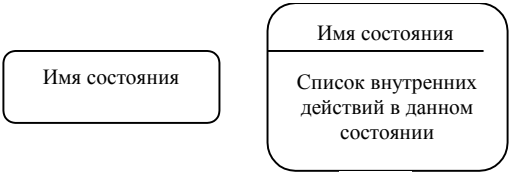


Рис. 35. Графические изображение состояний на диаграмме состояний

Имя состояния представляет собой строку текста, которая раскрывает содержательный смысл данного состояния Имя всегда записывается с заглавной буквы. Поскольку состояние системы является составной частью процесса ее функционирования, рекомендуется в качестве имени использовать глаголы в настоящем времени (звенит, печатает, ожидает) или соответствующие причастия (занят, свободен, передано, получено). Имя у состояния может отсутствовать, т. е. оно является необязательным для некоторых состояний.

Секция список внутренних действий содержит перечень внутренних действий или деятельностей, которые выполняются в процессе нахождения моделируемого элемента в данном состоянии. Каждое из действий записывается в виде отдельной строки и имеет следующий формат:

<метка-действия '/' выражение-действия >

Метка действия указывает на обстоятельства или условия, при которых будет выполняться деятельность, определенная выражением действия. При этом выражение действия может использовать любые атрибуты и связи, которые принадлежат области имен или контексту моделируемого объекта. Если список выражений действия пустой, то разделитель в виде наклонной черты '/' может не указываться.

Перечень меток действия имеет фиксированные значения в языке UML, которые не могут быть использованы в качестве имен событий. Эти значения следующие:

- *entry* – эта метка указывает на действие, специфицированное следующим за ней выражением действия, которое выполняется в момент входа в данное состояние (входное действие);
- *exit* – эта метка указывает на действие, специфицированное следующим за ней выражением действия, которое выполняется в момент выхода из данного состояния (выходное действие);
- *do* – эта метка специфицирует выполняющуюся деятельность («do activity»), которая выполняется в течение всего времени, пока объект нахо-

дится В данном состоянии, или до тех пор, пока не закончится вычисление, специфицированное следующим за ней выражением действия.

В последнем случае при завершении события генерируется соответствующий результат;

- *include* – эта метка используется для обращения к подавтомату, при этом следующее за ней выражение действия содержит имя этого подавтомата.

Во всех остальных случаях метка действия идентифицирует событие, которое запускает соответствующее выражение действия. Эти события называются *внутренними переходами* и семантически эквивалентны переходам в само это состояние, за исключением той особенности, что выход из этого состояния или повторный вход в него не происходит. Это означает, что действия входа и выхода не выполняются.

В качестве примера состояния рассмотрим ситуацию ввода пароля пользователя при аутентификации входа в некоторую программную систему (рис. 6.3). В этом случае список внутренних действий и данном состоянии не пуст и включает 4 отдельных действия, первые два из которых стандартные и описаны выше, а два последних определяются своей спецификацией.

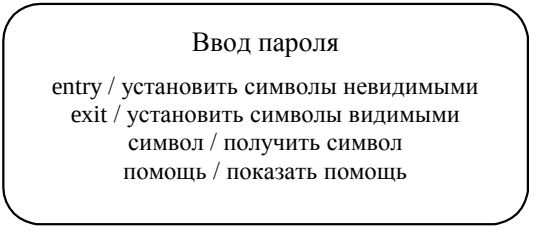


Рис. 36. Пример состояния с непустой секцией внутренних действий

Начальное состояние и конечное (финальное) состояние представляют собой частный случай состояния, которое не содержит никаких внутренних действий (псевдосостояния). В начальном состоянии находится объект по умолчанию в начальный момент времени: Оно служит для указания на диаграмме состояний графической области, от которой начинается процесс изменения состояний. Графически начальное состояние в языке UML обозначается в виде закрашенного кружка (рис. 37, а), из которого может только выходить стрелка, соответствующая переходу.

(а) Начальное состояние (б) Конечное состояние

Рис. 37. Графическое изображение начального
и конечного состояний на диаграмме состояний

В конечном состоянии будет находиться объект по умолчанию после завершения работы в конечный момент времени. Оно служит для указания на диаграмме состояний графической области, в которой завершается процесс изменения состояний или жизненный цикл данного объекта. Графически конечное состояние в языке UML обозначается в виде закрашенного кружка, помещенного в окружность (рис. 37,б), в которую может только входить стрелка, соответствующая переходу.

Простой переход (simple transition) представляет собой отношение между двумя последовательными состояниями, которое указывает на факт смены одного состояния другим. Пребывание моделируемого объекта в первом состоянии может сопровождаться выполнением некоторых действий, а переход во второе состояние будет возможен после завершения этих действий, а также после удовлетворения некоторых дополнительных условии. В этом случае говорят, что переход срабатывает, или происходит срабатывание перехода. До срабатывания перехода объект находится в предыдущем от него состоянии, называемым исходным состоянием, или в источнике (не путать *с* начальным состоянием – это разные понятия), а после его срабатывания объект находится в последующем от него состоянии (целевом состоянии).

Переход осуществляется при наступлении некоторого события: окончания выполнения деятельности (do activity), получении объектом сообщения или приемом сигнала. На переходе указывается имя события. Кроме того, на переходе могут указываться действия, производимые объектом в ответ на внешние события при переходе из одного состояния в другое. Срабатывание перехода может зависеть не только от наступления некоторого события, но и от выполнения определенного условия, называемого сторожевым условием. Объект перейдет из одного состояния в другое в том случае, если произошло указанное событие и сторожевое условие приняло значение «истина».

На диаграмме состояний переход изображается сплошной линией со стрелкой, которая направлена в целевое состояние (например, «выход из строя»). Каждый переход может помечен строкой текста, которая имеет следующий общий формат:

<сигнатура события>'['<сторожевое условие>']' <выражение действия>.

При этом сигнатура события описывает некоторое событие с необходимыми аргументами:

<имя события>\'<список параметров, раздоенных запятыми>')'.

Термин *событие* (event) требует отдельного пояснения, поскольку является самостоятельным элементом языка UML. Формально, событие представляет собой спецификацию некоторого факта, имеющего место в пространстве и во времени. Про события говорят, что они «происходят», при этом отдельные события должны быть упорядочены во времени. После наступления некоторого события нельзя уже вернуться к предыдущим событиям, если такая возможность не предусмотрена явно в модели.

Семантика понятия события фиксирует внимание на внешних проявлениях качественных изменений, происходящих при переходе моделируемого объекта из состояния в состояние. Например, при включении электрического переключателя происходит некоторое событие, в результате которого комната становится освещенной.

В языке UML события играют роль стимулов, которые инициируют переходы из одних состояний в другие. В качестве событий можно рассматривать сигналы, вызовы, окончание фиксированных промежутков времени или моменты окончания выполнения определенных действий. Имя события идентифицирует каждый отдельный переход на диаграмме состояний и может содержать строку текста, начинающуюся со строчной буквы. В этом случае принято считать переход *триггерным,* т.е. таким, который специфицирует событие-триггер.

Если рядом со стрелкой перехода не указана никакая строка текста, то соответствующий переход является *нетриггерным,* и в этом случае из контекста диаграммы состояний должно быть ясно, после окончания какой деятельности он срабатывает. После имени события могут следовать круглые скобки для явного задания параметров соответствующего события-триггера. Если параметров нет, то список параметров со скобками может отсутствовать.

Сторожевое условие (guard condition), если оно есть, всегда записывается в прямых скобках после события-триггера и представляет собой некоторое булевское выражение. Напомним, что булевское выражение должно принимать одно их двух взаимно исключающих значений: «истина» или «ложь». Из контекста диаграммы состояний должна явно следовать семантика этого выражения, а для записи выражения может использоваться синтаксис языка объектных ограничений.

Введение для перехода сторожевого условия позволяет явно специфицировать семантику его срабатывания. Если сторожевое условие принимает значение «истина», то соответствующий переход может сработать, в результате чего объект перейдет в целевое состояние. Если же

сторожевое условие принимает значение «ложь», то переход не может сработать, и при отсутствии других переходов объект не может перейти в целевое состояние по этому переходу. Однако вычисление истинности сторожевого условия происходит только после возникновения ассоциированного с ним события-триггера, инициирующего соответствующий переход.

В общем случае из одного состояния может быть несколько переходов с одним и тем же событием-триггером. При этом никакие два сторожевых условия не должны одновременно принимать значение «истина». Каждое из сторожевых условий необходимо вычислять всякий раз при наступлении соответствующего события-триггера.

Графически фрагмент логики моделирования почтовой программы может быть представлен в виде следующей диаграммы состояний (рис. 38). Как можно заключить из контекста, в начальном состоянии программа не выполняется, хотя и имеется на компьютере пользователя. В момент ее включения происходит ее активизация. В этом состоянии программа может находиться неопределенно долго, пока пользователь ее не закроет, т.е. не выгрузит из оперативной памяти компьютера. После окончания активизации программа переходит в конечное состояние. В активном состоянии программы пользователь может читать сообщения электронной почты, сознавать собственные послания и выполнять другие действия, не указанные явно на диаграмме.

Однако при необходимости получить новую почту, пользователь должен установить телефонное соединение с провайдером, что и показано явно на диаграмме верхним переходом. Другими словами, пользователь инициирует событие-триггер «установить телефонное соединение». В качестве параметра этого события выступает конкретный телефонный номер модемного пула провайдера. Далее следует проверка сторожевого условия «телефонное соединение установлено», которое следует понимать как вопрос. Только в случае положительного ответа «да», т.е. «истина», происходит переход почтовой программы-клиента из состояния «активизация почтовой программы» в состояние «загрузка почты с сервера провайдера». В противном случае (линия занята, неверный ввод пароля, отключенный логин) никакой загрузки почты не произойдет, и программа останется в прежнем своем состоянии.

Второй триггерный переход на диаграмме инициирует автоматический разрыв телефонного соединения с провайдером после окончания загрузки почты на компьютер пользователя. В этом случае событие-триггер «закончить загрузку почты» происходит после проверки сторожевого условия «почтовый ящик на сервере пуст», которое также следует понимать в форме вопроса. При положительном ответе на этот вопрос (вся почта загружена или ее просто нет в ящике) почтовая программа прекращает загрузку почты и переходит в состояние активиза-

ции. В случае же отрицательного ответа загрузка почты будет продолжена.

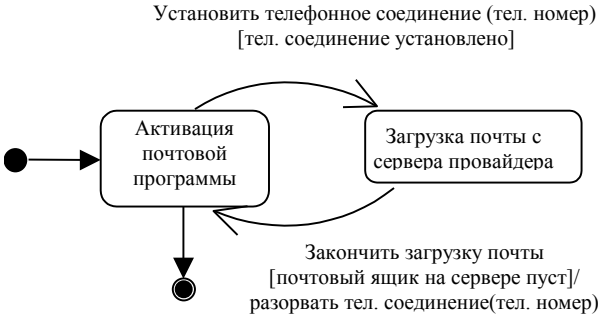


Рис. 38. Диаграмма состояний для моделирования почтовой программы-клиента

Выражение действия (action expression) выполняется в том и только в том случае, когда переход срабатывает. Представляет собой атомарную операцию (достаточно простое вычисление), выполняемую сразу после срабатывания соответствующего перехода до начала каких бы то ни было действий в целевом состоянии. Атомарность действия означает, что оно не может быть прервано никаким другим действием до тех пор, пока не закончится его выполнение. Данное действие может оказывать влияние как на сам объект, так и на его окружение, если это с очевидностью следует из контекста модели. Выражение записывается после знака "/" в строке текста, присоединенной к соответствующему переходу.

В общем случае, выражение действия может содержать целый список отдельных действий, разделенных символом ";". Обязательное требование – все действия из списка должны четко различаться между собой и следовать в порядке их записи.

В качестве примера выражения действия (рис. 38) может служить «разорвать телефонное соединение (телефонный номер), которое должно быть выполнено сразу после установления истинности («истина») сторожевого условия «почтовый ящик на сервере пуст».

Диаграммы состояний хорошо использовать для описания поведения некоторого объекта в нескольких различных вариантах использования. Они не слишком пригодны для описания поведения ряда взаимодействующих объектов. По существу, диаграммы состояний полезно сочетать с другими средствами. Например, диаграммы взаимодействия хороши для описания поведения нескольких объектов в единственном

варианте использования, а диаграммы деятельностей хороши для описания общей последовательности действий для нескольких объектов и вариантов использования.

Не следует строить диаграммы состояний для каждого класса в системе; их стоит использовать только для тех классов, поведение которых действительно интересует, и построение диаграмм состояний помогает лучше его понять. Например, пользовательский интерфейс и управляющие объекты обладают именно таким поведением, которое полезно изображать с помощью диаграмм состояний.

9.2.6. Диаграммы деятельностей

При моделировании поведения проектируемой или анализируемой системы возникает необходимость не только представить процесс изменения ее состояний, но и детализировать особенности алгоритмической и логической реализации выполняемых системой операций. Традиционно для этой цели использовались блок-схемы или структурные схемы алгоритмов.

Для моделирования процесса выполнения операций в языке UML используются так называемыё *диаграммы деятельности.* Применяемая в них графическая нотация во многом похожа на нотацию диаграммы состояний, поскольку на диаграммах деятельности также присутствуют обозначения состояний и переходов. Отличие заключается в семантике состояний, которые используются для представления не деятельностей, а действий и в отсутствии на переходах сигнатуры событий. Каждое состояние на диаграмме деятельности соответствует выполнению некоторой элементарной операции, а переход в следующее состояние срабатывает только при завершении этой операции в предыдущем состоянии. Графически диаграмма деятельности представляется в форме графа деятельности, вершинами которого являются состояния действия, а дугами – переходы от одного состояния действия к другому.

Таким образом, диаграммы деятельности можно считать частным случаем диаграмм состояний. Именно они позволяют реализовать на языке UML особенности процедурного и синхронного управления, обусловленного завершением внутренних деятельностей и действий. Основным направлением использования диаграмм деятельности является визуализация, особенностей реализации операций классов, когда необходимо представить алгоритмы их выполнения. При этом каждое состояние может являться выполнением операции некоторого класса либо ее части, позволяя использовать диаграммы деятельности для описания реакций на внутренние события системы.

Состояние действия (action state) является специальным случаем состояния с некоторым входным действием и по крайней мере одним выходящим из состояния переходом. Этот переход неявно предполагает,

что входное действие уже завершилось. Состояние действия не может иметь внутренних переходов, поскольку оно является элементарным. Обычное использование состояния действия заключается в моделировании одного шага выполнения алгоритма (процедуры) или потока управления.

Графически состояние действия изображается фигурой, напоминающей прямоугольник, боковые стороны которого заменены выпуклыми дугами (рис. 39). Внутри этой фигуры записывается *выражение действия* (action-expression), которое должно быть уникальным в пределах одной диаграммы деятельности.

(а) простое действие (б) выравнивание

Рис. 39. Графическое изображение состояния действий

Действие может быть записано на естественном языке, некотором псевдокоде или языке программирования. Никаких дополнительных или неявных ограничений при записи действий не накладывается. Рекомендуется в качестве имени простого действия использовать глагол с пояснительными словами (рис. 39, а). Если же действие может быть представлено в некотором формальном виде, то целесообразно записать его на том языке программирования, на котором предполагается реализовывать конкретный проект (рис. 39, б).

Каждая диаграмма деятельности должна иметь единственное начальное и единственное конечное состояния. Они имеют такие же обозначения, как и на диаграмме состояний. При этом каждая деятельность начинается в начальном состоянии и заканчивается в конечном состоянии. Саму диаграмму деятельности принято располагать таким образом, чтобы действия следовали сверху вниз.

Переход как элемент языка UML был рассмотрен при рассмотрении диаграммы состояния. При построении диаграммы деятельности используются только нетриггерные переходы, т.е. такие, которые срабатывают сразу после завершения деятельности или выполнения соответствующего действия. Этот переход переводит деятельность в последующее состояние сразу, как только закончится действие в предыдущем состоянии. На диаграмме такой переход изображается сплошной линией со стрелкой.

В качестве примера рассмотрим фрагмент известного алгоритма нахождения корней квадратного уравнения. Графически фрагмент процедуры вычисления корней квадратного уравнения может быть представлен в виде диаграммы деятельности с тремя состояниями действия и ветвлением (рис. 40).

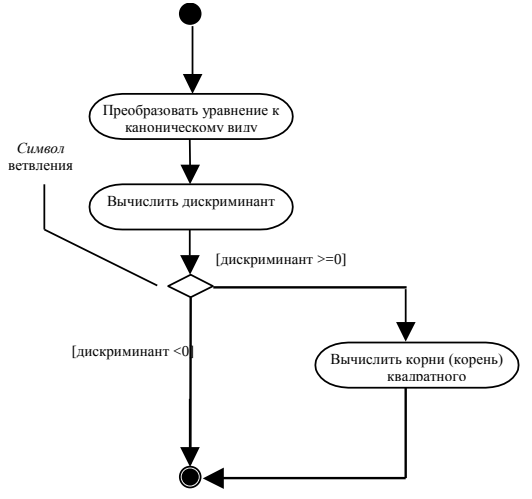


Рис. 40. Фрагмент диаграммы деятельности для алгоритма нахождения корней квадратного уравнения

Диаграмма деятельностей предоставляет свободу выбора порядка выполнения действий. Другими словами, она только устанавливает основные правила последовательности, которым необходимо следовать.

Такая возможность важна при моделировании бизнес-процессов. Среди бизнес-процессов нередко встречаются такие, которые не обязаны выполняться последовательно. В таких ситуациях данный метод хорошо работает, так как он позволяет реализовывать процессы параллельно.

Диаграммы деятельностей являются также полезными при параллельном программировании, поскольку можно графически изобразить все ветви и определить, когда их необходимо синхронизировать.

Диаграмма деятельностей не обязательно должна включать явно определенную конечную точку. Конечная точка на диаграмме деятельностей – это точка, в которой заканчивается выполнение всех деятельностей.

Такая способность диаграмм деятельностей описывать одновременно поведение множества вариантов использования является очень полезной. Каждый вариант использования дает информацию, отражающую взгляд извне на предметную область под определенным углом; когда мы смотрим на внутреннюю картину, желательно охватить ее всю целиком. Диаграммы классов дают полную картину взаимодействующих классов, то же самое делают диаграммы деятельностей в отношении поведения системы.

Любая деятельность может быть подвергнута дальнейшей декомпозиции. Описание декомпозированной деятельности может быть представлено в виде текста, кода или другой диаграммы деятельностей.

Когда строится диаграмма деятельностей, представляющая собой декомпозицию деятельности более высокого уровня, на ней должна присутствовать только одна начальная точка. С другой стороны, конечных точек может быть столько, сколько выходов у деятельности более высокого уровня. Таким образом, декомпозированная диаграмма возвращает значение, определяемое последним выходом.

Самым большим достоинством диаграмм деятельностей является поддержка параллелизма. Благодаря этому они являются мощным средством моделирования потоков работ и, по существу, параллельного программирования. Самый большой их недостаток заключается в том, что связи между действиями и объектами просматриваются не слишком четко.

Эти связи можно попытаться определить с помощью меток с именами объектов, но этот способ не обладает такой же простотой, как использование диаграмм взаимодействия. Диаграммы деятельностей предпочтительнее использовать в следующих ситуациях:

- анализ варианта использования. На этой стадии нас не интересует связь между действиями и объектами, а нужно только понять, какие действия должны иметь место и каковы зависимости в поведении системы. Связывание методов и объектов выполняется позднее с помощью диаграмм взаимодействия;
- анализ потоков работ (workflow) в различных вариантах использования. Когда варианты использования взаимодействуют друг с другом, диаграммы деятельностей являются мощным средством представления и анализа их поведения.

Не рекомендуется использовать диаграммы деятельностей в следующих ситуациях:

- анализ взаимодействия объектов. Для этого гораздо лучше подходят диаграммы взаимодействия, поскольку они проще и обеспечивают более наглядное представление;
- анализ поведения объекта в течение его жизненного цикла. Для этой цели используются диаграммы состояний.

9.2.7. Диаграммы компонентов

Диаграммы компонентов показывают, как выглядит модель системы на физическом уровне. На диаграмме изображены компоненты программного обеспечения и связи между ними. При этом выделяют два типа компонентов: исполняемые компоненты и библиотеки кода.

Каждый класс модели преобразуется в компонент исходного кода. После создания они сразу добавляются к диаграмме компонентов. Между отдельными компонентами изображают зависимости, соответствующие зависимостям на этапе компиляции или выполнения программы.

У системы может быть несколько диаграмм компонентов в зависимости от числа подсистем или исполняемых файлов. Каждая подсистема явля-

ется пакетом компонентов. В общем случае пакеты – это совокупности компонентов.

Диаграммы компонентов применяются теми участниками проекта, кто отвечает за компиляцию системы. Из нее видно, в каком порядке надо компилировать компоненты, а также какие исполняемые компоненты будут созданы системой. На такой диаграмме показано соответствие классов реализованным компонентам. Она нужна там, где начинается генерация кода.

9.2.8. Диаграммы размещения

Диаграмма размещения (deployment diagram) отражает физические взаимосвязи между программными и аппаратными компонентами системы. Она является хорошим средством для того, чтобы показать маршруты перемещения объектов и компонентов в распределенной системе.

Каждый узел на диаграмме размещения представляет собой некоторый тип вычислительного устройства, в большинстве случаев – часть аппаратуры. Эта аппаратура может быть простым устройством или датчиком, а может быть и мэйнфреймом.

Компоненты на диаграмме размещения представляют собой физические модули программного кода. Как правило, они в точности соответствуют компонентам на диаграмме компонентов. Таким образом, диаграмма размещения отражает выполнение каждого компонента в системе.

На практике диаграммы размещения используются не слишком часто. Многие разработчики действительно пользуются такими диаграммами, однако они представляют собой просто неформальные картинки. С другой стороны, каждая система имеет свои собственные физические характеристики, которые желательно явно выделить, и в дальнейшем потребуется большая степень формализма по мере достижения лучшего понимания того, какие проблемы следует решать в первую очередь с помощью диаграмм размещения.

Контрольные вопросы

1. В чем заключаются основные принципы объектно-ориентированного подхода?
2. В чем состоят достоинства и недостатки объектно-ориентированного подхода?
3. Назовите основные диаграммы языка UML.
4. Каковы принципиальные различия и что общего между структурным и объектно-ориентированным подходами?

10. ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА ИНТЕРФЕЙСА ПО

Интерфейс человек-компьютер включает такие аспекты вычислительной системы, которые касаются непосредственно пользователя.

Это важный фактор, обеспечивающий успешную работу вычислительной системы и существенно влияющий на производительность пользователя.

Чтобы работа с компьютером была *удобной,* пользователь должен при взаимодействии с системой ощущать *комфорт* можно выделить три большие группы факторов, влияющих на комфорт пользователя.

Общий психологический климат в организации (например, стиль работы администрации и социальная защищенность работника), а также форма преподнесения сведений о вычислительной системе могут вызвать предубеждение против этой системы задолго до того, как пользователь познакомится с ней практически. Функции, которые возлагает система на пользователя, и способ выполнения этих функций могут нарушить сложившиеся рабочие группы, лишить исполнителя привычного общения или испортить его отношения с руководством. Эти *социальные факторы* могут усилить или ослабить опасения пользователя относительно системы. Если пользователь относится к системе без предубеждений, *эргономические характеристики* реальной системы могут существенно улучшить или ухудшить его отношение к ней.

Конструктивные особенности оборудования (как компьютера, так и дополнительных устройств) и размещение его в рамках рабочей станции могут повлиять на чувство *физического комфорта* пользователя при работе с системой. Может ли оператор легко читать символы на экране дисплея, или экран расположен так, что из-за отражения солнечных лучей ему приходится щуриться, чтобы разглядеть текст.

Третий фактор – это *психологическая эргономика;* в то время как физическая эргономика занимается изучением соответствия функций системы физиологическим процессам человека, психологическая эргономика занимается изучением соответствия функций системы психологическим процессам человека. На этот аспект работы систем, получивший название *диалога,* разработчики программного обеспечения могут повлиять как в положительную, так и в отрицательную сторону.

С понятием психологической эргономики тесно связаны еще два фактора, хотя они важны сами по себе и заслуживают отдельного рассмотрения. Это *доступность* и *чувствительность* системы.

При проектировании интерфейса возникают трудности. Поскольку для успешного функционирования интерфейса нужно, чтобы он соответствовал физиологическим и психологическим потребностям пользо-

вателя, то при его проектировании и реализации возникает очевидная проблема: все люди разные.

Естественно, что у разных людей разные физические данные и возможности. Удобное расположение клавиш с точки зрения одного пользователя может оказаться неудобным для другого, отличающегося от первого физическими параметрами. Уровень яркости, необходимый одному пользователю для различения символов на экране, может оказаться недостаточным для другого.

Аналогично у людей разные психологические запросы, которые зависят как от конкретного человека, так и от решаемой задачи. Начинающему пользователю, незнакомому с системой или, редко на ней работающему, потребуются более подробные сообщения системы по сравнению с опытным пользователем, работающим регулярно. Очень раздражает необходимость каждый раз читать подробные объяснения о том, как войти в систему, если вы осуществляете эту операцию несколько раз в день на протяжении последних шести месяцев. Почти также раздражает отсутствие всяких пояснений при первом сеансе работы с системой! Для реализации сложной задачи, состоящей из последовательности закодированных ответов, нужно гораздо больше оперативной памяти и поддержки со стороны интерфейса, чем для реализации простого запроса.

И хотя знания наши растут, мы гораздо хуже разбираемся в том, как работает наш ум, по сравнению с тем, как работает компьютер. Разработчик интерфейса, хорошо знакомый с организацией работ, должен тем не менее уметь применять эти неполные знания для создания удобств в работе совершенно разных людей. Он должен не только удовлетворять их требования с точки зрения вычислительных задач, но и создать интерфейс, удобный с точки зрения физических и психологических потребностей пользователя.

10.1. Стратегия разработки интерфейса

Существует стратегия проектирования интерфейса, которая может привести к созданию систем, удобных для использования людьми. Приведем основные ее положения.

10.1.1. Интерфейс человек-компьютер как отдельный компонент системы

При проектировании программного средства так же как структуры данных в системе можно изолировать от алгоритмов обработки этих структур, так и разработку интерфейса нужно отделить от разработки обрабатывающих модулей программной системы. Состав и форма пред-

ставления входных и выходных данных должны стать предметом тщательного анализа разработчиков интерфейса.

10.1.2. Возможности аппаратных и программных средств

Разработчики систем, как, естественно, и другие специалисты, пользуются в работе своими старыми навыками. Этот внутренний консерватизм усиливается, а не ослабевает вследствие стремительного развития за последнее время аппаратных и программных средств. Однако невозможно разработать компонент, не понимая возможностей и ограничений основных элементов, из которых он может быть построен.

10.1.3. Будьте последовательны

Желательно развивать «семейство» программ, в рамках которых все они работают одинаково. Предполагалось, что этому будет способствовать ведение *книги стиля,* в которой содержались бы рекомендации по разработке интерфейса. Подобные книги ведут газеты и журналы, чтобы обеспечить единство стиля разных журналистов. Этому также может способствовать библиотека стандартных модулей, которые могут использоваться для разработки программных интерфейсов различных систем.

10.1.4. Используйте принятые принципы разработки интерфейса

Физическое взаимодействие пользователя с рабочей станцией имеет много общего с взаимодействием человека с машиной вообще. Поэтому существует большое число общепринятых в эргономике рекомендаций, которые можно легко перенести на разработку и организацию рабочей станции. В то же время форма представления информации на экране не одинакова для различных вычислительных систем: графический дизайн зависит от распределения информации на экране, словарного состава предложений, способа выделения ключевых элементов представления данных и т.д.

Разработчики должны знать эти принципы и стараться также использовать знания из других областей при затруднениях в решении своих проблем.

10.1.5. «Поймите» задачу и пользователя

Разработчик должен понимать не только вычислительный процесс, необходимый для решения задачи, но и оценивать действия пользователя, направленные на достижение цели задачи. Ему нужно знать особенности потенциальных пользователей системы.

10.1.6. Привлекайте пользователей

Рекомендацию о том, что надо «понимать пользователя и задачу», легче дать, чем выполнить. Вряд ли можно ожидать, что системный

аналитик или разработчик хорошо знаком со всеми областями применения своих разработок или глубоко чувствует психологические потребности потенциальных пользователей.

Существуют принципы, которым нужно следовать и которые описаны в любом учебнике по системному анализу. Один из типичных принципов заключается в том, что аналитик получает сведения путем опроса будущих пользователей. Полезный способ подбора нужных вопросов – это поставить себя на место пользователя, работающего с системой.

Однако единственный способ оценки доступности интерфейса – это посмотреть, как на самом деле пользователь взаимодействует с системой в нормальных рабочих условиях. Вряд ли можно ожидать, что пользователь правильно оценит достоинства организации данных по образцам на бумаге, представляемых часто в формате расположения данных на экране дисплея. Аналогично особенности организации ввода трудно понять из описания: пользователь должен нажимать соответствующие клавиши в ответ на сообщения системы.

Это требование противоречит традиционному взгляду на разработку программ как на линейный процесс, состоящий из нескольких этапов:

Анализ -> Разработка -> Компоновка -> Реализация

Нужен интерактивный подход, приводящий к разработке опытных образцов интерфейсов, с которыми работают пользователи и которые изменяются в соответствии с их реакцией до тех пор, пока не будет создан приемлемый продукт. Это значит, что нужно применять гибкие методы компоновки элементов интерфейса.

10.1.7. Предусматривайте средства адаптации в рамках интерфейса

Хотя общие принципы определяют основу создания интерфейса, они не могут удовлетворять любого пользователя. Разработка интерфейса в расчете на среднего пользователя – это как бы поиск наименьшего общего знаменателя. Точно так же привлечение пользователей к разработке не может гарантировать ее абсолютной приемлемости. Даже если условия задачи остаются практически постоянными, потребности пользователей, как и они сами, меняются.

Правильно спроектированный интерфейс должен быть настраиваемым на нужды разных пользователей, а также на одного пользователя в разные периоды его работы.

10.2. Составные части интерфейса человек-компьютер

С точки зрения программного обеспечения в состав интерфейса входят два компонента: набор процессов ввода-вывода и процесс диа-

лога. Пользователь вычислительной системы взаимодействует с интерфейсом: через интерфейс он посылает входные данные и принимает выходные.

Процессы ввода-вывода служат для того, чтобы принять от пользователя и передать ему данные через различные физические устройства. Отделяя физический процесс ввода-вывода от процесса диалога, можно добиться того, что смена устройств ввода-вывода не приведет к изменению процесса диалога, а сведется лишь к замене процесса ввода-вывода.

Диалог между человеком и компьютером можно определить как обмен информацией между вычислительной системой и пользователем, проводимый с помощью интерактивного терминала и по определенным правилам.

Процесс диалога – это механизм обмена информацией, который можно рассматривать как оболочку, включающую все входящие в систему процессы по выполнению определенных заданий. Процессы ввода-вывода обеспечивают обмен на самом верхнем уровне; на этом уровне диалоговый процесс должен правильно интерпретировать каждое слово и даже каждый звук. Задачи диалогового процесса:

1. Определение задания, которое пользователь возлагает на систему.

2. Прием логически связанных входных данных от пользователя и размещение их в переменных соответствующего процесса в нужном формате.

3. Вызов процесса выполнения требуемого задания.

4. Вывод результатов обработки по окончании процесса в подходящем для пользователя формате.

Отделение процесса диалога от процесса выполнения задания позволяет использовать один и тот же процесс (например, добавить запись к связанному списку) для выполнения различных заданий пользователя (например, принять заказ клиента или обновить счет в банке) путем включения его в разные процессы диалога.

Во время диалога происходит обмен информацией между его участниками. Информация передается в виде сообщений. В любом диалоге существует несколько типов этих сообщений (рис. 41).

Подсказка – это выходное сообщение системы, побуждающее пользователя вводить данные.

Реакция пользователя на подсказку может вызвать процесс выполнения некоторого задания (просмотреть прейскурант) или какую-либо функцию диалогового процесса (например, вывести справку для правильного ответа на подсказку) или же передать процессу выполнения задания входные данные. Например, входное сообщение может выбрать процесс, который выводит содержимое файла, или задать имя файла, который нужно вывести. Такие сообщения называют *входными управляющими сообщениями,* или командами, так как они управляют ходом

диалога. Второй тип сообщений называют *входными данными.* Могут быть и сложные сообщения, которые за один сеанс обмена вызывают нужный процесс и вводят требуемые данные.

Рис. 41. Типы сообщений диалога

Обычно для диалогового процесса нужно проверять введенные пользователем данные на наличие ошибок. Характер проверки зависит от формата входного сообщения, но в любом случае проверяется совпадение входного управляющего сообщения с одним элементом из списка возможных значений или нахождение входных данных в пределах допустимого диапазона. Во многих случаях объем проверок, проводимый диалоговым процессом, ограничен. Например, если на вход подается имя файла, то диалоговый процессор может проверить, соответствует ли введенное значение требуемому формату (в ходе выполнения задания будет установлено существование такого файла). *Сообщение об ошибке* – это сигнал диалогового процесса о том, что невозможно дальнейшее выполнение работы, потому что он или вызванный процесс выполнения задания не может обработать сообщение, введенное пользователем.

Диалоговый процесс организует связь пользователя с другими доступными ему процессами в системе. Таким образом, введенное пользователем сообщение часто преобразуется диалоговым процессом в стандартный формат и передается на вход другого процесса. Это задание возвратит диалоговому процессу либо подтверждение о получении входных данных, либо результат обработки. Диалоговый процесс, в свою очередь преобразует это выходное сообщение в подходящий для

пользователя формат и выводит его в виде данных или как сообщение о состоянии системы.

Выходные данные – это данные, которые возвращает процесс по окончании обработки. Например, если во входном сообщении содержится запрос на вывод содержимого файла, то, по всей видимости, будет выведено одно или несколько сообщений с содержимым этого файла. Процесс выполнения задания передает выходные данные в стандартной форме на вход диалогового процесса, который преобразует их в подходящий для пользователя формат.

Сообщение о состоянии системы – это информация для пользователя о том, что произошло или происходит в системе. Например, если по запросу системы пользователь вводит закодированные данные, то система может подтвердить получение элемента данных. Система может сообщить, что выполнение – конкретного задания закончилось (например, бухгалтерский расчет выполнен), или в случае большой задержки при выводе данных система может выдать сообщение, подтверждающее, что она работает («проверяю цену»).

Еще один вид сообщений выводится в тех случаях, когда пользователь не может ответить на запрос системы, потому что ему не понятен запрос или он забыл, что именно нужно вводить. При затруднении пользователя диалоговый процесс может вывести *справочную информацию,* поясняющую, что делать дальше и почему.

Диалог можно классифицировать с учетом формата входных сообщений и гибкости, позволяющей пользователю вводить входные сообщения, когда ему удобно. Входное сообщение позволяет:

- выбрать режимы диалога, например получение справки;
- выбрать нужный процесс выполнения задания;
- ввести данные для выполнения задания.

Диалог*, управляемый системой,* – это диалог, в котором процесс жестко задает, какое задание можно выбрать и какие данные вводить. Это осуществляется с помощью системы подсказок пользователю, которые определяют, какую информацию нужно вводить на каждом этапе. Такими подсказками в диалоге между пользователем и вычислительной системой являются вопросы, задаваемые системой и ответы пользователя, пункты меню и формы для ввода данных.

Диалог, *управляемый пользователем,* – это диалог, в котором инициатива принадлежит пользователю, т.е. он непосредственно подает команду на выполнение нужного на данном этапе задания. Для этого пользователь обычно вводит комбинированные данные, с помощью которых он одновременно выбирает процесс и вводит необходимые для обработки данные, такую операцию называют командой. Большинство операционных систем включают в свой состав диалоги подобного типа.

Таким образом основой для классификации интерфейса человек-компьютер является структура диалога. Она определяет степень взаимодействия между системой и пользователем; она также задает основные формы управления этим взаимодействием. Традиционно выделяют четыре основные структуры типов, каждая, из которых основывается на аналогии с определенным типом взаимодействия между людьми:

- вопрос и ответ;
- меню;
- экранных форм;
- на базе команд.

10.3. Что такое хороший диалог?

Степень пригодности любого диалога, разработанного и используемого в программе можно оценить с помощью пяти основных критериев – естественность, последовательность, краткость, поддержка пользователя и гибкость.

10.3.1. Естественность

Естественный диалог – это такой, который не вынуждает пользователя, взаимодействующего с системой существенно изменять свои традиционные способы решения задачи. Это не значит, что компьютер должен во всем подражать человеку. Как минимум это означает, что диалог должен вестись на родном языке пользователя.

Стиль ведения диалога должен быть разговорным, а не письменным. Следует избегать как чрезмерной напыщенности, так и «фамильярности».

Фразы, по возможности, не должны требовать дополнительных пояснений. Использование жаргона вполне допустимо, он дает возможность хорошо идиоматически прочувствовать диалог, но при условии, что этот жаргон используется в среде пользователей, а не специалистов в области вычислительной техники.

Другим важным аспектом естественности является тот порядок, в котором система запрашивает информацию. Всегда следует придерживаться такого порядка, в котором пользователь обычно обрабатывает информацию. Не следует заставлять человека вручную обрабатывать данные перед вводом в систему или после вывода из системы, облегчая жизнь компьютеру или программисту.

Разработчик должен также избегать бессмысленного (с человеческой точки зрения) порядка постановки вопросов.

Неестественный диалог часто является следствием того, что разработчик не знаком с теми способами, которыми пользователь обычно решает задачу без поддержки системы, следовало бы определить это с

помощью разработки и испытания опытного образца системы. Например, дополнительная работа пользователя по подготовке данных перед вводом и после вывода налицо, если он в процессе работы за терминалом будет пользоваться карандашом и бумагой.

10.3.2. Последовательность

Диалог, отличающийся логической последовательностью, гарантирует, что пользователь, освоивший работу одной части системы, не запутается, разбираясь с особенностями описания и работы другой части системы.

Последовательность в построении фраз предполагает, что вводимые коды, например ключевые слова, всегда трактуются одинаково.

Стандартные ответы должны быть действительно «стандартными». Клавиша F1 не должна иметь других функций, кроме функций вызова справочной информации или помощи системы. Соглашения, принятые в диалоге для операций указания и выбора, должны быть одинаковыми.

Последовательность в использовании *формата* данных означает, что аналогичные поля всегда будут представляться системой в одном и том же формате.

Последовательность в *размещении* данных на экране в разных ситуациях, сходных по реализуемым функциям, является гарантией того, что пользователю известно, где искать на экране инструкции, сообщения об ошибках и т. д. Нужно соблюдать такую же последовательность и при выделении информации так, чтобы, например, появление на экране участка в инверсном изображении всегда указывало на важность сообщения.

10.3.3. Краткость

Краткий диалог требует от пользователя ввода только минимума информации, необходимой для работы системы. Он обладает тем достоинством, что взаимодействие с ним становится более быстрым и осуществляется с меньшим числом ошибок, поскольку оно прямо зависит от количества требуемых нажатий на клавиши.

В интерактивном диалоге пользователь никогда не должен вводить незначащие символы.

В диалоге не следует запрашивать информацию, которую можно сформировать автоматически или которая уже была введена ранее, например текущая дата. Если для описания какого-либо объекта код уже был определен (например, 153Н – это код клиента Дж. Х. Смита), но только его и нужно использовать для идентификации указанного объекта, почти всегда бессмысленно заставлять пользователя вводить одновременно как код, так и наименование объекта, поскольку контроль кода на соответствие наименованию объекта осуществляется редко. Точно

так же не следует требовать информации, которая не используется системой. Полезным способом сокращения вводимых данных является ввод значений по умолчанию. С той же целью используются сокращения, при этом разработчик должен избегать двусмысленности или неестественности сокращений ключевых слов, а также вводить уникальные идентификаторы, состоящие из одного символа.

Информация не должна выводиться сразу же только потому, что она доступна системе. Выходные сообщения должны содержать именно ту информацию, которая запрашивается/требуется пользователю, причем в приемлемой для восприятия форме с привлечением минимума средств для выделения части информации.

Диалог, который не удовлетворяет указанному критерию, часто оказывается таким потому, что программист, разработавший его, постарался облегчить свою задачу или сделать ее более интересной для себя.

10.3.4. Поддержка пользователя

Поддержка пользователя в процессе диалога – это мера помощи, которую диалог оказывает пользователю при его работе с системой. Тремя основными аспектами этой поддержки являются:

- количество и качество имеющихся инструкций;
- характер выдаваемых сообщений об ошибках;
- подтверждение каких-либо действий системы.

Инструкции для пользователя выводятся в виде подсказок, либо справочной информации. Характер и количество инструкций должны, очевидно, соответствовать опыту работы пользователя с системой и его намерениям. Расточительно выдавать инструкции, которые не требуются пользователю или не имеют отношения к его проблеме. Справочная информация должна появляться тогда, когда она требуется и в приемлемой форме. Знает ли пользователь, почему система задает какой-то вопрос, или он как раз забыл точный формат ответа?

Сообщение об ошибке должно точно объяснить, в чем заключается ошибка и какие действия следует предпринять, чтобы ее устранить, а не ограничиваться общими фразами типа «синтаксическая ошибка» или таинственными кодами.

Сообщения, подтверждающие какое-либо действие системы, требуются для того, чтобы пользователь мог еще раз убедиться в том, что система выполнила или будет выполнять требуемое действие. Хотя такие сообщения существенны в некоторых случаях, использовать их следует не очень часто. Очень раздражает постоянная необходимость подтверждения простых запросов.

Подтверждать ввод данных следует тогда, когда эти данные приведут к необратимым действиям системы (например, удаление записи из файла) или когда вводится код и пользователю необходимо проверить по соответствующему описанию, нужная ли запись выбрана (например, о клиенте или товаре). Также может оказаться желательным подтверждение, что было выполнено некоторое действие (например, в файл добавлена запись).

В системах со сложной иерархической структурой, пользователь может забыть, на каком уровне он находится. Может оказаться полезным не только подтвердить выполнение какого-то действия, но и вывести сообщение о состоянии системы, чтобы пользователь знал, где он находится. Это можно реализовать, отведя на экране специальную область, в которой будет отображаться тот путь, который прошел пользователь до текущего состояния.

Сущность диалога, поддерживающего пользователя, состоит в том, что пользователю необходимо обеспечить более или менее адекватную поддержку. Диалог может не удовлетворять этому критерию из-за того, что разработчик не смог правильно оценить потребность пользователя в поддержке, или потому, что средства поддержки не могут отразить всего многообразия требований пользователей.

10.3.5. Гибкость

Гибкость диалога – это мера того, насколько хорошо он соответствует различным уровням подготовки и производительности труда пользователя. Такая гибкость предполагает, что диалог может подстраивать свою структуру или входные данные.

Концепция адаптации диалога является одной из основных областей исследования взаимодействия человека и машины. Основная проблема состоит не в том, как организовать изменения в диалоге, а в том, какие признаки нужно использовать для определения необходимости внесения изменений и их сути.

Четыре традиционные структуры диалога в различной степени отвечают перечисленным основным критериям. С точки зрения взаимодействия пользователя и прикладной программы одни из них кажутся более естественными, чем другие. Эти структуры обеспечивают различный уровень поддержки пользователя.

10.3.6. Смешанная структура диалога

Четыре основные структуры диалога различаются весьма незначительно и в действительности являются просто разновидностями структуры типа вопрос – ответ.

Структура диалога типа меню – это такая модификация структуры диалога типа вопрос – ответ, когда справочная информация первого уровня, т.е. само меню, автоматически отображается до запроса возможных вариантов ответа. Структура типа экранных форм отображает сразу весь комплекс вопросов (собственно форму), а затем по очереди запрашивают ответы. Структура на основе языка команд, особенно когда у команды есть позиционные параметры, является структурой типа вопрос – ответ, в рамках которой пользователь широко применяет метод «опережающего ввода», т.е. в ответ на первый стандартный вопрос (командную подсказку) он сразу отвечает на серию неявных вопросов.

Структура типа меню подходит там, где

- диапазон возможных ответов достаточно мал, и все они могут быть явно отражены;
- пользователю необходимо видеть возможные варианты ответов.

Предполагается, что меню следует использовать в тех случаях, когда пользователь неопытен или он в основном пользуется методом ввода путем указания нужного объект; из ограниченного множества данных, например при выборе задачи для исполнения.

Структура на основе языка команд будет приемлема там, где

- число значений для ввода достаточно мало и их можно запомнить;
- ограниченного числа ответов достаточно, чтобы идентифицировать как требуемую задачу, так и необходимые данные.

Предполагается, что эта структура будет использоваться подготовленным пользователем там, где задачи обработки не имеют иерархической структуры и не требуется много данных на входе.

Структура типа экранных форм удобна там, где можно заранее определить стандартную последовательность вводимых данных, как, например, при обработке транзакций в виде таблиц. Ее целесообразно использовать для ввода табулированных данных.

Основная структура диалога типа вопрос – ответ представляет собой разумный компромисс для различных уровней подготовки пользователей. Ее можно использовать вместо любой из приведенных выше структур, но особенно хорошо она подходит там, где:

- диапазон входных величин слишком велик для структуры типа меню или слишком сложен для структуры на основе языков команд;
- последующий вопрос зависит от ответа на текущий вопрос.

Однако с простой классификацией, приведенной выше, связана очевидная проблема. Даже при требуемом уровне подготовки пользователя требования к данным в различных частях системы будут различаться.

Итак, несмотря на то, что большинство систем имеют в своей основе структуры диалога типа вопрос – ответ, меню или структуры на базе команд, редко удается построить диалог для всей системы, используя только одну структуру. Для различных частей диалога нужны различные структуры в зависимости от их конкретных характеристик. Другими словами, большинство диалогов должны базироваться на смешанных структурах, объединяющих в себе несколько основных структур.

Контрольные вопросы

1. Какие эргономические характеристики влияют на работу пользователя с ПК?
2. Что такое интерфейс?
3. Каких правил нужно придерживаться при разработке интерфейса?
4. Какой диалог пользователя с компьютером можно назвать хорошим диалогом?
5. Изложите основные принципы при проектировании диалога типа меню.
6. Каких правил нужно придерживаться при проектировании оконной формы диалога?
7. Какие правила нужно помнить при размещении и выделении информации на экране?
8. Перечислите требования для разработки модулей помощи и справки.

11. ТЕСТИРОВАНИЕ, ОТЛАДКА И СБОРКА ПО

11.1. Основные определения

Многие организации, занимающиеся созданием программного обеспечения, до 50% средств, выделенных на разработку программ, тратят на тестирование, что составляет миллиарды долларов по всему миру в целом. И все же, несмотря на громадные капиталовложения, знаний о сути тестирования явно не хватает, и большинство программных продуктов неприемлемо, ненадежно даже после «основательного тестирования».

Рассмотрение этапа тестирования ПС начнем с того, что дадим основные определения процессов, которые можно выделить в тестировании.

Тестирование (testing) – процесс выполнения программы (или части программы) с намерением (или целью) найти ошибки.

Доказательство (proof) – попытка найти ошибки в программе безотносительно к внешней для программы среде. Большинство методов доказательства предполагает формулировку утверждений о поведении программы, и затем вывод, и доказательство математических теорем о правильности программы. Доказательства могут рассматриваться как форма тестирования, хотя они и не предполагают прямого выполнения программы. Многие исследователи считают доказательство альтернативой тестированию – взгляд во многом ошибочный.

Контроль (verification) – попытка найти ошибки, выполняя программу в тестовой, или моделируемой, среде.

Испытание (validation) – попытка найти ошибки, выполняя **программу в заданной реальной среде.**

Аттестация (certification) – авторитетное подтверждение правильности программы. При тестировании с целью аттестации выполняется сравнение с некоторым заранее определенным стандартом.

Отладка (debugging) не является разновидностью тестирования. Хотя слова «отладка» и «тестирование» часто используются как синонимы, под ними подразумеваются разные виды деятельности. Тестирование – деятельность, направленная на обнаружение ошибок; отладка направлена на установление точной природы известной ошибки, а затем – на исправление этой ошибки. Эти два вида деятельности связаны – результаты тестирования являются исходными данными для отладки.

Эти определения представляют один взгляд на тестирование – со стороны среды, на которую оно опирается. Другой ряд определений, приведенный ниже, охватывает вторую сторону тестирования: типы ошибок, которые предполагается обнаружить, и стандарты, с которыми сопоставляются тестируемые программы.

Тестирование модуля, или автономное тестирование (module testing, unit testing), – контроль отдельного программного модуля, обычно в изолированной среде (т.е. изолированно от всех остальных модулей). Тестирование модуля иногда включает также математическое доказательство.

Тестирование сопряжений (integration testing) – контроль сопряжений между частями системы (модулями, компонентами, подсистемами).

Тестирование внешних функций (external function testing) – контроль внешнего поведения системы, определенного внешними спецификациями.

Комплексное тестирование (system testing) – контроль и/или испытание системы по отношению к исходным целям. Комплексное тестирование является процессом контроля, если оно выполняется в моделируемой среде, и процессом испытания, если выполняется в среде реальной, жизненной.

Тестирование приемлемости (acceptance testing) – проверка соответствия программы требованиям пользователя.

Тестирование настройки (installation testing) – проверка соответствия каждого конкретного варианта установки системы с целью выявить любые ошибки, возникшие в процессе настройки системы.

11.2. Экономика тестирования

В общем случае невозможно обнаружить все ошибки программы. А это в свою очередь порождает экономические проблемы, задачи, связанные с функциями человека в процессе отладки, способы построения тестов. В стратегии тестирования существуют два основных подхода – тестирование программы как «черного ящика» и как «белого ящика», проанализировав которые, можно убедиться в невозможности исчерпывающего тестирования

11.2.1. Тестирование программы как «черного ящика»

Первым является подход называемый стратегией «черного ящика», ***тестированием с управлением по данным*** **или** ***тестированием с управлением по входу-выходу.*** При использовании этой стратегии программа рассматривается как «черный ящик». Иными словами, такое тестирование имеет целью выяснение обстоятельств, в которых поведение программы не соответствует спецификации. Тестовые же данные используются только в соответствии со спецификацией программы (т.е. без учета знаний о ее внутренней структуре).

При таком подходе обнаружение всех ошибок в программе является критерием ***исчерпывающего входного тестирования.*** Последнее

может быть достигнуто, если в качестве тестовых наборов использовать все возможные наборы входных данных.

Если такое испытание представляется сложным, то еще сложнее создать исчерпывающий тест для большой программы. Образно говоря, число тестов можно оценить «числом большим, чем бесконечность».

Построение исчерпывающего входного теста невозможно. Это подтверждается двумя аргументами: во-первых, нельзя создать тест, гарантирующий отсутствие ошибок; во-вторых, разработка таких тестов противоречит экономическим требованиям. Поскольку исчерпывающее тестирование исключается, нашей целью должна стать максимизация результативности капиталовложений в тестирование (иными словами, максимизация числа ошибок, обнаруживаемых одним тестом). Для этого можно рассматривать внутреннюю структуру программы и делать некоторые разумные, но, конечно, не обладающие полной гарантией достоверности предположения.

11.2.2. Тестирование программы как «белого ящика»

Стратегия ***«белого ящика»,*** или стратегия тестирования, ***управляемого логикой программы,*** позволяет исследовать внутреннюю структуру программы. В этом случае тестирующий получает тестовые данные путем анализа логики программы.

При таком подходе должно быть проведено ***исчерпывающее тестирование маршрутов.*** Подразумевается, что программа проверена полностью, если с помощью тестов удается осуществить выполнение программы по всем возможным маршрутам ее потока (графа) передач управления.

Последнее утверждение имеет два слабых пункта. Первый из них состоит в том, что число не повторяющих друг друга маршрутов в программе – астрономическое. Второй слабый пункт утверждения заключается в том, что, хотя исчерпывающее тестирование маршрутов является полным тестом и хотя каждый маршрут программы может быть проверен, сама программа будет содержать ошибки. Это объясняется следующим образом.

Во-первых, исчерпывающее тестирование маршрутов не может дать гарантии того, что программа соответствует описанию. Например, вместо требуемой программы сортировки по возрастанию случайно была написана программа сортировки по убыванию. В этом случае ценность тестирования маршрутов невелика, поскольку после тестирования в программе окажется одна ошибка, т.е. программа неверна.

Во-вторых, программа может быть неверной в силу того, что пропущены некоторые маршруты. Исчерпывающее тестирование маршрутов не обнаружит их отсутствия.

В-третьих, исчерпывающее тестирование маршрутов не может обнаружить ошибок, ***появление которых зависит от обрабатываемых данных.***

Эти рассуждения приводят ко второму фундаментальному принципу тестирования: ***тестирование – проблема в значительной степени экономическая.*** Поскольку исчерпывающее тестирование невозможно, мы должны ограничиться чем-то меньшим. Каждый тест должен давать максимальную отдачу по сравнению с нашими затратами. Эта отдача измеряется вероятностью того, что тест выявит не обнаруженную прежде ошибку. Затраты измеряются временем и стоимостью подготовки, выполнения и проверки результатов теста. Считая, что затраты ограничены бюджетом и графиком, можно утверждать, что искусство тестирования, по существу, представляет собой искусство отбора тестов с максимальной отдачей.

11.3. Аксиомы (принципы) тестирования

Сформулируем основные принципы тестирования, используя предпосылку, что наиболее важными в тестировании программ являются вопросы психологии. Эти принципы интересны тем, что в основном они интуитивно ясны, но в то же время на них часто не обращают должного внимания.

Хорош тот тест, для которого высока вероятность обнаржить ошибку. Эта аксиома является фундаментальным принципом тестирования. Поскольку невозможно показать, что программа не имеет ошибок и, значит, все такие попытки бесплодны, процесс тестирования должен представлять собой попытки обнаружить в программе прежде не найденные ошибки.

Одна из самых сложных проблем при тестировании – решить, когда нужно его закончить. Как уже говорилось, исчерпывающее тестирование (т.е. испытание всех входных значений) невозможно. Таким образом, при тестировании мы сталкиваемся с экономической проблемой: как выбрать конечное число тестов, которое дает максимальную отдачу (вероятность обнаружения ошибок) для данных затрат. Известно слишком много случаев, когда написанные тесты имели крайне малую вероятность обнаружения новых ошибок, в то время как довольно очевидные хорошие тесты оставались незамеченными.

Не нужно тестировать свою собственную программу. Ни один программист не должен пытаться тестировать свою собственную программу. Это относится ко всем формам тестирования – как к тестированию системы, так и к тестированию внешних функций и даже отдельных модулей. Тестирование должно быть в высшей степени разрушительным процессом, но имеются глубокие психологические причины,

по которым программист не может относиться к своей собственной программе как разрушитель. Дополнительное давление (например, жесткий график) на отдельного программиста или весь коллектив разработчиков проекта часто мешает программисту или всему коллективу выполнить адекватное тестирование. Если модуль содержит дефекты вследствие каких-то ошибок перевода, довольно высока вероятность того, что программист допустит ту же ошибку перевода (например, неправильно интерпретирует спецификации) и при подготовке тестов. Все ошибки в его понимании других модулей и их сопряжении также отразятся на тестах.

Тестирование всегда должна выполнять внешняя группа, которая в некотором смысле стоит особняком от программиста и проекта. Вместо того чтобы выполнять автономное тестирование модулей самостоятельно, программист должен иметь набор тестов, подготовленных разработчиком одного из модулей, вызывающих тестируемый модуль (а может быть, даже выполненных на машине и проверенных этим разработчиком). Комплексное тестирование всегда должно выполняться независимой группой, например специальным отделом обеспечения качества, или группой пользователей-добровольцев.

Отсюда вовсе не следует, что программист не может тестировать свою программу. Многие программисты с этим вполне успешно справляются. Здесь лишь делается вывод о том, что тестирование является более эффективным, если оно выполняется кем-либо другим. Заметим, что все наши рассуждения не относятся к отладке, т. е. к исправлению уже известных ошибок. Эта работа эффективнее выполняется самим автором программы.

Необходимая часть всякого теста – описание ожидаемых выходных данных или результатов. Одна из самых распространенных ошибок при тестировании состоит в том, что результаты каждого теста не прогнозируются до его выполнения. Ожидаемые результаты нужно определять заранее, чтобы не возникла ситуация, когда «глаз видит то, что хочет увидеть». Чтобы совсем исключить такую возможность, лучше разрабатывать самопроверяющиеся тесты либо пользоваться инструментами тестирования, способными автоматически сверять ожидаемые и фактические результаты.

Хотя эта аксиома чрезвычайно важна, иногда, например, при тестировании математического программного обеспечения приходится допускать исключения. Математическое программное обеспечение обладает тем свойством, что выходные данные являются только приближением правильного результата. Это происходит из-за использования конечных вычислительных процессов вместо бесконечных математических процессов, из-за ошибок округления, связанных с конечной точностью машинной арифметики и неточностью представления чисел, а

также ошибок из-за конечной точности представления входных данных и констант. Поэтому во многих случаях оказывается важной не абсолютная точность, а корреляция ошибок. Например, когда математическая программа возвращает массив чисел, может оказаться важным, чтобы вычисленное решение было точным решением для набора входных данных, аппроксимирующего реальные выходные данные. Поэтому при тестировании математического программного обеспечения прогнозирование точных выходных данных затруднительно.

Избегайте невоспроизводимых тестов, не тестируйте «с лету». Использование диалоговых систем иногда мешает хорошему тестированию. Для того чтобы тестировать программу в пакетной системе, программист обычно должен оформить тест в виде специальной ведущей программы или в виде файла тестовых данных. В условиях диалога программист слишком часто выполняет тестирование «с лету», т.е. сидя за терминалом, задает конкретные значения и выполняет программу, чтобы «посмотреть, что получится». Это неряшливая и по многим причинам нежелательная форма тестирования. Основной ее недостаток в том, что такие тесты мимолетны; они исчезают по окончании их выполнения. Всякий раз, когда программу понадобится тестировать повторно (например, после исправления ошибок или после модификации), придется придумывать тесты заново.

Тестирование обходится слишком дорого и без этих дополнительных расходов. Никогда не используйте тестов, которые тут же выбрасываются. Тесты следует документировать и хранить в такой форме, чтобы каждый мог использовать их повторно.

Готовьте тесты, как для правильных, так и для неправильных входных данных. Многие программисты ориентируются в своих тестах на «разумные» условия на входе, забывая о последствиях появления непредусмотренных или ошибочных входных данных. Однако многие ошибки, которые потом неожиданно обнаруживаются в работающих программах, проявляются вследствие никак не предусмотренных действий пользователя программы. Тесты, представляющие неожиданные или неправильные входные данные, часто лучше обнаруживают ошибки, чем правильные тесты.

Детально изучите результаты каждого теста. По всей вероятности, это наиболее очевидный принцип, но и ему часто не уделяется должного внимания. Самые изощренные тесты ничего не стоят, если их результаты удостаиваются лишь беглого взгляда. Тестирование программы означает большее, нежели выполнение достаточного количества тестов; оно также предполагает изучение результатов каждого теста. «Да, я уже проверял такую ситуацию, но такого как-то не заметил в выдаче», – довольно распространенная реакция программиста на обнаруженную новую ошибку.

По мере того как число ошибок, обнаруженных в некотором компоненте программного обеспечения, увеличивается, растет относительная вероятность существования в нем необнаруженных ошибок. Этот противоречащий интуиции феномен, иллюстрируемый рис. 43, означает, что ошибки образуют кластеры, т.е. встречаются группами. С ростом числа ошибок, обнаруженных в компоненте программы (например, в модуле, подсистеме, функции пользователя), увеличивается также вероятность существования в этом компоненте еще не обнаруженных ошибок. Если при тестировании двух модулей в них обнаружены одна и восемь ошибок соответственно, кривая на рис. 43 показывает, что для модуля с восьмью ошибками вероятность того, что в нем еще есть ошибки, выше.

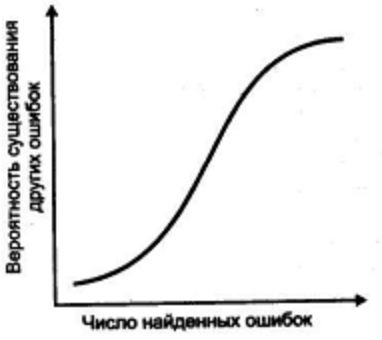


Рис. 43. График соотношения между обнаруженными и необнаруженными ошибками

Поручайте тестирование самым способным программистам. Тестирование и в особенности проектирование тестов – этап в разработке программного обеспечения, особенно требующий творческого подхода. К сожалению, во многих организациях на тестирование смотрят совсем не так. Его часто считают рутинной, нетворческой работой, вследствие чего коллективы, занимающиеся тестированием, укомплектованы в основном неопытными или молодыми программистами.

Однако практика показывает, что более правильным было бы сделать наоборот. Проектирование тестов требует даже больше творчества, чем разработка архитектуры и проектирование программного обеспечения.

Считайте тестируемость ключевой задачей вашей разработки.

Проект системы должен быть таким, чтобы каждый модуль подключался к системе только один раз. Множество проблем во многих больших программных системах возникает из-за нарушения этой аксиомы. Ситуация, когда во время цикла тестирования большой сис-

темы некоторые модули приходится подключать больше десяти раз, – не редкость. Каждая версия такого модуля содержит еще одну маленькую дополнительную функцию, необходимую для текущего уровня системы. Если следовать правилам и проектировать небольшие модули, каждый из которых выполняет отдельную функцию, бороться с этой проблемой станет легче.

Никогда не изменяйте программу, чтобы облегчить ее тестирование. Часто возникает соблазн изменить программу, чтобы было' легче ее тестировать. Например, программист, тестируя модуль, содержащий цикл, который должен повторяться 100 раз, меняет его так, чтобы цикл повторялся только 10 раз. Может быть, этот программист и занимается тестированием, но только другой программы.

Тестирование, как почти всякая другая деятельность, должно начинаться с постановки целей. Как уже говорилось, цикл тестирования подобен полному циклу разработки программного обеспечения. Тесты должны быть спроектированы, реализованы, проверены и, наконец, выполнены. Поэтому задачи тестирования должны быть сформулированы на каждом его этапе, например, для каждого конкретного типа тестирования должны быть определены ориентиры (число пройденных путей, проверенных условных переходов и т.п.) и доля ошибок, которые должны быть обнаружены на этом этапе.

Приведем еще раз три наиболее важных принципа тестирования.

Тестирование – это процесс выполнения программ с целью обнаружения ошибок.

Хорошим считается тест, который имеет высокую вероятность обнаружения еще не выявленной ошибки.

Удачным считается тест, который обнаруживает еще не выявленную ошибку.

11.4. Тестирование модулей

Вторым по важности аспектом тестирования после проектирования тестов является последовательность слияния всех модулей в систему или программу. Эта сторона вопроса обычно не получает достаточного внимания и часто рассматривается слишком поздно. Выбор этой последовательности, однако, является одним из самых жизненно важных решений, принимаемых на этапе тестирования, поскольку он определяет форму, в которой записываются тесты, типы необходимых инструментов тестирования, последовательность программирования модулей, а также тщательность и экономичность всего этапа тестирования. По этой причине такое решение должно приниматься на уровне проекта в целом и на достаточно ранней его стадии.

Тестирование модулей (или блоков) представляет собой процесс тестирования отдельных подпрограмм или процедур программы. Здесь подразумевается, что, прежде чем начинать тестирование программы в целом, следует протестировать отдельные небольшие модули, образующие эту программу. Такой подход мотивируется тремя причинами. Во-первых, появляется возможность управлять комбинаторикой тестирования, поскольку первоначально внимание концентрируется на небольших модулях программы. Во-вторых, облегчается задача отладки программы, т.е. обнаружение места ошибки и исправление текста программы. В-третьих, допускается параллелизм, что позволяет одновременно тестировать несколько модулей.

Цель тестирования модулей – сравнение функций, реализуемых модулем, со спецификациями его функций или интерфейса.

Тестирование модулей в основном ориентировано на принцип «белого ящика». Это объясняется, прежде всего, тем, что принцип «белого ящика» труднее реализовать при переходе в последующем к тестированию более крупных единиц, например программ в целом. Кроме того, последующие этапы тестирования ориентированы на обнаружение ошибок различного типа, т.е. ошибок, не обязательно связанных с логикой программы, а возникающих, например, из-за несоответствия программы требованиям пользователя.

Имеется большой выбор возможных подходов, которые могут быть использованы для слияния модулей в более крупные единицы. В большинстве своем они могут рассматриваться как варианты шести основных подходов, описанных ниже.

11.4.1. Пошаговое и монолитное тестирование

Реализация процесса тестирования модулей опирается на два ключевых положения: построение эффективного набора тестов и выбор способа, посредством которого модули комбинируются при построении из них рабочей программы. Второе положение является важным, так как оно задает форму написания тестов модуля, типы средств, используемых при тестировании, порядок кодирования и тестирования модулей, стоимость генерации тестов и стоимость отладки. Рассмотрим два подхода к комбинированию модулей: пошаговое и монолитное тестирование.

Возникает вопрос: «Что лучше – выполнить по отдельности тестирование каждого модуля, а затем, комбинируя их, сформировать рабочую программу или же каждый модуль для тестирования подключать к набору ранее оттестированных модулей?». Первый подход обычно называют ***монолитным методом,*** или ***методом «большого удара»,*** при тестировании и сборке программы; второй подход известен как ***пошаговый метод*** тестирования или сборки.

Метод пошагового тестирования предполагает, что модули тестируются не изолированно друг от друга, а подключаются поочередно для выполнения теста к набору уже ранее оттестированных модулей. Пошаговый процесс продолжается до тех пор, пока к набору оттестированных модулей не будет подключен последний модуль.

Детального разбора обоих методов мы делать не будем, приведем лишь некоторые общие выводы.

Монолитное тестирование требует больших затрат труда. При пошаговом же тестировании «снизу-вверх» затраты труда сокращаются.

Расход машинного времени при монолитном тестировании меньше.

Использование монолитного метода предоставляет большие возможности для параллельной организации работы на начальной фазе тестирования (тестирования всех модулей одновременно). Это положение может иметь важное значение при выполнении больших проектов, в которых много модулей и много исполнителей, поскольку численность персонала, участвующего в проекте, максимальна на начальной фазе.

При пошаговом тестировании раньше обнаруживаются ошибки в интерфейсах между модулями, поскольку раньше начинается сборка программы. В противоположность этому при монолитном тестировании модули «не видят друг друга» до последней фазы процесса тестирования.

Отладка программ при пошаговом тестировании легче. Если есть ошибки в межмодульных интерфейсах, а обычно так и бывает, то при монолитном тестировании они могут быть обнаружены лишь тогда, когда собрана вся программа. В этот момент локализовать ошибку довольно трудно, поскольку она может находиться в любом месте программы. Напротив, при пошаговом тестировании ошибки такого типа в основном связаны с тем модулем, который подключается последним.

Результаты пошагового тестирования более совершенны.

В заключение отметим, что п. 1, 4, 5, 6 демонстрируют преимущества пошагового тестирования, а п. 2 и 3 – его недостатки. Поскольку для современного этапа развития вычислительной техники характерны тенденции к уменьшению стоимости аппаратуры и увеличению стоимости труда, последствия ошибок в математическом обеспечении весьма серьезны, а стоимость устранения ошибки тем меньше, чем раньше она обнаружена; преимущества, указанные в п. 1, 4, 5, 6, выступают на первый план. В то же время ущерб, наносимый недостатками (п. 2 и 3), невелик. Все это позволяет нам сделать вывод, что ***пошаговое тестирование является предпочтительным.***

Убедившись в преимуществах пошагового тестирования перед монолитным, исследуем две возможные стратегии тестирования: ***нисходящее* и *восходящее.*** Прежде всего, внесем ясность в терминологию.

11.4.2. Восходящее тестирование

При восходящем подходе программа собирается и тестируется «снизу вверх». Только модули самого нижнего уровня («терминальные» модули; модули, не вызывающие других модулей) тестируются изолированно, автономно. После того как тестирование этих модулей завершено, вызов их должен быть так же надежен, как вызов встроенной функции языка или оператор присваивания. Затем тестируются модули, непосредственно вызывающие уже проверенные. Эти модули более высокого уровня тестируются не автономно, а вместе с уже проверенными модулями более низкого уровня. Процесс повторяется до тех пор, пока не будет достигнута вершина. Здесь завершается и тестирование модулей, и тестирование сопряжений программы.

При восходящем тестировании для каждого модуля необходим ***драйвер:*** нужно подавать тесты в соответствии с сопряжением тестируемого модуля. Одно из возможных решений – написать для каждого модуля небольшую ведущую программу. Тестовые данные представляются как «встроенные» непосредственно в эту программу переменные и структуры данных, и она многократно вызывает тестируемый модуль, с каждым вызовом передавая ему новые тестовые данные. Имеется и лучшее решение: воспользоваться программой тестирования модулей – это инструмент тестирования, позволяющий описывать тесты на специальном языке и избавляющий от необходимости писать драйверы.

11.4.3. Нисходящее тестирование

Нисходящее тестирование не является полной противоположностью восходящему, но в первом приближении может рассматриваться как таковое. При нисходящем подходе программа собирается и тестируется «сверху вниз». Изолированно тестируется только головной модуль. После того как тестирование этого модуля завершено, с ним соединяются (например, редактором связей) один за другим модули, непосредственно вызываемые им, и тестируется полученная комбинация. Процесс повторяется до тех пор, пока не будут собраны и проверены все модули.

При этом подходе немедленно возникают два вопроса: 1. «Что делать, когда тестируемый модуль вызывает модуль более низкого уровня (которого в данный момент еще не существует)?» и 2. «Как подаются тестовые данные?»

Ответ на первый вопрос состоит в том, что для имитации функций недостающих модулей программируются модули-***«заглушки»***, которые моделируют функции отсутствующих модулей.

Интересен и второй вопрос: в какой форме готовятся тестовые данные и как они передаются программе? Если бы головной модуль содержал все нужные операции ввода и вывода, ответ был бы прост: тесты пишутся в виде обычных для пользователей внешних данных и передаются программе через выделенные ей устройства ввода. Так, однако, случается редко. В хорошо спроектированной программе физические операции ввода-вывода выполняются на нижних уровнях структуры, поскольку физический ввод-вывод – абстракция довольно низкого уровня. Поэтому для того, чтобы решить проблему экономически эффективно, модули добавляются не в строго нисходящей последовательности (все модули одного горизонтального уровня, затем модули следующего уровня), а таким образом, чтобы ***обеспечить функционирование операций физического ввода-вывода как можно быстрее.*** Когда эта цель достигнута, нисходящее тестирование получает значительное преимущество: все дальнейшие тесты готовятся в той же форме, которая рассчитана на пользователя.

Нисходящий метод имеет как достоинства, так и недостатки по сравнению с восходящим. Самое значительное достоинство – то, что этот метод совмещает тестирование модуля, тестирование сопряжений и частично тестирование внешних функций. С этим же связано другое его достоинство: когда модули ввода-вывода уже подключены, тесты можно готовить в удобном виде. Нисходящий подход выгоден также в том случае, когда есть сомнения относительно осуществимости программы в целом или когда в проекте программы могут оказаться серьезные дефекты.

Преимуществом нисходящего подхода очень часто считают отсутствие необходимости в драйверах; вместо драйверов вам просто следует написать «заглушки».

Нисходящий метод тестирования имеет, к сожалению, некоторые недостатки. Основным из них является то, что модуль редко тестируется досконально сразу после его подключения. Дело в том, что основательное тестирование некоторых модулей может потребовать крайне изощренных заглушек.

11.4.4. Метод «больного скачка»

Вероятно, самый распространенный подход к интеграции модулей – метод «большого скачка». В соответствии с этим методом каждый модуль тестируется автономно. По окончании тестирования модулей они интегрируются в систему все сразу.

Метод «большого скачка» по сравнению с другими подходами имеет много недостатков и мало достоинств. Заглушки и драйверы необходимы для каждого модуля. Модули не интегрируются до самого

последнего момента, а это означает, что в течение долгого времени серьезные ошибки в сопряжениях могут остаться необнаруженными.

Если программа мала (как, например, программа загрузчика) и хорошо спроектирована, метод «большого скачка» может оказаться приемлемым. Однако для крупных программ метод «большого скачка» обычно губителен.

11.4.5. Метод сэндвича

Тестирование методом сандвича представляет собой компромисс между восходящим и нисходящим подходами. Здесь делается попытка воспользоваться достоинствами обоих методов, избежав их недостатков.

При использовании этого метода одновременно начинают восходящее и нисходящее тестирование, собирая программу как снизу, так и сверху и встречаясь, в конце концов, где-то в середине. Точка встречи зависит от конкретной тестируемой программы и должна быть заранее определена при изучении ее структуры.

Метод сандвича сохраняет такое достоинство нисходящего и восходящего подходов, как начало интеграции системы на самом раннем этапе. Поскольку вершина программы вступает в строй рано, проектировщик, как в нисходящем методе, уже на раннем этапе получаем работающий каркас программы. Так как нижние уровни программы создаются восходящим методом, снимаются проблемы нисходящего метода, которые были связаны с невозможностью тестировать некоторые условия в глубине программы.

11.4.6. Модифицированный метод сандвича

При тестировании методом сандвича возникает та же проблема, что и при нисходящем подходе, хотя здесь она стоит не так остро. Проблема эта в том, что невозможно досконально тестировать отдельные модули. Восходящий этап тестирования по методу сандвича решает эту проблему для модулей нижних уровней, но она может по-прежнему оставаться открытой для нижней половины верхней части программы. В модифицированном методе сандвича нижние уровни также тестируются строго снизу вверх. А модули верхних уровней сначала тестируются изолированно, а затем собираются нисходящим методом. Таким образом, модифицированный метод сандвича, также представляет собой компромисс между восходящим и нисходящим подходами.

11.5. Комплексное тестирование

Комплексное тестирование – процесс поисков несоответствия системы ее исходным целям. Элементами, участвующими в комплексном тестировании, служат сама система, описание целей продукта и вся до-

кументация, которая будет поставляться вместе с системой. Внешние спецификации, которые были ключевым элементом тестирования внешних функций, играют лишь незначительную роль в комплексном тестировании.

Если вы не сформулировали цели вашего продукта или если эти цели неизмеримы, вы не можете выполнить комплексное тестирование.

Комплексное тестирование может быть процессом и контроля, и испытаний. Процессом испытаний оно является тогда, когда выполняется в реальной среде пользователя или в обстановке, которая специально создана так, чтобы напоминать среду пользователя. Однако часто это невозможно по ряду причин, и в подобных случаях комплексное тестирование системы является процессом контроля (т.е. выполняется в имитируемой, или тестовой, среде). Например, в случае бортовой вычислительной системы космического корабля или системы противоракетной защиты. Кроме того некоторые типы комплексных тестов не осуществимы в реальной обстановке по экономическим соображениям, и лучше всего выполнять их в моделируемой среде.

11.5.1. Проектирование комплексного теста

Комплексное тестирование – наиболее творческий из всех обсуждавшихся до сих пор видов тестирования. Разработка хороших комплексных тестов требует часто даже больше изобретательности, чем само проектирование системы. Здесь нет простых рекомендаций типа тестирования всех ветвей или построения функциональных диаграмм. Однако следующие 15 пунктов дают некоторое представление о том, какие виды тестов могут понадобиться (рис. 43).

Тестирование стрессов. Распространенный недостаток больших систем состоит в том, что они функционируют как будто бы нормально при слабой или умеренной нагрузке, но выходят из строя при большой нагрузке и в стрессовых ситуациях реальной среды. Тестирование стрессов представляет собой попытки подвергнуть систему крайнему «давлению», например, попытку одновременно подключить к системе разделения времени 100 терминалов, насытить банковскую систему мощным потоком входных сообщений или систему управления – процессами аварийных сигналов от всех ее процессов.

Жесткие стрессовые ситуации возникают в реальной среде, когда все пользователи системы разделения времени пытаются подключиться в одно и то же время (например, когда произошел отказ системы на 1–2 минуты и система только что восстановлена). У банковских систем, обслуживающих терминалы покупателей, бывают пиковые нагрузки в первые часы работы магазинов и в час обеденного перерыва.

Тестирование объема. В то время как при тестировании стрессов делается попытка подвергнуть систему серьезным нагрузкам в короткий интервал времени, тестирование объема представляет собой попытку предъявить системе большие объемы данных в течение более длительного времени. На вход компилятора следует подать до нелепости громадную программу. Программа обработки текстов – огромный документ. Очередь заданий операционной системы следует заполнить до предела. Цель тестирования объема – показать, что система или программа не может обрабатывать данные в количествах, указанных в их спецификациях.

Тестирование конфигурации. Многие системы, например операционные системы или системы управления файлами, обеспечивают работу различных конфигураций аппаратуры и программного обеспечения. Число таких конфигураций часто слишком велико, чтобы можно было проверить все варианты. Однако следует тестировать по крайней мере максимальную и минимальную конфигурации. Система должна быть проверена со всяким аппаратным устройством, которое она обслуживает, или со всякой программой, с которой она должна взаимодействовать. Если сама программная система допускает несколько конфигураций (т.е. покупатель может выбрать только определенные части или варианты системы), должна быть тестирована каждая из них.

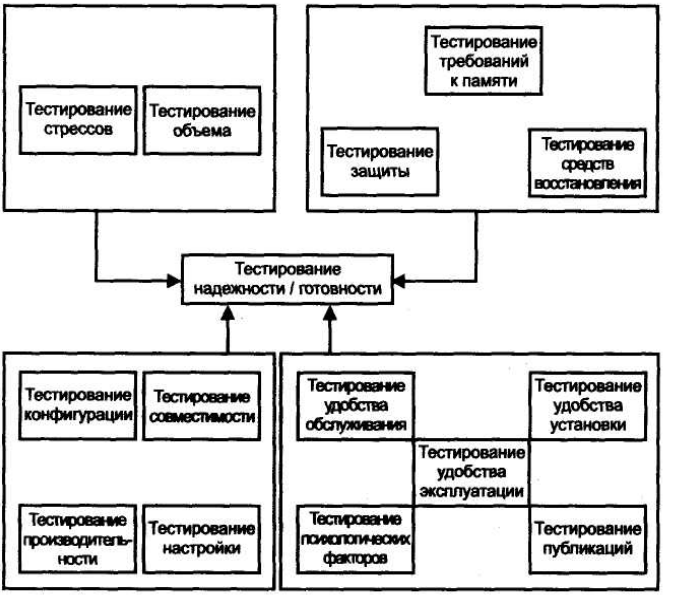


Рис. 44. Схема проектирования комплексного теста

Тестирование совместимости. В большинстве своем разрабатываемые системы не являются совершенно новыми; они представляют собой улучшение прежних версий или замену устаревших систем. В таких случаях на систему, вероятно, накладывается дополнительное требование совместимости, в соответствии с которым взаимодействие пользователя с прежней версией должно полностью сохраниться и в новой системе. Например, возможно, потребуется, чтобы в новом выпуске операционной системы язык управления заданиями, язык общения с терминалом и скомпилированные прикладные программы, использовавшиеся раньше, могли применяться без изменений. Такие требования совместимости следует тестировать. Как периодически подчеркивалось при разговоре обо всех других формах тестирования, цель при тестировании совместимости должна состоять в том, чтобы показать наличие несовместимости.

Тестирование защиты. Так как внимание к вопросам сохранения секретности в обществе возрастает, к большинству систем предъявляются определенные требования по обеспечению защиты от несанкционированного доступа. Например, операционная система должна устранить всякую возможность для программы пользователя увидеть данные или программу другого пользователя. Административная информационная система не должна позволять подчиненному получить сведения о зарплате тех, кто стоит выше его по служебной лестнице. Цель тестирования защиты – нарушить секретность в системе. Один из методов – нанять профессиональную группу «взломщиков», т.е. людей с опытом разрушения средств обеспечения защиты в системах. Для тестирования защиты важно построить такие тесты, которые нарушат программные средства защиты, например, механизм защиты памяти операционной системы или механизмы защиты данных системы управления базой данных.

Тестирование требований к памяти. При проектировании многих систем ставятся цели, определяющие объем основной и вторичной памяти, которую системе разрешено использовать в различных условиях. С помощью специальных тестов нужно попытаться показать, что система этих целей не достигает.

Тестирование производительности. При разработке многих программ ставится задача – обеспечить их производительность, или эффективность. Определяются такие характеристики, как время отклика и уровень пропускной способности при определенной нагрузке и конфигурации оборудования. Проверка системы в этих случаях сводится к демонстрации того, что данная программа не удовлетворяет поставленным целям. Поэтому необходимо разработать тесты, с помощью которых можно попытаться показать, что система не обладает требуемой производительностью.

Тестирование настройки. К сожалению, процедуры настройки многих систем сложны. Тестирование процесса настройки системы очень важно, поскольку одна из наиболее обескураживающих ситуаций, с которыми сталкивается покупатель, заключается в том, что он оказывается не в состоянии даже настроить новую систему.

Тестирование надежности/готовности. Конечно, назначение всех видов тестирования – повысить надежность тестируемой программы, но если в исходном документе, отражающем цели программы, есть особые указания, касающиеся надежности, можно разработать специальные тесты для ее проверки. Ключевой момент в комплексном тестировании заключается в попытке доказать, что система не удовлетворяет исходным требованиям к надежности (среднее время между отказами, количество ошибок, способность к обнаружению, исправлению ошибок и/или устойчивость к ошибкам и т.д.). Тестирование надежности крайне сложно, и все же следует постараться тестировать как можно больше этих требований.

Тестирование средств восстановления. Важная составная часть требований к операционным системам, системам управления базами данных и системам передачи данных – обеспечение способности к восстановлению, например восстановлению утраченных данных в базе данных или восстановлению после отказа в телекоммуникационной линии. Лучше всего попытаться показать, что эти средства работают неправильно, при комплексном тестировании системы.

Тестирование удобства обслуживания. Либо в требованиях к продукту, либо в требованиях к проекту должны быть перечислены задачи, определяющие удобство обслуживания (сопровождения) системы. Все сервисные средства системы нужно проверять при комплексном тестировании. Все документы, описывающие внутреннюю логику, следует проанализировать глазами обслуживающего персонала, чтобы понять, как быстро и точно можно указать причину ошибки, если известны только некоторые се симптомы. Все средства, обеспечивающие сопровождение и поставляемые вместе с системой, также должны быть проверены.

Тестирование публикаций. Проверка точности всей документации для пользователя является важной частью комплексного тестирования. Все комплексные тесты следует строить только на основе документации для пользователя. Ее проверяют при определении правильности представления предшествующих тестов системы. Пользовательская документация должна быть и предметом инспекции при проверке ее на точность и ясность. Любые примеры, приведенные в документации, следует оформить как тест и подать на вход программы.

Тестирование психологических факторов. Хотя во время тестирования системы следует проверить и психологические факторы, эта

сторона не так важна, как другие, потому что обычно при тестировании уже слишком поздно исправлять серьезные просчеты в таких вопросах. Однако мелкие недостатки могут быть обнаружены и устранены при тестировании системы. Например, может оказаться, что ответы или сообщения системы плохо сформулированы или ввод команды пользователя требует постоянных переключений верхнего и нижнего регистров.

Тестирование удобства установки. Процедуры установки (настройки) некоторых типов систем программного обеспечения весьма сложны. Тестирование подобных процедур является частью процесса тестирования системы.

Тестирование удобства эксплуатации. Не менее важным видом тестирования системы является попытка выявления психологических (пользовательских) проблем, или проблем удобства эксплуатации. Анализ психологических факторов является, к сожалению, до сих пор весьма субъективным, так как при производстве вычислительной техники уделялось недостаточное внимание изучению и определению психологических аспектов программных систем. Большинство систем обработки данных либо является компонентами более крупных систем, предполагающих деятельность человека, либо сами регламентируют такую деятельность во время своей работы. Нужно проверить, что вся эта деятельность (например, поведение оператора или пользователя за терминалом) удовлетворяет определенным условиям.

Основное правило при комплексном тестировании – все годится. Пишите разрушительные тесты, проверяйте все функциональные границы системы, пишите тесты, представляющие ошибки пользователя (например, оператора системы). Нужно, чтобы тестовик думал так же, как пользователь или покупатель, а это предполагает доскональное понимание того, для чего система будет применяться.

Как уже упоминалось, компонентами комплексного теста являются исходные цели, документация, публикации для пользователей и сама система. Все комплексные тесты должны быть подготовлены на основе публикаций для пользователя (а не внешних спецификаций). К внешним спецификациям обращаться следует только для того, чтобы разбираться в противоречиях между системой и публикациями о ней.

11.5.2. Выполнение комплексного теста

Один из методов, позволяющих вовлечь в тестирование пользователей, – ***опытная эксплуатация***. Для проведения опытной эксплуатации с одной или несколькими организациями пользователей заключаются контракты на установку у них созданной системы. Это часто выгодно обеим сторонам: организация-разработчик оповещается об ошибках в программном обеспечении, которые она не заметила сама, а орга-

низация-пользователь получает возможность изучить систему и экспериментировать с ней до того, как она станет доступной официально.

Второй полезный метод – использовать систему в организации-изготовителе для внутренних нужд. Это можно сделать часто, но далеко не всегда (например, компании по разработке программного обеспечения негде использовать систему управления процессом очистки нефти).

Воспользуйтесь собственным продуктом, прежде чем передавать его другим.

При комплексном тестировании часто начинают с простых тестов, приберегая более сложные тесты к концу. Так делать не нужно, потому что комплексное тестирование приходится на самый конец цикла разработки, так что на отладку и исправление найденных ошибок остается мало времени. Поскольку сложные тесты часто обнаруживают более сложные для исправления ошибки, измените последовательность: начните с самых трудных тестов, а затем переходите к более простым [49].

11.6. ГОСТ Р ИСО/МЭК 12119-2000

Рассмотрим требования, которые предъявляет ГОСТ Р ИСО/ МЭК 12119-2000 к тестированию пакетов программ.

ГОСТ Р ИСО/МЭК 12119-2000 содержит указания, которые определяют порядок тестирования продукта на соответствие его требованиям к качеству. Эти указания охватывают как тестирование для характеристик, присущих аналогичным продуктам, так и тестирование для характеристик, указанных в описании продукта. Указания охватывают тестирование путем проверки документов и тестирование программ и данных по принципу «черного ящика».

ГОСТ описывает только функциональное тестирование (тестирование по принципу «черного ящика»), а структурное тестирование не охватывается, потому что для его проведения необходимо наличие исходного кода. В нем рассматривают только тестирование продукта в необходимых для него системах. Эргономическую оценку на рабочем пространстве вычислительной системы в настоящем стандарте тоже не рассматривают.

11.6.1. Работы по тестированию

Описание продукта, документация пользователя, программы и любые данные, поставляемые как части пакета программ, должны быть протестированы на выполнение ими формулировок и требований.

Программы должны быть протестированы во всех вычислительных системах, указанных в описании продукта. При наличии нескольких вариантов программы должен быть протестирован каждый из них. Любая из функций, которые в соответствии с описанием продукта и доку-

ментацией пользователя одинаковы в ряде вариантов, может быть протестирована в одном из вариантов. Программы и данные должны быть протестированы с использованием контрольных примеров, разработанных на основе описания продукта и документации пользователя. Другие материалы (например, исходные программы) не проверяют, за исключением случаев, когда это необходимо при тестировании формулировок из описания продукта или документации пользователя.

Контрольные примеры должны быть методологически и систематически проработаны.

Если в документации пользователя приведены примеры, то они должны быть использованы в качестве контрольных, но проводимое тестирование не должно быть ограничено только этими примерами.

Могут быть использованы контрольные примеры, предоставляемые поставщиком программного пакета, но проводимое тестирование не должно быть ограничено только этими примерами.

Установка (инсталляция). Если в соответствии с описанием продукта установка пакета может быть выполнена пользователем, должна быть проверена возможность инсталляции программ и протестирована возможность успешной установки пакета согласно описанию, приведенному в руководстве по установке.

Любым способом должно быть обеспечено, чтобы техническая и программная среда, в которой установлены программы, соответствовала формулировкам из описания продукта в части рассматриваемой вычислительной системы.

Выполнение программы. Контрольные примеры должны охватывать все функции, приведенные в описании продукта и документации пользователя, а также учитывать комбинации функций, характерные для рабочей задачи.

Программы должны быть протестированы по всем граничным значениям (в соответствии с описанием продукта и документацией пользователя) в необходимой системе, для которой заданы эти значения.

При тестировании должны быть использованы исходные данные и последовательности команд, которые в документации пользователя явно не рекомендуются или объявляются запрещенными.

11.6.2. Протоколы тестирования

Протоколы по каждому тесту должны содержать информацию, достаточную для повторения теста. Данная информация должна включать:

- план тестирования или технические требования (спецификацию) к тестированию, содержащие контрольные примеры (для каждого контрольного примера указаны его цели);

• все результаты, связанные с контрольными примерами, включая все ошибки, выявленные при выполнении теста;

• штат персонала, вовлеченного в тестирование.

11.6.3. Отчет о тестировании

В отчете о тестировании должны быть суммированы цели и результаты тестирования (описанные в протоколах тестирования для каждого теста). Отчет о тестировании должен иметь следующую структуру.

• Обозначение продукта.

• Вычислительные системы, использованные при тестировании (технические средства, программные средства и их конфигурация).

• Использованные документы (включая их обозначения).

• Результаты тестирования описания продукта, документации пользователя, программ и данных.

• Перечень несоответствий требованиям.

• Перечень несоответствий рекомендациям либо перечень неучтенных в продукте рекомендаций, либо формулировка того, что продукт не был протестирован на соответствие рекомендациям.

• Дата окончания тестирования.

11.6.4. Дополнительное тестирование

Когда продукт, который уже был протестирован, тестируется повторно (с учетом результатов предыдущего тестирования), тогда выполняются следующие требования:

• все измененные части документов, функций и данных должны быть протестированы как новый продукт;

• все неизмененные части, на которые могут влиять измененные части или изменения в необходимой системе (в соответствии с опытной оценкой тестировщика), должны быть протестированы как новый продукт;

• все другие части должны быть, по крайней мере, выборочно протестированы.

Контрольные вопросы

1. Что такое тестирование программы?
2. Чем отличается процесс тестирования от процесса отладки?
3. Перечислите принципы тестирования.
4. Какие методы тестирования вы знаете?
5. Что понимают под процессом сборки модулей, какие методы сборки вы знаете?
6. Какие виды ошибок вы знаете?

7. Когда должна заканчиваться стадия тестирования ПО?

8. Как можно охарактеризовать процесс тестирования по стоимости и продолжительности?

9. Что такое тестирование «белого ящика»?

10. Что такое тестирование «черного ящика»?

11. Расскажите про метод сандвича.

12. В чем заключается метод большого скачка?

13. Как узнать о необходимости завершения тестирования?

14. Можно ли на практике обнаружить все ошибки в программном средстве, если можно, то как это сделать?

15. Опишите место и роль тестирования в процессе разработки программного обеспечения.

16. Перечислите основные аксиомы (принципы) тестирования.

17. Что представляет собой тестирование психологических факторов?

12. НАДЕЖНОСТЬ И КАЧЕСТВО ПРОГРАММНЫХ СРЕДСТВ

Опыт создания и применения сложных информационных систем (ИС) в последние десятилетия выявил множество ситуаций, при которых сбои и отказы их функционирования были обусловлены дефектами комплексов программ, что приводило к большому экономическому ущербу. Вследствие ошибок в программах автоматического управления погибло несколько отечественных, американских и французских спутников, происходили отказы и катастрофы в сложных административных, банковских и технологических информационных системах.

В результате около двадцати лет назад появились первые обобщающие работы, в которых были сформулированы концепция и основные положения теории надежности программных средств, для информационных систем. В это время были заложены основы методологии и технологии создания высоконадежных сложных комплексов программ. Стало ясно, что для обеспечения высокой надежности функционирования и безопасности применения, создаваемых сложных комплексов программ необходимы четкая организация и высокая квалификация всего коллектива специалистов, участвующих в таком проекте. В коллективе разработчиков целесообразно выделять специалистов, ответственных за соблюдение технологии создания и развития программ, за обеспечение и контроль качества, а также за надежность и безопасность проекта ПС и его компонентов.

Обеспечение качества и надежности должно реализовываться специалистами в жизненном цикле программных средств на основе использования современной методологии, технологического инструментария, стандартов и нормативных документов. Для обеспечения качества и надежности программных средств необходимы разработка и применение эффективных методов и средств, предупреждающих и выявляющих дефекты, а также удостоверяющих надежность программ и оперативно защищающих функционирование ПС при их проявлениях. Для систематической, координированной борьбы с угрозами надежности должны проводиться исследования конкретных факторов, влияющих на качество функционирования и безопасность применения программ со стороны реально существующих и потенциально возможных дефектов в создаваемых комплексах программ. В каждом проекте должен целенаправленно разрабатываться скоординированный комплекс методов и средств обеспечения заданной надежности функционирования ПС при реально достижимом снижении уровня дефектов и ошибок разработки.

Одним из эффективных путей повышения качества и надежности ПС является ***стандартизация технологических процессов и объектов*** проектирования, разработки и сопровождения программ. В стандартах

жизненного цикла ПС обобщаются опыт и результаты исследований множества специалистов и рекомендуются наиболее эффективные современные методы и процессы. В результате таких обобщений отрабатываются технологические процессы и приемы разработки, а также методическая база для их автоматизации. Стандарты ЖЦ ПС могут использоваться как непосредственно директивные, руководящие или как рекомендательные документы, а также как организационная база при создании средств автоматизации соответствующих технологических этапов или процессов. Подобная стандартизация процессов отражается не только на их технико-экономических показателях, но и, что особенно важно, на качестве создаваемых ПС. Надежность программ тесно связана с методами и технологией их разработки, поэтому важной группой стандартов в этой области являются стандарты по обеспечению качества ПС.

Поддержка этапов и работ ЖЦ ПС международными стандартами весьма неравномерная. Наиболее полно стандартизированы этапы ЖЦ ПС, прошедшие длительное историческое развитие и требующие наименее квалифицированных специалистов. При создании сложных проектов ПС и обеспечении их ЖЦ целесообразно применять выборку из всей совокупности существующих стандартов, а имеющиеся весьма обширные пробелы в стандартизации заполнять утвержденными технологическими документами, регламентирующими применение выбранных средств автоматизации разработки ПС. В результате на начальном этапе проектирования следует формировать весь комплект документов – ***профиль,*** обеспечивающий регламентирование всех этапов и работ при создании надежных ПС. Для реализации положений этих документов должны быть выбраны инструментальные средства, совместно образующие взаимосвязанный комплекс технологической поддержки и автоматизации ЖЦ и не противоречащие предварительно скомпонованному набору нормативных документов профиля. Применение профилей при проектировании ПС позволяет ориентироваться на построение систем из крупных функциональных узлов, отвечающих требованиям стандартов профиля, применять достаточно, отработанные и проверенные проектные методы и решения.

12.1. Показатели качества программных средств

Существует множество определений качества, в основе понятия качества продукта или услуги лежит идея об удовлетворении потребностей конечного пользователя – реального или потенциального потребителя. Вот определение этого понятия в соответствии со стандартом ISO 8402:1994.

Качество – совокупность характеристик объекта, относящихся к его способности удовлетворить установленные и предполагаемые потребности.

Можно выделить три большие группы факторов, влияющих на качество программного обеспечения:

функциональная – связана с полнотой и удобством использования реализованных функций программного средства;

административная – связана с квалификацией персонала, организационной структурой и управлением персоналом;

программно-архитектурная – связана с процессом разработки программного обеспечения, выбранными методологиями, инструментальными средствами, использованными на различных этапах жизненного цикла программного обеспечения, а также архитектурой программного средства. Формализации показателей качества программных средств посвящена группа нормативных документов. В международном стандарте ***ISO 9126:1991*** при отборе минимума стандартизируемых показателей выдвигались и учитывались следующие принципы: ясность и измеряемость значений, отсутствие перекрытия между используемыми показателями, соответствие установившимся понятиям и терминологии, возможность последующего уточнения и детализации. Выделены характеристики, которые позволяют оценивать ПС с позиции пользователя, разработчика и управляющего проектом. Рекомендуются 6 основных характеристик качества ПС, каждая из которых детализируется несколькими (всего 21) субхарактеристиками (рис. 45).

Функциональная пригодность детализируется пригодностью для применения, точностью, защищенностью, способностью к взаимодействию и согласованностью со стандартами и правилами проектирования.

Надежность рекомендуется характеризовать уровнем завершенности (отсутствия ошибок), устойчивостью к ошибкам и перезапускаемостью.

Применимость предлагается описывать понятностью, обучаемостью и простотой использования.

Эффективность рекомендуется характеризовать ресурсной и временной экономичностью.

Сопровождаемость характеризуется удобством для анализа, изменяемостью, стабильностью и тестируемостью.

Переносимость предлагается отражать адаптируемостью, структурированностью, замещаемостью и внедряемостью.

Характеристики и субхарактеристики в стандарте определены очень кратко, без комментариев и рекомендаций по их применению к конкретным системам и проектам.

Рис. 45. Схема характеристик качества ПС по стандарту ISO 9126:1991

Близким к описанному стандарту по идеологии, структуре и содержанию является **ГОСТ 28195-459.** На верхнем, первом, уровне выделено 6 показателей – факторов качества: надежность, корректность, удобство применения, эффективность, универсальность и сопровождаемость. Эти факторы детализируются в совокупности 19 критериями качества на втором уровне. Дальнейшая детализация показателей качества представлена метриками и оценочными элементами, которых насчитывается около 240. Каждый из них рекомендуется экспертно оценивать в пределах от 0 до 1. Состав используемых факторов, критериев и метрик предлагается выбирать в зависимости от назначения, функций и этапов жизненного цикла ПС.

В стандарте **ГОСТ 28806-90** формализуются общие понятия программы, программного средства, программного продукта и их качества. Даются определения 18 наиболее употребляемых терминов, связанных с оценкой характеристик программ. Уточнены понятия базовых показателей качества, приведенных в ГОСТ 28195-89.

12.2. Обеспечение качества программного обеспечения

Программное обеспечение является важной составляющей многих сфер жизни, используется повсеместно в промышленности, медицине, активно начинает использоваться в образовании (дистанционное образование, открытое образование). От программного обеспечения зависит не только эффективность производственного процесса, но и жизнь людей (медицина, военная, космическая сфера). По этой причине встает вопрос о качестве программного обеспечения.

Современная техника управления качеством (например, концепция Total Quality Management (TQM)) базируется именно на управлении качеством. На современном этапе уже недостаточно иметь только методы оценки качества произведенного и используемого программного средства (выходной контроль), необходимо иметь возможность планировать качество, измерять его на всех этапах жизненного цикла программного средства и корректировать процесс производства программного обеспечения для улучшения качества. Международные стандарты серии ISO 9000 регламентируют создание системы управления качеством. Однако они являются общими, лишь рекомендательными. Каждая компания, производящая программное обеспечение и желающая внедрить у себя действенную систему управления качеством на основе стандартов ISO 9000-й серии, должна учесть специфику своей отрасли и разработать систему показателей качества, которая бы отражала реальное влияние факторов качества на программный продукт.

Программное обеспечение как продукт имеет некоторые отличия от других промышленных продуктов:

- наращивание объемов выпуска какого-то вида программного продукта происходит практически мгновенно и имеет низкую стоимость, так как производство следующей единицы программного продукта связано только с копированием информации на носитель (компакт-диск, дискету или жесткий диск);
- большие ресурсы затрачиваются на стадии планирования, реализации и тестирования;
- сильное влияние человеческого фактора на производство программного продукта, так как производство программного продукта – интеллектуальная и творческая деятельность;
- в жизненном цикле программного продукта, как правило, отсутствует этап утилизации;
- программный продукт не подвержен физическому старению, а только моральному.

Все эти, а также многие другие особенности должны быть учтены в программе оценки качества и управления качеством.

Сейчас остро стоит задача измерения качества программного обеспечения с целью оперативного воздействия на процесс производства программного продукта. Для измерения некоторых показателей качества могут служить тестирование, тестирование пользователем (так называемое (тестирование), а также информация от пользователя о найденных проблемах, получаемая от службы технической поддержки. Вышеперечисленные действия дают обильную пищу для анализа (выраженную в количественных единицах, а значит, измеряемую). Главное – найти между ними зависимости (например, зависимость количества ошибок, обнаруженных специалистом по тестированию, и количества ошибок, зафиксированных пользователем, может служить показателем надежности программного средства), тогда можно будет говорить об измерении качества программного средства.

При построении системы качества могут быть использованы математические методы: методы корреляционного анализа (для выяснения выявления зависимости и тесноты связи между отдельными свойствами программного продукта и степенью удовлетворения пользователя), методы факторного анализа (для построения функции качества), методы кластеризации.

Сегодня наступил этап планирования качества программного обеспечения, мониторинга качества и управления им в процессе производства. Заинтересованность пользователя и производителя программных средств есть; аппарат для управления качеством программного обеспечения разрабатывается зарубежными и российскими учеными.

Мероприятия, обеспечивающие приемлемый уровень качества программного средства, можно условно разделить на административные и технологические.

К административным можно отнести следующие мероприятия:

- Проведение обучения персонала, переподготовки.
- Тщательное документирование всех изменений в структуре программного средства. Для этого используются средства поддержки версионности.
- Назначение ответственных лиц за каждую доработку программного средства.
- Уделение внимания текущему контролю качества и заключительному контролю качества.
- Обеспечение мониторинга качества, например, фиксирование ошибок, поступивших от пользователя программного средства. Использование систематических испытательных методов, где испытания будут разработаны параллельно с разработкой программы.
- Введение внутренних стандартов. Такие стандарты обычно содержат соглашения о именовании переменных в программном коде, наименовании файлов данных, процедур и функций.
- Организация отдела тестирования как самостоятельного подразделения.
- Проведение совместных аттестаций с пользователем.
- Обращение внимания на уровень и простоту обслуживаемости программного обеспечения. Здесь речь идет как о решении проблем, возникших у пользователя, так и о простоте и надежности внесения изменений в программное обеспечение. Даже если очень надежный кусок программного обеспечения был разработан, он может вскоре стать ненадежным, если сложно сделать изменения в программе. Часто изменения должны быть выполнены вследствие новых потребностей внешней среды, например вследствие изменений, в законодательстве или требований заказчика.

К технологическим относятся следующие мероприятия.

- Выбор стандарта качества и четкое следование ему на всех этапах. Создание модели проекта с регулярными проверками, которые будут выполняться независимыми командами экспертизы. Такая модель может быть построена, например, на основе стандартов качества (например, ISO 9000).
- Единая среда разработки. Лучшие результаты дают программные продукты разработки, которые поддерживают несколько или все этапы жизненного цикла программного обеспечения. На данный момент такими комплексными решениями являются, например, продукты Oracle Designer, продукты фирмы Rational.

- Использовать формальный язык спецификаций (например, UML, DESIGN IDEF).
- Выбор надежной СУБД (если программное средство работает с массивами информации и использование СУБД оправдано).
- Тщательное тестирование программного обеспечения.
- Широкое внедрение автоматизации тестирования.
- Использование полностью проверенной программной среды окружения (ОС) и языка программирования, которые минимизируют опасность внесения ошибки.
- Использование статистических методов для сбора информации о качестве ПС.
- Изучение результатов испытаний (тестов) и ошибок для использования в постоянном усовершенствовании программы. Источник в случае возникновения отказа должен быть найден и устранен. Недостаточно найти ошибку в программном обеспечении и исправить ее. Изменения должны быть сделаны в процессе разработки ПО.
- Использование испытательной среды, которая предостережет от передачи пользователю ненадежного программного обеспечения. Создание автоматических средств приемки.

Как видно, качество программного обеспечения тесно связано с жизненным циклом программного обеспечения и тестированием.

Контрольные вопросы

1. Каковы цели управления разработкой ПО?
2. Какие характеристики качества выделяет стандарт ISO 9126:1991?
3. Перечислите основные характеристики качества ПО?
4. Как происходит оценка качества ПО?
5. Какие показатели качества выделяет ГОСТ 28195-459?

13. ОСНОВНЫЕ ПОНЯТИЯ И ПОКАЗАТЕЛИ НАДЕЖНОСТИ ПРОГРАММНЫХ СРЕДСТВ

Основные понятия надежности систем. По определению, установленному в ГОСТ 13377-75, ***надежность*** – свойство объекта выполнять заданные функции, сохраняя во времени значения установленных эксплуатационных показателей в заданных пределах, соответствующих заданным режимам и условиям использования, технического обслуживания, ремонта, хранения и транспортирования. Таким образом, надежность является внутренним свойством системы, заложенным при ее создании и проявляющимся ***во времени*** при функционировании и эксплуатации.

Свойства надежности программных систем изучаются ***теорией надежности программных средств***, которая является системой определенных идей, математических моделей и методов, направленных на решение проблем предсказания, оценки и оптимизации различных показателей надежности. Предметом изучения теории надежности комплексов программ (Software Reliability) является работоспособность сложных программ обработки информации в реальном времени. К задачам теории и анализа надежности сложных программных средств можно отнести следующие:

- формулирование основных понятий, используемых при исследовании и применении показателей надежности программных средств;
- выявление и исследование основных факторов, определяющих характеристики надежности сложных программных комплексов;
- выбор и обоснование критериев надежности для комплексов программ различного типа и назначения;
- исследование дефектов и ошибок, динамики их изменения при отладке и сопровождении, а также влияния на показатели надежности программных средств;
- исследование и разработка методов структурного построения сложных ПС, обеспечивающих их необходимую надежность;
- исследование методов и средств контроля и защиты от искажений программ, вычислительного процесса и данных путем использования различных видов избыточности и помехозащиты;
- разработка методов и средств определения и прогнозирования характеристик надежности в жизненном цикле комплексов программ с учетом их функционального назначения, сложности, структурного построения и технологии разработки.

Результаты решения этих задач являются основой для создания современных сложных программных средств с заданными показателями надежности. Использование и объединение результатов эксперимен-

тальных и теоретических исследований надежности ПС позволили заложить основы теории и методов в этой области. В жизненном цикле ПС значения показателей качества и надежности компонентов и комплексов программ в целом рекомендуется непрерывно анализировать и прогнозировать с целью гарантированного обеспечения заданных показателей надежности. В реальных проектах работы по исследованию и обеспечению надежности программ целесообразно выделять в отдельную группу под единым руководством со специальным планом.

В основе теории надежности лежат понятия о двух возможных состояниях объекта или системы: работоспособном и неработоспособном. ***Работоспособным*** называется такое состояние объекта, при котором он способен выполнять заданные функции с параметрами, установленными технической документацией. В процессе функционирования возможен переход объекта из работоспособного состояния в неработоспособное и обратно. С этими переходами связаны события отказа и восстановления.

Определение степени работоспособности системы предполагает наличие в ней средств, способных установить соответствие ее характеристик требованиям технической документации. Для этого должны использоваться ***методы и средства контроля и диагностики функционирования системы.*** Глубина и полнота проверок, степень автоматизации контрольных операций, длительность и порядок их выполнения влияют на работоспособность системы и достоверность ее оценки. Методы и средства диагностического контроля предназначены для установления степени работоспособности системы, локализации отказов, определения их характеристик и причин, скорейшего восстановления работоспособности, для накопления, обобщения и анализа данных, характеризующих работоспособность системы. Диагноз состояния системы принято делить на тестовый и функциональный. При тестовом диагнозе используются специально подготовленные исходные данные и эталонные результаты, которые позволяют проверять работоспособность определенных компонентов системы. Функциональный диагноз организуется на базе реальных исходных данных, поступающих в систему при ее использовании по прямому назначению. Основные задачи технической диагностики включают в себя:

- контроль исправности системы и полного соответствия ее состояния и функций технической документации;
- проверку работоспособности системы и возможности выполнения всех функций в заданном режиме работы в любой момент времени с характеристиками, заданными технической документацией;
- поиск, выявление и локализацию источников и результатов сбоев, отказов и неисправностей в системе.

Надежность функционирования ПС наиболее широко характеризуется ***устойчивостью,*** или способностью к безотказному функциониро-

ванию, и ***восстанавливаемостью*** работоспособного состояния после произошедших сбоев или отказов. В свою очередь, устойчивость зависит от уровня неустраненных дефектов и ошибок и способности ПС реагировать на их проявления так, чтобы это не отражалось на показателях надежности. Последнее определяется эффективностью контроля данных, поступающих из внешней среды, и средств обнаружения аномалий функционирования ПС.

Восстанавливаемость характеризуется полнотой и длительностью восстановления функционирования программ в процессе перезапуска – рестарта. Перезапуск должен обеспечивать возобновление нормального функционирования ПС, на что требуются ресурсы ЭВМ и время. Поэтому полнота и длительность восстановления функционирования после сбоев отражают качество, надежность ПС и возможность его использования по прямому назначению.

Показатели надежности ПС в значительной степени адекватны аналогичным характеристикам, принятым для других технических систем. Наиболее широко используется ***критерий длительности наработки на отказ.*** Для определения этой величины измеряется время работоспособного состояния системы между двумя последовательными отказами или началом нормального функционирования системы после них. Вероятностные характеристики этой величины в нескольких формах используются как разновидности критериев надежности. Критерий надежности восстанавливаемых систем учитывает возможность многократных отказов и восстановлений. Для оценки надежности таких систем, которыми чаще всего являются сложные ПС, кроме вероятностных характеристик наработки на отказ, важную роль играют характеристики функционирования после отказа в процессе восстановления. Основным показателем процесса восстановления являются ***длительность восстановления*** и ее вероятностные характеристики. Этот критерий учитывает возможность многократных отказов и восстановлений. Обобщение характеристик отказов и восстановлений производится в критерии ***коэффициент готовности.*** Этот показатель отражает вероятность иметь восстанавливаемую систему в работоспособном состоянии в произвольный момент времени. Значение коэффициента готовности соответствует доле времени полезной работы системы на достаточно большом интервале, содержащем отказы и восстановления.

Наработка на отказ учитывает ситуации потери работоспособности, когда длительность восстановления достаточно велика и превышает пороговое значение времени, разделяющее события сбоя и отказа. При этом большое значение имеют средства оперативного контроля и восстановления. Качество проведенной отладки программ более полно отражает значение длительности между потерями работоспособности программ – ***наработка на отказовую ситуацию, или устойчивость,*** не-

зависимо от того, насколько быстро произошло восстановление. Средства оперативного контроля и восстановления не влияют на наработку на отказовую ситуацию, однако могут значительно улучшать показатели надежности программ. Поэтому при оценке необходимой отладки целесообразно измерять и контролировать наработку на отказовую ситуацию, а объем и длительность тестирования в ряде случаев устанавливать по наработке на отказ с учетом эффективности средств рестарта.

В реальных системах достигаемая при отладке и испытаниях наработка на отказ ПС обычно должна быть соизмерима с наработкой на отказ аппаратуры, на которой исполняется программа. Для систем обработки информации и управления в реальном времени наработка на отказ программ измеряется десятками и сотнями часов, а для особо важных или широко тиражируемых систем может достигать десятков тысяч часов. При достаточно развитом программном оперативном контроле и восстановлении наработка на отказовую ситуацию может быть на один – два порядка меньше, чем наработка на отказ. Реально очень трудно достичь наработки на отказовую ситуацию на уровне сотен часов, и поэтому для получения высокой надежности программ невозможно ограничиваться тестированием и отладкой без использования средств рестарта. Невозможно обеспечить абсолютное отсутствие дефектов проектирования в сложных ПС, вследствие чего надежность их функционирования имеет всегда конечное, ограниченное значение.

13.1. Дестабилизирующие факторы и методы обеспечения надежности функционирования программных средств

При любом виде деятельности людям свойственно непредумышленно ошибаться, результаты чего проявляются в процессе создания или применения изделий или систем. В общем случае под ошибкой подразумевается дефект, погрешность или неумышленное искажение объекта или процесса. При этом предполагается, что известно правильное, эталонное состояние объекта, по отношению к которому может быть определено наличие отклонения – ***дефекта или ошибки***. Для систематической, координированной борьбы с ними необходимы исследования факторов, влияющих на надежность ПС со стороны случайных, существующих и потенциально возможных дефектов в конкретных программах. Это позволит целенаправленно разрабатывать комплексы методов и средств обеспечения надежности сложных ПС различного назначения при реально достижимом снижении уровня дефектов проектирования.

Последующий анализ надежности ПС базируется на модели взаимодействия основных компонентов, представленных на рис. 46. ***Объектами уязвимости,*** влияющими на надежность ПС, являются**:**

- динамический вычислительный процесс обработки данных, автоматизированной подготовки решений и выработки управляющих воздействий;
- информация, накопленная в базах данных, отражающая объекты внешней среды, и процессы ее обработки;
- объектный код программ, исполняемых вычислительными средствами в процессе функционирования ПС;
- информация, выдаваемая потребителям и на исполнительные механизмы, являющаяся результатом обработки исходных данных и информации, накопленной в базе данных**.**

На эти объекты воздействуют различные ***дестабилизирующие факторы,*** которые можно разделить на внутренние, присущие самим объектам уязвимости, и внешние, обусловленные средой, в которой эти объекты функционируют. ***Внутренними источниками угроз*** надежности функционирования сложных ПС можно считать следующие дефекты программ:

- системные ошибки при постановке целей и задач создания ПС. при формулировке требований к функциям и характеристикам решения задач, определении условий и параметров внешней среды, в которой предстоит применять ПС;
- алгоритмические ошибки разработки при непосредственной спецификации функций программных средств, при определении структуры и взаимодействия компонентов комплексов программ, а также при использовании информации баз данных;
- ошибки программирования в текстах программ и описаниях данных, а также в исходной и результирующей документации на компоненты и ПС в целом;
- недостаточную эффективность используемых методов и средств оперативной защиты программ и данных от сбоев и отказов и обеспечения надежности функционирования ПС в условиях случайных негативных воздействий.

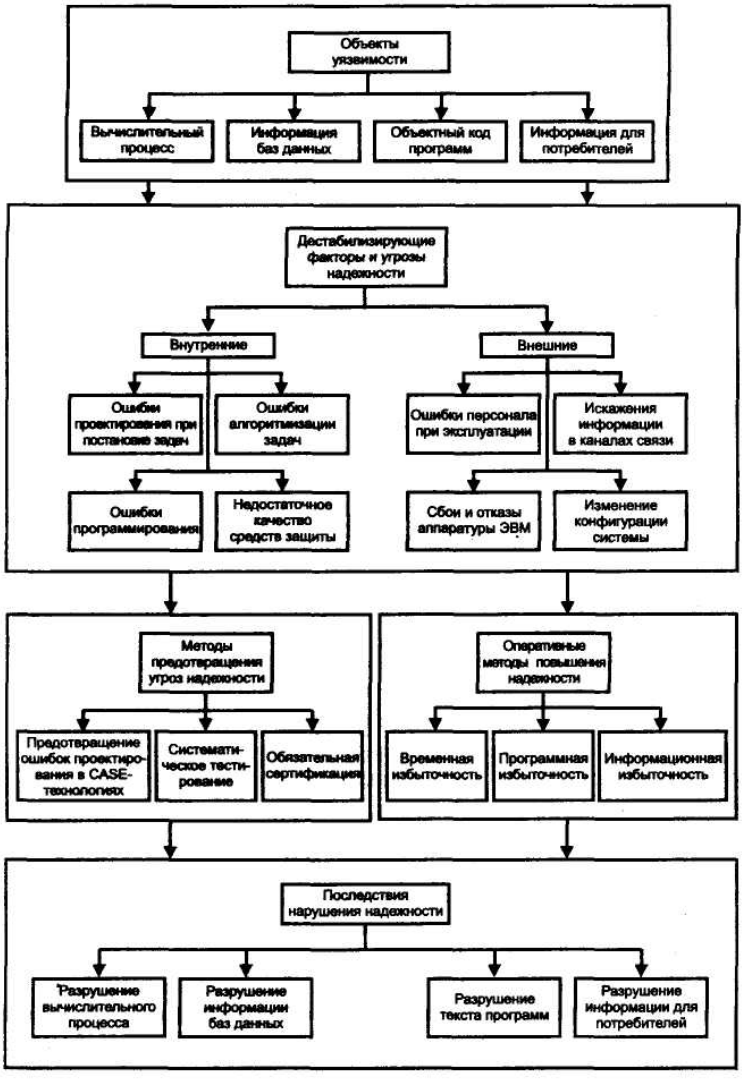


Рис. 46. Схема модели анализа надежности ПС

Внешними дестабилизирующими факторами, отражающимися на надежности функционирования перечисленных объектов уязвимости в ПС, являются:

- ошибки оперативного и обслуживающего персонала в процессе эксплуатации ПС;
- искажения в каналах телекоммуникации информации, поступающей от внешних источников и передаваемой потребителям, а также недопустимые для конкретной информационной системы характеристики потоков внешней информации;
- сбои и отказы в аппаратуре вычислительных средств;
- изменения состава и конфигурации комплекса взаимодействующей аппаратуры информационной системы за пределы, проверенные при испытаниях или сертификации и отраженные в эксплуатационной документации.

Полное устранение перечисленных негативных воздействий и дефектов, отражающихся на надежности функционирования сложных ПС, принципиально невозможно. Проблема состоит в выявлении факторов, от которых они зависят, создании методов и средств уменьшения их влияния на надежность ПС, а также в эффективном распределении ресурсов на защиту для обеспечения необходимой надежности комплекса программ, равноправного при их реальных воздействиях.

Различия между ожидаемыми и полученными результатами функционирования программ могут быть следствием ошибок не только в созданных программах и данных, но и системных ошибок в первичных требованиях спецификаций, явившихся исходной базой при создании ПС. Тем самым проявляется объективная реальность, заключающаяся в невозможности абсолютной корректности и полноты исходных данных для проектирования спецификаций сложных ПС. На практике в процессе разработки ПС исходные требования уточняются и детализируются по согласованию между заказчиком и разработчиком. Базой таких уточнений являются неформализованные представления и знания специалистов, а также результаты промежуточных этапов проектирования. Однако установить ***ошибочность исходных данных и спецификаций*** еще труднее, чем обнаружить ошибки в созданных программах и данных, так как принципиально отсутствуют формализованные данные, которые можно использовать как эталонные, и их заменяют неформализованные представления заказчиков и разработчиков.

Степень влияния всех внутренних дестабилизирующих факторов, а также некоторых внешних угроз на надежность ПС определяется в наибольшей степени ***качеством технологий проектирования, разработки, сопровождения и документирования ПС*** и их основных компонентов. При ограниченных ресурсах на разработку ПС для достижения заданных требований по надежности необходимо управление обеспечени-

ем качества в течение всего цикла создания программ и данных. Такое управление предполагает высокую дисциплину и проектировочную культуру всего коллектива специалистов, использование им методик, типовых нормативных документов и средств автоматизации разработки. Кроме того, обеспечение качества ПС предполагает формализацию и сертификацию технологии их разработки, а также выделение в специальный процесс поэтапного измерения и анализа текущего качества создаваемых и применяемых компонентов. Попытки создания сложных, распределенных ПС на базе мультипроцессорных ЭВМ и концепции клиент-сервер без использования эффективных технологий и средств автоматизации проектирования связаны с высоким риском полного провала проектов вследствие трудностей обеспечения необходимой надежности функционирования таких систем.

13.2. Методы обеспечения надежности программных средств

В современных автоматизированных технологиях создания и развития, сложных ПС с позиции обеспечения их необходимой и заданной надежности можно выделить методы и средства, позволяющие:

- ***создавать программные модули и функциональные компоненты*** высокого, гарантированного качества;
- ***предотвращать дефекты проектирования*** за счет эффективных технологий и средств автоматизации обеспечения всего жизненного цикла комплексов программ и баз данных;
- ***обнаруживать и устранять различные дефекты и ошибки*** проектирования, разработки и сопровождения программ путем систематического тестирования на всех этапах жизненного цикла ПС;
- ***удостоверять достигнутое качество и надежность функционирования*** ПС в процессе их испытаний и сертификации перед передачей в регулярную эксплуатацию;
- ***оперативно выявлять последствия дефектов программ и данных*** и восстанавливать нормальное, надежное функционирование комплексов программ.

Комплексное, скоординированное применение этих методов и средств в процессе создания, развития и применения ПС позволяет исключать некоторые виды угроз или значительно ослаблять их влияние. Тем самым уровень достигаемой надежности ПС становится предсказуемым и управляемым, непосредственно зависящим от ресурсов, выделяемых на его достижение, а главное от качества и эффективности технологии, используемой на всех этапах жизненного цикла ПС.

Все принципы и методы обеспечения надежности в соответствии с их целью можно разбить на четыре группы: ***предупреждение ошибок,***

обнаружение ошибок, исправление ошибок и ***обеспечение устойчивости к ошибкам.*** К первой группе относятся принципы и методы, позволяющие минимизировать или вообще исключить ошибки. Методы второй группы сосредоточивают внимание на функциях самого программного обеспечения, помогающих выявлять ошибки. К третьей группе относятся функции программного обеспечения, предназначенные для исправления ошибок или их последствий. Устойчивость к ошибкам (четвертая группа) – это мера способности системы программного обеспечения продолжать функционирование при наличии ошибок.

13.2.1. Предупреждение ошибок

К этой группе относятся принципы и методы, цель которых – не допустить появления ошибок в готовой программе. Большинство методов концентрируется на отдельных процессах перевода и направлено на предупреждение ошибок в этих процессах. Их можно разбить на следующие категории:

- методы, позволяющие справиться со сложностью, свести ее к минимуму, так как это – главная причина ошибок перевода;
- методы достижения большей точности при переводе;
- методы улучшения обмена информацией;
- методы немедленного обнаружения и устранения ошибок. Эти методы направлены на обнаружение ошибок на каждом шаге перевода, не откладывая до тестирования программы после ее написания.

Должно быть очевидно, что предупреждение ошибок – оптимальный путь к достижению надежности программного обеспечения.

Лучший способ обеспечить надежность – прежде всего не допустить возникновения ошибок. Гарантировать отсутствие ошибок, однако, невозможно никогда. Другие три группы методов опираются на предположение, что ошибки все-таки будут.

13.2.2. Обнаружение ошибок

Если предполагать, что в программном обеспечении какие-то ошибки все же будут, то лучшая (после предупреждения ошибок) стратегия – включить средства обнаружения ошибок в само программное обеспечение.

Большинство методов направлено по возможности на незамедлительное обнаружение сбоев. Немедленное обнаружение имеет два преимущества: можно минимизировать влияние ошибки и последующие затруднения для человека, которому придется извлекать информацию о ней, находить ее и исправлять.

Меры по обнаружению ошибок можно разбить на две подгруппы: ***пассивные*** попытки обнаружить симптомы ошибки в процессе «обычной» работы программного обеспечения и ***активные*** попытки про-

граммной системы периодически обследовать свое состояние в поисках признаков ошибок.

Пассивное обнаружение. Меры по обнаружению ошибок могут быть приняты на нескольких структурных уровнях программной системы. Здесь мы будем рассматривать уровень подсистем, или компонентов, т.е. нас будут интересовать меры по обнаружению симптомов ошибок, предпринимаемые при переходе от одного компонента к другому, а также внутри компонента. Все это, конечно, применимо также к отдельным модулям внутри компонента.

Разрабатывая эти меры, мы будем опираться на следующее.

Взаимное недоверие. Каждый из компонентов должен предполагать, что все другие содержат ошибки. Когда он получает какие-нибудь данные от другого компонента или из источника вне системы, он должен предполагать, что данные могут быть неправильными, и пытаться найти в них ошибки.

Немедленное обнаружение. Ошибки необходимо обнаружить как можно раньше. Это не только ограничивает наносимый ими ущерб, но и значительно упрощает задачу отладки.

Избыточность. Все средства обнаружения ошибок основаны на некоторой форме избыточности (явной или неявной).

Когда разрабатываются меры по обнаружению ошибок, важно принять согласованную стратегию для всей системы. Действия, предпринимаемые после обнаружения ошибки в программном обеспечении, должны быть единообразными для всех компонентов системы. Это ставит вопрос о том, какие именно действия следует предпринять, когда ошибка обнаружена. Наилучшее решение – немедленно завершить выполнение программы или (в случае операционной системы) перевести центральный процессор в состояние ожидания. Когда подобная стратегия бывает нецелесообразной (например, может оказаться, что приостанавливать работу системы нельзя). В таком случае используется метод ***регистрации ошибок.*** Описание симптомов ошибки и «моментальный снимок» состояния системы сохраняются во внешнем файле, после чего система может продолжать работу. Этот файл позднее будет изучен обслуживающим персоналом. Но всегда приостановить выполнение программы, чем регистрировать ошибки.

Активное обнаружение ошибок. Не все ошибки можно выявить пассивными методами, поскольку эти методы обнаруживают ошибку лишь тогда, когда ее симптомы подвергаются соответствующей проверке. Можно делать и дополнительные проверки, если спроектировать специальные программные средства для активного поиска признаков ошибок в системе. Такие средства называют ***средствами активного обнаружения ошибок.***

Активные средства обнаружения ошибок обычно объединяются в ***диагностический монитор:*** параллельный процесс, который периодически анализирует состояние системы с целью обнаружить ошибку. Большие программные системы, управляющие ресурсами, часто содержат ошибки, приводящие к потере ресурсов на длительное время. Например, управление памятью операционной системы сдает блоки памяти «в аренду» программам пользователей и другим частям операционной системы. Ошибка в этих самых «других частях» системы может иногда вести к неправильной работе блока управления памятью, занимающегося возвратом сданной ранее в аренду памяти, что вызывает медленное вырождение системы.

13.2.3. Исправление ошибок

Следующий шаг – методы исправления ошибок; после того как ошибка обнаружена, либо она сама, либо ее последствия должны быть исправлены программным обеспечением. Исправление ошибок самой системой – плодотворный метод проектирования надежных систем аппаратного обеспечения. Некоторые устройства способны обнаружить неисправные компоненты и перейти к использованию идентичных запасных. Аналогичные методы неприменимы к программному обеспечению вследствие глубоких внутренних различий между сбоями аппаратуры и ошибками в программах. Если некоторый программный модуль содержит ошибку, идентичные «запасные» модули также будут содержать ту же ошибку.

Другой подход к исправлению связан с попытками восстановить разрушения, вызванные ошибками, например искажения записей в базе данных или управляющих таблицах системы. Польза от методов борьбы с искажениями ограничена, поскольку предполагается, что разработчик заранее предугадает несколько возможных типов искажений и предусмотрит программно реализуемые функции для их устранения. Это похоже на парадокс, поскольку, если знать заранее, какие ошибки возникнут, можно было бы принять дополнительные меры по их предупреждению. Если методы ликвидации последствий сбоев не могут быть обобщены для работы со многими типами искажений, лучше всего направлять силы и средства на предупреждение ошибок. Вместо того, чтобы, разрабатывая операционную систему, оснащать ее средствами обнаружения и восстановления цепочки искаженных таблиц или управляющих блоков, следовало бы лучше спроектировать систему так, чтобы только один модуль имел доступ к этой цепочке, а затем настойчиво пытаться убедиться в правильности этого модуля.

13.2.4. Устойчивость к ошибкам

Методы этой группы ставят своей целью обеспечить функционирование программной системы при наличии в ней ошибок. Они разбива-

ются на три подгруппы: динамическая избыточность, методы отступления и методы изоляции ошибок.

1) Истоки концепции ***динамической избыточности*** лежат в проектировании аппаратного обеспечения. Один из подходов к динамической избыточности – ***метод голосования***. Данные обрабатываются независимо несколькими идентичными устройствами, и результаты сравниваются. Если большинство устройств выработало одинаковый результат, этот результат и считается правильным. И опять, вследствие особой природы ошибок в программном обеспечении ошибка, имеющаяся в копии программного модуля, будет также присутствовать во всех других его копиях, поэтому идея голосования здесь, видимо, неприемлема. Предлагаемый иногда подход к решению этой проблемы состоит в том, чтобы иметь несколько неидентичных копий модуля. Это значит, что все копии выполняют одну и ту же функцию, но либо реализуют различные алгоритмы, либо созданы разными разработчиками. Этот подход бесперспективен по следующим причинам. Часто трудно получить существенно разные версии модуля, выполняющие одинаковые функции. Кроме того, возникает необходимость в дополнительном программном обеспечении для организации выполнения этих версий параллельно или последовательно и сравнения результатов. Это дополнительное программное обеспечение повышает уровень сложности системы, что, конечно, противоречит основной идее предупреждения ошибок – стремиться в первую очередь минимизировать сложность.

Второй подход к динамической избыточности – выполнять эти запасные копии только тогда, когда результаты, полученные с помощью основной копии, признаны неправильными. Если это происходит, система автоматически вызывает запасную копию. Если и ее результаты неправильны, вызывается другая запасная копия и т.д.

2) Вторая подгруппа методов обеспечения устойчивости к ошибкам называется ***методами отступления*** или сокращенного обслуживания. Эти методы приемлемы обычно лишь тогда, когда для системы программного обеспечения существенно важно корректно закончить работу. Например, если ошибка оказывается в системе, управляющей технологическими процессами, и в результате эта система выходит из строя, то может быть загружен и выполнен особый фрагмент программы, призванный подстраховать систему и обеспечить безаварийное завершение всех управляемых системой процессов.

3) Последняя подгруппа – ***методы изоляции ошибок***. Основная их идея – не дать последствиям ошибки выйти за пределы как можно меньшей части системы программного обеспечения, так чтобы, если ошибка возникнет, то не вся система оказалась неработоспособной; отключаются лишь отдельные функции в системе либо некоторые ее пользователи. Например, во многих операционных системах изолиру-

ются ошибки отдельных пользователей, так что сбой влияет лишь на некоторое подмножество пользователей, а система в целом продолжает функционировать. В телефонных переключательных системах для восстановления после ошибки, чтобы не рисковать выходом из строя всей системы, просто разрывают телефонную связь. Другие методы изоляции ошибок связаны с защитой каждой из программ в системе от ошибок других программ. Ошибка в прикладной программе, выполняемой под управлением операционной системы, должна оказывать влияние только на эту программу. Она не должна сказываться на операционной системе или других программах, функционирующих в этой системе.

В большой вычислительной системе изоляция программ является ключевым фактором, гарантирующим, что отказы в программе одного пользователя не приведут к отказам в программах других пользователей или к полному выводу системы из строя. Основные правила изоляции ошибок перечислены ниже.

1) Прикладная программа не должна иметь возможности непосредственно ссылаться на другую прикладную программу или данные в другой программе и изменять их.

2) Прикладная программа не должна иметь возможности непосредственно ссылаться на программы или данные операционной системы и изменять их. Связь между двумя программами (или программой и операционной системой) может быть разрешена только при условии использования четко определенных сопряжений и только в случае, когда обе программы дают согласие на эту связь.

3) Прикладные программы и их данные должны быть защищены от операционной системы до такой степени, чтобы ошибки в операционной системе не могли привести к случайному изменению прикладных программ или их данных.

4)Операционная система должна защищать все прикладные программы и данные от случайного их изменения операторами системы или обслуживающим персоналом.

5) Прикладные программы не должны иметь возможности ни остановить систему, ни вынудить ее изменить другую прикладную программу или ее данные.

6) Когда прикладная программа обращается к операционной системе, должна проверяться допустимость всех параметров. Прикладная программа не должна иметь возможности изменить эти параметры между моментами проверки и реального их использования операционной системой.

7) Никакие системные данные, непосредственно доступные прикладным программам, не должны влиять на функционирование операционной системы. Ошибка в прикладной программе, вследствие кото-

рой содержимое этой памяти, может быть, случай, но изменено, приводит, в конце концов, к сбою системы.

8) Прикладные программы не должны иметь возможности в обход операционной системы прямо использовать управляемые ею аппаратные ресурсы. Прикладные программы не должны прямо вызывать компоненты операционной системы, предназначенные для использования только ее подсистемами.

9) Компоненты операционной системы должны быть изолированы друг от друга так, чтобы ошибка в одной из них не привела к изменению других компонентов или их данных.

10) Если операционная система обнаруживает ошибку в себе самой, она должна попытаться ограничить влияние этой ошибки одной прикладной программой и, в крайнем случае, прекратить выполнение только этой программы.

11) Операционная система должна давать прикладным программам возможность по требованию исправлять обнаруженные в них ошибки, а не безоговорочно прекращать их выполнение.

Реализация многих из этих принципов влияет на архитектуру лежащего в основе системы аппаратного обеспечения.

13.3. Модели надежности программного обеспечения

Термин ***модель надежности программного обеспечения***, как правило, относится к математической модели, построенной для оценки зависимости надежности программного обеспечения от некоторых определенных параметров. Значения таких параметров либо предполагаются известными, либо могут быть измерены в ходе наблюдений или экспериментального исследования процесса функционирования программного обеспечения.

Рассмотрим классификацию моделей надежности ПС, приведенную на рис. 47. Модели надежности программных средств (МНПС) подразделяются на аналитические и эмпирические. Аналитические модели дают возможность рассчитать количественные показатели надежности, основываясь на данных о поведении программы в процессе тестирования (измеряющие и оценивающие модели). Эмпирические модели базируются на анализе структурных особенностей программ. Они рассматривают зависимость показателей надежности от числа межмодульных связей, количества циклов в модулях, отношения количества прямолинейных участков программы к количеству точек ветвления и т.д. Часто эмпирические модели не дают конечных результатов показателей надежности, однако они включены в классификационную схему, так как развитие этих моделей позволяет выявлять взаимосвязь между сложностью ПС и его надежностью. Эти модели можно использовать на этапе проектирования ПС, когда осуществлена разбивка на модули и известна его структура.

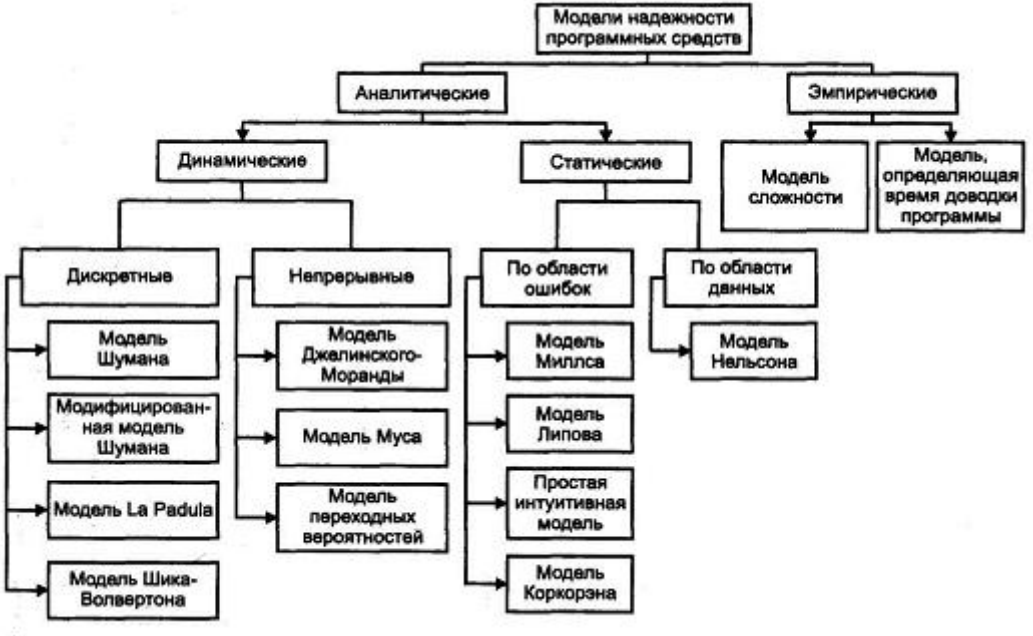


Рис. 47. Классификация моделей надежности

Аналитические модели представлены двумя группами: ***динамические модели*** и ***статические.*** В динамических МНПС поведение ПС (появление отказов) рассматривается во времени. В статических моделях появление отказов не связывают со временем, а учитывают только зависимость количества ошибок от числа тестовых прогонов (по области ошибок) или зависимость количества ошибок от характеристики входных данных (по области данных).

Для использования динамических моделей необходимо иметь данные о появлении отказов во времени. Если фиксируются интервалы каждого отказа, то получается непрерывная картина появления отказов во времени (группа динамических моделей с непрерывным временем). Может фиксироваться только число отказов за произвольный интервал времени. В этом случае поведение ПС может быть представлено только в дискретных точках (группа динамических моделей с дискретным временем). Рассмотрим основные предпосылки, ограничения и математический аппарат моделей, представляющих каждую группу, выделенную по схеме.

13.3.1. Аналитические модели надежности

Аналитическое моделирование надежности ПС включает четыре шага:

- определение предположений, связанных с процедурой тестирования ПС;
- разработка или выбор аналитической модели, базирующейся на предположениях о процедуре тестирования;
- выбор параметров моделей с использованием полученных данных;
- применение модели – расчет количественных показателей надежности по модели.

Динамические модели надежности рассмотрим на примере **модели Шумана.** Исходные данные для модели Шумана, которая относится к динамическим моделям дискретного времени, собираются в процессе тестирования ПС в течение фиксированных или случайных временных интервалов. Каждый интервал – это стадия, на которой выполняется последовательность тестов и фиксируется некоторое число ошибок.

Модель Шумана может быть использована при определенным образом организованной процедуре тестирования. Использование модели Шумана предполагает, что тестирование проводится в несколько этапов. Каждый этап представляет собой выполнение программы на полном комплексе разработанных тестовых данных. Выявленные ошибки регистрируются (собирается статистика об ошибках), но не исправляются. По завершении этапа на основе собранных данных о поведении ПС на очередном этапе тестирования может быть использована модель Шумана для расчета количественных показателей надежности. После

этого исправляются ошибки, обнаруженные на предыдущем этапе, при необходимости корректируются тестовые наборы и проводится новый этап тестирования. При использовании модели Шумана предполагается, что исходное количество ошибок в программе постоянно и в процессе тестирования может уменьшаться по мере того, как ошибки выявляются и исправляются. Новые ошибки при корректировке не вносятся. Скорость обнаружения ошибок пропорциональна числу оставшихся ошибок. Общее число машинных инструкций в рамках одного этапа тестирования постоянно.

Предполагается, что до начала тестирования в ПС имеется ЕТ ошибок. В течение времени тестирования т обнаруживается εс ошибок в расчете на команду в машинном языке.

Таким образом, удельное число ошибок на одну машинную команду, оставшихся в системе после т времени тестирования, равно:

$$\varepsilon_{\text{г}}(\tau) = \frac{E_{\text{Т}}}{I_{\text{Т}}} \cdot \varepsilon_{\text{С}}(\tau), \qquad (1)$$

где $I_{\text{Т}}$ – общее число машинных команд, которое предполагается постоянным в рамках этапа тестирования.

Автор предполагает, что значение функции частоты отказов Z(t) пропорционально числу ошибок, оставшихся в ПС после израсходованного на тестирование времени τ:

$$Z(\tau) = C\varepsilon_{\text{г}}(\tau), \qquad (2)$$

где C – некоторая константа;

t – время работы ПС без отказа.

Тогда, если время работы ПС без отказа t отсчитывается от точки t =0, а т остается фиксированным; функция надежности, или вероятность безотказной работы на интервале времени от 0 до *t,* равна:

$$R(t,\tau) = \exp\{-t[E_{\text{Т}}/I_{\text{Т}} - \varepsilon_{\text{с}}(\tau)]\}; \qquad (3)$$

$$t_{\text{ñð}} = \frac{1}{C[E_{\text{Т}}/I_{\text{Т}} - \varepsilon_{\text{с}}(\tau)]}. \qquad (4)$$

Из величин, входящих в формулы (2) и (3), не известны начальное значение ошибок в ПС (ЕТ) и коэффициент пропорциональности С. Для их определения прибегают к следующим рассуждениям. В процессе тестирования собирается информация о времени и количестве ошибок на каждом прогоне, т.е. общее время тестирования τ складывается из времени каждого прогона:

$$\tau = \tau_1 + \tau_2 + \tau_3 + \ldots + \tau_{\text{n}}. \qquad (5)$$

Предполагая, что интенсивность появления ошибок постоянна и равна λ, можно вычислить ее как число ошибок в единицу времени:

$$\lambda = \frac{\sum_{i=1}^{k} A_i}{\tau}, \qquad (6)$$

где Ai — количество ошибок на i-м прогоне;

$$t_{ñð} = \frac{\tau}{\sum_{i=1}^{k} A_i}. \qquad (7)$$

Имея данные для двух различных моментов тестирования τ_A и τ_b, которые выбираются произвольно с учетом требования, чтобы $\varepsilon_c(\tau_b) > \varepsilon_c(\tau_A)$, можно сопоставить уравнения (4) и (7) при τ_A и τ_b.

$$\frac{1}{\lambda_{\tau À}} = \frac{1}{Ñ\left(A_Ò / I_T - \varepsilon_C\left(\tau_A\right)\right)}; \qquad (8)$$

$$\frac{1}{\lambda_{\tau À}} = \frac{1}{Ñ\left(A_Ò / I_T - \varepsilon_C\left(\tau_B\right)\right)}. \qquad (9)$$

Вычисляя отношения (8) и (9), получим:

$$Å_Ò = \frac{Å_Ò\left[\tau_â / \lambda\tau_À \varepsilon_Ñ\left(\tau_À\right) - \varepsilon_Ñ\left(\tau_Â\right)\right.}{\left(\tau_Â / \lambda\tau_À\right) - 1}. \qquad (10)$$

Подставив полученную оценку параметров E_T в выражение (8), получим оценку для второго неизвестного параметра:

$$Ñ = \frac{\lambda\tau_À}{\left(A_Ò / I_T - \varepsilon_C\left(\tau_A\right)\right)}. \qquad (11)$$

Получив неизвестные ЕТ и С, можно рассчитать надежность программы по формуле (2).

13.3.2. Статические модели надежности

Статические модели принципиально отличаются от динамических, прежде всего тем, что в них не учитывается время появления ошибок в процессе тестирования и не используется никаких предположений о поведении функции риска λ(t). Эти модели строятся на твердом статистическом фундаменте.

В качестве примера статических моделей рассмотрим **модель Миллса**. Использование этой модели предполагает необходимость перед началом тестирования искусственно вносить в программу («засорять») некоторое количество известных ошибок. Ошибки вносятся слу-

чайным образом и фиксируются в протоколе искусственных ошибок. Специалист, проводящий тестирование, не знает ни количества, ни характера внесенных ошибок до момента оценки показателей надежности по модели Миллса. Предполагается, что все ошибки (как естественные, так и искусственно внесенные) имеют равную вероятность быть найденными в процессе тестирования.

Тестируя программу в течение некоторого времени, собирают статистику об ошибках. В момент оценки надежности по протоколу искусственных ошибок все ошибки делятся на собственные и искусственные. Соотношение

$$N = \frac{S \cdot n}{V} \qquad (12)$$

дает возможность оценить N – первоначальное число ошибок в программе. В данном соотношении, которое называется формулой Миллса, S – количество искусственно внесенных ошибок, n – число найденных собственных ошибок, V – число обнаруженных к моменту оценки искусственных ошибок.

Вторая часть модели связана с проверкой гипотезы от N. Предположим, что в программе имеется K собственных ошибок, и внесем в нее еще S ошибок. В процессе тестирования были обнаружены все S внесенных ошибок и n собственных ошибок.

Тогда по формуле Миллса мы предполагаем, что первоначально в программе было N = n ошибок. Вероятность, с которой можно высказать такое предположение, возможно, рассчитать по следующему соотношению:

$$\tilde{N} = \begin{cases} 1, \text{åñëè } n > K; \\ \dfrac{S}{S+K+1}, \text{åñëè } n <= K \end{cases}. \qquad (13)$$

Таким образом, величина *C* является мерой доверия к модели и показывает вероятность того, насколько правильно найдено значение *N*. Эти два связанных между собой по смыслу соотношения образуют полезную модель ошибок: первое предсказывает возможное число первоначально имевшихся в программе ошибок, а второе используется для установления доверительного уровня прогноза.

13.3.3. Эмпирические модели надежности

Эмпирические модели в основном базируются на анализе структурных особенностей программного средства (или программы). Как указывалось ранее, эмпирические модели часто не дают конечных результатов показателей надежности, однако их использование на этапе проектирования ПС полезно для прогнозирования требующихся ресур-

сов тестирования, уточнения плановых сроков завершения проекта и т.д.

Модель сложности. В литературе неоднократно подчеркивается тесная взаимосвязь между сложностью и надежностью ПС. Если придерживаться упрощенного понимания сложности ПС, то она может быть описана такими характеристиками, как размер ПС (количество программных модулей), количество и сложность межмодульных интерфейсов.

Под программным модулем в данном случае следует понимать программную единицу, выполняющую определенную функцию (ввод, вывод, вычисление и т.д.) и взаимосвязанную с другими модулями ПС. Сложность модуля ПС может быть описана, если рассматривать структуру программы.

В качестве структурных характеристик модуля ПС используются:

- отношение действительного числа дуг к максимально возможному числу дуг, получаемому искусственным соединением каждого узла с любым другим узлом дугой;
- отношение числа узлов к числу дуг;
- отношение числа петель к общему числу дуг.

Для сложных модулей и для больших многомодульных программ составляется имитационная модель, программа которой «засоряется» ошибками и тестируется по случайным входам. Оценка надежности осуществляется по модели Миллса.

При проведении тестирования известна структура программы, имитирующей действия основной, но не известен конкретный путь, который будет выполняться при вводе определенного тестового входа. Кроме того, выбор очередного тестового набора из множества тест-входов случаен, т.е. в процессе тестирования не обосновывается выбор очередного тестового входа. Эти условия вполне соответствуют реальным условиям тестирования больших программ.

Полученные данные анализируются, проводится расчет показателей надежности по модели Миллса (или любой другой из описанных выше), и считается, что реальное ПС, выполняющее аналогичные функции, с подобными характеристиками и в реальных условиях должно вести себя аналогичным или похожим образом.

Преимущества оценки показателей надежности по имитационной модели, создаваемой на основе анализа структуры будущего реального ПС, заключаются в следующем:

- модель позволяет на этапе проектирования ПС принимать оптимальные проектные решения, опираясь на характеристики ошибок, оцениваемые с помощью имитационной модели;
- модель позволяет прогнозировать требуемые ресурсы тестирования;

- модель дает возможность определить меру сложности программ и предсказать возможное число ошибок и т.д.

К недостаткам можно отнести высокую стоимость метода, так как он требует дополнительных затрат на составление имитационной модели, и приблизительный характер получаемых показателей.

Контрольные вопросы

1. Дайте определение «надежность» согласно ГОСТ 13377-75.
2. Какими факторами характеризуется надежность программного средства?
3. Назовите основные факторы, влияющие на надежность программного средства.
4. Опишите основные методы обеспечения надежности программного средства.
5. Какова классификация моделей надежности?
6. От чего зависит выбор модели надежности для расчета показателей надежности?
7. В чем заключается различие между аналитическими и эмпирическими моделями надежности ПС?
8. Объясните основные различия между статическими и динамическими аналитическими моделями.

14. ДОКУМЕНТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

14.1. Общая характеристика состояния стандартов документирования программного обеспечения

Основу отечественной нормативной базы в области документирования ПС составляет комплекс стандартов Единой системы программной документации (ЕСПД). Основная и большая часть комплекса ЕСПД была разработана в 70-е и 80-е годы. Сейчас этот комплекс представляет собой систему межгосударственных стандартов стран СНГ (ГОСТ), действующих на территории Российской Федерации на основе межгосударственного соглашения по стандартизации.

Стандарты ЕСПД в основном охватывают ту часть документации, которая создается в процессе разработки ПС, и связаны, по большей части, с документированием функциональных характеристик ПС. Следует отметить, что стандарты ЕСПД (ГОСТ 19) носят рекомендательный характер. Впрочем, это относится и ко всем другим стандартам в области ПС (ГОСТ 34, Международному стандарту ISO/IEC, и др.). Дело в том, что в соответствии с Законом РФ «О стандартизации» эти стандарты становятся обязательными на контрактной основе – то есть при ссылке на них в договоре на разработку (поставку) ПС.

Говоря о состоянии ЕСПД в целом, можно констатировать, что большая часть стандартов ЕСПД морально устарела.

К числу основных недостатков **ЕСПД** можно отнести:

- ориентацию на единственную, «каскадную» модель жизненного цикла (ЖЦ) ПС;
- отсутствие четких рекомендаций по документированию характеристик качества ПС;
- отсутствие системной увязки с другими действующими отечественными системами стандартов по ЖЦ и документированию продукции в целом, например, ЕСКД;
- нечетко выраженный подход к документированию ПС как товарной продукции;
- отсутствие рекомендаций по самодокументированию ПС, например, в виде экранных меню и средств оперативной помощи пользователю («хелпов»);
- отсутствие рекомендаций по составу, содержанию и оформлению перспективных документов на ПС, согласованных с рекомендациями международных и региональных стандартов.

Стандарты комплекса **ГОСТ 34** на создание и развитие автоматизированных систем (АС) – обобщенные, но воспринимаемые как весьма жесткие по структуре ЖЦ и проектной документации. Но эти стандарты многими считаются бюрократическими до вредности и консервативны-

ми до устарелости. Насколько это так, а насколько ГОСТ 34 остается работающим с пользой – полезно разобраться.

ЕСПД нуждается в полном пересмотре на основе стандарта ИСО.МЭК 12207-95 на процессы ЖЦ ПС.

Тем не менее, до пересмотра всего комплекса, многие стандарты ЕСПД могут с пользой применяться в практике документирования ПС. Эта позиция основана на следующем:

- стандарты ЕСПД вносят элемент упорядочения в процесс документирования ПС;
- предусмотренный стандартами ЕСПД состав программных документов вовсе не такой «жесткий», как некоторым кажется: стандарты позволяют вносить в комплект документации на ПС дополнительные виды
- стандарты ЕСПД позволяют вдобавок мобильно изменять структуры и содержание установленных видов ПД исходя из требований заказчика и пользователя.

При этом стиль применения стандартов может соответствовать современному общему стилю адаптации стандартов к специфике проекта: заказчик и руководитель проекта выбирают уместное в проекте подмножество стандартов и ПД, дополняют выбранные ПД нужными разделами и исключают ненужные, привязывают создание этих документов к той схеме ЖЦ, которая используется в проекте.

14.2. Единая система программной документации

Стандарты ЕСПД (как и другие ГОСТы) подразделяют на группы, приведенные в табл. 3.

Таблица 3

Группы стандарта ЕСПД

Код группы	Наименование группы
0	Общие положения
1	Основополагающие стандарты
2	Правила выполнения документации разработки
3	Правила выполнения документации изготовления
4	Правила выполнения документации сопровождения
5	Правила выполнения эксплуатационной документации
6	Правила обращения программной документации
7	Резервные группы
8	
9	Прочие стандарты

Обозначение стандарта ЕСПД строят по классификационному признаку:

Обозначение стандарта ЕСПД должно состоять из:

- числа 19 (присвоенных классу стандартов ЕСПД);
- одной цифры (после точки), обозначающей код классификационной группы стандартов, указанной таблице;
- двузначного числа (после тире), указывающего год регистрации стандарта.

14.2.1. Перечень документов ЕСПД.

1. ГОСТ 19.001-77 ЕСПД. Общие положения.
2. ГОСТ 19.101-77 ЕСПД. Виды программ и программных документов.
3. ГОСТ 19.102-77 ЕСПД. Стадии разработки.
4. ГОСТ 19.103-77 ЕСПД. Обозначение программ и программных документов.
5. ГОСТ 19.104-78 ЕСПД. Основные надписи.
6. ГОСТ 19.105-78 ЕСПД. Общие требования к программным документам.
7. ГОСТ 19.106-78 ЕСПД. Требования к программным документам, выполненным печатным способом.
8. ГОСТ 19.201-78 ЕСПД. Техническое задание. Требования к содержанию и оформлению.
9. ГОСТ 19.202-78 ЕСПД. Спецификация. Требования к содержанию и оформлению.
10. ГОСТ 19.301-79 ЕСПД. Порядок и методика испытаний.
11. ГОСТ 19.401-78 ЕСПД. Текст программы. Требования к содержанию и оформлению.
12. ГОСТ 19.402-78 ЕСПД. Описание программы.
13. ГОСТ 19.404-79 ЕСПД. Пояснительная записка. Требования к содержанию и оформлению.
14. ГОСТ 19.501-78 ЕСПД. Формуляр. Требования к содержанию и оформлению.
15. ГОСТ 19.502-78 ЕСПД. Описание применения. Требования к содержанию и оформлению.
16. ГОСТ 19.503-79 ЕСПД. Руководство системного программиста. Требования к содержанию и оформлению.
17. ГОСТ 19.504-79 ЕСПД. Руководство программиста.
18. ГОСТ 19.505-79 ЕСПД. Руководство оператора.
19. ГОСТ 19.506-79 ЕСПД. Описание языка.

20. ГОСТ 19.508-79 ЕСПД. Руководство по техническому обслуживанию. Требования к содержанию и оформлению.

21. ГОСТ 19.604-78 ЕСПД. Правила внесения изменений в программные документы, выполняемые печатным способом.

22. ГОСТ 19.701-90 ЕСПД. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения.

23. ГОСТ 19.781-90. Обеспечение систем обработки информации программное.

14.2.2. Термины и определения

Из всех стандартов ЕСПД остановимся только на тех, которые могут чаще использоваться на практике.

Первым укажем стандарт, который можно использовать при формировании заданий на программирование.

ГОСТ (СТ СЭВ) 19.201-78 (1626-79). ЕСПД. Техническое задание. Требование к содержанию и оформлению. (Переиздан в ноябре 1987г с изм.1).

Техническое задание (ТЗ) содержит совокупность требований к ПС и может использоваться как критерий проверки и приемки разработанной программы. Поэтому достаточно полно составленное (с учетом возможности внесения дополнительных разделов) и принятое заказчиком и разработчиком, ТЗ является одним из основополагающих документов проекта ПС.

Техническое задание должно содержать следующие разделы:

- введение;
- основания для разработки;
- назначение разработки;
- требования к программе или программному изделию;
- требования к программной документации;
- технико-экономические показатели;
- стадии и этапы разработки;
- порядок контроля и приемки;
- в техническое задание допускается включать приложения.

В зависимости от особенностей программы или программного изделия допускается уточнять содержание разделов, вводить новые разделы или объединять отдельные из них.

Следующий стандарт *ГОСТ (СТ СЭВ) 19.101-77 (1626-79). ЕСПД. Виды программ и программных документов (Переиздан в ноябре 1987г с изм.1*). устанавливает виды программ и программных документов для вычислительных машин, комплексов и систем независимо от их назначения и области применения.

Таблица 4

Виды программ

Вид программы	Определение
Компонент	Программа, рассматриваемая как единое целое, выполняющая законченную функцию и применяемая самостоятельно или в составе комплекса
Комплекс	Программа, состоящая из двух или более компонентов и (или) комплексов, выполняющих взаимосвязанные функции, и применяемая самостоятельно или в составе другого комплекса

Таблица 5

Виды программных документов

Вид программного документа	Содержание программного документа
Спецификация	Состав программы и документации на нее
Ведомость держателей подлинников	Перечень предприятий, на которых хранят подлинники программных документов
Текст программы	Запись программы с необходимыми комментариями
Описание программы	Сведения о логической структуре и функционировании программы
Программа и методика испытаний	Требования, подлежащие проверке при испытании программы, а также порядок и методы их контроля
Техническое задание	Назначение и область применения программы, технические, технико-экономические и специальные требования, предъявляемые к программе, необходимые стадии и сроки разработки, виды испытаний
Пояснительная записка	Схема алгоритма, общее описание алгоритма и (или) функционирования программы, а также обоснование принятых технических и технико-экономических решений
Эксплуатационные документы	Сведения для обеспечения функционирования и эксплуатации программы

Таблица 6

Виды эксплуатационных документов

Вид эксплуатационного документа	Содержание эксплуатационного документа
Ведомость эксплуатационных документов	Перечень эксплуатационных документов на программу
Формуляр	Основные характеристики программы, комплектность и сведения об эксплуатации программы
Описание применения	Сведения о назначении программы, области применения, применяемых методах, классе решаемых задач, ограничениях для применения, минимальной конфигурации технических средств
Руководство системного программиста	Сведения для проверки, обеспечения функционирования и настройки программы на условия конкретного применения
Руководство программиста	Сведения для эксплуатации программы
Руководство оператора	Сведения для обеспечения процедуры общения оператора с вычислительной системой в процессе выполнения программы
Описание языка	Описание синтаксиса и семантики языка
Руководство по техническому обслуживанию	Сведения для применения тестовых и диагностических программ при обслуживании технических средств

В зависимости от способа выполнения и характера применения программные документы подразделяются на подлинник, дубликат и копию (ГОСТ 2.102-68), предназначенные для разработки, сопровождения и эксплуатации программы.

ГОСТ 19.102-77. ЕСПД. Стадии разработки устанавливает стадии разработки программ и программной документации для вычислительных машин, комплексов и систем независимо от их назначения и области применения

Таблица 7

Стадии разработки, этапы и содержание работ

Стадии разработки	Этапы работ	Содержание работ
1	2	3
Техническое задание	Обоснование необходимости разработки программы	Постановка задачи. Сбор исходных материалов. Выбор и обоснование критериев эффективности и качества разрабатываемой программы. Обоснование необходимости проведения научно-исследовательских работ.
	Научно-исследовательские работы	Определение структуры входных и выходных данных. Предварительный выбор методов решения задач. Обоснование целесообразности применения ранее разработанных программ. Определение требований к техническим средствам. Обоснование принципиальной возможности решения поставленной задачи.
	Разработка и утверждение технического задания	Определение требований к программе. Разработка технико-экономического обоснования разработки программы. Определение стадий, этапов и сроков разработки программы и документации на нее. Выбор языков программирования. Определение необходимости проведения научно-исследовательских работ на последующих стадиях. Согласование и утверждение технического задания.
Эскизный проект	Разработка эскизного проекта	Предварительная разработка структуры входных и выходных данных. Уточнение методов решения задачи. Разработка общего описания алгоритма решения задачи. Разработка технико-экономического обоснования.
	Утверждение эскизного проекта	Разработка пояснительной записки. Согласование и утверждение эскизного проекта.

Окончание табл. 7

1	2	3
Технический проект	Разработка технического проекта	Уточнение структуры входных и выходных данных. Разработка алгоритма решения задачи. Определение формы представления входных и выходных данных. Определение семантики и синтаксиса языка. Разработка структуры программы. Окончательное определение конфигурации технических средств.
	Утверждение технического проекта	Разработка плана мероприятий по разработке и внедрению программ. Разработка пояснительной записки. Согласование и утверждение технического проекта.
Рабочий проект	Разработка программы	Программирование и отладка программы
	Разработка программной документации	Разработка программных документов в соответствии с требованиями ГОСТ 19.101-77.
	Испытания программы	Разработка, согласование и утверждение программы и методики испытаний. Проведение предварительных государственных, межведомственных, приемо-сдаточных и других видов испытаний. Корректировка программы и программной документации по результатам испытаний.
Внедрение	Подготовка и передача программы	Подготовка и передача программы и программной документации для сопровождения и (или) изготовления Оформление и утверждение акта о передаче программы на сопровождение и (или) изготовление. Передача программы в фонд алгоритмов и программ.

Примечания.

1. Допускается исключать вторую стадию разработки, а в технически обоснованных случаях – вторую и третью стадии. Необходимость проведения этих стадий указывается в техническом задании.

2. Допускается объединять, исключать этапы работ и (или) их содержание, а также вводить другие этапы работ по согласованию с заказчиком.

ГОСТ 19.103-77 ЕСПД. Обозначение программ и программных документов

Программа и ее документ «Спецификация» имеют следующую структуру обозначения:

Структура обозначения других программных документов:

• Код страны-разработчика и код организации-разработчика присваивают в установленном порядке.

• Регистрационный номер присваивается в порядке возрастания, начиная с 00001 до 99999, для каждой организации-разработчика.

• Номер издания программы или номер редакции. номер документа данного вида, номер части документа присваиваются в порядке возрастания с 01 до 99. (Если документ состоит из одной части, то дефис и порядковый номер части не указывают).

• Номер редакции спецификации и ведомости эксплуатационных документов на программу должны совпадать с номером издания этой же программы.

ГОСТ 19.105-78 ЕСПД. Общие требования к программным документам. Настоящий стандарт устанавливает общие требования к оформлению программных документов для вычислительных машин, комплексов и систем, независимо от их назначения и области применения и предусмотренных стандартами Единой системы программной документации (ЕСПД) для любого способа выполнения документов на различных носителях данных.

Программный документ может быть представлен на различных типах носителей данных и состоит из следующих условных частей:

– титульной;

– информационной;

– основной.

Правила оформления документа и его частей на каждом носителе данных устанавливаются стандартами ЕСПД на правила оформления документов на соответствующих носителях данных.

ГОСТ 19.106-78 ЕСПД. Требования к программным документам, выполненным печатным способом.

Программные документы оформляют:

• на листах формата А4 (ГОСТ 2.301-68) при изготовлении документа машинописным или рукописным способом;

• допускается оформление на листах формата А3;

• при машинном способе выполнения документа допускаются отклонения размеров листов, соответствующих форматам А4 и А3, определяемые возможностями применяемых технических средств; на листах

форматов А4 и А3, предусматриваемых выходными характеристиками устройств вывода данных, при изготовлении документа машинным способом;

- на листах типографических форматов при изготовлении документа типографским способом.

Расположение материалов программного документа осуществляется в следующей последовательности:

- *титульная часть:*
- лист утверждения (не входит в общее количество листов документа);
- титульный лист (первый лист документа);
- информационная часть:

– аннотация;
– лист содержания;

- основная *часть:*

– текст документа (с рисунками, таблицами и т.п.)
– перечень терминов и их определений;
– перечень сокращений;
– приложения;
– предметный указатель;
– перечень ссылочных документов;

- часть регистрации изменений:

– лист регистрации изменений.

Перечень терминов и их определений, перечень сокращений, приложения, предметных указатель, перечень ссылочных документов выполняются при необходимости.

Следующий стандарт ориентирован на документирование результирующего продукта разработки – *ГОСТ 19.402-78 ЕСПД. Описание программы.*

Состав документа «Описание программы» в своей содержательной части может дополняться разделами и пунктами, почерпнутыми из стандартов для других описательных документов и руководств: *ГОСТ 19.404-79 ЕСПД. Пояснительная записка, ГОСТ 19.502-78 ЕСПД. Описание применения, ГОСТ 19.503-79 ЕСПД. Руководство системного программиста, ГОСТ 19.504-79 ЕСПД. Руководство программиста, ГОСТ 19.505-79 ЕСПД. Руководство оператора.*

Есть также группа стандартов, определяющая требования к фиксации всего набора программ и ПД, которые оформляются для передачи ПС. Они порождают лаконичные документы учетного характера и могут быть полезны для упорядочения всего хозяйства программ и ПД (ведь очень часто требуется просто навести элементарный порядок!). Есть и стандарты, определяющие правила ведения документов в «хозяйстве» ПС.

Надо также выделить

ГОСТ 19.301-79 ЕСПД. Программа и методика испытаний, который (в адаптированном виде) может использоваться для разработки документов планирования и проведения испытательных работ по оценке готовности и качества ПС.

Наконец, последний по году принятия стандарт.

ГОСТ 19.701-90 ЕСПД. Схемы алгоритмов, программ, данных и систем. Обозначения условные графические и правила выполнения.

Он устанавливает правила выполнения схем, используемых для отображения различных видов задач обработки данных и средств их решения и полностью соответствует стандарту ИСО 5807:1985.

Наряду с ЕСПД на межгосударственном уровне действуют еще два стандарта, также относящихся к документированию ПС и принятых не так давно, как большая часть ГОСТ ЕСПД.

ГОСТ 19781-90 Обеспечение систем обработки информации программное. Термины и определения. Разработан взамен ГОСТ 19.781-83 и ГОСТ 19.004-80 и устанавливает термины и определения понятий в области программного обеспечения (ПО) систем обработки данных (СОД), применяемые во всех видах документации и литературы, входящих в сферу работ по стандартизации или использующих результаты этих работ.

ГОСТ 28388-89 Системы обработки информации. Документы на магнитных носителях данных. Порядок выполнения и обращения. Распространяется не только на программные, но и на конструкторские, технологические и другие проектные документы, выполняемые на магнитных носителях.

14.3. Государственные стандарты РФ (ГОСТ Р)

В РФ действует ряд стандартов в части документирования ПС, разработанных на основе прямого применения международных стандартов ИСО. Это самые «свежие» по времени принятия стандарты. Некоторые из них впрямую адресованы руководителям проекта или директорам информационных служб. Вместе с тем они неоправданно мало известны в среде профессионалов. Вот их представление.

ГОСТ Р ИСО/МЭК 9294-93 Информационная технология. Руководство по управлению документированием программного обеспечения. Стандарт полностью соответствует международному стандарту ИСО/МЭК ТО 9294:1990 и устанавливает рекомендации по эффективному управлению документированием ПС для руководителей, отвечающих за их создание. Целью стандарта является оказание помощи в определении стратегии документирования ПС; выборе стандартов по

документированию; выборе процедур документирования; определении необходимых ресурсов; составлении планов документирования.

ГОСТ Р ИСО/МЭК 9126-93 Информационная технология. Оценка программной продукции. Характеристики качества и руководства по их применению. Стандарт полностью соответствует международному стандарту ИСО/МЭК 9126:1991. В его контексте под характеристикой качества понимается «набор свойств (атрибутов) программной продукции, по которым ее качество описывается и оценивается». Стандарт определяет шесть комплексных характеристик, которые с минимальным дублированием описывают качество ПС (ПО, программной продукции): функциональные возможности; надежность; практичность; эффективность; сопровождаемость; мобильность. Эти характеристики образуют основу для дальнейшего уточнения и описания качества ПС.

ГОСТ Р ИСО 9127-94 Системы обработки информации. Документация пользователя и информация на упаковке для потребительских программных пакетов. Стандарт полностью соответствует международному стандарту ИСО 9127:1989. В контексте настоящего стандарта под потребительским программным пакетом (ПП) понимается «программная продукция, спроектированная и продаваемая для выполнения определенных функций; программа и соответствующая ей документация, упакованные для продажи как единое целое». Под документацией пользователя понимается документация, которая обеспечивает конечного пользователя информацией по установке и эксплуатации ПП. Под информацией на упаковке понимают информацию, воспроизводимую на внешней упаковке ПП. Ее целью является предоставление потенциальным покупателям первичных сведений о ПП.

ГОСТ Р ИСО/МЭК 8631-94 Информационная технология. Программные конструктивы и условные обозначения для их представления. Описывает представление процедурных алгоритмов.

Контрольные вопросы

1. Как можно охарактеризовать понятие «программная документация»?
2. Что представляет собой внешняя и внутренняя программная документация?
3. Дайте определение понятию «единая система программной документации».
4. В чем заключаются основные недостатки единой системы программной документации?
5. Дайте определение понятию «техническое задание»
6. Объясните смысл понятия «документация пользователя».
7. Какими свойствами должна обладать документация пользователя?
8. Дайте краткую характеристику документации пользователя.

СПИСОК ЛИТЕРАТУРЫ

1. Автоматизированные информационные технологии в экономике / Под общ. ред. И.Т. Трубилина. – М.: Финансы и статистика, 2000.
2. Автоматизированные информационные технологии в экономике: Учебник / Под ред. Г.А. Титоренко. – М.: Компьютер: ЮНИТИ, 1998.
3. Благодатских В.А., Волнин В.А., Поскакалов К.Ф. Стандартизация разработки программных средств. – М.: Финансы и статистика, 2003.
4. Благодатских В.А. Экономика, разработка и использование программного обеспечения ЭВМ. – М.: Финансы и статистика, 1995.
5. Боггс У., Боггс М. UML и Rational Rose: Пер. с англ. – М.: ЛОРИ, 2002
6. Брауде Э.Дж. Технология разработки программного обеспечения: Пер. с англ. – СПб.: Питер, 2004.
7. Вигерс К. Разработка требований к программному обеспечению: Пер. с англ. – М.: Русская редакция, 2004
8. Вендров А.М. Проектирование программного обеспечения экономических информационных систем – М.: Финансы и статистика, 2002.
9. Вендрова А.М Практикуме по проектированию программного обеспечения экономических информационных систем – М.: Финансы и статистика, 2002.
10. Информатика. Базовый курс / Под ред. С.В. Симоновича. – СПб., 2000.
11. Калянов Г.Н. CASE структурный системный анализ (автоматизация и применение). – М.: «ЛОРИ», 1996.
12. Калянов Г.Н. Теория и практика реорганизации бизнесс-процессов. – М.: СИНТЕГ, 2000
13. Компьютерные технологии обработки информации / Под. ред. С.В. Назарова. – М.: Финансы и статистика, 1995.
14. Калашян А.Н., Калянов Г.Н. Структурные модели бизнеса: DFD-технологии. – М.: Финансы и статистика, 2003.
15. Керн А. Быстрая разработка программного обеспечения: Пер. с англ. – М.: ЛОРИ, 2002.
16. Коберн А. Быстрая разработка программного обеспечения: Пер. с англ. – М.: ЛОРИ, 2002.
17. Котов С.Л. Нормирование жизненного цикла программной продукции. – М.: ЮНИТИ-ДАНА, 2002.
18. Коутс Р., Влейминк И. Интерфейс «человек-компьютер»: Пер. с англ. – М.: Мир, 1990.

19. Леоненков А.В. Самоучитель UML – СПб.: БХВ-Петербург, 2001.
20. Липаев В.В. Надежность программных средств – М.: СИНТЕГ, 1998.
21. Липаев В.В. Документирование и управление конфигурацией программных средств – М.: СИНТЕГ, 1998
22. Липаев В.В. Обеспечение качества программных средств. Методы и стандарты. – М.: СИНТЕГ, 2001.
23. Липаев В.В. Системное проектирование сложных программных систем – М: СИНТЕГ, 1998
24. Майерс Г. Искусство тестирования программ: Пер. с англ. М. – М.: Финансы и статистика, 1982.
25. Маклаков С.В. Моделирование бизнес-процессов с BPwin 4.0. – М: Диалог МИФИ, 2002.
26. Орлов С.А. Технологии разработки программного обеспечения: Разработка сложных программных систем: Учебное пособие для студентов вузов, обуч. по напр. подготовки бакалавров и магистров «Информатика и выч.техника». – СПб.: Питер, 2002.
27. Пальчун Б.П., Юсупов Р.М. Оценка надежности программного обеспечения. – СПб.: Наука, 1994.
28. Ройс У. Управление проектами по созданию программного обеспечения: Пер. с англ. – М.: ЛОРИ, 2002.
29. Трофимов С.А. CASE – технологии: практическая работа в Rational Rose. Изд. 2-е. – М.: Бином-Пресс, 2002.
30. Фридман А.Л. Основы объектно-ориентированной разработки программных систем – М.: Финансы и статистика, 2000.
31. Черемных С.В., Семенов И.О., Ручкин В.С. Структурный анализ систем: IDEF-технологии – М.: Финансы и статистика, 2001.
32. Черемных С.В., Семенов И.О., Ручкин В.С. Моделирование и анализ систем: IDEF-технологии: практикум – М.: Финансы и статистика, 2002.
33. WWW. GOST.RU
34. WWW. STANDARD.RU

Список государственных стандартов

1. ГОСТ 19.001-77 ЕСПД. Общие положения.
2. ГОСТ 19.005-85 ЕСПД. Схемы алгоритмов и программ. Обозначения условные графические и правила выполнения.
3. ГОСТ 19.101-77 ЕСПД. Виды программ и программных документов.
4. ГОСТ 19.102-77 ЕСПД. Стадии разработки.

5. ГОСТ 19.103-77 ЕСПД. Обозначение программ и программных документов.

6. ГОСТ 19.104-78 ЕСПД. Основные надписи.

7. ГОСТ 19.105-78 ЕСПД. Общие требования к программным документам.

8. ГОСТ 19.106-78 ЕСПД. Требования к программным документам, выполненным печатным способом.

9. ГОСТ 19.201-78 ЕСПД. Техническое задание. Требования к содержанию и оформлению.

10. ГОСТ 19.202-78 ЕСПД. Спецификация. Требования к содержанию и оформлению.

11. ГОСТ 19.301-79 ЕСПД. Порядок и методика испытаний.

12. ГОСТ 19.401-78 ЕСПД. Текст программы. Требования к содержанию и оформлению.

13. ГОСТ 19.402-78 ЕСПД. Описание программы.

14. ГОСТ 19.403-79 ЕСПД. Ведомость держателей подлинников.

15. ГОСТ 19.404-79 ЕСПД. Пояснительная записка. Требования к содержанию и оформлению.

16. ГОСТ 19.501-78 ЕСПД. Формуляр. Требования к содержанию и оформлению.

17. ГОСТ 19.502-78 ЕСПД. Описание применения. Требования к содержанию и оформлению.

18. ГОСТ 19.503-79 ЕСПД. Руководство системного программиста. Требования к содержанию и оформлению.

19. ГОСТ 19.504-79 ЕСПД. Руководство программиста. Требования к содержанию и оформлению.

20. ГОСТ 19.505-79 ЕСПД. Руководство оператора. Требования к содержанию и оформлению.

21. ГОСТ 19.506-79 ЕСПД. Описание языка. Требования к содержанию и оформлению.

22. ГОСТ 19.507-79 ЕСПД. Ведомость эксплуатационных документов.

23. ГОСТ 19.508-79 ЕСПД. Руководство по техническому обслуживанию. Требования к содержанию и оформлению.

24. ГОСТ 19.601-78 ЕСПД. Общие правила дублирования, учета и хранения.

25. ГОСТ 19.602-78 ЕСПД. Правила дублирования, учета и хранения программных документов, выполненных печатным образом.

26. ГОСТ 19.603-78 ЕСПД. Общие правила внесения изменений.

27. ГОСТ 19.604-78 ЕСПД. Правила внесения изменений в программные документы, выполняемые печатным способом.

28. ГОСТ 19.701-90 ЕСПД. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения.

29. ГОСТ 19781 -90. Обеспечение систем обработки информации программное. Термины и определения.

30. ГОСТ 34.601-90. Информационная технология. Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Стадии создания.

31. ГОСТ 34.602-89. Информационная технология. Комплекс стандартов на автоматизированные системы. Техническое задание на создание автоматизированной системы.

32. ГОСТ 34.603-92. Информационная технология. Виды испытаний автоматизированных систем.

33. MIL-STD-498. Разработка и документирование программного обеспечения.

34. ISO 9126:1991. Информационная технология. Оценка программного продукта. Характеристики качества и руководство по их применению.

35. IEEE 1074-1995. Процессы жизненного цикла для развития программного обеспечения.

36. ANSI/IEEE 829-1983. Документация при тестировании программ.

37. ANSI/IEEE 1008-1986. Тестирование программных модулей и компонентов ПС.

38. ANSI/IEEE 983-1986. Руководство по планированию обеспечения качества программных средств.

39. ГОСТ Р ИСО/МЭК 9294-93. Информационная технология. Руководство по управлению документированием программного обеспечения.

40. ГОСТ Р ИСО/МЭК 9126-93. Информационная технология. Оценка программной продукции. Характеристики качества и руководство по их применению.

41. ГОСТ Р ИСО/МЭК 9127-94. Системы обработки информации. Документация пользователя и информация на упаковке для потребительских программных пакетов.

42. ГОСТ Р ИСО/МЭК 8631-94. Информационная технология. Программные конструктивы и условные обозначения для их представления.

43. ГОСТ Р ИСО/МЭК 12119:1994. Информационная технология. Пакеты программных средств. Требования к качеству и испытания.

44. ГОСТ Р ИСО/МЭК 12207. Процессы жизненного цикла программных средств.

ОГЛАВЛЕНИЕ