Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный университет
имени Франциска Скорины»

Н.Б. ОСИПЕНКО

СТАНДАРТИЗАЦИЯ И СЕРТИФИКАЦИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

ТЕКСТЫ ЛЕКЦИЙ
для студентов математических специальностей

Гомель 2012

ГГУ ИМЕНИ Ф. СКОРИНЫ

УДК 004.41.057.2
ББК 32.973.26–018.2ц.я73
О

Рецензенты:

кафедра математических проблем управления
учреждения образования
«Гомельский государственный университет
имени Франциска Скорины».

Рекомендовано к изданию научно–методическим советом учреждения образования «Гомельский государственный университет имени Франциска Скорины»

Осипенко, Н. Б.
О Стандартизация и сертификация программного обеспечения : тексты лекций для студентов математических специальностей / Н.Б. Осипенко; М–во образ. РБ, Гомельский государственный университет им. Ф. Скорины. – Гомель: ГГУ им. Ф. Скорины, 2012. – 155с.

Тексты лекций ставят своей целью оказание помощи студентам в усвоении знаний по требованиям к стандартам на разработку и использование, а также по оценке качества функционирования программного обеспечения.

Адресованы студентам математических специальностей.

ГГ

УДК 004.41.057.2
ББК 32.973.26–018.2ц.я73

© Осипенко Н. Б., 2012
© УО «ГГУ им. Ф. Скорины», 2012

Содержание

ВВЕДЕНИЕ 4
РАЗДЕЛ 1 ОБЩИЕ ПОЛОЖЕНИЯ О СТАНДАРТАХ.................................... 5
Тема 1 Основные понятия 5
Тема 2 Организации, разрабатывающие стандарты 11
РАЗДЕЛ 2 ЖИЗНЕННЫЙ ЦИКЛ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.........20
Тема 3 Систематизация процессов жизненного цикла....................................20
Тема 4 Основные модели жизненного цикла 27
РАЗДЕЛ 3 СТАНДАРТЫ ДОКУМЕНТИРОВАНИЯ ПРОГРАММНЫХ СРЕДСТВ 39
Тема 5 Общая характеристика проблем и задач документирования программного обеспечения..39
Тема 6 Единая система программной документации 42
РАЗДЕЛ 4 НАДЕЖНОСТЬ И КАЧЕСТВО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ 52
Тема 7 Основные понятия и показатели надежности программного обеспечения 52
Тема 8 Дестабилизирующие факторы и методы обеспечения надежности функционирования программных средств 65
Тема 9 Модели надежности программного обеспечения 78
Тема 10 Обеспечение качества и надежности в процессе разработки сложных программных средств 98
РАЗДЕЛ 5 ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.................112
Тема 11 Основные понятия 112
Тема 12 Тестирование надежности программного обеспечения 116
Тема 13 Тестирование программного обеспечения...131
Тема 14 Виды тестирования программного обеспечения 136
РАЗДЕЛ 6 CASE – ИНСТРУМЕНТАРИЙ АВТОМАТИЗАЦИИ АНАЛИЗА, ПРОЕКТИРОВАНИЯ И РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.....147
Тема 15 Классификация CASE – инструментария 147
Тема 16 Концептуальные основы CASE – технологий...................................151
ЛИТЕРАТУРА 159

ГГУ имени Ф. Скорины

ВВЕДЕНИЕ

В области инженерии программного обеспечения (ПО) актуальными являются проблемы оценки качества программного обеспечения. Одним из важных способов улучшения качества программного обеспечения в соответствии с требованиями пользователей программной продукции является ее стандартизация и аттестация работы программного обеспечения; контроль за внедрением и соблюдением стандартов. Поэтому специалист по программному обеспечению информационных технологий должен уметь оценивать программного обеспечения с точки зрения его стандартизации и практической пригодности в различных предметных областях. Создание конкурентоспособной программной продукции невозможно без использования соответствующих стандартов на всех этапах ее разработки.

Стандарты как нормативно–технические документы устанавливают комплекс норм, правил, требований к объекту стандартизации. Применение стандартов наряду с улучшением качества ПО способствует повышению развития информатизации процессов, росту эффективности внедрения и эксплуатации программных средств и устраняет разнобой при создании их различными разработчиками.

Тексты лекций ставят своей целью оказание помощи студентам в усвоении знаний по требованиям к стандартам на разработку и использование, а также по оценке качества функционирования программного обеспечения и средств вычислительной техники.

ГГУ ИМЕНИ ... РИНЫ

РАЗДЕЛ 1 ОБЩИЕ ПОЛОЖЕНИЯ О СТАНДАРТАХ

Тема 1 Основные понятия

1.1 Нормативные документы по стандартизации и виды стандартов.
1.2 Схема классификации стандартов в области информационных технологий.
1.3 Стандарты в области программного обеспечения.
1.4 Стандарты комплекса ГОСТ 34 на создание и развитие автоматизированных систем.
1.5 Сертификация

1.1 Нормативные документы по стандартизации и виды стандартов

Стандартизация связана с такими понятиями, как объект стандартизации и область стандартизации. Объектом стандартизации обычно называют продукцию, процесс, услугу, для которых разрабатывают те или иные требования, характеристики, параметры, правила и т.п. Стандартизация может касаться либо объекта в целом, либо его отдельных составляющих (характеристик). Областью стандартизации называют совокупность взаимосвязанных объектов стандартизации.

В процессе стандартизации вырабатываются нормы, правила, требования, характеристики, касающиеся объекта стандартизации, которые оформляются в виде нормативного документа. Разновидности нормативных документов, которые рекомендуются руководством Международной организации по стандартизации (ИСО) и Международной электротехнической комиссии (МЭК), а также принятые в государственной системе стандартизации Беларуси: стандарты, документы технических условий, своды правил, регламенты (технические регламенты).

Стандарт (от англ. standard – норма, образец) – в широком смысле слова образец, эталон, модель, принимаемые за исходные для сопоставления с ними других подобных объектов. Стандарт как нормативно–технический документ устанавливает комплекс норм, правил, требований к объекту стандартизации. Стандарт может быть разработан как на материальные предметы (продукцию, эталоны, образцы веществ), так и на нормы, правила, требования в различных областях.

Стандарт – это **нормативный документ, разработанный на основе консенсуса, утвержденный признанным органом, направленный на достижение оптимальной степени упорядочения** в определенной области. В стандарте устанавливаются для всеобщего и многократного использования общие принципы, правила, характеристики, касающиеся различных видов деятельности или их результатов. Стандарт должен быть основан на обобщенных результатах научных исследований,

технических достижений и практического опыта, тогда его использование принесет оптимальную выгоду.

Предварительный стандарт – это **временный** документ, который принимается органом по стандартизации и доводится до широкого круга потенциальных потребителей.

В практике термин «стандарт» может употребляться и по отношению к эталону, образцу или описанию продукта, процесса (услуги), хотя эталон правильнее относить к области метрологии, а термин «стандарт» использовать применительно к нормативному документу.

Документ технических условий (technical specification) устанавливает технические требования к продукции, услуге, процессу. Обычно в документе технических условий должны быть указаны методы или процедуры, которые следует использовать для проверки соблюдения требований данного нормативного документа в таких ситуациях, когда это необходимо.

Свод правил, как и предыдущий нормативный документ, может быть самостоятельным стандартом либо самостоятельным документом, а также частью стандарта. Он разрабатывается для процессов проектирования, монтажа оборудования и конструкций, технического обслуживания или эксплуатации объектов, конструкций, изделий.

Все указанные выше нормативные документы являются **рекомендательными**. В отличие от них регламент носит обязательный характер. *Регламент* – это документ, в котором содержатся обязательные правовые нормы.

1.2 Схема классификации стандартов в области информационных технологий

Схема классификаций стандартов в области информационных технологий приведена на рис.1.1. В зависимости от масштаба стандарты бывают международными, национальными, отраслевыми, внутрифирменными и принимаются соответствующими органами по стандартизации. В зависимости от возникновения стандарты бывают «де–факто» и «де–юре».

Стандарты на организацию жизненного цикла ПО подразделяются на стандарты обеспечения качества, надежности, тестирования, документирования, разработки ПО; а последние, в свою очередь, – на стандарты интерфейса, программирования, обмена данными и др.

В области информационных технологий зарекомендовали себя следующие стандарты моделей разработок: RUP, Tickit, CMM, IEEE Software Enginiring Standarts, IEEE/ EIA 12207, Cleanroom Software Enginiring, метод ORACLE (CDM, PJM, AIM, BPR, DWM) и др.

1.3 Стандарты в области программного обеспечения

В [5, стр.243] понятие стандартизация определяется как принятие соглашения по спецификации, производству и использованию аппаратных и программных средств вычислительной техники; установление и применение стандартов, норм, правил и т.п.

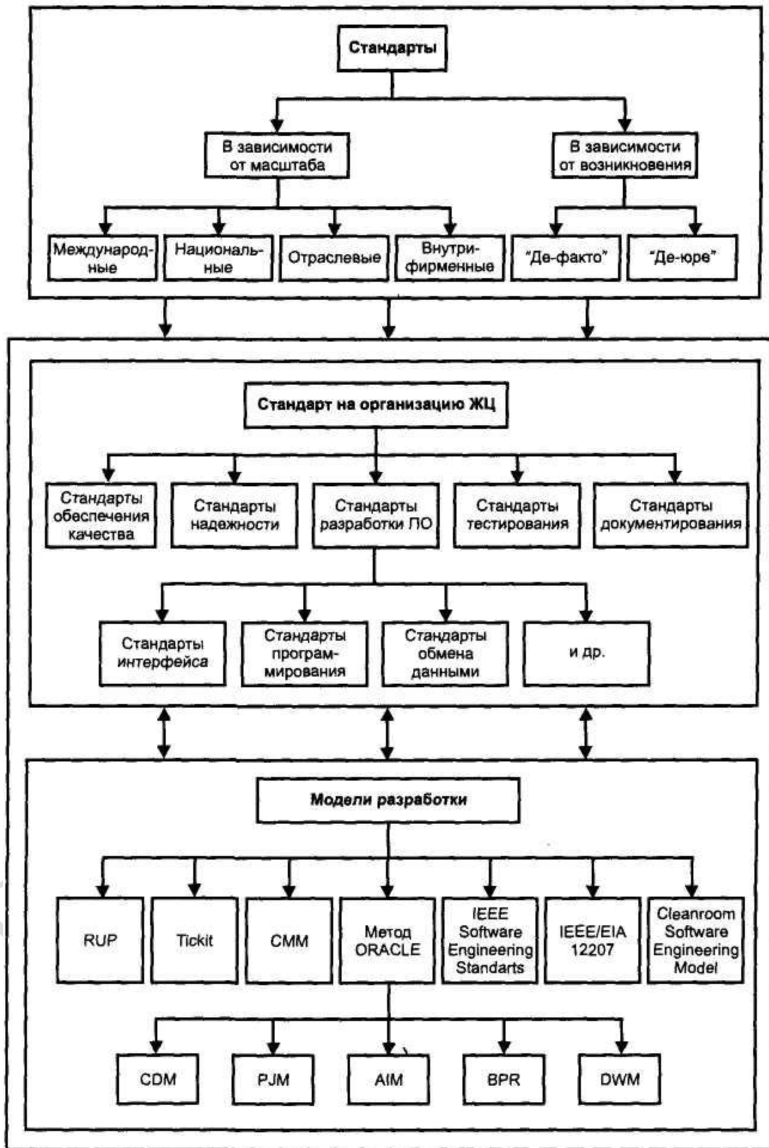

Рисунок 1.1 – Схема классификаций стандартов в области информационных технологий

Стандарты имеют большое значение – они обеспечивают возможность разработчикам ПО использовать данные и программы других разработчиков, осуществлять экспорт/импорт данных.

Такие стандарты регламентируют взаимодействие между различными программами. Для этого предназначены стандарты межпрограммного интерфейса, например OLE (Object Linking and Embedding – связывание и встраивание объектов). Без таких стандартов программные продукты были бы «закрытыми» друг для друга.

Стандарты занимают все более значительное место в направлении развития индустрии информационных технологий. Более 250 подкомитетов в официальных организациях по стандартизации работают над стандартами в области информационных технологий. Более 1000 стандартов или уже приняты этими организациями, или находятся в процессе разработки. Процесс стандартизации информационных технологий далеко не закончен (и вряд ли когда–либо будет закончен, так как область информационных технологий постоянно динамично развивается).

Все компании–разработчики должны обеспечить приемлемый уровень качества выпускаемого ПО. Для этих целей предназначены стандарты качества программного обеспечения или отдельные разделы в стандартах разработки программного обеспечения, посвященные требованиям к качеству программного обеспечения.

С точки зрения пользователя, все многообразие ПО должно управляться единообразно. Должна быть единообразная навигация – перемещение по программе, единообразные органы управления ПО и единая реакция программного обеспечения на действия пользователя. Для этого разработаны стандарты на пользовательский интерфейс – GUI (Graphical User Interface). Все это регламентируется стандартами, действующими в сфере информационных технологий.

Необходимость стандартизации разработки программного обеспечения наиболее удачно описана во введении в стандарт ISO/ IEC 12207: «Программное обеспечение является неотъемлемой частью информационных технологий и традиционных систем, таких, как транспортные, военные, медицинские и финансовые. Имеется множество разнообразных стандартов, процедур, методов, инструментальных средств и типов операционной среды для разработки и управления программным обеспечением. Это разнообразие создает трудности при проектировании и управлении программным обеспечением, особенно при объединении программных продуктов и сервисных программ. Стратегия разработки программного обеспечения требует перехода от этого множества к общему порядку, который позволит специалистам, практикующимся в программном

обеспечении, «**говорить на одном языке**» при разработке и управлении программным обеспечением. Этот международный стандарт обеспечивает такой общий порядок».

Стандарт «де–факто» – термин, обозначающий продукт какого–либо поставщика, который захватил большую долю рынка и который другие поставщики стремятся эмулировать, копировать или использовать для того, чтобы захватить свою часть рынка.

Одна из главных причин значимости современной программы стандартизации – осознание опасности злоупотребления стандартами «де–факто». В 60–е и 70–е годы XX века создание стандартов «де–факто» ставило пользователей в зависимое от производителей положение при использовании основных средств обработки данных и телекоммуникаций. Важный аспект сегодняшней работы по стандартизации – преодоление этой зависимости через продвижение стандартных интерфейсов. Долгое время такими стандартами были SQL (Structured Query Language) и язык диаграмм Д.Росса SADT (Structured Analysis and Design Technique).

Стандарт «де–юре» создается формально признанной стандартизующей организацией. Он разрабатывается при соблюдении правил консенсуса в процессе открытой дискуссии, в которой каждый имеет шанс принять участие. Ни одна группа не может действовать независимо, создавая стандарты для промышленности. Если какая–либо группа поставщиков создаст стандарт, не учитывающий требования пользователей, она потерпит неудачу. То же самое происходит, если пользователи создают стандарт, с которым не могут или не будут соглашаться поставщики, – этот стандарт также не будет успешным. Стандарты «де–юре» не могут быть изменены, не пройдя процесс согласования под контролем организации, разрабатывающей стандарты. Стандарты OSI (Open Systems Interconnection reference model), Ethernet, POSIX, SQL и большинство стандартов языков – примеры такого рода стандартов.

Следует отметить, что в области информационных технологий существуют два основных **исторически сложившихся подхода к разработке стандартов**. Первый – когда назревает проблема, – необходимость в стандарте. В этом случае собирается группа экспертов в каком–то разделе информационных технологий и обсуждает локальные решения, придуманные отдельными компаниями – производителями программного обеспечения и научными организациями, проводит анализ этих решений и разрабатывается единый интегральный стандарт, который включает в себя лучшие идеи и наработки.

Но рынок живет по несколько иным законам. Первый подход обладает инертностью, проблема уже назрела, ее надо решать, и неизвестно, когда соберутся эксперты и разработают необходимый стандарт. Во втором случае компании – разработчики ПО разрабатывают каждая свое решение, и самое популярное, массовое с точки зрения частоты использования решение обретает статус стан-

дарта (необязательно юридически). Недостаток этого подхода состоит в том, что стандартом становится не самое сильное, а самое массовое коммерческое решение.

В качестве примера появления стандарта можно привести появление ныне популярного UML – Unified Modeling Language. Основные разработки по методам объектно–ориентированного анализа и проектирования появились между 1988 и 1992 гг. К 1994 г. было большое количество неформальных лидеров разработчиков–практиков (около полутора десятков), которые продвигали свои методологии. Все их методы были схожи, при этом зачастую отличия между ними заключались во второстепенных деталях. Назревал разговор о стандартизации. Команда из OMG пыталась рассмотреть проблему стандартизации, но в ответ получила открытое письмо с протестом от всех авторов. В 1996 г. три ведущих специалиста в области объектно–ориентированного анализа и проектирования Джеймс Рамбо (James Rumbaugh), Гради Буч (Grady Booch), Ивар Якобсон (Ivar Jacobson) объединились, и появился на свет Унифицированный метод версии 0.8, в 1996 г. «трое друзей» работали над своим методом, который получил название Unified Modeling Language. В январе 1997 г. различные организации представили свои предложения по стандартизации методов, предусматривающие в первую очередь возможность обмена информацией между различными моделями. В результате сейчас имеется единственное предложение – стандарт UML.

1.4 Стандарты комплекса ГОСТ 34 на создание и развитие автоматизированных систем

Стандарты комплекса ГОСТ 34 на создание и развитие автоматизированных систем (АС) – обобщенные, но воспринимаемые как весьма жесткие по структуре жизненного цикла (ЖЦ) и проектной документации. Кроме того, эти стандарты многими считаются бюрократическими до вредности и консервативными до устарелости. ГОСТ 34 задумывался в конце 80–х годов 20 века как всеобъемлющий комплекс взаимоувязанных межотраслевых документов. Объектами стандартизации являются АС различных видов и все виды их компонентов, а не только ПО и баз данных.

Комплекс рассчитан на **взаимодействие заказчика и разработчика**. Аналогично ISO 12207 предусмотрено, что заказчик может разрабатывать АС для себя сам (если создаст для этого специализированное подразделение). Однако формулировки ГОСТ 34 не ориентированы на столь явное и, в известном смысле, симметричное отражение действий обеих сторон, как ISO 12207. Поскольку ГОСТ 34 в основном уделяет внимание **содержанию проектных документов**, распределение действий между сторонами обычно делается, отталкиваясь от этого содержания. Стадии и этапы, выполняемые организациями – участниками работ по созданию АС, устанавливаются в договорах и техническом задании, что близко к подходу ISO.

Введение единой, качественно определенной терминологии, наличие разумной классификации работ, документов, видов обеспечения и других, безусловно, полезно. ГОСТ 34 способствует более полной и качественной стыковке действительно разных систем, что особенно важно в условиях, когда разрабатываются сложные комплексы АС, которые включают в свой состав АСУ ТП, АСУП, САПР–конструктора, САПР–технолога и другие системы.

В связи с этим для каждого серьезного проекта разработки АС приходится создавать комплекты нормативных и методических документов, регламентирующих процессы создания конкретного прикладного ПО, а поэтому в отечественных разработках целесообразно использовать современные международные стандарты.

1.5 Сертификация

Сертификация в переводе с латыни – сделать верно. Для того, чтобы убедиться в том, что продукция сделана верно, надо знать, каким требованиям она должна соответствовать и каким образом возможно получить достоверные доказательства этого соответствия. Общепризнанным способом такого доказательства служит сертификат соответствия. С этим понятием связан термин испытание. Под **испытанием** понимается техническая операция, заключающаяся в определении одной или нескольких характеристик данной продукции в соответствии с установленной процедурой по принятым правилам. Испытание осуществляют в испытательных лабораториях, причём это название применяют как к юридическому, так и к техническому лицу. Систематическую проверку степени соответствия заданным требованиям принято называть оценкой соответствия. Более частым понятием оценки соответствия считают контроль, который рассматривает оценку соответствия путём измерения конкретных характеристик продукта.

Тема 2 Организации, разрабатывающие стандарты

2.1 ИСО – международная организация по стандартизации.
2.2 Международные организации, разрабатывающие стандарты.
2.3 Закрепление интеллектуальной собственности в Республике Беларусь.
2.4 Внутрифирменные (внутрикорпоративные) стандарты.

2.1 ИСО – международная организация по стандартизации

Международная организация по стандартизации (ИСО) создана в 1946г. двадцатью пятью национальными организациями по стандартизации.

При создании организации и выборе ее названия учитывалась необходимость того, чтобы аббревиатура наименования звучала одинаково на всех языках. Для этого было решено использовать греческое слово «isos» – равный.

Вот почему на всех языках мира Международная организация по стандартизации имеет краткое название ISO (ИСО).

Сфера деятельности ИСО касается стандартизации во всех областях, кроме электротехники и электроники, относящихся к компетенции Международной электротехнической комиссии (МЭК). Некоторые виды работ выполняются совместными усилиями этих организаций. Кроме стандартизации ИСО занимается и проблемами сертификации.

ИСО определяет свои задачи следующим образом: содействие развитию стандартизации и смежных видов деятельности в мире с целью обеспечения международного обмена товарами и услугами, а также развития сотрудничества в интеллектуальной, научно–технической и экономической областях.

Вопросы информационной технологии, микропроцессорной техники и т.п. входят в область совместных разработок ИСО/ МЭК. В последние годы ИСО уделяет много внимания стандартизации систем обеспечения качества. Практическим результатом усилий в этих направлениях являются разработка и издание международных стандартов. При их разработке ИСО учитывает ожидания всех заинтересованных сторон – производителей продукции (услуг), потребителей, правительственных кругов, научно–технических и общественных организаций.

На сегодняшний день в состав ИСО входят 120 стран своими национальными организациями по стандартизации. Всего в составе ИСО более 80 комитетов–членов. Кроме комитетов–членов членство в ИСО может иметь статус членов–корреспондентов, которыми являются организации по стандартизации развивающихся государств.

Довольно широки деловые контакты ИСО: с ней поддерживают связь около 500 международных организаций, в том числе все специализированные агентства ООН, работающие в смежных направлениях.

Крупнейший партнер ИСО – Международная электротехническая комиссия (МЭК). В целом эти три организации охватывают международной стандартизацией все области техники. Кроме того, они стабильно взаимодействуют в области информационных технологий и телекоммуникации.

Международные стандарты ИСО не имеют статуса обязательных для всех стран–участниц. Любая страна мира вправе применять или не применять их. Решение вопроса о применении международного стандарта ИСО связано в основном со степенью участия страны в международном разделении труда и состоянием ее внешней торговли.

ИСО поддерживает постоянные рабочие отношения с региональными организациями по стандартизации. Практически члены таких организаций одновременно являются членами ИСО. Поэтому при разработке региональных стандартов за основу принимается стандарт ИСО нередко еще на стадии проекта. Наиболее тесное сотрудничество поддерживается между ИСО и Европейским комитетом по стандартизации (СЕН).

Основная цель СЕН — помощь европейской экономике в глобальной торговле, благосостоянии европейских граждан и окружающей среды путем разработки европейских стандартов (евронорм), на которые могли бы ссылаться в своих межправительственные организации; путем обеспечения единообразного применения в странах-членах международных стандартов ИСО и МЭК; сотрудничества со всеми организациями региона, занимающимися стандартизацией; предоставления услуг по сертификации на соответствие европейским стандартам. СЕН разрабатывает европейские стандарты в таких областях, как оборудование для авиации, водонагревательные газовые приборы, газовые баллоны, комплектующие детали для подъемных механизмов, газовые плиты, сварка и резка, трубопроводы и трубы, насосные станции и др.

История стандартов качества ИСО 9000 восходит к Британским стандартам BSI 5750, которые были одобрены Британским институтом стандартов (British Standard Institute – BSI) в 1979 году. В свою очередь эти стандарты часто считаются восходящими к американским военным стандартам MIL–Q9858, принятым в конце 50–х годов в США. Стандарты серии ИСО 9000 – это пакет документов по созданию систем качества и обеспечению качества, подготовленный членами международной организации, известной как «ИСО/Технический Комитет 176» (ISO/TC 176). Ныне стандарт BSI 5750 известен как стандарт ИСО 9000 версии 1987 года. Термин «версии» означает, что в настоящее время данный стандарт пересмотрен. Причиной пересмотра стала необходимость учесть в стандартах требования к качеству ряда специфических продуктов, которые не были учтены при разработке первой версии стандартов. Кстати, одним из таких специфических продуктов было ПО, которое теперь тоже подлежит сертификации по ИСО. Три стандарта из серии ИСО 9000 (ИСО 9001, ИСО 9002 и ИСО 9003) являются фундаментальными документами Системы Качества, определяют методологию обеспечения качества и представляют собой три различные модели функциональных или организационных взаимоотношений между участниками системы качества (как правило «поставщик», «потребитель»). Собственно именно по этим стандартам и проводится сертификация «поставщика» являющегося основным объектом управления качеством.

Общие принципы и правила организации работ по сертификации систем качества в Российской Федерации определяет введенный в действие с 1 октября 1995 года национальный стандарт ГОСТ Р 40.001–95.

Базовая серия (семейство) ИСО 9000 состоит из следующих стандартов:

– ISO 9000 «Общее руководство качеством и стандарты по обеспечению качества. Руководящие указания по выбору и применению»
– ISO 9001 «Системы качества. Модель для обеспечения качества при проектировании и\или разработке, монтаже и обслуживании»
– ISO 9002 «Системы качества. Модель для обеспечения качества при производстве и монтаже»
– ISO 9003 «Системы качества. Модель для обеспечения качества при окончательном контроле и испытаниях»
– ISO 9004 «Общее руководство качеством и элементы системы качества. Руководящие указания.»

ГГУ ИМЕНИ Ф. СКОРИНЫ

Система стандартов (точнее ее подмножество – 9001–9003) обладает определенной вложенностью, т. е. каждый последующий стандарт определяет систему качества для более узкой области, нежели предыдущей (рисунок 2.1). Стандарты 9000 и 9004 определяют общие требования к системе качества и модели управления качеством.

Рисунок 2.1 – Вложенность системы стандартов ISO 9001–9003

СКОРИНЫ

2.2 Международные организации, разрабатывающие стандарты

Международная электротехническая комиссия (МЭК) создана на международной конференции, в работе которой участвовали 13 стран, в наибольшей степени заинтересованных в такой организации. Датой начала международного сотрудничества по электротехнике считается 1881г., когда состоялся первый Международный конгресс по электричеству. Позже, в 1904г., правительственные делегаты конгресса решили, что необходима специальная организация, которая бы занималась стандартизацией параметров электрических машин и терминологией в этой области.

После второй мировой войны, когда была создана ИСО, МЭК стала автономной организацией в ее составе. МЭК занимается стандартизацией в области электротехники, электроники, радио–связи, приборостроения. Эти области не входят в сферу деятельности ИСО.

Объединенный технический комитет (**JTC1 –** Joint Technical Committee 1 – Объединенный технический комитет 1) был создан в 1987г. путем объединения деятельности в области стандартизации информационных технологий (ИТ) двух организаций: ИСО и МЭК. JTC1 формирует всеобъемлющую систему базовых стандартов в области ИТ и их расширений для конкретных сфер деятельности.

JTC1 имеет 17 подкомиссий, чья работа покрывает все: от техники программного обеспечения до языков программирования, компьютерной графики

и обработки изображения, соединения оборудования, методов защиты и т.д. Работа над стандартами ИТ в JTC1 тематически распределена по подкомитетам (Subcommittees – SC).

В дополнение создана специальная группа по функциональным стандартам (Special Group on Functional Standards – SGFS) для обработки предложений по международным стандартизованным профилям (International Standardized Profiles – ISPs), представляющим определения профилей ИТ.

Ниже перечислены подкомитеты и группы JTC1, связанные с разработкой стандартов ИТ, относящихся к окружению открытых систем (Open Systems Environment – OSE):

– C2 – Символьные наборы и кодирование информации;
– SC6 – Телекоммуникация и информационный обмен между системами;
– SC7 – Разработка программного обеспечения и системная документация;
– SC18 – Текстовые и офисные системы;
– SC21 – Открытая распределенная обработка (Open Distributed Processing – ODP), управление данными (Data Management – DM) и взаимосвязь открытых систем (OSI);
– SC22 – Языки программирования, их окружение и интерфейсы системного программного обеспечения;
– SC24 – Компьютерная графика;
– SC27 – Общие методы безопасности для ИТ–приложений;
– SGFS – Специальная группа по функциональным стандартам.

Современная техника управления качеством, например, концепция **Total Quality Management** (TQM), базируется на управлении качеством. На современном этапе 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.

Американский национальный институт стандартов и технологий

Национальным органом по стандартизации в США является Американский национальный институт стандартов и технологии (NIST). Его предшественники: 1) Американский комитет технической стандартизации, который в 1928г. был реорганизован в Американскую ассоциацию по стандартизации (ASA); 2) Организация по стандартизации США (USASI), просуществовавшая менее трех лет и преобразованная в ANSI, а теперь – NIST.

NIST – неправительственная некоммерческая организация, координирующая работы по добровольной стандартизации в частном секторе экономики, руководящая деятельностью организаций – разработчиков стандартов, принимающая решения о придании стандарту статуса национального (если в нем заинтересованы различные фирмы и стандарт приобретает межотраслевой характер). NIST не разрабатывает стандарты, но является единственной организацией в США, принимающей (утверждающей) национальные стандарты. Это отвечает основной задаче NIST – содействию решения проблем, имеющих общегосударственное значение (экономия энергоресурсов, защита окружающей среды, обеспечение безопасности жизни людей и условий производства). Институт разрабатывает целевые программы. Программно–целевое планирование охватывает производство и транспортировку топлива, снабжение электроэнергией, применение ядерной, солнечной и других видов энергии. Значительно меньше внимания уделяется разработке стандартов на готовую продукцию, поскольку в этой области действуют фирменные нормативные документы.

Национальные (федеральные) стандарты содержат обязательные к выполнению требования, касающиеся в основном аспектов безопасности. Наряду с обязательными федеральными стандартами в США действуют технические регламенты, утверждаемые органами государственного управления – Министерством торговли, Министерством обороны, Управлением служб общего назначения, Федеральным агентством по охране окружающей среды, Федеральным агентством по охране труда и здоровья на производстве, Федеральным управлением по безопасности пищевых продуктов и медикаментов, Комиссией по безопасности потребительских товаров и некоторыми другими. NIST поддерживает тесные деловые контакты с этими организациями, в частности, по информационному обеспечению фирм, частных организаций, разрабатывающих стандарты. Сами указанные выше органы управления нередко участвуют в разработке фирменных стандартов и учитывают наличие таковых при планировании создания федерального стандарта. Нередки случаи, когда фирменный стандарт, удовлетворяя их требованиям, принимается в качестве федерального.

Разрабатывают федеральные стандарты авторитетные организации, аккредитованные Американским национальным институтом стандартов. Наиболее известные из них: Американское общество по контролю качества (ASQC); Американское общество инженеров–механиков (ASME); Институт инженеров по электротехнике и электронике (IEEE) и др.

Эти организации разрабатывают не только федеральные стандарты, но и стандарты, носящие добровольный характер. В учебнике Крылова Г.Д. приводятся данные, что всего в США разработкой добровольных стандартов занимаются более 400 различных организаций и фирм, а добровольных стандартов насчитывается более 35 тыс. [3, стр. 17].

На сегодняшний день членами NIST состоят более 1200 фирм, свыше 250 производственных и торговых компаний, научно–технических и инженерных обществ.

2.3 Закрепление интеллектуальной собственности в Республике Беларусь

Законом Республики Беларусь от 30.12.2002 г. №171–З «О внесении изменений дополнений в Гражданский процессуальный кодекс Республики Беларусь» расширена компетенция судебной коллегии по патентным делам. В настоящее время коллегии подсудны не только дела по спорам, кающимся объектов промышленной собственности, но и дела, связанные с защитой авторских и смежных прав, т.е. любые дела относительно всех объектов интеллектуальной собственности в целом.

Кодексом Республики Беларусь «О судоустройстве и статусе судей» от 29.06.2006 г. №139–З судебная коллегия по патентным делам Верховного Суда Республики Беларусь переименована в судебную коллегию по делам интеллектуальной собственности Верховного Суда Республики Беларусь.

Национальным центром интеллектуальной собственности от 29.08.2007г. № 146 разработана инструкция о порядке осуществления регистрации компьютерных программ.

2.4 Внутрифирменные (внутрикорпоративные) стандарты

Внутрифирменные стандарты действуют внутри организации – разработчика ПО или любой другой компании, связанной с информационными технологиями. Такие стандарты, как правило, регламентируют порядок оформления документации, приказов и технической литературы внутри компании, пользовательский интерфейс разрабатываемых приложений (например, запрет на использование некоторых элементов интерфейса), стиль программирования, спецификацию модулей, имена используемых переменных, таблиц баз данных (БД). Внутрикорпоративные (внутрифирменные) стандарты имеют узкую сферу полномочий (одна или несколько фирм), но играют большую роль, так как они абсолютно конкретны.

Внутрифирменные (внутрикорпоративные) стандарты занимают особое место в классификации стандартов. Это связано с тем, что они регламентируют технологические процессы, происходящие внутри фирмы (например, процессы анализа, кодирования, тестирования), они максимально конкретны и детализируют уровень мероприятий, если пользоваться управленческой терминологией.

Внутрифирменные стандарты, как правило, базируются на применении методик и технологий, которые:

– зарекомендовали себя лучшим образом в аналогичных проектах;
– получили наибольшее распространение в области разработки программного обеспечения;
– получили наибольшее распространение в области, для которой программное обеспечение создается;
– являются передовыми и многообещающими.

Вместе с тем внутрифирменные стандарты учитывают особенности предприятия – разработчика программного обеспечения. Его конкретные особенности связаны со средством разработки, на котором кодируется программное средство, квалификацией персонала, финансовым положением фирмы.

Можно ли разработать универсальный стандарт и тиражировать его на различных предприятиях? К сожалению, нет. Существуют общие подходы, известны технологии разработки внутрикорпоративных стандартов, но всякий раз этот процесс уникален, поскольку не существует двух совершенно одинаковых предприятий – они различаются отраслевой спецификой, размерами, стратегией, организационной структурой и многими другими факторами. Кроме того, документы, особенно относящиеся к внутреннему документообороту, различаются в силу устоявшихся бизнес–правил, традиций, корпоративной культуры, отношений между подразделениями. Внутрикорпоративные стандарты, разработанные для одного предприятия, не подойдут для другого. Поэтому типового внутрикорпоративного стандарта просто не может быть. При этом следует различать структуру бизнес–процессов, которая действительно может быть типовой, и внутрикорпоративный стандарт, согласующий структуру бизнес–процессов и организационную структуру конкретного предприятия.

Любой внутрикорпоративный стандарт должен иметь юридическую силу внутри предприятия, т.е. быть оформлен в виде документа и быть введен в действие приказом или распоряжением. В приказе ввода в действие внутрикорпоративного стандарта, как правило, должны содержаться следующие пункты: срок действия стандарта (например, «со дня подписания», «с 1 сентября 2007 г.»);область действия (распространяется на процесс кодирования и тестирования); способ доведения до исполнителей (например, «Руководителям подразделений зачитать приказ в вверенных им подразделениях»); ответственные лица за контролем исполнения (например, «Контроль за исполнением стандарта»); ответственность (например, «За невыполнение пунктов стандарта сотрудник лишается премии»).

Если вышеперечисленные пункты отсутствуют, то сложнее разбирать конфликтные ситуации, которые могут произойти. Если стандарт вообще не оформлен в виде документа, то фактически это обозначает, что его не существует вовсе, в этом случае конфликтные ситуации неизбежны. На практике на вопрос о правомерности применения того или иного проектного решения (например, использования элемента интерфейса) можно услышать: «Так было всегда».

Такая практика вредна, стандарт должен быть оформлен, а не передаваться старожилами из уст в уста.

Выявляется и ряд отрицательных моментов, связанных с внутрифирменными стандартами. Первый момент – стандарты должны тщательно разрабатываться, продумываться, и, создавая их, фирма должна учесть большое количество нюансов, чтобы не переделывать стандарт через месяц. Стандарт – это то, что дает стабильность. Второй момент находится в некотором противоречии с первым – стандарты могут тормозить использование современных технологий, средств. Это особенно важно в сфере информационных технологий, где развитие технологий и их смена идут очень быстро. Этого можно избежать, если разработать внутри фирмы механизм регулярного пересмотра стандарта для включения в него современных и передовых элементов. В комиссию по пересмотру стандартов должны входить специалисты высокой квалификации из всех заинтересованных подразделений, мнение конечного потребителя также должно быть учтено (например, если вопрос касается пользовательского интерфейса или совместимости с другими программными средствами).

Внутрифирменные стандарты можно разделить по отношению к процессам производства на *производственные* и *управленческие* стандарты. Производственные стандарты – те стандарты, которые регламентируют процессы производства программного обеспечения по этапам и стадиям жизненного цикла. Управленческие стандарты регламентируют порядок управления производственными процессами.

С помощью внутрифирменных стандартов:

– достигаются лучшие показатели обучения персонала. Соответственно проще заменить человека в случае его увольнения. Отсюда следует, что можно брать на работу специалистов более низкой квалификации и доучивать их на месте без серьезных затрат для фирмы;

– повышаются надежность и качество программного обеспечения;

– повышается дружественность программного продукта, сокращаются сроки обучения конечного пользователя;

– улучшается обслуживание, сокращаются сроки внедрения программы.

ГГУ

РАЗДЕЛ 2 ЖИЗНЕННЫЙ ЦИКЛ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Тема 3 Систематизация процессов жизненного цикла

3.1 Жизненный цикл программного обеспечения и его стандартизация.
3.2 Систематизация процессов жизненного цикла программного средства.
3.3 Основные процессы жизненного цикла программного средства.
3.4 Вспомогательные и организационные процессы жизненного цикла программного средства.

3.1 Жизненный цикл программного обеспечения и его стандартизация

При возникновении потребностей в заказе, приобретении, разработке, эксплуатации и сопровождении программ перед всеми сторонами, вовлеченными в жизненный цикл программного средства (ПС), возникает целый ряд вопросов, связанных с определением и детальным структурированием *жизненного цикла* (ЖЦ) ПС, с организационными и техническими правами и обязанностями сторон, с управлением ЖЦ и контролем за его реализацией. Одним из действенных инструментов для решения данных вопросов является использование унифицированных подходов, закрепленных в современных международных и российских стандартах.

Слова «жизненный цикл системы» или «жизненный цикл программного средства» часто появляются в статьях и звучат в разговорах разработчиков, по крайней мере руководителей проектов и подразделений. Всем понятно, что относятся они к тому, что и в какой последовательности должно делаться при создании и эксплуатации систем. Но прежде чем две организации или два специалиста договорятся о том, что конкретно входит или не входит в ЖЦ, проходит значительное время. А позже вполне может обнаружиться, что эти двое (две «стороны») все–таки по–разному понимают, какие работы будут входить в ЖЦ, а какие – нет, какие проверки будут планироваться, когда и т. д. Естественно, общие принципы организации работ описаны давно, но что делать сторонам в конкретном проекте – это каждый раз приходится решать заново.

В стандартах, регламентирующих жизненный цикл программных средств, обобщаются опыт и результаты исследований множества специалистов и рекомендуются наиболее эффективные современные методы и процессы создания и развития комплексов программ. В результате таких обобщений оттачиваются технологические процессы и приемы разработки, а также методическая база для их автоматизации.

ЖЦ ПС в стандартах представляет собой набор этапов, частных работ и операций в последовательности их выполнения и взаимосвязи, регламентирующих ведение работ от подготовки технического задания до завершения испытаний ряда версий и окончания эксплуатации ПС или информационной системы

(ИС). Стандарты включают правила описания исходной информации, способов и методов выполнения операций, устанавливают правила контроля технологических процессов, требования к оформлению их результатов, а также регламентируют содержание технологических и эксплуатационных документов на комплексы программ. Они определяют организационную структуру коллектива, обеспечивают распределение и планирование заданий, а также контроль за ходом создания ПС.

Кроме вопросов выбора типа общего устройства ЖЦ есть **проблемы** с решением частных вопросов о включении или не включении в ЖЦ отдельных работ, очень важных для качества ПС и системы: что документировать при создании системы и ПС, какие работы должны будут гарантировать качество продукта, с какой степенью организационной независимости должны выполняться проверочные процедуры разных типов, чем будет обеспечиваться соответствие разрабатываемого ПС требованиям ко всей системе и соответствие ПС потребностям в системе.

Для того чтобы привнести порядок и понимание, общие для любых сторон, участвующих в ЖЦ систем и ПС, давно разрабатывались стандарты различных уровней утверждения – национальные и международные.

Существующее многообразие номенклатуры и функциональных возможностей эксплуатируемых, разрабатываемых и перспективных ПС затрудняет использование для них традиционных методов стандартизации групп (видов) однородной продукции. В то же время обязательная реализация в ходе проекта типовых процессов ЖЦ (заказ, поставка, разработка, эксплуатация, сопровождение и т.д.) дает возможность использовать принципы и методы функциональной стандартизации, основанные на применении *базовых стандартов* и разработанных на их основе *профилей стандартов* для конкретного типа объекта (в нашем случае – проекта и системы).

Под *базовым стандартом* понимается принятый нормативный документ, регламентирующий типовые (возможно, многовариантные) требования, нормы и правила применительно к данному объекту стандартизации.

Под *профилем стандарта* понимается принятый нормативный документ, регламентирующий требования, нормы и правила, выбранные из базовых стандартов и при необходимости дополненные и/или уточненные (ограниченные) применительно к конкретной классификационной группе данного объекта стандартизации.

Основные современные зарубежные стандарты ориентированы на описание ЖЦ сложных ПС обработки информации и управления в реальном времени. К таким ПС предъявляются наиболее высокие требования по качеству функционирования, они создаются большими коллективами специалистов в течение длительного времени.

В странах СНГ первые основы построения и использования профилей стандартов ЖЦ ПС заложены принятием в качестве базового стандарта России

ГОСТ Р ИСО/МЭК12207. Данный документ тесно взаимоувязан с рядом стандартов, принятых ранее, и с некоторыми стандартами, разрабатываемыми в данное время на основе прямого применения стандартов ИСО.

Актуальность стандарта ГОСТ Р ИСО/МЭК 12207 для современных условий настолько высока, что принятие в ISO его исходного, международного варианта вскоре вызвало самую положительную оценку российских экспертов. Был дан ряд рекомендаций по его использованию в реальных условиях.

В данном стандарте программное обеспечение 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.

Следует отметить, что в Белоруссии и России создание ПО первоначально, в 70–е годы 20 века, 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, сроки их 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 стандарте ГОСТ Р ИСО/МЭК 12207 впервые реализован принцип структурной стандартизации ЖЦ ПС на основе регламентации требований к процессам, работам и задачам, входящим в полную типовую структуру ЖЦ ПС.

3.2 Систематизация процессов жизненного цикла программного средства

Процессы ЖЦ ПС выделены по принципу ответственности субъекта (заказчика, поставщика, разработчика и т. д.), реализующего конкретный процесс. В свою очередь, каждый из процессов состоит из ряда работ и решаемых при выполнении соответствующей работы задач. С точки зрения соподчиненности и

важности данных процессов они разбиты на три группы: основные; вспомогательные; организационные.

Группа *основных* процессов включает в себя процессы: *приобретение; поставка; разработка; эксплуатация; сопровождение.*

Группа *вспомогательных процессов* включает в себя процессы, обеспечивающие выполнение основных процессов: *документирование; управление конфигурацией; обеспечение качества; верификация; аттестация; оценка; аудит; решение проблем.*

Группа *организационных* процессов включает в себя процессы: *управление проектами; создание инфраструктуры проекта; определение, оценка и улучшение самого ЖЦ; обучение.*

Очень важное отличие ISO: каждый процесс, действие или задача инициируется и выполняется другим процессом по мере необходимости, причем нет заранее определенных последовательностей (естественно, при сохранении логики связей по исходным сведениям задач и т. п.).

3.3 Основные процессы жизненного цикла программного средства

Процесс поставки (supply process) охватывает действия и задачи, выполняемые поставщиком, который снабжает заказчика программным продуктом или услугой.

Процесс разработки (development process) предусматривает действия и задачи, выполняемые разработчиком, и охватывает работы по созданию ПС и его компонентов в соответствии с заданными требованиями, включая оформление проектной и эксплуатационной документации; подготовку материалов, необходимых для проверки работоспособности и соответствующего качества программных продуктов, материалов, необходимых для организации обучения персонала, и т. д. *Квалификационное тестирование ПС* проводится разработчиком в присутствии заказчика (по возможности) для демонстрации того, что ПС удовлетворяет своим спецификациям и готово к использованию в условиях эксплуатации. Квалификационное тестирование выполняется для каждого компонента ПС по всем разделам требований при широком варьировании тестов. При этом также проверяются полнота технической и пользовательской документации и ее адекватность самим компонентам ПС.

Процесс эксплуатации (operation process) охватывает действия и задачи оператора – организации, эксплуатирующей систему

Процесс сопровождения (maintenance process) 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 ПС. В соответствии со стандартом IEEE–90 под *сопровождением* понимается внесение изменений в ПС

в целях исправления ошибок, повышения производительности или адаптации к изменившимся условиям работы или требованиям.

3.4 Вспомогательные и организационные процессы жизненного цикла программного средства

Вспомогательные процессы жизненного цикла программного средства. *Процесс документирования* (documentation process) предусматривает формализованное описание информации, созданной в течение ЖЦ ПС. 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 руководители, технические специалисты и пользователи системы

Процесс управления конфигурацией (configuration management process) предполагает применение административных и технических процедур на всем протяжении ЖЦ ПС для определения состояния компонентов ПС в системе, управления модификациями ПС, описания и подготовки отчетов о состоянии компонентов ПС и запросов на модификацию, обеспечения полноты, совместимости и корректности компонентов ПС, управления хранением и поставкой ПС. Согласно стандарту IEEE–90 под *конфигурацией ПС* понимается совокупность его функциональных и физических характеристик, установленных в технической документации и реализованных в ПС.

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

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

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 документации на них требованиям заказчика, предусмотренным в договоре.

Обеспечение качества процесса предполагает гарантирование соответствия процессов ЖЦ ПС, 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[1] process) 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 ПС). 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.*

В процессе верификации проверяются следующие условия:

– непротиворечивость требований к системе и степень учета потребностей пользователей;

– возможности поставщика выполнить заданные требования;

– соответствие выбранных процессов ЖЦ ПС условиям договора;

– адекватность стандартов, процедур и среды разработки процессам ЖЦ ПС;

– соответствие проектных спецификаций ПС 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 ПС в систему;

– адекватность, полнота и непротиворечивость документации.

Процесс аттестации (validation[2] process) предусматривает определение полноты соответствия заданных требований и созданной системы или программного продукта их конкретному функциональному назначению. Под *аттестацией* обычно понимаются подтверждение и оценка достоверности проведенного тестирования ПС. Аттестация должна гарантировать полное соответствие ПС 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

---

[1] verification – проверка, подтверждение, засвидетельствование;

[2] validation – утверждение ратификация,легализация, придание законной силы.

стадиях ЖЦ ПС или как часть работы по приемке ПС 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 совместной оценки (joint review process) предназначен для оценки состояния работ по проекту и ПС, создаваемому 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 аудита (audit process) 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 ПС 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 ПС. Они определяют состояние работ, использование ресурсов, соответствие документации спецификациям и стандартам, корректность тестирования

Процесс разрешения проблем (problem resolution process) 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

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

Процесс усовершенствования (improvement process) предусматривает оценку, измерение, контроль и усовершенствование процессов ЖЦ ПС. Усовершенствование процессов ЖЦ ПС 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) охватывает первоначальное обучение и последующее постоянное повышение квалификации персонала. Приобретение, поставка, разработка, эксплуатация и сопровождение ПС в значительной степени зависят от уровня знаний и квалификации персонала. Например, разработчики ПС 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

Тема 4 Основные модели жизненного цикла

4.1 Классический жизненный цикл программных средств.
4.2 Макетирование
4.3 Стратегии конструирования программных средств
4.4 Спиральная модель жизненного цикла программных средств.
4.5 Компонентно-ориентированная модель

4.1 Классический жизненный цикл программных средств

Модель жизненного цикла: структура, состоящая из процессов, работ и задач, включающих в себя разработку, эксплуатацию и сопровождение программного продукта, охватывающая жизнь системы от установления требований к ней до прекращения ее использования.

Старейшей парадигмой процесса разработки ПО является классический жизненный цикл (автор Уинстон Ройс, 1970) [10].

Очень часто классический жизненный цикл называют каскадной или водопадной моделью, подчеркивая, что разработка рассматривается как последовательность этапов, причем переход на следующий, иерархически нижний этап происходит только после полного завершения работ на текущем этапе.

К настоящему времени наибольшее распространение получили следующие основные модели ЖЦ:

– классическая или каскадная модель (70–80-е годы 20 века);

– спиральная модель (80–90-е годы 20 века).

В изначально существовавших однородных информационных системах каждое приложение представляло собой единое целое. Для разработки такого типа приложений применялся каскадный способ. Его основной характеристикой является разбиение всей разработки на этапы формирование требований к ПО; проектирование; реализация; тестирование; ввод в действие; эксплуатация и сопровождение.

Переход с одного этапа на следующий происходит только после того, как будет полностью завершена работа на текущем. Каждый этап завершается выпуском полного комплекта документации, достаточной для того, чтобы разработка могла быть продолжена другой командой разработчиков.

Положительные стороны применения каскадного подхода заключаются в следующем: на каждом этапе формируется законченный набор проектной документации, отвечающий критериям полноты и согласованности; выполняемые в логичной последовательности этапы работ позволяют планировать сроки завершения всех работ и соответствующие затраты.

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

Каскадный подход хорошо зарекомендовал себя при построении информационных систем, для которых в самом начале разработки можно достаточно точно и полно сформулировать все требования, с тем чтобы предоставить разработчикам свободу реализовать их как можно лучше с технической точки зрения. В эту категорию попадают сложные расчетные системы, системы реального времени и другие подобные задачи.

Общепринятая модель жизненного цикла является идеальной, так как только очень простые задачи проходят все этапы без каких–либо итераций – возвратов на предыдущие шаги технологического процесса. При программировании, например, может обнаружиться, что реализация некоторой функции очень громоздка, неэффективна и вступает в противоречие с требуемой от системы производительностью. В этом случае требуется перепроектирование, а может быть, и переделка спецификаций. При разработке больших нетрадиционных систем необходимость в итерациях возникает регулярно на любом этапе жизненного цикла как из–за допущенных на предыдущих шагах ошибок и неточностей, так и из–за изменений внешних требований к условиям эксплуатации системы.

4.2 Макетирование

Достаточно часто заказчик не может сформулировать подробные требования по вводу, обработке или выводу данных для будущего программного продукта. С другой стороны, разработчик может сомневаться в адаптации продукта под операционную систему, форме диалога с пользователем или в эффективности реализуемого алгоритма. В этих случаях целесообразно использовать макетирование.

Основная цель макетирования – снять неопределенности в требованиях заказчика. Макетирование (прототипирование) – это процесс создания модели требуемого программного продукта.

Модель может принимать одну из трех форм: бумажный макет или макет на основе ПК (изображает или рисует человеко–машинный диалог); работающий макет (выполняет некоторую часть требуемых функций); существующая программа (характеристики которой затем должны быть улучшены).

Как показано на рис. 4.1, макетирование основывается на многократном повторении итераций, в которых участвуют заказчик и разработчик.

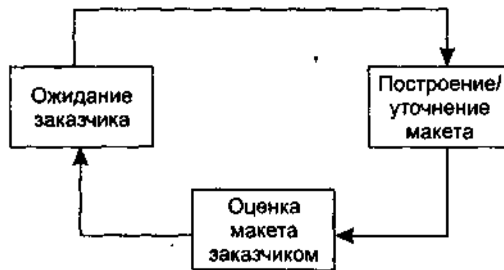

Рисунок 4.1 – Макетирование

Последовательность действий при макетировании представлена на рис. 4.2. Макетирование начинается со сбора и уточнения требований к создаваемому ПО Разработчик и заказчик встречаются и определяют все цели ПО, устанавливают, какие требования известны, а какие предстоит доопределить. Затем выполняется быстрое проектирование. В нем внимание сосредоточивается на тех характеристиках ПО, которые должны быть видимы пользователю. Быстрое проектирование приводит к построению макета. Макет оценивается заказчиком и используется для уточнения требований к ПО. Итерации повторяются до тех пор, пока макет не выявит все требования заказчика и, тем самым, не даст возможность разработчику понять, что должно быть сделано.

Достоинство макетирования: обеспечивает определение полных требований к ПО.

Недостатки макетирования: заказчик может принять макет за продукт; разработчик может принять макет за продукт.

Поясним суть недостатков. Когда заказчик видит работающую версию ПО, он перестает сознавать, что детали макета скреплены «жевательной резинкой и проволокой»; он забывает, что в погоне за работающим вариантом оставлены нерешенными вопросы качества и удобства сопровождения ПО. Когда заказчику говорят, что продукт должен быть перестроен, он начинает возмущаться и требовать, чтобы макет «в три приема» был превращен в рабочий продукт. Очень часто это отрицательно сказывается на управлении разработкой ПО.

С другой стороны, для быстрого получения работающего макета разработчик часто идет на определенные компромиссы. Могут использоваться не самые подходящие язык программирования или операционная система. Для простой демонстрации возможностей может применяться неэффективный алгоритм. Спустя некоторое время разработчик забывает о причинах, по которым эти средства не подходят. В результате далеко не идеальный выбранный вариант интегрируется в систему.

Очевидно, что преодоление этих недостатков требует борьбы с житейским соблазном – принять желаемое за действительное.

ГГУ ИМЕНИ Ф. СКОРИНЫ

Рисунок 4.2 – Последовательность действий при макетировании

4.3 Стратегии конструирования программных средств

Существуют 3 стратегии конструирования ПО:

- *однократный проход* (водопадная стратегия) – линейная последовательность этапов конструирования;
- *инкрементная стратегия.* В начале процесса определяются все пользовательские и системные требования, оставшаяся часть конструирования выполняется в виде последовательности версий. Первая версия реализует часть запланированных возможностей, следующая версия реализует дополнительные возможности и т. д., пока не будет получена полная система;
- *эволюционная стратегия.* Система также строится в виде последовательности версий, но в начале процесса определены не все требования. Требования уточняются в результате разработки версий.

Характеристики стратегий конструирования ПО в соответствии с требованиями стандарта IEEE/EIA 12207.2 приведены в табл. 4.1.

Инкрементная модель является классическим примером инкрементной стратегии конструирования (рис. 4.3). Она объединяет элементы последовательной

водопадной модели с итерационной философией макетирования. Каждая линейная последовательность здесь вырабатывает поставляемый инкремент ПО. Например, ПО для обработки слов в 1–м инкременте реализует функции базовой обработки файлов, функции редактирования и документирования; во 2–м инкременте – более сложные возможности редактирования и документирования; в 3–м инкременте – проверку орфографии и грамматики; в 4–м инкременте – возможности компоновки страницы. Первый инкремент приводит к получению базового продукта, реализующего базовые требования (правда, многие вспомогательные требования остаются нереализованными).

Таблица 4.1 – Характеристики стратегий конструирования

Стратегия конструирования	В начале процесса определены все требования?	Множество циклов конструирования?
Однократный проход	Да	Нет
Инкрементная (запланированное улучшение продукта)	Нет	Да
Эволюционная	Нет	Да

План следующего инкремента предусматривает модификацию базового продукта, обеспечивающую дополнительные характеристики и функциональность.

По своей природе инкрементный процесс итеративен, но, в отличие от макетирования, инкрементная модель обеспечивает на каждом инкременте работающий продукт.

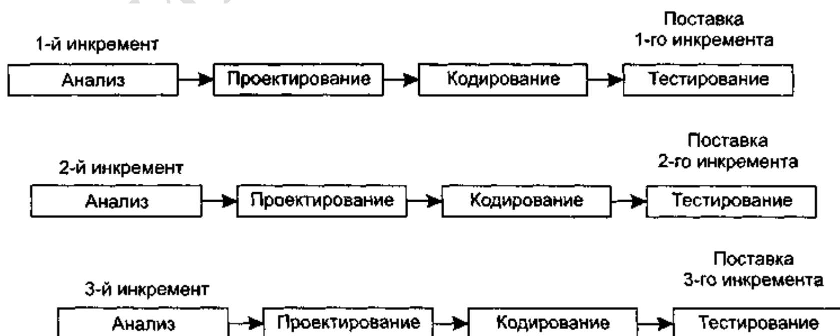

Рисунок 4.3 – Инкрементная модель

Примером современной реализации инкрементного подхода является экстремальное программирование XP. Оно ориентировано на очень малые приращения функциональности.

Модель быстрой разработки приложений (Rapid Application Development) – второй пример применения инкрементной стратегии конструирования (рис. 4.4). RAD–модель обеспечивает экстремально короткий цикл разработки. RAD – высокоскоростная адаптация линейной последовательной модели, в которой быстрая разработка достигается за счет использования компонентно–ориентированного конструирования. Если требования полностью определены, а проектная область ограничена, RAD–процесс позволяет группе создать полностью функциональную систему за очень короткое время (60–90 дней). RAD–подход ориентирован на разработку информационных систем и выделяет следующие этапы:

- **бизнес–моделирование.** Моделируется информационный поток между бизнес–функциями. Ищется ответ на следующие вопросы: Какая информация руководит бизнес–процессом? Какая генерируется информация? Кто генерирует ее? Где информация применяется? Кто обрабатывает ее?
- **моделирование данных**. Информационный поток, определенный на этапе бизнес–моделирования, отображается в набор объектов данных, которые требуются для поддержки бизнеса. Идентифицируются характеристики (свойства, атрибуты) каждого объекта, определяются отношения между объектами;
- **моделирование обработки.** Определяются преобразования объектов данных, обеспечивающие реализацию бизнес–функций. Создаются описания обработки для добавления, модификации, удаления или нахождения (исправления) объектов данных;
- **генерация приложения.** Предполагается использование методов, ориентированных на языки программирования 4–го поколения. Вместо создания ПО с помощью языков программирования 3–го поколения, RAD–процесс работает с повторно используемыми программными компонентами или создает повторно используемые компоненты. Для обеспечения конструирования используются утилиты автоматизации;
- **тестирование и объединение.** Поскольку применяются повторно используемые компоненты, многие программные элементы уже протестированы. Это уменьшает время тестирования (хотя все новые элементы должны быть протестированы).

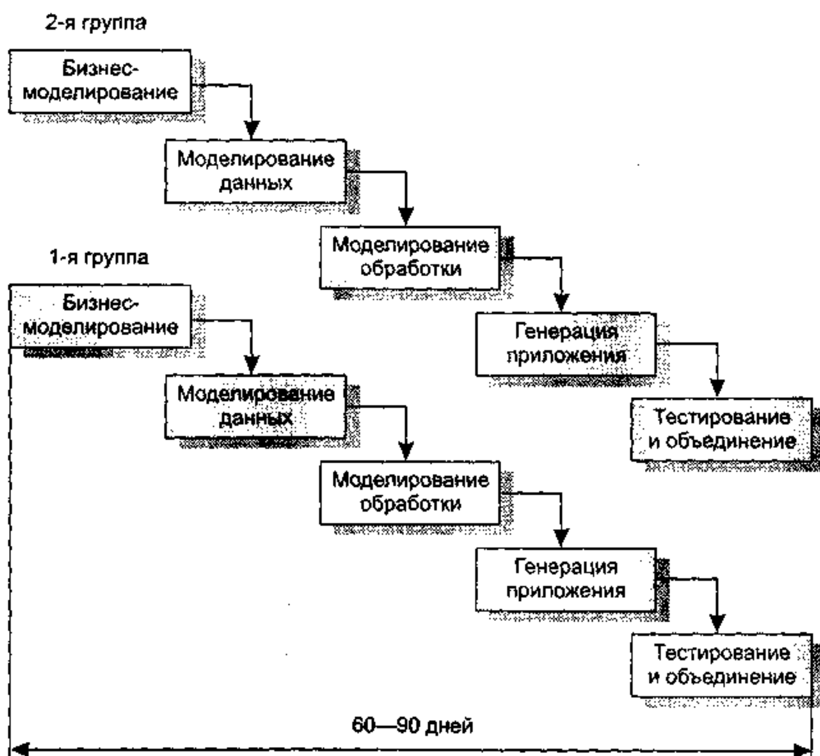

Рисунок 4.4 – Модель быстрой разработки приложений

Применение RAD возможно в том случае, когда каждая главная функция может быть завершена за 3 месяца. Каждая главная функция адресуется отдельной группе разработчиков, а затем интегрируется в целую систему.

Применение RAD имеет и свои недостатки, и ограничения.

1. Для больших проектов в RAD требуются существенные людские ресурсы (необходимо создать достаточное количество групп).
2. RAD применима только для таких приложений, которые могут декомпозироваться на отдельные модули и в которых производительность не является критической величиной.
3. RAD не применима в условиях высоких технических рисков (то есть при использовании новой технологии).

4.4 Спиральная модель жизненного цикла программных средств

Для преодоления перечисленных проблем была предложена спиральная модель ЖЦ, общая схема которой приведена на рисунке 4.5а). Разработка ПО осуществляется поэтапно (определение требований, анализ, проектирование, реализация и тестирование, интеграция) в виде итераций, делая упор на начальных этапах ЖЦ: анализ и проектирование. На этих этапах реализуемость технических решений проверяется путем создания прототипов. Каждый виток спирали соответствует созданию фрагмента или версии ПС, на нем уточняются цели и характеристики проекта, определяется его качество и планируются работы следующего витка спирали. Таким образом, углубляются и последовательно конкретизируются детали проекта, и в результате выбирается обоснованный вариант, который доводится до реализации.

Разработка итерациями отражает объективно существующий спиральный цикл создания системы. Неполное завершение работ на каждом этапе позволяет переходить на следующий этап, не дожидаясь полного завершения работы на текущем. При итеративном способе разработки недостающую работу можно будет выполнить на следующей итерации. Главная же задача – как можно быстрее показать пользователям системы работоспособный продукт, тем самым, активизируя процесс уточнения и дополнения требований.

Основная проблема спирального цикла – определение момента перехода на следующий этап. Для ее решения необходимо ввести временные ограничения на каждый из этапов жизненного цикла. Переход осуществляется в соответствии с планом, даже если не вся запланированная работа закончена. План составляется на основе статистических данных, полученных в предыдущих проектах, и личного опыта разработчиков.

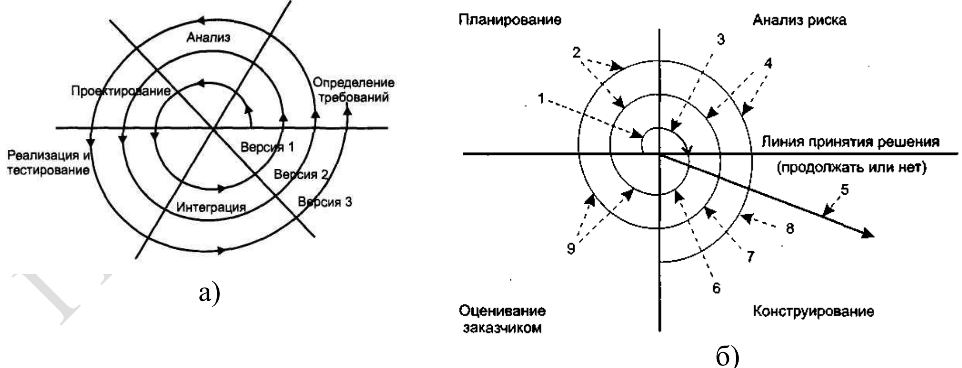

Рисунок 4.5 – Спиральная модель жизненного цикла

а) общая схема спиральной модели жизненного цикла;
б) модифицированная спиральная модель: *1* – начальный сбор требований и планирование проекта; *2* – та же работа, но на основе рекомендаций заказчика; 3 – анализ риска на ос-

нове начальных требований; *4* – анализ риска на основе реакции заказчика; 5 – переход к комплексной системе; 6 – начальный макет системы; 7 – следующий уровень макета; *8* – сконструированная система; 9 – оценивание заказчиком

На 4.5б) приведена модифицированная спиральная модель, предложенная Барри Боэм, базирующаяся на лучших свойствах классического жизненного цикла и макетирования, к которым добавляется новый элемент – анализ риска, отсутствующий в этих парадигмах. Как показано на 4.5б), модель определяет четыре действия, представляемые четырьмя квадрантами спирали: планирование – определение целей, вариантов и ограничений; анализ риска – анализ вариантов и распознавание/выбор риска; конструирование – разработка продукта следующего уровня; оценивание – оценка заказчиком текущих результатов конструирования.

Интегрирующий аспект спиральной модели (рис.4.5б)) очевиден при учете радиального измерения спирали. С каждой итерацией по спирали (продвижением от центра к периферии) строятся все более полные версии ПО. В первом витке спирали определяются начальные цели, варианты и ограничения, распознается и анализируется риск. Если анализ риска показывает неопределенность требований, на помощь разработчику и заказчику приходит макетирование (используемое в квадранте конструирования). Для дальнейшего определения проблемных и уточненных требований может быть использовано моделирование. Заказчик оценивает инженерную (конструкторскую) работу и вносит предложения по модификации (квадрант оценки заказчиком). Следующая фаза планирования и анализа риска базируется на предложениях заказчика. В каждом цикле по спирали результаты анализа риска формируются в виде «продолжать, не продолжать». Если риск велик, проект может быть остановлен.

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

Достоинства спиральной модели: наиболее реально (в виде эволюции) отображает разработку программного обеспечения; позволяет явно учитывать риск на каждом витке эволюции разработки; включает шаг системного подхода в итерационную структуру разработки; использует моделирование для уменьшения риска и совершенствования программного изделия.

Недостатки спиральной модели: новизна (отсутствует достаточная статистика эффективности модели); повышенные требования к заказчику; трудности контроля и управления временем разработки.

4.5 Компонентно–ориентированная модель

ГГУ ИМЕНИ Ф. СКОРИНЫ

Компонентно–ориентированная модель является развитием спиральной модели (рис.4.5б)) и тоже основывается на эволюционной стратегии конструирования. В этой модели конкретизируется содержание квадранта конструирования – оно отражает тот факт, что в современных условиях новая разработка должна основываться на повторном использовании существующих программных компонентов (рис. 4.6).

Программные компоненты, созданные в реализованных программных проектах, хранятся в библиотеке. В новом программном проекте, исходя из требований заказчика, выявляются кандидаты в компоненты. Далее проверяется наличие этих кандидатов в библиотеке. Если они найдены, то компоненты извлекаются из библиотеки и используются повторно. В противном случае создаются новые компоненты, они применяются в проекте и включаются в библиотеку.

Достоинства компонентно–ориентированной модели: уменьшает на 30% время разработки программного продукта; уменьшает стоимость программной разработки до 70%; увеличивает в полтора раза производительность разработки.

Для традиционных подходов итерация – это исправление ошибок, т.е. процесс, который с трудом поддается технологическим нормам и регламентам. При объектно–ориентированном подходе итерации никогда не отменяют результаты друг друга, а всегда только дополняют и развивают их. Особенностью объектно–ориентированного моделирования жизненного цикла является учет непрерывно поступающих требований к разрабатываемому проекту.

ГГУ ИМЕНИ Ф. СКОРИНЫ

Рисунок 4.6 – Компонентно–ориентированная модель

ГГУ ИМЕНИ

РАЗДЕЛ 3 СТАНДАРТЫ ДОКУМЕНТИРОВАНИЯ ПРОГРАММНЫХ СРЕДСТВ

Тема 5 Общая характеристика проблем и задач документирования программного обеспечения

5.1 Проблемы и задачи создания программной документации.

5.2 Общая характеристика состояния в области документирования программных средств.

5.3 Основные недостатки ЕСПД

5.1 Проблемы и задачи создания программной документации

Создание программной документации – важный этап, так как пользователь начинает свое знакомство с программным продуктом именно с документации. Для чего предназначен программный продукт, как установить программный продукт, как начать с ним работать – вот одни из первых вопросов, на которые должна отвечать программная документация (Installation Guide, Getting Started). Составлением программной документации обычно занимаются специальные люди – технические писатели (иногда программную документацию пишут сами программисты или аналитики). Этот этап является самым неприятным и тяжелым в программистской работе. К сожалению, обычно этому либо не учат совсем, либо в лучшем случае не обращают на качество получаемых документов должного внимания. Тем не менее, владение этим искусством является одним из важнейших факторов, определяющих качество программиста.

Умение создавать программную документацию определяет профессиональный уровень программиста. Заказчик не будет вникать в тонкости и особенности даже самой замечательной программы. Заказчик будет сначала читать документацию. Большую роль играет в этом и психологический фактор.

Грамотно созданный пакет программной документации избавит разработчиков ПО от многих неприятностей. В частности, избавиться от назойливых вопросов и необоснованных претензий, отослав пользователя к документации. Это касается, прежде всего, важнейшего документа – Технического задания. Многомиллионный иск в 70–х годах 20 века, предъявленный к компании IBM крупным издательством о неудовлетворительном качестве вычислительной техники и программного обеспечения, был в суде выигран благодаря предъявленному подписанному обеими сторонами Техническому заданию.

Вообще программную документацию можно разделить по отношению к пользователю на внутреннюю и внешнюю. Внешняя – всевозможные руководства для пользователей, техническое задание, справочники; внутренняя документация – та, которая используется в процессе разработки программного обеспе-

чения и недоступна конечному пользователю (различные внутренние стандарты, комментарии исходного текста, технологии программирования и т.д.).

Когда программист–разработчик получает в той или иной форме задание на программирование, перед ним, перед руководителем проекта и перед всей проектной группой встают вопросы: Что должно быть сделано, кроме собственно программы? Что и как должно быть оформлено в виде документации? Что передавать пользователям, а что – службе сопровождения? Как управлять всем этим процессом? Что должно входить в само задание на программирование?

На эти и другие вопросы когда–то отвечали государственные стандарты на программную документацию – комплекс стандартов 19–й серии ГОСТ ЕСПД. Но уже тогда у программистов была масса претензий к этим стандартам. Что–то требовалось дублировать в документации много раз (как оказалось – неоправданно), а многое не было предусмотрено, как, .например, отражение специфики документирования программ, работающих с интегрированной базой данных.

5.2 Общая характеристика состояния в области документирования программных средств

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 20 века. Сейчас этот комплекс представляет собой систему межгосударственных стандартов стран СНГ (ГОСТ), действующих на территории Беларуси, на основе межгосударственного соглашения по стандартизации.

Единая система программной документации – это комплекс государственных стандартов, устанавливающих взаимоувязанные правила разработки, оформления и обращения программ и программной документации.

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 «О стандартизации» эти стандарты становятся обязательными на контрактной основе, т.е. при ссылке на них в договоре на разработку (поставку) ПС.

В состав ЕСПД входят:

– основополагающие и организационно–методические стандарты;
– стандарты, определяющие формы и содержание программных документов, применяемых при обработке данных;
– стандарты, обеспечивающие автоматизацию разработки программных документов.

5.3 Основные недостатки ЕСПД

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 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.

ЕСПД нуждается в полном пересмотре на основе стандарта ИСО/МЭК 12207–95 на процессы жизненного цикла ПС. Тем не менее, до пересмотра всего комплекса многие стандарты 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 программных документов (ПД), необходимых в конкретных проектах, и исключать многие ПД;

– 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.

Надо сказать, что наряду с комплексом ЕСПД официальная нормативная база СНГ в области документирования ПС и в смежных областях включает ряд перспективных стандартов (отечественного, межгосударственного и международного уровней). Международный стандарт ISO/IEC 12207:1995 на организа-

цию ЖЦ продуктов ПО, казалось бы, весьма неконкретный, но вполне новый и отчасти «модный» стандарт.

Тема 6 Единая система программной документации

6.1 Общая характеристика Единой системы программной документации.
6.2 Виды программ и программных документов (ГОСТ 19.101–77 ЕСПД).
6.3 Стадии разработки (ГОСТ 19.102–77 ЕСПД).
6.4 Краткая характеристика некоторых ГОСТов программной документации.

6.1 Общая характеристика Единой системы программной документации

Стандарты ЕСПД (как и другие ГОСТы) подразделяют на группы, приведенные в таблице 6.1.

Таблица 6.1 – Группы стандартов ЕСПД

Код группы	Наименование группы
о	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 8	Резервные группы
9	Прочие стандарты

Обозначение стандарта ЕСПД должно состоять из:
– числа 19 (присвоенных классу стандартов ЕСПД);
– одной цифры (после точки), обозначающей код классификационной группы стандартов, указанной в таблице 6.1;
– двух цифр, обозначающих порядковый номер документа в группе стандартов соответствующего кода;
– двузначного числа (после тире), указывающего год регистрации стандарта.

Вообще перечень документов ЕСПД очень обширен. В него, в частности, входят следующие ГОСТы:

ГОСТ 19.001–77 ЕСПД. Общие положения.

ГОСТ 19.005–85 ЕСПД. Р–схемы алгоритмов и программ. Обозначения условные графические и правила выполнения.

ГОСТ 19.101–77 ЕСПД. Виды программ и программных документов.

ГОСТ 19.102–77 ЕСПД. Стадии разработки.

ГОСТ 19.103–77 ЕСПД. Обозначение программ и программных документов.

ГОСТ 19.104–78 ЕСПД. Основные надписи.

ГОСТ 19.105–78 ЕСПД. Общие требования к программным документам.

ГОСТ 19.106–78 ЕСПД. Требования к программным документам, выполненным печатным способом.

ГОСТ 19.201–78 ЕСПД. Техническое задание. Требования к содержанию и оформлению.

ГОСТ 19.202–78 ЕСПД. Спецификация. Требования к содержанию и оформлению.

ГОСТ 19.301–79 ЕСПД. Порядок и методика испытаний.

ГОСТ 19.401–78 ЕСПД. Текст программы. Требования к содержанию и оформлению.

ГОСТ 19.402–78 ЕСПД. Описание программы.

ГОСТ 19.403–79 ЕСПД. Ведомость держателей подлинников.

ГОСТ 19.404–79 ЕСПД. Пояснительная записка. Требования к содержанию и оформлению.

ГОСТ 19.501–78 ЕСПД. Формуляр. Требования к содержанию и оформлению.

ГОСТ 19.502–78 ЕСПД. Описание применения. Требования к содержанию и оформлению.

ГОСТ 19.503–79 ЕСПД. Руководство системного программиста. Требования к содержанию и оформлению.

ГОСТ 19.504–79 ЕСПД. Руководство программиста. Требования к содержанию и оформлению.

ГОСТ 19.505–79 ЕСПД. Руководство оператора. Требования к содержанию и оформлению.

ГОСТ 19.506–79 ЕСПД. Описание языка. Требования к содержанию и оформлению.

ГОСТ 19.507–79 ЕСПД. Ведомость эксплуатационных документов.

ГОСТ 19.508–79 ЕСПД. Руководство по техническому обслуживанию. Требования к содержанию и оформлению.

ГОСТ 19.601–78 ЕСПД. Общие правила дублирования, учета и хранения.

ГОСТ 19.602–78 ЕСПД. Правила дублирования, учета и хранения программных документов, выполненных печатным образом.

ГОСТ 19.603–78 ЕСПД. Общие правила внесения изменений.

ГОСТ 19.604–78 ЕСПД. Правила внесения изменений в программные документы, выполняемые печатным способом.

ГОСТ 19.701–90 ЕСПД. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения.

ГОСТ 19781–90. Обеспечение систем обработки информации программное. Термины и определения.

Прежде чем приступить к рассмотрению правил составления программной документации, необходимо сделать следующее замечание. Каждый документ желательно предварять некоторым введением. Во введении говорится об актуальности, о необходимости и т.п. Цель исполнителя здесь – показать значимость и необходимость выполнения этой работы.

Из всех стандартов ЕСПД остановимся только на тех, которые чаще используются на практике.

Первым укажем стандарт, который устанавливает виды программ и программных документов для вычислительных машин, комплексов и систем независимо от их назначения и области применения.

6.2 Виды программ и программных документов (ГОСТ 19.101–77 ЕСПД)

ГОСТ подразделяет программы на следующие виды: компонент и комплекс. *Компонент* – программа, рассматриваемая как единое целое, выполняющая законченную функцию и применяемая самостоятельно или в составе комплекса. *Комплекс* – программа, состоящая из двух или более компонентов, выполняющих взаимосвязанные функции, и применяемая самостоятельно или в составе другого комплекса.

Документация, разработанная на программу, может использоваться для реализации и передачи программы на носителях данных, а также для изготовления программного изделия. К числу программных данный ГОСТ относит документы, содержащие сведения, необходимые для разработки, изготовления, сопровождения и эксплуатации программ. Рассмотрим виды программных документов и их содержание:

Спецификация – содержит состав программы и документацию на нее.

Ведомость держателей подлинников – содержит перечень предприятий, на которых хранят подлинники программных документов.

Текст программы – представляет запись программы с необходимыми комментариями.

Описание программы – содержит сведения о логической структуре и функционировании программы.

Программа и методика испытаний – содержит требования, подлежащие проверке при испытании программы, а также порядок и методы их контроля.

Техническое задание – описывает назначение и область применения программы, технические, технико–экономические и специальные требования, предъявляемые к программе, необходимые стадии и сроки разработки, виды испытаний.

Пояснительная записка – содержит схему алгоритма, общее описание алгоритма и (или) функционирования программы, а также обоснование принятых технических и технико–экономических решений.

Эксплуатационные документы – содержат сведения для обеспечения функционирования и эксплуатации программы.

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.

Допускается объединять отдельные виды эксплуатационных документов (за исключением ведомости эксплуатационных документов и формуляра). Необходимость объединения этих документов указывается в техническом задании. Объединенному документу присваивают наименование и обозначение одного из объединяемых документов. В объединенных документах должны быть приведены сведения, которые необходимо включать в каждый объединяемый документ.

6.3 Стадии разработки (ГОСТ 19.102–77 ЕСПД)

Устанавливает стадии разработки программ и программной документации для вычислительных машин, комплексов и систем независимо от их назначения и области применения (таблица 6.2).

Допускается исключать вторую стадию разработки, а в технически обоснованных случаях – вторую и третью стадии. Необходимость проведения этих стадий указывается в техническом задании.

Допускается объединять, исключать этапы работ и (или) их содержание, а также вводить другие этапы работ по согласованию с заказчиком.

6.4. Краткая характеристика некоторых ГОСТов по программной документации

Общие требования к программным документам (ГОСТ 19.105–78 ЕСПД). Данный 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. Титульная часть оформляется согласно ГОСТ 19.104–78. Информационная часть должна состоять из аннотации и содержания. В аннотации приводят сведения о назначении документа и краткое изложение основной части. Содержание включает перечень записей о структурных элементах основной части документа. Состав и структура основной части программного документа устанавливаются стандартами ЕСПД на соответствующие документы.

Техническое задание. Требования к содержанию и оформлению(ГОСТ 19.201–78 ЕСПД.)

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.

Рассмотрим требования к программной документации. В данном разделе должны быть указаны предварительный состав программной документации и при необходимости специальные требования к ней.

Описание программы (ГОСТ 19.402–78 ЕСПД.)

Данный стандарт определяет состав и требования к содержанию программного документа «Описание программы».

Описание программы включает:

1 Общие сведения.
2 Функциональное назначение.
3 Описание логической структуры.
4 Используемые технические средства.
5 Вызов и загрузка.
6 Входные данные.

7 Выходные данные.

Таблица 6.2 – Стадии разработки, этапы и содержание работ

Стадия разработки	Этап работы	Содержание работ
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
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
	Разработка программы	Программирование и отладка программы
	Разработка программной документации	Разработка программных документов в соответствии с требованиями ГОСТ 19.101–77
Рабочий проект	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

В разделе *Общие сведения* указывают:

– обозначение и наименование программы;

– программное обеспечение, необходимое для функционирования программы;

– языки программирования, на которых написана программа.

Раздел *Функциональное назначение* должен отражать классы решаемых задач и/или назначение программы, сведения о функциональных ограничениях на применение.

При описании *логической структуры* должны быть отражены:

– алгоритм программы;

– используемые методы;

– структура программы с описанием функций составных частей и связей между ними;

– связи программы с другими программами.

В разделе *Используемые технические средства* указывают типы ЭВМ и устройств, которые используются при работе программы.

При описании раздела *Вызов и загрузка* указывают способ вызова программы с соответствующего носителя данных и входные точки в программу.

Раздел Входные данные отражает:

– характер, организацию и предварительную подготовку входных данных;

– формат, описание и способ кодирования входных данных.

– Раздел *Выходные данные* отражает:

– характер и организацию выходных данных;

– формат, описание и способ кодирования выходных данных.

Пояснительная записка. **(ГОСТ 19.404–79 ЕСПД).** Требования к содержанию и оформлению.

Согласно данному стандарту пояснительная записка должна включать следующие разделы:

1 Введение.

2 Назначение и область применения.

3 Технические характеристики.

4 Ожидаемые технико–экономические показатели.

5 Источники, использованные при разработке.

Введение должно содержать наименование программы и/или обозначение темы разработки, а также документы, на основе которых ведется разработка.

При описании *назначения и области применения* указывают назначение программы, краткую характеристику области применения программы.

В разделе *Технические характеристики* содержатся:

– постановка задачи на разработку программы, описание при меняемых математических методов и различных ограничений, связанных с выбранным математическим аппаратом;

– описание алгоритма и/или функционирования программы с обоснованием выбора схемы алгоритма решения задачи, возможного взаимодействия программы с другими программами;

– описание и обоснование выбора метода организации входных и выходных данных;

– описание и обоснование выбора состава технических и программных средств на основе проведенных расчетов и анализов, распределение носителей данных, которые использует программа.

В качестве *ожидаемых технико–экономических показателей* указывают показатели, обосновывающие преимущество выбранного варианта технического решения, а также при необходимости ожидаемые оперативные показатели.

При описании источников, использованных при разработке, необходимо привести перечень научно–технических публикаций, нормативно–технических документов и других научно–технических материалов, на которые есть ссылки в основном тексте.

Руководство системного программиста. Требования к содержанию и оформлению (ГОСТ 19.503–79 ЕСПД.)

Руководство системного программиста должно содержать следующие разделы:

1 Общие сведения о программе.

2 Структура программы.

3 Настройка программы.

4 Проверка программы.

5 Дополнительные возможности.

6 Сообщения системному программисту.

При необходимости допускается опускать раздел, описывающий дополнительные возможности.

При описании *общих сведений о программе* необходимо указать назначение и функции программы и сведения о технических и программных средствах, обеспечивающих выполнение данной программы.

В разделе *Структура программы* приводятся сведения о структуре программы, ее составных частях и связях с другими программами.

Раздел *Настройка программы* должен содержать описание действий по настройке программы на условия конкретного применения.

При описании *проверки программы* необходимо привести и описать способы проверки, позволяющие дать общее заключение о работоспособности программы (контрольные примеры, методы прогона, результаты).

Раздел *Дополнительные возможности* должен содержать описание дополнительных разделов функциональных возможностей программы и способов их выбора.

В разделе *Сообщения системному программисту* необходимо указать тексты сообщений, выдаваемых в ходе выполнения программы, описание содержания и действий, которые необходимо предпринять по этим сообщениям.

Руководство программиста. Требования к содержанию и оформлению (ГОСТ 19.504–79 ЕСПД.)

Руководство программиста должно содержать разделы:

1 Назначение и условия применения программы.

2 Характеристики программы.

3 Обращение к программе.

4 Входные и выходные данные.

5 Сообщения.

При описании *назначения и условий применения программы* необходимо указать назначение и функции, выполняемые программой; условия, необходимые для выполнения программы: объем оперативной памяти, требования к составу и параметрам периферийных устройств; требования к ПО и т.д.

В разделе *Характеристики программы* необходимо привести описание основных характеристик и особенностей программы: временных характеристик, режима работы, средств контроля правильности выполнения и самовосстанавливаемости программы и т.д.

Раздел *Обращение к программе* представляет собой описание процедур вызова программы (способов передачи управления и параметров данных и др.).

Раздел *Входные и выходные данные* должен содержать описание организации используемой входной и выходной информации и при необходимости ее кодирования.

При описании *сообщений* необходимо привести тексты сообщений, выдаваемых программисту или оператору в ходе выполнения программы, описание их содержания и действия, которые необходимо предпринять по этим сообщениям.

Документация пользователя.

Документация пользователя (user documentation): полный комплект документов, поставляемых в печатном или другом виде, который обеспечивает применение продукта, а также является его неотъемлемой частью.

Документация пользователя должна отвечать следующим характеристикам.

Полнота (completeness). Документация пользователя должна содержать информацию, необходимую для использования продукта. В ней должны быть полностью описаны все функции, установленные в описании продукта, и все вызываемые пользователем функции из программы. Кроме того, граничные значения, заданные в описании продукта, должны быть продублированы в документации пользователя. Если установка (инсталляция) продукта может быть проведена пользователем, то в документацию пользователя должно быть включено руководство по установке продукта, содержащее всю необходимую информацию. Если сопровождение продукта может проводиться пользователем, то в документацию пользователя должно быть включено руководство по сопровож-

дению программы, содержащее всю информацию, которая необходима для обеспечения данного вида сопровождения.

Правильность (correctness). Вся информация в документации пользователя должна быть правильной. Кроме того, представление данной информации не должно содержать неоднозначных толкований и ошибок.

Непротиворечивость (consistency). Документы, входящие в комплект документации пользователя, не должны противоречить сами себе, друг другу и описанию продукта. Каждый термин должен иметь один и тот же смысл во всех документах.

Понятность (understandability). Документация пользователя должна быть понятной для сообщества пользователей, выполняющих указанную рабочую задачу, например, посредством использования в ней соответствующим образом подобранных терминов, графических вставок, уточняющих пояснений и путем ссылок на полезные источники информации.

Простота обозрения (ease of overview). Документация пользователя должна быть достаточно проста для изучения пользователем, чтобы он мог выявить все описываемые в ней взаимосвязи компонентов продукта. В каждый документ могут быть включены оглавление и предметный указатель.

Резюмируя, скажем, что возникла настоятельная потребность во введении в отечественные стандарты на документирование ПС тех норм, правил, требований и рекомендаций, которые установлены на международном и передовом зарубежном уровнях. Но при проведении этих работ нельзя ограничиваться прямым переводом отдельных международных стандартов. Нужно, чтобы новые стандарты правильно стыковались со всем имеющимся и будущим множеством нормативных документов.

ГГУ ИМ

РАЗДЕЛ 4 НАДЕЖНОСТЬ И КАЧЕСТВО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Тема 7 Основные понятия и показатели надежности программного обеспечения

7.1 Проблема и пути обеспечения надежности сложных информационных систем.

7.2 Пути обеспечения надежности сложных информационных систем.

7.3 Особенности применения основных понятий теории надежности сложных систем к жизненному циклу и оценке качества программного обеспечения.

7.4 Показатели качества и надежности программных средств.

7.1 Проблема обеспечения надежности сложных информационных систем

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 надежности должно реализовываться специалистами в жизненном цикле программных средств **на основе использования** современной методологии, технологического инструментария, стандартов и нормативных документов.

Для обеспечения 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. Учет факторов, влияющих на затраты ресурсов при создании конкретного ПС, должен позволять рационализировать их использование и добиваться заданной надежности функционирования ПС при минимальных или допустимых затратах.

Для обеспечения надежности программных средств в конкретных проектах **должны быть организованы и стимулированы разработка, освоение и применение современных автоматизированных технологий и инструментальных средств**, обеспечивающих *предупреждение или исключение* большинства видов дефектов и ошибок при создании и модификации ПС и их компонентов. Ограниченные ресурсы на разработку приводят к необходимости упорядоченного применения методов и рационального использования средств автоматизации проектирования. Поэтому процесс разработки должен планироваться и последовательно проходить этапы, охватывающие все компоненты ПС. *Контроль надежности и безопасности* создаваемых и модифицируемых программ должен сопровождать весь жизненный цикл ПС посредством специальной, достаточно эффективной технологической системы обеспечения их качества. Разработка и сопровождение сложных ПС на базе CASE–технологий позволяют предупреждать и устранять наиболее опасные системные и алгоритмические ошибки на ранних стадиях проектирования, а также использовать неоднократно проверенные в других проектах программные и информационные компоненты высокого качества. Предупреждение ошибок должно поддерживаться высококачественной документацией в процессе создания ПС в целом и их компонентов.

7.2 Пути обеспечения надежности сложных информационных систем

Одним из эффективных путей повышения 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.

Для обнаружения и устранения ошибок проектирования все этапы разработки и сопровождения ПС должны быть поддержаны методами и средствами систематического, автоматизированного *тестирования* и *испытаний.* При разработке ПС целесообразно применять различные методы, эталоны и виды тестирования, каждый из которых ориентирован на обнаружение, локализацию или диагностику определенных типов дефектов. Удостоверению достигнутого качества и надежности функционирования сложных критических ПС должна сопутствовать *обязательная сертификация* аттестованными проблемно–ориентированными сертификационными лабораториями.

В сложных комплексах программ при любой технологии разработки невозможно гарантировать абсолютное отсутствие дефектов и ошибок. Непредсказуемость вида, места и времени проявления дефектов ПС в процессе эксплуатации приводит к необходимости создания специальных, дополнительных *систем автоматической оперативной защиты* от непредумышленных, случайных искажений вычислительного процесса, программ и данных.

В отечественных ИС все больше применяются программные компоненты зарубежных фирм, которые также не могут быть абсолютно гарантированы от проявления дефектов проектирования, программирования и документации. Для

обеспечения надежности функционирования комплексов программ с использованием импортных компонентов следует закупать только *лицензионно–чистые продукты,* поддерживаемые гарантированным сопровождением конкретных фирм–поставщиков. Эти компоненты должны сопровождаться полной эксплуатационной и технической документацией, сертификатом соответствия и комплектами тестов. В контрактах на закупку должны специально фиксироваться обязательства поставщиков по длительному сопровождению и замене версий ПС при выявлении дефектов или совершенствовании функций. Все версии зарубежных ПС следует проверять на надежность функционирования в конкретном окружении проекта ИС путем повторных испытаний или отдельными проверками, подтверждающими зарубежный сертификат.

Экспериментальное определение реальной надежности функционирования сложных комплексов программ – весьма трудоемкая, трудно автоматизируемая и не всегда безопасная часть жизненного цикла ПС. Накоплен значительный опыт определения надежности ПС, применяемых в авиационной, ракетно–космической и других областях современной высокоинтеллектуальной техники. В этих областях недопустимо для тестирования и определения надежности ПС использовать функционирование реальных объектов. В результате особое значение приобрели методы и средства моделирования внешней среды для автоматизированной генерации тестов при испытаниях надежности таких ПС. В этих случаях на базе программных моделей и компонентов реальных систем должны создаваться моделирующие испытательные стенды, обеспечивающие возможность определения надежности функционирования конкретных ПС в условиях штатных и критических внешних воздействий, соответствующих подлинным характеристикам внешней среды.

Быстрый рост числа сфер использования, сложности и ответственности функций, выполняемых комплексами программ в информационных системах, резко повысил в последнее время требования к надежности их функционирования и безопасности применения. Для удовлетворения таких требований в жизненном цикле ПС необходимы выделение задач и работ по обеспечению надежности программ и концентрация усилий разработчиков на теоретическом и практическом их решении. Для каждого проекта ПС, выполняющего ответственные функции, должны разрабатываться и применяться специальные план и программа, методология и инструментальные средства, обеспечивающие требуемые надежность и безопасность. Только скоординированное, комплексное применение в проектах ПС современных методов и средств обеспечения надежности функционирования и безопасности применения комплексов программ путем автоматизации их разработки и испытаний позволяет достигать высокого качества ПС, необходимого для их применения в критических и сложных системах управления и обработки информации.

7.3 Особенности применения основных понятий теории надежности сложных систем к жизненному циклу и оценке качества программного обеспечения

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.

Надежность определяется как уровень, при котором система программ удовлетворяет поставленным требованиям и пригодна для эксплуатации. При этом следует отличать надежность от *корректности,* которая определяется как степень удовлетворения требованиям. Надежность является составной частью более общего понятия – *качества.* Качественная программа не только надежна, но и компактна, совместима с другими программами, эффективна, удобна в сопровождении, портативна и вполне понятна.

Свойства надежности изделий изучаются *теорией надежности,* которая является системой определенных идей, математических моделей и методов, направленных на решение проблем предсказания, оценки и оптимизации различных показателей надежности. Надежность технических систем определяется в основном двумя факторами: *надежностью компонентов и дефектами в конструкции,* допущенными при проектировании или изготовлении. Относительно невысокая физическая надежность компонентов, их способность к разрушению, старению или снижению надежности в процессе эксплуатации привели к тому, что этот фактор оказался доминирующим для большинства комплексов аппаратуры. Этому способствовала также невысокая сложность многих технических систем, вследствие чего дефекты проектирования проявлялись относительно редко.

Надежность сложных программных средств определяется этими же факторами, однако доминирующими являются дефекты и ошибки проектирования, так как физическое хранение программ на магнитных носителях характеризуется очень высокой надежностью. Программа любой сложности и назначения при строго фиксированных исходных данных и абсолютно надежной аппаратуре исполняется по однозначно определенному маршруту и дает на выходе строго определенный результат. Однако случайное изменение исходных данных и накопленной при обработке информации, а также множество условных переходов в программе создают огромное число различных маршрутов исполнения каждого сложного ПС. Источниками ненадежности являются непроверенные сочетания исходных данных, при которых функционирующее ПС дает неверные результаты или отказы. В результате комплекс программ не соответствует требова-

ниям функциональной пригодности и работоспособности. В учебном пособии Благодатских, В.А. др. «Стандартизация разработки программных средств» приведен пример, который опубликован в статье Юрия Батурина в журнале «Новое время» [1, стр. 36]. Автор статьи приводит несколько факторов, которые описывают проблемы функционирования сложных программных средств. Так, в качестве одного из факторов выступает фактор сложности. «Существуют фундаментальные причины, почему программное обеспечение невозможно сделать достаточно надежным, чтобы можно было не сомневаться в том, что система «звездных войн» действительно сработает», – считает Д. Парнас, крупнейший авторитет по крупномасштабному программированию. Он был назначен Организацией по осуществлению Стратегической Оборонной Инициативы (СОИ) членом консультативного комитета по программированию управления боевыми операциями. «Мне обещали 1000 долларов в день плюс накладные расходы». Но, ознакомившись подробнее с тем, чего от него ждут, Д. Парнас отклонил сделанное ему предложение, одновременно представив восемь технических документов, которые объясняли, почему программа не сможет работать так, как требуется. В качестве примера приведем еще один из факторов – фактор надежности. О том, насколько уязвимо математическое обеспечение, можно судить по следующему примеру. Когда в 1979 г. американский космический зонд, запущенный на Венеру, не достиг своей цели, в космос вылетело почти полмиллиарда долларов. Причина в том, что в программе коррекции курса зонда **запятая была спутана с двоеточием**.

При применении понятий надежности к программным средствам следует учитывать *особенности и отличия этих объектов от традиционных технических систем,* для которых первоначально разрабатывалась теория надежности:

– не для всех видов программ применимы понятия и методы теории надежности;

– их можно использовать только к программным средствам, функционирующим в реальном времени и непосредственно взаимодействующим с внешней средой;

– при оценке качества программных компонентов к ним неприменимы понятия надежности функционирования, если при обработке информации они не используют значения реального времени и не взаимодействуют непосредственно с внешней средой;

– доминирующими факторами, определяющими надежность программ, являются дефекты и ошибки проектирования и разработки, и второстепенное значение имеет физическое разрушение программных компонентов при внешних воздействиях; относительно редкое разрушение программных компонентов и необходимость их физической замены приводят к принципиальному изменению понятий сбоя и отказа программ и к разделению их по длительности восстановления относительно некоторого допустимого времени простоя для функционирования информационной системы;

– для повышения надежности комплексов программ особое значение имеют

методы автоматического сокращения длительности восстановления и преобразования отказов в кратковременные сбои путем введения в программные средства временной, программной и информационной избыточности;
– непредсказуемость места, времени и вероятности проявления дефектов и ошибок, а также их редкое обнаружение при реальной эксплуатации достаточно надежных программных средств, не позволяют эффективно использовать традиционные методы априорного расчета показателей надежности сложных систем, ориентированные на стабильные, измеряемые значения надежности составляющих компонентов;
– традиционные методы форсированных испытаний надежности систем путем физического воздействия на их компоненты неприменимы для программных средств, и их следует заменять на методы форсированного воздействия информационных потоков внешней среды.

С учетом перечисленных особенностей применение основных понятий теории надежности сложных систем к жизненному циклу и оценке качества комплексов программ позволяет адаптировать и развивать эту теорию в особом направлении – *надежности программных средств.* 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 экспериментальных 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.

7.4 Показатели качества и надежности программных средств

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 (рисунок 7.1).

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 перезапускаемостъю.

Рисунок 7.1 – – Схема характеристик качества программных средств по стандарту ISO 9126:1991

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.

Близким к описанному стандарту по идеологии, структуре и содержанию является ГОСТ 28195–89. 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.

Функциональная пригодность – это набор атрибутов, определяющий назначение, номенклатуру, основные необходимые и достаточные функции ПС, заданные техническим заданием заказчика или потенциального пользователя. В процессе проектирования ПС атрибуты функциональной пригодности конкретизируются в спецификации на компоненты. Эти атрибуты можно численно представить точностью вычислений, относительным числом поэтапно изменяемых функций, числом спецификаций требований заказчиков и т.д. Кроме них функциональную пригодность отражает множество различных специализированных критериев, которые тесно связаны с конкретными функциями программ. Их можно рассматривать как частные критерии или как факторы, влияющие на основные показатели. В наиболее общем виде функциональная пригодность проявляется в *корректности и надежности* ПС.

Понятие *корректной (правильной) программы* может рассматриваться статически вне ее исполнения во времени. Корректность программы не определена вне области изменения исходных данных, заданных требованиями спецификации, и не зависит от динамики функционирования программы в реальном времени. Степень некорректности программ определяется вероятностью попадания реальных исходных данных в область, которая задана требованиями спецификации и технического задания (ТЗ), однако не была проверена при тестировании и испытаниях. Значения этого показателя зависят от функциональной корректности применяемых компонентов и могут рассматриваться в зависимости от методов их достижения и оценивания: детерминировано, стохастически и в реальном времени. При анализе видов корректности и способов их измерения, есте-

ственно, они связываются с водами и методами процесса тестирования и испытания программ.

Надежная программа, прежде всего, должна обеспечивать достаточно низкую вероятность отказа в процессе функционирования в реальном времени. Быстрое реагирование на искажения программ, данных или вычислительного процесса и восстановление работоспособности за время, меньшее, чем порог между сбоем и отказом, обеспечивают высокую надежность программ. При этом некорректная программа может функционировать абсолютно надежно. В реальных условиях по различным причинам исходные данные могут попадать в области значений, вызывающих сбои, не проверенные при испытаниях, а также не заданные требованиями спецификации и технического задания. Если в этих ситуациях происходит достаточно быстрое восстановление, такое, что не фиксируется отказ, то такие события не влияют на основные показатели надежности – наработку на отказ и коэффициент готовности. Следовательно, надежность функционирования программ является понятием динамическим, проявляющимся во времени, и существенно отличается от понятия корректности программ.

При оценке качества программ по показателям надежности регистрируются только такие искажения в процессе динамического тестирования с исполнением программ, которые приводят к потере работоспособности ПС или их крупных компонентов. Первопричиной нарушения работоспособности программ при безотказности аппаратуры всегда является конфликт между реальными исходными данными, подлежащими обработке, и программой, осуществляющей эту обработку. Работоспособность ПС можно гарантировать при конкретных исходных данных, которые использовались при отладке и испытаниях. Реальные исходные данные могут иметь значения, отличающиеся от заданных техническим заданием и от использованных при применении программ. При таких исходных данных функционирование программ трудно предсказать заранее и весьма вероятны различные аномалии, завершающиеся отказами.

Непредсказуемость вида, места и времени проявления дефектов ПС в процессе эксплуатации приводит к необходимости создания специальных, дополнительных систем оперативной защиты от непредумышленных, случайных искажений вычислительного процесса, программ и данных. Системы оперативной защиты предназначены для выявления и блокирования распространения негативных последствий проявления дефектов и уменьшения их влияния на надежность функционирования ПС до устранения их первичных источников. Для этого в ПС должна вводиться временная, программная и информационная избыточность, осуществляющая оперативное обнаружение дефектов функционирования, их идентификацию и автоматическое восстановление (рестарт) нормального функционирования ПС. Надежность ПС должна повышаться за счет средств обеспечения помехоустойчивости, оперативного контроля и восстановления функционирования программ и баз данных. Эффективность такой

защиты зависит от используемых методов, координированности их применения и выделяемых вычислительных ресурсов на их реализацию.

Основным *принципом классификации сбоев и отказов* в программах при отсутствии их физического разрушения является разделение по временному показателю длительности восстановления после любого искажения программ, данных или вычислительного процесса, регистрируемого как нарушение работоспособности. При длительности восстановления, меньшей заданного порога, дефекты и аномалии при функционировании программ следует относить к *сбоям,* а при восстановлении, превышающем по длительности пороговое значение, происходящее искажение соответствует *отказу.* Классификация программных сбоев и отказов по длительности восстановления приводит к необходимости анализа динамических характеристик абонентов, являющихся потребителями данных, обработанных исследуемым ПС, а также временных характеристик функционирования программ. Временная зона перерыва нормальной выдачи информации и потери работоспособности, которую следует рассматривать как зону сбоя, тем шире, чем более инертный объект находится под воздействием сообщений, подготовленных данным ПС. Пороговое время восстановления работоспособного состояния системы, при превышении которого следует фиксировать отказ, близко к периоду решения задач для подготовки информации соответствующему абоненту.

При нормальном темпе решения задач и выдаче их результатов потребителю отклонения его характеристик от траектории, рассчитываемой ПС, находятся в допустимых пределах. Для любого потребителя информации существует допустимое время отсутствия данных от ПС, при котором его характеристики, изменяясь по инерции, достигают предельного отклонения от значения, которое должно быть рассчитано программами. Соответствующая этому отклонению временная зона перерыва выдачи информации потребителю позволяет установить границу допустимой длительности нарушения работоспособности, которая разделяет зоны сбоев и отказов.

Чем более инерционным является потребитель информации, тем больше может быть время отсутствия у него результатов функционирования и воздействий от ПС без катастрофических последствий нарушения работоспособности, соответствующего отказу. Это допустимое отклонение результатов после перерыва функционирования ПС зависит в основном от динамических характеристик источников и потребителей информации. Таким образом, установив в результате системного анализа динамических характеристик объектов информационной системы величину порогового значения, можно определить интервал времени функционирования ПС при отсутствии выдачи потребителю данных, которые разделяют события сбоя и отказа без физического разрушения программ.

Надежность функционирования ПС наиболее широко характеризуется *устойчивостью,* или способностью к безотказному функционированию, 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,* независимо 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.

Тема 8 Дестабилизирующие факторы и методы обеспечения надежности функционирования программных средств

8.1 Модель факторов, определяющих надежность программных средств.

8.2 Методы обеспечения надежности программных средств.

8.3 Систематизация принципов и методов обеспечения надежности в соответствии с их целью

8.4 Обработка сбоев аппаратуры.

8.1 Модель факторов, определяющих надежность 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 не умышленное 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.

При строго фиксированных исходных данных программы исполняются по определенным маршрутам и выдают совершенно определенные результаты. Многочисленные варианты исполнения программ при разнообразных исходных данных представляются для внешнего наблюдателя как случайные. В связи с

этим дефекты функционирования программных средств, не имеющие злоумышленных источников, проявляются внешне как случайные, имеют разную природу и последствия. В частности, они могут приводить к последствиям, соответствующим нарушениям работоспособности, и к отказам при использовании ПС.

Последующий анализ надежности ПС базируется на модели взаимодействия основных компонентов, представленных на рисунке 8.1. *Объектами уязвимости,* влияющими на надежность ПС, являются:

ГГУ ИМЕНИ Ф. СКОРИНЫ

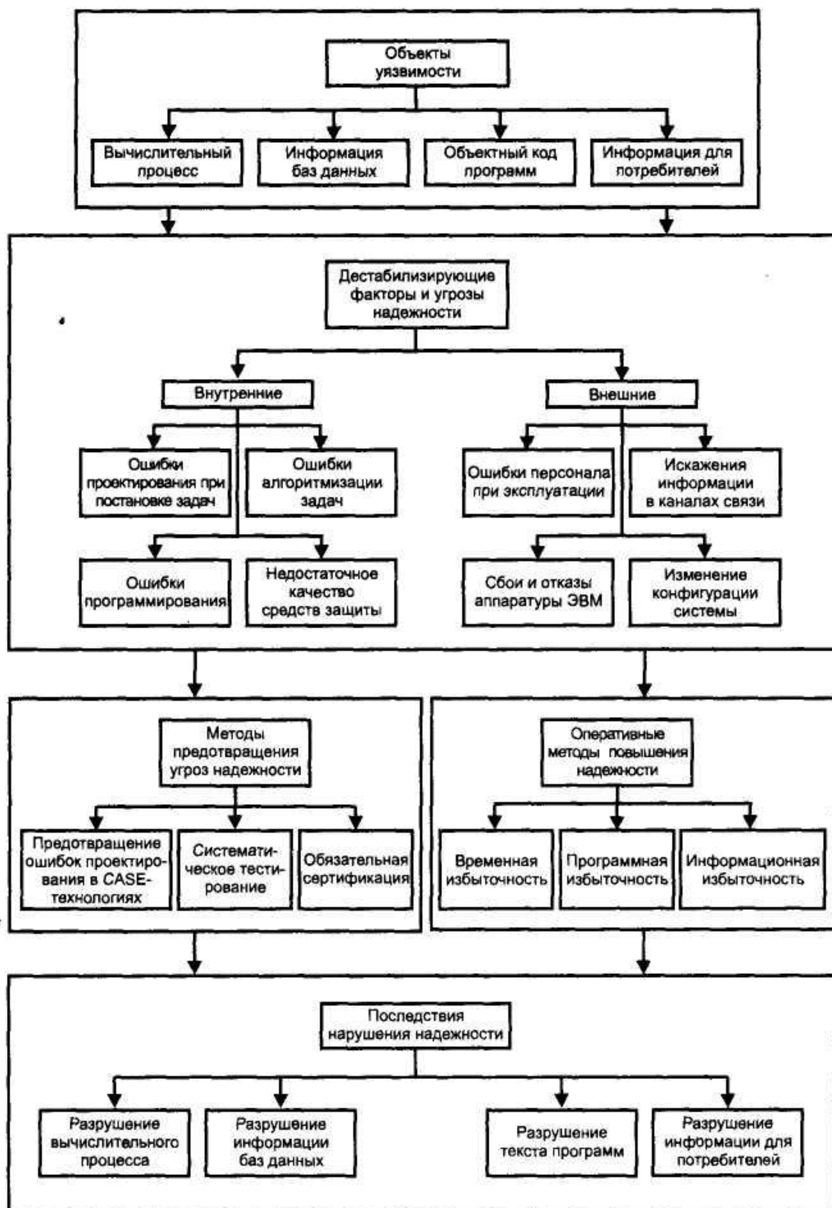

Рисунок 8.1 – Схема модели анализа надежности программных средств

– 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.

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.

Современные достижения микроэлектроники значительно снизили влияние сбоев и отказов вычислительных средств на надежность функционирования ПС. Однако ошибки персонала, искажения данных в каналах телекоммуникации, а также случайные (при отказах части аппаратуры) и необходимые изменения конфигурации вычислительных средств остаются существенными внешними угрозами надежности ПС. Негативное влияние этих факторов может быть значительно снижено соответствующими методами и средствами защиты и восстановления программ и данных. Внешние дестабилизирующие факторы имеют различную природу и широкий спектр характеристик, которые представлены во многих публикациях. Поэтому ниже внимание акцентируется на внутренних дестабилизирующих факторах, различного рода дефектах и ошибках проектирования и эксплуатации, которые оказывают наибольшее влияние на надежность функционирования ПС.

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 обеспечением 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.

8.2 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.

8.3 Систематизация принципов и методов обеспечения надежности в соответствии с их целью

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.

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:

1 методы, позволяющие справиться со сложностью, свести ее к минимуму, так как это – главная причина ошибок перевода;

2 методы достижения большей точности при переводе;

3 методы улучшения обмена информацией;

4 методы немедленного обнаружения и устранения ошибок. Эти методы направлены на обнаружение ошибок на каждом шаге перевода, не откладывая до тестирования программы после ее написания.

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.

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 программной 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.

1 *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.

2 *Немедленное обнаружение.* Ошибки необходимо обнаружить как можно раньше. Это не только ограничивает наносимый ими ущерб, но и значительно упрощает задачу отладки.

3 *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. С точки зрения предоставления человеку, отлаживающему программу, например системному программисту, самых благоприятных условий для диагностики ошибок немедленное завершение представляется наилучшей стратегией. Конечно, во многих системах 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.

Всегда, когда это возможно, лучше приостановить выполнение программы, чем регистрировать ошибки (либо обеспечить как дополнительную возможность работу системы в любом из этих режимов). Различие между этими методами проиллюстрируем на способах выявления причин возникающего иногда скрежета вашего автомобиля. Если автомеханик находится на заднем сиденье, то он может обследовать состояние машины в тот момент, когда скрежет возникает. Если вы выбираете метод регистрации ошибок, задача диагностики станет сложнее.

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 называются *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 ре-

сурсов 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.

Диагностический монитор можно реализовать как периодически выполняемую задачу (например, она планируется на каждый час) либо как задачу с низким приоритетом, которая планируется для выполнения в то время, когда система переходит в состояние ожидания. Как и прежде, выполняемые монитором конкретные проверки зависят от специфики системы, но некоторые идеи будут понятны из примеров. Монитор может обследовать основную память, чтобы обнаружить блоки памяти, не выделенные ни одной из выполняемых задач и не включенные в системный список свободной памяти. Он может проверять также необычные ситуации: например, процесс не планировался для выполнения в течение некоторого разумного интервала времени. Монитор может осуществлять поиск «затерявшихся» внутри системы сообщений или операций ввода–вывода, которые необычно долгое время остаются незавершенными, участков памяти на диске, которые не помечены как выделенные и не включены в список свободной памяти, а также различного рода странностей в файлах данных.

Иногда желательно, чтобы в чрезвычайных обстоятельствах монитор выполнял диагностические тесты системы. Он может вызывать определенные системные функции, сравнивая их результат с заранее определенным и проверяя, насколько разумно время выполнения. Монитор может также периодически предъявлять системе «пустые» или «легкие» задания, чтобы убедиться, что система функционирует хотя бы самым примитивным образом.

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.

Устойчивость к ошибкам

Методы этой группы ставят своей целью обеспечить функционирование программной системы при наличии в ней ошибок. Они разбиваются 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. Аналогичные средства часто необходимы в операционных системах. Если операционная система обнаруживает, что вот–вот выйдет из строя, она может загрузить аварийный фрагмент, ответственный за оповещение пользователей у терминалов о предстоящем сбое и за сохранение всех критических для системы данных.

Последняя подгруппа – *методы изоляции ошибок.* Основная их идея – не дать как можно большей части последствиям ошибки выйти за пределы 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 изолируются 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. Хотя в формулировке многих из них употребляются слова «операционная система», они применимы к любой программе (будь то операционная система, монитор телеобработки или подсистема управления файлами), которая занята обслуживанием других программ.

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. 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.
6 Никакие системные данные, непосредственно доступные прикладным программам, не должны влиять на функционирование операционной системы. Ошибка в прикладной программе, вследствие которой содержимое этой памяти может быть случайно изменено, приводит в конце концов к сбою системы.
7 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.
8 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.
9 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.
10 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.

Из рассмотренных выше трех подгрупп методов обеспечения устойчивости к ошибкам только третья, изоляция ошибок, применима *для* большинства систем программного обеспечения.

Важное обстоятельство, касающееся всех четырех подходов, состоит в том, что обнаружение, исправление ошибок и устойчивость к ошибкам в некотором отношении противоположны методам предупреждения ошибок. В частности, обнаружение, исправление и устойчивость требуют дополнительных функций от самого программного обеспечения. Тем самым не только увеличивается сложность готовой системы, но и появляется возможность внести новые ошибки при реализации этих функций. Как правило, все рассматриваемые методы предупреждения и многие методы обнаружения ошибок применимы к любому программному проекту. Методы исправления ошибок и обеспечения устойчивости применяются не очень широко. Это, однако, зависит от области приложения. Ес-

ли рассматривается, скажем, система реального времени, то ясно, что она должна сохранить работоспособность и при наличии ошибок, а тогда могут оказаться желательными и методы исправления и обеспечения устойчивости. К системам такого типа относятся телефонные переключательные системы, системы управления технологическими процессами, аэрокосмические и авиационные диспетчерские системы и операционные системы широкого назначения.

8.4 Обработка сбоев аппаратуры

Улучшая общую надежность системы, следует заботиться не только об ошибках в программном обеспечении (хотя надежность программного обеспечения требует наибольшего внимания). Другая сторона, о которой необходимо подумать, – это ошибки во входных данных системы (ошибки пользователя).

Наконец, еще один интересующий нас класс ошибок – сбои аппаратуры. Во многих случаях они обрабатываются самой аппаратурой за счет использования кодов, исправляющих ошибки, исправления последствий сбоев (например, переключением на запасные компоненты) и средств, обеспечивающих устойчивость к ошибкам (например, голосование). Некоторые сбои, однако, нельзя обработать только аппаратными средствами, они требуют помощи со стороны программного обеспечения. Ниже приводится список возможностей, которые часто бывают необходимы в программных системах для борьбы со сбоями аппаратуры.

1 *Повторное выполнение операций.* Многие сбои аппаратуры не постоянны (например, скачки напряжения, шум в телекоммуникационных линиях, колебания при механическом движении). Всегда имеет смысл попытаться выполнить операцию, искаженную сбоем (например, команду машины или операцию ввода–вывода), несколько раз, прежде чем принимать другие меры.

2 *Восстановление памяти.* Если обнаруженный случайный сбой аппаратуры вызывает искажение области основной памяти и эта область содержит статические данные (например, команды объектной программы), то последствия сбоя можно ликвидировать, повторно загрузив эту область памяти.

3 *Динамическое изменение конфигурации.* Если аппаратная подсистема, такая, как процессор, канал ввода–вывода, блок основной памяти или устройство ввода–вывода, выходит из строя, работоспособность системы можно сохранить, динамически исключая неисправное устройство из набора ресурсов системы.

4 *Восстановление файлов.* Системы управления базами данных обычно обеспечивают избыточность данных, сохраняя копию текущего состояния базы данных на выделенных устройствах ввода–вывода, регистрируя все изменения базы данных или периодически автономно копируя всю базу данных. Поэтому программы восстановления могут воссоздать базу данных в случае катастрофического сбоя ввода–вывода.

5 *Контрольная точка рестарт.* Контрольная точка – это периодически обновляемая копия состояния прикладной программы или всей системы. Если происходит отказ аппаратуры, такой, как ошибка ввода–вывода, сбой памяти или питания, программа может быть запущена повторно с последней контрольной точки.
6 *Предупреждение отказов питания.* Некоторые вычислительные системы, особенно те, в которых используется энергозависимая память, предусматривают прерывание, предупреждающее программу о предстоящем отказе питания. Это дает возможность организовать контрольную точку или перенести жизненно важные данные во вторичную память.

Регистрация ошибок. Все сбои аппаратуры, с которыми удалось справиться, должны регистрироваться во внешнем файле, чтобы обслуживающий персонал мог получать сведения о постепенном износе устройств.

Тема 9 Модели надежности программного обеспечения

9.1 Классификация моделей надежности программного обеспечения
9.2 Аналитические модели надежности.
9.3 Эмпирические модели надежности.
9.4 Сертификация комплексов программ.

9.1 Классификация моделей 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. Данный термин может быть использован также применительно к математической зависимости между определенными параметрами, которые хотя и имеют отношение к оценке надежности программного обеспечения, но тем не менее не содержат ее характеристик в явном виде. Например, поведение некоторой ветви программы на подмножестве наборов входных данных, с помощью которых эта ветвь контролируется, существенным образом связано с надежностью программы, однако характеристики этого поведения могут быть оценены независимо от оценки самой надежности. Другим таким параметром является частота ошибок, которая позволяет оценить именно качество систем реального времени, функционирующих в непрерывном режиме, и в то же время получать только косвенную информацию относительно надежности программного обеспечения (например, в предположении экспоненциального распределения времени между отказами).

Одним из видов модели надежности программного обеспечения, которая заслуживает особого внимания, является так называемая *феноменологическая,* или *эмпирическая,* модель. При разработке моделей такого типа предполагается, что связь между надежностью и другими параметрами является статической. С помощью подобного подхода пытаются количественно оценить те характеристики программного обеспечения, которые свидетельствуют либо о высокой, либо о низкой его надежности. Так, например, параметр *сложность программы* характеризует степень уменьшения уровня ее надежности, поскольку усложнение программы всегда приводит к нежелательным последствиям, в том числе к неизбежным ошибкам программистов при составлении программ и трудности их обнаружения и устранения. Иначе говоря, при разработке феноменологической модели надежности программного обеспечения стремятся иметь дело с такими параметрами, соответствующее изменение значений которых должно приводить к повышению надежности программного обеспечения.

Рассмотрим классификацию моделей надежности ПС, приведенную на рисунке 9.1. 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.

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.

Рисунок 9.1 – Классификационная схема моделей надежности ПС

Аналитическое моделирование надежности ПС включает четыре шага:

1 определение предположений, связанных с процедурой тестирования ПС;

2 разработка или выбор аналитической модели, базирующейся на предположениях о процедуре тестирования;

3 выбор параметров моделей с использованием полученных данных;

4 применение модели – расчет количественных показателей надежности по модели.

9.2 Аналитические модели надежности

Динамические модели надежности

Модель 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 предпо-

лагает, 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 τ обнаруживается ε_c ошибок в расчете на команду в машинном языке.

Таким образом, удельное число ошибок на одну машинную команду, оставшихся в системе после времени тестирования τ, равно:

$$\varepsilon_r(\tau) = \frac{E_T}{I_T} \cdot \varepsilon_c(\tau), \qquad (1)$$

где I_T – 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:

$$Z(t) = C\varepsilon_r(\tau),$$

где C – некоторая константа; t – время работы ПС без отказа.

Тогда, если время работы ПС без отказа t отсчитывается от точки t = 0, а τ остается фиксированным; то функция надежности, или вероятность безотказной работы на интервале времени от 0 до t, равна:

$$R(t,\tau) = \exp\{C[E_T / I_T - \varepsilon_c(\tau)]t\}; \qquad (2)$$

$$t_{\text{ср}} = \frac{1}{C[E_T / I_T - \varepsilon_c(\tau)]}. \qquad (3)$$

ГГУ ИМЕНИ Ф. СКОРИНЫ

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:

$$\tau = \tau_1 + \tau_2 + \tau_3 + \ldots + \tau_n.$$

Предполагая, что интенсивность появления ошибок постоянна и равна l, можно вычислить ее как число ошибок в единицу времени:

$$\lambda = \frac{\sum_{i=1}^{k} A_i}{\tau}, \qquad (4)$$

где A_i – количество ошибок на i–м прогоне;

$$t_{\text{ср}} = \frac{\tau}{\sum_{i=1}^{k} A_i}. \qquad (5)$$

Имея данные для двух различных моментов тестирования τ_A и τ_B, которые выбираются произвольно с учетом требования, чтобы $\varepsilon_\tau(\iota_B) > \varepsilon_\tau(\iota_A)$, можно сопоставить уравнения (3) и (5) при τ_A и τ_B.

$$\frac{1}{\lambda_{\tau A}} = \frac{1}{C\left[E_T / I_T - \varepsilon_c(\tau_A)\right]}; \qquad (6)$$

$$\frac{1}{\lambda_{\tau b}} = \frac{1}{C\left[E_T / I_T - \varepsilon_c(\tau_b)\right]}. \qquad (7)$$

Вычисляя отношения (6) и (7), получим:

$$E_T = \frac{I_T\left[\lambda\tau_b / \lambda\tau_A \varepsilon_c(\tau_A) - \varepsilon_c(\tau_b)\right]}{(\lambda\tau_b / \lambda\tau_A) - 1} \qquad (8)$$

Подставив полученную оценку параметров E_T в выражение (6), получим оценку для второго неизвестного параметра:

$$C = \frac{\lambda\tau_A}{\left[E_T / I_T - \varepsilon_c(\tau_A)\right]}. \qquad (9)$$

Получив неизвестные E_T и C, можно рассчитать надежность программы по формуле (2).

ГГУ ИМ. Ф. СКОРИНЫ

Модель La Padula. По этой модели выполнение последовательности тестов производится в m этапов. Каждый этап заканчивается внесением изменений (исправлений) в ПС. Возрастающая функция надежности базируется на числе ошибок, обнаруженных в ходе каждого тестового прогона.

Надежность ПС в течение i–го этапа:

$$R(t) = R(\infty) — A / (i),\ i = 1,2\ldots,$$

где A – параметр роста;

$R(\infty) = \lim\limits_{i \to \infty} R(i)$ –предельная надежность ПС.

Эти неизвестные величины автор предлагает вычислить, решив следующие уравнения:

$$\sum_{i=1}^{m} \left\{ \frac{S_i - m_i}{S_i} - R(\infty) + \frac{A}{i} \right\} = 0;$$

$$\sum_{i=1}^{m} \left[\left(\frac{S_i - m_i}{S_i} - R(\infty) + \frac{A}{i} \right) \left(\frac{1}{i} \right) \right] = 0,$$

где S_i – число тестов;

m_i, – число отказов во время i–го этапа; i= 1,2, ..., m;

m – число этапов;.

Определяемый по этой модели показатель есть надежность ПС на i–м этапе: $R(t) = R(\infty) – A/(i)$, $i = m + 1, m + 2...$

Преимущество модели заключается в том, что она является прогнозной и, основываясь на данных, полученных в ходе тестирования, дает возможность предсказать вероятность безотказной работы программы на последующих этапах ее выполнения.

Модель Джелинского – Моранды. Модель Джелинского – Моранды относится к динамическим моделям непрерывного времени. Исходные данные для использования этой модели собираются в процессе тестирования ПС. При этом фиксируется время до очередного отказа. Основное положение, на котором базируется модель, заключается в том, что значение интервалов времени тестирования между обнаружением двух ошибок имеет экспоненциальное распределение с частотой ошибок (или интенсивностью отказов), пропорциональной числу еще не выявленных ошибок. Каждая обнаруженная ошибка устраняется, число оставшихся ошибок уменьшается на единицу.

Функция плотности распределения времени обнаружения i–й ошибки, отсчитываемого от момента выявления (i–1)–й ошибки, имеет вид:

$$P(t_i) = \lambda_i e^{-\lambda_i t_i}, \tag{10}$$

ГГУ ИМЕНИ Ф. СКОРИНЫ

где λ_i – частота отказов (интенсивность отказов), которая пропорциональна числу еще не выявленных ошибок в программе.

$$\lambda_i = C(N - i + 1), \tag{11}$$

где N – число ошибок, первоначально присутствующих в программе;

С – коэффициент пропорциональности.

Наиболее вероятные значения величин $\hat{N}$ и $\hat{C}$ (оценка максимального правдоподобия) можно определить на основе данных, полученных при тестировании. Для этого фиксируют время выполнения программы до очередного отказа $t_1, t_2, t_3, \ldots, t_k$.

Значения $\hat{N}$ и $\hat{C}$ предлагается получить, решив систему уравнений:

$$\sum_{i=1}^{k} (\hat{N} - i + 1)^{-1} = K / (\hat{N} + 1 - QK),$$
$$\hat{C} = \frac{K / A}{\hat{N} + 1 - QK}, \tag{12}$$

где

$$Q = B / AK; \;\; A = \sum_{i=1}^{k} t_i; \;\; B = \sum_{i=1}^{k} i \cdot t_i.$$

Поскольку полученные значения $\hat{N}$ и $\hat{C}$–вероятностные и точность их зависит от количества интервалов тестирования (или количества ошибок), найденных к моменту оценки надежности, асимптотические оценки дисперсий авторы предлагают определить с помощью следующих формул:

$$Var(\hat{N}) = K / C^2 D,$$

$$Var(\hat{C}) = S / D,$$

где

$$D = KS / C^2 - A^2 \text{ и } S = \sum_{i=1}^{k} (N - i + 1)^2.$$

Чтобы получить числовые значения λ_i, нужно подставить вместо N и C их возможные значения $\hat{N}$ и $\hat{C}$. Рассчитав K значений по формуле (11) и подставив их в формулу (10), можно определить вероятность безотказной работы на различных временных интервалах. На основе полученных расчетных данных строится график зависимости вероятности безотказной работы от времени.

Модель Шика – Волвертона. Модификация модели Джелинского – Моранды для случая возникновения на рассматриваемом интервале более одной ошибки предложена Волвертоном и Шиком. При этом считается, что исправление ошибок производится лишь после истечения интервала времени, на котором они возникли. В основе модели Шика – Волвертона лежит предположение, согласно которому частота ошибок пропорциональна не только количеству ошибок в программах, но и времени тестирования, т.е. вероятность обнаружения ошибок с течением времени возрастает. Частота ошибок (интенсивность обнаружения ошибок) λ_i, предполагается постоянной в течение интервала времени t_i и пропорциональна числу ошибок, оставшихся в программе по истечении (i–1)–го интервала; но она пропорциональна также и суммарному времени, уже затраченному на тестирование (включая среднее время выполнения программы в текущем интервале):

$$\lambda_i = C(N - n_{i-1})(T_{i-1} + t_i / 2). \quad (13)$$

В данной модели наблюдаемым событием является число ошибок, обнаруживаемых в заданном временном интервале, а не время ожидания каждой ошибки, как это было для модели Джелинского – Моранды. В связи с этим модель относят к группе дискретных динамических моделей.

Модель Муса. Модель Муса относят к динамическим моделям непрерывного времени. Это значит, что в процессе тестирования фиксируется время выполнения программы (тестового прогона) до очередного отказа. Но считается, что не всякая ошибка ПС может вызвать отказ, поэтому допускается обнаружение более одной ошибки при выполнении программы до возникновения очередного отказа.

Считается, что на протяжении всего жизненного цикла ПС может произойти M_0 отказов и при этом будут выявлены все N_0 ошибки, которые присутствовали в ПС до начала тестирования.

Общее число отказов M_0 связано с первоначальным числом ошибок N_0 соотношением

$$N_0 = BM_0, \quad (14)$$

где B – коэффициент уменьшения числа ошибок.

В момент, когда проводится оценка надежности, после тестирования, на которое потрачено определенное время t, зафиксировано m отказов и выявлено n ошибок.

Тогда из соотношения

$$n = Bm \quad (15)$$

можно определить коэффициент уменьшения числа ошибок B как число, характеризующее количество устраненных ошибок, приходящихся на один отказ.

В модели Муса различают два вида времени:

ПГУ ИМЕНИ Ф. СКОРИНЫ

1) суммарное время функционирования τ , которое учитывает чистое время тестирования до контрольного момента, когда проводится оценка надежности;

2) оперативное время t – время выполнения программы, планируемое от контрольного момента и далее при условии, что дальнейшего устранения ошибок не будет (время безотказной работы в процессе эксплуатации).

Для суммарного времени функционирования τ предполагается:

– интенсивность отказов пропорциональна числу неустраненных ошибок;
– скорость изменения числа устраненных ошибок, измеряемая относительно суммарного времени функционирования, пропорциональна интенсивности отказов.

Один из основных показателей надежности, который рассчитывается по модели Муса, – средняя наработка на отказ. Этот показатель определяется как математическое ожидание временного интервала между последовательными отказами и связан с надежностью:

$$T = \int_0^\infty tf(t)dt = \int_0^\infty R(t)dt,$$

где t – время работы до отказа.

Если интенсивность отказов постоянна (т.е. когда длительность интервалов между последовательными отказами имеет экспоненциальное распределение), то средняя наработка на отказ обратно пропорциональна интенсивности отказов.

Модель переходных вероятностей. Эта модель основана на марковском процессе, протекающем в дискретной системе с непрерывным временем.

Процесс, протекающий в системе, называется марковским (или процессом без последствий), если для каждого момента времени вероятность любого состояния системы в будущем зависит только от состояния системы в настоящее время (t_o) и не зависит от того, каким образом система пришла в это состояние. Процесс тестирования ПС рассматривается как марковский процесс.

В начальный момент тестирования (t = 0) в ПС было n ошибок. Предполагается, что в процессе тестирования выявляется по одной ошибке. Тогда последовательность состояний системы ($n, n–1, n–2, n–3$) и т.д. соответствует периодам времени, когда предыдущая ошибка уже исправлена, а новая еще не обнаружена. Например, в состоянии $n–5$ пятая ошибка уже исправлена, а шестая еще не обнаружена.

Последовательность состояний {m, m – 1, m – 2, m – 3 и т.д.} соответствует периодам времени, когда ошибки исправляются. Например, в состоянии m – 1 вторая ошибка уже обнаружена, но еще не исправлена. Ошибки обнаруживаются с интенсивностью X, а исправляются с интенсивностью μ .

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 $\lambda(t)$. Эти модели строятся на твердом статистическом фундаменте.

Модель Миллса. Использование этой модели предполагает необходимость перед началом тестирования искусственно вносить в программу («засорять») некоторое количество известных ошибок. Ошибки вносятся случайным 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

$$N = \frac{S \cdot n}{V} \tag{16}$$

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:

$$C = \begin{cases} 1, \text{если } n > K; \\ \dfrac{S}{S + K + 1}, \text{если } n =< K. \end{cases} \tag{17}$$

Таким образом, величина С 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.

Модель Липова. Липов модифицировал модель Миллса, рассмотрев вероятность обнаружения ошибки при использовании различного числа тестов. Если

сделать то же предположение, что и в модели Миллса, т.е. что собственные и искусственные ошибки имеют равную вероятность быть найденными, то вероятность обнаружения n собственных и V внесенных ошибок равна:

$$Q(n,V)=(\frac{m}{n+V})q^{n+V}(1-q)^{m-n-V}\frac{\frac{N}{n}\cdot\frac{S}{V}}{\frac{N+S}{n+V}},$$

где m – количество тестов, используемых при тестировании;
q – вероятность обнаружения ошибки в каждом из m тестов, рассчитанная по формуле

$$q=\frac{n+V}{n};$$

S – общее количество искусственно внесенных ошибок;
N – количество собственных ошибок, имеющихся в ПС до начала тестирования.

Для использования модели Липова должны выполняться следующие условия:

$$N\geq n\geq 0;\ S\geq V\geq 0;\ m\geq n+V\geq 0.$$

Оценки максимального правдоподобия (наиболее вероятное значение для N) задаются соотношениями

$$N=\frac{S\cdot n}{V}\ \text{при}\ n\geq 1,\quad V\geq 1;$$

$$n\cdot S\ \text{при}\ V=0,\quad 0\ \text{при}\ n=0.$$

Модель Липова дополняет модель Миллса, давая возможность оценить вероятность обнаружения определенного количества ошибок к моменту оценки.

Простая интуитивная модель. Использование этой модели предполагает проведение тестирования двумя группами программистов (или двумя программистами в зависимости от величины программы) независимо друг от друга, использующими независимые тестовые наборы. В процессе тестирования каждая из групп фиксирует все найденные ею ошибки. При оценке числа оставшихся в программе ошибок результаты тестирования обеих групп собираются и сравниваются.

Получается, что первая группа обнаружила N_1 ошибок, вторая – N_2, а N_{12} – это ошибки, обнаруженные обеими группами.

Если обозначить через N неизвестное количество ошибок, присутствовавших в программе до начала тестирования, то можно эффективность тестирования каждой из групп определить как

ГГУ ИМ. Ф. СКОРИНЫ

$$E_1 = \frac{N_1}{N};\ E_2 = \frac{N_2}{N}. \quad (18)$$

Предполагая, что возможность обнаружения всех ошибок одинакова для обеих групп, можно допустить, что если первая группа обнаружила определенное количество всех ошибок, она могла бы определить то же количество любого случайным образом выбранного подмножества. В частности, можно допустить:

$$E_1 = \frac{N_1}{N} = \frac{N_{12}}{N_2}. \quad (19)$$

Из формулы (18) $N_2 = E_2N$, подставив в (19), получим:

$$E_1 = \frac{N_{12}}{E_2 N} = \frac{N_1 N_2}{N_{12}}.$$

Модель Коркорэна. Модель Коркорэна относится к статическим моделям надежности ПС, так как в ней не используются параметры времени тестирования и учитывается только результат N испытаний, в которых выявлено N_i ошибок i–го типа. Модель использует изменяющиеся вероятности отказов для различных типов ошибок.

В отличие от двух рассмотренных выше статических моделей, по модели Коркорэна оценивается вероятность безотказного выполнения программы на момент оценки:

$$R = \frac{N_0}{N} + \sum_{i=1}^{K} Y_i (N_i - 1) / N,$$

где N_0 – число безотказных выполнений программы; N – общее число прогонов; K – априори известное число типов.

$$Y_i = \begin{cases} a_i, & \text{если } N_i > 0 \\ 0, & \text{если } N_i = 0 \end{cases}$$

a_i – вероятность выявления при тестировании ошибки i–го типа.

В этой модели вероятность a_i должна оцениваться на основе априорной информации или данных предшествующего периода функционирования однотипных программных средств.

Модель Нельсона. Данная модель при расчете надежности ПС учитывает вероятность выбора определенного тестового набора для очередного выполнения программы.

Предполагается, что область данных, необходимых для выполнения тестирования программного средства, разделяется на K взаимоисключающих подобластей Zi, $i = 1, 2, \ldots, к$. Пусть Pi – вероятность того, что набор данных Zi будет вы-

ГГУ ИМЕНИ Ф. СКОРИНЫ

бран для очередного выполнения программы. Предполагая, что к моменту оценки надежности было выполнено N_i прогонов программы на Z_i наборе данных и из них n_i количество прогонов закончилось отказом, надежность ПС в этом случае равна:

$$R = 1 - \sum_{i=1}^{k} \frac{(n_i)}{N_i} P_i. \quad (20)$$

На практике вероятность выбора очередного набора данных для прогона (P_i) определяется путем разбиения всего множества значений входных данных на подмножества и нахождения вероятностей того, что выбранный для очередного прогона набор данных будет принадлежать конкретному подмножеству. Определение этих вероятностей основано на эмпирической оценке вероятности появления тех или иных входов в реальных условиях функционирования.

9.3 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 ресурсов тестирования, уточнения плановых сроков завершения проекта и т.д.

Модель сложности. В учебном пособии Благодатских В.А. «Стандартизация разработки программных средств» указывается, что есть тесная взаимосвязь между сложностью и надежностью ПС [1, стр. 172]. 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:

1 отношение действительного числа дуг к максимально возможному числу дуг, получаемому искусственным соединением каждого узла с любым другим узлом дугой;

2 отношение числа узлов к числу дуг;

3 отношение числа петель к общему числу дуг.

Для сложных модулей и для больших многомодульных программ составляется имитационная модель, программа которой «засоряется» ошибками и тести-

руется 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.

Модель, определяющая время доводки программ. Эта модель используется для ПС, которые имеют иерархическую структуру, т.е. ПС как система может содержать подсистемы, которые состоят из компонентов, а те, в свою очередь, состоят из W модулей. Таким образом, ПС может иметь W различных уровней композиции. На любом уровне иерархии возможна взаимная зависимость между любыми парами объектов системы. Все взаимозависимости рассматриваются в терминах зависимости между парами модулей.

Анализ модульных связей строится на том, что каждая пара модулей имеет конечную (возможно, нулевую) вероятность, изменения в одном модуле вызовут изменения в другом модуле.

Данная модель позволяет на этапе тестирования, а точнее при тестовой сборке системы, определять возможное число необходимых исправлений и время, необходимое для доведения ПС до рабочего состояния.

Основываясь на описанной процедуре оценки общего числа изменений, требуемых для доводки ПС, можно построить две различные стратегии корректировки ошибок:

– фиксировать все ошибки в одном выбранном модуле и устранить все по-

бочные эффекты, вызванные изменениями этого модуля, отрабатывая таким образом последовательно все модули;

– фиксировать все ошибки нулевого порядка в каждом модуле, затем фиксировать все ошибки первого порядка и т.д.

Исследование этих стратегий доказывает, что время корректировки ошибок на каждом шаге тестирования определяется максимальным числом изменений, вносимых в ПС на этом шаге, а общее время – суммой максимальных времен на каждом шаге. Это подтверждает известный факт, что тестирование обычно является последовательным процессом и обладает значительными возможностями для параллельного исправления ошибок, что часто приводит к превышению затрачиваемых на него ресурсов над запланированными.

9.4 Сертификация комплексов программ

Для *удостоверения качества, надежности и безопасности применения* сложных, критических ИС используемые в них ПС следует подвергать *обязательной сертификации* аттестованными, проблемно–ориентированными испытательными лабораториями. Такие испытания необходимо проводить, когда программы управляют сложными процессами или обрабатывают столь важную информацию, что дефекты в них или недостаточное качество могут нанести значительный ущерб. Сертификационные испытания должны устанавливать соответствие комплексов программной документации и допускать их к эксплуатации в пределах изменения параметров внешней среды, исследованных при проведенных проверках. Эти виды испытаний характеризуются наибольшей строгостью и глубиной проверок и должны проводиться специалистами, не зависимыми от разработчиков и заказчиков (пользователей). Испытания ПС должны опираться на стандарты, формализованные методики и нормативные документы разных уровней. Множество видов испытаний целесообразно упорядочивать и проводить поэтапно в процессе разработки для сокращения затрат на завершающихся сертификационных испытаниях.

Сертификация комплексов программ является их испытанием в наиболее жестких условиях тестирования особым третейским коллективом специалистов, имеющим право на официальный государственный или ведомственный контроль функций и качества ПО и гарантирующим их соответствие стандартам и другим нормативным документам, а также надежность и безопасность применения. Получение и обобщение результатов испытаний, а также принятие решения о выдаче сертификата являются прерогативой *испытательных лабораторий.* Они должны быть специализированными для проведения испытаний объектов определенных классов, целенаправленно и систематически работать по созданию и совершенствованию методик и средств автоматизации испытаний ПО конкретного функционального назначения.

Специалисты–сертификаторы имеют право на расширение условий испытаний и на создание различных критических и стрессовых ситуаций в пределах нормативной документации, при которых должны обеспечиваться заданное качество и надежность решения предписанных задач. Если все испытания проходят успешно, то на соответствующую версию ПС оформляется специальный документ – *сертификат соответствия.* Этот документ официально подтверждает соответствие стандартам, нормативным и эксплуатационным документам функций и характеристик испытанных средств, а также допустимость их применения в определенной области.

Методология принятия решений о допустимости выдачи сертификата на ПС определяется оценкой степени его соответствия действующим и/или специально разработанным документам. В исходных нормативных документах должны быть сосредоточены все функциональные и эксплуатационные характеристики ПС, обеспечивающие заказчику и пользователям возможность корректного применения сертифицированного объекта во всем многообразии его функций и показателей качества. Выбор и ранжирование показателей должны проводиться с учетом классов ПС, их функционального назначения, режимов эксплуатации, степени критичности и жесткости требований к результатам функционирования и проявлениям возможных дефектов и ошибок. При этом могут привлекаться документы предшествующих этапов испытаний и документы, подтверждающие соблюдение аттестованных технологий при разработке программ на всех этапах. Испытания ПС в конкретных проблемно–ориентированных системах проводятся по правилам и методикам, принятым для соответствующих классов критических информационных систем, например, авиационных или космических комплексов.

Работы по сертификации объединяются *в технологический процесс,* на каждом этапе которого регистрируются документы, отражающие состояние и качество результатов разработки ПС. В зависимости от характеристик объекта сертификации на ее выполнение выделяются ресурсы различных видов. В результате сложность программ, а также доступные для сертификации ресурсы становятся косвенными критериями или факторами, влияющими на выбор методов испытаний, а также на достигаемые качество и надежность ПС.

Сертификационные испытания удостоверяют качество и надежность ПС только в условиях, ограниченных конкретными стандартами и нормативными документами, с некоторой конечной вероятностью. В реальных условиях эксплуатации принципиально возможны отклонения от характеристик внешней среды функционирования ПС за пределы, ограниченные сертификатом, и ситуации, не проверенные при сертификационных испытаниях. Эти обстоятельства способны вызывать катастрофические последствия, угрожающие надежности функционирования и безопасности применения ПС. Наличие сертификата у ПС для критических систем является необходимым условием их допуска к эксплуатации. Однако любой сертификат на сложные системы не может гарантировать

абсолютную их надежность применения, и всегда остается некоторый риск возникновения отказовых ситуаций.

Отсутствие гарантии достижения в процессе создания ПС абсолютной надежности их функционирования за счет использования высоких технологий, тестирования и сертификации заставляет искать дополнительные методы и средства повышения надежности ПС. Для этого разрабатываются и применяются *методы оперативного обнаружения дефектов и искажений при исполнении программ путем введения в них временной, информационной и программной избыточности.* Эти же виды избыточности используются для оперативного восстановления искаженных программ и данных и предотвращения возможности развития результатов реализации угроз до уровня, нарушающего надежность функционирования ПС. Основная задача ввода избыточности состоит в ограничении или исключении возможности аварийных последствий от возмущений, соответствующих отказу системы. Любые аномалии при исполнении программ необходимо блокировать и по возможным последствиям сводить до уровня сбоя путем быстрого восстановления.

Особенности обеспечения надежности функционирования импортных программных средств. При использовании зарубежных ПС в принципе в них возможны как злоумышленные, так и случайные, непредумышленные искажения вычислительного процесса, программ и данных, отражающиеся на надежности их функционирования. Злоумышленные вирусы и «закладки», хотя и мало–вероятны в серийных, широко тиражируемых в мире ПС, тем не менее требуют особых методов и средств целенаправленного их обнаружения и устранения. Зарубежным специалистам свойственно ошибаться так же, как и отечественным, однако более высокое качество используемых технологий разработки и современная проектировочная культура позволяют значительно снижать уровень дефектов в изделиях, поступающих на рынок и в эксплуатацию. Тем не менее *в любых сложных импортных ПС всегда не гарантировано полное, абсолютное отсутствие случайных ошибок,* которые остаются важнейшими дестабилизирующими факторами. Их применение в критических отечественных ПС требует соответствующего дополнительного контроля качества и специальных работ по обеспечению надежности при эксплуатации.

Представленные выше *объекты уязвимости, дестабилизирующие факторы и угрозы надежности* присущи любым программам и данным независимо от фирм–разработчиков. Однако методы предотвращения и снижения влияния угроз надежности для зарубежных ПС значительно отличаются. Разрыв в пространстве и времени при проектированиий конкретного ПС между первичными зарубежными создателями программных компонентов и потребителями, интеграторами, непосредственными создателями отечественных ИС затрудняет взаимодействие по предотвращению ошибок за счет применения CASE–технологий. Отечественный покупатель импортных ПС обычно не знает, какая технология была применена при их разработке и какие классы ошибок могли

быть оставлены. В составе пользовательской документации, как правило, отсутствуют исходные тексты программ и номенклатура тестов, использованных при их отладке. Поэтому методы предотвращения ошибок в импортных программах и данных почти всегда остаются недоступными и неизвестными отечественным специалистам. Это отражается на хроническом недоверии к качеству и надежности применения зарубежных программных компонентов или на слепой вере в их абсолютную безупречность.

Комплексирование готовых импортных прикладных ПС в конкретной отечественной ИС создает условия для их функционирования, не всегда адекватные предусмотренным разработчиками и проверенным при испытаниях, хотя и не выходящие за пределы требований эксплуатационной документации. Это способствует проявлению ранее скрытых дефектов и ошибок проектирования и их устранению. Для этого ответственные и квалифицированные поставщики зарубежных ПС имеют службы сопровождения, регистрации и накопления претензий пользователей и быстрого реагирования для устранения реальных дефектов функционирования. Легальная закупка и использование лицензионно чистых ПС, обеспеченных сопровождением солидной фирмы–поставщика, позволяют в значительной степени снижать влияние на надежность ПС дефектов, не предотвращенных в процессе проектирования.

Этому же может способствовать применение разработчиками ИС той же CASE–технологии, которая использовалась зарубежными решателями применяемых ПС. Для этого, в частности, наиболее популярные СУБД при продаже комплектуются средствами соответствующей CASE–технологии. Поставки прикладных программ различного назначения могут содержать рекомендации по использованию определенных CASE–технологий при комплексировании импортных компонентов в составе конкретной ИС. Применение той же CASE–технологии позволяет более полно понимать функциональные и технические возможности закупленных ПС в процессе их комплексирования в проблемно–ориентированной ИС. Это предотвращает наиболее сложные системные ошибки при использовании и интегрировании импортных ПС. Таким образом, хотя непосредственное предотвращение и исправление ошибок импортных ПС отечественными потребителями в процессе разработки ИС затруднительны, при соответствующем взаимодействии с конкретными зарубежными фирмами надежность ИС при использовании зарубежных программных продуктов можно поставить под достаточно жесткий контроль.

Систематическое тестирование импортных ПС в процессе проектирования производится самими разработчиками ИС. При отработке критических ПС целесообразны создание или закупка комплектов и генераторов тестов для тестирования конкретных ПС в составе ИС или автономно. Такое дополнительное тестирование повышает уверенность в качестве и надежности применяемых импортных продуктов в конкретном окружении, а также может приводить к обнаружению некоторых ошибок проектирования и комплексирования зарубеж-

ных программных компонентов. Их устранение в большинстве случаев целесообразно проводить силами зарубежной фирмы–разработчика с использованием организационно и юридически оформленного механизма сопровождения изделий поставщиком.

Обязательная сертификация зарубежных ПС для сложных, критических ИС предполагает сопровождение закупаемых, лицензионно чистых изделий сертификатом соответствия, выданным специализированной испытательной фирмой. Такое юридическое утверждение качества и надежности применения импортного изделия может быть недостаточным для особо важных, критических ИС, так как сертификат соответствия не всегда сопровождается протоколами испытаний и использованными при этом тестами, что не позволяет оценить полноту испытаний. В этих случаях *следует ориентироваться на дополнительную сертификацию импортных ПС отечественными проблемно–ориентированными, аттестованными сертификационными лабораториями.*

Такие испытания позволяют удостовериться в надежности применяемых зарубежных ПС, а также дополнительно выявить некоторые некорректности программ или документации. Их устранение требует взаимодействия с зарубежной фирмой–поставщиком для корректировки изделий и исключения дефектов. Самостоятельное исправление выявленных ошибок отечественными специалистами сопряжено с риском внесения дополнительных вторичных ошибок из–за недостаточной квалификации и неполной информации о детальном содержании текстов программ и описаний данных. Кроме того, любые изменения в сертифицированных изделиях помимо фирмы–поставщика приводят к автоматическому аннулированию выданного ею сертификата. Дополнительное подтверждение сертификата соответствия отечественными специалистами может значительно повысить уверенность в надежности зарубежных ПС.

Оперативные методы повышения надежности функционирования ПС предусматриваются в некоторых зарубежных изделиях и, в частности, в механизмах обеспечения целостности информации баз данных в реляционных СУБД. Однако разнообразие условий функционирования импортных ПС в сложных отечественных ИС не позволяет удовлетвориться только штатными методами оперативного обнаружения аномалий и восстановления вычислительного процесса, программ и данных. Методы и средства для этого могут быть в ряде случаев достаточно автономными и ориентированными на оперативное повышение надежности конкретной ИС в целом, а не только отдельных используемых ПС. Эти специализированные методы и средства могут разрабатываться отечественными специалистами для обеспечения комплексной надежности с использованием всех импортных компонентов. Такой подход позволяет обеспечить комплексирование разнородных ПС различных зарубежных поставщиков и специализированной отечественной системы оперативной защиты в едином комплексе программ. При этом важно использовать концепцию и стандарты открытых систем при взаимодействии между как закупаемыми, так и вновь разрабатываемыми компонен-

тами ПС, а также при их взаимодействии с внешней средой. Применение стандартизированных интерфейсов открытых систем между прикладными программами и CASE–технологией является эффективным современным методом повышения надежности информационных систем при наличии разнородных поставщиков компонентов.

Таким образом, для обеспечения надежности функционирования зарубежных ПС в составе отечественных ИС прежде всего следует полностью отказаться от применения нелегальных импортных программ и баз данных. Процессы закупки, контроля и применения импортных ПС для сложных отечественных ИС должны быть организованы и поддержаны дополнительными испытаниями. Использование лицензионно чистых ПС и тесное взаимодействие с их зарубежными фирмами–поставщиками позволяют эффективно продолжать тестирование программ при их комплексировании в отечественных ИС, оценивать и повышать надежность функционирования. При закупке зарубежных ПС целесообразно требовать сертификат соответствия и сопроводительную документацию по методам, тестам и результатам испытаний. В ряде случаев может быть необходима дополнительная сертификация импортных программ отечественными сертификационными лабораториями. Кроме того, для каждой критической ИС должна разрабатываться специализированная система обеспечения надежности ее функционирования путем оперативного контроля, выявления дефектов и восстановления вычислительного процесса, программ и данных при искажениях, угрожающих надежности и безопасности применения.

В импортных программах, кроме случайных ошибок, возможны *преднамеренные фрагменты – »закладные элементы» и вирусы с* целью реализации вредных для эксплуатации функций, которые не описаны в документации. До наступления определенного события «закладной элемент» остается неактивным, а при выполнении некоторого условия осуществляет разрушительные действия, приводящие к отказу и не предусмотренные функциональным назначением и документацией. Сертификация импортных программ для удостоверения отсутствия в них вирусов или «закладных элементов» может осуществляться в двух *ситуациях:*

– при наличии в составе поставляемой документации исходных текстов программ на языке программирования и описаний алгоритмов обработки информации;

– при наличии только эксплуатационной документации, недостаточной для анализа содержания и текстов программ.

В первом случае определение наличия в программе посторонних компонентов может производиться последовательной сверкой текста программы на языке программирования с описанием программы и спецификации. По тексту программы составляется блок–схема анализируемого алгоритма, которая сравнивается с алгоритмом, изложенным в описании программы. Если логическая структура алгоритмов различается, то следует проводить дополнительный

анализ элементов блок–схем, в которых обнаружены различия. Такие различия могут быть обусловлены дефектами документации на программу, случайными или предумышленными дефектами самой программы. Дефекты программы подлежат подробному анализу, классификации и корректировке, после чего ее следует подвергнуть полному тестированию и повторной сертификации на полное соответствие всей документации и отсутствие вредных компонентов.

Во втором случае, который является наиболее массовым, задача значительно усложняется, так как исходные документы о структуре и содержании программ и алгоритмов не поставляются. Для получения текста программы и алгоритма необходимо провести дизассемблирование объектного кода программы и выразить каждую функциональную команду кода ассемблера в виде логической процедуры для представления как оператора блок–схемы алгоритма. Построенная блок–схема подлежит анализу на наличие сомнительных конструкций, тупиков и висячих вершин, которые могут оказаться «закладными элементами». Каждая сомнительная группа процедур подлежит дальнейшему анализу на возможность ее принадлежности «закладному элементу», вирусу или случайной ошибке. Выявленные участки программы, содержащие случайные и предумышленные дефекты, должны корректироваться. После их исключения программа подлежит полному тестированию на соответствие эксплуатационной документации.

Четкое экономическое и юридическое взаимодействие с определенными фирмами – поставщиками импортных ПС позволяет держать под контролем не только достижимую надежность ИС, но и значительно снижает вероятность злоумышленных аномалий в поставляемых ими изделиях. Обнаружение и публикация сведений о предумышленных негативных компонентах в программных продуктах способны нанести значительный ущерб репутации и бизнесу фирмы.

ИМЛ

Тема 10 Обеспечение качества и надежности в процессе разработки сложных программных средств

10.1 Концепции повышения надежности в процессе разработки сложных программных средств.

10.2 Схема проектирования разработки программного обеспечения..

10.3 Требования к технологии и средствам автоматизации разработки сложных программных средств.

10.4 Качество программного обеспечения.

10.1 Концепции повышения надежности в процессе разработки сложных программных средств

Сложность – это одна из главных причин ненадежности программного обеспечения. Сложность почти не поддается ни точному определению, ни из-

мерению. Однако можно сказать, что мерой сложности объекта является количество интеллектуальных усилий, необходимых для понимания этого объекта.

В общем случае сложность объекта является функцией взаимодействия между его компонентами. Например, сложность внешнего проекта программной системы в некоторой степени определяется связями между всеми ее внешними сопряжениями, например между командами пользователя и соотношениями между входной и выходной информацией системы. Сложность архитектуры системы определяется связями между подсистемами. Сложность проекта программы – функция связей между модулями. Сложность отдельного модуля – функция связей между его командами.

В борьбе со сложностью программного обеспечения можно привлечь две концепции из общей теории систем. Первая – *независимость.* В соответствии с этой концепцией для минимизации сложности необходимо максимально усилить независимость компонентов системы. По существу это означает такое разбиение системы, чтобы высокочастотная динамика ее была заключена в единых компонентах, а межкомпонентные взаимодействия представляли лишь низкочастотную динамику системы.

Вторая концепция – *иерархическая структура.* Каждый уровень представляет собой совокупность структурных отношений между элементами нижних уровней. Концепция уровня позволяет понять систему, скрывая несущественные уровни детализации. Например, система, которую мы называем «человек», представляется иерархией. Социолог может интересоваться взаимоотношениями людей, не заботясь об их внутреннем устройстве. Психолог работает на более низком уровне иерархии. Он может исследовать различные логические и физические процессы в мозге, не рассматривая внутреннего строения областей мозга. Еще ниже в этой иерархии находится нейролог – он имеет дело со структурой основных компонентов мозга. Однако он может изучать мозг на этом уровне, не заботясь о молекулярной структуре отдельных белков в нейроне. Химик–органик интересуется построением сложных аминокислот из таких компонентов, как атомы углерода, водорода, кислорода и хлора. Физик–ядерщик изучает систему на уровне элементарных частиц в атоме и взаимодействия между ними.

Иерархия позволяет проектировать, описывать и понимать сложные системы. Если бы нельзя было принять описанный подход к изучению человека, социологу пришлось бы рассматривать его как необъятное и сложное множество субатомных частиц. Очевидно, что такое количество деталей подавило бы его, так что невозможны были бы даже те ограниченные знания о человеке, которыми мы располагаем.

К этим двум концепциям сокращения сложности (независимость и иерархическая структура) можно добавить третью: *проявление связей* всюду, где они возникают. Основная проблема многих больших программных систем – огромное количество независимых побочных эффектов, создаваемых компонентами системы. Из–за этих побочных эффектов систему невозможно понять. И можно

быть уверенным, что систему, в которой нельзя разобраться, было очень трудно спроектировать хотя бы с минимальной гарантией надежности.

Отношения с пользователем. Две самые распространенные ошибки при работе над программными проектами – это отказ от вовлечения пользователя системы в процессы принятия решений и неспособность понять его культурный уровень и окружающую его обстановку. При работе над многими проектами имеется тенденция умышленно исключать пользователя из процесса принятия решений. Обычно причина этого в том, что разработчик программного обеспечения чувствует: если вовлечь пользователя, тот никогда не придет к окончательному решению, его требования будут постоянно меняться. Для такой тревоги есть некоторые основания, но на практике преимущества от участия пользователя значительно перевешивают эти возможные неудобства. Вторая ошибка в программных проектах – разработчик системы часто слабо знает (или не знает вовсе) обстановку, в которой находится пользователь, т. е. плохо понимает, с какими именно трудностями сталкивается пользователь и как он будет применять программную систему. Бывает, например, так, что в проектировании операционной системы участвуют люди, сами никогда не использовавшие операционные системы. Есть разработчики языков программирования, никогда не пробовавшие реализовать прикладную систему на языке высокого уровня. Есть разработчики систем управления базами данных, которые никогда не пытались использовать базу данных в прикладной программе. Это не может не вести к серьезным ошибкам в программном обеспечении.

Единственно возможный способ избежать этих ошибок – поддерживать прочный контакт с пользователем в течение всего цикла разработки. С коллективом пользователей должны быть установлены такие отношения, чтобы те серьезно участвовали в процессе принятия решений на этапах определения требований, целей и внешнего проектирования. Привлечение пользователей на последующих этапах также желательно, особенно в процессе тестирования, когда пользователь может помочь разработчику системы значительно лучше понять, как следует тестировать систему. Будьте, однако, осмотрительны, привлекая пользователя к обсуждению деталей, судить о которых он некомпетентен. Например, хорошо, чтобы пользователь принимал участие в проектировании внешних характеристик системы, но привлекать его к такой работе, как анализ логики конкретного модуля, неразумно.

Имеется (уже упоминавшаяся) опасность, что пользователь может изменять свои требования к системе. Отметим, однако, что это никак не связано с непосредственным его участием в работе над проектом. Если требования к системе должны измениться, это произойдет независимо от того, привлечен ли пользователь непосредственно к работе или нет. В действительности если сам пользователь в работе не участвует, разработчик, вероятно, не узнает об изменении требований до тех пор, пока не станет слишком поздно. Если же пользователь непосредственно привлечен к работе, он может значительно лучше представлять

себе стоимость каждого изменения. Если правильно предусмотреть условия для изменения требований, участие пользователя, может оказаться выгодным и с этой точки зрения.

Участие потенциальных пользователей в создании новых систем, которые разрабатываются не по заказу и сведения, о которых составляют коммерческую тайну, также не является недопустимым.

Хотя такой продукт предназначен не для конкретного потребителя, разработчик и в этом случае, вероятно, хорошо представляет себе возможных покупателей. С одним или несколькими из них может быть подписано соглашение о сохранении коммерческой тайны, что позволит и возможным покупателям системы участвовать в ее разработке до того, как о ней будет публично объявлено.

Преодоление второй трудности – непонимание запросов пользователя и окружающей его обстановки – требует, чтобы проектировщики программной системы досконально представили себе особенности его работы. Обычно принимается весьма неэффективное решение – командировать основных проектировщиков для изучения положения дел. Проектировщики получают поверхностное представление о существующих системах и совсем не получают сколько–нибудь глубокого представления о том, как система используется и в чем же состоят подлинные проблемы.

10.2 Схема проектирования разработки программного обеспечения

Большинство процессов разработки программного обеспечения – это процессы решения некоторых задач. Внешнее проектирование сводится к решению такой задачи: «Переведите множество целей системы во внешние спецификации», где цели – данные, а внешние спецификации – неизвестные. В задаче проектирования логики модуля даны внешние спецификации модуля, а неизвестное – текст его программы. Отладка – это задача на построение исправления ошибки (неизвестное) по описанию ее симптомов (данные).

Приведем один из методов решения задачи.

1 *Поймите задачу*

Изучите данные.

Изучите неизвестные.

Достаточно ли данных для решения? Непротиворечивы ли они?

2 *Составьте план*

Чего вы должны добиваться?

Какие методы проектирования будут использоваться?

Встречалась ли вам уже такая задача?

Не знаете ли вы близкой задачи?

Можете ли вы воспользоваться ее результатом?

Можете ли вы решить более специализированную или аналогичную задачу?

Можете ли вы решить часть задачи?

3 *Выполните план*

Следуйте своему плану решения задачи. Проверяйте правильность каждого шага.

4 *Проанализируйте решение* Все ли данные вы использовали? Проверьте правильность решения. Можете ли вы воспользоваться полученным результатом или примененным методом при решении других задач?

Рассмотрим основные положения этого метода.

Поймите задачу

Худшая из ошибок, которые могут быть сделаны при решении задачи, – не вполне разобраться в ее постановке. Понять задачу – это значит понять два ее компонента: данные и неизвестное. Данные – это все элементарные факты, касающиеся задачи, и связи между фактами и неизвестным. Усвоение всех данных о сложной задаче – большая, но абсолютно неизбежная работа. При этом в первую очередь необходимо хорошо охватить «общую картину» данных без деталей, которые, однако, также запоминаются «в сторонке», чтобы их можно было легко вспомнить позже. Есть много способов добиться этого. Например, кое–кто физически разрезает спецификации на куски, которые затем расклеиваются на стене в определенном порядке. Это позволяет увидеть общую картину и при этом определить место для каждой детали.

Вторая часть задачи – неизвестное. Проектировщику следует понимать, какую форму должно иметь решение. Если, например, задача – детальное внешнее проектирование программы, то проектировщик должен ясно представлять назначение внешних спецификаций, их потенциальных читателей, формат и т. д.

Исследуя задачу, проектировщик должен также исследовать данные, чтобы убедиться, что их достаточно для решения задачи и они не противоречат друг другу.

Составьте план

Прежде чем приступить к решению, следует разработать его план. Отсутствие плана – очень распространенная ошибка. Например, проектировщики программной системы, которые потратили время на то, чтобы понять задачу, но затем немедленно приступили к ее решению, не пожелав тратить время на планирование своих усилий, в конце концов, могут прийти к хорошему решению, но не раньше чем после нескольких ненужных фальстартов.

Прежде всего, в плане нужно определить, чего вы хотите добиться. Десять человек могут иметь десять разных мнений относительно «правильного» ответа на задачу проектирования; проектировщик должен предусмотреть те конкретные аспекты решения, которые требуют наибольшего внимания. К сожалению, в большинстве проектов разработчики имеют слишком много свободы в этом отношении: каждый проектировщик принимает компромиссные решения, основываясь исключительно на собственном мнении, что приводит к несогласованности многих решений в системе. Идея *целей проекта* является решением этой проблемы. Суть идеи состоит в том, что на уровне всего проекта определяются

общие цели, которыми следует руководствоваться во всех решениях при проектировании.

Ключевым компонентом успешного проектирования является методология. Выбор подходящей методологии для каждого конкретного процесса проектирования должен быть зафиксирован в качестве одной составляющей плана.

Накопленный опыт, образование и имеющиеся решения проблем также существенно влияют на успех дела. Обработка данных в своей эволюции достигла такой точки, когда проектировщик крайне редко сталкивается с задачей, которая уже не была бы решена частично или полностью. Например, разработчик новой операционной системы должен понимать, что уже созданы сотни операционных систем и на эту тему написан не один учебник. Проектировщик, столкнувшись с задачей сортировки, должен знать, что уже придумано и проанализировано множество алгоритмов сортировки. При разумном подходе к решению задач начинать следует с анализа своего опыта и опыта других с тем, чтобы проверить, не была ли задача уже решена. Даже если готовое решение найти не удается, вероятно, когда–то была решена близкая задача. Проектировщик системы резервирования авиабилетов может сообразить, что у нее есть много сходства с другими системами резервирования, например с системой резервирования мест в гостинице. Тогда ему, возможно, удастся выделить и применить у себя элементы решения задачи о резервировании мест в гостинице.

Если все эти методы не приносят успеха, может оказаться эффективным решение более специализированной задачи или части задачи. Если количество деталей в постановке задачи слишком велико, разработчику следует посмотреть, нельзя ли упростить ее, отбросив часть деталей. В результате либо станет ясно, как следует изменить упрощенное решение, чтобы учесть и отброшенные детали, либо удастся лучше увидеть возможные решения, так что можно будет прекратить заниматься упрощенным вариантом и начать сначала.

Выполните план

Следующий шаг – действительно решить задачу в соответствии с запланированным подходом. Поскольку решение обычно состоит из ряда последовательных шагов, разработчик в процессе решения должен пытаться проверить правильность каждого шага.

Проанализируйте решение

После того как результат получен, нужно еще его проверить. Разработчик должен просмотреть все данные, чтобы убедиться, что учтено все, что имеет отношение к делу. Полезно для этого еще раз перечитать буквально каждое слово постановки задачи, вычеркивая каждый использованный в решении факт, а затем проверить, насколько существенно для задачи то, что осталось незачеркнутым. Разработчик должен также проверить правильность решения задачи.

10.3 Требования к технологии и средствам автоматизации разработки сложных программных средств

В стандартах и моделях жизненного цикла ПС с различной глубиной определено содержание этапов и частных работ при создании и модификации компонентов и ПС в целом. Для планирования и управления обеспечением качества и надежности ПС эти модели служат структурной базой объектов, работ и документов при детализации и реализации требований к показателям качества ожидаемых результатов. Необходимая надежность объектов формируется и обеспечивается в процессе выполнения частных работ каждого этапа и окончательно удостоверяется испытаниями и документами при их завершении. Для обеспечения качества и надежности ПС стандартами *рекомендуется формулировать требования:*

– к объекту разработки на данном этапе – к его программным и информационным компонентам, а также к интерфейсу между ними и внешней средой;
– к процессу, технологии и организации выполнения совокупности работ и документов каждого этапа;
– к методам и характеристикам средств автоматизации выполнения работ, обеспечивающим необходимую надежность функционирования и качество ПС;
– к методам и средствам контроля, измерения и документирования качества процессов и результатов выполненных работ.

Выполнение этих требований должно контролироваться путем измерения объектов и процессов разработки. Измерения объектов разработки сводятся к регулярной поэтапной регистрации показателей качества, а также к сопоставлению их с заданными требованиями. При обнаружении отклонений от требований должны приниматься меры либо для улучшения реальных показателей, либо по корректировке требований к показателям для данного компонента на контролируемом этапе.

Требования к инструментальным средствам автоматизации разработки надежных ПС наиболее полно изложены в стандарте *IEEE 1209–1992*. Стандарт содержит рекомендации по оценке и выбору инструментальных средств, поддерживающих процессы жизненного цикла программных средств, включая процессы управления проектами, процессы разработки и процессы, следующие за разработкой, а также интегральные процессы жизненного цикла ПС. Для оценки и выбора инструментальной среды и CASE–средств стандартом рекомендуется использовать приведенные ниже наборы правил и критериев. Группы критериев в стандарте выделены и сформированы с учетом общих требований стандарта ISO 9126:1991 по оценке качества программных продуктов.

Технологическую среду и CASE–средства стандартом рекомендуется описывать и выбирать в соответствии с показателями:

– соответствие стандартам среды, указанным в списке характеристик и функций, поддерживаемых CASE–средством, включая стандарты на языки, базы данных, репозиторий, коммуникации, графический интерфейс пользователя, документацию, разработку, управление конфигурацией, безопасность, обмен

информацией, интеграцию данных, управление или пользовательский интерфейс;
– совместимость с другими инструментальными средствами, включая возможность взаимодействия и/или прямого обмена данными (например, с системами подготовки текстов и другими средствами документирования, базами данных, репозиториями и другими CASE–средствами);
– поддержка конкретных методологий, например, объектно–ориентированного анализа, объектно–ориентированного проектирования, проектирования «сверху–вниз»;
– языковая поддержка, включая языки программирования, языки определения данных, языки структурированных запросов, графические языки;
– ввод и редактирование спецификаций требований к разрабатываемому ПС, включая требования к функциям, данным, интерфейсам, качеству, производительности, среде функционирования, стоимости и планированию;
– языки спецификаций требований – возможность CASE–средств импортировать, экспортировать или редактировать информацию требований, используя формальный язык, контроль непротиворечивости спецификаций и полноты;
– возможность моделировать аспекты потенциального функционирования разрабатываемой системы на основе требований и/или проектных данных, имеющихся в распоряжении CASE–средства, включая эффективность системы, интерфейс оператора, архитектурную производительность (время отклика, загрузку, пропускную способность);
– прототипирование – возможность проектирования и генерации предварительной версии всей системы или ее части на основе требований и/или проектных данных, имеющихся в распоряжении CASE–средства;
– формирование структуры отчетов, которые будут создаваться разрабатываемой системой.

В соответствии со стандартом должен быть обеспечен *анализ потенциальной корректности и надежности* входящих в ПС программных компонентов, включающий:
– процедуры оценки сложности программ, связанной с числом вложенных циклов, полноты покрытия тестами, оценку количества остающихся ошибок;
– обратную (реверсную) инженерию, т.е. возможность ввода действующего исходного кода в одном или нескольких языках и получения из него проектных данных с предоставлением результатов пользователю;
– реструктуризацию исходного кода: ввод исходного кода в одном или нескольких языках, модифицирование его формата и/или структуры и выдачу файла исходного кода на том же самом языке;
– анализ исходного кода и предоставления результатов пользователю: измерения размеров, вычисления метрик сложности, генерации перекрестных ссылок, обзора соответствия использованным стандартам;
– отладку: поддержку идентификации и изоляции ошибок в программе, включая выполнение программ с трассировкой, обеспечение обратного выполнения и ло-

вушек, идентификацию мест, где имеются ошибки, и часто выполняемых сегментов в терминах исходного кода.

Требования стандарта к средствам *управления проектом сложного ПС* включают:

– способность CASE–средства оценивать стоимость, формировать планы и другие показатели проекта по данным, вводимым пользователем;
– управление действиями и ресурсами путем поддержки ввода пользователем данных для планирования проекта, данных о фактических действиях и анализ этих данных, включая планы, ресурсы компьютеров, назначение персонала, бюджет проекта, а также возможность определения условий выполнения проекта;
– управление тестовыми процедурами: возможность поддержки управления действиями по тестированию и тестовыми программами, планирования действий по тестированию, регистрации результатов тестирования, генерации отчетов о состоянии тестируемых программ;
– управление качеством разрабатываемого ПС – ввод и обработка данных о качестве, их анализ и генерация отчетов об управлении качеством;
– управление действиями по корректировке плана проекта, отчетов о проблемах и дефектах, возникших в ходе выполнения проекта.

Управление конфигурацией версий проекта ПС должно обеспечивать:

– возможность управлять физическим доступом к элементам данных и их изменением, включая возможность специфицировать с помощью идентификаторов компоненты, к которым возможен доступ только для чтения, запрещен доступ, а также возможность отлаживать элементы данных для их модификации, ограничивать доступ к ним до тех пор, пока они не исправлены и не проверены, и отменять ограничения после внесения изменений;
– трассирование модификаций – запись всех модификаций, сделанных в системе при ее разработке или сопровождении;
– управление версиями, возможность записи и выполнения функций управления многократными версиями системы, которые могут иметь общие компоненты;
– учет конфигурационного статуса и предоставление пользователю отчетов, устанавливающих историю, содержимое и статус различных единиц конфигурации, находящихся под управлением;
– генерацию выпусков (релизов) ПС и его компонентов, возможность поддержки определения пользователем шагов, необходимых для создания версии и автоматизированного выполнения этих шагов;
– возможности автоматического архивирования элементов данных для последующего поиска и применения.

Поддержка разработки технологической и эксплуатационной документации на комплекс программ и его компоненты по требованиям стандарта IEEE 1209 должна включать:

– редактирование текстов – возможность вводить и редактировать данные в текстовом формате;
– графическое редактирование – ввод и редактирование данных в графическом формате;
– редактирование на базе форм – поддержка ввода и редактирование данных в форме, определенной пользователем;
– возможности настольного издательства для оформления документации;
– контроль соответствия выходных результатов CASE–средства стандартам на документацию ПС;
– автоматическое извлечение текстовых и графических данных и генерация документов, специфицированных пользователем.

Критерии удобства применения CASE–средства в процессе разработки ПС включают:
– непротиворечивость пользовательского интерфейса, включая размещение и представление экранных элементов, совместно появляющихся на экране, и методы входа пользователя в систему;
– легкость изучения, измеряемую количеством времени и усилий, которые требуются от пользователя, чтобы понять штатные операции CASE–средства и производительно его использовать;
– адаптируемость CASE–средства силами пользователя к его специфичным потребностям, включая различные наборы символов, разные способы представления символов и графики, разные форматы данных, методы ввода и вывода;
– качество документации CASE–средства, включая полноту, ясность, читаемость, полезность;
– доступность и качество учебных материалов, включая учебные материалы, доступные в режиме on–line, руководства по обучению, курсы обучения и визуальные материалы;
– уровень требований к знаниям пользователя, необходимым для эффективного использования CASE–средства, и легкость работы с CASE–средством как для новичков, так и для опытных пользователей;
– общность пользовательского интерфейса между CASE–средством и другими инструментальными средствами, функционирующими в среде проектируемой системы;
– полноту и качество функций помощи в режиме «help»;
– ясность диагностики – понимаемость и полезность диагностических сообщений, получаемых пользователем;
– приемлемое время отклика – время, требующееся для того, чтобы ответить на запрос пользователя в условиях применяемой операционной среды CASE–средства;
– легкость инсталляции CASE–средства, как первоначальной, таки при последующих изменениях.

Критерии оценки эффективности CASE–средства по требованиям стандарта должны учитывать данные для выполняемых объектов и работ как типичного, так и максимального размера и сложности:

– оптимальные требования к объему внешней, общей памяти, чтобы обеспечить работу с любыми требующимися и/или генерируемыми данными на приемлемом уровне производительности;

– оптимальные требования к объему оперативной памяти, адресуемой центральным процессором, для того, чтобы CASE–средство могло загружаться и функционировать на приемлемом уровне производительности;

– оптимальные требования к процессору для функционирования CASE–средства на приемлемом уровне производительности;

– производительность, измеряемую как время, в течение которого CASE–средство выполняет характерные задачи, например, время ответа на запрос.

10.4 Качество 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.

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.

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 производ-

ство 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 бета–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:

1 Проведение обучения персонала, переподготовки.

2 Тщательное документирование всех изменений в структуре программного средства. Для этого используются средства поддержки версионности.

3 Назначение ответственных лиц за каждую доработку программного сред-

ства.
4 Уделение внимания текущему контролю качества и заключительному контролю качества.
5 Обеспечение мониторинга качества, например, фиксирование ошибок, поступивших от пользователя программного средства. Использование систематических испытательных методов, где испытания будут разработаны параллельно с разработкой программы.
6 Введение внутренних стандартов. Такие стандарты обычно содержат соглашения о именовании переменных в программном коде, наименовании файлов данных, процедур и функций.
7 Организация отдела тестирования как самостоятельного подразделения.
8 Проведение совместных аттестаций с пользователем.
9 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.
1 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).
2 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.
3 Использовать формальный язык спецификаций (например, UML, DESIGN IDEF).
4 Выбор надежной СУБД (если программное средство работает с массивами информации и использование СУБД оправдано).
5 Тщательное тестирование программного обеспечения.
6 Широкое внедрение автоматизации тестирования.
7 Использование полностью проверенной программной среды окружения (ОС) и языка программирования, которые минимизируют опасность внесения ошибки.
8 Использование статистических методов для сбора информации о качестве ПС.
9 Изучение результатов испытаний (тестов) и ошибок для использования в постоянном усовершенствовании программы. Источник в случае возникнове-

ния отказа должен быть найден и устранен. Недостаточно найти ошибку в программном обеспечении и исправить ее. Изменения должны быть сделаны в процессе разработки ПО.

10. Использование испытательной среды, которая предостережет от передачи пользователю ненадежного программного обеспечения. Создание автоматических средств приемки.

Отметим, что однородной картины в области контроля качества и действий по его улучшению в связи с разработкой программного обеспечения нет. Как видно, качество программного обеспечения тесно связано с жизненным циклом программного обеспечения и с тестированием. Качество является комплексной проблемой.

ГГУ ИМЕНИ Ф. СКОРИНЫ

РАЗДЕЛ 5 ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Тема 11 Основные понятия

11.1 Проблематика тестирования программного обеспечения.
11.2 Основные определения.
11.3 Экономика тестирования.
11.4 Аксиомы (принципы) тестирования.

11.1 Проблематика тестирования программного обеспечения

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».

О состоянии дел лучше всего свидетельствует тот факт, что большинство людей, работающих в области обработки данных, даже не могут правильно определить понятие «тестирование», и это на самом деле главная причина неудач. Если попросить любого профессионала определить понятие «тестирование» либо открыть (как правило, слишком краткую) главу о тестировании любого учебника программирования, то скорее всего можно встретить такое определение: «Тестирование – процесс, подтверждающий правильность программы и демонстрирующий, что ошибок в программе нет». Основной **недостаток** подобного определения заключается в том, что оно совершенно неправильно; фактически это почти определение антонима слова «тестирование». Поэтому определение описывает невыполнимую задачу, а так как тестирование зачастую все же выполняется с успехом, по крайней мере, с некоторым успехом, то такое определение логически некорректно.

Правильное определение тестирования таково: *Тестирование – процесс выполнения программы с намерением найти ошибки.*

Тестирование оказывается довольно необычным процессом (вот почему оно и считается трудным), так как это процесс разрушительный. Ведь цель проверяющего (тестовика) – заставить программу сбиться. Он доволен, если это ему удается; если же программа на его тесте не сбивается, он не удовлетворен.

Невозможно гарантировать отсутствие ошибок в программе; в лучшем случае можно попытаться показать наличие ошибок. Если программа правильно ведет себя для значительного набора тестов, нет оснований утверждать, что в ней нет ошибок; со всей определенностью можно лишь утверждать, что неизвестно, когда эта программа не работает. Конечно, если есть причины считать данный набор тестов способным с большой вероятностью обнаружить все

возможные ошибки, то можно говорить о некотором уровне уверенности в правильности программы, устанавливаемой этими тестами.

О тестировании говорить довольно трудно, поскольку, хотя оно и способствует повышению надежности программного обеспечения, его значение ограничено. Надежность невозможно внести в программу в результате тестирования, она определяется правильностью этапов проектирования. Наилучшее решение проблемы надежности – с самого начала не допускать ошибок в программе. Однако вероятность того, что удастся безупречно спроектировать большую программу, бесконечно мала. Роль тестирования состоит как раз в том, чтобы определить местонахождение немногочисленных ошибок, оставшихся в хорошо спроектированной программе. Попытки с помощью тестирования достичь надежности плохо спроектированной программы совершенно бесплодны.

11.2 Основные определения

Хотя в тестировании можно выделить несколько различных процессов, такие термины, как «тестирование», «отладка», «доказательство», «контроль» и «испытание», часто используются как синонимы и, к сожалению, для разных людей имеют разный смысл. Нашу классификацию различных форм тестирования мы начнем с того, что дадим эти определения, слегка их дополнив и расширив их список.

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.

11.3 Экономика тестирования

Дав такое определение тестированию, необходимо на следующем шаге рассмотреть возможность создания теста, обнаруживающего все ошибки программы. Покажем, что ответ будет отрицательным даже для самых тривиальных программ. 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.

Тестирование программы как «черного ящика»

Одним из способов изучения поставленного вопроса является исследование стратегии тестирования, называемой 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 мы можем рассматривать внутреннюю структуру программы и делать некоторые разумные, но, конечно, не обладающие полной гарантией достоверности предположения.

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 (к сожалению, здесь часто не используется спецификация программы).

Сравним способ построения тестов при данной стратегии с исчерпывающим входным тестированием стратегии «черного ящика». Непосвященному может показаться, что достаточно построить такой набор тестов, в котором каждый оператор исполняется хотя бы один раз; нетрудно показать, что это неверно. Не вдаваясь в детали, укажем лишь, что исчерпывающему входному тестированию может быть поставлено в соответствие *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.*

11.4 Аксиомы (принципы) тестирования

Сформулируем **основные принципы** тестирования.

Хорош тот тест, для которого высока вероятность обнаружить ошибку.

Одна из самых сложных проблем при тестировании – решить, когда нужно его закончить.

Необходимая часть всякого теста – описание ожидаемых выходных данных или результатов.

Избегайте невоспроизводимых тестов, не тестируйте «с лету».

Готовьте тесты как для правильных, так и для неправильных входных данных.

Детально изучите результаты каждого теста.

По мере того как число ошибок, обнаруженных в некотором компоненте программного обеспечения, увеличивается, растет относительная вероятность существования в нем необнаруженных ошибок.

Поручайте тестирование самым способным программистам.

Считайте тестируемость ключевой задачей вашей разработки.

Проект системы должен быть таким, чтобы каждый модуль подключался к системе только один раз

Никогда не изменяйте программу, чтобы облегчить ее тестирование.

Тестирование, как почти всякая другая деятельность, должно начинаться с постановки целей.

Приведем еще раз три наиболее **важных принципа** тестирования.

1 Тестирование – это процесс выполнения программ с целью обнаружения ошибок.

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

3 Удачным считается тест, который обнаруживает еще не выявленную ошибку.

ГГ

Тема 12 Тестирование надежности программного обеспечения

12.1 Философия тестирования.

12.2 Тестирование модулей.

12.3 Комплексное тестирование.

12.4 Организация и этапы тестирования при испытаниях надежности сложных программных средств.

12.1 Философия тестирования

Тестирование программного обеспечения охватывает ряд видов деятельности, аналогичный последовательности процессов разработки программного обеспечения. Сюда входят постановка задачи для теста, проектирование, написание тестов, тестирование тестов, выполнение тестов и изучение результатов тестирования. Решающую роль играет проектирование теста. Возможен целый спектр подходов к выработке философии, или стратегии проектирования. Чтобы ориентироваться в стратегиях проектирования тестов, стоит рассмотреть два крайних подхода, находящихся на границах спектра. Следует отметить также, что многие из тех, кто работает в этой области, часто бросаются в одну или другую крайность.

Сторонник подхода, соответствующего левой границе спектра (рисунок 12.1), проектирует свои тесты, исследуя внешние спецификации или спецификации сопряжения программы или модуля, которые он тестирует. Программу он рассматривает как «черный ящик». Позиция его такова: «Меня не интересует, как выглядит эта программа и выполнил ли я все команды или все пути. Я буду удовлетворен, если программа будет вести себя так, как указано в спецификациях». Его идеал – проверить все возможные комбинации и значения на входе.

Приверженец подхода, соответствующего другому концу спектра, проектирует свои тесты, изучая логику программы. Он начинает с того, что стремится подготовить достаточное число тестов для того, чтобы каждая команда была выполнена по крайней мере один раз. Если он немного более искушен, то проектирует тесты так, чтобы каждая команда условного перехода выполнялась в каждом направлении хотя бы раз. Его идеал – проверить каждый путь, каждую ветвь алгоритма. При этом его совсем (или почти совсем) не интересуют спецификации.

Рисунок 12.1 – Схема спектра подходов к проектированию тестов

Ни одна из этих крайностей не является хорошей стратегией. Первая из них, а именно та, в соответствии с которой программа рассматривается как «черный ящик», предпочтительней. К сожалению, она страдает тем недостатком, что совершенно неосуществима. Рассмотрим попытку тестирования тривиальной программы, получающей на входе три числа и вычисляющей их среднее арифметическое. Тестирование этой программы для всех значений входных данных невозможно. Даже для машины с относительно низкой точностью вычислений количество тестов исчислялось бы миллиардами. Даже имея вычислительную мощность, достаточную для выполнения всех тестов в разумное время, мы потра-

тили бы на несколько порядков больше времени для того, чтобы эти тесты подготовить, а затем проверить. Такие программы, как системы реального времени, операционные системы и программы управления данными, которые сохраняют «память» о предыдущих входных данных, еще хуже. Нам потребовалось бы тестировать программу не только для каждого входного значения, но и для каждой последовательности, каждой комбинации входных данных. Поэтому исчерпывающее тестирование для всех входных данных любой программы неосуществимо.

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. Каждый тест должен быть представителем некоторого класса входных значений, так чтобы его правильное выполнение создавало у нас некоторую убежденность в том, что для определенного класса входных данных программа будет выполняться правильно. Это обычно требует некоторого знания алгоритма и структуры программы, и мы, таким образом, смещаемся к правому концу спектра.

12.2 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 концен-

трируется 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: пошаговое тестирование; восходящее тестирование; нисходящее тестирование; метод «большого скачка»; метод сандвича; модифицированный метод сандвича.

Метод сандвича. 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 снизу и сверху, встречаясь где– то в середине. Точка встречи зависит от конкретной тестируемой программы и должна быть заранее определена при изучении ее структуры. Например, если разработчик может представить свою систему в виде уровня прикладных модулей, затем уровня модулей обработки запросов, затем уровня примитивных функций, то он может решить применять нисходящий метод на уровне прикладных модулей (программируя заглушки вместо модулей обработки запросов), а на остальных уровнях применить восходящий метод. Метод сандвича сохраняет такое достоинство нисходящего и восходящего подходов, как начало интеграции системы на самом раннем этапе. Поскольку вершина программы вступает в строй рано, мы, 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.

Модифицированный метод сандвича. При тестировании методом сандвича возникает та же проблема, что и при нисходящем подходе, но 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.

Восходящее тестирование. 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.

Здесь отсутствуют проблемы, связанные с невозможностью или трудностью создания всех тестовых ситуаций, характерные для нисходящего тестирования. Драйвер как средство тестирования применяется непосредственно к тому модулю, который тестируется, где нет промежуточных модулей, которые следует принимать во внимание. Анализируя другие проблемы, возникающие при нисходящем тестировании, можно заметить, что при восходящем тестировании невозможно принять неразумное решение о совмещении тестирования с проектированием программы, поскольку нельзя начать тестирование до тех пор, пока не спроектированы модули нижнего уровня. Не существует также и трудностей с незавершенностью тестирования одного модуля при переходе к тестированию другого, потому что при восходящем тестировании с применением нескольких версий заглушки нет сложностей с представлением тестовых данных.

Нисходящее тестирование. Нисходящее тестирование (называемое также нисходящей разработкой) 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 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

ГГУ ИМЕНИ Ф. СКОРИНЫ

основательное тестирование некоторых модулей может потребовать крайне изощренных заглушек. Программист часто решает не тратить массу времени на их программирование, а вместо этого пишет простые заглушки и проверяет лишь часть условий в модуле. Он, конечно, собирается вернуться и закончить тестирование рассматриваемого модуля позже, когда уберет заглушки. Такой план тестирования – определенно не лучшее решение, поскольку об отложенных условиях часто забывают.

Второй тонкий недостаток нисходящего подхода состоит в том, что он может породить веру в возможность начать программирование и тестирование верхнего уровня программы до того, как вся программа будет полностью спроектирована. Эта идея на первый взгляд кажется экономичной, но обычно дело обстоит совсем наоборот. Большинство опытных проектировщиков признает, что проектирование программы – процесс итеративный. Редко первый проект оказывается совершенным. Нормальный стиль проектирования структуры программы предполагает по окончании проектирования нижних уровней вернуться назад и подправить верхний уровень, внеся в него некоторые усовершенствования или исправляя ошибки, либо иногда даже выбросить проект и начать все сначала, потому что разработчик внезапно увидел лучший подход. Если же головная часть программы уже запрограммирована и оттестирована, то возникает серьезное сопротивление любым улучшениям ее структуры. В конечном итоге за счет таких улучшений обычно можно сэкономить больше, чем те несколько дней или недель, которые рассчитывает выиграть проектировщик, приступая к программированию слишком рано.

Метод «большого 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.

Пошаговое 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.

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

2.Расход машинного времени при монолитном тестировании меньше.

3.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.

ГГУ имени Ф. Скорины

4.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 после дней фазы процесса тестирования.

5.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.

6.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.

Во– первых, термины «нисходящее тестирование», «нисходящая разработка», «нисходящее проектирование» часто используются как синонимы. Действительно, термины «нисходящее тестирование» и «нисходящая разработка» являются синонимами (в том смысле, что они подразумевают определенную стратегию при тестировании и создании текстов модулей), но нисходящее проектирование – это совершенно иной и независимый процесс. Программа, спроектированная нисходящим методом, может тестироваться и нисходящим, и восходящим методами.

Во– вторых, восходящая разработка, или тестирование, часто отождествляется с монолитным тестированием. Это недоразумение возникает из– за того, что начало восходящего тестирования идентично монолитному при тестировании нижних или терминальных модулей. Но выше мы показали, что восходящее тестирование на самом деле представляет собой пошаговую стратегию.

РЕПОЗИТОРИЙ ГГУ ИМЕНИ Ф. СКОРИНЫ

12.3 Комплексное тестирование

Комплексное тестирование, вероятно, самая непонятная форма тестирования. Во всяком случае, комплексное тестирование не является тестированием всех функций полностью собранной системы; тестирование такого типа называется *тестированием внешних функций.* Комплексное тестирование – процесс поисков несоответствия системы ее исходным целям. Элементами, участвующими в комплексном тестировании, служат сама система, описание целей продукта и вся документация, которая будет поставляться вместе с системой. Внешние спецификации, которые были ключевым элементом тестирования внешних функций, играют лишь незначительную роль в комплексном тестировании.

Часть аргументов в пользу этого должна быть уже очевидной: измеримые цели необходимы, чтобы определить правила для процессов проектирования. Остальные соображения должны проясниться сейчас. Если цели сформулированы, например, в виде требования, чтобы система была достаточно быстрой, вполне надежной и чтобы в разумных пределах обеспечивалась безопасность, тогда нет способа определить при тестировании в какой степени система достигает своих целей.

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 такая роскошь часто недоступна 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 вопрос о реальной среде (запуск настоящего космического корабля или выстрел настоящей ракетой) обычно не стоит. Кроме того, как мы увидим дальше, некоторые типы комплексных тестов не осуществимы в реальной обстановке по экономическим соображениям, и лучше всего выполнять их в моделируемой среде.

Комплексное тестирование – наиболее творческий из всех 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.

Комплексное тестирование системы – такая особая и такая важная работа, что в будущем возможно появление компаний, специализирующихся в основном на комплексном тестировании систем, разработанных другими.

По своей природе комплексные тесты никогда не сводятся к проверке отдельных функций системы. Они часто пишутся в форме сценариев, представляющих ряд последовательных действий пользователя. Например, один комплексный тест может представлять подключение терминала к системе, выдачу последовательно 10–20 команд и затем отключение от системы. Вследствие их особой сложности тесты системы состоят из нескольких компонентов: сценария, входных данных и ожидаемых выходных данных. В сценарии точно указываются действия, которые должны быть совершены во время выполнения теста.

12.4 Организация и этапы тестирования при испытаниях надежности сложных программных средств

Основные этапы тестирования и испытаний комплекса программ и его компонентов представлены на рисунке 12.2. Для каждого этапа на рисунке представлены основные исходные данные и результаты тестирования и испытаний. При тестировании, отладке и испытаниях *корректности компонентов комплексов программ выделены следующие этапы:*

– комплексирование модулей и отладка автономных групп программ в статике без взаимодействия с другими компонентами и, возможно, без подключения к операционной системе реального времени;

– тестирование и отладка групп программ в статике с учетом взаимодействия с некоторыми другими важнейшими компонентами и с базой данных;
– тестирование и отладка отдельных программных компонентов в реальном времени во взаимодействии с другими функциональными компонентами и с основными компонентами операционной системы и базы данных.

Сложность тестирования компонентов на этих этапах в значительной степени обусловлена несинхронным процессом их разработки и отладки. Первично спланированная логика сопряжения между собой отдельных компонентов и подключения их к операционной системе не всегда выполняется из–за задержек в автономной отладке некоторых из них. Целесообразная последовательность отладки определенных компонентов может нарушаться неготовностью к сопряжению с ними других взаимодействующих программ. В результате к комплексному тестированию и испытаниям ПС могут быть готовы не все необходимые компоненты, и их приходится начинать с некоторой не всегда самой важной и целесообразной совокупности групп программ.

Для обеспечения имитации объектов внешней среды и других взаимодействующих групп программ на этих этапах используются частные генераторы соответствующих тестов. Эти генераторы тестов целесообразно разрабатывать и оформлять как отдельные модули или группы программ, функционирующие на той же технологической ЭВМ и в той же операционной среде, что и отлаживаемые компоненты. Совместно с ними также реализуются и функционируют частные специализированные программы для обработки отдельных результатов отладки соответствующих групп программ, что также требует некоторых ресурсов ЭВМ. При этом не всегда удается полностью реализовать реальный масштаб времени для отдельных функциональных компонентов и приходится применять стартстопный режим тестирования или растягивать циклы решения функциональных задач и имитировать псевдореальный масштаб.

После комплексирования основных функциональных компонентов начинаются тестирование и испытания ПС в целом. Для них наиболее характерны следующие *стадии комплексного тестирования и испытаний ПС в реальном времени:*
– по данным моделирующего стенда или генераторов тестов, имитирующих отдельные объекты внешней среды;
– с имитаторами отдельных объектов внешней среды и с реальными воздействиями от операторов–пользователей;
– в полностью адекватной реальной или имитированной внешней среде и с реальными воздействиями от операторов–пользователей.

На всех стадиях отладки, кроме операций непосредственной проверки функционирования программ, можно выделить еще две важные группы работ. *Первая группа* – это работы по методическому обеспечению тестирования и по созданию средств автоматизированной генерации тестов. *Вторая группа* работ

должна обеспечивать возможность обработки результатов тестирования и оценки достигнутых показателей качества функционирования программ.

Средства генерации тестов и обработки результатов отладки можно разделить на три вида (см. рисунок 12.2). Одни и те же средства автоматизации тестирования в статике обычно обеспечивают отладку групп программ как автономно, так и во взаимодействии с другими компонентами. Средства, имитирующие внешнюю среду в реальном времени, чаще всего ориентированы на отладку как функциональных компонентов, так и ПС в целом. Еще один вид генераторов тестов в той или иной степени использует реальные объекты внешней среды. Первоначально такими объектами являются имитирующие стенды с участием реального функционирования операторов–пользователей. Затем источниками тестов могут быть полные комплексы реальной аппаратуры внешних объектов или их аппаратные аналоги.

ГГУ ИМЕНИ Ф. СКО

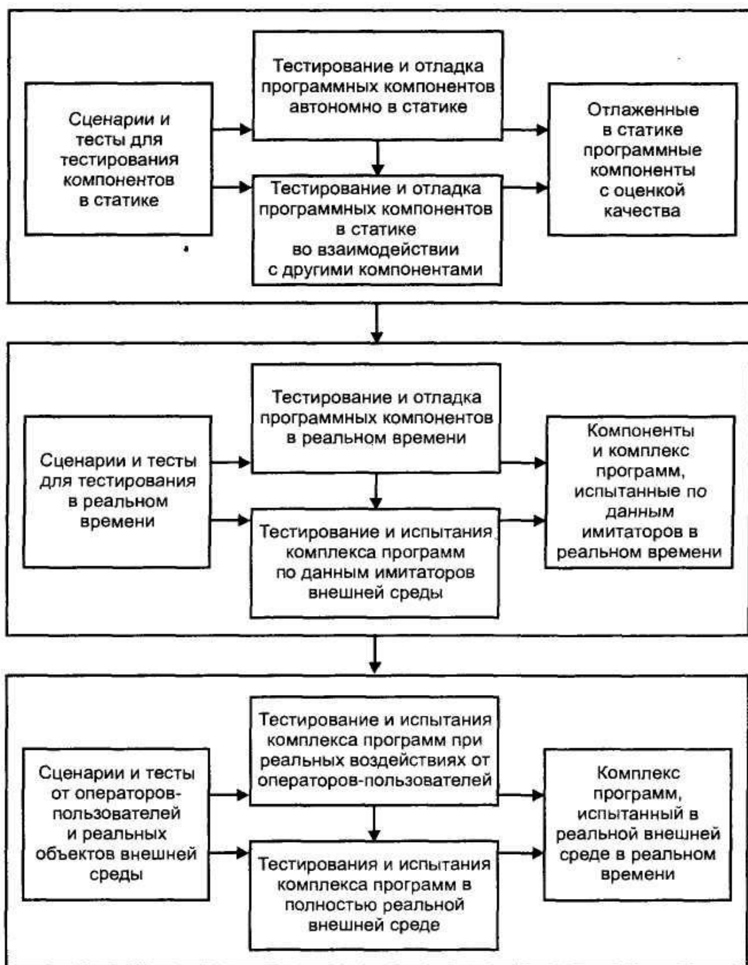

Рисунок. 12.2 – Схема этапов тестирования и испытаний сложных комплексов программ

Каждая из выделенных стадий тестирования может рассматриваться как обеспечивающая создание определенного промежуточного продукта – функциональной группы программ или программного средства с некоторыми ограниченными характеристиками качества. Эти характеристики выделяются и детализируются на основе первичного технического задания и спецификации требований на ПС. В процессе проектирования ПС они уточняются и конкретизируются в спецификациях требований на группы программ и их компоненты. В

результате создается совокупность эталонов, имеющих последовательно расширяющиеся номенклатуру и наборы значений показателей качества, которым должны соответствовать отлаживаемые и испытываемые компоненты на каждой стадии тестирования.

Подобная совокупность эталонных показателей качества промежуточных продуктов обеспечивает возможность управления процессом тестирования с целью достижения необходимого качества конечного продукта – ПС при минимальных или разумных затратах. Для этого каждая стадия тестирования должна иметь фиксированный и документированный результат с определенными эталонами и достигнутыми характеристиками программ. Подобный систематизированный контроль тестирования гарантирует высокое качество ПС реального времени, к которым предъявляются обычно высокие требования по надежности. Кроме того, компоненты, прошедшие все стадии тестирования, с большой уверенностью могут применяться как повторно используемые компоненты в последующих версиях ПС.

Организация завершающих испытаний комплексов программ. Испытания главного конструктора, которые зачастую совмещаются с завершением комплексной отладки, должны оформляться документально и являются основанием для предъявления ПС заказчику на завершающиеся совместные испытания. Любые испытания ограничены допустимым объемом проверок и длительностью работы комиссии, поэтому не могут гарантировать абсолютную проверку изделия. Для повышения достоверности определения и улучшения характеристик ПС после испытаний главного конструктора программы целесообразно передавать некоторым пользователям на *опытную эксплуатацию в типовых условиях.* Это позволяет более глубоко оценить эксплуатационные характеристики созданного комплекса и устранить некоторые дефекты и ошибки. Опытная эксплуатация проводится разработчиками с участием испытателей и некоторых пользователей, назначаемых заказчиком. Результаты и показатели надежности опытной эксплуатации после испытаний главного конструктора могут учитываться при проведении совместных испытаний для их сокращения.

Совместные приемо–сдаточные испытания проводятся комиссией заказчика, в которой участвуют главный конструктор разработки и некоторые ведущие разработчики, аттестованные сертификационной лабораторией. *Комиссия при испытании руководствуется следующими документами:*

– утвержденным заказчиком и согласованным с разработчиком техническим заданием и спецификациями на ПС;
– действующими государственными и ведомственными стандартами на проектирование и испытания программ и на техническую документацию, а также согласованными с заказчиком стандартами «де–факто»;
– программой испытаний по всем требованиям технического задания;
– методиками испытаний по каждому разделу требований технического задания;

– комплектом эксплуатационной документации на комплекс про грамм.

Программа испытаний является планом проведения серии экспериментов и разрабатывается с позиции минимизации объема тестирования в процессе проведения испытаний для проверки выполнения требований технического задания и соответствия предъявленной документации. Программа испытаний, методики их проведения и оценки результатов, разработанные совместно заказчиком и разработчиком, должны быть согласованы и утверждены. Они должны содержать уточнения и детализацию требований технического задания для данного ПС, а также гарантировать корректную проверку всех заданных характеристик, в том числе надежности. *Программа испытаний должна содержать следующие четко сформулированные разделы:*

– объект испытаний, его назначение и перечень основных документов, определивших его разработку;
– цель испытаний с указанием всех требований технического задания, подлежащих проверке, и ограничений на проведение испытаний;
– собственно программу испытаний, содержащую проверку комплектности спроектированного ПС в соответствии с техническим заданием, и план тестирования для проверки по всем разделам технического задания и дополнительным требованиям, формализованным отдельными решениями разработчиков и заказчика;
– методики испытаний, однозначно определяющие все понятия проверяемых характеристик, условия и сценарии тестирования, средства, используемые для испытаний;
– методики обработки и оценки результатов тестирования по каждому разделу программы испытаний.

Большой объем разнородных данных, получаемых при испытаниях крупномасштабных ПС, и разнообразие возможных способов их обработки, интерпретации и оценки приводят к тому, что важнейшими факторами становятся *методики обработки и оценки результатов, а также протоколы проверки по пунктам программы испытаний.* В соответствии с методиками испытаний средства автоматизации должны обеспечивать всю полноту проверок характеристик по каждому разделу методик. Результаты испытаний фиксируются в протоколах, которые обычно содержат следующие разделы:

– назначение тестирования и раздел требований технического задания, по которому проводились испытания;
– указания методик, в соответствии с которыми проводились испытания, обработка и оценка результатов;
– условия и сценарии проведения тестирования и характеристики исходных данных;
– обобщенные результаты испытаний с оценкой их на соответствие требованиям технического задания и другим руководящим документам, а также технической документации;

– выводы о результатах испытаний и соответствии созданного ПС определенному разделу требований технического задания.

Протоколы по всей программе испытаний *обобщаются в акте,* в результате чего делается заключение о соответствии системы требованиям заказчика и завершении работы с положительным или отрицательным итогом. При полном выполнении всех требований технического задания заказчик обязан принять систему, и работа считается завершенной.

Наиболее полным и разносторонним испытаниям должна подвергаться первая базовая версия ПС. При *испытаниях очередных модернизированных версий ПС* возможны значительные сокращения объемов тестирования повторно используемых компонентов. Однако комплексные и завершающие испытания каждой новой версии ПС, как правило, проводятся в полном объеме, гарантирующем проверку выполнения всех требований измененного технического задания. Для обеспечения выявления дефектов в процессе эксплуатации серийных образцов в каждом из них должен быть предусмотрен некоторый минимум средств проверки функционирования и обнаружения искажений результатов. Этот минимум средств должен позволять фиксировать условия неправильной работы программ и характер проявления дефектов. Последующее исправление ошибок должно проводиться специалистами, осуществляющими сопровождение.

При завершающих испытаниях основное внимание, кроме проверок функциональной пригодности, должно сосредоточиваться на подготовке стрессовых тестов, тестировании в режимах предельного использования ресурсов, надежности функционирования ПС. Задача испытателей и заказчика при проведении совместных испытаний состоит в выделении условий и области изменения переменных, которые недостаточно проверены разработчиком и важны для последующего надежного функционирования программ. При этом разработчик контролирует, чтобы планируемые сценарии и тесты не выходили из областей, заданных техническим заданием и спецификацией требований. Испытания за пределами технического задания могут квалифицироваться как его расширение или могут исключаться по требованию разработчика.

До начала испытаний подлежат проверке и паспортизации средства, обеспечивающие получение эталонных данных, средства имитации тестов от внешних объектов, средства фиксирования и обработки результатов тестирования. При испытаниях важную роль играют оценка и обеспечение близких значений методической и статистической достоверности результатов испытаний. *Методическая достоверность приемо–сдаточных испытаний ПС определяется следующими факторами:*

– полнотой программы испытаний и корректностью методик тестирования по охвату возможных условий и сценариев функционирования программ и областей изменения исходных данных;

– достоверностью и точностью эталонных значений, с которыми сравниваются результаты тестирования испытываемой про граммы или которые служат

опорными при расчете параметров, зафиксированных в техническом задании;
– адекватностью и точностью моделей, используемых для имитации тестов от внешней среды и подыгрыша их реакции на управляющие воздействия;
– точностью и корректностью регистрации и обработки результатов тестирования, а также сравнения полученных данных с требованиями технического задания.

Представленная выше организация испытаний сложных ПС ориентирована на наличие конкретного заказчика комплекса программ и ограниченное число пользователей, контролируемых заказчиком. Несколько иначе организуются испытания коммерческих пакетов прикладных программ, создаваемых по инициативе разработчиков для широкого круга пользователей при отсутствии конкретного заказчика. Для таких коммерческих прикладных программ принято проводить испытания в два последовательных этапа – *Альфа– и Бета–тестирование.* Испытания проводятся на соответствие критериям, формализованным руководителем проекта. Они заключаются в нормальной и форсированной (стрессовой) опытной эксплуатации конечными пользователями оформленного программного продукта в соответствии с сопроводительной документацией и различаются количеством участвующих пользователей.

При Альфа–тестировании привлекаются конечные пользователи, работающие в той же компании, но не участвовавшие непосредственно в разработке комплекса программ. Для Бета–тестирования привлекаются добровольные пользователи (потенциальные покупатели), которым бесплатно передается версия ПС для опытной эксплуатации. При этом особое значение имеет выделение компетентных, тщательных и доброжелательных пользователей, способных своими рекомендациями улучшить качество испытываемых программ. Их деятельность стимулируется бесплатным и ранним получением и освоением нового программного продукта и собственной оценкой его качества. Эти пользователи обязуются сообщать разработчикам сведения о всех выявленных дефектах и ошибках, а также вносить изменения в программы и данные или заменять версии по указаниям разработчиков. Только после успешной эксплуатации и Бета–тестирования ограниченным контингентом пользователей, руководителем проекта или фирмы разработчиков принимается решение о передаче ПС в продажу для широкого круга пользователей. Обобщение результатов Бета–тестирования может использоваться как часть или основа сертификационных испытаний.

При Альфа– и Бета–испытаниях принято разделять прогрессивное и регрессивное тестирование. Под *прогрессивным* понимается тестирование новых программных компонентов, для выявления дефектов и ошибок в исходных текстах программ и спецификациях. *Регрессивное* тестирование предназначено для контроля качества и корректности изменении в программах и данных после проведения корректировок. Необходимость и широта регрессивного тестирования определяются тем, что значительная доля изменений после Альфа– и Бета–

тестирования, в свою очередь, содержит ошибки. Объем тестов и длительность обоих этапов тестирования определяются руководителями проекта в зависимости от сложности комплекса программ и интенсивности потока изменений.

Тема 13 Тестирование программного обеспечения

13.1 Тестирование программного обеспечения (ПО)
13.2 Место и цель этапа тестирования ПО
13.3 Виды тестирования
13.4 Передовые технологии в тестировании (автоматизация тестирования).

13.1 Тестирование программного обеспечения

На современном этапе развития информационных технологий ПО характеризуется большой степенью сложности. Особенностью ПО, разрабатываемого для сферы экономики, является то, что оно постоянно изменяется. Связано это, прежде всего с изменениями, происходящими в предметной области (введение новых услуг, бизнес–процессов, изменение законодательства).

Создание и поддержка банка тестов – сложная задача и требует высокой квалификации сотрудников отдела тестирования. За все надо платить, но качество конечного продукта того стоит. Тестирование – это дорогостоящий и трудоемкий процесс, поэтому зарубежными компаниями ведутся разработки в области автоматизации тестирования. Попытки применения автоматизации тестирования связаны с тем, что в принципе невозможно полностью протестировать программный продукт, соответственно специализированные пакеты приближают «покрытие» тестами программы к 100%. На рынке специальных сред для тестирования программного обеспечения можно отметить разработки ведущих в этой области фирм: Rational (Visual Test, Rational Robot, Team Test и др.), Mercury Interactive (WinRunner), Segue Software (QA Partner).

Сегодня можно и нужно говорить о программном обеспечении как о промышленном продукте, соответственно о создании ПО – как о производстве. Существует множество стандартов поддержки жизненного цикла программного обеспечения. Разработано множество стандартов и методик поддержки стадий ЖЦ ПО, например стандарты ISO 9000 и ISO 9001, разработанные Международной организацией по стандартизации (ISO).

13.2 Место и цель этапа тестирования программного обеспечения

На сегодняшний день автоматизировано большинство этапов разработки программного обеспечения, в том числе и этап тестирования. Однако в отечественной практике тестированию программных средств отведена незаслуженно

маленькая роль. Причинами могут быть отсутствие денег на приобретение дорогостоящих CASE–средств, поддерживающих все этапы разработки ПО, в том числе тестирование, или нежелание держать такую штатную единицу, как специалист по тестированию. Обычно приобретают средства автоматизации проектирования (создание ER–моделей, информационных моделей и пр.) и программирования (автоматическое создание БД на основе ER–модели, создание интерфейса на основе ER–модели, визуальное программирование). С тестированием дела обстоят сложнее.

Тестирование занимает важное место в жизненном цикле программного обеспечения, это трудоемкий и дорогостоящий процесс. В организационной структуре современной фирмы – разработчика ПО должен быть отдел по тестированию программного обеспечения или специалист по тестированию программного обеспечения, так как одним из принципов тестирования является избежание тестирования автором, а тестирование программного средства напрямую связано с его качеством.

Отечественные производители коммерческого программного обеспечения только сейчас серьезно задумались о качестве своей продукции и, как следствие, о тестировании.

В качестве объективных причин, почему это происходит именно сейчас и почему этого не было раньше, можно выделить следующие.

1 Сформировалась нормативно–правовая база для осуществления деятельности по разработке программного обеспечения, база в области авторского права и смежных прав, а также защиты прав потребителей.

2 Законодательно закрепленные принципы работают и активно используются в правоприминительной практике. Особенно это относится к защите авторских прав.

3 Исполнительная власть начала пресекать попытки распространения нелицензионного программного обеспечения. Есть прецеденты процессов, по которым возмещаются потери от нарушения авторских прав.

4 Возросли требования заказчика к качеству программного обеспечения. Данный момент связан с правовой базой (законы о защите прав потребителей), жесткой конкуренцией на рынке.

5 Накоплен большой опыт создания программных средств, российские компании выходят на другие рынки (в том числе на мировой рынок), что влечет необходимость выполнения новых норм по качеству программных средств.

По этим причинам процессы обеспечения и контроля качества, одним из которых является тестирование, приобретают в настоящий момент большое значение и актуальность. Целью тестирования является обнаружение максимального количества ошибок, а не всех ошибок в программе. Обнаружение всех ошибок невозможно. Полное и абсолютное тестирование выглядит скорее мечтой, чем реальностью. Вот простой пример: еще в 1979 г. Майерс (Myers) описал некоторый простой алгоритм. В нем был всего один цикл и несколько операто-

ров условного перехода. В большинстве языков программирования для кодирования такого алгоритма потребуется не более 20 строк кода. Но такая программа имеет более 100 триллионов путей выполнения! Самому быстрому тестировщику *для* полного тестирования потребовался бы как минимум миллион лет. Аналогичная программа из 100 строк имеет 10^{18} триллионов путей выполнения, если на проверку выполнения одного пути тратить 1 секунду, то на полное ее завершение не хватит времени существования всей нашей вселенной, время жизни которой меньше $4 \cdot 10^{17}$ секунд!

Таким образом, целью тестирования является не тотальное обнаружение всех ошибок (это принципиально невозможно), а выявление наибольшего количества наиболее критичных ошибок. Если исправление их задерживается, то пользователи программного продукта должны быть предупреждены о наличии такого рода ошибок и рекомендуемых путях обхода.

Если процесс тестирования становится бесконечным, а полное тестирование невозможно, то чем же определяется принятие решения о выпуске в свет исследуемой версии программного продукта? Основными критериями завершенности тестирования является **отсутствие критичных ошибок**, каждая из которых может сделать абсолютно невозможной реализацию декларированной в системе прикладной функциональности (решение принимается по результатам функционального тестирования). Кроме того, при принятии решения учитывается общее количество зарегистрированных, но неисправленных ошибок. Компания–разработчик обычно заранее выбирает по каждому программному продукту общее количество ошибок (лимит), с которым уже нельзя выпускать программный продукт.

Количественная оценка завершенности процесса тестирования и готовности программного продукта для эксплуатации может быть получена при помощи моделей надежности программного обеспечения. Самый простой способ представления информации для принятия решения – графический: по одной оси откладывается время от начала процесса тестирования, по другой – количество обнаруженных ошибок в программном средстве. По графику (знаку производной) определяется необходимость продолжения тестирования. Существует множество методов, которые помогают принять решение в выпуске программного обеспечения, однако самое, веское слово остается за специалистом, осуществляющим тестирование программного обеспечения, так как на основе количества и характера найденных проблем он может судить о том, удовлетворит данный продукт потребности и ожидания пользователя или нет.

Тестирование программного обеспечения имеет тесную связь с качеством программного обеспечения.

Внутреннее и внешнее качество программного обеспечения

Современные идеологи проблем качества разделяют понятие «качество» на внешнее (external) и внутреннее (internal). Внешнее качество программного обеспечения – его способность удовлетворить потребность конечного пользователя.

Именно на это и направлен процесс тестирования программного обеспечения – обнаружение ошибок и несоответствий, т.е. в процессе тестирования выявляются те моменты (ошибки, неправильная реализация или отсутствие функциональных возможностей), которые не удовлетворили бы конечного пользователя. Тестирование программного обеспечения обеспечивает контроль качества продукта, поставляемого конечным пользователям.

Внутреннее качество программного обеспечения связано с удобством его производства для тех, кто его производит, с его технологичностью, стандартизованностью, безопасностью. Вопросы внутреннего качества в большей степени связаны с реализацией процессов жизненного цикла программного обеспечения, с процессами управления разработкой программного обеспечения. Улучшая документированность тестов, их более простую адаптируемость от версии к версии программного обеспечения, специалист по тестированию улучшает внутреннее качество программного обеспечения.

13.3 Виды тестирования

Наиболее важные виды тестирования программных средств перечислены ниже.

1 *Функциональное* – тестирование возможностей системы, ее реакция на те или иные ситуации. Обычно результат тестирования (реакция системы) сравнивается с постановкой задачи, при несоответствии фиксируется ошибка.

2 *Регрессионное* – проверка полноты реализуемых функций системы по сравнению с предыдущей версией программного продукта.

3 *Нагрузочное* – тестирование работы системы на пиковую на грузку, при этом делается вывод о производительности системы. Например, выясняется среднее время ввода одного документа (если программное обеспечение предназначено для хранения и обработки документов). Условием для нагрузочного тестирования является выполнение испытаний на одной и той же конфигурации системы. Если тестируется производительность на 2–х разных СУБД, то конфигурация системы должна быть идентичной (тот же сервер, те же рабочие станции), в испытаниях меняются лишь СУБД. На основе нагрузочного тестирования выдвигаются требования к аппаратной части и программной части системы (операционная система, СУБД).

4 *Контроль после исправления* (обратная связь). Этот вид тестирования подразумевает под собой проверку уже исправленных ошибок.

5 *Стрессовое тестирование* – проверка реакции системы на вне штатные ситуации. Примером может служить проверка системы на восстановление работоспособности после отключения питания на сервере базы данных.

6 *Адаптационное тестирование* – проверка корректности пере вода программного обеспечения на другой национальный язык.

13.4 Передовые технологии в тестировании (автоматизация тестирования)

При контроле качества, лучшие результаты дает использование автоматических тестов с применением специальных промышленных средств автоматизации тестирования. Высокую эффективность имеют также специальные тестовые процедуры, написанные на языке программирования, на котором написано само ПС. Преимущества использования автоматических тестов перед тестированием очевидны: они беспристрастны; позволяют выполнять проверку необходимое количество раз; не устают и не ошибаются; могут протестировать гораздо больше за меньшее количество времени; не требуют дополнительной оплаты при работе по ночам и выходным; более четко отвечают на вопрос, что протестировано и с каким результатом.

Создание и поддержка банка тестов сложная задача, требующая высокой квалификации сотрудников отдела тестирования. Автоматизация тестирования связана с тем, что в принципе невозможно полностью протестировать программный продукт, соответственно специализированные пакеты приближают «покрытие» тестами программы к 100%. На рынке специальных сред для тестирования программного обеспечения можно отметить разработки ведущих в этой области фирм: Rational (Visual Test, Rational Robot, Team Test и др.), Mercury Interactive (WinRunner), Segue Software (QA Partner). Такое ПО весьма специфично и имеет достаточно высокую цену – порядка нескольких десятков тысяч долларов. Пакеты тестирования можно разделить по поддерживаемой стратегии тестирования на пакеты, поддерживающие стратегию «белого ящика» и на поддерживающие стратегию «черного ящика».

Пакеты, реализующие стратегию «белого ящика», позволяют: записывать, а потом воспроизводить последовательность пользовательского ввода (нажатие клавиатуры, движения «мышью»); распознавать объекты и их свойства (окна Windows, текст в окне и пр.); запоминать копию экрана; сравнивать состояние программы относительно предыдущего тестового прогона; производить математические вычисления на основе данных из тестируемой программы; замерять выполнение одной и той же последовательности действий в различных условиях; эмулировать выполнение программы несколькими пользователями одновременно; записывать подробный протокол выполнения автоматического теста; другие функции.

Пакеты, реализующие стратегию «черного ящика», позволяют: отслеживать выполнение того или иного фрагмента кода программы; подсчитывать количество выполнения того или иного фрагмента кода программы; вычислять время выполнения участка кода (важно при пересмотрах кода и его оптимизации); подсчитывать общее «покрытие» программы; автоматически контролировать значение переменных и выдавать ошибку или предупреждение, если значения не совпадают с теми, которые ожидаются; на основе данных, по-

лученных от пакета автоматизации тестирования, возможно выполнять расчеты о надежности программного обеспечения; получать различные статистические данные о программе.

Средства автоматизации тестирования не предполагают отсутствие инженера по тестированию, а требуют от него новых знаний. Программа автоматизации тестов не выполнит всю работу по тестированию сама. Для нее нужны специальные инструкции – сценарии тестов, написанные на специально разработанном языке. Таким образом, автоматизация заключается в избавлении инженера по тестированию от рутинной работы, теперь тестер занимается разработкой тестов и программированием тестов на языке системы автоматизации тестирования.

Тема 14 Виды тестирования программного обеспечения

14.1 Функциональные виды тестирования
14.2 Нефункциональные виды тестирования. Тестирование производительности
14.3. Связанные с изменениями виды тестирования
14.4 Тестирование удобства пользования
14.5 Тестирование на отказ и восстановление
14.6 Конфигурационное тестирование

14.1 Функциональные виды тестирования

Все **виды тестирования программного обеспечения**, в зависимости от преследуемых целей, можно условно разделить на следующие группы: 1) функциональные; 2) нефункциональные; 3) связанные с изменениями.

Функциональные тесты базируются на функциях и особенностях, а также взаимодействии с другими системами, и могут быть представлены на всех уровнях тестирования: компонентном или модульном (Component/Unit testing), интеграционном (Integration testing), системном (System testing) и приемочном (Acceptance testing). Функциональные виды тестирования рассматривают внешнее поведение системы. Далее перечислены самые распространенные виды функциональных тестов:

– **Функциональное тестирование** (Functional testing)
– **Тестирование безопасности** (Security and Access Control Testing)
– **Тестирование взаимодействия** (Interoperability Testing)

Функциональное тестирование. Этот вид тестирования проверяет соответствие реализованных функций требованиям, техническому заданию, спецификациям, различным другим проектным документам и просто ожиданиям пользователя. Проверяется каждая из функций приложения и все они в комплексе. Исследуются все сценарии использования. Проверяется адекватность хранимых и

выходных данных, методы их обработки, обработка вводимых данных, методы хранения данных, методы импорта и экспорта данных и т.д. в зависимости от специфики приложения.

Функциональные тесты основываются на функциях, выполняемых системой, и могут проводиться на всех уровнях тестирования (компонентном, интеграционном, системном, приемочном). Как правило, эти функции описываются в требованиях, функциональных спецификациях или в виде случаев использования системы (use cases).

Тестирование функциональности может проводиться в двух аспектах: «**требования**»; «**бизнес–процессы**».

Тестирование в перспективе «**требования**» использует спецификацию функциональных требований к системе как основу для дизайна тестовых случаев (Test Cases). В этом случае необходимо сделать список того, что будет тестироваться, а что нет, приоритезировать требования на основе рисков (если это не сделано в документе с требованиями), а на основе этого приоритезировать тестовые сценарии (test cases). Это позволит сфокусироваться и не упустить при тестировании наиболее важный функционал.

Тестирование в перспективе «**бизнес–процессы**» использует знание этих самых бизнес–процессов, которые описывают сценарии ежедневного использования системы. В этой перспективе тестовые сценарии (test scripts), как правило, основываются на случаях использования системы (use cases).

Преимущества функционального тестирования: имитирует фактическое использование системы. **Недостатки** функционального тестирования: возможность упущения логических ошибок в программном обеспечении; вероятность избыточного тестирования.

Достаточно распространенной является **автоматизация функционального тестирования**.

Тестирование безопасности. Стратегия тестирования, используемая для проверки безопасности системы, а также для анализа рисков, связанных с обеспечением целостного подхода к защите приложения, атак хакеров, вирусов, несанкционированного доступа к конфиденциальным данным. Тестирование безопасности может выполняться как автоматизированно так и в ручную, включая проверку как позитивных, так и негативных тестовых случаев. Основывается на трех основных <u>принципах</u> – это **конфиденциальность, целостность и доступность** (confidentiality, integrity, availability)

Конфиденциальность – это сокрытие определенных ресурсов или информации. Под конфиденциальностью можно понимать ограничение доступа к ресурсу некоторой категории пользователей, или другими словами, при каких условиях пользователь авторизован получить доступ к данному ресурсу.

Существует два основных критерия при определении понятия **целостности**:

1. Доверие. Ожидается, что ресурс будет изменен только соответствующим способом определенной группой пользователей.

ГГУ ИМЕНИ Ф. СКОРИНЫ

2. Повреждение и восстановление. В случае, когда данные повреждаются или неправильно меняются авторизованным или не авторизованным пользователем, необходимо определить на сколько важной является процедура восстановления данных.

Доступность представляет собой требования о том, что ресурсы должны быть доступны авторизованному пользователю, внутреннему объекту или устройству. Как правило, чем более критичен ресурс, тем выше уровень доступности должен быть.

Тестирование взаимодействия. С развитием сетевых технологий и интернета взаимодействие разных систем, сервисов и приложений друг с другом приобрело значительную актуальность, так как любые связанные с этим проблемы могут привести к падению авторитета компании, что как следствие повлечет за собой финансовые потери. Поэтому к тестированию взаимодействия стоит подходить со всей серьезностью.

Тестирование взаимодействия – это функциональное тестирование, проверяющее способность приложения взаимодействовать с одним и более компонентами или системами и включающее в себя тестирование совместимости (compatibility testing) и интеграционное тестирование (integration testing).

Программное обеспечение с хорошими характеристиками взаимодействия может быть легко интегрировано с другими системами, не требуя каких–либо серьезных модификаций. В этом случае, количество изменений и время, требуемое на их выполнение, могут быть использованы для измерения возможности взаимодействия.

14.2 Нефункциональные виды тестирования. Тестирование производительности

Нефункциональное тестирование описывает тесты, необходимые для определения характеристик программного обеспечения, которые могут быть измерены различными величинами. В целом, это тестирование того, «Как» система работает. Далее перечислены основные виды нефункциональных тестов:

- **Тестирование производительности**:
  (a) нагрузочное тестирование (Performance and Load Testing)
  (b) стрессовое тестирование (Stress Testing)
  (c) тестирование стабильности или надежности (Stability / Reliability Testing)
  (d) объемное тестирование (Volume Testing)
- **Тестирование установки** (Installation testing)
- **Тестирование удобства пользования** (Usability Testing)
- **Тестирование на отказ и восстановление** (Failover and Recovery Testing)
- **Конфигурационное тестирование** (Configuration Testing)

ГГУ ИМЕНИ Ф. СКОРИНЫ

Нагрузочное тестирование или тестирование производительности. Задачей тестирования производительности является определение масштабируемости приложения под нагрузкой, при этом происходит:

– измерение времени выполнения выбранных операций при определенных интенсивностях выполнения этих операций;

– определение количества пользователей, одновременно работающих с приложением;

– определение границ приемлемой производительности при увеличении нагрузки (при увеличении интенсивности выполнения этих операций);

– исследование производительности на высоких, предельных, стрессовых нагрузках.

Стрессовое тестирование позволяет проверить насколько приложение и система в целом работоспособны в условиях стресса и также оценить способность системы к регенерации, т.е. к возвращению к нормальному состоянию после прекращения воздействия стресса. Стрессом в данном контексте может быть повышение интенсивности выполнения операций до очень высоких значений или аварийное изменение конфигурации сервера. Также одной из задач при стрессовом тестировании может быть оценка деградации производительности, таким образом цели стрессового тестирования могут пересекаться с целями тестирования производительности.

Задачей **объемного тестирования** является получение оценки производительности при увеличении объемов данных в базе данных приложения, при этом происходит:

– измерение времени выполнения выбранных операций при определенных интенсивностях выполнения этих операций

– может производиться определение количества пользователей, одновременно работающих с приложением

Задачей **тестирования стабильности** (надежности) является проверка работоспособности приложения при длительном (многочасовом) тестировании со средним уровнем нагрузки. Времена выполнения операций могут играть в данном виде тестирования второстепенную роль. При этом на первое место выходит отсутствие утечек памяти, перезапусков серверов под нагрузкой и другие аспекты, влияющие именно на стабильность работы.

14.3 Связанные с изменениями виды тестирования

После проведения необходимых изменений, таких как исправление бага/дефекта, программное обеспечение должно быть перетестировано для подтверждения того факта, что проблема была действительно решена. Ниже перечислены виды тестирования, которые необходимо проводить после установки

ГГУ ИМЕНИ Ф. СКОРИНЫ

программного обеспечения, для подтверждения работоспособности приложения или правильности осуществленного исправления дефекта:

– **Дымовое тестирование** (Smoke Testing)

– **Регрессионное тестирование** (Regression Testing)

– **Тестирование сборки** (Build Verification Test)

– **Санитарное тестирование или проверка согласованности/исправности** (Sanity Testing)

Понятие **дымовое тестирование** пошло из инженерной среды. При вводе в эксплуатацию нового оборудования («железа») считалось, что тестирование прошло удачно, если из установки не пошел дым. В области же тестирования программного обеспечения, оно направлено на поверхностную проверку всех модулей приложения на предмет работоспособности и наличие быстро находимых критических и блокирующих дефектов. По результатам дымового тестирования делается вывод о том, принимается или нет установленная версия программного обеспечения в тестирование, эксплуатацию или на поставку заказчику. Для облегчения работы, экономии времени и людских ресурсов рекомендуется внедрить автоматизацию тестовых сценариев для дымового тестирования.

Регрессионное тестирование – это вид тестирования, направленный на проверку изменений, сделанных в приложении или окружающей среде (починка дефекта, слияние кода, миграция на другую операционную систему, базу данных, веб сервер или сервер приложения), для подтверждения того факта, что существующая ранее функциональность работает как и прежде (см. также Санитарное тестирование или проверка согласованности/исправности). Регрессионными могут быть как **функциональные,** так и **нефункциональные** тесты.

Как правило, для регрессионного тестирования используются тест кейсы, написанные на ранних стадиях разработки и тестирования. Это дает гарантию того, что изменения в новой версии приложения не повредили уже существующую функциональность. Рекомендуется делать автоматизацию регрессионных тестов, для ускорения последующего процесса тестирования и обнаружения дефектов на ранних стадиях разработки программного обеспечения.

Сам по себе термин «Регрессионное тестирование», в зависимости от контекста использования может иметь разный смысл. Сэм Канер, к примеру, описал **3 основных типа** регрессионного тестирования:

– **Регрессия багов (Bug regression)** – попытка доказать, что исправленная ошибка на самом деле не исправлена.

– **Регрессия старых багов (Old bugs regression)** – попытка доказать, что недавнее изменение кода или данных сломало исправление старых ошибок, т.е. старые баги стали снова воспроизводиться.

ГГУ ИМЕНИ Ф. СКОРИНЫ

– **Регрессия побочного эффекта (Side effect regression)** – попытка доказать, что недавнее изменение кода или данных сломало другие части разрабатываемого приложения.

Санитарное тестирование или проверка согласованности/исправности (Sanity Testing) – это узконаправленное тестирование, достаточное для доказательства того, что конкретная функция работает согласно заявленным в спецификации требованиям. Является подмножеством регрессионного тестирования. Используется для определения работоспособности определенной части приложения после изменений произведенных в ней или окружающей среде. Обычно выполняется вручную.

Отличие санитарного тестирования от дымового. В некоторых источниках ошибочно полагают, что санитарное и дымовое тестирование – это одно и тоже. Мы же полагаем, что эти виды тестирования имеют «вектора движения», направления в разные стороны. В отличии от дымового (Smoke testing), санитарное тестирование (Sanity testing) направлено вглубь проверяемой функции, в то время как дымовое направлено вширь, для покрытия тестами как можно большего функционала в кратчайшие сроки.

Тестирование сборки (Build Verification Test) – это тестирование, направленное на определение соответствия, выпущенной версии, критериям качества для начала тестирования. По своим целям является аналогом Дымового Тестирования, направленного на приемку новой версии в дальнейшее тестирование или эксплуатацию. Вглубь оно может проникать дальше, в зависимости от требований к качеству выпущенной версии.

Тестирование Установки (Installation Testing) – направленно на проверку успешной инсталляции и настройки, а также обновления или удаления программного обеспечения. В настоящий момент наиболее распространена установка ПО при помощи **инсталляторов** (специальных программ, которые сами по себе так же требуют надлежащего тестирования). В реальных условиях инсталляторов может не быть. В этом случае придется самостоятельно выполнять установку программного обеспечения, используя документацию в виде инструкций или readme файлов, шаг за шагом описывающих все необходимые действия и проверки. В распределенных системах, где приложение разворачивается на уже работающем окружении, простого набора инструкций может быть мало. Для этого, зачастую, пишется план установки (Deployment Plan), включающий не только шаги по инсталляции приложения, но и шаги отката (roll–back) к предыдущей версии, в случае неудачи. Сам по себе план установки также должен пройти процедуру тестирования для избежания проблем при выдаче в реальную эксплуатацию. Особенно это актуально, если установка выполняется на системы, где каждая минута простоя – это потеря репутации и большого количества средств, например: банки, финансовые компании или даже баннерные сети. Поэтому тестирование установки можно назвать одной из важнейших задач по обеспечению качества программного обеспечения.

ГГУ ИМЕНИ Ф. СКОРИНЫ

Именно такой комплексный подход с написанием планов, пошаговой проверкой установки и отката инсталляции, полноправно можно назвать тестированием установки или Installation Testing.

14.4 Тестирование удобства пользования

Тестирование удобства пользования (Usability Testing**).** Иногда мы сталкиваемся с непонятными, нелогичными приложениями, многие функции и способы использования которых часто не очевидны. После такой работы редко возникает желание использовать приложение снова, и мы ищем более удобные аналоги. Для того чтобы приложение было популярным, ему мало быть функциональным – оно должно быть еще и удобным. Если задуматься, интуитивно понятные приложения экономят нервы пользователям и затраты работодателя на обучение. А значит они более конкурентоспособные! Поэтому тестирование удобства использования, о котором пойдет речь далее является неотъемлемой частью тестирования любых массовых продуктов.

Тестирование удобства пользования – это метод тестирования, направленный на установление степени удобства использования, обучаемости, понятности и привлекательности для пользователей разрабатываемого продукта в контексте заданных условий. [ISO 9126]

Тестирование удобства пользования дает оценку уровня удобства использования приложения по следующим пунктам:

– **производительность, эффективность (efficiency)** – сколько времени и шагов понадобится пользователю для завершения основных задач приложения, например, размещение новости, регистрации, покупка и т.д.? (меньше – лучше);

– **правильность (accuracy)** – сколько ошибок сделал пользователь во время работы с приложением? (меньше – лучше);

– **активизация в памяти (recall)** – как много пользователь помнит о работе приложения после приостановки работы с ним на длительный период времени? (повторное выполнение операций после перерыва должно проходить быстрее чем у нового пользователя);

– **эмоциональная реакция (emotional response)** – как пользователь себя чувствует после завершения задачи – растерян, испытал стресс? Порекомендует ли пользователь систему своим друзьям? (положительная реакция – лучше).

Уровни проведения. Проверка удобства использования может проводиться как по отношению к готовому продукту, посредством тестирования черного ящика (black box testing), так и к интерфейсам приложения (API), используемым при разработке – тестирование белого ящика (white box testing). В этом случае проверяется удобство использования внутренних объектов, классов, методов и переменных, а также рассматривается удобство изменения, расширения системы и интеграции ее с другими модулями или системами. Использование удобных интерфейсов (API) может улучшить качество, увеличить скорость написания и

ГГУ ИМЕНИ Ф. СКОРИНЫ

поддержки разрабатываемого кода, и как следствие улучшить качество продукта в целом.

Отсюда становится, очевидно, что тестирование удобства пользования может производиться на разных уровнях разработки программного обеспечения: модульном, интеграционном, системном и приемочном. При этом оно целиком и полностью будет зависит от того, кто будет использовать приложение на выделенном конкретном уровне – разработчик, бизнес пользователь системы и т.д.

Советы по улучшению удобства пользования. Для дизайна удобных приложений полезно следовать принципам «пока–йока» или fail–safe. У нас это более известно как «защита от дурака». Простой пример, если поле требует цифровое значение, логично ограничить пользователю диапазон ввода только цифрами – будет меньше случайных ошибок. Для повышения юзабилити существующих приложений можно использовать цикл Демминга Plan–Do–Check–Act, собирая отзывы о работе и дизайне приложения у существующих пользователей, и, в соответствии с их замечаниями, планируя и проводя улучшения.

Заблуждения о тестировании удобства пользования

1. Тестирование пользовательского интерфейса = Тестирование удобства пользования. Тестирование удобства пользования не имеет ничего общего с тестированием функциональности пользовательского интерфейса, оно лишь проводится на пользовательском интерфейсе равно как и на многих других возможных компонентах продукта. При этом тип тестирования и тесткейсы будут совсем другие, так как речь может идти об удобстве использования не визуальных компонентов (если таковые имеются) или процессе администрирования, например, распределенного клиент–серверного продукта и т.д.

2. Тестирование удобства пользования можно провести без участия эксперта. Не всегда человек, не разбирающийся в предметной области, способен провести его самостоятельно. Представьте, что тестировщику нужно протестировать удобство пользования стратегического бомбардировщика. Ему придется проверить основные функции: удобство ведения боя, навигации, пилотирования, обслуживания, наземной транспортировки и т.д. Очевидно, что без привлечения эксперта это будет весьма проблематично, и можно даже сказать, что невозможно.

14.5 Тестирование на отказ и восстановление

Тестирование на отказ и восстановление (Failover and Recovery Testing) проверяет тестируемый продукт с точки зрения способности противостоять и успешно восстанавливаться после возможных сбоев, возникших в связи с ошибками программного обеспечения, отказами оборудования или проблемами связи (например, отказ сети). **Целью** данного вида тестирования является проверка систем восстановления (или дублирующих основной функционал систем), которые,

ГГУ ИМЕНИ Ф. СКОРИНЫ

в случае возникновения сбоев, обеспечат сохранность и целостность данных тестируемого продукта.

Тестирование на отказ и восстановление очень важно для систем, работающих по принципу “24x7”. Если Вы создаете продукт, который будет работать, например, в интернете, то без проведения данного вида тестирования Вам просто не обойтись. Т.к. каждая минута простоя или потеря данных в случае отказа оборудования, может стоить вам денег, потери клиентов и репутации на рынке.

Методика подобного тестирования заключается в симулировании различных условий сбоя и последующем изучении и оценке реакции защитных систем. В процессе подобных проверок выясняется, была ли достигнута требуемая степень восстановления системы после возникновения сбоя.

Для наглядности рассмотрим некоторые варианты подобного тестирования и общие методы их проведения. Объектом тестирования в большинстве случаев являются весьма вероятные эксплуатационные проблемы, такие как:

– Отказ электричества на компьютере–сервере

– Отказ электричества на компьютере–клиенте

– Незавершенные циклы обработки данных (прерывание работы фильтров данных, прерывание синхронизации).

– Объявление или внесение в массивы данных невозможных или ошибочных элементов.

– Отказ носителей данных.

Данные ситуации могут быть воспроизведены, как только достигнута некоторая точка в разработке, когда все системы восстановления или дублирования готовы выполнять свои функции. Технически реализовать тесты можно следующими путями:

– Симулировать внезапный отказ электричества на компьютере (обесточить компьютер).

– Симулировать потерю связи с сетью (выключить сетевой кабель, обесточить сетевое устройство)

– Симулировать отказ носителей (обесточить внешний носитель данных)

– Симулировать ситуацию наличия в системе неверных данных (специальный тестовый набор или база данных).

При достижении соответствующих условий сбоя и по результатам работы систем восстановления, можно оценить продукт с точки зрения тестирования на отказ. Во всех вышеперечисленных случаях, по завершении процедур восстановления, должно быть достигнуто определенное требуемое состояние данных продукта:

– Потеря или порча данных в допустимых пределах.

– Отчет или система отчетов с указанием процессов или транзакций, которые не были завершены в результате сбоя.

ГГУ ИМЕНИ Ф. СКОРИНЫ

Стоит заметить, что тестирование на отказ и восстановление – это весьма продукт–специфичное тестирование. Разработка тестовых сценариев должна производиться с учетом всех особенностей тестируемой системы. Принимая во внимание довольно жесткие методы воздействия, стоит также оценить целесообразность проведения данного вида тестирования для конкретного программного продукта.

14.6 Конфигурационное тестирование

Конфигурационное тестирование (Configuration Testing) — специальный вид тестирования, направленный на проверку работы программного обеспечения при различных конфигурациях системы (заявленных платформах, поддерживаемых драйверах, при различных конфигурациях компьютеров и т.д.)

В зависимости от типа проекта конфигурационное тестирование может иметь разные цели:

1. *Проект по профилированию работы системы*. Цель Тестирования: определить оптимальную конфигурацию оборудования, обеспечивающую требуемые характеристики производительности и времени реакции тестируемой системы.

2. *Проект по миграции системы с одной платформы на другую*. Цель Тестирования: Проверить объект тестирования на совместимость с объявленным в спецификации оборудованием, операционными системами и программными продуктами третьих фирм.

Уровни проведения тестирования. Для клиент–серверных приложений конфигурационное тестирование можно условно разделить на два уровня (для некоторых типов приложений может быть актуален только один): серверный или клиентский.

На первом (серверном) уровне, тестируется взаимодействие выпускаемого программного обеспечения с окружением, в которое оно будет установлено:

1. Аппаратные средства (тип и количество процессоров, объем памяти, характеристики сети / сетевых адаптеров и т.д.)

2. Программные средства (ОС, драйвера и библиотеки, стороннее ПО, влияющее на работу приложения и т.д.)

Основной упор здесь делается на тестирование с целью определения оптимальной конфигурации оборудования, удовлетворяющего требуемым характеристикам качества (эффективность, портативность, удобство сопровождения, надежность).

На следующем (клиентском) уровне, программное обеспечение тестируется с позиции его конечного пользователя и конфигурации его рабочей станции. На этом этапе будут протестированы следующие характеристики: удобство использования, функциональность. Для этого необходимо будет провести ряд тестов с различными конфигурациями рабочих станций:

ГГУ ИМЕНИ Ф. СКОРИНЫ

1 Тип, версия и битность операционной системы (подобный вид тестирования называется кросс–платформенное тестирование)

2 Тип и версия Web браузера, в случае если тестируется Web приложение (подобный вид тестирования называется кросс–браузерное тестирование)

3 Тип и модель видео адаптера (при тестировании игр это очень важно)

4 Работа приложения при различных разрешениях экрана

5 Версии драйверов, библиотек и т.д. (для JAVA приложений версия JAVA машины очень важна, тоже можно сказать и для .NET приложений касательно версии .NET библиотеки) и т.д.

Порядок проведения конфигурационного тестирования. Перед началом проведения конфигурационного тестирования рекомендуется:

– создавать матрицу покрытия (матрица покрытия – это таблица, в которую заносят все возможные конфигурации),

– проводить приоритезацию конфигураций (на практике, скорее всего, все желаемые конфигурации проверить не получится),

– шаг за шагом, в соответствии с расставленными приоритетами, проверяют каждую конфигурацию.

Уже на начальном этапе становится очевидно, что чем больше требований к работе приложения при различных конфигурациях рабочих станций, тем больше тестов необходимо будет провести. В связи с этим, рекомендуется автоматизировать этот процесс. Конечно же автоматизированное тестирование не является панацеей, но в данном случае оно окажется очень эффективным помощником.

ГГУ ИМЕНИ Ф. СКОРИНЫ

РАЗДЕЛ 6 CASE – ИНСТРУМЕНТАРИЙ АВТОМАТИЗАЦИИ АНАЛИЗА, ПРОЕКТИРОВАНИЯ И РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Тема 15 Классификация CASE – инструментария

15.1 Классификация по типам.
15.2 Классификация по категориям.
15.3 Классификация по уровням.
15.4 Эволюция CASE – инструментария.

15.1 Классификация по типам

В монографии Калянова Г.Н. «Case–технологии. Консалтинг при автоматизации бизнес–процессов» приведена классификация CASE[1]–инструментария: на типы, категории и уровни [9, стр.173]. Классификация ***по типам*** отражает функциональную ориентацию CASE–инструментария в технологическом процессе.

1 ***АНАЛИЗ И ПРОЕКТИРОВАНИЕ.*** Средства данной группы используются для создания спецификаций системы и ее проектирования; они поддерживают широко известные методологии проектирования [9, стр.173].. К таким средствам относятся: *The Developer (ASYST Technologies), POSE (Computer Systems Advisers), ProKit*Workbench (McDonnell Douglas), Excelerator (Index Technology), Design–Aid (Nastec), Design Machine (Optima), MicroStep (Mela Systems), vsDesigner (Visual Software), Analist/Designer (Yourdon), Design/IDEF (Meta Software), BPWin (Logic Works), SELECT (Select Software Tools), System Architect (Popkin Software & Systems), Westmount I–CASE Yourdon (Westmount Technology B. V. & CADRE Technologies), CASE/4/0 (microTOOL GmbH); CASE.Аналитик (Эйтэкс)*. Их целью является определение системных требований и свойств, которыми система должна обладать, а также создание проекта системы, удовлетворяющей этим требованиям и обладающей соответствующими свойствами. На выходе продуцируются спецификации компонент системы и интерфейсов, связывающих эти компоненты, а также «калька» архитектуры системы и детальная «калька» проекта, включающая алгоритмы и определения структур данных.

2 ***ПРОЕКТИРОВАНИЕ БАЗ ДАННЫХ И ФАЙЛОВ.*** Средства данной группы обеспечивают логическое моделирование данных, автоматическое преобразование моделей данных в Третью Нормальную Форму, автоматическую генерацию схем БД и описаний форматов файлов на уровне программного кода: *ERWin (Logic Works), Chen Toolkit (Chen &Asssociates), S–Designor (SDP), Designer2000 (Oracle), Silverrun (Computer Systems Advisers).*

---

[1] Аббревиатура CASE расшифровывается как Computer Aided Software Engineering (программная инженерия с компьютерной поддержкой)

3 ***ПРОГРАММИРОВАНИЕ.*** Средства этой группы поддерживают этапы программирования и тестирования, а также автоматическую кодогенерацию из спецификаций, получая полностью документированную выполняемую программу: *COBOL 2/Workbench (Mikro Focus), DECASE (DEC), NETRON/CAP (Netron), APS (Sage Software).* Помимо диграммеров различного назначения и средств поддержки работы с репозитарием, в эту группу средств включены и традиционные генераторы кодов, анализаторы кодов (как в статике, так и в динамике), генераторы наборов тестов, анализаторы покрытия тестами, отладчики.

4 **СОПРОВОЖДЕНИЕ И РЕИНЖИНИРИНГ.** К таким средствам относятся документаторы, анализаторы программ, средства реструктурирования и реинжениринга: Adpac CASE Tools (Adpac), Scan/COBOL u Superstructure (Computer Data Systems), Inspector/Recoder (Language Technology). Их целью является корректировка, изменение, анализ, преобразование и реинжиниринг существующей системы. Средства позволяют осуществлять поддержку всей системной документации, включая коды, спецификации, наборы тестов; контролировать покрытие тестами для оценки полноты тестируемости; управлять функционированием системы и т.п. Особый интерес представляют средства обеспечения мобильности (в CASE они получили название средств миграции) и реинжиниринга. К средствам миграции относятся трансляторы, конверторы, макрогенераторы и др., позволяющие обеспечить перенос существующей системы в новое операционное или аппаратурное окружение. Средства реинжиниринга включают:

– статические анализаторы для продуцирования схем системы ПО из ее кодов, оценки влияния модификаций (например,»эффекта ряби» – внесение изменений с целью исправления ошибок порождает новые ошибки);

– динамические анализаторы (обычно, компиляторы и интерпретаторы с встроенными отладочными возможностями);

– документаторы, позволяющие автоматически получать обновленную документацию при изменении кода;

– редакторы кодов, автоматически изменяющие при редактировании и все предшествующие коду структуры (например, спецификации);

– средства доступа к спецификациям, их модификации и генерации нового (модифицированного) кода;

– средства реверсного инжиниринга, транслирующие коды в спецификации.

5 ***ОКРУЖЕНИЕ.*** Средства поддержки платформ для интеграции, создания и придания товарного вида CASE–средствам: *Multi/Cam (AGS Management Systems), Design/OA (Meta Software).*

6 ***УПРАВЛЕНИЕ ПРОЕКТОМ.*** Средства, поддерживающие планирование, контроль, руководство, взаимодействие, т.е. функции, необходимые в процессе разработки и сопровождения проектов: *Project Workbench (Applied Business Technology).*

15.2 Классификация по категориям

Классификация по категориям определяет уровень интегрированности по выполняемым функциям и включает вспомогательные программы (tools), пакеты разработчика (toolkit) и инструментальные средства (workbench). Категория **tools** обозначает вспомогательный пакет, решающий небольшую автономную задачу, принадлежащую проблеме более широкого масштаба. Категория **toolkit** представляет совокупность интегрированных программных средств, обеспечивающих помощь для одного из классов программных задач; использует репозитарий для всей технической и управляющей информации о проекте, концентрируясь при этом на поддержке, как правило, одной фазы или одного этапа разработки ПО. Категория **workbench** представляет собой интеграцию программных средств, которые поддерживают системный анализ, проектирование и разработку ПО; используют репозитарий, содержащий всю техническую и управляющую информацию о проекте; обеспечивают автоматическую передачу системной информации между разработчиками и этапами разработки; организуют поддержку практически полного ЖЦ (от анализа требований и проектирования ПО до получения документированной выполняемой программы). Workbench, по сравнению с toolkit, обладает более высокой степенью интеграции выполняемых функций, большей самостоятельностью и автономностью использования, а также наличием тесной связи с системными и техническими средствами аппаратно–вычислительной среды, на которой workbench функционирует. По существу, workbench может рассматриваться как автоматизированная рабочая станция, используемая как инструментарий для автоматизации всех или отдельных совокупностей работ по созданию ПО.

15.3 Классификация по уровням

Классификация по уровням связана с областью действия CASE в пределах жизненного цикла ПО. Однако четкие критерии определения границ между уровнями не установлены, поэтому данная классификация имеет, вообще говоря, качественный характер.

Верхние (Upper) CASE часто называют средствами компьютерного планирования. Они призваны повышать эффективность деятельности руководителей фирмы и проекта путем сокращения затрат на определение политики фирмы и на создание общего плана проекта. Этот план включает цели и стратегии их достижения, основные действия в свете целей и задач фирмы, установление стандартов на различные виды взаимосвязей и т.д. Использование верхних CASE позволяет построить модель предметной области, отражающую всю существующую специфику. Она направлена на понимание общего и частного механизмов функционирования, имеющихся возможностей, ресурсов, целей проекта в соответствии с назначением фирмы. Эти средства позволяют проводить анализ различных сцена-

риев (в том числе наилучших и наихудших), накапливая информацию для принятия оптимальных решений.

Средние (Middle) CASE считаются средствами поддержки этапов анализа требований и проектирования спецификаций и структуры ПО. Их использование существенно сокращает цикл разработки проекта; при этом важную роль играет возможность накопления и хранения знаний, обычно имеющихся только в голове разработчика–аналитика, что позволит использовать накопленные решения при создании других проектов. Основная выгода от использования среднего CASE состоит в значительном облегчении проектирования систем, проектирование превращается в итеративный процесс, включающий следующие действия: пользователь обсуждает с аналитиком требования к проектируемой системе; аналитик документирует эти требования, используя диаграммы и словари входных данных; пользователь проверяет эти диаграммы и словари, при необходимости модифицируя их; аналитик отвечает на эти модификации, изменяя соответствующие спецификации.

Кроме того, средние CASE обеспечивают возможности быстрого документирования требований и быстрого прототипирования.

Нижние (Lower) CASE являются средствами разработки ПО (при этом может использоваться до 30% спецификаций, созданных средствами среднего CASE). Они содержат системные словари и графические средства, исключающие необходимость разработки физических спецификаций. Имеются системные спецификации, которые непосредственно переводятся в программные коды разрабатываемой системы (при этом автоматически генерируется до 80–90% кодов). На эти средства возложены также функции тестирования, управления конфигурацией, формирования документации. Главными преимуществами нижних CASE являются: значительное уменьшение времени на разработку, облегчение модификаций, поддержка возможностей прототипирования (совместно со средними CASE).

15.4 Эволюция CASE – инструментария

С самого начала CASE–технологии развивались с целью преодоления ограничений ручных применений методологий структурного анализа и проектирования 60–70–х годов (сложности понимания, большой трудоемкости и стоимости использования, трудности внесения изменений в проектные спецификации и т.д.) за счет их автоматизации и интеграции поддерживающих средств. Таким образом CASE–технологии не могут считаться самостоятельными методологиями, они только делают более эффективными пути их применения. CASE – не революция в программотехнике: современные CASE–средства являются естественным продолжением эволюции всей отрасли средств разработки ПО. Традиционно выделяют шесть периодов, качественно отличающихся применяемой техникой и методами разработки ПО, которые характеризуются использованием в качестве инструментальных следующих средств:

– *ассемблеров, дампов памяти, анализаторов;*

– *компиляторов, интерпретаторов, трассировщиков;*

– *символических отладчиков, пакетов программ;*

– *систем анализа и управления исходными текстами;*

– *CASE–средств анализа требований, проектирования спецификаций и структуры, редактирования интерфейсов(первая генерация CASE–1)*

– *CASE–средств генерация исходных текстов и реализации интегрированного окружения поддержки полного жизненного цикла (ЖЦ) разработки ПО (2–ая генерация CASE–2)*

CASE–I является первой технологией, адресованной непосредственно системным аналитикам и проектировщикам, и включающей средства для поддержки графических моделей, проектирования спецификаций, экранных редакторов и словарей данных. Она не предназначена для поддержки полного ЖЦ и концентрирует внимание на функциональных спецификациях и начальных шагах проекта – системном анализе, определении требований, системном проектировании, логическом проектировании БД.

CASE–II отличается значительно более развитыми возможностями, улучшенными характеристиками и исчерпывающим подходом к полному ЖЦ. В ней в первую очередь используются средства поддержки автоматической кодогенерации, а также обеспечивается полная функциональная поддержка для порождения графических системных требований и спецификаций проектирования; контроля, анализа и связывания системной информации, а также информации по управлению проектированием; построения прототипов и моделей системы; тестирования, верификации и анализа сгенерированных программ; генерации документов по проекту; контроля на соответствие стандартам по всем этапам ЖЦ. CASE–II может включать свыше 100 функциональных компонент, поддерживающих все этапы ЖЦ, при этом пользователям предоставляется возможность выбора необходимых средств и их интеграции в нужном составе.

Тема 16 Концептуальные основы CASE – технологий

16.1 CASE–модель жизненного цикла программного обеспечения.

16.2 Состав и структура и функциональные особенности CASE–инструментария.

16.3 Поддержка графических моделей.

16.4 Поддержка процесса проектирования и разработки.

16.1 CASE–модель жизненного цикла программного обеспечения

CASE–технологии предлагают новый, основанный на автоматизации, подход к концепции ЖЦ ПО. При использовании CASE изменяются все фазы ЖЦ, при

этом наибольшие изменения касаются фаз анализа и проектирования. Простейшая модель ЖЦ реализуется этапами: анализа, проектирования, кодирования, тестирования и сопровождения; в соответствующей ей CASE–модели ЖЦ (прототипирование, проектирование спецификаций, контроль проекта, кодогенерация, системное тестирование, сопровождение) фаза прототипирования заменяет традиционную фазу системного анализа. Необходимо отметить, что наиболее автоматизируемыми фазами являются фазы контроля проекта и кодогенерации (хотя все остальные фазы также поддерживаются CASE–средствами).

В таблице 16.1 приведены оценки трудозатрат по фазам ЖЦ. Первая строка таблицы соответствует традиционной разработке, вторая – разработке с использованием структурных методологий проектирования, третья – разработке с использованием CASE–технологий. В таблицу 16.2 сведены основные изменения в ЖЦ при использовании CASE–технологий по сравнению с традиционной разработкой.

Таблица 16.1 – Оценки трудозатрат по фазам ЖЦ

Способ разработки	Анализ	Проектирование	Кодирование	Тестирование
Традиционная разработка	20%	15%	20%	45%
Использование структурных методологий проектирования	30%	30%	15%	25%
Использование CASE–технологий	40%	40%	5%	15%

Таблица 16.2 – Основные отличия ЖЦ при использовании CASE–технологий по сравнению с традиционной разработкой

	Традиционная разработка	CASE
1	Основные усилия – на кодирование и тестирование	Основные усилия – на анализ и проектирование
2	«Бумажные» спецификации	Быстрое итеративное прототипирование
3	Ручное кодирование	Автоматическая кодогенерация
4	Ручное документирование	Автоматическая генерация документации
5	Тестирование кодов	Автоматический контроль проекта
6	Сопровождение кодов	Сопровождение спецификаций проектирования

На рисунке 16.1 представлены результаты сравнения традиционной разработки программных проектов и разработки с применением CASE–технологий.

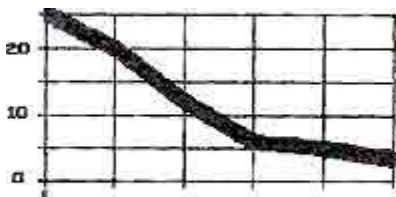

Рисунок 16.1 – Уменьшение затрат на проектирование ПО за счет CASE–технологий

16.2 Состав и структура и функциональные особенности CASE–инструментария

CASE–инструментарий предназначен для поддержки и усиления методов структурного анализа и проектирования. Эти инструменты поддерживают работу пользователей при создании и редактировании графического проекта в интерактивном режиме, они способствуют организации проекта в виде иерархии уровней абстракции, выполняют проверки соответствия компонентов. Фактически CASE–средства представляют собой новый тип графически–ориентированных инструментов, восходящих к системе поддержки ЖЦ ПО. Обычно к ним относят любое программное средство, обеспечивающее автоматическую помощь при разработке ПО, его сопровождении или деятельности по управлению проектом, и проявляющее следующие дополнительные черты:

– мощная графика для описания и документирования систем ПО, а также для улучшения интерфейса с пользователем, развивающая творческие возможности специалистов и не отвлекающая их от процесса проектирования на решение второстепенных вопросов;

– интеграция, обеспечивающая легкость передачи данных между средствами и позволяющая управлять всем процессом проектирования и разработки ПО непосредственно через процесс планирования проекта;

– использование компьютерного хранилища (репозитария) для всей информации о проекте, которая может разделяться между разработчиками и исполнителями как основа для автоматического продуцирования ПО и повторного его использования в будущих системах.

Помимо перечисленных основополагающих принципов графической ориентации, интеграции и локализации всей проектной информации в репозитарии в основе концептуального построения CASE–инструментария лежат следующие положения:

1 Человеческий фактор, определяющий разработку ПО как легкий, удобный и экономичный процесс.

2 Широкое использование базовых программных средств, получивших массовое распространение в других приложениях (БД и СУБД, компиляторы с различных языков программирования, отладчики, документаторы, издатель-

ские системы, оболочки экспертных систем и базы знаний, языки четвертого поколения и др).

3 Автоматизированная или автоматическая кодогенерация, выполняющая несколько видов генерации кодов: преобразования для получения документации, формирования БД, ввода/модификации данных, получения выполняемых машинных кодов из спецификаций ПО, автоматической сборки модулей из словарей и моделей данных и повторно используемых программ, автоматической конверсии ранее используемых файлов в форматы новых требований.

4 Ограничение сложности, позволяющее получать компоненты, поддающиеся управлению, обозримые и доступные для понимания, а также обладающие простой и ясной структурой.

5 Доступность для разных категорий пользователей.

6 Рентабельность.

7 Сопровождаемость, обеспечивающая способность адаптации применении требований и целей проекта.

Интегрированный CASE–пакет содержит четыре основные компоненты:

1 Средства централизованного хранения всей информации о проектируемом ПО в течении всего ЖЦ (репозитарнй) являются основой CASE–пакета. Соответствущая БД должна иметь возможность поддерживать большую систему описании и характеристик и предусматривать надежные меры по защите от ошибок и потерь информации. Репозитарий должен обеспечивать:

– инкрементный режим при вводе описаний объектов;
– распространение действия нового или скорректированного описания на информационное пространство всего проекта;
– синхронизацию поступления информации от различных пользователей;
– хранение версий проекта и его отдельных компонент;
– сборку любой запрошенной версии;
– контроль информации на корректность, полноту и состоятельность.

2 Средства ввода предназначены для ввода данных в репозитарий, а также для организации взаимодействия с CASE–пакетом. Эти средства должны поддерживать различные методологии и использоваться на всем ЖЦ разными категориями разработчиков: аналитиками, проектировщиками, инженерами, администраторами и т. д.

3 Средства анализа, проектирования п разработки предназначены для того, чтобы обеспечить планирование и анализ различных описаний, а также их преобразования в процессе разработки.

4 Средства вывода служат для документирования, управления проектом и кодовой генерации.

Все перечисленные компоненты в совокупности должны:

– поддерживать графические модели;
– контролировать ошибки;
– организовывать и поддерживать репозитарий;

– поддерживать процесс проектирования и разработки.

16.3 Поддержка графических моделей

Графическая ориентация CASE заключается в том, что программы являются схематическими проектами и формами, которые много проще в использовании, чем многостраничные описания. Для представления программ применяются структурные диаграммы различных типов, дополнительное достоинство которых заключается в их использовании в качестве наглядной «двумерной» документации по проекту.

Для CASE существенны 4 типа диаграмм: диаграммы функционального проектирования (для этих целей наиболее часто употребляются DFD – диаграммы потоков данных), диаграммы моделирования данных (как правило, ERD – диаграммы «сущность–связь»), диаграммы моделирования поведения (как правило, STD – диаграммы переходов состояний) и структурные диаграммы (карты), применяющиеся на этапе проектирования и описывающие отношения между модулями и внутри модульную структуру. Создание и модификация подобных диаграмм осуществляется с помощью специальных графических редакторов (диаграммеров), являющихся сервисными средствами на этапах анализа требований и проектирования спецификаций. Современные диаграммеры обеспечивают:

– создание иерархически связанных диаграмм, в которых комбинируются графические и текстовые объекты;
– создание и редактирование объектов в любом месте диаграммы;
– создание, перемещение и выравнивание групп объектов, изменение их размеров, масштабирование;
– сохранение связей между объектами при их перемещении и изменении размеров;
– автоматический контроль ошибок и др.

Реализация подобных возможностей позволяет пользователю целиком сосредоточится на собственно проектировании, не отвлекаясь на решение второстепенных вопросов, связанных с размещением элементов диаграмм, их компоновкой и т.п.

Полученные диаграммы дают ясное понимание и решение проблемы, позволяют проанализировать функционирование создаваемого ПО, фиксируют связи между разработчиками, пользователями и руководителями, обеспечивают стандартизацию представления структуры программы и данных.

Важность **контроля ошибок** на этапах анализа требований и проектирования спецификаций обуславливается возможностью их автоматического обнаружения на ранних этапах ЖЦ. CASE обеспечивает автоматическую верификацию и контроль проекта на полноту и состоятельность на ранних этапах ЖЦ, что влияет на успех разработки в целом. В подтверждение этого можно привести следующие

статистические данные, основанные на отчетах фирмы TRW по анализу 5 крупных проектов [9]:

– при традиционной организации работ ошибки проектирования и кодирования составляют, соответственно, 64% и 32% от общего числа ошибок;

– ошибки проектирования в 100 раз труднее обнаружить на этапе сопровождения ПО, чем на этапах анализа требований и проектирования спецификаций.

В CASE диаграммеры и верификаторы способны осуществлять следующие типы контроля:

1 *Контроль синтаксиса диаграмм и типов их элементов.* Обычно такой контроль осуществляется при вводе и редактировании элементов диаграмм. Примеры контролируемых ситуаций:

– *по синтаксису:* любой функциональный элемент диаграммы должен иметь по крайней мере один входной и один выходной поток; два элемента данных не могут быть непосредственно связаны;

– *по типам:* функциональный элемент должен всегда использоваться для представления процедурной компоненты; поток данных всегда должен быть представлен компонентой данных.

2 *Контроль полноты и состоятельности диаграмм:* все элементы диаграмм должны быть идентифицированы и отражены в репозитарии. Например, для DFD контролируются неименованные или несвязанные потоки данных, процессы и хранилища данных; источники и стоки данных (внешние сущности) вне контекстной диаграммы; хранилища данных на контекстной диаграмме и т.п. При анализе словаря данных необходимо выявлять циклические определения, эквивалентные определения, неопределенные объекты.

3 *Контроль декомпозиции функций* включает оценку качества на основе различных метрик ПО и частичный семантический контроль.

4 Сквозной контроль диаграмм одного или различных типов на предмет их состоятельности по уровням *вертикальное и горизонтальное балансирование диаграмм.* При вертикальном балансировании (диаграммы одного типа) выявляются несбалансированные потоки данных между детализируемой и детализирующей диаграммами. Горизонтальное балансирование определяет некорректности между DFD, ERD, STD, словарями данных и миниспецификациями процессов. Так при балансировании DFD–ERD контролируется соответствие каждого хранилища данных на DFD сущности или отношению на ERD. Контроль DFD–STD осуществляется по следующим правилам: каждый управляющий процесс на DFD детализируется спецификацией управления STD, и наоборот, каждой STD должен соответствовать управляющий процесс; каждое условие (действие) в STD должно соответствовать входному (выходному) управляющему потоку на DFD, и наоборот, каждому управляющему потоку в зависимости от его направленности должно соответствовать условие/действие на STD. При балансировании DFD–словарь данных–миниспецификация должны проверяться следующие правила:

– каждый поток и хранилище данных должны быть определены в словаре данных (контроль неопределенных значений), и наоборот, каждое определение в словаре должно быть отражено на диаграмме, в миниспецификации или другом определении (контроль неиспользуемых значений);
– каждый процесс на DFD должен детализироваться с помощью DFD или миниспецификации (но не тем и другим одновременно), и наоборот, каждая миниспецификация должна соответствовать единственному процессу;
– ссылки к данным в миниспецификациях должны соответствовать объектам на диаграммах и в словаре данных;
– по возможности должна контролироваться семантика миниспецификации: например, если входные и/или выходные потоки связаны с хранилищем данных, то это должно быть отражено в миниспецификации (операторами READ, GET, WRITE, PUT и т.п.).

16.4 Поддержка процесса проектирования и разработки

При поддержке процесса проектирования и разработки основную роль играют следующие возможности CASE–пакетов: покрытие ЖЦ, поддержка прототипирования, поддержка структурных методологий, автоматическая кодогеиерация.

При *покрытии ЖЦ* наибольшее внимание уделяется его наиболее критичным этапам – анализу требований и проектированию спецификаций. Последние являются основой всего проекта, поэтому их полнота и корректность влияют на успех разработки в целом.

Важную роль при автоматизации ранних этапов ЖЦ играют возможности *поддержки прототипирования.* Соответствующие средства используются для определения системных требований и ответа на вопросы об ожидаемом поведении системы. Такие средства как генераторы меню, экранов и отчетов позволяют быстро построить прототипы пользовательских интерфейсов и снабдить моделью функционирования системы с позиций конечного пользователя. Использование языков четвертого поколения (4GL) позволяет строить более сложные модели, при этом прототип позволяет промоделировать основные функции системы, но не способен контролировать ее ожидаемое поведение. Исполняемые языки спецификаций преобразуют процесс разработки в следующий итеративный процесс: спецификации определяются и выполняются, затем производится переопределение или корректировка. Созданные таким образом прототипы позволяют определять, является ли проектируемая система полной и корректной.

Поддержка структурных методологий осуществляется за счет средств их автоматизации на следующих двух уровнях:
– подготовка документации, графическая поддержка построения структурных диаграмм различных типов, продуцирование спецификаций для детализации функциональных блоков в диаграммах и структур данных на нижних уровнях (для таких спецификаций введен специальный термин – «миниспецификация»);

– корректное использование шагов обработки в методологиях.

Кодогеперация осуществляется на основе репозитария и позволяет автоматически построить до 80–90% объектных кодов или текстов программ на языках высокого уровня. При этом различными CASE–пакетами поддерживаются практически все известные языки программирования, однако наиболее часто в качестве целевых языков выступают COBOL, C и ADA. Средства кодогенерации по отношению к полноте целевого продукта разделяются на средства генерации каркаса ПО и средства генерации полного продукта. В первом случае автоматически строится откомментированная логика (потоки управления) ПО, а также коды для БД, файлов, экранов, отчетов и т.п., остальные фрагменты ПО кодируются вручную. Во втором случае из проектных спецификаций генерируется полная документированная программа, включая выполняемый код, пользовательскую и программную документацию, наборы тестов и т.д. Все эти компоненты полной программы связываются в единый объект, хранящийся в репозитарии для облегчения доступа и сопровождения.

Идея автоматической кодогенерации на основе модели заключается в следующем. Любая программа схематически может быть представлена в виде тройки: обрабатываемые данные, логический каркас обработки, линейные участки обработки. Схема базы данных может быть легко сгенерирована на основании модели «сущность–связь», и современные средства информационного моделирования (например, ERWin, Designer/2000 и др.) обеспечивают такую генерацию. Логика обработки генерируется на основе диаграмм потоков данных: известны алгоритмы автоматического .преобразования иерархии DFD в структурные карты, а с задачей получения из структурных карт соответствующих кодов легко справляется теория компиляции. Наконец, линейным участкам соответствуют миниспецификации модели. И именно здесь лежит ключ к высокому проценту автоматически сгенерированного кода, существенно зависящему от метода задания миниспецификаций.

ГГУ ИМ

ЛИТЕРАТУРА

1 Благодатских, В.А. Стандартизация разработки программных средств [Текст] : учебное пособие / В.А. Благодатских, В.А. Волнин, К.Ф. Поскакалов; под ред. О.С. Разумова. – М.: Финансы и статистика, 2005.

2 Вендров, А.М. Проектирование программного обеспечения экономических информационных систем : учебник / А.М. Вендров. – М.: Финансы и статистика, 2000.

3 Крылова, Г.Д. Основы стандартизации, сертификации, метрологии : учебник для вузов / Г.Д. Крылова – 2–е изд.. – М.: ЮНИ– ТИ–ДАНА, 2001.

4 Майерс, Г. Надежность программного обеспечения. / Г. Майерс: – М.: Мир, 1980.

5 Першиков, В.И. Толковый словарь по информатике. [Текст] / В.И. Першиков, В.М. Савинков–2–е изд. – М.: Финансы и статистика, 1995.

6 Тейер, Т. Надежность программного обеспечения / Т. Тейер, М. Липов, Э. Нельсон / Пер. с англ. – М.: Мир, 1981.

7 Грек, В.В. Стандартизация и метрология систем обработки данных : учебное пособие / В.В. Грек., И.В. Максимей. – Мн.: Выш. шк., 1994.

8 Бейзер, Б. Тестирование черного ящика. Технологии функционального тестирования программного обеспечения и систем. / Б. Бейзер.– СПб.: Питер, 2004.

9 Калянов, Г.Н. Case–технологии. Консалтинг при автоматизации бизнес–процессов. [Текст] / Г.Н. Калянов. – 3–е изд. – М.:.Горячая линия – Телеком. 2000.

10 Соммервилл, И. Инженерия программного обеспечения / И. Соммервил. – М., 2002.

ГГУ ИМ

Учебное издание

Осипенко Наталья Борисовна

СТАНДАРТИЗАЦИЯ И СЕРТИФИКАЦИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Тексты лекций для студентов математических специальностей

В авторской редакции

Учреждение образования
«Гомельский государственный университет
имени Франциска Скорины»
246019, г. Гомель, ул. Советская, 104.