МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное автономное образовательное учреждение
высшего образования
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

КАФЕДРА КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ
И ПРОГРАММНОЙ ИНЖЕНЕРИИ

МЕТРОЛОГИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Учебное пособие

Санкт-Петербург
2017

Составители: Копкин Е.В., Каргин В.А., Павлов Е.В.

Рецензент:

Метрология программного обеспечения. Учебное пособие – СПб ГУАП, 2017. – 145 с.

В учебном пособии приводятся описания различных метрик, критериев, моделей качества и надежности программного обеспечения, а также осуществляется их сравнительный анализ.

Учебное пособие предназначено для изучения теоретического материала по дисциплине «Метрология программного обеспечения» студентами различных форм обучения по направлениям 09.03.04 «Программная инженерия» и 01.03.02 «Прикладная математика и информатика».

Подготовлены кафедрой компьютерных технологий и программной инженерии.

ОГЛАВЛЕНИЕ

СПИСОК СОКРАЩЕНИЙ

ГОСТ	– государственный стандарт	
ЖЦ	– жизненный цикл	
ИСО (ISO)	– международная организация по стандартизации	
МЭК (IEC)	– международная электротехническая комиссия	
ПИ	– программное изделие	
ПО	– программное обеспечение	
ПС	– программное средство	
СТО	– средняя тяжесть ошибок	
ЭВМ	– электронно-вычислительная машина	
ЭПК	– элемент показателя качества	
ATAM	– Architecture Tradeoff Analysis Method	
CBP	– Critical Best Practices	
CMM	– Capability Maturity Model for Software	
CMMI	– Capability Maturity Model Integration	
CTL	– Computation Tree Logic	
LOC	– Lines of Code	
SAAM	– Software Architecture Analysis Method	
SEI	– Software Engineering Institute	
SLA	– Service Level Agreement	
SLOC	– Source Lines of Code	
SQuaRE	– Systems and software Quality Requirements and Evaluation	
SWEBOK	– Software Engineering Body of Knowledge	

ПРЕДИСЛОВИЕ

В учебном пособии рассматривается материал учебной дисциплины «Метрология программного обеспечения», которая входит в базовую часть программы подготовки бакалавров по направлениям 01.03.02 (Прикладная математика и информатика) и 09.03.04 (Программная инженерия).

Учебное пособие состоит из введения, четырех разделов, заключения и списка литературы.

В первом разделе настоящего учебного пособия с точки зрения актуальных международных и российских стандартов рассматриваются общие вопросы, связанные с анализом качества программного обеспечения как одного из разделов программной инженерии. Описываются основные модели качества программного обеспечения и используемые в рамках этих моделей характеристики и подхарактеристики качества.

Второй раздел посвящен вопросам оценивания сложности разработки программного обеспечения, большое внимание уделено различным метрикам, используемым для оценивания этой сложности.

В третьем разделе рассматриваются вопросы, связанные с анализом и оцениванием корректности программных продуктов.

Четвертый раздел посвящен рассмотрению вопросов, связанных с оцениванием надежности программного обеспечения, достаточно много внимания уделено моделям надежности программ.

Для смыслового выделения текста при определении терминов используется полужирный шрифт.

При формировании данного учебного пособия авторы стремились максимально использовать актуальные стандарты, которыми должны руководствоваться в своей деятельности специалисты в области разработки программного обеспечения.

ВВЕДЕНИЕ

На протяжении многих лет отдельные авторы и целые организации определяли термин «качество» применительно к программному обеспечению по-разному. Фил Кросби (Phil Crosby) в 1979 году дал определение качеству как «соответствие пользовательским требованиям». Уотс Хемпфри (Watts Hamphrey) описывает качество как «достижение отличного уровня пригодности к использованию». Компания IBM, в свою очередь, ввела в оборот фразу «качество, управляемое рыночными потребностями» («market-driven quality»). Критерий Бэлдриджа (Baldrige) для организационного качества использует похожую фразу – «качество, задаваемое потребителем» («customer-driven quality»), рассматривая удовлетворение потребителя в качестве главного соображения в отношении качества.

В настоящее время понятие качества используется в соответствии с ГОСТ ISO 9000-2015 [16] как «степень соответствия присущих характеристик требованиям».

Эти взгляды перекликаются и с термином «приемлемое качество», определяемым не только уровнем запросов конечных потребителей в отношении параметров создаваемого продукта, но и заданным контекстом (ограничениями проекта). Это не значит, что «приемлемое качество» противопоставляется «качеству, диктуемому заказчиком». Конечно, не стоит и проводить параллель «приемлемого качества» с «продуктом второй свежести». Введение категории «приемлемости» в отношении качества является лишь прагматичным взглядом на желаемую степень совершенства создаваемого продукта (услуги), способную удовлетворить пользователей и достижимую в рамках заданных проектных ограничений. Интересно, что и сама «степень приемлемости» также выступает в роли ограничения проекта, а в приложении к индустрии программного обеспечения (ПО) представлена практически во всех областях проектной деятельности – от управления требованиями, до тестирования. В какой-то степени, «приемлемое качество» можно сравнивать с уровнем обслуживания в рамках заданного SLA (Service Level Agreement) – соглашения, давно уже принятого на вооружение в телекоммуникационной индустрии. Таким образом, приемлемое качество может рассматриваться как

количественно выраженный компромисс между заказчиком и исполнителем в отношении характеристик продукта, создаваемого исполнителем в интересах решения задач заказчика с учетом других ограничений проекта (в частности, стоимости, что часто именуется как «cost of quality» – «стоимость качества»). Можно сказать, что такой взгляд может в какой-то степени рассматриваться как расширение определения ГОСТ Р ИСО 9000 с учетом достигнутого компромисса между заказчиком и исполнителем (поставщиком) в отношении характеристик качества.

Качество программного обеспечения является постоянным объектом заботы программной инженерии и обсуждается во многих областях знаний (см. рис. 1).

Рис. 1. Область знаний «Качество программного обеспечения»

Наибольшие достижения в развитии и применении современных методов и средств обеспечения качества крупномасштабных программных средств были сосредоточены в фирмах, работающих по заказам министерства обороны.

В ноябре 1986 года американский институт Software Engineering Institute (SEI) при Университете Карнеги-Меллона (Carnegie Mellon University) совместно с Mitre Corporation начали разработку обзора зрелости процессов разработки программного обеспечения, который был предназначен для помощи в улучшении их внутренних процессов.

Разработка такого обзора была вызвана запросом американского федерального правительства на предоставление метода оценки субподрядчиков для разработки ПО. Реальная же проблема состояла в неспособности управлять большими проектами. Во многих компаниях проекты выполнялись со значительным опозданием и с превышением запланированного бюджета. Необходимо было найти решение данной проблемы.

В сентябре 1987 года SEI выпустил краткий обзор процессов разработки ПО с описанием их уровней зрелости, а также опросник, предназначавшийся для выявления областей в компании, в которых были необходимы улучшения. Однако большинство компаний рассматривало данный опросник в качестве готовой модели, вследствие чего через 4 года он был преобразован в реальную модель оценки зрелости применяемых технологических процессов жизненного цикла программных средств, которая получила название Capability Maturity Model for Software (CMM)[1]. Первая версия CMM, вышедшая в 1991 году, была пересмотрена в 1992 году.

Эта модель основана на использовании пяти уровней зрелости технологии, которые характеризуются:

- степенью формализации, измерения и документирования процессов и продуктов жизненного цикла (ЖЦ) программных средств (ПС);
- широтой применения стандартов и инструментальных средств автоматизации работ;
- наличием и полнотой функций систем обеспечения качества технологических процессов.

Уровень 1 – «Начальный» (initial). На предприятии начального уровня организации не существует стабильных условий для созданий качественного программного обеспечения. Результат любого проекта целиком и полностью зависит от личных качеств менеджера и опыта программистов, причем успех в одном проекте может быть повторен только в случае назначения тех же менеджеров и программистов на следующий проект. Более того, если такие менеджеры или программисты уходят с предприятия, то с их уходом резко падает качество производимых программных продуктов. В стрессовых

[1]URL: http://web.archive.org/web/20070928200421/http://www.ryabikin.com/sw-cmm/index.htm
(перевод на русский язык)

ситуациях процесс разработки сводится к написанию кода и его минимальному тестированию.

Уровень 2 – «Повторяемый» (repeatable). В организации внедрены технологии управления проектами. При этом планирование и управление проектами основывается на накопленном опыте, существуют стандарты на разрабатываемое программное обеспечение (причем обеспечивается следование этим стандартам) и существует специальная группа обеспечения качества. В случае необходимости организация может взаимодействовать с субподрядчиками. В критических условиях процесс имеет тенденцию скатываться на начальный уровень. В середине 1999 года лишь 20 % организаций имели 2-й уровень или выше.

Уровень 3 – «Определенный» (defined). Он характеризуется тем, что стандартный процесс создания и сопровождения программного обеспечения задокументирован (включая и разработку ПО, и управление проектами). Подразумевается, что в процессе стандартизации происходит переход на наиболее эффективные практики и технологии. Для создания и поддержания подобного стандарта в организации должна быть создана специальная группа. Наконец, обязательным условием для достижения данного уровня является наличие в организации программы постоянного повышения квалификации и обучения сотрудников. Начиная с этого уровня, организация перестает зависеть от качеств конкретных разработчиков, и не имеет тенденции скатываться на уровень ниже в стрессовых ситуациях.

Уровень 4 – «Управляемый» (manageable). В организации устанавливаются количественные показатели качества – как на программные продукты, так и на процесс в целом. Таким образом, более совершенное управление проектами достигается за счет уменьшения отклонений различных показателей проекта. При этом осмысленные вариации в производительности процесса можно отличить от случайных вариаций (шума), особенно в хорошо освоенных областях.

Уровень 5 – «Оптимизирующий» (optimizing). Характеризуется тем, что мероприятия по улучшению применяются не только к существующим процессам, но и для оценки эффективности ввода новых технологий. Основной задачей всей организации на этом уровне является постоянное улучшение существующих процессов. При этом улучшение процессов в идеале должно помогать предупреждать возможные ошибки или дефекты. Кроме того, должны вестись работы по уменьшению стоимости разработки программного обеспечения, например, с помощью создания и повторного использования компонент.

При сертификации проводится оценка соответствия всех ключевых областей по 10-балльной шкале. Для успешной квалификации данной ключевой области необходимо набрать не менее 6 баллов.

Методология обеспечения качества программных средств в дальнейшем активно развивалась, однако внедрение системы требовало больших усилий и затрат. Это ограничило ее массовое использование для относительно простых и средней сложности проектов. Поэтому только несколько процентов компаний США достигло четвертого и пятого уровня зрелости применения CMM, а подавляющее большинство фирм оценивается вторым-третьим или даже первым уровнями.

Специальное подразделение экспертов SPMN (Software Program Managers Network) разработало стратегию использования критически важных практических приемов CBP (Critical Best Practices), которая быстро позволяет организации достигнуть третьего – четвертого уровня зрелости применяемых технологических процессов жизненного цикла программных средств.

Согласно данной стратегии предлагается:

- сосредоточиться на количественных параметрах завершения проекта;
- регулярно измерять продвижение к цели и активность реализации проекта;
- как можно раньше выявлять ошибки и логические неувязки;
- детально планировать работу;
- сводить к минимуму неконтролируемые изменения.

Дальнейшим развитием методологии CMM является CMMI[2] (Capability Maturity Model Integration) – набор моделей (методологий) совершенствования процессов в организациях разных размеров и видов деятельности. CMMI содержит набор рекомендаций в виде практик, реализация которых позволяет реализовать цели, необходимые для полной реализации определённых областей деятельности.

Набор моделей CMMI включает в свой состав три модели:

- CMMI for Development (CMMI-DEV);
- CMMI for Services (CMMI-SVC);
- CMMI for Acquisition (CMMI-ACQ).

Наиболее известной является модель CMMI for Development, ориентированная на организации, занимающиеся разработкой программного обеспечения, аппаратного обеспечения, а также комплексных систем. Все действующие версии моделей имеют номер 1.3 (вышли в ноябре 2010 года).

[2]URL: http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=9661;
https://ashamray.wordpress.com/stati_hf/methodology/cmmi_dev_v13/ (перевод на русский язык)

CMMI определяет 22 процессные области (process areas). Для каждой из процессных областей существует ряд целей (goals), которые должны быть достигнуты при внедрении CMMI в данной процессной области. Некоторые цели являются уникальными — они называются специфическими (specific). Общие (generic) цели применяются ко всем процессным областям. Цели достигаются при помощи реализации практик (рекомендаций модели), либо их адекватных альтернатив. В соответствии с отношением к какой-либо цели, практики делятся на специфические и общие.

Существуют два представления CMMI: непрерывное (continuous) и ступенчатое (staged). При реализации практик CMMI с использованием непрерывного представления, выбор процессных областей не фиксирован (хотя в модели содержатся рекомендации по порядку реализации областей). Для оценки уровня институционализации процессной области используется шкала из шести (0-5) уровней способности (capability level). Ступенчатое представление определяет пять (1-5) уровней зрелости (maturity level) организации:

Уровень 1 – начальный. Процессы непредсказуемые, слабо контролируемые. Процессы появляются в ответ на определенные события.

Уровень 2 – управляемый. Процессы определены на уровне проектов. Зачастую процессы появляются в ответ на определенные события.

Уровень 3 – определенный. Процессы определены на уровне всей организации. Процессы исполняются заблаговременно.

Уровень 4 – управляемый на основе количественных данных. Процессы измеряются и контролируются.

Уровень 5 – оптимизируемый. Основные усилия направлены на совершенствование процессов.

Для достижения каждого уровня зрелости (кроме первого) необходимо выполнить требования по реализации целей определённого набора процессных областей. Первый уровень зрелости в модели не определён.

Дебаты в отношении того, какой именно стандарт стоит использовать инженерам для обеспечения качества программного обеспечения – CMMI или ГОСТ ИСО 9001 [17], продолжаются с самого создания этих стандартов. Сегодня можно сказать о том, что данные стандарты все же рассматривают как взаимодополняющие и, что сертификация по ИСО 9001 помогает в достижении старших уровней зрелости по CMMI.

Итогом усилий профессионального сообщества (IEEE Computer Society) в объединении знаний по инженерии программного обеспечения является документ, подготовленный комитетом Software Engineering Coordinating

Committee – SWEBOK[3] (Software Engineering Body of Knowledge) – свод знаний по программной инженерии.

SWEBOK аккумулирует в себе представление профессионального сообщества о том, что должен знать и какими навыками владеть инженер-программист.

Одним из разделов SWEBOK, в котором описываются пути достижения качества программного обеспечения, является раздел «Software Quality».

В SWEBOK представлены так называемые «общепринятые» знания и навыки, применение которых не является обязательным во всех возможных случаях, но владение которыми необходимо для любого квалифицированного разработчика. SWEBOK отражает практику подготовки инженеров-программистов в США: это знания, которыми должны обладать разработчики, имеющие степень бакалавра и четыре года практической работы, для успешного прохождения лицензионного экзамена.

Предполагается, что, обладая багажом знаний, определенным в SWEBOK, разработчик сможет выбрать оптимальное технологическое решение.

В России методы и опыт создания сложных комплексов программ высокого качества в 1960-1980-е годы были сосредоточены на предприятиях оборонной промышленности. Этот опыт сегодня в значительной степени утрачен, поэтому в области обеспечения жизненного цикла и качества сложных комплексов программ существует и применяется небольшая группа государственных стандартов серии 19 и 34. В результате многие проекты сложных программных средств оказываются недостаточного качества и требуют длительной доработки для устранения множества системных и технических дефектов и ошибок.

Первое место в неформальном состязании за место «самой дорого обошедшейся ошибки в программном обеспечении» долгое время удерживала ошибка, приведшая к неудаче первого запуска ракеты «Ариан-5» 4 июня 1996 г. стоившая около 500 миллионов долларов США. После произошедшего 14 августа 2003 г. обширного отключения электричества на северо-востоке Северной Америки, стоившего экономике США и Канады от 4 до 10 миллиардов долларов, это место закрепилось за вызвавшей его ошибкой в системе управления электростанцией.

Таким образом, инженеры должны понимать смысл, вкладываемый в концепцию качества, характеристики и значение качества в отношении разрабатываемого или сопровождаемого программного обеспечения. Очевидно, что инженеры по программному обеспечению должны воспринимать вопросы

[3] URL: https://www.computer.org/web/swebok

качества программного обеспечения как часть своей профессиональной культуры.

Понятие «качество», на самом деле, не столь очевидно и просто, как это может показаться на первый взгляд. Для любого инженерного продукта существует множество интерпретаций качества, в зависимости от конкретной «системы координат».

Характеристики качества могут требоваться в той или иной степени, могут отсутствовать или могут задавать определенные требования, все это может быть результатом определенного компромисса (что вполне перекликается с пониманием «приемлемого качества», как менее жесткой точки зрения на обеспечение качества, как достижение совершенства).

Движущей силой программных проектов является желание создать программное обеспечение, обладающее определенной ценностью (значимое для решения определенных задач или достижения целей). Ценность программного обеспечения может выражаться в форме стоимости, а может и нет. Заказчик, как правило, имеет свое представление о максимальных стоимостных вложениях, возврат которых ожидается в случае достижения основных целей создания программного обеспечения. Заказчик также может иметь определенные ожидания в отношении качества ПО. Иногда заказчики не задумываются о вопросах качества и связанной с ними стоимостью. Являются ли характеристики качества чисто декоративными (умозрительными) или, все же, это неотъемлемая часть программного обеспечения? Ответ, вероятно, находится где-то посередине, как почти всегда бывает в таких случаях, и является предметом обсуждения степени вовлечения заказчика в процесс принятия решений и полного понимания заказчиком стоимости и выгоды, связанной с достижением того или иного уровня качества. В идеальном случае, большинство такого рода решений должно приниматься в процессе работы с требованиями, однако эти вопросы могут (и должны) подниматься на протяжении всего жизненного цикла программного обеспечения. Не существует каких-то «стандартных» правил того, как именно необходимо принимать такие решения. Однако инженеры должны быть способны представить различные альтернативы (в достижении различного уровня качества) и их стоимость.

1. МЕТОДОЛОГИЯ ОБЕСПЕЧЕНИЯ КАЧЕСТВА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

1.1. Международная стандартизация в области качества программного обеспечения

Для выполнения разнообразных функций как в бизнесе, так и для персонального назначения в современных условиях все большее распространение получают программные продукты и программные вычислительные системы. Реализация целей и задач для удовлетворения личных потребностей, для успеха в бизнесе и для безопасности человека опирается на высококачественные программное обеспечение и системы. Высококачественные программные продукты и программные вычислительные системы имеют важное для заинтересованных сторон значение в производстве материальных ценностей и предотвращении возможных негативных последствий.

У программных продуктов и программных вычислительных систем много заинтересованных сторон, в число которых входят разработчики, приобретатели и пользователи. Ключевыми факторами в обеспечении полезности для заинтересованных сторон являются подробная спецификация и оценка качества программного обеспечения и программных вычислительных систем. Оценка может быть выполнена на основе определения необходимых и требуемых характеристик качества, связанных с задачами заинтересованных сторон и целями системы, включая характеристики качества, относящиеся к системе программного обеспечения и данным, а кроме того, и воздействие системы на ее заинтересованные стороны. Важно, чтобы, по возможности, характеристики качества были определены, измерены и оценены с использованием проверенных или широко распространенных показателей и методов измерения.

Профессиональное сообщество, занимающееся разработкой программных продуктов и программных вычислительных систем, руководствуется в своей деятельности серией международных стандартов SQuaRE («Systems and software Quality Requirements and Evaluation») – «Требования и оценка качества систем и программного обеспечения».

Одним из стандартов этой серии является международный стандарт ИСО/МЭК 25010:2011 «Системная и программная инженерия. Требования и оценка качества систем и программного обеспечения (SQuaRE). Модели качества систем и программных продуктов» (ISO/IEC 25010:2011 «Systems and software engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) – System and Software Quality Models»). В Российской Федерации в настоящее время действует ГОСТ Р ИСО/МЭК 25010-2015 Информационные технологии. Системная и программная инженерия. Требования и оценка качества систем и программного обеспечения (SQuaRE). Модели качества систем и программных продуктов [12].

Организация серии международных стандартов SQuaRE, представленная на рис. 2, состоит из семейства стандартов, называемых также разделами.

<table>
<tr><td rowspan="3">Раздел
«Требования к качеству»
ИСО/МЭК 2503n</td><td>Раздел
«Модель качества»
ИСО/МЭК 2501n</td><td rowspan="3">Раздел
«Оценка качества»
ИСО/МЭК 2504n</td></tr>
<tr><td>Раздел
«Менеджмент качества»
ИСО/ МЭК 2500n</td></tr>
<tr><td>Раздел
«Измерение качества»
ИСО/МЭК 2502n</td></tr>
</table>

Рис. 2. Организация серии международных стандартов SQuaRE

Серия стандартов SQuaRE состоит из следующих разделов стандартов:

- ИСО/МЭК 2500n – раздел «Менеджмент качества». Содержит общие модели, термины и определения, используемые далее во всех других международных стандартах серии SQuaRE. Представлены требования и методические материалы, касающиеся функций поддержки, которые отвечают за управление требованиями к программному продукту, его спецификацией и оценкой;
- ИСО/МЭК 2501n – раздел «Модель качества». Содержит детализированные модели качества вычислительных систем и программного обеспечения, качества при использовании и качества данных. Кроме того, представлено практическое руководство по использованию модели качества;
- ИСО/МЭК 2502n – раздел «Измерение качества». Включает в себя эталонную модель измерения качества программного продукта, математические определения показателей качества и практическое

руководство по их использованию. Представлены показатели внутреннего качества программного обеспечения, показатели внешнего качества программного обеспечения и показатели качества при использовании. Кроме того, определены и представлены элементы показателей качества (ЭПК), формирующие основу для вышеперечисленных показателей;

- ИСО/МЭК 2503n – раздел «Требования к качеству». Определяет требования к качеству на основе моделей качества и показателей качества. Такие требования к качеству могут использоваться в процессе формирования требований к качеству программного продукта перед разработкой или как входные данные для процесса оценки;

- ИСО/МЭК 2504n – раздел «Оценка качества». Формулирует требования, рекомендации и методические материалы для оценки программного продукта, выполняемой как оценщиками, так и заказчиками или разработчиками.

1.2. Модели качества программного обеспечения

Качество программного обеспечения (software quality) – это степень удовлетворения программным продуктом заявленных и подразумеваемых потребностей различных заинтересованных сторон при использовании в указанных условиях.

Заинтересованные стороны – это следующие три типа пользователей:

1. Основной пользователь – лицо, взаимодействующее с системой для достижения основных целей.

2. Вторичные пользователи – лица, осуществляющие поддержку, например:

- провайдер контента, системные инженер/администратор, руководитель безопасности;

- специалист по обслуживанию, анализатор, специалист по портированию, установщик.

3. Косвенный пользователь – лицо, которое получает результаты, но не взаимодействует с системой.

Эти заявленные и подразумеваемые потребности представлены в международных стандартах серии SQuaRE посредством моделей качества.

Модель качества (quality model) – это определенное множество характеристик и взаимосвязей между ними, которые обеспечивают основу для определения требований к качеству и оценки качества.

Такие модели представляют качество продукта в виде разбивки на классы характеристик, которые в отдельных случаях далее разделяются на

подхарактеристики. (Некоторые подхарактеристики разделяются далее на под-подхарактеристики.) Подобная иерархическая декомпозиция обеспечивает удобную разбивку качества продукта на классы. Однако множество подхарактеристик, связанных с характеристикой, избранной для представления типичных проблем, необязательно будет исчерпывающим.

Измеримые, связанные с качеством свойства системы, называют свойствами качества, связанными с соответствующими показателями качества.

Свойства качества – это неотъемлемые свойства программного обеспечения, которые обеспечивают качество. Свойства качества могут быть разделены на одну или несколько подхарактеристик.

Измеряются свойства качества посредством метода измерения. Метод измерения представляет собой логическую последовательность операций, используемых для количественного определения свойств относительно конкретной шкалы. Результат применения метода измерения называют элементом показателя качества (ЭПК).

Характеристики и подхарактеристики качества могут быть количественно определены с помощью функции измерения. Функция измерения – это алгоритм, используемый для объединения элементов показателя качества. Показателем качества программного обеспечения называют результат применения функции измерения.

Таким образом, показатели качества программного обеспечения становятся количественными показателями характеристик и подхарактеристик качества. Для измерения характеристики или подхарактеристики качества могут быть использованы несколько показателей качества программного обеспечения.

На рис.3 представлена эталонная модель измерения качества программной продукции.

Показатель качества (quality measure) – это показатель, получаемый как функция измерения двух или больше значений элементов показателя качества.

Функция измерения (measurement function) – это алгоритм или вычисление, выполняемое для комбинации не менее чем двух элементов показателя качества.

Элемент показателя качества (quality measure element) – это показатель, определенный в терминах свойства и метода измерения для количественного определения этого свойства, включая выборочно преобразования с помощью математической функции [13].

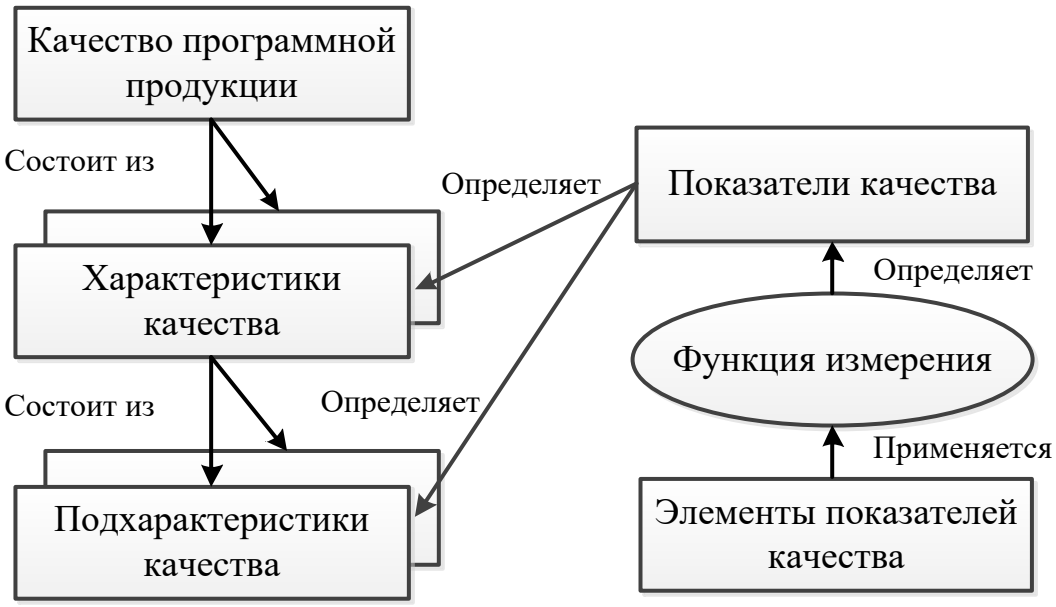


Рис.3. Эталонная модель измерения качества программного продукта

Качество программного продукта может быть оценено путем измерения либо внутренних свойств (обычно это статические показатели промежуточных продуктов), либо внешних свойств (как правило, оценивая поведение кода при выполнении) или посредством измерения свойства качества при использовании (когда продукт используется в реальных или моделируемых условиях) (см. рис. 4).

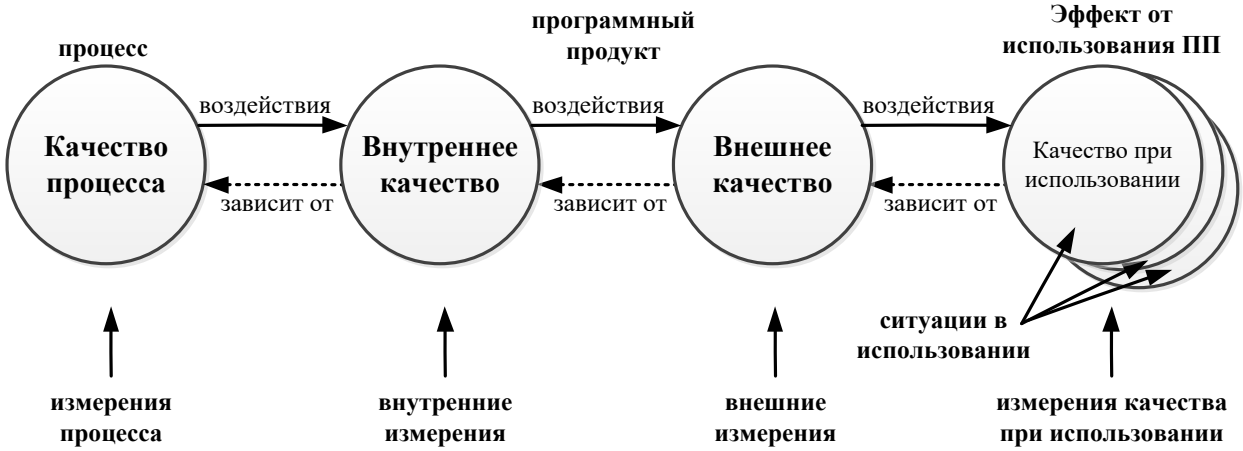


Рис. 4. Качество в жизненном цикле

Повышение качества процесса (качества любого из процессов жизненного цикла, определенных в ГОСТ Р ИСО/МЭК 12207-2010 [10] и ГОСТ Р ИСО/МЭК 15288-2005 [6]) способствует повышению качества продукции, а повышение качества продукции – повышению качества при использовании системы. В связи с этим оценка и улучшение процесса являются средствами повышения качества продукции, а оценка и повышение качества продукции, в

свою очередь, являются одним из средств повышения качества при использовании системы. Аналогичным образом оценка качества при использовании системы может обеспечить обратную связь для улучшения продукта, а оценка продукта может обеспечить обратную связь для улучшения процесса.

Модель жизненного цикла качества (см. рис. 5) рассматривает качество на трех основных этапах жизненного цикла программного продукта:

- на этапе разработки продукта, когда предметом рассмотрения являются показатели внутреннего качества программного обеспечения;
- на этапе тестирования продукта, когда рассматриваются показатели внешнего качества программного обеспечения;
- на этапе использования продукта, когда оценивается качество при использовании.

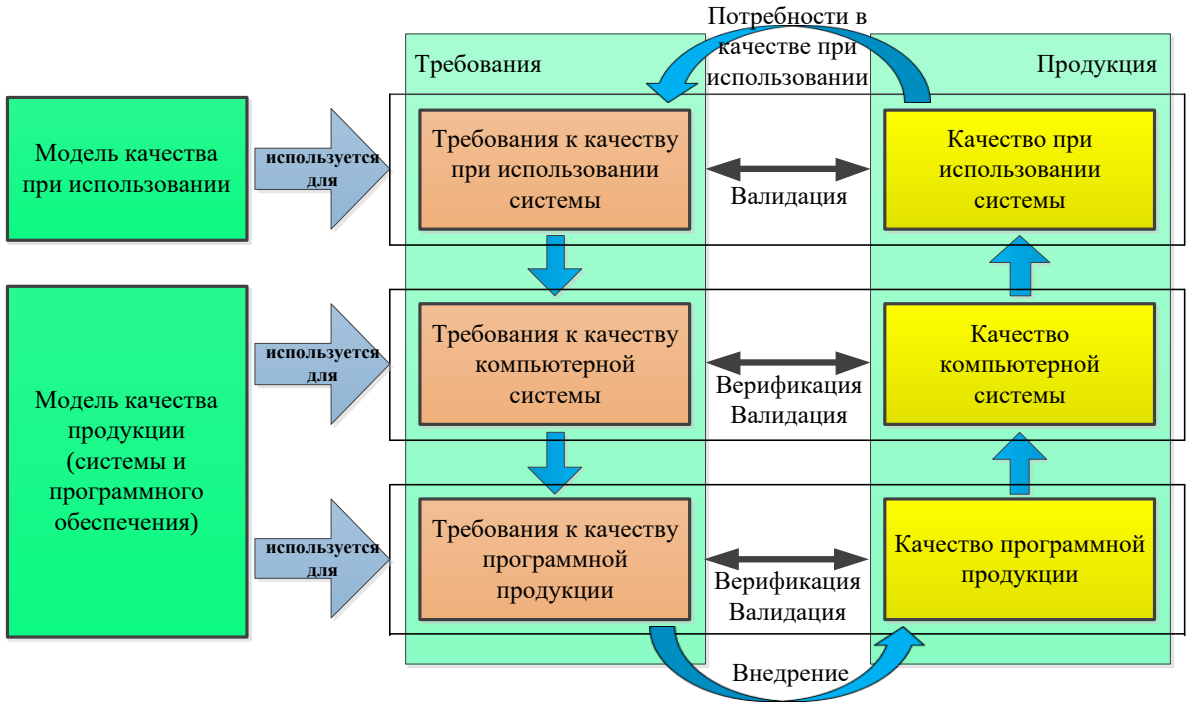


Рис. 5. Модель жизненного цикла качества системы (программного обеспечения)

Показатель внутреннего качества программного обеспечения (internal measure of software quality) – это показатель степени, с которой множество статических свойств программной продукции удовлетворяет заявленным и подразумеваемым требованиям для этой продукции при использовании в заданных условиях (например, сложность, количество, серьезность и частота отказов из-за дефектов, определенных при тестировании; ошибки спецификации, проектирования и кодирования).

Статические свойства включают в себя те свойства, которые имеют отношение к архитектуре программного обеспечения, его структуре и компонентам. Статические свойства могут быть верифицированы путем визуального анализа, проверки, моделирования и/или с использованием автоматических средств.

Показатель внешнего качества программного обеспечения (external measure of software quality) – это показатель степени, с которой программная продукция позволяет функционированию системы удовлетворять заявленным и реализованным требованиям к этой системе, включая программное обеспечение, при использовании в заданных условиях (например, число отказов, обнаруженных во время тестирования).

Функционирование системы может быть проверено при верификации и/или валидации с помощью выполнения функций программным обеспечением во время тестирования и эксплуатации.

Качество при использовании (quality in use) – это степень, с которой продукция или система могут быть применены определенными пользователями для удовлетворения их требований в достижении целей эффективности (в том числе и экономической), избегания риска, удовлетворенности и охвата контекста в заданных условиях использования.

Кроме того, модель жизненного цикла качества системы (программного обеспечения) требует, чтобы достижение надлежащих уровней качества для каждого типа качества было неотъемлемой частью процессов разработки, включая определение требований, реализацию и подтверждение достоверности результатов.

Требования к качеству при использовании определяют требуемые уровни качества с точки зрения пользователей. Основой этих требований являются требования пользователей и других заинтересованных сторон (таких как разработчики программного обеспечения, системные интеграторы, приобретатели или владельцы). Выполнение требований к качеству при использовании является целью валидации пользователем программного продукта. Требования к характеристикам качества при использовании должны быть утверждены в спецификации требований к качеству с применением при оценке продукта критериев для показателей качества при использовании.

Требования к качеству при использовании системы обеспечивают идентификацию и определение требований к внешнему качеству программного обеспечения. Например, определенные типы пользователей могут решить определенные задачи в требуемое время.

Требования к показателям внешнего качества компьютерной системы определяют требуемые уровни качества с точки зрения извне. Они включают в себя требования, основой которых являются требования к качеству заинтересованных сторон, включая требования к качеству при использовании. Выполнение требований к внешнему качеству программного обеспечения является целью технической верификации и валидации программного продукта. Требования к показателям внешнего качества должны быть количественно утверждены в спецификации требований к качеству с применением при оценке продукта критериев для показателей внешнего качества.

Требования к показателям внешнего качества обеспечивают идентификацию и определение требований к показателям внутреннего качества программного обеспечения. Например, пользователи адекватно реагируют на сообщения об ошибках и успешно отменяют ошибки. Оценка внешнего качества может использоваться для прогнозирования качества при использовании систем.

Требования к показателям внутреннего качества программного обеспечения определяют уровень требуемого качества с точки зрения представления продукта изнутри. Они включают в себя требования, основанные на требованиях к внешнему качеству. Требования к показателям внутреннего качества программного обеспечения используются для определения свойств промежуточных программных продуктов (спецификации, исходного кода и т.д.). Кроме того, требования к внутреннему качеству программного обеспечения могут быть использованы для определения свойств поставляемого компонента и неисполнимых программных продуктов, таких как документация и руководства. Требования к показателям внутреннего качества программного обеспечения могут служить целью верификации на различных этапах разработки. Они могут также использоваться для определения стратегии разработки и критериев оценки и проверки в ходе разработки.

Показатели внутреннего качества программного обеспечения могут быть использованы для прогноза показателей внешнего качества программного обеспечения. Например, все сообщения об ошибках определяют корректирующее действие, и любой ввод данных пользователем может быть отменен.

В международном стандарте ISO/IEC 25030:2007 Software engineering. Software product quality requirements and evaluation (SQuaRE). Quality requirements (Разработка программного обеспечения. Требования к качеству и

оценка качества программного продукта. Требования к качеству). приводятся требования к качеству программного обеспечения, а ГОСТ Р ИСО/МЭК 25040-2014 [14] определяет процесс оценки качества программного обеспечения.

Для достижения целей качества в процессе разработки модели и связанные с ними показатели могут быть использованы для управления деятельностью по разработке и реализации. Ключевое значение моделей качества и связанных с ними показателей состоит в возможности получить оценку качества программного обеспечения на ранних стадиях. Эта оценка может быть использована для управления качеством на протяжении всего жизненного цикла и предсказания того, насколько, вероятно, будут удовлетворены требования к качеству.

Модели качества программных продуктов часто включают метрики для определения уровня каждой характеристики качества, присущей продукту. Если характеристики качества выбраны правильно, такие измерения могут поддержать качество (уровень качества) многими способами. Метрики могут помочь в управлении процессом принятия решений. Метрики могут способствовать поиску проблемных аспектов и узких мест в процессах. Метрики являются инструментом оценки качества своей работы самими инженерами-разработчиками с точки зрения более долгосрочного процесса совершенствования достигаемого качества.

С увеличением внутренней сложности и изощренности программного обеспечения, вопросы качества выходят далеко за рамки констатации факта – работает или не работает программное обеспечение. Вопрос ставится – насколько хорошо достигаются количественно оцениваемые цели качества.

К настоящему времени в серии SQuaRE имеются три модели качества:

- модель качества при использовании;
- модель качества продукта;
- модель качества данных.

Модель качества при использовании включает в свой состав пять характеристик, некоторые из которых, в свою очередь, подразделены на подхарактеристики. Эти характеристики касаются результата взаимодействия при использовании продукта в определенных условиях. Данная модель применима при использовании полных человеко-машинных систем, включая как вычислительные системы, так и программные продукты.

Качество при использовании системы характеризует воздействие продукции (системы или программного продукта) на заинтересованную сторону. Оно определяется качествами программного обеспечения,

аппаратных средств, операционной среды, а также характеристиками пользователей, задач и социальной среды. Все эти факторы вносят свой вклад в качество системы при использовании.

Модель качества продукта включает в свой состав восемь характеристик, которые, в свою очередь, подразделены на подхарактеристики. Характеристики относятся к статическим и динамическим свойствам программного обеспечения и вычислительных систем. Модель применима как к компьютерным системам, так и к программным продуктам.

Модель качества данных, определенная в стандарте ISO/IEC 25012:2008 (Программная инженерия – Требования и оценка качества программной продукции (SQuaRE) – Модель качества данных) дополняет модель качества продукта.

Качество данных (data quality) – это степень, с которой характеристики данных удовлетворяют заявленным и подразумеваемым требованиям при использовании в заданных условиях

Как присущее качество данных, так и измеренное внутреннее качество программного обеспечения вносят свой вклад в общее качество компьютерной системы.

Показатели системно-зависимого качества данных и показатели внешнего качества программного обеспечения оценивают аналогичные аспекты компьютерной системы. Различие лишь в том, что показатели системно-зависимого качества данных фокусируются на непосредственно их вкладе в качество компьютерной системы, в то время как показатели внешнего качества программного обеспечения фокусируются на вкладе именно программного обеспечения. Однако в обоих случаях измеряются свойства компьютерной системы.

Данные модели обеспечивают множество характеристик качества, в которых заинтересован широкий круг лиц, таких как: разработчики программного обеспечения, системные интеграторы, приобретатели, владельцы, специалисты по обслуживанию, подрядчики, профессионалы обеспечения и управления качеством, пользователи.

1.3. Характеристики качества программного обеспечения

1.3.1. Характеристики качества при использовании.

Модель качества при использовании определяет пять характеристик, связанных с результатами взаимодействия с системой (см. рис. 6). Каждая характеристика применима для различных видов деятельности

заинтересованных лиц, например, для взаимодействия оператора или поддержки разработчика.

Рис. 6. Модель качества при использовании

1. Эффективность, результативность (effectiveness) – точность и полнота, с которой пользователи достигают определенных целей.

2. Производительность (efficiency) – связь точности и полноты достижения пользователями целей с израсходованными ресурсами.

Соответствующие ресурсы могут включать в себя время выполнения задачи (человеческие ресурсы), материалы или финансовые затраты на использование.

3. Удовлетворенность (satisfaction) – способность продукта или системы удовлетворить требованиям пользователя в заданном контексте использования.

Удовлетворенность – это реакция пользователя на взаимодействие с продуктом или системой, которая включает в себя отношение к использованию продукта.

Для пользователей, не взаимодействующих с продуктом или системой непосредственно, имеют значение только выполнение цели и доверие.

3.1 Полноценность (usefulness) – степень удовлетворенности пользователя достижением прагматических целей, включая результаты использования и последствия использования.

3.2 Доверие (trust) – степень уверенности пользователя или другого заинтересованного лица в том, что продукт или система будут выполнять свои функции так, как это предполагалось.

3.3 Удовольствие (pleasure) – степень удовольствия пользователя от удовлетворения персональных требований.

В число персональных требований могут входить потребности получения новых знаний и навыков, личное общение и ассоциации с приятными воспоминаниями.

3.4 Комфорт (comfort) – степень удовлетворенности пользователя физическим комфортом.

4. Свобода от риска (freedom from risk) – способность продукта или системы смягчать потенциальный риск для экономического положения, жизни, здоровья или окружающей среды.

Риск является функцией вероятности возникновения такой угрозы и потенциальных неблагоприятных последствий этой угрозы.

4.1 Смягчение отрицательных последствий экономического риска (economic risk mitigation) – способность продукта или системы смягчать потенциальный риск для финансового положения и эффективной работы, коммерческой недвижимости, репутации или других ресурсов в предполагаемых условиях использования.

4.2 Смягчение отрицательных последствий риска для здоровья и безопасности (health and safety risk mitigation) – способность продукта или системы смягчать потенциальный риск для людей в предполагаемых условиях использования.

4.3 Смягчение отрицательных последствий экологического риска (environmental risk mitigation) – способность продукта или системы смягчать потенциальный риск для имущества или окружающей среды в предполагаемых условиях использования.

5. Покрытие контекста (context coverage) – степень, в которой продукт или система могут быть использованы с эффективностью, результативностью, свободой от риска и в соответствии с требованиями как в первоначально определенных условиях использования, так и в условиях, выходящих за спецификации.

Контекст использования имеет отношение как к качеству при использовании, так и к некоторым характеристикам или подхарактеристикам качества продукта (в этом случае о нем говорят, как об «определенных условиях»).

5.1 Полнота контекста (context completeness) – степень, в которой продукт или система могут быть использованы с эффективностью, результативностью, свободой от риска и в соответствии с требованиями при всех указанных условиях использования.

Полнота контекста может быть задана или измерена либо как степень, в которой продукт может использоваться конкретными пользователями для достижения определенных целей с эффективностью, результативностью, свободой от риска и в соответствии с требованиями во всех намеченных контекстах использования, либо как наличие свойств продукта, которые поддерживают использование во всех намеченных контекстах использования.

Пример – степень, в которой программное обеспечение применимо при использовании маленького экрана, с низкой сетевой пропускной способностью, неквалифицированными пользователями и в отказоустойчивом режиме (например, при отсутствии сети).

5.2 Гибкость (flexibility) – степень, в которой продукт или система могут быть использованы с эффективностью, результативностью, свободой от риска и в соответствии с требованиями в условиях, выходящих за рамки первоначально определенных в требованиях.

Гибкость может быть достигнута путем адаптации продукта для дополнительных групп пользователей, задач и культур.

Гибкость позволяет использовать продукт в условиях обстоятельств, возможностей и индивидуальных настроек, которые не были предусмотрены заранее.

Если продукт не обладает гибкостью, то он не может быть безопасно использован в непредусмотренных условиях.

Гибкость может быть определена либо как степень, до которой продукт может быть использован пользователями непредусмотренного типа для достижения дополнительных целей с эффективностью, результативностью, свободой от риска и в соответствии с требованиями при дополнительных условиях использования, либо как возможность изменения для поддержки адаптации к новым типам пользователей, задач и сред, а также пригодности для индивидуализации.

1.3.2. Характеристики качества продукта.

Модель качества продукта сводит свойства качества системы (программного продукта) к восьми характеристикам. Каждая характеристика, в свою очередь, состоит из ряда соответствующих подхарактеристик (см. рис. 7).

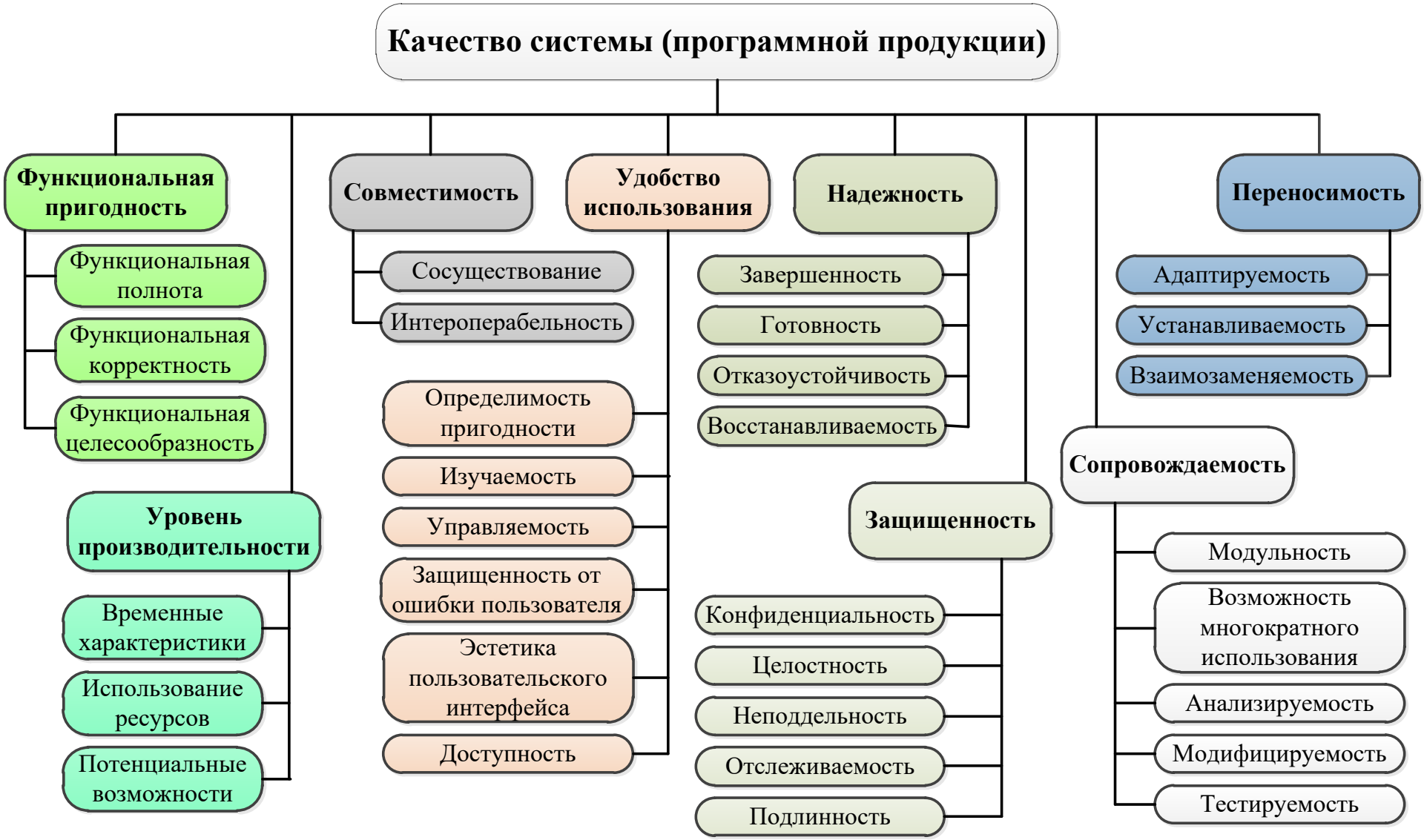


Рис. 7. Модель качества продукта

Модель качества продукта можно применять как для программного продукта, так и для компьютерной системы, в состав которой входит программное обеспечение, поскольку большинство подхарактеристик применимо и к программному обеспечению, и к системам.

1. Функциональная пригодность (functional suitability) – степень, в которой продукт или система обеспечивают выполнение функции в соответствии с заявленными и подразумеваемыми потребностями при использовании в указанных условиях.

Функциональная пригодность относится лишь к соответствию функций заявленным и подразумеваемым потребностям, но не имеет отношения к спецификации функций.

1.1 Функциональная полнота (functional completeness) – степень покрытия совокупностью функций всех определенных задач и целей пользователя.

1.2 Функциональная корректность (functional correctness) – степень обеспечения продуктом или системой необходимой степени точности корректных результатов.

1.3 Функциональная целесообразность (functional appropriateness) – степень функционального упрощения выполнения определенных задач и достижения целей.

Пример – для решения задачи пользователю предоставляется возможность выполнять только необходимые шаги, исключая любые ненужные.

2. Уровень производительности (performance efficiency) – производительность относительно суммы использованных при определенных условиях ресурсов.

Ресурсы могут включать в себя другие программные продукты, конфигурацию программного и аппаратного обеспечения системы и материалы (например, бумагу для печати, носители).

2.1 Временные характеристики (time behaviour) – степень соответствия требованиям по времени отклика, времени обработки и показателей пропускной способности продукта или системы.

2.2 Использование ресурсов (resource utilization) – степень удовлетворения требований по потреблению объемов и видов ресурсов продуктом или системой при выполнении их функций.

2.3 Потенциальные возможности (capacity) – степень соответствия требованиям предельных значений параметров продукта или системы.

В качестве параметров могут быть: возможное количество сохраняемых элементов, количество параллельно работающих пользователей, емкость канала, пропускная способность по транзакциям и размер базы данных.

3. **Совместимость** (compatibility) – способность продукта, системы или компонента обмениваться информацией с другими продуктами, системами или компонентами, и/или выполнять требуемые функции при совместном использовании одних и тех же аппаратных средств или программной среды.

3.1 Сосуществование (совместимость) (co-existence) – способность продукта совместно функционировать с другими независимыми продуктами в общей среде с разделением общих ресурсов и без отрицательного влияния на любой другой продукт.

3.2 Интероперабельность (функциональная совместимость) (interoperability) – способность двух или более систем, продуктов или компонент обмениваться информацией и использовать такую информацию.

4. Удобство использования (usability) – степень, в которой продукт или система могут быть использованы определенными пользователями для достижения конкретных целей с эффективностью, результативностью и удовлетворенностью в заданном контексте использования.

Удобство использования может быть либо задано или измерено как характеристика качества продукта в терминах ее подхарактеристик, либо задано или измерено непосредственно показателями, которые составляют подмножество качества при использовании.

4.1 Определимость пригодности (appropriateness recognizability) – возможность пользователей понять, подходит ли продукт или система для их потребностей, сравним ли с функциональной целесообразностью (functional appropriateness).

Определимость пригодности зависит от возможности распознать уместность продукта или функций системы от первоначальных впечатлений о продукте или системе и/или от какой-либо связанной с ними документации.

Информация, предоставляющая продукт или систему, может включать в себя демонстрации, обучающие программы, документацию, а для веб-сайта – информацию на домашней странице.

4.2 Изучаемость (learnability) – возможность использования продукта или системы определенными пользователями для достижения конкретных целей обучения для эксплуатации продукта или системы с эффективностью, результативностью, свободой от риска и в соответствии с требованиями в указанном контексте использования.

Изучаемость может быть задана или измерена либо как степень возможности использования продукта или системы определенными пользователями для достижения конкретных целей обучения, для эксплуатации продукта или системы с эффективностью, результативностью, свободой от риска и в соответствии с требованиями в указанном контексте использования, либо как свойство продукта, соответствующего пригодности для обучения.

4.3 Управляемость (operability) – наличие в продукте или системе атрибутов, обеспечивающих простое управление и контроль.

Управляемость (operability) соответствует управляемости (controllability), устойчивости к ошибкам (оператора) и согласованности с ожиданиями пользователей.

4.4 Защищенность от ошибки пользователя (user error protection) – уровень системной защиты пользователей от ошибок.

4.5 Эстетика пользовательского интерфейса (user interface aesthetics) – степень «приятности» и «удовлетворенности» пользователя интерфейсом взаимодействия с пользователем.

Это свойство относится к тем свойствам продукта или системы, которые повышают привлекательность интерфейса для пользователя, таким как использование цвета и естественного графического дизайна.

4.6 Доступность (accessibility) – возможность использования продукта или системы для достижения определенной цели в указанном контексте использования широким кругом людей с самыми разными возможностями.

В диапазон возможностей входят ограничения возможностей, связанные с возрастом.

Доступность для людей с ограниченными возможностями может быть задана или измерена либо как степень, в которой продукт или система могут быть применены пользователями с указанными ограниченными возможностями для достижения определенных целей с эффективностью, результативностью, свободой от риска и в соответствии с требованиями в указанном контексте использования, либо как наличие свойств продукта для поддержки доступности.

5. Надежность (reliability) – степень выполнения системой, продуктом или компонентом определенных функций при указанных условиях в течение установленного периода времени.

В программном обеспечении износа не происходит. Проблемы с надежностью возникают из-за недостатков в требованиях, при разработке и реализации или из-за изменений условий использования.

Характеристики функциональной надежности программного обеспечения включают в себя готовность и либо присущие ей, либо внешние влияющие факторы, такие как надежность и доступность (включая отказоустойчивость и восстанавливаемость), безопасность (включая обеспечение конфиденциальности и целостность), пригодность для обслуживания, долговечность и техническую поддержку.

5.1 Завершенность (maturity) – степень соответствия системы, продукта или компонента при нормальной работе требованиям надежности.

Понятие завершенности может также быть применено и к другим характеристикам качества для определения степени соответствия требованиям при нормальной работе.

5.2 Готовность (availability) – степень работоспособности и доступности системы, продукта или компонента.

В общем, готовность можно оценить как долю общего времени, в течение которого система, продукт или компонент находятся в работающем состоянии. Готовность, таким образом, определяется сочетанием завершенности, которая определяет частоту отказов, отказоустойчивости и восстанавливаемости, которая, в свою очередь, определяет продолжительность времени бездействия после каждого отказа.

5.3 Отказоустойчивость (fault tolerance) – способность системы, продукта или компонента работать как предназначено, несмотря на наличие дефектов программного обеспечения или аппаратных средств.

5.4 Восстанавливаемость (recoverability) – способность продукта или системы восстановить данные и требуемое состояние системы в случае прерывания или сбоя.

В некоторых случаях после сбоя вычислительная система находится в нерабочем состоянии некоторое время, продолжительность которого определяется ее восстанавливаемостью.

6. Защита, защищенность (security) – степень защищенности информации и данных, обеспечиваемая продуктом или системой путем ограничения доступа людей, других продуктов или систем к данным в соответствии с типами и уровнями авторизации.

Защищенность применима также и к данным при передаче в случаях, когда данные сохраняются непосредственно в продукте или системе или вне их.

Жизнестойкость (survivability), т.е. степень, в которой продукт или система продолжают выполнять свою миссию, предоставляя основные услуги своевременно, несмотря на присутствие атак, обеспечивается восстанавливаемостью.

Защищенность, иммунитет (immunity), т.е. степень устойчивости продукта или системы к атакам, обеспечивается целостностью.

Защищенность (security) вносит свой вклад в доверие (trust).

6.1 Конфиденциальность (confidentiality) – обеспечение продуктом или системой ограничения доступа к данным только для тех, кому доступ разрешен.

6.2 Целостность (integrity) – степень предотвращения системой, продуктом или компонентом несанкционированного доступа или модификации компьютерных программ или данных.

6.3 Неподдельность (non-repudiation) – степень, с которой может быть доказан факт события или действия таким образом, что этот факт не может быть отвергнут когда-либо позже.

6.4 Отслеживаемость (accountability) – степень, до которой действия объекта могут быть прослежены однозначно.

6.5 Подлинность (authenticity) – степень достоверности (тождественности) объекта или ресурса требуемому объекту или ресурсу.

7. Сопровождаемость, модифицируемость (maintainability) – результативность и эффективность, с которыми продукт или система могут быть модифицированы предполагаемыми специалистами по обслуживанию.

Модификация может включать в себя исправления, улучшения или адаптацию программного обеспечения к изменениям в условиях использования, в требованиях и функциональных спецификациях. Модификации могут быть выполнены как специализированным техническим персоналом, так и рабочим или операционным персоналом и конечными пользователями.

Сопровождаемость включает в себя установку разного рода обновлений.

Сопровождаемость можно интерпретировать либо как присущее продукту или системе свойство, упрощающее процесс обслуживания, либо как качество при использовании, проверенное на практике специалистами по обслуживанию в целях поддержки продукта или системы.

7.1 Модульность (modularity) – степень представления системы или компьютерной программы в виде отдельных блоков таким образом, чтобы изменение одного компонента оказывало минимальное воздействие на другие компоненты.

7.2 Возможность многократного использования (reusability) – степень, в которой актив может быть использован в нескольких системах или в создании других активов.

Под активом (asset) подразумевается что-либо, имеющее ценность для человека или организации, т.е. такие продукты деятельности, как документы требований, модули исходного кода, определения измерений и т.д.

7.3 Анализируемость (analysability) – степень простоты оценки влияния изменений одной или более частей на продукт или систему или простоты диагностики продукта для выявления недостатков и причин отказов, или простоты идентификации частей, подлежащих изменению.

Конкретная реализация продукта или системы может включать в себя механизмы анализа собственных дефектов и формирования отчетов об отказах и других событиях.

7.4 Модифицируемость (modifiabiIity) – степень простоты эффективного и рационального изменения продукта или системы без добавления дефектов и снижения качества продукта.

Модифицируемость – это сочетание изменяемости и устойчивости.

Реализация модификации включает в себя кодирование, разработку, документирование и проверку изменений.

На модифицируемость могут оказывать влияние модульность и анализируемость.

7.5 Тестируемость (testability) – степень простоты эффективного и рационального определения для системы, продукта или компонента критериев тестирования, а также простоты выполнения тестирования с целью определения соответствия этим критериям.

8. Переносимость, мобильность (portability) – степень простоты эффективного и рационального переноса системы, продукта или компонента из одной среды (аппаратных средств, программного обеспечения, операционных условий или условий использования) в другую.

Переносимость можно интерпретировать либо как присущее продукту или системе свойство продукта или системы, упрощающее процесс переноса, либо как качество при использовании, предназначенное для переноса продукта или системы.

8.1 Адаптируемость (adaptability) – степень простоты эффективной и рациональной адаптации для отличающихся или усовершенствованных аппаратных средств, программного обеспечения, других операционных сред или условий использования.

В адаптируемость входит и масштабируемость внутренних потенциальных возможностей (например, экранных полей, таблиц, объемов транзакции, форматов отчетов и т.д.).

Адаптация может быть выполнена как специализированным техническим персоналом, так и рабочим или операционным персоналом и конечными пользователями.

Если система должна быть адаптирована конечным пользователем, то адаптируемость соответствует пригодности для индивидуализации.

8.2 Устанавливаемость (installability) – степень простоты эффективной и рациональной, успешной установки и/или удаления продукта или системы в заданной среде.

Если продукт или система должны устанавливаться конечным пользователем, устанавливаемость может повлиять на результирующие функциональную целесообразность и управляемость.

8.3 Взаимозаменяемость (replaceability) – способность продукта заменить другой конкретный программный продукт для достижения тех же целей в тех же условиях.

Взаимозаменяемость новой версии программного продукта важна для пользователя при обновлении продукта.

Во взаимозаменяемость могут быть включены атрибуты как устанавливаемости, так и адаптируемости.

Взаимозаменяемость снижает риск блокировки таким образом, что, например, при стандартизации форматов файлов допускается применение других программных продуктов вместо используемого.

ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ К РАЗДЕЛУ 1

1. Основные нормативно-технические документы в области качества программного обеспечения.
2. Дайте определение качества программного обеспечения.
3. Дайте определение модели качества.
4. Дайте определение показателя качества.
5. Дайте определение элемента показателя качества.
6. Опишите эталонную модель измерения качества программного продукта.
7. Охарактеризуйте модель жизненного цикла качества системы (программного обеспечения).
8. Дайте определения показателей внешнего и внутреннего качества программного обеспечения.
9. Охарактеризуйте основные модели качества.
10. Опишите основные характеристики и подхарактеристики качества при использовании.
11. Опишите основные характеристики и подхарактеристики качества продукта.

2. ОЦЕНИВАНИЕ СЛОЖНОСТИ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

2.1. Понятие сложности программного обеспечения

Понятие «сложность ПО» конкретизируется при установлении его связи с конкретными ресурсами, необходимыми для проектирования и эксплуатации ПО. Это понятие многогранно, причем одна из «граней» обычно доминирует над остальными. Возможны ситуации, когда две или более составляющих сложности более или менее равновесны, тогда приходится применять многокритериальные оценки сложности.

На практике сложность алгоритмов и реализующих их программ целесообразно характеризовать минимальными затратами наиболее критического ресурса для данного этапа жизненного цикла ПО. На этапах проектирования и эксплуатации необходимы различные ресурсы. Показатели сложности можно разбить на 2 большие группы (см. рис. 8):

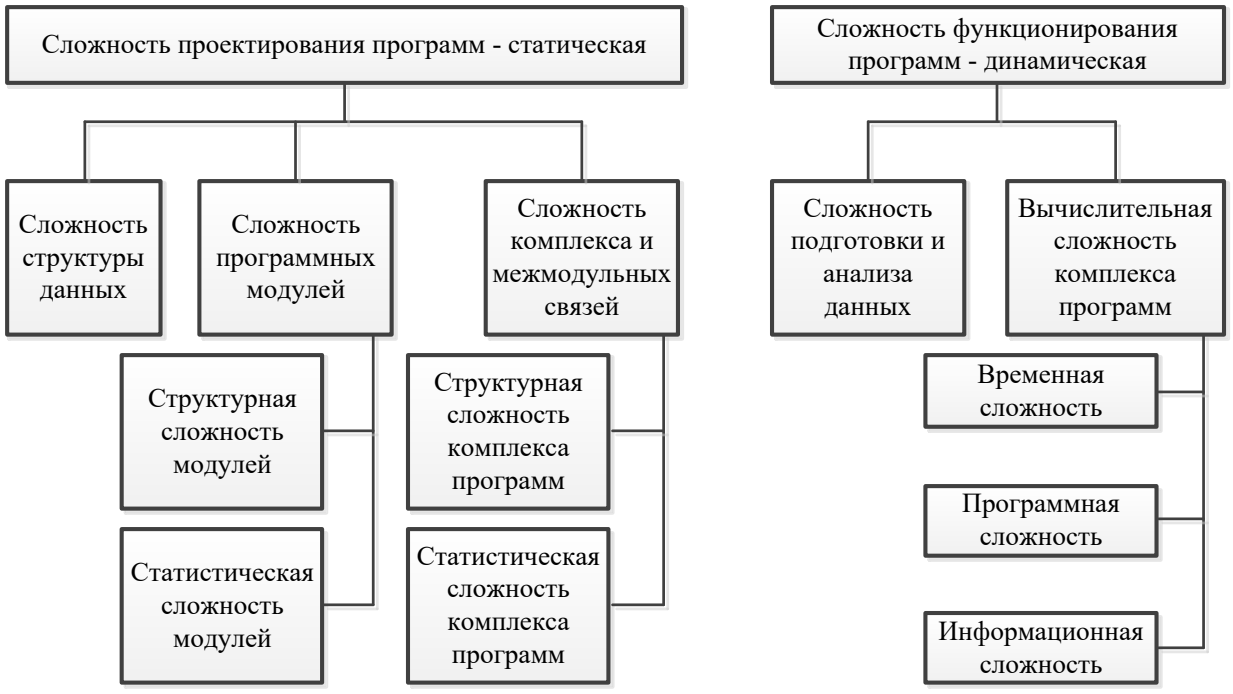


Рис.8. Виды сложности проектирования и функционирования ПО

Сложность структуры данных определяется количеством и структурой глобальных и обменных переменных, регулярностью их размещения в массивах, а также сложностью доступа к этим переменным.

Сложность комплекса и межмодульных связей определяется глубиной взаимодействия модулей и регулярностью структуры межмодульных связей.

Сложность подготовки и анализа данных определяется сложностью функционирования человека совместно с ЭВМ и включает в себя сложность ручной подготовки, корректировки и сверки больших массивов данных.

Вычислительная сложность комплекса программ определяется величиной ресурсов времени и памяти, необходимых для решения задачи.

Временная сложность определяется временем обработки на ЭВМ совокупности исходных данных до получения требуемых результатов. Для алгоритма временная сложность – это время счета, затрачиваемое программой для получения результатов на эталонной ЭВМ в зависимости от объема исходных данных.

Программная сложность характеризуется длиной программы или объемом памяти ЭВМ, необходимым для размещения программного комплекса (числом символов, необходимым для полного описания программы).

Информационная сложность характеризуется объемом памяти, необходимым для оперативного накопления и хранения данных, используемых для решения задачи (емкостью всех ячеек памяти и регистров, к которым осуществляется обращение при обработке операндов).

Анализ сложности проектирования (создания) программных модулей развивается по двум основным направлениям.

1. Исследование внутренней структуры и функций программных модулей, обобщение характеристик структуры с целью получения значений сложности, в той или иной степени связанных с трудоемкостью разработки модуля (структурное направление). При этом наибольшее внимание уделяется анализу маршрутов исполнения программ и объему тестов, необходимых для проверки программных модулей.

2. Определение некоторых внешних измеряемых характеристик программных модулей. Обобщение этих характеристик позволило представить статистическую сложность программы и связать ее с трудоемкостью их создания и количеством ошибок в модулях (статистическое направление).

Оба направления с различных позиций связывают значения сложности с основным ресурсом при создании программ – трудоемкостью разработки. При этом промежуточными или зависимыми параметрами считаются

интенсивность обнаружения ошибок, объем корректировок программ, размер программы в количестве строк текста или количество условных переходов и т.д.

Основная цель этих исследований – создание методов и средств измерения сложности модулей программ для прогнозирования и контроля показателей их качества, начиная с ранних этапов разработки.

В отличие от большинства отраслей материального производства, в вопросах проектов создания ПО недопустимы простые подходы, основанные на умножении трудоемкости на среднюю производительность труда. Это вызвано, прежде всего, тем, что экономические показатели проекта нелинейно зависят от объема работ, а при вычислении трудоемкости допускается большая погрешность.

Поэтому для решения этой задачи используются комплексные и достаточно сложные методики, которые требуют высокой ответственности в применении и определенного времени на адаптацию (настройку коэффициентов).

Современные комплексные системы оценки характеристик проектов создания ПО могут быть использованы для решения следующих задач:

- предварительная, постоянная и итоговая оценка экономических параметров проекта: трудоемкость, длительность, стоимость;
- оценка рисков по проекту: риск нарушения сроков и невыполнения проекта, риск увеличения трудоемкости на этапах отладки и сопровождения проекта и пр.;
- принятие оперативных управленческих решений – на основе отслеживания определенных метрик проекта можно своевременно предупредить возникновение нежелательных ситуаций и устранить последствия непродуманных проектных решений.

В общем случае применение метрик позволяет руководителям проектов и предприятий изучить сложность разработанного или даже разрабатываемого проекта, оценить объем работ, стилистику разрабатываемой программы и усилия, потраченные каждым разработчиком для реализации того или иного решения. Однако метрики могут служить лишь рекомендательными характеристиками, ими нельзя полностью руководствоваться, так как при разработке ПО программисты, стремясь минимизировать или максимизировать ту или иную меру для своей программы, могут прибегать к хитростям вплоть до снижения эффективности работы программы. Кроме того, если, к примеру, программист написал малое количество строк кода или внес небольшое число структурных изменений, это вовсе не значит, что он ничего не делал, а может означать, что дефект программы было очень

сложно отыскать. Последняя проблема, однако, частично может быть решена при использовании метрик сложности, т.к. в более сложной программе ошибку найти сложнее.

Метрики сложности программ принято разделять на три основные группы [37]:

1) Метрики размера программ.

2) Метрики сложности потока управления программ.

3) Метрики сложности потока данных программ.

2.2. Метрики размера программ

Эти метрики базируются на определении количественных характеристик, связанных с размером программы, и отличаются относительной простотой. К наиболее известным метрикам данной группы относятся:

- число операторов программы;
- количество строк исходного текста;
- набор метрик Холстеда;
- метрики Джилба.

Метрики этой группы ориентированы на анализ исходного текста программ. Поэтому они могут использоваться для оценки сложности промежуточных продуктов разработки.

2.2.1. Количество строк кода

Самой элементарной метрикой является **количество строк кода** – **LOC** (Lines of Code) или **SLOC** (Source Lines of Code).

Данная метрика была изначально разработана для оценки трудозатрат по проекту. Однако из-за того, что одна и та же функциональность может быть разбита на несколько строк или записана в одну строку, метрика стала практически неприменимой с появлением языков, в которых в одну строку может быть записано больше одной команды. Поэтому различают логические и физические строки кода.

Физические строки кода – это общее число строк исходного кода, включая комментарии и пустые строки.

Логические строки кода – это количество команд программы.

Такой вариант описания также имеет свои недостатки, так как сильно зависит от используемого языка программирования и стиля программирования.

Кроме метрик **LOC** (**SLOC**) к количественным характеристикам относят также:

- количество пустых строк;

- количество комментариев;

- процент комментариев (отношение числа строк, содержащих комментарии к общему количеству строк, выраженное в процентах);

- среднее число строк для функций (классов, файлов);

- среднее число строк, содержащих исходный код для функций (классов, файлов);

- среднее число строк для модулей.

Иногда важно не просто посчитать количество строк комментариев в коде и просто соотнести с логическими строчками кода, а узнать плотность комментариев. В таких случаях можно оценить стилистику программы F.

$$F = N_{ком}/N_{стр} ,$$

где $N_{ком}$ – количество комментариев в программе; $N_{стр}$ – количество строк или операторов исходного текста.

Исходя из практического опыта принято считать, что $F >= 0{,}1$, т. е. на каждые десять строк программы должен приходиться минимум один комментарий. Как показывают исследования, комментарии распределяются по тексту программы неравномерно: в начале программы их избыток, а в середине или в конце – недостаток. Это объясняется тем, что в начале программы, как правило, расположены операторы описания идентификаторов, требующие более «плотного» комментирования. Кроме того, в начале программы также расположены «шапки», содержащие общие сведения об исполнителе, характере, функциональном назначении программы и т. п.

Для оценивания стилистики F программа разбивается на n равных фрагментов и вычисляются оценки для каждого фрагмента по формуле

$$F_i = \text{SIGN}\ (N_{ком.\ i} / N_{стр.i} - 0{,}1),$$

где $N_{ком.i}$ – количество комментариев в i-м фрагменте, $N_{стр.i}$ – общее количество строк кода в i-м фрагменте.

Тогда общая оценка для всей программы будет определяться следующим образом:

$$F = \sum_{i=1}^{n} F_i \ .$$

Уровень комментированности программы считается нормальным, если выполняется условие

$$F = n.$$

В противном случае какой-либо фрагмент программы дополняется комментариями до нормального уровня.

Пример из жизни: оценка по количеству строк в коде влечёт за собой соблазн написать побольше строк, дабы взять побольше денег. Разумеется, об оптимизации в таком продукте никто уже думать не станет. Вспомним историю о том, как планетарный центр аутсорсинга – Индия, после того, как заказчики вменили им метрику **LOC**, на второй день показал удвоение и утроение строк кода.

И главное помнить: метрика **LOC** (**SLOC)** не отражает трудоемкости по созданию программы.

2.2.2. Метрики Холстеда

Также к группе метрик, основанных на подсчете некоторых единиц в коде программы, относят метрики Холстеда [18], которые позволяют оценить ПО в плане корректности, сложности и надежности.

В любой реализации ПО на любом языке можно выделить измеримые параметры: **операнды** (переменные или константы) и **операторы** (символы или комбинации символов, влияющие на формирование значений и порядок использования операндов)

Определение статистической сложности программ базируется на возможности их представления, состоящего только из операторов и операндов.

Основные измеряемые параметры:

- число уникальных операторов η_1, включая символы-разделители, имена процедур и знаки операций (словарь операторов);

- число уникальных операндов η_2, (словарь операндов);

- общее число операторов N_1;

- общее число операндов N_2;

- теоретическое число уникальных операторов η_1^*;

- теоретическое число уникальных операндов η_2^*.

Предложено в качестве показателя сложности использовать сумму числа операторов N_1 и операндов N_2 в программе. Прямой подсчет числа операторов и операндов по тексту сложных программ громоздок, поэтому высказано предположение, что оценка величины $N_c = N_1 + N_2$ может производиться косвенно. Основой такой оценки является связь между длиной текста и размером словаря, из символов которого составлен текст. Путем рассуждений получено выражение для оценки длины программы N_c через используемый словарь $\eta = \eta_1 + \eta_2$.

В качестве исходных данных для расчета оценки N_c используется количество типов операторов η_1 и типов операндов η_2, примененных в программе. Сумма $\eta = \eta_1 + \eta_2$ эквивалентна запасу слов или полному словарю анализируемой программы. Частота использования операторов j-го типа составляет $f_{1,j}$, а операндов – $f_{2,j}$. По аналогии с основными понятиями теории информации предполагается, что частота использования операторов и операндов в программе пропорциональна $\log_2$ от количества их типов. Тогда

$$N_1 = \sum_{j=1}^{\eta_1} f_{1,j} = \eta_1 \log_2 \eta_1;$$
$$N_2 = \sum_{j=1}^{\eta_2} f_{2,j} = \eta_2 \log_2 \eta_2. \quad (1)$$

В результате метрика – **полная длина программы** N_c может оцениваться по количеству типов операторов η_1 и операндов η_2.

$$N_c = \eta_1 \cdot \log_2 \eta_1 + \eta_2 \cdot \log_2 \eta_2. \quad (2)$$

Проведены расчеты, которые показали, что оценки N_c по сложной программе в целом и по составляющим модулям отличаются незначительно.

Пример 1. Программа на языке Паскаль. Задача Евклида.

```
function GCD(a,b:integer):integer;
Label L1;
Var G,R:integer;
Begin
   if (a=0) then begin
      GCD:=b; return;
   end;
   if (b=0) then begin
      GCD:=a; return;
   end;
   L1:
   G:=a/b;
   R:=a-b*G;
   if (R=0) then begin
      GCD:=b; return;
   end;
   a:=b; b:=R; GOTO L1;
end;
```

Операторы и операнды этой программы сведены в табл.1.

Таблица 1– Перечень операторов и операндов

Оператор	Номер (i)	f_{1i}	Операнд	Номер (j)	f_{2j}
:=	1	7	GCD	1	3
=	2	3	G	2	2
if then	3	3	R	3	3
/	4	1	a	4	5
*	5	1	b	5	7
-	6	1	0	6	3
() begin end	7	8			
goto	8	1			
function GCD	9	1			
return	10	3			
;	11	14			

Нетрудно подсчитать, что

$$\eta_1 = 11; \eta_2 = 6; \eta = 11 + 6 = 17;$$

$$N_1 = 43; N_2 = 23; N = 43 + 23 = 66.$$

Наличие постоянного статистического соотношения между η_1 и η_2 или N_1 и N_2 привело к простой линейной аппроксимации расчетной длины программы N_c как функции от количества операторов на языке программирования или от числа команд в машинном коде. В основном приводятся две аппроксимации рассчитанной длины программы для модулей, составленных на машинно-ориентированном языке:

$$N_c = 2,67d \quad (3)$$

$$N_c = 2,77d^{1,0157}, \quad (4)$$

где в качестве параметра использовалось число команд d объектного кода.

Аппроксимации близки по форме, однако они базируются на небольшом статистическом материале, недостаточном для широкого использования предложенных формул и предпочтения одной из них.

Приведенные выражения целесообразно использовать для оценки только потенциально корректных программ, т.е. таких, которые не имеют ошибочной избыточности, увеличивающей количество операторов или операндов без изменения результатов.

Метрика – **объем текста программы** V для определенной задачи зависит от алгоритма и языка, на котором записан текст. Количество символов, необходимых для описания алгоритма, определяется, в частности, полным словарем операторов и операндов данного алгоритма. Полный словарь η характеризует минимальное число бит, которое требуется для описания всех

операторов и операндов на соответствующем языке программирования. Чем выше язык программирования, тем компактнее словарь и тем меньше объем программы в битах, который предложено описывать выражением

$$V = N_c \ \log_2 \eta, \tag{5}$$

где $N_c = N_1 + N_2$.

Тем самым объем программы характеризуется числом двоичных разрядов, необходимых для записи программы. Для сравнения объема программ на различных языках и при различном качестве программирования может быть введен так называемый **потенциальный объем описания программы** *V**, который соответствует наиболее компактному тексту программы для фиксированного алгоритма, написанному на языке самого высокого уровня:

$$V^* = \eta^* \log_2 \eta^* = (\eta_1^* + \eta_2^*) \log_2 (\eta_1^* + \eta_2^*). \tag{6}$$

Значения η_1^* и η_2^* соответствуют минимальному количеству уникальных операторов и операндов, необходимых для полного корректного описания алгоритма решения рассматриваемой задачи. При снижении качества программирования упрощаются по содержанию компоненты текста программы, в результате чего расширяется объем программы при том же исходном алгоритме.

Тем самым качество программирования можно измерить степенью расширения текста программы ($L < 1$) относительно ее потенциального объема:

$$L = V^* / V. \tag{7}$$

Однако потенциальный объем программы *V** обычно неизвестен, так как невозможно для каждого алгоритма сформировать язык высокого уровня, позволяющий его представлять в самом компактном виде.

Поэтому метрику – **уровень качества программы** *L* целесообразно определять косвенно, не используя значения *V**. Для описания любой функции в пределе достаточно двух типов операторов: альтернативы (различения) и простой последовательности операций (присваивания).

В результате минимальное значение $\eta_1^* = 2$, однако, в типовых языках программирования практически используется большее количество уникальных операторов. Разнообразие операндов не ограничивается, однако уровень качества программы тем выше, чем компактнее представляются операнды, т.е. чем ближе их общее количество N_2 к минимально необходимому объему словаря η_2.

Учитывая, что L пропорционально η_1^* / η_1, а также пропорционально η_2 / N_c, комбинируя эти выражения и полагая коэффициент пропорциональности равным единице, можно представить оценку качества программирования в виде

$$L = \frac{\eta_1^*}{\eta_1} \cdot \frac{\eta_2}{N_2} = \frac{2}{\eta_1} \cdot \frac{\eta_2}{N_2}. \quad (8)$$

Выдвинута гипотеза о наличии своеобразного закона сохранения, по которому при любом фиксированном алгоритме произведение уровня качества программирования *L* на потенциальный объем текста программы *V** остается постоянным и характеризует уровень языка программирования, т.е.

$$LV^* = const.$$

Константу в выражении предложено представлять как метрику – **уровень языка программирования** λ**:**

$$\lambda = LV^* = L^2V. \quad (9)$$

Статистические исследования программ показали достаточно стабильные значения оценок уровней языков программирования (0,88 для Ассемблера).

Оценки значений λ для реальных языков программирования показывают бесплодность мелких частных улучшений языков и нецелесообразность создания их версий. Такие усовершенствования не способны заметно изменить качество программ *L* или их объем *V*. В результате практически не изменяется сложность разработки программ, однако теряется возможность их применения в разных проектах.

Рациональная стратегия развития и применения языков должна базироваться на нескольких языках, достаточно различающихся уровнем, на полном исключении версий и подмножеств этих языков, ограничивающих использование программ на одном языке в разных проектах. Величина

$$\lambda = LV^* = L^2V, \quad (10)$$

где все члены в правой части доступны непосредственному измерению, может рассматриваться как интеллектуальное содержание конкретного алгоритма и практически не зависит от языка программирования.

Стабильность этой величины проверялась при программировании одних и тех же алгоритмов на семи наиболее распространенных языках программирования. Отклонение от среднего составило в пределах ±10 % для каждого алгоритма.

Затраты умственных усилий на создание программы отражают ее сложность в процессе разработки. Предположим, что алгоритм, подлежащий программированию, известен и корректен. Разработка программы состоит в

освоении описания алгоритма, составлении блок-схемы программы, написании ее текста и ручной проверке текста для исключения ошибок. Таким образом, отладка с исполнением на ЭВМ, испытания, оформление документации и т.д. при этом не учитываются. В полном цикле проектирования сложных программ учитываемые работы составляют важную и наиболее просто формализуемую часть, однако следует иметь в виду, что они требуют только около 20 % общей трудоемкости создания программы.

Для выполнения перечисленных работ необходимо произвести множество анализов и выборов решений. Количество умственных сравнений и решений характеризует метрику – **интеллектуальные усилия на создание программы** объемом *V*:

$$E = V / L. \qquad (11)$$

Эта величина характеризует среднее количество различаемых элементов, которые составляют программу и которые необходимо разработать. Различать и анализировать компоненты программы приходится на всех этапах ее проектирования, поэтому совокупные затраты на создание программного модуля пропорциональны его интеллектуальной сложности *E*.

Учитывая выражение (7), интеллектуальные усилия можно представить в виде

$$E = V^2 / V^*. \qquad (12)$$

Таким образом, затраты на создание программ по любым алгоритмам пропорциональны квадрату их объема, независимо от языка программирования. Отсюда очевидна рациональность модульных структур программ, так как при делении на модули затраты сокращаются пропорционально квадрату объема (квадрат суммы всегда больше, чем сумма квадратов).

Первичная экспериментальная проверка корреляции интеллектуальной сложности *E* с количеством ошибок в программе дала очень высокий коэффициент корреляции 0,98. Следовательно, величина интеллектуальных усилий может полностью объяснить все ошибки в анализируемых программах. При этом достаточно хорошо сохраняется пропорциональность между количеством ошибок и величиной *E*.

Метрику – **трудоемкость разработки программы** можно определить, если известна интенсивность *S* анализа и принятия решений по каждой компоненте при подготовке программ. На основе психологических экспериментов получены оценки различимости символов и количества решений, принимаемых сосредоточенным человеком в единицу времени в

пределах $5 < S < 20$ в секунду. Кроме того, значение S (Число Страуда) определялось путем анализа интенсивности разработки программ и оказалось $S \sim 18$ в секунду. Эти значения позволяют рассчитать оценку времени программирования в зависимости от объема программы V:

$$\tau_n = E / S = \frac{V}{SL} = \frac{V^2}{SV^*}, \tag{13}$$

так как $L = \frac{\eta_1^*}{\eta_1} \cdot \frac{\eta_2}{N_2} = \frac{2}{\eta_1} \cdot \frac{\eta_2}{N_2}$ и $V = N_c \log_2 \eta$, то

$$\tau_n = \frac{\eta_1 N_2 (N_1 + N_2) \log_2 \eta}{2\eta_2 S}. \tag{14}$$

Это выражение для определения временных трудозатрат на программирование проверялось путем анализа двух серий реальных разработок программ на языках *PL*-1, Фортран.

Экспериментальное значение времени работы специалиста над программой включало:

- время на чтение формулировки задачи, затраты на написание подробного алгоритма и первичной версии программы;
- продолжительность написания окончательной уточненной версии программы и ее проверку за столом;
- время, затраченное на поиск и корректировку ошибок в программе без ее исполнения.

Коэффициент корреляции в первой серии экспериментов на 12 программных модулях разной сложности оказался равным 0,94. Во второй серии для 11 программ получен коэффициент корреляции между экспериментальными и расчетными данными, равный 0,93.

Для случая, когда известна только длина программы N_c, предложена аппроксимация для расчета длительности разработки программ. Аппроксимация базируется на уравнении (14) и предположениях: $\eta_1 = \eta_2 = \eta / 2$ и $N_1 = N_2 = N_c/2$, тогда

$$\tau_n = \frac{N_c^{\,2} \log_2 \eta}{4S}. \tag{15}$$

Таким образом, время разработки и интеллектуальная сложность возрастают квадратично при увеличении длины или объема программы.

Приведенные выражения используются для оценок длины и объема программы, времени и интеллектуальных усилий, затрачиваемых на программирование. В качестве независимых параметров рекомендуется

использовать число η_2^* единых по смыслу операндов (аргументов и результатов) и уровень λ языка программирования.

При применении метрик Холстеда частично компенсируются недостатки, связанные с возможностью записи одной и той же функциональности разным количеством строк и операторов.

2.2.3. Метрика Джилба

Еще одним типом метрик ПО, относящихся к количественным, является метрика Джилба. Она показывают сложность программного обеспечения на основе насыщенности программы условными операторами или операторами цикла. Данная метрика, не смотря на свою простоту, довольно хорошо отражает сложность написания и понимания программы, а при добавлении такого показателя, как максимальный уровень вложенности условных и циклических операторов, эффективность данной метрики значительно возрастает.

Логическая сложность программы определяется как насыщенность программы выражениями типа **IF-THEN-ELSE**. При этом вводятся две характеристики:

CL – абсолютная сложность программы, характеризующаяся количеством операторов условия;

cl – относительная сложность программы, характеризующаяся насыщенностью программы операторами условия, т. е. **cl** определяется как отношение **CL** к общему числу операторов

$$\mathbf{cl} = \mathbf{CL} / N_1.$$

Используя метрику Джилба, ее дополнили еще одной составляющей, а именно характеристикой максимального уровня вложенности оператора **CLI**, что позволило применить ее к анализу циклических конструкций.

2.3. Метрики сложности потока управления программ

Метрики этой группы базируются на анализе управляющего графа программы. Управляющий граф программы, который используют метрики данной группы, может быть построен на основе алгоритмов модулей. Поэтому метрики этой группы могут применяться для оценки сложности промежуточных продуктов разработки.

Как правило, с помощью этих метрик оперируют либо плотностью управляющих переходов внутри программ, либо взаимосвязями этих переходов.

Перед тем как непосредственно описывать сами метрики, для лучшего понимания опишем управляющий граф программы и способ его построения.

Пусть представлена некоторая программа. Для данной программы строится ориентированный граф, содержащий лишь один вход и один выход, при этом вершины графа соотносят с теми участками кода программы, в которых имеются лишь последовательные вычисления, и отсутствуют операторы ветвления и цикла, а дуги соотносят с переходами от блока к блоку и ветвями выполнения программы. Условие при построении данного графа: каждая вершина достижима из начальной, и конечная вершина достижима из любой другой вершины.

Структурная сложность программы определяется числом взаимодействующих компонент, числом связей между компонентами и сложностью их взаимодействия.

При функционировании программы разнообразие ее поведения и разнообразие связей входных и результирующих данных в значительной степени определяются набором путей – маршрутов, по которым исполняется программа.

Установлено, что сложность программного модуля связана не столько с размером программы по числу команд, сколько с числом отдельных путей ее исполнения, существующих в программе.

Маршруты возможной обработки данных должны быть достаточно полно проверены при создании программы и тем самым определяют сложность с позиции ее разработки. Такое измерение сложности можно использовать в качестве показателя для оценки трудоемкости тестирования и обслуживания модуля, а также для оценки потенциальной надежности его функционирования.

Определение сложности на основе количества маршрутов, исполняемых в программе, в ряде работ уточнялось и модифицировалось. Предложено, кроме числа маршрутов, учитывать число условий, являющихся предикатами в операторах принятия решений.

Также предложен более полный индекс сложности, учитывающий четыре фактора: множество операторов, перечень операндов, взаимодействие операторов и структуру передач управления.

На сравнительно небольшом статистическом материале показано, что такой метод оценки сложности ближе к субъективным психологическим экспертным оценкам, чем на основе числа маршрутов.

В ряде исследований подтверждена достаточно высокая адекватность использования структурной сложности программ для оценки трудоемкости

тестирования, вероятности не выявленных ошибок и затрат на разработку программных модулей в целом.

Поэтому основное внимание сосредоточим на анализе сложности программных модулей по числу маршрутов ***M***, необходимых для их проверки, и суммарному числу условий ξ, которое необходимо задать в тестах для прохождения всех маршрутов программы

$$\xi = \sum_{i=1}^{M} \xi_i \, .$$

Маршруты исполнения программного модуля можно условно разделить на два вида.

1. Маршруты исполнения преимущественно вычислительной части программы и преобразования квазинепрерывных переменных.

Такие маршруты предназначены для преобразования многоразрядных величин, являющихся результатами измерения некоторых непрерывных физических характеристик (квазинепрерывные переменные). Для их проверки во всем диапазоне исходных переменных следует выбрать несколько характерных точек (предельные значения и несколько промежуточных), при которых проверяется программа.

Предполагается, что в большинстве остальных промежуточных точек благодаря гладкой функции результирующих значений программу можно не проверять. В точках разрыва функции необходимо планировать дополнительные проверки.

При оценке сложности вычислительных программ необходим учет количества операндов, участвующих в вычислениях.

Сложность проверки такого программного модуля будет определяться числом маршрутов ***M*** исполнения программы и числом обрабатываемых операндов на каждом маршруте.

Расчет показателя сложности программного модуля по такой схеме достаточно сложен и имеет значительную неопределенность из-за произвола при варьировании исходных данных.

В то же время доля вычислительной части во многих сложных комплексах программ управления и обработки информации относительно невелика. Поэтому основное внимание сосредоточено на анализе структурной сложности модулей программ.

2. Маршруты принятия логических решений и преобразования логических переменных.

Этот вид маршрутов является результатом функционирования схем принятия решений и преобразования логических переменных. В результате в

программе образуется множество маршрутов обработки исходных данных, которое определяют сложность структуры программ.

Структурная сложность программного модуля может быть определена путем расчета количества маршрутов ***M*** в программе и сложности каждого маршрута. Эти показатели в совокупности определяют минимальную сложность тестов для проверки программных модулей, а, следовательно, и трудоемкость их разработки и вероятность ошибки в программе.

Наилучшим, по-видимому, является критерий, позволяющий выделить все реальные маршруты исполнения программы при любых сочетаниях исходных данных. Но такое выделение маршрутов трудно формализовать, и оно представляется излишне трудоемким для оценки показателей сложности структуры. Поэтому используются более простые критерии выделении маршрутов, учитывающие только структурные характеристики программных модулей.

Простейший критерий для оценки структурной сложности программ состоит в выборе минимального множества маршрутов программы, охватывающих все последовательности передач управления (ветвления при условных переходах и переключателях) и учитывающих исполнение программы по каждому направлению при ветвлении.

По этому критерию граф программы по управлению должен быть покрыт минимальным набором путей, проходящих через каждый оператор ветвления по каждой дуге. Повторная проверка дуг не оценивается и считается избыточной. При этом в процессе проверки гарантируется выполнение всех передач управления между операторами программы и каждого оператора не менее одного раза.

Существуют алгоритмы, позволяющие минимизировать покрытие маршрутами графов при таком критерии.

Пример 2. На рис. 9 представлен исходный граф модуля программы, содержащий 14 вершин, 20 дуг и 3 цикла.

Такая программа сравнительно невысокой сложности содержит от 100 до 200 команд и может рассматриваться как достаточно типичная. Для полной проверки модуля программы по первому критерию достаточно четырех маршрутов, которые представлены справа. Самый длинный по количеству вершин последний маршрут не охватывает только 3 вершины из 14 и только 6 дуг из 20.

После проверки трех последних маршрутов вне контроля остаются одна вершина и две дуги. Однако при этом критерии не учитывается комбинаторика сочетания условий на разных участках маршрутов, например при сочетаниях ветвлений в вершинах 3 и 12.

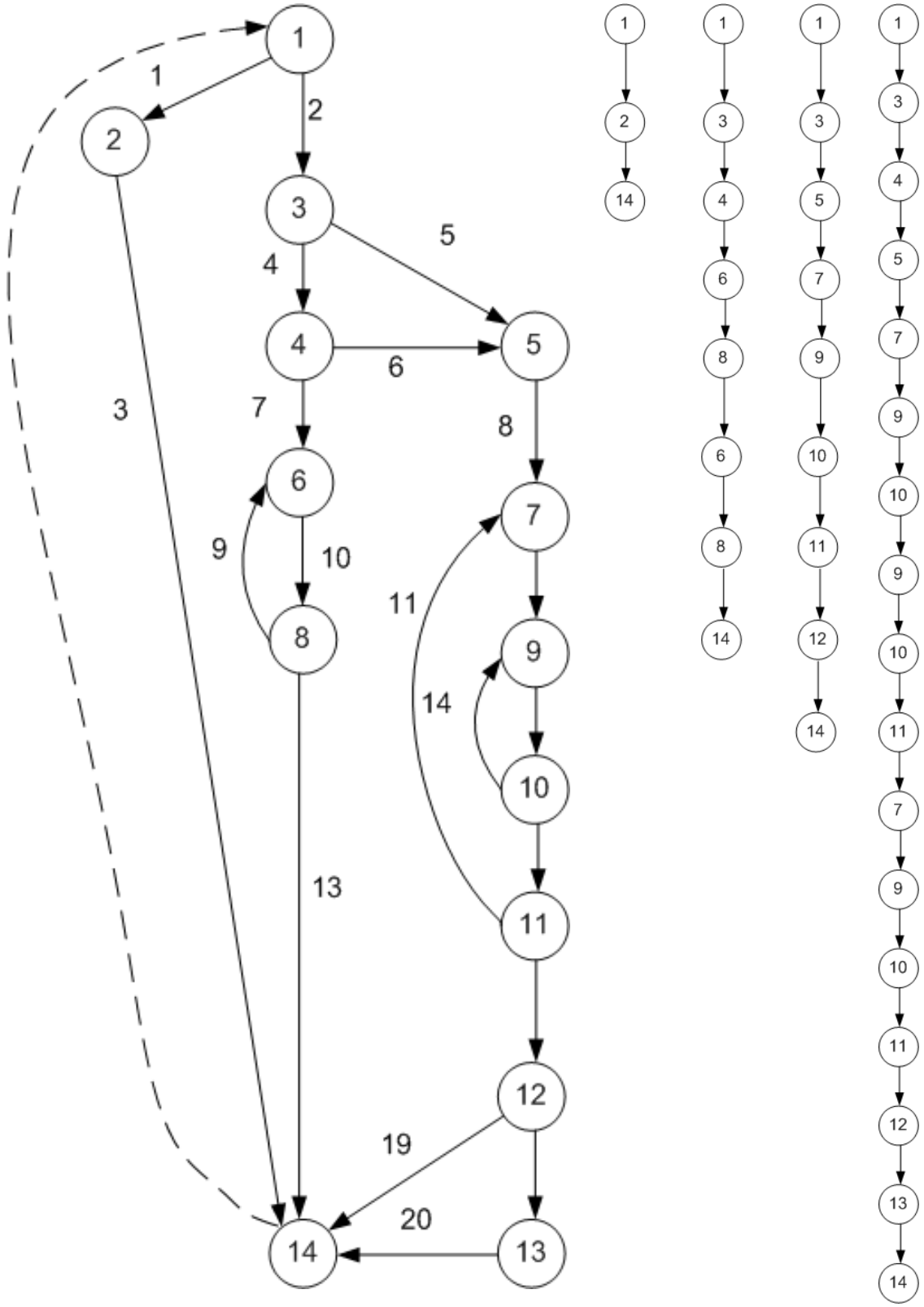


Рис. 9. Пример исполнения маршрутов по первому критерию

Сложность программы при выделении маршрутов по этому критерию характеризуется числом маршрутов (***M*** = 4) и сложностью тестов

$$\xi = \sum_{i=1}^{4} \xi_i = 20,$$

где ξ_i – количество вершин ветвления в *i*-м маршруте без учета последней.

Величина ξ характеризует суммарное количество условий, которое необходимо задать в тестах для полной проверки всех маршрутов, выделенных по первому критерию.

2.3.1. Метрика Мак-Кейба

Второй критерий выбора маршрутов для оценки сложности структуры (**метрика Мак-Кейба** [34]) заключается в анализе базовых маршрутов в программе, формируемых и оцениваемых на основе определения **цикломатического числа** исходного графа проверяемой программы, предложенного Мак-Кейбом в 1976 г.

Этот критерий наиболее широко применяется для оценки сложности программных модулей и наиболее полно исследован при анализе корреляции сложности и трудоемкости создания программ.

Для определения цикломатического числа Z исходного графа программы используется полное количество его вершин, количество связывающих дуг и число компонент связности:

$$Z(G) = e - v + 2p,$$

где e – число дуг ориентированного графа G, v – число вершин, p – число компонент связности графа.

Число компонент связности графа можно рассматривать как количество дуг, которые необходимо добавить для преобразования графа в сильно связный.

Сильно связным называется граф, любые две вершины которого взаимно достижимы. Для графов корректных программ, т. е. графов, не имеющих недостижимых от точки входа участков и «висячих» точек входа и выхода, сильно связный граф, как правило, получается путем замыкания дугой вершины, обозначающей конец программы, на вершину, обозначающую точку входа в эту программу (см. пунктир на рис. 9).

По сути $Z(G)$ определяет число линейно независимых контуров в сильно связном графе. Цикломатическое число Мак-Кейба показывает требуемое количество проходов для покрытия всех контуров сильно связного графа или количества тестовых прогонов программы, необходимых для исчерпывающего тестирования по принципу «работает каждая ветвь».

В корректно написанных программах $p = 1$, и поэтому формула для расчета цикломатической сложности приобретает вид:

$$Z(G) = e - v + 2.$$

Таким образом, данный критерий требует проверки каждого линейно независимого цикла в максимально связном графе программы. В исходном

графе программы это соответствует однократной проверке каждого линейно независимого ациклического маршрута и каждого линейно независимого цикла, в совокупности образующих базовые маршруты.

Каждый линейно независимый маршрут или цикл отличается от всех остальных хотя бы одной вершиной или дугой, т.е. его структура не может быть полностью образована компонентами других маршрутов.

Для программы, граф которой изображен на рис.10, цикломатическое число при $e = 10$, $v = 8$, $p = 1$ определится как

$$Z(G) = 10 - 8 + 2 = 4.$$

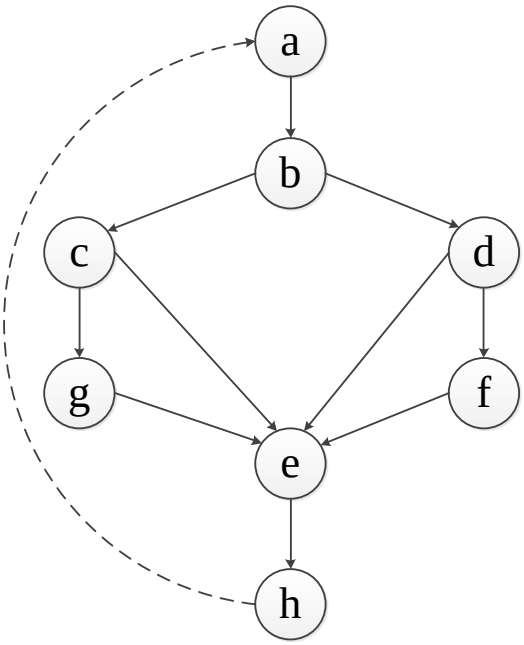


Рис.10. Пример управляющего графа программы

Для рассматриваемого примера сильносвязный граф имеет 4 линейно независимых маршрута:

1) a–b–c–g–e–h–a;

2) a–b–c–e–h–a;

3) a–b–d–f–e–h–a;

4) a–b–d–e–h–a.

Пример 3. Для графа программы, представленного на рис. 11, цикломатическое число равно: $Z = 21 - 14 + 2 = 9$.

Множество проверяемых по этому критерию структур образуется из трех линейно-независимых циклов и пяти линейно-независимых ациклических структур. Последние соответствуют выделяемым ациклическим маршрутам, различающимся хотя бы одной дугой. Эти ациклические структуры исходного графа образуют циклы в максимально сильно связном графе, если

конечный оператор искусственно соединить дугой с входным (пунктир на рис. 11).

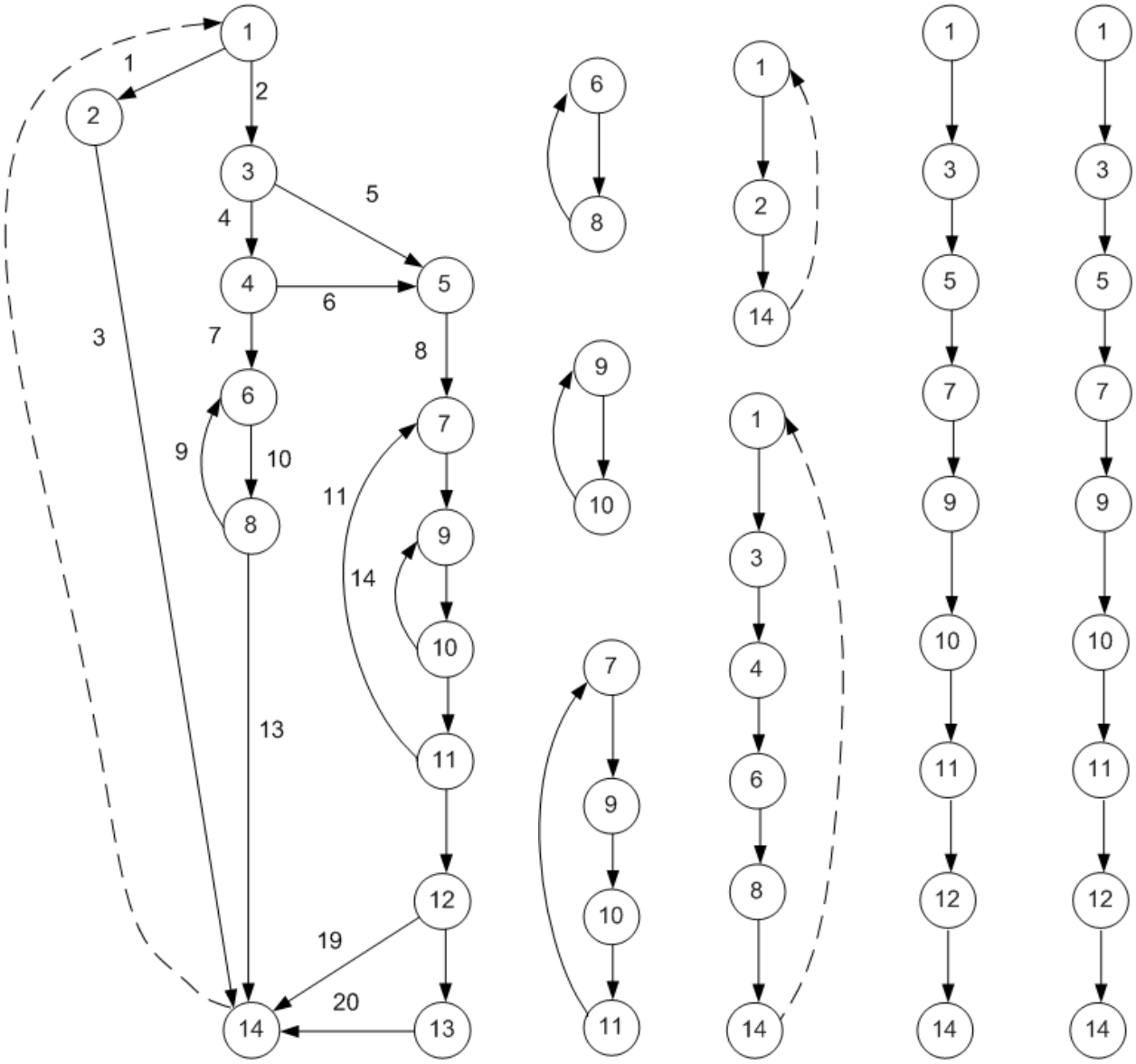


Рис. 11. Пример исполнения маршрутов по метрике Мак-Кейба

Подсчет цикломатической сложности для структурированных программ может производиться еще двумя способами. При этом структурированными считаются программы, которые:

- не имеют циклов с несколькими выходами;
- не имеют переходов, внутрь циклов или условных операторов;
- не имеют выходов из внутренней части циклов или условных операторов.

Для расчета цикломатической сложности и автоматического выделения операторов, формирующий список их связей, используют:

1) **матрицу смежности** (рис. 12), в которой единица (знак «+») располагается в позиции (*i*; *j*), если из вершины *i* можно перейти к вершине *j* за один шаг (возможно определение количества связывающих дуг);

2) **матрицу достижимости** (рис. 13), в которой на позиции (*i*; *j*) помещается 1 (знак «+»), если из вершины *i* можно перейти к вершине *j* за любое число шагов.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1														
2	+													
3	+													
4			+											
5			+	+										
6				+				+						
7					+						+			
8						+								
9							+			+				
10									+					
11										+				
12											+			
13												+		
14		+						+				+	+	

Рис. 12. Матрица смежности для графа программы на рис. 11

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1														
2	+													
3	+													
4	+		+											
5	+		+	+										
6	+		+	+		+		+						
7	+		+	+	+		+		+	+	+			
8	+		+	+		+		+						
9	+		+	+	+		+		+	+	+			
10	+		+	+	+		+		+	+	+			
11	+		+	+	+		+		+	+	+			
12	+		+	+	+		+		+	+	+			
13	+		+	+	+		+		+	+	+	+		
14	+	+	+	+	+	+	+	+	+	+	+	+	+	

Рис. 13. Матрица достижимости для графа программы на рис. 11

Матрица достижимости позволяет сравнительно просто выделить циклы и некоторые другие структурные некорректности (лишние тупиковые участки графа программы).

Цикломатическая сложность подсчитывается по матрице достижимости графа программы, у которого предварительно конечная вершина замкнута с начальной (см. пунктир на рис. 11). По этим же матрицам в списковой форме можно автоматически выделить цикломатические маршруты исполнения программы, рассчитать сложность тестов и представить выделенные маршруты в распечатках для контроля и планирования отладки.

Цикломатическая сложность достаточно просто определяется для обычных программных модулей, имеющих, как правило, около 10–20 и не более 50 вершин. Чаще всего цикломатическая сложность не превышает $Z = 10$ и, по-видимому, эта величина является рациональным пределом, обеспечивающим достаточно полную тестируемость программных модулей.

Встречаются программы с $Z \approx 30–60$ и установлено, что такие модули имеют высокую структурную сложность и, соответственно, повышенную вероятность ошибок.

Более сильные критерии проверки и определения сложности структуры программы включают требования однократной проверки не только линейно независимых, но и всех линейно зависимых циклов и ациклических маршрутов.

Стремление к увеличению глубины проверок приводит к третьему критерию – **критерию выделения полного состава базовых структур графа программы**. Он заключается в анализе (хотя бы один раз) каждого из реальных ациклических маршрутов исходного графа программы и каждого цикла, достижимого из всех этих маршрутов. В результате исходный граф программы представляется полным множеством базовых структур, каждая из которых образуется очередным ациклическим маршрутом и одним или несколькими циклическими маршрутами, так что в совокупности должны быть охвачены все циклы, достижимые из данного ациклического маршрута. Если из некоторого ациклического маршрута исходного графа достижимы несколько элементарных циклов, то при тестировании должны исполняться все достижимые циклы.

Приведенные критерии для оценки структурной сложности программных модулей характеризуют в каждом случае минимально необходимые величины проверок по каждому критерию. При этом может численно оцениваться полнота проверок с учетом количества непроверенных маршрутов или условий по выбранному критерию. Оценить достаточность проверок программы значительно труднее, так как при этом, кроме сложности структуры, необходимо анализировать сложность преобразования каждой переменной.

Наибольшие трудности при оценке структурной сложности возникают при анализе программных модулей, содержащих два и более цикла, когда циклы являются вложенными либо имеют зацепление некоторой своей частью. В этом случае число маршрутов возрастает значительно быстрее, чем при одиночных не связанных циклах, и трудно поддается оценке в типовых случаях.

Однако практика разработки программ привела многих специалистов к выводу о целесообразности избегать вложенных и зацепленных циклов для упрощения программы и выносить из тела цикла максимальное число операторов программы.

Показатель цикломатической сложности позволяет не только произвести оценку трудоемкости реализации отдельных элементов программного проекта и скорректировать общие показатели оценки длительности и стоимости проекта, но и оценить связанные риски и принять необходимые управленческие решения.

Как правило, при вычислении цикломатической сложности логические операторы не учитываются.

Показатель цикломатической сложности может быть рассчитан для модуля и других структурных единиц программы.

Метрика Мак-Кейба не способна различать циклические и условные конструкции. Еще одним существенным недостатком подобного подхода является то, что программы, представленные одними и теми же графами, могут иметь совершенно разные по сложности предикаты (предикат – логическое выражение, содержащее хотя бы одну переменную).

2.3.2. Метрика Майерса

Для исправления отмеченного недостатка **Г. Майерсом** была разработана новая методика. В качестве оценки он предложил взять интервал (эта оценка еще называется интервальной) $[Z(G); Z(G)+h]$, где h для простых предикатов (например, для оператора **DO**) равно нулю, а для n-местных (например, для оператора **CASE** с n исходами) $h = n\text{-}1$. Данный метод позволяет различать разные по сложности предикаты, однако на практике он почти не применяется.

2.3.3. Метрика Хансена

Еще одна модификация метрики Мак-Кейба – метрика Хансена. Мера сложности программы в данном случае представляется в виде пары (A, B), где A – метрика Мак-Кейба, B – число исполняемых операторов.

Преимуществом данной меры является ее чувствительность к структурированности программного обеспечения.

2.3.4. Метрика Чена

Топологическая мера Чена выражает сложность программы через число пересечений границ между областями, образуемыми графом программы. Этот подход применим только к структурированным программам, допускающим лишь последовательное соединение управляющих конструкций. Для неструктурированных программ мера Чена существенно зависит от условных и безусловных переходов. В этом случае можно указать верхнюю и нижнюю границы меры. Верхняя – есть m+1, где m – число логических операторов при их взаимной вложенности. Нижняя – равна 2. Когда управляющий граф программы имеет только одну компоненту связности, мера Чена совпадает с цикломатической мерой Мак-Кейба.

2.3.5. Метрики Харрисона, Мейджела

Продолжая тему анализа управляющего графа программы, можно выделить еще одну подгруппу метрик – метрики Харрисона, Мейджела.

Данные меры учитывают уровень вложенности и протяженность программы.

Каждой вершине графа присваивается своя сложность в соответствии с оператором, который она изображает. Эта начальная сложность вершины может вычисляться любым способом, включая использование мер Холстеда. Выделим для каждой предикатной вершины подграф, порожденный вершинами, которые являются концами исходящих из нее дуг, а также вершинами, достижимыми из каждой такой вершины (нижняя граница подграфа), и вершинами, лежащими на путях из предикатной вершины в какую-нибудь нижнюю границу. Этот подграф называется сферой влияния предикатной вершины.

Приведенной сложностью предикатной вершины называется сумма начальных или приведенных сложностей вершин, входящих в ее сферу влияния, плюс первичная сложность самой предикатной вершины.

Функциональная мера (**SCOPE**) программы – это сумма приведенных сложностей всех вершин управляющего графа.

Функциональным отношением (**SCORT**) называется отношение числа вершин в управляющем графе к его функциональной сложности, причем из числа вершин исключаются терминальные.

SCORT может принимать разные значения для графов с одинаковым цикломатическим числом.

2.3.6. Метрика Пивоварского

Это очередная модификация меры цикломатической сложности. Она позволяет отслеживать различия не только между последовательными и вложенными управляющими конструкциями, но и между структурированными и неструктурированными программами. Она выражается отношением

$$N(G) = Z^*(G) + \sum P_i,$$

где $Z^*(G)$ – модифицированная цикломатическая сложность, вычисленная так же, как и $Z(G)$, но с одним отличием: оператор **CASE** с n выходами рассматривается как один логический оператор, а не как n - 1 операторов; P_i – глубина вложенности i-й предикатной вершины.

Для подсчета глубины вложенности предикатных вершин используется число «сфер влияния». Под глубиной вложенности понимается число всех «сфер влияния» предикатов, которые либо полностью содержатся в сфере рассматриваемой вершины, либо пересекаются с ней. Глубина вложенности увеличивается за счет вложенности не самих предикатов, а «сфер влияния». Мера Пивоварского возрастает при переходе от последовательных программ к вложенным и далее к неструктурированным, что является ее огромным преимуществом перед многими другими мерами данной группы.

2.3.7. Мера Вудворда

Данная мера – это количество пересечений дуг управляющего графа. Так как в хорошо структурированной программе таких ситуаций возникать не должно, то данная метрика применяется в основном в слабо структурированных языках (Ассемблер, Фортран). Точка пересечения возникает при выходе управления за пределы двух вершин, являющихся последовательными операторами.

2.3.8. Метрика граничных значений

Этот метод так же основан на анализе управляющего графа программы. При его использовании подсчитывается число входов и выходов из узловых точек графа программы.

Для описания метрики граничных значений необходимо ввести несколько дополнительных понятий.

Пусть G – управляющий граф программы с единственной начальной и единственной конечной вершинами.

В этом графе число входящих в вершину дуг называется отрицательной степенью вершины, а число исходящих из вершины дуг – положительной степенью вершины. Тогда набор вершин графа можно разбить на две группы: вершины, у которых положительная степень <=1; вершины, у которых положительная степень >=2.

Вершины первой группы называют принимающими вершинами, а вершины второй группы – вершинами отбора.

Для получения оценки по методу граничных значений необходимо разбить граф G на максимальное число подграфов G^*, удовлетворяющих следующим условиям:

- вход в подграф осуществляется только через вершину отбора;

- каждый подграф включает вершину (называемую в дальнейшем нижней границей подграфа), в которую можно попасть из любой другой вершины подграфа.

Например, вершина отбора, соединенная сама с собой дугой-петлей, образует подграф (рис.14, табл.2).

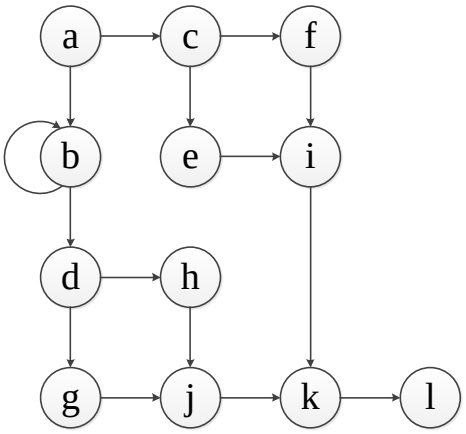


Рис. 14. Граф программы для анализа ее сложности методом граничных значений

Таблица 2– Подграфы программы

Характеристики подграфов программы	Вершины отбора			
	a	b	c	d
Вершины перехода	b, c	b, d	e, f	g, h
Скорректированная сложность вершины графа	10	2	3	3
Вершины подграфа	b, c, d, e, f, g, h, i, j	b	e, f	g, h
Нижняя граница подграфа	k	d	i	j

Число вершин, образующих такой подграф, равно скорректированной сложности вершины отбора (табл.3).

Каждая принимающая вершина имеет скорректированную сложность, равную 1, кроме конечной вершины, скорректированная сложность которой равна 0. Приведенные сложности всех вершин графа G суммируются, образуя абсолютную граничную сложность программы. После этого определяется относительная граничная сложность программы:

$$S_o=1-(v-1)/S_a,$$

где S_o – относительная граничная сложность программы, S_a – абсолютная граничная сложность программы, v – общее число вершин графа программы. Таким образом, относительная граничная сложность программы равна

$$S_o = 1-11/25 = 0,56.$$

Таблица 3– Скорректированная сложность вершин графа программы

Вершина графа программы	Скорректированная сложность вершины графа	Вершина графа программы	Скорректированная сложность вершины графа
a	10	g	1
b	2	h	1
c	3	i	1
d	3	j	1
e	1	k	1
f	1	l	0

2.4. Метрики сложности потока данных программ

Эти метрики базируются на оценке использования, конфигурации и размещения данных в программе. В первую очередь это касается глобальных переменных.

2.4.1. Метрика Чепина

Суть метода состоит в оценке информационной прочности отдельно взятого программного модуля с помощью анализа характера использования переменных из списка ввода-вывода.

Все множество переменных, составляющих список ввода-вывода, разбивается на 4 функциональные группы:

1) P – вводимые переменные для расчетов и для обеспечения вывода;

2) M – модифицируемые, или создаваемые внутри программы переменные;

3) C – переменные, участвующие в управлении работой программного модуля (управляющие переменные);

4) T – не используемые в программе («паразитные») переменные.

Поскольку каждая переменная может выполнять одновременно несколько функций, необходимо учитывать ее в каждой соответствующей функциональной группе.

Метрика Чепина имеет следующий вид:

$$Q = a_1P + a_2M + a_3C + a_4T,$$

где a_1, a_2, a_3, a_4 — весовые коэффициенты.

Весовые коэффициенты использованы для отражения различного влияния на сложность программы каждой функциональной группы. По мнению автора метрики, наибольший вес, равный 3, имеет функциональная группа C, так как она влияет на поток управления программы (a_3=3). Весовые коэффициенты остальных групп распределяются следующим образом: a_1=1; a_2=2; a_4=0,5. Весовой коэффициент группы T не равен 0, поскольку «паразитные» переменные не увеличивают сложность потока данных программы, но иногда затрудняют ее понимание. С учетом весовых коэффициентов

$$Q = P + 2M + 3C + 0{,}5T.$$

2.4.2. Метрика спена

Метрика, использующая **спен**, основывается на локализации обращений к данным внутри каждой программной секции. Спен – это число утверждений, содержащих данный идентификатор, между его первым и последним появлением в тексте программы. Следовательно, идентификатор, появившийся n раз, имеет спен, равный n-1.

Предположим, что в программе обнаружен идентификатор, спен которого равен 100. Тогда при построении трассы программы по этому идентификатору придется ввести в тело программы, по крайней мере, 100 контролирующих утверждений, что осложняет тестирование и отладку.

2.4.3. Метрика обращения к глобальным переменным.

Еще одна метрика, учитывающая сложность потока данных – это метрика, связывающая сложность программ с обращениями к глобальным переменным.

Пара «модуль-глобальная переменная» обозначается как (p, r), где p – модуль, имеющий доступ к глобальной переменной r. В зависимости от наличия в программе реального обращения к переменной r формируются два типа пар «модуль-глобальная переменная»: фактические и возможные.

Возможное обращение к *r* с помощью *p* показывает, что область существования *r* включает в себя *p*.

Данная характеристика обозначается A_{Up} и говорит о том, сколько раз модули U_p действительно получали доступ к глобальным переменным, а число P_{Up} – сколько раз они могли бы получить доступ.

Отношение числа фактических обращений к возможным определяется по формуле

$$R_{Up} = A_{Up} / P_{Up}.$$

Эта формула показывает приближенную вероятность ссылки произвольного модуля на произвольную глобальную переменную. Очевидно, что чем выше эта вероятность, тем выше вероятность «несанкционированного» изменения какой-либо переменной, что может существенно осложнить работы, связанные с модификацией программы.

2.4.4. Метрика Кафура

На основе концепции информационных потоков создана **метрика Кафура**. Для использования данной метрики вводятся понятия локального и глобального потока: локальный поток информации из **A** в **B** существует, если:

1. Модуль **A** вызывает модуль **B** (прямой локальный поток);

2. Модуль **B** вызывает модуль **A** и **A** возвращает **B** значение, которое используется в **B** (непрямой локальный поток);

3. Модуль **C** вызывает модули **A**, **B** и передаёт результат выполнения модуля **A** в **B**.

Далее следует дать понятие глобального потока информации: глобальный поток информации из **A** в **B** через глобальную структуру данных **D** существует, если модуль **A** помещает информацию в **D**, а модуль **B** использует информацию из **D**.

На основе этих понятий вводится величина **I** – информационная сложность процедуры:

$$\mathbf{I} = \text{length} * (\text{fan\_in} * \text{fan\_out})^2,$$

где length – сложность текста процедуры (измеряется через какую-нибудь из метрик объёма, типа метрик Холстеда, Маккейба, **LOC** и т.п.);

fan_in – число локальных потоков, входящих внутрь процедуры, плюс число структур данных, из которых процедура берёт информацию;

fan_out – число локальных потоков, исходящих из процедуры, плюс число структур данных, которые обновляются процедурой.

Информационную сложность модуля можно определить как сумму информационных сложностей входящих в него процедур.

На следующем шаге рассматривается информационная сложность модуля относительно некоторой структуры данных. Информационная мера сложности модуля относительно структуры данных оценивается по формуле

$$\mathbf{J} = \mathbf{W} * \mathbf{R} + \mathbf{W} * \mathbf{RW} + \mathbf{RW} * \mathbf{R} + \mathbf{RW} * (\mathbf{RW} - 1),$$

где **W** – число процедур, которые только обновляют структуру данных;

R – только читают информацию из структуры данных;

RW – и читают, и обновляют информацию в структуре данных.

2.4.5. Метрика Овиедо

Поток данных считается более сложным, если он более сложный в составляющих частях программы, где при определённом разбиении программы он считается достаточно просто.

Для использования данной метрики программа разбивается на линейные непересекающиеся участки – лучи операторов, которые образуют управляющий граф программы.

Автор метрики исходит из следующих предположений:

- программист может найти отношение между определяющими и использующими вхождениями переменной внутри луча более легко, чем между лучами;

- число различных определяющих вхождений в каждом луче более важно, чем общее число использующих вхождений переменных в каждом луче.

Обозначим через $R(i)$ множество определяющих вхождений переменных, которые расположены в радиусе действия луча i (определяющее вхождение переменной находится в радиусе действия луча, если переменная либо локальна в нём и имеет определяющее вхождение, либо для неё есть определяющее вхождение в некотором предшествующем луче, и нет локального определения по пути). Обозначим через $V(i)$ множество переменных, использующие вхождения, которые уже есть в луче i. Тогда мера сложности i-го луча $DF(i)$ задаётся как

$$DF(i) = \sum_{j=i}^{\|V(i)\|} DEF(v_j),$$

где $DEF(v_j)$ – число определяющих вхождений переменной v_j из множества $R(i)$, а $\|V(i)\|$ – мощность множества $V(i)$.

2.5. Сложность комплексов программ

2.5.1. Структурная сложность комплексов программ

Для анализа структурной сложности важное значение имеет степень взаимосвязи модулей. Наименьшая сложность комплекса достигается при наименьшей взаимосвязи модулей. Следовательно, необходимо уменьшить межмодульные связи по управлению и информации. Для этого введены понятия прямые предки и прямые наследники рассматриваемого *i*-го модуля.

Правила взаимодействия в хорошо структурированных комплексах программ:

- исполнение программы начинается с корневого модуля – диспетчера, который является единственным;
- управление может передаваться только модулю, для которого вызывающий модуль является прямым предком;
- только прямой предок может обращаться к прямому наследнику;
- когда модуль завершает свое исполнение, управление предается обратно вызывающему модулю;
- каждый модуль имеет только одну точку входа и одну точку выхода из модуля без возврата.

Оценка производится при различном учете глубины информационных и управляющих связей между модулями. В простейшем случае комплексы программ оцениваются:

- по числу входящих в них модулей;
- по суммарному количеству команд и операторов;
- по числу иерархических уровней и т.д.,

но определяющая роль при этом принадлежит межмодульным связям.

Все это привело:

- к формализации правил структурного построения;
- к формализации организации межмодульного интерфейса;
- к введению ограничений на размеры модулей (30-100 операторов на языках высокого уровня).

Правила взаимодействия модулей предполагают использование критериев оценки структурной сложности групп и комплексов программ.

2.5.1.1. Оценка по степени отличия структуры от древовидной.

Пусть N_i – количество модулей для уровней от 0-го до 1-го;

$G_i = N_i - 1$ – число связей, образующих древовидную структуру;

A_i – реальное количество связей;

$C_i = A_i - G_i$ – отличие структуры от древовидной;

$R_i = \dfrac{C_i}{A_i}$ – относительная сложность связей *i*-го уровня.

Тогда глубину связей модуля *i*-го уровня с предыдущим уровнем можно оценить по формуле

$$D_i = 1 - \frac{G_i}{A_i}.$$

Вывод: оценка значения C_i (отличия структуры от древовидной) используется для снижения или выравнивания сложности межмодульных связей и приближения структуры групп программ к простейшим. Модули, наиболее сильно влияющие на C_i, т.е. имеющие высокую нагрузку по взаимодействию, желательно перерабатывать для упрощения связей или наиболее тщательно тестировать при комплексной отладке.

2.5.1.2. Оценка с учетом полного числа связей.

В данном случае производится учет связей по информации и управлению (рис. 15).

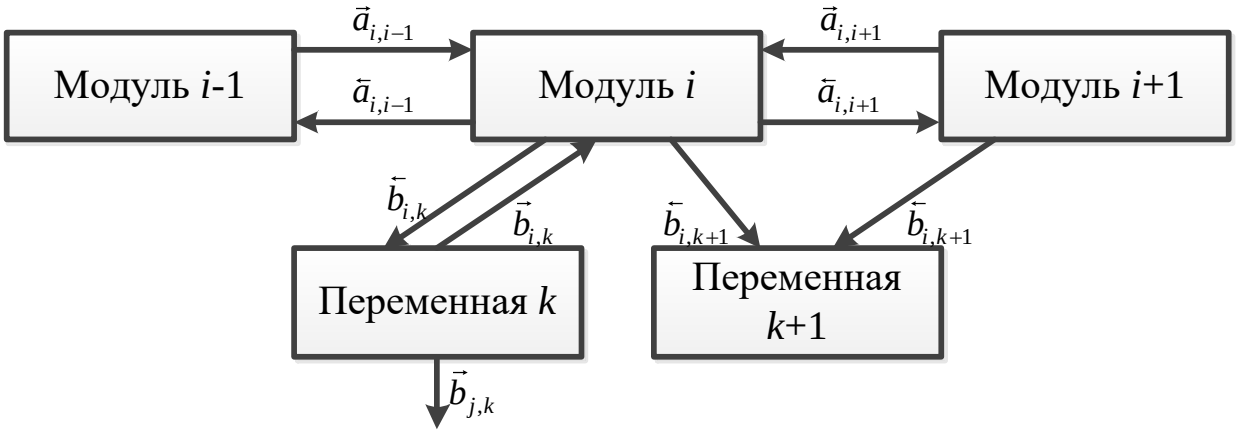


Рис. 15. Взаимодействие модулей и переменных

Управляющие связи:

$\vec{a}_{i,j}$ – управляющая связь *i*-го модуля, которая вызывает его для исполнения;

$\overleftarrow{a}_{i,j}$ – управляющая связь *i*-го модуля, посредством которой *i*-й модуль вызывает другие модули.

Сложность управляющих связей *i*-го модуля определяется по формуле

$$\Lambda_i = \sum_j \left(\vec{a}_{i,j} + \overleftarrow{a}_{i,j} \right).$$

Сложность управляющих связей для всех модулей определяется по формуле

$$\Lambda = \sum_i \Lambda_i = \sum_i \sum_j \left(\vec{a}_{i,j} + \bar{a}_{i,j} \right),$$

которая определяет число возможных маршрутов исполнения программы на уровне модулей, а также число связей, подлежащих проверке в процессе тестирования и комплексной отладки.

Рассмотрим информационные связи, каждая из которых может характеризоваться информационной прочностью.

Сложность информационных связей *i*-го модуля определяется по формуле

$$\Phi_i = \sum_k \left(\vec{b}_{i,k} + \vec{b}_{i,k} \right) = \vec{\Phi}_i + \vec{\Phi}_i,$$

где $\vec{b}_{i,k}$ – количество имен переменных на входе *i*-го модуля, необходимых для его нормального функционирования;

$\vec{b}_{i,k}$ – количество имен переменных, подготавливаемых *i*-м модулем;

$\vec{\Phi}_i = \sum_k \vec{b}_{i,k}$ – информационная зависимость *i*-го модуля на входе от количества входных переменных $\vec{b}_{i,k}$;

$\vec{\Phi}_i = \sum_k \vec{b}_{i,k}$ – информационная зависимость *i*-го модуля от остальных модулей по числу $\vec{b}_{i,k}$ переменных на выходе.

Сложность информационных связей для группы программ определяется по формуле

$$\Phi = \sum_i \Phi_i = \sum_i \sum_k \left(\vec{b}_{i,k} + \vec{b}_{i,k} \right).$$

Вывод: для уменьшения сложности информационных связей следует сокращать количество глобальных переменных и степень их использования модулями, а также шире использовать обменные переменные.

Исследование характеристик связей между модулями реальных комплексов программ позволили получить ориентир по глубине связей и выявить тенденции их изменения:

1 вид. Структура близка к бинарному дереву; количество иерархических уровней – 10-15; количество модулей с каждым уровнем иерархии увеличивается по мере снижения, но медленнее, чем в регулярном бинарном дереве.

2 вид. Количество иерархических уровней – 3-6; один или несколько модулей вызывают последовательно большое количество модулей; основное число модулей сосредоточено на нижнем уровне иерархии.

Программные модули можно разделить на 3 основных типа:

1. Диспетчерские, управляющие программные модули (высокий иерархический уровень), для которых

$$\sum_j \vec{a}_{i,j} \sim 10;$$

$$\sum_j \vec{a}_{i,j} \sim 10 \div 100.$$

Такие модули характеризуются значительными связями по управлению и слабыми информационными связями.

2. Функциональные программные модули, для которых

$$\sum_j \vec{a}_{i,j} \sim 3{,}1;$$

$$\sum_j \vec{a}_{i,j} \sim 3{,}8;$$

$$\sum_k \left(\vec{b}_{i,k} + \vec{b}_{i,k} \right) \sim 5 \div 7\%$$ от числа автокодных операторов.

3. Стандартные программные модули, для которых

$$\sum_j \vec{a}_{i,j} \sim 0{,}8$$ (редко вызывают);

$$\sum_j \vec{a}_{i,j} \sim 20 \div 30.$$

Такие модули характеризуются минимальными информационными связями.

2.5.1.3. Оценка с учетом прочности каждой связи

Сложность межмодульной связи можно охарактеризовать вероятностью ошибки при ее формализации и степенью влияния этой ошибки на последующее функционирование модулей. Следовательно, необходимо установить шкалу прочности различных типов связей.

Прочность связи двух модулей можно охарактеризовать вероятностью $P_{i,j}$ того, что модуль j придется изменить при любом изменении модуля i.

Стремятся ослабить эти связи, т.е. получить $P_{i,j} \sim 0{,}01 \div 0{,}1$ и учитывают связи без их прочности.

В частности, предлагается оценивать сложность i-го модуля как взвешенную сложность связей в зависимости от числа прямых предков f_i и прямых наследников g_i данного модуля, т.е.

$$\Lambda_i = f_i X_i + g_i Y_i,$$

где X_i – сложность управляющих структур и переменных, используемых при формировании вызова данного *i*–го модуля;

Y_i – сложность управляющих структур и переменных, используемых при формировании вызова прямых наследников *i*–го модуля.

2.5.2. Статистические характеристики комплексов программ

В результате исследований сформулированы следующие статистические характеристики комплексов программ:

- число модулей – основная единица размера и сложности;
- мера сложности создания версии – отношение числа модулей, подвергшихся изменениям и дополненных к комплексу к общему числу модулей;
- интенсивность работ по созданию версии – число измененных модулей в единицу времени.

При этом учитывают следующие ограничения:

- размеры модулей ограничены;
- правила оформления стандартизованы;
- отсутствуют ограничения на используемые ресурсы памяти и производительности.

2.6. Вычислительная сложность функционирования комплексов программ

Вычислительная сложность измеряется величиной ресурсов времени и памяти, необходимых для решения поставленных задач.

Методы исследования сложности вычислений условно разделяют на:

- интуитивно-прагматические (эвристические);
- формально-логические (аксиоматические).

В первом случае учитываются практические нужды оценки ресурсов ЭВМ, необходимых для решения задач различных классов. Сложность ассоциируется с одним из ресурсов ЭВМ, в максимальной степени ограничивающих предельную размерность при решении конкретных задач.

Во втором случае учитываются общие закономерности изменения сложности вычислений от объема данных (экстремальные условия, предельные характеристики ускорения решения задач).

2.6.1. Временная сложность

Временная сложность программ в основном определяется алгоритмами, используемыми для решения задач. Несмотря на возрастание быстродействия современных ЭВМ, имеется много задач, которые невозможно решать некоторыми точными алгоритмами при необходимом объеме входных данных. При ограниченной области изменения входных данных имеется возможность эффективного ускорения вычислений. Теоретически доказана принципиальная возможность размена длительности решения любых задач на объем памяти для хранения программ и данных. При реальной производительности ЭВМ достижимое качество программ может существенно определяться их временной сложностью. Это означает, что длительность исполнения программ по тестовым данным и длительность расчета эталонных значений возрастают так быстро, что реальные ресурсы современных ЭВМ ограничивают допустимую полноту отладки и объем получаемых результатов.

Оценка временной сложности независимо от быстродействия ЭВМ характеризуется количеством операций или условных «шагов» работы ЭВМ, требуемых для обработки входных данных размера $\overline{n}$. Алгоритм можно охарактеризовать функцией $f(\overline{n})$, определяющей скорость роста объема вычислений при возрастании $\overline{n}$. Вид этой функции является показателем временной сложности алгоритма.

У «хороших» алгоритмов скорость роста времени вычислений имеет порядок $\overline{n}^c$, где c = const > 1 (полиномиальные алгоритмы).

У «плохих» алгоритмов указанная скорость имеет порядок c^n или даже быстрее.

Временная сложность значительно сильнее влияет на размер ряда задач, чем изменение быстродействия ЭВМ. Следовательно, асимптотическая сложность алгоритмов по времени счета является важной мерой их качества, принципиально ограничивающей возможность решения некоторых задач при большом объеме входных данных. Поэтому особенно важно изучать предельные свойства наиболее важных абстрактных задач и общие методы упрощения задач конкретных достаточно массовых типов. Для практики имеет большое значение оценка возможности получения результата за любое ограниченное время.

2.6.2. Программная сложность

Программную сложность в простейшем случае можно оценить по числу символов в тексте программы, необходимому для ее полного описания на

некотором алгоритмическом языке, или по числу операторов (команд) при реализации программы на некоторой ЭВМ. Разнообразие алгоритмических языков и структур команд ЭВМ затрудняет обобщенные оценки и сравнения программной сложности различных программ. Введение соглашений структурного программирования и перечней стандартных операторов программирования позволяет в значительной степени унифицировать языковую базу, на которой проектируются программы. В связи с этим возможный разброс объема программ, построенных по одному алгоритму, при использовании различных языков программирования намного меньше, чем диапазон изменения объема программ при изменении алгоритмов решения задачи.

Относительная простота определения числа машинных команд в программе (или операторов в тексте на ассемблере) привела к широкому применению этого параметра в качестве инженерной меры программной сложности. Однако различия количества используемых байтов на одну операцию затрудняют подсчет сложности текстов программ по числу команд, а также сопоставление программной сложности при применении ЭВМ с различной структурой команд. Программная сложность в наибольшей степени определяет трудоемкость создания большинства крупных комплексов программ управления и обработки информации.

Программная сложность является в основном статической, так как в процессе исполнения размер программы не модифицируется и не зависит от объема входных данных. Динамический характер сложности проявляется при размещении программы во внешней памяти.

2.6.3. Информационная сложность

Информационная **сложность программ** в первую очередь зависит от количества типов и структуры данных, поступающих на вход и выдаваемых программой. Число различных видов операндов, используемых в программе, достаточно полно характеризует ее информационную сложность. Для приближенных инженерных оценок в качестве меры информационной сложности применяется емкость памяти, необходимая для оперативного накопления и хранения данных, используемых при решении задачи. Понятие информационной сложности включает емкость всех ячеек памяти и регистров, к которым осуществляется обращение при обработке операндов в процессе решения задачи. При этом важно установить объем каждого элемента памяти или стандартизировать их размеры. Сложность представления некоторого множества данных может быть отражена

совокупностью сложности табулирования опорных данных и сложности программ их декодирования для раскрытия всех значений.

Информационная сложность является динамической характеристикой и широко варьируется в зависимости от размерности входных данных. Информационная сложность зависит от языка кодирования операндов, от структуры и способа взаимодействия программных регистров.

Рассмотрим набор данных $X = \{x_1, x_2, ..., x_n\}$. Для наиболее экономичного описания любого элемента $x_i \in X$ необходимо такое количество двоичных разрядов, которое можно охарактеризовать энтропией $H(X) = \log_2 \overline{n}$. Если известны функциональные связи элементов множества, то можно сократить объем данных, необходимых для табулирования множества. Также при этом необходимо иметь механизм декодирования (трансляции).

Таким образом, информационная сложность складывается из сложности табулирования исходных данных и сложности декодирования.

Сложность текста – это длина самого короткого двоичного слова, содержащего всю информацию, необходимую для восстановления рассматриваемого текста при помощи некоторого способа декодирования.

Таким образом, использование трех видов вычислительной сложности приводит к необходимости введения векторных характеристик сложности и соответствующих методов их измерения. В простейшем случае – это взвешенный учет составляющих вектора, либо введение доминирования по одному из показателей в зависимости от их значения.

ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ К РАЗДЕЛУ 2

1. Охарактеризуйте понятие сложности программного обеспечения.
2. Опишите основные метрики размера программ.
3. Опишите состав метрик Холстеда.
4. Опишите метрику Джилба.
5. Опишите основные метрики сложности потока управления программ.
6. Опишите метрику Мак-Кейба.
7. Опишите порядок построения и использования матриц смежности и достижимости.
8. Опишите основные метрики сложности потока данных программ
9. Опишите метрику Чепина.
10. Охарактеризуйте структурную сложность комплексов программ.
11. Охарактеризуйте статистическую сложность комплексов программ.
12. Охарактеризуйте статистические характеристики комплексов программ.
13. Охарактеризуйте временную сложность функционирования комплексов программ.
14. Охарактеризуйте программную сложность функционирования комплексов программ.
15. Охарактеризуйте информационную сложность функционирования комплексов программ.

3. ОЦЕНИВАНИЕ КОРРЕКТНОСТИ ПРОГРАММ

3.1. Понятие корректности программ

Понятие «корректность» или «правильность» подразумевает способность программного продукта обеспечивать правильные или приемлемые результаты в соответствии с требованиями заказчика и пользователей [23].

Под корректностью программы понимают её соответствие некоторому эталону или совокупности формализованных эталонных правил и характеристик.

Эталонами для выбора требований к корректности при проектировании могут быть верифицированные и взаимоувязанные требования к функциям комплекса, компонентов и модулей программ, а также правила их структурного построения, организация взаимодействия и интерфейсов. Эти требования при разработке должны быть прослежены сверху вниз до модулей и использоваться как эталоны при установлении необходимой корректности соответствующих компонентов.

Требования к корректности могут представляться в виде описания двух основных свойств, которым должны соответствовать все программные компоненты и комплекс в целом. Первое требование состоит в выполнении определенной степени (%) прослеживаемости сверху вниз реализации требований технического задания и спецификации на программный продукт при последовательной детализации и верификации описаний программных компонентов вплоть до текстов и объектного кода программ. Второе требование заключается в выборе степени и стратегии покрытия тестами структуры и функций программных компонентов, совокупности маршрутов исполнения модулей и всего комплекса программ для процесса верификации и тестирования, достаточного для функционирования программного продукта с требуемым качеством и точностью результатов, при реальных ограничениях экономических ресурсов на тестирование.

Наиболее полным эталоном корректности программ является программная спецификация. Её особенностью является задание требований поведения программы для допустимых наборов входных данных. Поэтому корректная программа может неправильно работать или даже сбиваться на

недопустимых наборах входных данных. Свойством устойчивости к недопустимым наборам входных данных обладает надежная программа – в этом заключается разница между надёжной и корректной программами.

Понятие корректной (правильной) программы может рассматриваться статически вне ее исполнения во времени. Корректность программы не определена вне области изменения исходных данных, заданных требованиями спецификации. Степень некорректности программ определяется вероятностью попадания реальных исходных данных в пространство значений, которое задано требованиями спецификации и технического задания, однако не было проверено при тестировании и испытаниях. Значения этого показателя зависят от функциональной корректности применяемых компонентов и могут рассматриваться в зависимости от методов их достижения и оценивания: детерминированно, стохастически и в реальном времени.

Таким образом, корректность или правильность программы наиболее полно определяется степенью ее соответствия предъявляемым к ней формальным требованиям программной спецификации.

При отсутствии полностью формализованной спецификации требований в качестве эталона, которому должны соответствовать программа и результаты ее функционирования, иногда используются неформализованные представления разработчика, пользователя или заказчика программ.

Однако понятие корректности программ по отношению к запросам пользователя или заказчика сопряжено с неопределенностью самого эталона, которому должна соответствовать программа. Вследствие этого понятие корректности программ становится субъективным и его невозможно определить количественно.

Неопределенность программных спецификаций уменьшается в процессе разработки программ путем уточнения требований по согласованию между разработчиком и заказчиком.

Для сложных комплексов программ всегда существует риск обнаружить их некорректность по мнению пользователя или заказчика относительно имеющихся спецификаций вследствие неточности самих спецификаций.

Формальные правила проектирования программ устанавливаются стандартами и инструкциями подготовки текстов программ и их структурного построения. Эталоны этого вида включают описание языка программирования, правила оформления текстов программ и описания данных, они являются наиболее универсальными и в ряде случаев формализуются на уровне ГОСТов на языки программирования и базы данных.

Будем считать программное изделие правильным (корректным), если оно:

- решает действительно ту задачу, для которой оно было разработано;
- не «зависает» и не заканчивает свою работу аварийно;
- удовлетворяет всем требованиям из документа «Соглашение о требованиях» («Техническое задание») с учетом их уточнений в процессе проектирования программного изделия;
- разработано в соответствии с формальными правилами проектирования программного обеспечения.

Требования к корректности различаются в зависимости от двух видов критериев качества:

- для функциональных критериев они определяются предметной областью и функциями выполняемой программы;
- для конструктивных критериев они определяются общими для всех программ свойствами.

В зависимости от проверяемых компонентов программ различают четыре вида их корректности, представленные на рис. 16.

Рис.16. Классификация видов корректности программ

Корректность текстов программ имеет только конструктивную составляющую. Благодаря жестким правилам языков программирования синтаксическая и семантическая корректность программы на этапе ее трансляции и после трансляции является корректной с этой точки зрения.

Корректность текстов программ – степень соответствия исходных программ формализованным правилам языков спецификаций и программирования.

Синтаксическая корректность текстов программ – степень соответствия входного текста программ синтаксису языка программирования.

Семантическая корректность текстов программ – степень соответствия входного текста программ базовым конструкциям языка программирования.

Корректность программных модулей имеет и конструктивную и функциональную составляющую.

Конструктивная составляющая определяется правилами построения структуры модулей, задаваемых в технологии и языке программирования.

Структурная корректность модулей – соответствие их структуры общим правилам структурного программирования и конкретным правилам оформления и внутреннего построения программных модулей в данном заказе.

Структурная корректность программных модулей определяется правилами структурного, модульного построения программных комплексов и общими правилами организации межмодульных связей. Эта составляющая может быть проверена формализованными автоматизированными методами.

Формализованный структурный контроль программ основывается на статической проверке соответствия структуры программ и последовательности основных операций использования памяти в системе эталонных правил.

Функциональная составляющая корректности модулей зависит от предметной области и функциональных спецификаций программы.

Функциональная корректность модулей – корректность обработки исходных данных и получения результатов.

Функциональная составляющая может проверяться в различных условиях.

При проверке детерминированной составляющей для фиксированных наборов входных данных должны быть получены конкретные значения результатов.

При проверке стохастической составляющей входные данные задаются случайными величинами с известными законами распределения, и результаты также должны быть случайными величинами с требуемыми законами распределения и заданными корреляционными связями между входными и выходными данными.

Динамическая составляющая характерна для систем реального времени и определяется согласованием во времени порядка поступления входных данных и порядка выдачи результатов выполнения программы.

В общем случае функциональные спецификации программ определяют и функциональные требования к программам, и характеристики, с которыми они должны обеспечиваться, как это показано на рис. 17.

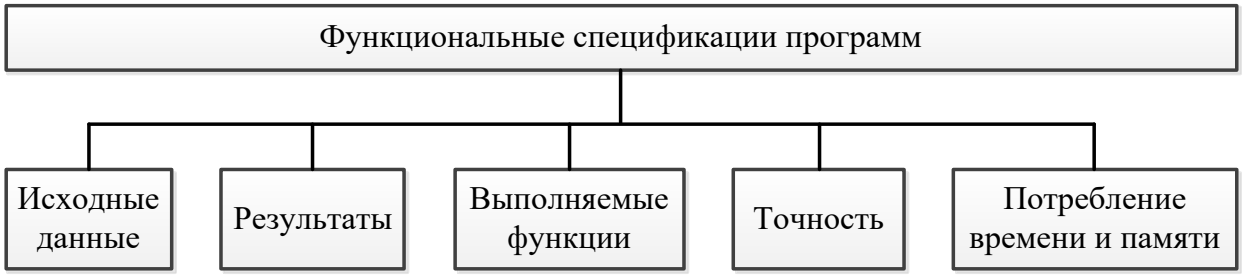


Рис.17. Функциональные спецификации программы

Корректность данных имеет конструктивную и функциональную составляющие.

Структурная корректность данных относится к конструктивной составляющей и предполагает правильность построения структурированных данных в программе (массивов, стеков, очередей и т.п.).

Корректность конкретных значений данных (функциональная составляющая) связана, в основном, с конкретизацией их содержания в процессе исполнения программ, и определяется диапазонами изменения их значений и соответствием типов полей структур типам значений данных.

Корректность комплексов программ также имеет конструктивную и функциональную составляющие.

Конструктивная составляющая определяется корректностью структуры межмодульных связей по управлению и данным, определяемым в интерфейсных требованиях к программе. Функциональная корректность комплексов программ определяется также, как и функциональная корректность модулей.

Функциональная корректность комплексов программ наиболее трудно формализуется вследствие большого количества возможных эталонных значений и распределений. В наиболее сложном случае для программ реального времени ее можно разделить на:

- детерминированную корректность – должно быть обеспечено однозначное соответствие исходных и результирующих данных исполняемых программ определенным эталонным значениям;

- стохастическую корректность – статистическое соответствие распределений результирующих случайных величин заданным эталонным распределениям при соответствующих распределениях исходных данных;

- динамическую корректность – соответствие изменяющихся во времени результатов исполнения программ эталонным данным.

3.2. Программные эталоны и методы проверки корректности программ

Эталоны для проверки корректности программ могут использоваться в следующих трех формах, поясняемых с помощью рис. 18:

- формализованные правила;
- программные спецификации;
- тесты.

1. **Формализованные правила** имеют достаточно неопределенностей, так как определяются двумя видами требований:

- требования стандартов (общероссийских и стандартов предприятий);
- требования языков и технологий программирования.

Для полной убедительности в корректности программ одних формализованных правил недостаточно.

2. **Программные спецификации** относятся к функциональным эталонам и, в основном, обеспечивают проверку корректности программ в статике. Они представляют собой совокупность взаимоувязанных непротиворечивых спецификаций на все компоненты комплекса программ

3. **Тесты** – это частные реализации взаимосвязанных исходных и результирующих данных.

Невозможно гарантировать полное покрытие тестами области определения исходных данных, заданных спецификациями. Для исключения неоднозначности определения эталонных тестовых значений необходимо сопровождать их получение детальной методикой и расшифровкой содержания.

Рис. 18. Средства проверки корректности программ

На рис.19. показана схема взаимодействия компонент, определяющих обнаруживаемые отклонения программ от эталонов.

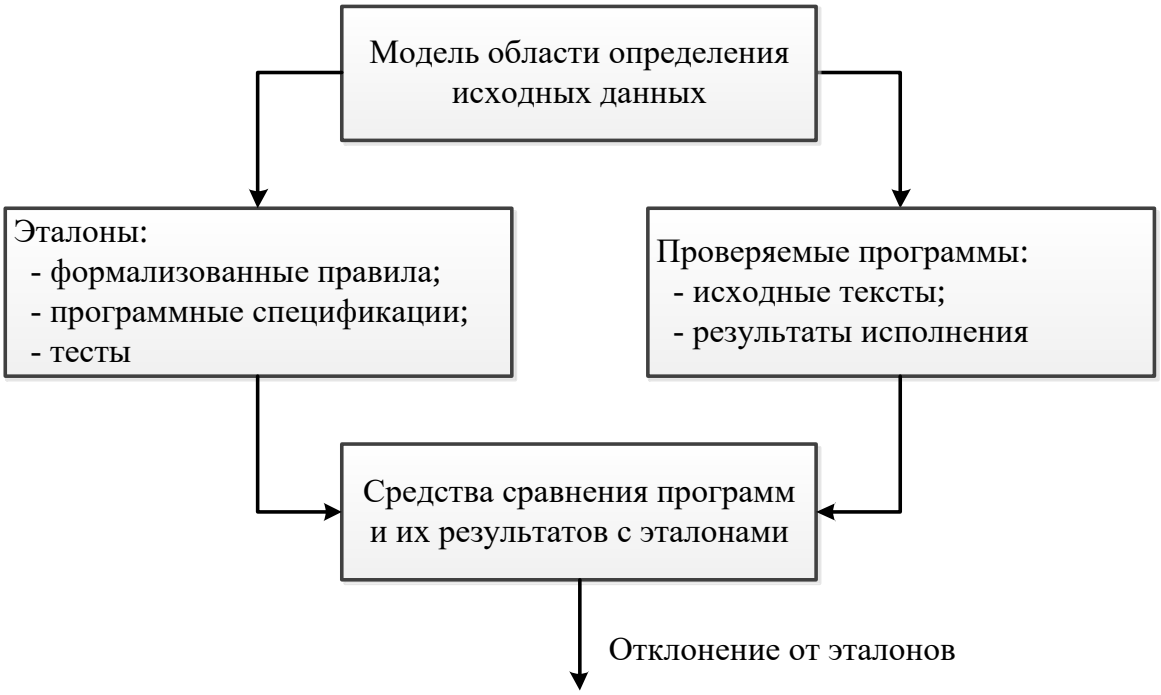


Рис. 19. Схема взаимодействия компонент, определяющих обнаруживаемые отклонения программ от эталонов

Как формируются эталоны для тестирования? Существует несколько способов формирования эталонов (рис. 20):

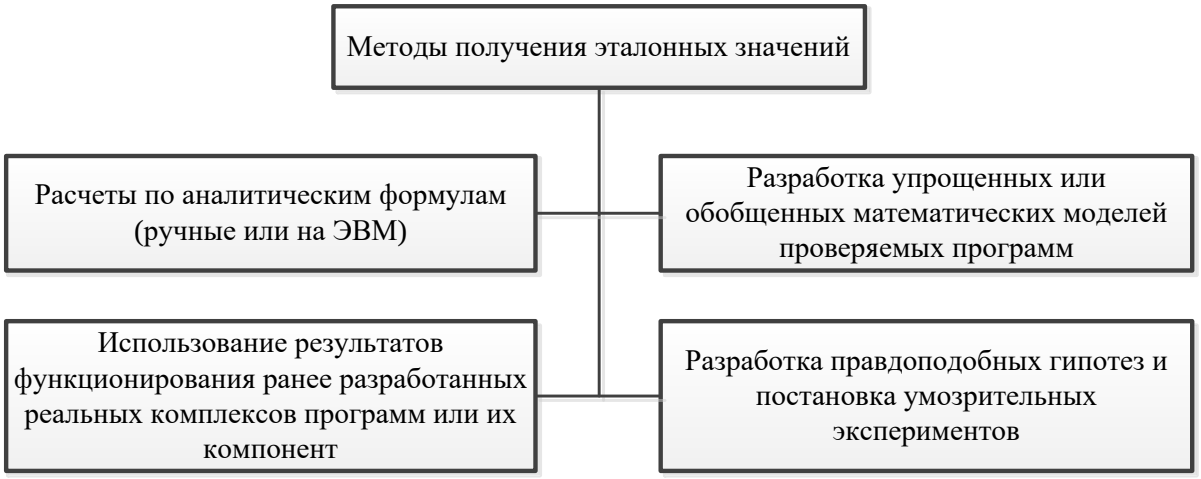


Рис. 20. Методы получения эталонных значений

1) Использование аналитических выражений. Этот способ особенно подходит при детерминированном тестировании, так как имеется возможность сравнить результаты тестирования с ожидаемыми результатами. Имеются ограничения в использовании этого метода, если неизвестны или отсутствуют аналитические выражения, связывающие входные данные и результаты; иногда требуется использовать много допущений.

2) Использование моделирования на ЭВМ. Способ является универсальным. При этом ряд данных моделируется другим способом и по другим алгоритмам, нежели испытываемая программа, и на других ЭВМ. При этом наборы входных данных создаются по случайным законам, что обеспечивает высокую гибкость этого способа.

3) Использование результатов функционирования ранее разработанных реальных комплексов программ или их компонент. При этом используется ранее накопленный опыт испытателя или других исследователей, выраженный в экспертных оценках ожидаемых результатов.

Степень достоверности проверки корректности программ при использовании этих методов убывает по номерам способов формирования эталонов.

В первом случае обеспечивается 100% гарантия корректности программ, в третьем случае такой уверенности нет, но мы можем убедиться в том, что программа работает так же или иначе, чем аналогичный вариант. Менее достоверные тесты приходится использовать из-за недостаточности сил и средств.

3.3. Верификация программ

Корректность является статическим свойством программы, поскольку она не зависит от времени (если, конечно, не изменяются цели разработки) и отражает специфику ошибок разработки программ (ошибок проекта и кодирования).

Различают два типа проверки корректности:

- **верификация** (verification) – установление соответствия между программой и ее спецификацией;

- **валидация** (validation) – установление соответствия между тем, что делает программа, и тем, что нужно заказчику.

Верификация и валидация программного обеспечения – это упорядоченный подход в оценке программных продуктов, применяемый на протяжении всего жизненного цикла. Усилия, прилагаемые в рамках работ по верификации и валидации, направлены на обеспечение качества как неотъемлемой характеристики программного обеспечения и удовлетворение пользовательских требований» (пользовательские требования – это не user requirements в понимании управления требованиями, а потребности пользователей, ради удовлетворения которых создается программное обеспечение).

Верификация и валидация напрямую адресуется вопросам качества программного обеспечения и использует соответствующие техники тестирования для обнаружения тех или иных дефектов. Верификация и валидация может применяться для промежуточных продуктов, однако, в том объеме, который соответствует промежуточным «шагам» соответствующих процессов жизненного цикла.

Процессы верификации и валидации определяют, в какой степени продукт (результат) тех или иных работ по разработке и сопровождению соответствует требованиям, сформулированным в рамках этих работ, а конечный продукт удовлетворяет заданным целям и пользовательским требованиям (корректнее было бы говорить не только и, может быть, не столько о «требованиях», то есть потребностях, сколько об ожиданиях).

Верификация – попытка обеспечить правильную разработку продукта (продукт построен правильным образом; обычно, для промежуточных, иногда, для конечного продукта), в том значении, что получаемый в рамках соответствующей деятельности продукт соответствует спецификациям, заданным в процессе предыдущей деятельности.

Валидация – попытка обеспечить создание правильного продукта (построен правильный продукт; обычно, в контексте конечного продукта), с точки зрения достижения поставленной цели.

Оба процесса – верификация и валидация – начинаются на ранних стадиях разработки и сопровождения. Они обеспечивают исследованию (экспертизу) ключевых возможностей продукта как в контексте непосредственно предшествующих результатов (промежуточных продуктов), так и с точки зрения удовлетворения соответствующих спецификаций.

Верификация и валидация являются видами деятельности, направленными на контроль качества программного обеспечения и обнаружение ошибок в нем. Имея общую цель, они отличаются источниками проверяемых в их ходе свойств, правил и ограничений, нарушение которых считается ошибкой.

Валидация является менее формализованной деятельностью, чем верификация. Она всегда проводится с участием представителей заказчиков, пользователей, бизнес-аналитиков или экспертов в предметной области – тех, чье мнение можно считать достаточно хорошим выражением реальных нужд и потребностей пользователей, заказчиков и других заинтересованных лиц. Методы ее выполнения часто используют специфические техники выявления знаний и действительных потребностей участников.

Определение верификации симметрично ввиду относительного характера свойства корректности: если программа не соответствует своей спецификации, то либо программа, либо спецификация, либо оба этих объекта содержат ошибки. Следовательно, можно сказать, что верификация способна доказать отсутствие ошибок в программе, но не всегда доказывает их присутствие.

Различие между верификацией и валидацией проиллюстрировано на рис. 21.

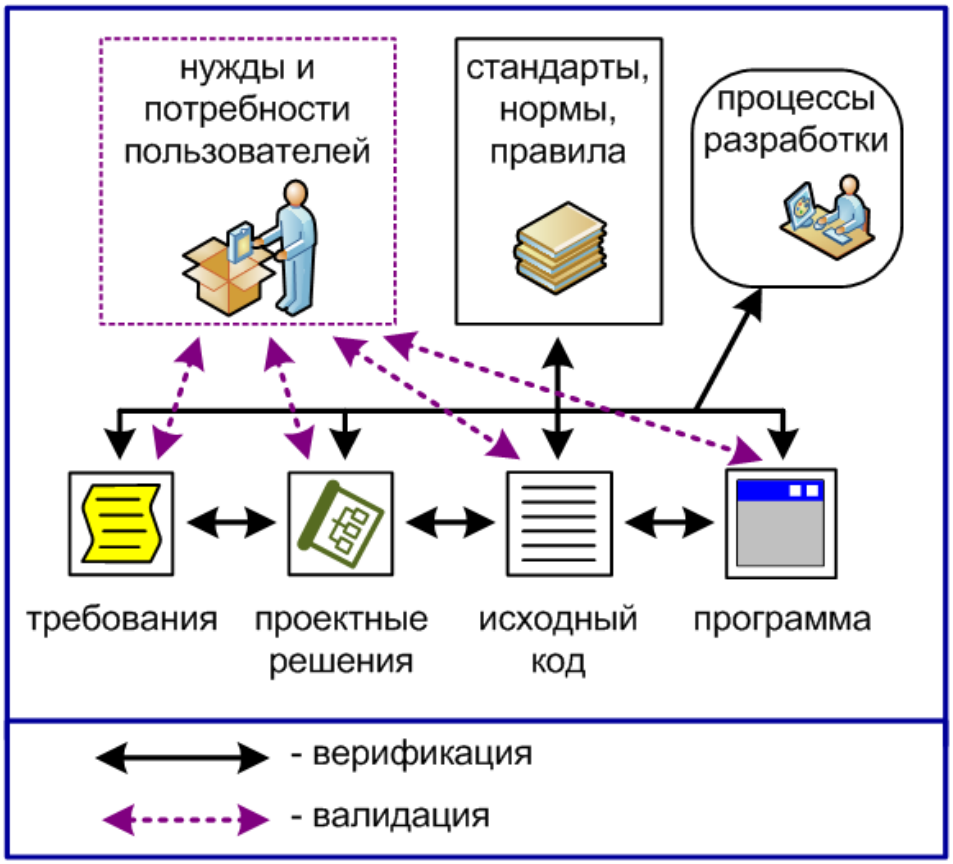


Рис. 21. Отличие между верификацией и валидацией

Таким образом, верификация отвечает на вопрос «Делаем ли мы продукт правильно?», а валидация – на вопрос «Делаем ли мы правильный продукт?»

В отличие от тестирования, верификация предполагает аналитическое исследование свойств программы по ее тексту (без выполнения программы). Таким образом, для проведения верификации необходимо построить формально-логическую систему, в которой были бы формально определены свойства корректности программы. Для этих целей наиболее широкое применение получило исчисление предикатов первого порядка, расширенное аксиоматической арифметикой.

Можно сказать, что верификация – это вычисление истинности предиката от двух аргументов: программы и спецификации. Истинность

этого предиката устанавливает свойство корректности программы, если спецификация не содержит ошибок, однако, его ложность, вообще говоря, не означает наличие ошибок в программе, а требует более точного разбора.

Учитывая специфику проявления ошибок в программах в процессе их выполнения на ЭВМ, целесообразно выделить в понятие корректности программы два свойства:

- **частичную корректность** – удовлетворение внешним (входной и выходной) спецификациям программы при условии завершения выполнения программы;
- **завершенность** – достижение в процессе выполнения выхода программы при определенных входной спецификацией данных.

Свойство завершенности не является тривиальным для такого объекта, как программа, ввиду возможности циклических и рекурсивных вычислений, а также наличия частично-определенных операций. Случаи, когда свойство завершенности не удовлетворяется, довольно обычны для программ с ошибками, и приводят к заведомо некорректным программам независимо от спецификации выхода.

Каждое из двух выделенных свойств корректности программы может удовлетворяться или не удовлетворяться. Таким образом, можно выделить шесть основных задач анализа корректности программы:

1) доказательство частичной корректности (при условии завершенности);
2) доказательство частичной некорректности (некорректность при условии завершенности);
3) доказательство завершения программы;
4) доказательство не завершения программы;
5) доказательство тотальной (полной) корректности (т. е. одновременное решение 1-й и 3-й задач);
6) доказательство некорректности (решение 2-й или 4-й задачи).

Разделение свойств частичной корректности и завершенности следует рассматривать как методологический прием, направленный на уменьшение сложности верификации программ.

Методы доказательства частичной корректности программ, как правило, опираются на аксиоматический подход к формализации семантики языков программирования. Аксиоматическая семантика языка программирования представляет собой совокупность аксиом и правил вывода. С помощью аксиом задается семантика простых операторов языка (присваивания, ввода-вывода, вызова процедур). С помощью правил вывода описывается семантика составных операторов или управляющих структур

(последовательности, условного выбора, циклов). Среди этих правил вывода надо отметить правило вывода для операторов цикла, так как оно требует знания инварианта цикла (формулы, истинность которой не изменяется при любом прохождении цикла).

Наиболее известным из методов доказательства частичной корректности программ является метод индуктивных утверждений, предложенный Флойдом [1, 30] и усовершенствованный Хоаром [28]. Один из важных этапов этого метода – получение аннотированной программы. На этом этапе для синтаксически правильной программы должны быть заданы утверждения на языке логики предикатов первого порядка: входной предикат; выходной предикат.

Эти утверждения задаются для входной точки цикла и должны характеризовать семантику вычислений в цикле.

Доказательство неистинности условий корректности свидетельствует о неправильности программы или ее спецификации, или программы и спецификации.

Пример. Пусть A – это некоторая операция, тогда **формула корректности (correctness formula)** – это выражение вида

$$\{P\}\ A\ \{Q\},$$

где P является предусловием, Q – постусловием, а A обозначает операцию.

Формула корректности, называемая также триадой Хоара, говорит следующее: любое выполнение операции A, начинающееся в состоянии, где P истинно, завершится и в заключительном состоянии будет истинно Q.

Приведенная выше формула корректности определяет **полную корректность** (**total correctness**), которая гарантирует завершаемость операции A. Помимо этого существует понятие **частичной корректности (partial correctness)**, которое гарантирует выполнение постусловия Q только при условии завершения выполнения операции A.

Рассмотрим следующую триаду Хоара:

$$\{x = 5\}\ x = x^2\ \{x > 0\}.$$

Эта триада корректна, так как если перед выполнением операции x^2, предусловие выполняется и значение $x = 5$, то после выполнения этой операции, постусловие $(x > 0)$ будет гарантированно выполняться (при условии корректной реализации целочисленной арифметики). Из этого примера видно, что приведенное постусловие не является самым сильным. В приведенном примере самым сильным постусловием при заданном предусловии является $\{x = 25\}$, а самым слабым предусловием при заданном постусловии является $\{x > 0\}$. Из выполняемой формулы

корректности всегда можно породить новые выполняемые формулы, путем ослабления постусловия или усиления предусловия.

Понятие «сильнее» и «слабее» пришли из логики. Говорят, что условие P_1 сильнее, чем P_2, а P_2 слабее, чем P_1, если выполнение условия P_1 влечет за собой выполнение условия P_2, но они не эквивалентны. (Например, условие $x > 5$ (P_1), сильнее условия $x > 0$ (P_2), поскольку при выполнении условия P_1 выполняется и условие P_2 (ведь, если $x > 5$, то, естественно, что $x > 0$), при этом эти условия не эквивалентны). Из этого следует, что условие *True* является слабейшим (поскольку при выполнении любого условия, выполняется условие *True*), а *False* – сильнейшим (поскольку по своему определению, любое условие, не равное *False* когда-либо выполняется) (см. рис. 22).

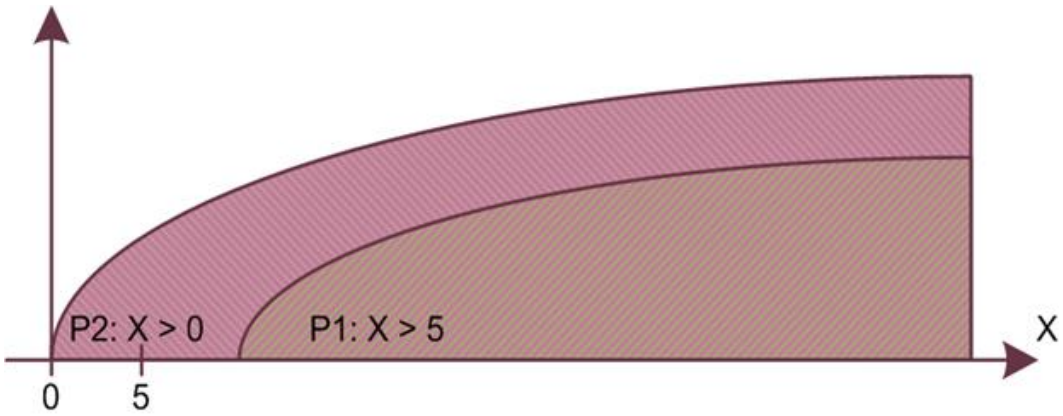


Рис. 22. Сильные и слабые условия (P_1 сильнее P_2)

Методы верификации программного обеспечения предназначены для подтверждения фактов соответствия свойств ПО заявленным требованиям. Такие методы разнообразны и разнородны, как по своему назначению, так и по способам достижения конечного результата и включают как эмпирические, так и формальные доказательства подтверждения наличия у ПО предопределенных свойств.

В процессе верификации ПО может достигаться и еще одна цель, состоящая в поиске некорректно реализованных свойств, невыполненных требований и сопутствующем обнаружении ошибок в ПО. В связи с этим, современные методы верификации ПО могут включать в себя методы тестирования ПО. Методы верификации ПО в целом можно разделить на структурные и функциональные, а также имеющие в своей основе формальную математическую модель, либо зависящие от квалификации лиц, принимающих решение о корректности ПО.

Методы верификации ПО, нацеленные на оценку технического состояния и работоспособности, разделяются на следующие группы (рис. 23):

- Экспертиза;
- Статический анализ;
- Динамические методы;
- Формальные методы;
- Синтетические методы.

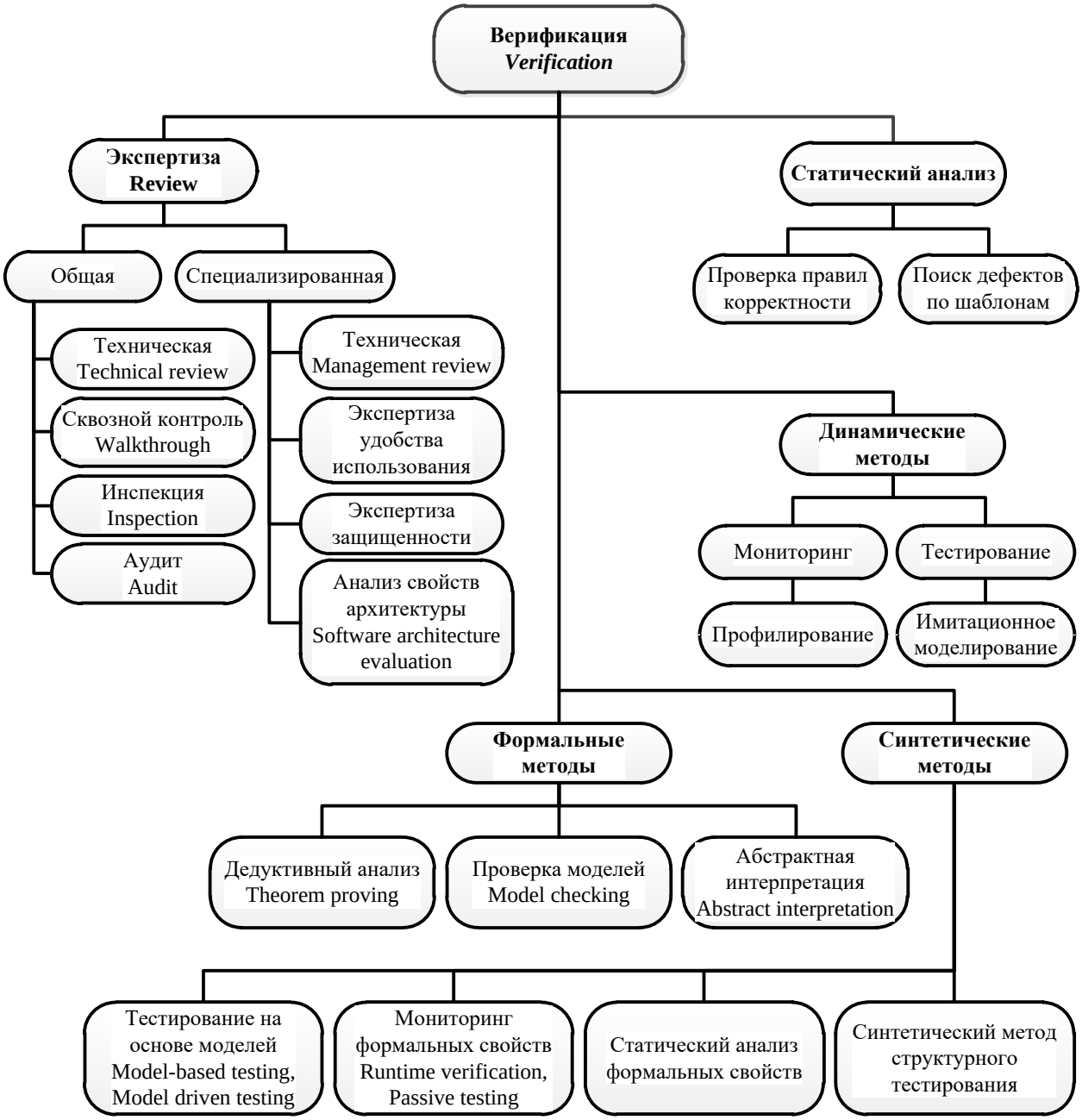


Рис. 23. Методы верификации ПО

Методы первой группы в основном, представляют собой тестирование и экспертный анализ свойств ПО и его соответствие некоторому образцу.

Экспертиза (review, переводится как рецензирование, просмотр, обзор, оценка, анализ). Отличительной особенностью экспертизы при верификации ПО является наличие ряда международных стандартов[5-11, 36, 38].

В целом, при таком подходе к верификации ПО, наблюдается ориентированность на экспертные оценки, которые при рассмотрении различных парадигм программирования изменяются. Таким образом, их нельзя отнести к универсальным и строго формализованным. Кроме того, экспертиза применима только непосредственно к самому ПО, а не к формальным моделям, или результатам работы. В качестве видов экспертиз выделяются:

- организационные экспертизы (management review);
- технические экспертизы (technical review);
- сквозной контроль (walkthrough);
- инспекции (inspection);
- аудиты (audit).

Различные виды экспертиз применяются к различным свойствам ПО на различных этапах проектирования. Своевременное обнаружение дефектов позволяет снизить до минимума риск некорректной работы ПО в дальнейшем.

Особое место среди экспертиз занимают систематические методы анализа архитектуры ПО, такие как SAAM (Software Architecture Analysis Method) [31] и ATAM (Architecture Tradeoff Analysis Method) [32, 33].

Методы статического анализа проводят сравнение некоторого ПО с заранее определенным шаблоном.

Статический анализ корректного построения исходного кода или поиск часто встречающихся дефектов по некоторым шаблонам хорошо автоматизирован и практически не нуждается в процедурах принятия решений человеком. Однако, круг ошибок, выявляемых статическим анализом, достаточно узок. Одним из применений шаблонов типичных ошибок является их включение в семантические правила компиляторов языков программирования.

Динамические методы анализируют уже готовый, работающий продукт и выявляют проблемы с работоспособностью.

Динамические методы верификации применяются для анализа и оценки уже результатов работы ПО, либо некоторых ее прототипов и моделей.

Примерами таких методов являются тестирование ПО или имитационное тестирование, мониторинг и профилирование.

Динамические методы применяются на этапе эксплуатации ПО, поэтому необходимо иметь уже работающую систему или хотя бы некоторые ее

работающие компоненты. Такие методы нельзя использовать на стадиях разработки и проектирования ПО, зато с их помощью можно контролировать характеристики работы системы в ее реальном окружении, которые иногда невозможно точно определить с помощью других подходов. Динамические методы позволяют обнаруживать в ПО только ошибки, проявляющиеся при его функционировании. Это обстоятельство ограничивает область их применения, например, с их помощью невозможно найти ошибки ПО, связанные с удобством сопровождения программы.

Классификация видов тестирования достаточно сложна, потому что может проводиться по нескольким разным классификационным признакам [22].

По уровню или масштабу проверяемых элементов системы тестирование делится на следующие виды:

- модульное или компонентное (unit testing, component testing);
- интеграционное (integration testing);
- системное (system testing);
- регрессионное (regression testing).

По источникам данных, используемых для построения тестов, тестирование относится к одному из следующих видов:

1) Тестирование черного ящика (black-box testing), часто также называется тестированием соответствия (conformance testing), или функциональным тестированием (functional testing);

2) Тестирование белого ящика (white-box testing, glass-box testing), а также структурное тестирование (structural testing);

3) Тестирование серого ящика (grey-box testing);

4) Тестирование, нацеленное на ошибки.

По роли команды, выполняющей тестирование, оно может относиться к следующим видам:

- внутреннее тестирование;
- независимое тестирование;
- аттестационное тестирование (приемочные испытания);
- пользовательское тестирование.

Отличительной особенностью формальных методов верификации является возможность проведения поиска ошибок на математической модели, без обращения к физической реализации, что в некоторых случаях довольно удобно и экономично. Для проведения анализа формальных моделей применяются специфические техники, такие как дедуктивный анализ (theorem proving), проверка моделей (model checking) [20], абстрактная интерпретация (abstract interpretation). К сожалению, для

построения таких моделей всегда необходимо исходить так же из корректности и адекватности модели ПО. Лишь после правильного построения этой модели можно автоматически проанализировать некоторые из ее свойств. Тем не менее, в большинстве случаев для эффективного анализа от специалистов потребуются глубокие знания математической логики и алгебры и некоторого набора навыков работы с этим аппаратом.

Понятия логических и алгебраических исчислений довольно близки. Но для некоторой ясности и определенности можно сказать, что логика применима к утверждениям, записанным на каком либо языке, а алгебра работает с равенствами и неравенствами, записанными на языке выражений.

Логические исчисления могут быть классифицированы следующим образом:

1) Исчисление высказываний (пропозициональное исчисление, propositional calculus);

2) Исчисление предикатов (predicate calculus);

3) Исчисления предикатов более высоких порядков (higher-order calculi);

4) λ-исчисление (lambda calculus);

5) Модальные логики (modal logics);

6) Временные логики (temporal logics);

7) Логика дерева вычислений (Computation Tree Logic, CTL);

8) μ-исчисление (или исчисление неподвижных точек);

9) Логики явного времени (timed temporal logics).

Другим направлением построения моделей верификации являются алгебраические модели, к которым относятся:

- реляционные алгебры;
- алгебраические модели абстрактных типов данных;
- алгебры (исчисления) процессов (process calculi).

Внимание исследователей привлекают и другие модели:

1. Исполнимые (операционные) модели (executable models) – это модели, с помощью которых изучаются свойства моделируемого ПО. Каждая исполнимая модель является, по сути, программой для некоторой виртуальной машины.

Примерами исполнимых моделей являются:

- конечные автоматы (finite state machines, FSM);
- конечные системы помеченных переходов (или просто системы переходов, labeled transition systems, LTS);
- расширенные конечные автоматы (extended finite state machines, EFSM);

- взаимодействующие автоматы (communicating finite state machines, CFSM);

- иерархические автоматы (hierarchical state machines);

- временные автоматы (timed automata) [41];

- гибридные автоматы (hybrid automata);

- сети Петри (Petri nets);

- обычные конечные автоматы (их принято называть ω-автоматами);

- машины абстрактных состояний (abstract state machines).

2. Модели промежуточного типа имеют черты как одних – логико-алгебраических, так и других – исполнимых. Стоит отметить, что часть перечисленных выше примеров может быть отнесена к обоим этим классам. Есть, однако, виды моделей, в которых логико-алгебраические и исполнимые элементы сочетаются более глубоко. К ним относятся:

- логики Хоара (Hoare logics);

- динамические или программные логики (dynamic logics, program logics);

- программные контракты (software contracts).

Чаще всего для проверки этих моделей используются методы дедуктивного анализа (theorem proving), проверки моделей (model checking), согласованности (conformance).

Синтетические методы верификации сочетают техники нескольких типов – статический анализ, формальный анализ свойств ПО, тестирование. К таким методам относятся:

- тестирование на основе моделей (model based testing);

- мониторинг формальных свойств ПО (runtime verification и passive testing);

- метод статического анализа формальных свойств;

- синтетические методы генерации структурных тестов

ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ К РАЗДЕЛУ 3

1. Что такое корректность программ?
2. Приведите классификацию видов корректности комплексов программ.
3. Охарактеризуйте корректность текстов программ.
4. Охарактеризуйте корректность программных модулей.
5. Охарактеризуйте корректность данных.
6. Охарактеризуйте корректность групп и комплексов программ.
7. Опишите программные эталоны и методы проверки корректности программ.
8. Охарактеризуйте основные задачи анализа корректности программы
9. Дайте определения терминам «верификация» и «валидация».
10. Опишите основные методы верификации программного обеспечения.
11. Опишите основные методы тестирования программного обеспечения.

4. ОЦЕНИВАНИЕ НАДЕЖНОСТИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

4.1. Основные понятия надежности программного обеспечения

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 [15]. Таким образом, надежность является внутренним свойством системы, заложенным при ее создании и проявляющимся во времени при функционировании и эксплуатации.

Надежность программного обеспечения определяется как уровень, при котором система программ удовлетворяет поставленным требованиям и пригодна для эксплуатации [27]. 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.

Надежность технических систем определяется в основном двумя факторами: надежностью компонентов и дефектами в конструкции, допущенными при проектировании или изготовлении.

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 требованиям функциональной пригодности и работоспособности.

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

В ПО отсутствуют физический износ и старение, в связи с чем исключается возникновение дефектов, имеющих стохастическую природу. Поэтому нет необходимости в проведении планово-предупредительных ремонтов ПО в процессе эксплуатации. Модернизация ПО, целенаправленное внесение изменений в него создает, по существу, другое ПО, имеющее свои характеристики надежности. Поэтому при сопровождении ПО, в отличие от других видов продукции, можно обеспечивать не только поддержание показателей надежности на некотором уровне, но и увеличение этого уровня.

Для ПО характерен более высокий темп морального старения, в связи с чем требуется создание новых видов ПО для систем обработки данных. Корректировка при модификации ПО в процессе его эксплуатации является одним из факторов снижения надежности ПО.

Введение различных видов избыточности в ПО, в частности его резервирование, малоэффективно. Избыточность как традиционное средство повышения надежности в случае ПО обычно реализуется за счет контроля входных, промежуточных и окончательных результатов вычислений. Недостаточность такого контроля является одним из основных факторов утраты работоспособности ПО. Еще одним видом избыточности является использование механизма контрольных точек, когда фиксируется состояние вычислительной среды в определенные моменты времени и выполняются рестарты промежуточных вычислений в случае необходимости.

К другим отличиям ПО по сравнению с техническими системами можно отнести следующие факторы:

- число способов и элементов контроля у ПО значительно больше, чем у аппаратуры;
- в ПО гораздо проще вносить исправления и дополнения, чем в аппаратуру, но это трудно делать корректно и безошибочно.

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 помехозащищенности;
- 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.

Основные определения

Надежность программных средств – совокупность свойств, характеризующая способность программных средств сохранять заданный уровень пригодности в заданных условиях в течение заданного интервала времени.

С учетом необходимости оценивания характеристики «надежность» применительно к ПС необходимо ввести и понятие отказа программных средств.

Отказом в теории надежности технических средств называется событие, состоящее в потере работоспособности системы. Отказ как событие может не только быть связанным с полной потерей работоспособности (для простых систем), но и с потерей части качества процесса функционирования системы, части свойства ее работоспособности (для сложных систем). Понятие «отказ» для сложной системы достаточно сложно сформулировать. Его необходимо связывать с понятием эффективности процесса функционирования системы, то есть со свойством более сложным, чем надежность. Поэтому часто

отмечают две стороны этого понятия: субъективную (понятие «отказ» формулируется специалистами по-разному) и объективную. Далее будем иметь в виду первую точку зрения, не формулируя понятие отказа программного обеспечения с позиций эффективности процесса функционирования.

Отказом программного средства будем называть событие, характеризующееся неправильным функционированием информационной системы, обусловленным, по мнению специалистов, ПС информационной системы. Данное определение позволяет оценивать свойство надежности программных средств по правилам, сходным с оцениванием надежности аппаратных средств. В дальнейшем это дает возможность получать оценки надежности информационной системы с учетом различных объектов, входящих в ее состав.

Природа отказов ПС принципиально отличается от природы отказов аппаратных средств. Программные средства, в отличие от технических, не подвержены физическому старению, на них не воздействуют физико-химические процессы, протекающие в информационной системе. Причины отказов ПС заключаются в наличии дефектов, не выявленных на ранних стадиях создания ПС.

Под **дефектом программы** будем понимать «изъян», приводящий к нарушению процесса ее функционирования. Иначе говоря, дефект есть несоответствие свойств, функций, реализованных в программе, смысловому содержанию решаемой задачи. Важным фактором появления дефектов в программе является отсутствие эталона, который должен соответствовать тексту программы и результатам ее функционирования.

Дефекты программных средств достаточно многообразны, как и причины, их обуславливающие. Поскольку дефекты имеют различную природу (например, неверная трактовка требований заказчика, неправильный код программы и т.д.), то для определенности будем говорить, что отказы ПС вызываются их ошибками.

Ошибки ПС являются функцией от входной информации и состояния системы. Ошибки имеют систематический характер.

Под **ошибкой** будем подразумевать причину отказа, которая обусловлена некоторым допущенным дефектом при создании или модификации ПС.

Основными причинами, непосредственно вызывающими нарушение нормального функционирования программы, являются:

1. Скрытые ошибки программы, к которым относятся:

- ошибки вычислений;
- логические ошибки;

- ошибки ввода-вывода;
- ошибки манипулирования данными;
- ошибки совместимости;
- ошибки сопряжений;

2. Искажения входной информации, подлежащей обработке, причинами которых являются:
- искажения данных на первичных носителях информации;
- сбои и отказы в аппаратуре ввода данных с первичных носителей информации;
- шумы и сбои в каналах связи при передаче сообщений;

3. Неверные действия пользователя;

4. Неисправность аппаратуры, на которой реализуется вычислительный процесс.

Дать строгое определение **программной ошибки** непросто, поскольку это определение является функцией от самой программы, то есть зависит от того, какого функционирования ожидает от программы пользователь. По этой причине вместо строгого определения будут перечислены только признаки, указывающие на наличие ошибки в программе:
- Авария операционной системы;
- Аварийный отказ прикладного программного обеспечения;
- Программные ошибки пользователя;
- Снижение производительности;
- Зацикливание;
- Нарушение защиты данных;
- Потеря или искажение данных;
- Потеря функциональных возможностей (50% отказов).

Этот список можно считать открытым, поскольку он может быть продолжен разработчиками по мере накопления ими опыта в повышении надежности программного обеспечения.

Основные свойства программных ошибок:

1) Ошибки в программе – величина ненаблюдаемая, наблюдаются не сами ошибки, а результат их проявления – отказы;

2) Надежность связана с частотой проявления ошибок, но не с их количеством – разные ошибки имеют разную частоту проявления;

3) Отказ может быть следствием не одной, а сразу нескольких ошибок;

4) Ошибки могут компенсировать друг друга – после исправления ошибки интенсивность отказов может увеличиться;

5) В результате исправления ошибки или любого другого изменения получается новая программа с другими показателями надежности.

Процесс проявления ошибок ПС, то есть процесс отказов ПС, носит случайный характер. Это обусловлено случайным для пользователя сочетанием путей вычислительного процесса, что приводит к случайному характеру проявления ошибки ПС. Особенно это справедливо для систем реального времени.

Надежность ПО определяется его безотказностью, безошибочностью и восстанавливаемостью.

Безотказность программного обеспечения есть его свойство сохранять работоспособность при использовании в процессе обработки информации на компьютере.

Безотказность ПО можно оценивать вероятностью его работы без отказов при определенных условиях внешней среды в течение заданного времени наблюдения.

Безотказность программного средства можно также характеризовать средним временем между возникновениями отказов в функционировании программы. При этом предполагается, что аппаратура компьютера находится полностью в работоспособном состоянии.

Безошибочность программного обеспечения есть его свойство соответствовать предъявленным требованиям в процессе функционирования в течение заданного времени в условиях определенной среды.

Важной характеристикой надежности ПО является его **восстанавливаемость**, которая определяется затратами времени и труда на устранение отказа из-за проявившейся ошибки в программе и его последствий.

Восстановление после отказа в программе может заключаться в корректировке и восстановлении текста программы, исправлении данных, внесении изменений в организацию вычислительного процесса.

Восстанавливаемость ПО может быть оценена средней продолжительностью устранения ошибки в программе и восстановления ее работоспособности. Восстанавливаемость ПО зависит от многих факторов: от сложности структуры комплекса программ, алгоритмического языка, на котором разрабатывалась программа, стиля программирования, качества документации на программу и т.д.

Надежность ПС базируется на понятиях корректности и устойчивости.

Программа считается корректной, если она выполняет запланированные действия и не имеет побочных эффектов. Корректность – узкое понятие, так как ее полная проверка для большинства программных систем практически невыполнима. Корректность базируется на тщательной спецификации требований пользователя. Возможна ситуация, когда программа корректна с

точки зрения разработчика, но не корректна с точки зрения пользователя. Если ПО удовлетворяет своей спецификации, то оно корректно. Систему можно считать надежной, если велика вероятность того, что при обращении к ней можно получить требуемую услугу. Программа рассматривается как ненадежная, если она допускает сбои в особых ситуациях, даже если эти ситуации имеют небольшой процент от времени пользования системой.

Под устойчивостью программы понимается ее способность правильно выполнять правильное действие при наличии отказов в работе аппаратуры и ошибках в исходных данных. При оценке устойчивости должны быть заданы параметры окружающей среды, по отношению к которым программа должна быть устойчивой. Обычно разработчик ПО не располагает средствами эффективного контроля за окружающей средой и его задача сводится к локализации нарушений или изменения характеристик окружающей среды и описанию способов их устранения.

4.2. Показатели надежности программного обеспечения

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.

К основным показателям надежности ПО относят следующие:

1. **Вероятность безотказной работы** $P(t) = P(T \geq t)$ – это вероятность того, что в пределах заданной наработки отказ системы не возникает; T – случайная величина, характеризующая время работы ПО до отказа или наработку ПО до отказа.

2. **Вероятность отказа** $Q(t)$ – вероятность того, что в пределах заданной наработки отказ системы возникает.

Этот показатель оценивается по формуле

$$Q(t) = 1 - P(t).$$

3. **Интенсивность отказов** – это условная плотность вероятности возникновения отказа ПС в определенный момент времени при условии, что до этого времени отказ не возник

$$\lambda(t) = \frac{f(t)}{P(t)},$$

где $f(t)$ – плотность вероятности отказа в момент времени t, которая определяется по формуле

$$f(t) = \frac{d}{dt}Q(t) = \frac{d}{dt}(1 - P(t)) = -\frac{d}{dt}P(t).$$

Существует следующая связь между интенсивностью отказов системы и вероятностью безотказной работы:

$$P(t) = \exp\left(-\int_0^t \lambda(t)dt\right).$$

В частном случае, при $\lambda = const$

$$P(t) = \exp(-\lambda).$$

Если в процессе тестирования фиксируется число отказов за определённый временной интервал, то интенсивность отказов системы есть число отказов в единицу времени.

4. **Среднее время безотказной работы** $T_{\text{ср}}$ — математическое ожидание времени работы ПС до очередного отказа

$$T_{\text{ср}} = \int_0^\infty P(t)dt.$$

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 показателей надежности.

Поскольку программы имеют явно выраженные производственные циклы работы, то наработка программы может быть выражена либо через календарное время, либо через машинное время, либо через количество отработанных операторов, решённых задач и т.п.

Один из способов оценивания – наблюдение за поведением программы в определённый временной период. Тогда величину $T_{\text{ср}}$ можно определить так:

$$T_{\text{ср}} = \frac{H}{n-r},$$

где n – общее количество прогонов ПО;

r – количество прогонов ПО без ошибок;

$l = n - r$ – количество прогонов с ошибками;

H – общее количество часов успешного прогона программы, определяемое из выражения

$$H = \sum_{i=1}^{r} T_i - \sum_{j=1}^{n} t_j,$$

где T_i – время непрерывного прогона безошибочной работы ПО (в часах);

t_j – время прогона до проявления ошибки ПО (в часах).

Полагая количество ошибок постоянным, можно вычислить интенсивность отказов ПО, приведённую к одному часу работы, и среднее время между соседними отказами ПО.

$$\lambda = \frac{n-r}{H} = \frac{l}{H};\ T_{\text{ср}} = \frac{1}{\lambda} = \frac{H}{l}.$$

5. **Среднее время восстановления** $T_{\text{в}}$ – математическое ожидание времени восстановления $\left(t_{\text{в}i}\right)$, складывающегося из времени, затраченного на восстановление и локализацию отказа $\left(t_{\text{о.л.}i}\right)$, времени устранения отказа $\left(t_{\text{у.о.}i}\right)$, времени пропускной проверки работоспособности $\left(t_{\text{п.п.}i}\right)$:

$$t_{\text{в}i} = t_{\text{о.л.}i} + t_{\text{у.о.}i} + t_{\text{п.п.}i};$$

$$T_{\text{в}} = \frac{1}{n}\sum_{i=1}^{n} t_{\text{в}i}.$$

Для этого показателя термин «время» означает время, затраченное специалистом по тестированию на перечисленные виды работ.

6. **Коэффициент готовности** $K_{\text{г}}$ — вероятность того, что ПС окажется в работоспособном состоянии в произвольный момент времени его использования по назначению

$$K_{\text{г}} = \frac{T_{\text{ср}}}{T_{\text{ср}} + T_{\text{в}}}.$$

Значение коэффициента готовности соответствует доле времени полезной работы системы на достаточно большом интервале, содержащем отказы и восстановления.

7. Традиционные критерии надежности ПО характеризуют наличие ошибок программы (производственных дефектов), но ни один из них не характеризует характер этих ошибок и возможные их последствия. Поэтому вводится дополнительный критерий надежности ПО – **средняя тяжесть ошибок** (СТО) [2], определяемый выражением

$$CTO = \frac{1}{Q}\sum_{i=1}^{m} b_i p_i z_i ,$$

где Q – вероятность сбоя ПО; b_i – функция принадлежности тяжести последствий ошибки, возникшей при i-м наборе входных данных, к максимально тяжелым последствиям; p_i – вероятность ввода i-го набора входных данных при эксплуатации ПО; z_i – дихотомическая переменная, равная 1, если при i-м наборе входных данных был зафиксирован сбой, и 0 в противном случае; m – общее число наборов входных данных.

Значение показателя надежности СТО лежит на интервале [0; 1]. Чем ближе значение СТО к единице, тем тяжелее последствия ошибок ПО, и тем менее надежна программа. Близость СТО к нулю показывает незначительность последствий ошибок программы.

Введение данного показателя надежности ПО позволяет характеризовать не столько безошибочность ПО, сколько его безопасность. Однако следует помнить, что значение этого критерия субъективно и может быть различным для одного и того же программного продукта в зависимости от области его применения. Это объясняется тем, что при использовании конкретного ПО, например для выполнения студенческих расчетов и для выполнения конструкторских расчетов в космической промышленности последствия ошибок программы несопоставимы. В ряде случаев, если к ПО предъявляются жесткие требования, лучше оценивать максимальную тяжесть ошибок ПО.

Необходимо стремиться повышать уровень надежности ПС, но достижение 100%-й надежности лежит за пределами возможного. Количественные показатели надежности могут использоваться для оценки достигнутого уровня технологии программирования и для выбора метода проектирования будущего программного средства.

Способы обеспечения и повышения надежности ПО:

1) Совершенствование технологии программирования;

2) Выбор алгоритмов, не чувствительных к нарушениям вычислительного процесса (использование алгоритмической избыточности);

3) Резервирование программ – дуальное и *N*-версионное программирование (использование структурной избыточности);

4) Контроль и тестирование программ с последующей коррекцией.

4.3. Модели надежности программного обеспечения

Модели надежности строятся на основании данных о сбоях, собранных в процессе тестирования программного обеспечения или его использования. Такие модели могут быть использованы для предсказания будущих сбоев и помогают в принятии решения о прекращении тестирования.

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 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 [4], приведенную в табл. 4.

Таблица 4 – Классификация моделей надежности ПС

<table>
<tr><td colspan="5">Модели надежности программных средств</td></tr>
<tr><td colspan="4">Аналитические</td><td rowspan="2">Эмпирические</td></tr>
<tr><td colspan="2">Динамические</td><td colspan="2">Статические</td></tr>
<tr><td>Дискретные</td><td>Непрерывные</td><td>По области ошибок</td><td>По области данных</td><td>Модель сложности</td></tr>
<tr><td>Модель Шумана</td><td>Модель Джелинского-Моранды</td><td>Модель Миллса</td><td rowspan="4">Модель Нельсона</td><td rowspan="4">Модель, определяющая время доводки программ</td></tr>
<tr><td>Модель La Padula</td><td>Модель Муса</td><td>Модель Липова</td></tr>
<tr><td rowspan="2">Модель Шика-Волвертона</td><td rowspan="2">Модель переходных вероятностей</td><td>Простая интуитивная модель</td></tr>
<tr><td>Модель Коркорэна</td></tr>
</table>

Модели надежности программных средств 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 моделях 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.

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.

Следует признать, что абсолютно надежных программ не существует, так как абсолютная степень надежности не может быть теоретически доказана и, следовательно, недостижима. Однако важно знать, насколько надежно конкретное ПО. Модели представляют теоретический подход и, как правило, имеют ограниченное применение. До сих пор не предложено ни одного надежного количественного метода оценки надежности ПО, не содержащего чрезмерного количества ограничений.

Практика разработки ПО предполагает приоритет задачи обеспечения надежности над задачей ее оценки. Ситуация выглядит парадоксально: совершенно очевидно, что прежде чем обеспечивать надежность, следует научиться ее измерять. Но для этого нужно иметь практически приемлемую единицу измерения надежности ПО и модель ее расчета.

4.3.1. Аналитические динамические модели надежности

Модель Шумана. Исходные данные для модели Шумана [40], 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.

Тестирование проводится в несколько этапов, для каждого этапа программа выполняется 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 рассчитываются количественные показатели надежности. После этого исправляются ошибки, обнаруженные на предыдущем этапе, при необходимости корректируются тестовые наборы и проводится новый этап тестирования.

При рассмотрении этой модели вводится ряд допущений и условий, которые сводятся к следующему:

1. Предполагается, что в начальный момент компоновки программных средств системы в них имеются небольшие ошибки (E – количество ошибок). С этого времени отсчитывается время отладки τ, которое включает затраты времени на выявление ошибок с помощью тестов, на контрольные проверки и т.д. При этом время исправного функционирования системы не учитывается. В течение времени τ устанавливается $\varepsilon_0(\tau)$ ошибок в расчете на одну команду машинного языка. Таким образом, удельное число ошибок на одну машинную команду, остающихся в системе после времени τ работы равно

$$\varepsilon_\tau(\tau) = \frac{E}{I} - \varepsilon_0(\tau),$$

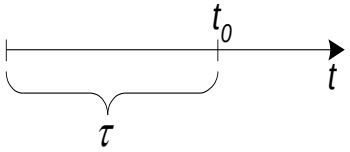
где I – общее число машинных команд.

2. Предполагается, что значение функции частоты или интенсивности отказов $\lambda(t)$ пропорционально числу ошибок, оставшихся в ПО после израсходования на отладку времени τ, то есть

$$\lambda(t) = C\,\varepsilon_\tau(\tau),$$

где C – коэффициент пропорциональности.

Тогда, если время работы системы t отсчитывается от момента времени t_0, а τ остается фиксированным (τ=const), то функция надежности или вероятность безотказной работы на интервале времени от 0 до t есть



$$P(t,\tau)=\exp\left\{-C\left[\frac{E}{I}-\varepsilon_0(\tau)\right]t\right\}.$$

Для определения C и E используются принцип максимального правдоподобия (пропорция).

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.

Надёжность ПС в течение i-го этапа оценивается по формуле

$$R(i)=R(\infty)-\frac{A}{i},\ i=\overline{1,m},$$

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

A – параметр роста (константа).

Эти неизвестные величины можно найти, решив систему уравнений

$$\begin{cases}\sum\limits_{i=1}^{m}\left[\dfrac{S_i-m_i}{S_i}-R(\infty)+\dfrac{A}{i}\right]=0;\\ \sum\limits_{i=1}^{m}\left[\left(\dfrac{S_i-m_i}{S_i}-R(\infty)+\dfrac{A}{i}\right)\cdot\dfrac{1}{i}\right]=0,\end{cases}$$

где S_i – число тестов; m_i – число отказов во время i-го этапа, $i=\overline{1,m}$; m – общее число этапов;

Определяемый по этой модели показатель есть надежность ПО на i-м этапе:

$$R(i)=R(\infty)-\frac{A}{i}\ (i=m+1,m+2,\ldots)$$

Преимущество данной 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 её выполнения.

Модель Джелинского-Моранды [35] представляет собой частный случай модели Шумана и предназначена для прогнозирования надежности программного обеспечения. Модель была использована при разработке таких значительных программных проектов, как программа «Аполло» (некоторых ее модулей).

Основное положение, на котором базируется модель, заключается в том, что в процессе тестирования ПО значение интервалов времени тестирования между обнаружением двух ошибок имеет экспоненциальное распределение с 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

$$P(t_i) = \lambda_i \cdot \exp(-\lambda_i \cdot t_i),$$

где λ_i – интенсивность отказов, которая пропорциональна числу еще не выявленных ошибок в программе

$$\lambda_i = C(N - i + 1),$$

где N – число ошибок, первоначально присутствующих в программе; C – коэффициент пропорциональности.

Наиболее вероятные значения величин $\overline{N}$ и $\overline{C}$ определяются на основе данных, полученных при тестировании. Для этого фиксируют время выполнения программы до очередного отказа $t_1, t_2, ..., t_k$. Значения $\overline{N}$ и $\overline{C}$ можно получить, решив систему уравнений

$$\begin{cases} \sum_{i=1}^{k} \left(\overline{N} - i + 1\right)^{-1} = \dfrac{k}{\overline{N} + 1 - Q \cdot k}; \\ \overline{C} = \dfrac{k}{A\left(\overline{N} + 1 - Q \cdot k\right)}, \end{cases}$$

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

Чтобы получить числовые значения λ_i, нужно подставить вместо N и C их возможные значения $\overline{N}$ и $\overline{C}$. Рассчитав k значений $\lambda_i \left(i = \overline{1, k}\right)$, 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 [39] представляет собой модификацию модели Джелинского-Моранды для случая возникновения на рассматриваемом интервале более одной ошибки.

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 (или 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):

$$\lambda_i = C\left(N - n_{i-1}\right)\left(T_{i-1} + \frac{t_i}{2}\right),$$

где C – коэффициент пропорциональности; N – число ошибок, первоначально присутствующих в программе; n_{i-1} – число ошибок, оставшихся в программе по истечении (i–1)-го интервала; T_{i-1} – суммарное время, затраченное на тестирование в течение (i–1) этапов; t_i – среднее время выполнения программы в текущем интервале.

Остальные расчеты аналогичны расчетам модели Джелинского-Моранды.

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 [26] 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 ошибок, которые присутствовали в ПС до начала тестирования.

Общее число отказов M_0 связано с первоначальным числом ошибок N_0 соотношением

$$M_0 = BN_0,$$

где B – коэффициент уменьшения числа ошибок, который определяют как число, характеризующее количество устраненных ошибок, приходящихся на один отказ.

После тестирования, за время которого зафиксировано m отказов и выявлено n ошибок, можно определить коэффициент

$$B = n / m.$$

В модели Муса различают два вида времени:

- суммарное время функционирования, которое учитывает чистое время тестирования до контрольного момента, когда проводится оценка надежности;
- оперативное 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 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.

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

По данной модели средняя наработка на отказ зависит от суммарного времени функционирования, т.е.

$$T = T_0 \exp\left(\frac{c \cdot \tau}{M_0 \cdot T_0}\right).$$

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 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, ...) 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, ...) соответствует периодам времени, когда ошибки исправляются.

Ошибки обнаруживаются и исправляются с различными интенсивностями.

Любое состояние модели определяется рядом переходных вероятностей (P_{ij}), где P_{ij} означает вероятность перехода из состояния i в состояние j и не зависит от предшествующих и последующих состояний системы, кроме состояний i и j. Вероятность перехода из состояния (n – k) к состоянию (m–k) есть $\lambda n - k \cdot \Delta t$ для $k = 0,1,2\ldots$, где λ — интенсивность проявления ошибок.

Надежность ПО в данной модели определяется выражением

$$R_k(t) = \exp\left(-\int_0^{\tau} \lambda(k,t)\,dt\right).$$

4.3.2. Аналитические динамические модели надежности

Модель Миллса. Использование этой модели [25] предполагает необходимость перед началом тестирования искусственно «засорять» программу, т.е. вносить в нее 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, называемое формулой Миллса,

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

дает возможность оценить первоначальное число ошибок в программе N. Здесь S – количество искусственно внесенных ошибок; n – число найденных собственных ошибок; V – число обнаруженных к моменту оценки искусственных ошибок.

В действительности N можно оценивать после каждой ошибки. Предлагается во время всего периода тестирования отмечать на графике число найденных ошибок и текущие оценки для N.

Вторая часть модели связана с выдвижением и проверкой гипотез о количестве ошибок N.

Примем, что в программе имеется не более k собственных ошибок и внесем в нее еще S ошибок. Теперь программа тестируется, пока не будут обнаружены все внесенные ошибки. Причем подсчитывается число обнаруженных собственных ошибок n. Уровень значимости C вычисляется по формуле

$$C = \begin{cases} 1, \text{ если } n > k; \\ \dfrac{S}{S+k+1}, \text{ если } n \le k, \end{cases}$$

где C – мера доверия к модели – вероятность того, что модель будет правильно отклонять ложные предположения.

Например, если утверждается, что в программе нет ошибок, и при внесении в программу 10 ошибок все они в процессе тестирования обнаружены, то с вероятностью 0,9 можно утверждать, что в программе нет ошибок. Но если была обнаружена 1 ошибка, то $C = 1$, так как $n > k$ и наши предположения о том, что в программе нет ошибок на 100% не подтвердились.

Таким образом, величина C является мерой доверия к модели и показывает вероятность того, насколько правильно найдено значение N.

Формулы для N и C образуют полезную модель ошибок.

Недостатки модели. Значение C нельзя предсказать до тех пор, пока не будут обнаружены все внесенные ошибки, это может не произойти до самого конца этапа тестирования.

Модификация формулы для C, если не все ошибки обнаружены:

$$C = \begin{cases} 1, \text{ если } n > k; \\ \dfrac{\left(\dfrac{S}{V-1}\right)}{\left(\dfrac{S+k+1}{k+V}\right)}, \text{ если } n \le k; \end{cases}$$

где числитель и знаменатель при $n \le k$ являются биноминальными коэффициентами вида:

$$\left(\frac{a}{b}\right) = \frac{a!}{b!(a-b)!}.$$

Модель математически проста и интуитивно привлекательна. Легко представить программу внесения ошибок, которая случайным образом выбирает модуль и вносит логические ошибки, изменяя или убирая операторы. Природа внесения ошибок должна оставаться в тайне, но все их следует регистрировать с целью последующего деления на собственные и несобственные.

Процесс внесения ошибок является самым слабым местом модели, поскольку предполагается, что для собственных и внесенных ошибок вероятность обнаружения одинакова, но неизвестна. Отсюда следует, что внесенные ошибки должны быть типичными, но на сегодня непонятно, какими именно они должны быть. Однако по сравнению с проблемами других моделей эта проблема кажется не очень сложной и разрешимой.

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 $Q(n,V)$ обнаружения n собственных и V внесенных ошибок равна

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

где m – количество тестов, используемых при тестировании; q — вероятность обнаружения ошибки в каждом из m тестов, которая рассчитывается 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:

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

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

оценки. Оценки максимального правдоподобия – наиболее вероятные значения – задаются для N следующим образом:

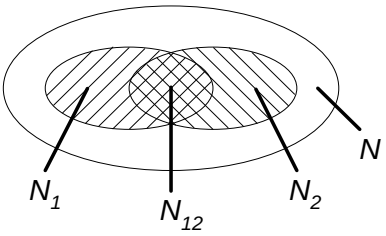
$$N = \begin{cases} \dfrac{S \cdot n}{V}, \text{ если } n \geq 1; V \geq 1; \\ S \cdot n, \text{ если } V = 0; \\ 0, \text{ если } n = 0. \end{cases}$$

Простая интуитивная модель. Эта модель (модель Б. Руднера) позволяет избавиться от главного недостатка модели Миллса.

Использование модели предполагает проведение тестирования двумя группами программистов, использующими независимые тестовые наборы, независимо одна от другой. 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. Положим, первая группа обнаружила N_1 ошибок, вторая – N_2, а N_{12} – это ошибки, обнаруженные дважды (обеими группами). Если обозначить через N неизвестное количество ошибок, присутствовавших в программе до начала тестирования, то можно эффективность тестирования каждой из групп определить как

$$E_1 = \frac{N_1}{N};$$

$$E_2 = \frac{N_2}{N}.$$



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, что

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

Тогда начальное количество ошибок, содержащихся в программе, можно оценить, используя гипергеометрическое распределение и метод максимального правдоподобия, как

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

Модель Коркорэна. В модели Коркорэна оценивается вероятность безотказного выполнения программы, при этом не используются параметры времени тестирования, а учитывается только результат N испытаний, в которых выявлено N_i ошибок i-го типа. Модель использует изменяющиеся вероятности отказов для различных типов ошибок.

Коркорэн предложил формулу для оценки вероятности безотказного выполнения программы на момент оценки на основании числа успешных прогонов, общего числа прогонов, априори известного числа типов, а также вероятности p_i выявления при тестировании ошибки i-го типа.

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

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

где N_0 – число успешных прогонов; N – общее число прогонов; k – априори известное число типов ошибок; N_i – число обнаруженных ошибок i-го типа, устраняемых с вероятностью p_i;

$$Y_i = \begin{cases} p_i, \text{ если } N_i \geq 1; \\ 0, \text{ если } N_i = 0. \end{cases}$$

В этой модели вероятность p_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 Z_i (i= 1,2, ... , k). Пусть P_i – вероятность того, что набор данных Z_i будет выбран для очередного выполнения программы. В предположении, 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, для оценки надежности ПС автор предложил простую формулу

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

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.

4.3.3. Эмпирические модели надежности программного обеспечения

Эти модели являются наиболее простыми. Они основаны на анализе накопленной информации о функционировании разработанных программ. Например [25], считалось, что если в программе на каждые 1000 операторов приходится 10 ошибок, то она пригодна к эксплуатации. По другим данным [19] уровень надежности программы считается приемлемым, если на 1000 операторов приходится одна ошибка, т.е. $N = 10^{-3}V$, где N, V – число ошибок и операторов в программе.

Эмпирические модели для оценивания числа ошибок в программах фирмой IBM представлялись в виде:

1) $N = 23M(10) + 2M(1)$, где $M(10)$ – число модулей, требующих 10 и более исправлений; $M(1)$ – число модулей, содержащих менее 10 ошибок;

2) $N = 21{,}1 + 0{,}1V + 0{,}5COMP$, где V – число операторов; $COMP$ – уровень сложности, являющийся функцией количества внутренних и внешних связей. Таким образом, в последней модели сделан упор на учет сложности ПО.

По данным работы [19] уровень дефектности программы связан с интенсивностью потока программных ошибок у пользователя зависимостью

$$\lambda_{\text{ппо}} = (10^{-2}...10^{-3})\frac{N}{V},$$

что в принципе позволяло оценивать надежность с помощью эмпирических моделей для прогнозирования состояния ПО на стадии разработки или на стадии сопровождения при модернизации ПО.

Модель М. Холстеда [29] позволяет оценивать количество ошибок в программе после окончания ее разработки по формуле

$$N = K_T\left(N_1 + N_2\right)\log_2(\eta_1 + \eta_2),$$

где K_T – коэффициент пропорциональности, учитывающий технические аспекты разработки ПО; η_1 и η_2 – число операторов и операндов в программе; N_1 и N_2 – число обращений к операторам и операндам программы соответственно.

Модель сложности. Известно, что есть тесная взаимосвязь между сложностью и надежностью ПС [3]. Если придерживаться упрощенного понимания сложности ПС, то она может быть описана такими характеристиками, как размер ПС (количество программных модулей), количество и сложность межмодульных интерфейсов.

Под программным модулем в данном случае следует понимать программную единицу, выполняющую определенную функцию (ввод, вывод, вычисление и т.д.) и взаимосвязанную с другими модулями ПС. Сложность модуля ПС может быть описана, если рассматривать структуру программы.

В качестве структурных характеристик модуля ПС используются:

- отношение действительного числа дуг к максимально возможному числу дуг, получаемому искусственным соединением каждого узла с любым другим узлом дугой;
- отношение числа узлов к числу дуг;
- отношение числа петель к общему числу дуг.

Для сложных модулей и для больших многомодульных программ составляется имитационная модель, программа которой «засоряется» ошибками и тестируется по случайным входам. Оценка надежности осуществляется по модели Миллса.

При проведении тестирования известна структура программы, имитирующей действия основной, но не известен конкретный путь, который будет выполняться при вводе определенного тестового входа. Кроме того, выбор очередного тестового набора из множества тест-входов случаен, т.е. в процессе тестирования не обосновывается выбор очередного тестового входа. Эти условия вполне соответствуют реальным условиям тестирования больших программ.

Полученные данные анализируются, проводится расчет показателей надежности по модели Миллса (или любой другой из описанных выше), и считается, что реальное ПС, выполняющее аналогичные функции, с подобными характеристиками и в реальных условиях должно вести себя аналогичным или похожим образом.

Преимущества оценки показателей надежности по имитационной модели, создаваемой на основе анализа структуры будущего реального ПС, заключаются в следующем:

- модель позволяет на этапе проектирования ПС принимать оптимальные проектные решения, опираясь на характеристики ошибок, оцениваемые с помощью имитационной модели;
- модель позволяет прогнозировать требуемые ресурсы тестирования;

- модель дает возможность определить меру сложности программ и предсказать возможное число ошибок и т.д.

К недостаткам можно отнести высокую стоимость метода, так как он требует дополнительных затрат на составление имитационной модели, и приблизительный характер получаемых показателей.

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:

1) фиксировать все ошибки в одном выбранном модуле и устранить все побочные эффекты, вызванные изменениями этого модуля, отрабатывая таким образом последовательно все модули;

2) 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.

4.4. Дефекты и ошибки в комплексах программ

Для эффективной организации процессов разработки ПО руководителям и специалистам необходимо знать и учитывать основные источники и типы дефектов и ошибок, возможные в сложных комплексах программ, которые следует изучать, прогнозировать и устранять при производстве.

Понятие дефекта или ошибки в программе подразумевает неправильность, погрешность или неумышленное искажение объекта или процесса, что может быть причиной ущерба – риска при функционировании и применении программного продукта.

Обычно, когда мы говорим «дефект», мы подразумеваем «сбой». Однако различные стандарты могут предполагать различное смысловое наполнение этих терминов.

Ошибка (error) – отличие между корректным результатом и вычисленным результатом, полученным с использованием программного обеспечения;

Недостаток (fault) – некорректный шаг, процесс или определение данных в компьютерной программе;

Сбой (failure) – некорректный результат, полученный в результате недостатка;

Дефект (defect) – результат сбоя программного обеспечения.

Человеческая/пользовательская ошибка (mistake) – действие человека, приведшее к некорректному результату.

При этом должно быть известно или задано требование или правильное, эталонное состояние объекта или процесса, по отношению к которому может быть определено наличие отклонения – ошибка или дефект. Исходным эталоном обычно является спецификация требований заказчика или потенциального пользователя, предъявляемая к программному компоненту или комплексу. Любое отклонение результатов функционирования программы от предъявляемых к ней требований и сформированных по ним эталонов – тестов, следует квалифицировать как ошибку – дефект в программе, наносящий некоторый ущерб при ее применении. Различие между ожидаемыми и полученными результатами функционирования комплекса программ могут быть следствием ошибок не только в созданных программах, но и ошибок в первичных требованиях спецификаций, явившихся базой при создании эталонов. Тем самым проявляется объективная реальность, заключающаяся в невозможности абсолютной корректности и полноты исходных требований и эталонов для программных компонентов и комплексов.

Источниками ошибок в комплексах программ являются специалисты – конкретные люди с их индивидуальными особенностями, квалификацией, талантом и опытом.

Статистика ошибок и дефектов в компонентах и комплексах программ и их характеристики в сложных заказных продуктах могут служить ориентирами для разработчиков при распределении ресурсов в жизненном цикле комплексов программ и предохранять их от излишнего оптимизма при оценке достигнутого качества программных продуктов. Изучение и прогнозирование характеристик дефектов и ошибок в программах непосредственно связаны с достигаемой корректностью, безопасностью и надежностью функционирования комплексов программ.

На практике исходные требования – эталоны поэтапно уточняются, модифицируются, расширяются и детализируются по согласованию между заказчиком и разработчиками. Базой таких уточнений являются неформализованные представления и знания специалистов – заказчиков и разработчиков, а также результаты промежуточных этапов проектирования и тестирования. Однако установить некорректность таких эталонов еще труднее, чем обнаружить дефекты в программах, так как принципиально отсутствуют точные, формализованные данные, которые можно использовать как исходные эталоны. В процессе декомпозиции и верификации исходной спецификации требований возможно появление ошибок в спецификациях на компоненты программ и на отдельные модули. Это способствует расширению спектра возможных дефектов и вызывает необходимость создания гаммы методов и средств тестирования для выявления некорректностей в спецификациях на компоненты разных уровней.

Важной особенностью процесса выявления ошибок в программах обычно является отсутствие полностью определенной программы-эталона, которой должны соответствовать текст и результаты функционирования разрабатываемой программы. Поэтому установить наличие и локализовать дефект непосредственным сравнением с программой без ошибок в большинстве случаев невозможно.

Ошибки можно разделить на первичные и вторичные.

Первичные ошибки – это искажения в тексте программ.

Вторичные ошибки – это искажение выходных результатов исполнения программы.

При тестировании обычно сначала обнаруживаются вторичные ошибки и риски, т.е. последствия и результаты проявления некоторых внутренних дефектов или некорректностей программ. Эти внутренние дефекты следует

квалифицировать как первичные ошибки или причины обнаруженных аномалий результатов. Последующая локализация и корректировка таких первичных ошибок должна приводить к устранению ошибок, первоначально обнаруживаемых в результатах функционирования программ (рис. 24).

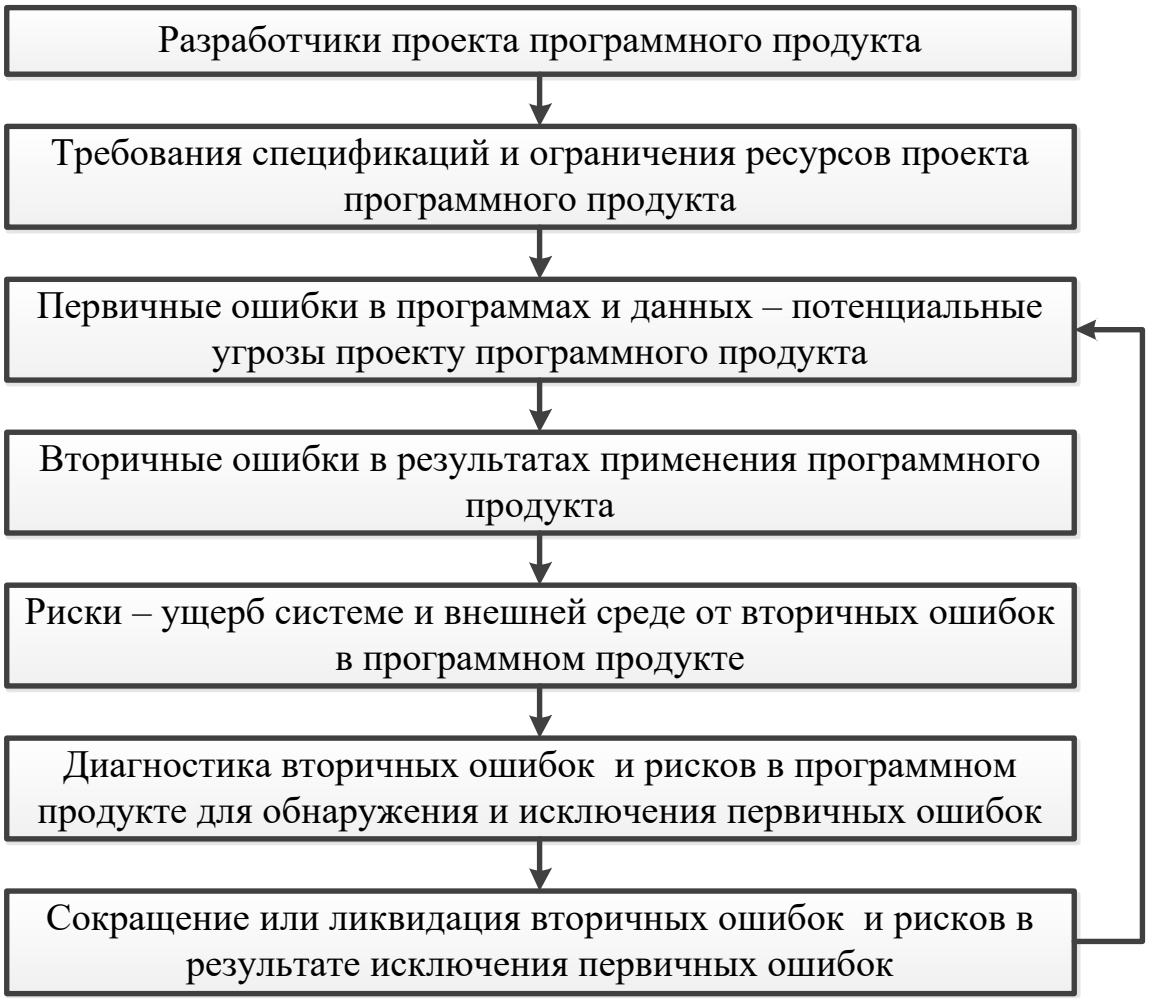


Рис. 24. Процесс выявления ошибок в программах

При производстве наибольшее число первичных ошибок вносится на этапах системного анализа, программирования, разработки или модификаций текстов программ. При этом на долю системного анализа приходятся наиболее сложные для обнаружения и устранения дефекты. Общие тенденции состоят в быстром росте затрат на выполнение каждого устранения ошибки на последовательных этапах процессов разработки компонентов и комплекса программ. При системном анализе интенсивность обнаружения ошибок относительно невелика, и ее трудно выделить из процесса проектирования компонента или комплекса программ. Интенсивность проявления и обнаружения вторичных ошибок наиболее велика на этапе активного тестирования и автономной отладки программных компонентов. Различия интенсивностей устранения первичных ошибок, на

основе их вторичных проявлений, и внесения первичных ошибок при корректировках программ определяют скорость достижения заданного качества компонентов и комплексов программ.

Потери эффективности и риски программ за счет неполной корректности в первом приближении можно считать прямо пропорциональными (с некоторым коэффициентом) вторичным ошибкам в выходных результатах.

Типичным является случай, когда одинаковые по величине и виду вторичные ошибки в различных результирующих данных существенно различаются по своему воздействию на общую эффективность и риски применения комплекса программ. Это влияние вторичных ошибок, в лучшем случае, можно оценить методами экспертного анализа при условии предварительной, четкой классификации видов возможных первичных ошибок в программах и выходных величин. Таким образом, оценка последствий, отражающихся на вторичных ошибках и функционировании программ, может, в принципе, производиться по значениям ущерба – риска вследствие неустраненных их причин.

Уровень серьезности последствий ошибок варьирует от классов проектов и от предприятия, но, в общем, можно разделить ошибки на три уровня.

1. **Небольшими ошибками** называют такие, на которые средний пользователь не обратит внимания при применении ПС вследствие отсутствия их проявления, и последствия которых обычно так и не обнаруживаются. Небольшие ошибки могут включать орфографические ошибки на экране, пропущенные разделы в справочнике и другие мелкие проблемы. Такие ошибки никогда не помешают выпуску и применению версии системы и программного продукта. По десятибалльной шкале рисков небольшие ошибки находятся в пределах от 1 до 3-го приоритета.

2. **Умеренными ошибками** называют те, которые влияют на конечного пользователя, но имеются слабые последствия или обходные пути, позволяющие сохранить достаточную функциональность ПС. Это такие дефекты, как неверные ссылки на страницах, ошибочный текст на экране и даже сбои, если эти сбои трудно воспроизвести и они не оказывают влияния на существенное число пользователей. Некоторые умеренные ошибки, возможно, проникают в конечный программный продукт. Ошибки, которые можно исправить на этом уровне, следует исправлять, если на это есть время и возможность. По десятибалльной шкале умеренные ошибки находятся в диапазоне от 4 до 7-го приоритета.

3. **Критические ошибки** останавливают выпуск версии программного продукта. Это могут быть ошибки с высоким влиянием, которые вызывают сбой в системе или потерю данных, отражаются на надежности и

безопасности применения ПС. Комплекс программ с такими ошибками никогда не передается пользователю. По десятибалльной шкале критические ошибки находятся в диапазоне от 8 до 10-го приоритета.

Совокупность ошибок, дефектов и последствий модификаций проектов крупномасштабных комплексов программ можно упорядочить и условно представить в виде перевернутой пирамиды в зависимости от потенциальной опасности и возможной величины корректировок их последствий (рис. 25).

Рис. 25. Типы дефектов, ошибок и модификаций при сопровождении программных продуктов

В верхней части перечня расположены модификации, дефекты и ошибки, последствия которых обычно требуют наибольших затрат ресурсов для реализации изменений, и они постепенно сокращаются при снижении по перечню. Такое представление величины типов корректировок программ и данных полезно использовать как ориентир для учета необходимых ресурсов

при разработке и сопровождении программных средств, однако оно может содержать значительные отклонения при упорядочении статистических данных реальных проектов.

Каждому типу ошибок соответствует более или менее определенная категория специалистов, являющихся их источником (табл. 5). Такую корреляцию целесообразно рассматривать и учитывать как общую качественную тенденцию при анализе и поиске причин ошибок и дефектов.

Таблица 5 – Категории специалистов, являющихся источником ошибок

Специалисты - источники дефектов и ошибок	Типы первичных дефектов и ошибок программного средства и документации
Заказчики проекта	Дефекты организации проекта и исходных требований заказчика
Менеджер проекта	Дефекты, обусловленные реальной сложностью проекта
Менеджер-архитектор комплекса программ	Ошибки планирования и системного проектирования программного средства
Проблемно-ориентированные аналитики и системные архитекторы	Системные и алгоритмические дефекты и ошибки проекта
Спецификаторы компонентов проекта	Алгоритмические ошибки компонентов и документов программного средства
Разработчики программных компонентов – программисты	Программные дефекты и ошибки компонентов и документов программного средства
Системные интеграторы	Системные ошибки и дефекты реализации версий программного средства и документации
Тестировщики	Программные и алгоритмические ошибки программного средства и документации
Управляющие сопровождением и конфигурацией, инструкторы интерфейсов	Ошибки проектирования и реализации версий программного продукта
Документаторы	Дефекты и ошибки обобщающих документов

Практический опыт показал, что наиболее существенными факторами, влияющими на характеристики обнаруживаемых ошибок, являются:

- методология, технология и уровень автоматизации системного и структурного проектирования компонентов и комплекса программ, а также непосредственного программирования компонентов;

- длительность с начала процесса тестирования компонентов и комплекса и текущий этап производства или сопровождения комплекса программ;

- класс комплекса программ, масштаб (размер) и типы компонентов, в которых обнаруживаются ошибки;

- методы, виды и уровень автоматизации верификации и тестирования, их адекватность характеристикам компонентов и потенциально возможным в программах ошибкам;

- виды и достоверность эталонов – тестов, которые используются для обнаружения ошибок.

4.5. Характеристики первичных ошибок программ

Первичные ошибки в программных продуктах можно разделить на следующие группы (рис. 26):

Технологические ошибки документации и фиксирования программ в памяти ЭВМ составляют 5...10 % от общего числа ошибок, обнаруживаемых при отладке. Большинство технологических ошибок выявляется автоматически формализованными методами.

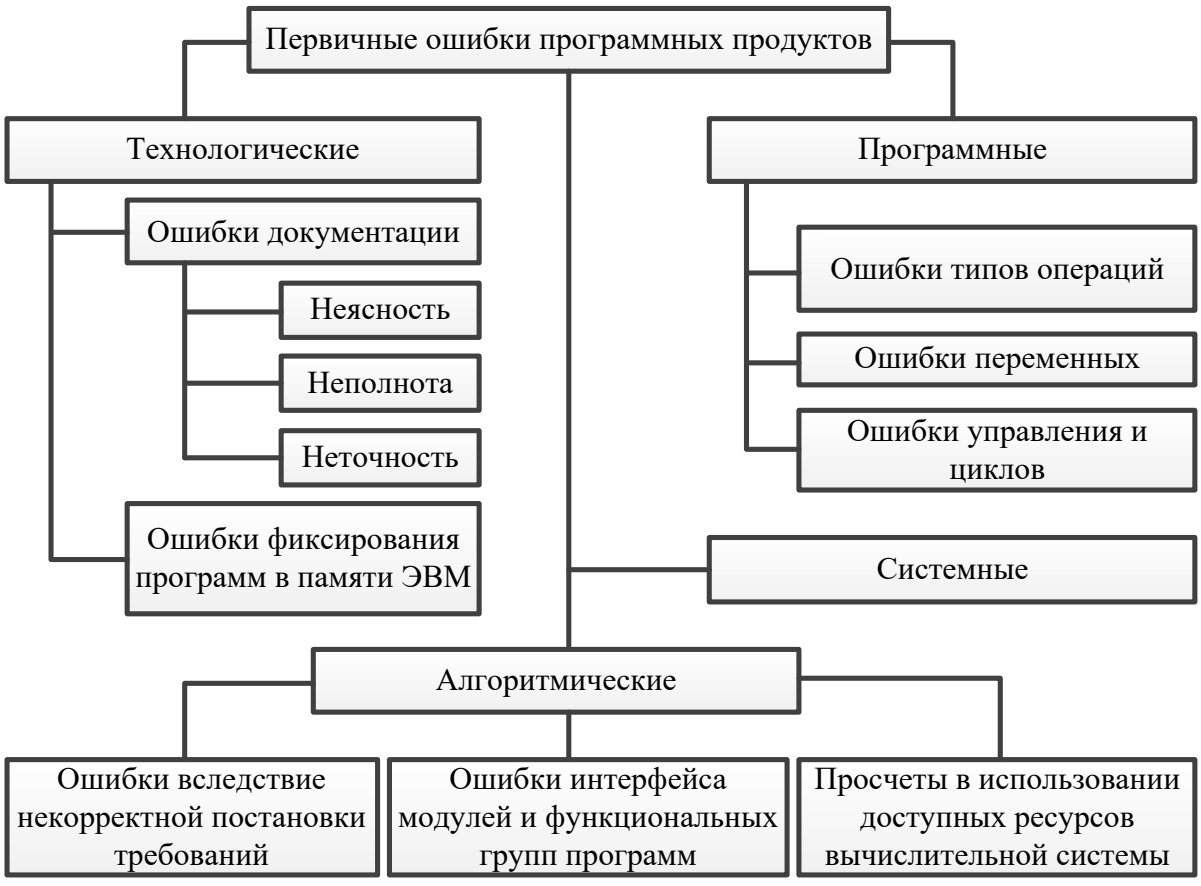


Рис. 26. Классификация первичных ошибок

Ошибки в документации состоят в том, что система делает что-то одним образом, а документация отражает сценарий, что она должна работать иначе. Во многих случаях права должна быть документация, поскольку она написана на основе оригинальной спецификации требований системы. Иногда документация пишется и включает допущения и комментарии о том, как, по мнению авторов документации, система должна работать. В других случаях ошибку можно проследить не до кода, а до документации конечных пользователей, внутренних технологических документов, характеризующих систему, и даже до экранных подсказок и файлов помощи. Ошибки документации можно разделить на три категории:

- неясность;
- неполнота;
- неточность.

Неясность – это когда пользователю не дается достаточно информации, чтобы определить, как сделать процедуру должным образом. Неполная документация оставляет пользователя без информации о том, как правильно реализовать и завершить задачу. Пользователь считает, что задача выполнена, хотя на самом деле это не так. Такие ошибки ведут к тому, что пользователь не удовлетворен версией программного средства, даже если программа в действительности может сделать все, что хочет пользователь. Неточная документация – это худший вид ошибок документации. Такие ошибки часто возникают, когда при сопровождении в систему позже вносятся изменения и об этих изменениях не сообщают лицу, пишущему документацию.

При ручной подготовке текстов машинных носителей при однократном фиксировании исходные данные имеют вероятность искажения около 10^{-3}-10^{-4} на символ. Дублированной подготовкой и логическим контролем вероятность технологической ошибки может быть снижена до уровня 10^{-5}- 10^{-7} на символ. Непосредственное участие человека в подготовке данных для ввода в ЭВМ и при анализе результатов функционирования программ по данным на дисплеях определяет в значительной степени их уровень достоверности и не позволяет полностью пренебрегать этим типом ошибок в программах.

Программные ошибки в первую очередь определяются степенью автоматизации программирования и глубиной формализованного контроля текстов программ. Количество программных ошибок зависит от квалификации программистов, от общего размера комплекса программ, от глубины информационного взаимодействия модулей и от ряда других факторов.

Программные ошибки можно классифицировать по видам используемых операций на следующие крупные группы: ошибки типов операций; ошибки переменных; ошибки управления и циклов. В логических компонентах программных средств эти виды ошибок близки по удельному весу, однако для автоматизации их обнаружения применяются различные методы.

На начальных этапах разработки и автономной отладки модулей программные ошибки составляют около 1/3 всех ошибок. Ошибки применения операций на начальных этапах разработки достигают 14%, а затем быстро убывают при повышении квалификации программистов. Ошибки в переменных составляют около 13%, а ошибки управления и организации циклов – около10%. Каждая программная ошибка влечет за собой необходимость изменения около 10 команд, что существенно меньше, чем при алгоритмических и системных ошибках. На этапах комплексной отладки и эксплуатации удельный вес программных ошибок падает и составляет около 15 и 3% соответственно от общего количества ошибок, выявляемых в единицу времени.

Ошибки реализации спецификаций компонентов наиболее обычны и, в общем, наиболее легки для исправления в системе, что не делает проблему легче для программистов. В отличие от ошибок требований и структурных ошибок, которые обычно специфичны для приложения, программисты часто совершают при кодировании одни и те же виды ошибок.

Первую категорию составляют дефекты, которые приводят к отображению для пользователя сообщений об ошибках при точном следовании порядку выполнения требуемых функций. Хотя эти сообщения могут быть вполне законны, пользователи могут посчитать это ошибкой, поскольку они делали все правильно и, тем не менее, получили сообщение об ошибке. Часто ошибки этого типа вызваны либо проблемами с ресурсами, либо специфическими зависимостями от данных.

Вторая категория таких ошибок может содержать ошибки, связанные с дефектами в графическом интерфейсе пользователя. Такие ошибки могут являться либо нестандартными модификациями пользовательского интерфейса, которые приводят к тому, что пользователь совершает неверные действия, либо они могут быть стандартными компонентами пользовательского интерфейса, используемыми иначе, чем ожидает конечный пользователь.

Третья категория может содержать пропущенные на стадии реализации функции, что всегда считается ошибкой, возможно, с большим риском. Многие тестировщики и пользователи бета-версий сообщают об ошибках, которые на самом деле являются желательными улучшениями. В данном

случае можно не замечать обнаруженные таким образом отсутствия функций, которых не было в спецификациях.

Алгоритмические ошибки программ трудно поддаются обнаружению методами статического автоматического контроля. Трудность их обнаружения и локализации определяется, прежде всего, отсутствием для многих логических программ строго формализованной постановки задачи, полной и точной спецификации, которую можно использовать в качестве эталона для сравнения результатов функционирования программ. К алгоритмическим ошибкам следует отнести, прежде всего, ошибки, обусловленные некорректной постановкой требований к функциональным задачам, когда в спецификациях не полностью оговорены все условия, необходимые для получения правильного результата. Эти условия формируются и уточняются в основном в процессе тестирования и выявления ошибок в результатах функционирования программ. Ошибки, обусловленные неполным учетом всех условий решения задач, являются наиболее частыми в этой группе и составляют до 50-70% всех алгоритмических ошибок или около 30 % общего количества ошибок на начальных этапах проектирования.

К алгоритмическим ошибкам следует отнести также ошибки интерфейса модулей и функциональных групп программ, когда информация, необходимая для функционирования некоторой части программы, оказывается не полностью подготовленной программами, предшествующими по времени включения, или неправильно передаются информация и управление между взаимодействующими модулями. Этот вид ошибок составляет около 6-8% от общего количества, и их можно квалифицировать как ошибки некорректной постановки задач. Алгоритмические ошибки проявляются в неполном учете диапазонов изменения переменных, в неправильной оценке точности используемых и получаемых величин, в неправильном учете корреляции между различными переменными, в неадекватном представлении формализованных условий решения задачи в виде частных спецификаций или блок-схем, подлежащих программированию. Эти обстоятельства являются причиной того, что для исправления каждой алгоритмической ошибки приходится изменять в среднем около 20 команд (строк текста), т.е. существенно больше, чем при программных ошибках.

Особую, весьма существенную, часть алгоритмических ошибок в системах реального времени, при сопровождении составляют просчеты в использовании доступных ресурсов вычислительной системы. Получающиеся при модификации программ попытки превышения использования выделенных ресурсов следует квалифицировать как ошибку,

так как затем всегда следует корректировка с целью удовлетворения имеющимся ограничениям. Одновременная разработка множества модулей различными специалистами затрудняет оптимальное и сбалансированное распределение ограниченных ресурсов ЭВМ по всем задачам, так как отсутствуют достоверные данные потребных ресурсов для решения каждой из них. В результате возникает либо недостаточное использование, либо, в подавляющем большинстве случаев, нехватка каких-то ресурсов ЭВМ для решения задач в первоначальном варианте. Наиболее крупные просчеты обычно допускаются при оценке времени реализации различных групп программ реального времени и при распределении производительности ЭВМ. Алгоритмические ошибки этого типа обусловлены технической сложностью расчета времени реализации программ и сравнительно невысокой достоверностью определения вероятности различных маршрутов обработки информации.

Системные ошибки в программных продуктах определяются, прежде всего, неполной информацией о реальных процессах, происходящих в источниках и потребителях информации. Кроме того, эти процессы зачастую зависят от самих алгоритмов и поэтому не могут быть достаточно определены и описаны заранее без исследования изменений функционирования программного продукта во взаимодействии с внешней средой. На начальных этапах не всегда удается точно и полно сформулировать целевую задачу всей системы, а также целевые задачи основных групп программ, и эти задачи уточняются в процессе проектирования. В соответствии с этим уточняются и конкретизируются спецификации на отдельные компоненты и выявляются отклонения от уточненного задания, которые могут квалифицироваться как системные ошибки.

Ошибки определения характеристик системы и внешней среды, принятых в процессе производства комплекса программ за исходные, могут быть результатом аналитических расчетов, моделирования или исследования аналогичных систем. В ряде случаев может отсутствовать полная адекватность предполагаемых и реальных характеристик, что является причиной сложных и трудно обнаруживаемых системных ошибок и дефектов развития проекта.

При автономной и в начале комплексной отладки версий программных продуктов относительная доля системных ошибок может быть невелика (около 10%), но она существенно возрастает (до 35-40%) на завершающих этапах комплексной отладки новых базовых версий программного продукта. В процессе сопровождения системные ошибки являются преобладающими

(около 60-80% от всех ошибок). Следует также отметить большое количество команд, корректируемых при исправлении каждой такой ошибки (около 20-50 команд на одну ошибку).

Одной из основных причин ошибок в сложных комплексах программ являются организационные дефекты создания требований и эталонов к программному продукту, которые отличаются от остальных типов. Ошибки и дефекты этого типа появляются из-за недостаточного понимания руководителями и коллективом специалистов целей и функций комплекса программ, а также вследствие отсутствия четкой их организации и поэтапного контроля требований качества компонентов и продуктов. Это порождается пренебрежением руководителей к организации всего технологического процесса формализации требований сложных программных продуктов и приводит к серьезной недооценке их дефектов, а также к трудоемкости и сложности их выявления. В условиях отсутствия при производстве планомерной и методичной разработки и тестирования требований и эталонов может оставаться не выявленным значительное количество ошибок, и прежде всего дефекты требований к взаимодействию отдельных функциональных компонентов между собой и с внешней средой. Для сокращения этого типа массовых ошибок активную роль должны играть лидеры – менеджеры и аналитики – системотехники, способные вести контроль и конфигурационное управление требованиями, изменениями и развитием версий и компонентов комплексов программ.

Сложность обнаружения и устранения ошибок значительно конкретизируется и становится измеримой, когда устанавливается связь этого понятия с конкретными ресурсами, необходимыми для решения соответствующей задачи и возможными проявлениями дефектов. При разработке и сопровождении программ основным лимитирующим ресурсом обычно являются допустимые трудозатраты специалистов, а также ограничения на сроки разработки, технологию проектирования корректировок комплекса. К факторам, влияющим на сложность обнаружи-ваемых ошибок комплексов программ, относятся:

- величина – размер создаваемой или модифицируемой программы, выраженная числом строк текста, функциональных точек или количеством программных компонентов в комплексе;
- количество обрабатываемых переменных или размер и структура памяти, используемой для размещения базы данных корректировок;
- трудоемкость разработки изменений компонентов и комплекса программ;
- длительность разработки и реализации корректировок;

- число специалистов, участвующих в производстве компонентов и комплекса программ.

Убывание ошибок в компонентах и комплексе программ и интенсивности их обнаружения в процессе производства не беспредельно. После тестирования в течение некоторого времени интенсивность обнаружения дефектов при самых жестких внешних условиях испытаний снижается настолько, что коллектив, ведущий разработку и тестирование, попадает в зону нечувствительности к ошибкам и возможным отказам функционирования. При такой интенсивности отказов вследствие их редкого проявления трудно прогнозировать затраты времени, необходимые для обнаружения очередной ошибки или дефекта. Создается представление о полном отсутствии случайных проявлений дефектов, о невозможности и бесцельности их поиска, поэтому усилия на тестирование сокращаются, и интенсивность обнаружения ошибок еще больше снижается. Этой предельной интенсивности обнаружения отказов соответствует наработка на обнаруженную ошибку, при которой прекращается улучшение характеристик программного комплекса на этапах тестирования или испытаний.

При серийном выпуске программного продукта, благодаря значительному расширению вариантов исходных данных и условий эксплуатации, возможно в течение некоторого времени возрастание суммарной (по всем экземплярам системы) интенсивности обнаружения дефектов и ошибок. Это позволяет дополнительно устранять ряд дефектов и тем самым увеличивать длительность между проявлениями ошибок в процессе эксплуатации.

4.6. Основные свойства вторичных ошибок

В первом приближении величину вторичной ошибки, появившейся из-за пропущенных при отладке первичных ошибок, можно оценить по формуле

$$b_j(\tau) = \sum_{k=1}^{m} P_k^1(\tau)\Delta k_j ,$$

где b_j – величина вторичной ошибки в j-м результате решения задачи; τ – длительность отладки; $P_k^1(\tau)$ – вероятность наличия в программе первичной ошибки k-го типа; Δk_j – дополнительная ошибка, вносимая k-й первичной ошибкой в результирующую переменную j; m – количество типов не выявленных первичных ошибок.

Потери эффективности и риски программ за счет неполной корректности в первом приближении можно считать пропорциональными вторичным ошибкам в выходных результатах. Типичным является случай, когда

одинаковые по величине и виду вторичные ошибки в различных результирующих данных существенно различаются по своему воздействию на общую эффективность и риски применения комплекса программ. Это воздействие можно учесть некоторым условным весом χ_j, который позволяет взвешивать последствия ошибок.

Величину общей средневзвешенной ошибки функционирования системы вследствие не выявленных первичных ошибок можно оценить по формуле

$$b(\tau) = \sum_{j=1}^{q} \chi_j b_j(\tau) = \sum_{j=1}^{q} \chi_j \sum_{k=1}^{m} P_k^1(\tau) \Delta k_j .$$

Таким образом, оценка последствий, отражающихся на вторичных ошибках и функционировании программ, может, в принципе, производиться по значениям ущерба – риска вследствие не устраненных их причин – первичных ошибок в программе. Вторичные ошибки являются определяющими для эффективности функционирования программ, однако не каждая первичная ошибка вносит заметный вклад в выходные результаты. Вследствие этого ряд первичных ошибок может оставаться не обнаруженным и, по существу, не влияет на функциональные характеристики компонента или комплекса программ.

Вторичная ошибка может проявляться как отказ – потеря работоспособности системы на длительное время. Значительное искажение программы, данных или вычислительного процесса может вызвать отказовую ситуацию, которая или превращается в отказ, или может быть быстро исправлена.

Для математического описания типов вторичных ошибок используются обобщенные характеристики ошибок, описывающие основные закономерности изменения суммарного количества вторичных ошибок в программах.

Математические модели позволяют оценивать характеристики ошибок в программах и прогнозировать их надёжность при проектировании и эксплуатации. Модели имеют вероятностный характер, и достоверность прогнозов зависит от точности исходных данных и глубины прогнозирования по времени. Эти математические модели предназначены для оценивания:

- показателей надёжности комплексов программ в процессе отладки;
- количества ошибок, оставшихся не выявленными;
- времени, необходимого для обнаружения следующей ошибки в функционирующей программе;
- времени, необходимого для выявления всех ошибок с заданной вероятностью.

Использование моделей позволяет эффективно проводить отладку и испытания комплексов программ, помогает принять рациональное решение о времени прекращения отладочных работ.

Точное определение полного числа не выявленных ошибок в программном обеспечении прямыми методами измерения невозможно, поскольку в противном случае их можно было бы все зафиксировать и устранить. Однако имеются косвенные пути для приближенной статистической оценки их полного числа или вероятности ошибки в каждой команде программы. Такие оценки базируются на построении математических моделей в предположении о жесткой корреляции между общим числом ошибок и их проявлениями в некотором ПО после его отладки в течение времени t, т.е. между следующими параметрами:

- суммарным числом первичных ошибок в ПО (N_0) или вероятностью ошибки в каждой команде программы (P_0):

- числом вторичных ошибок, выявляемых в единицу времени в процессе тестирования и отладки при постоянных усилиях на ее проведение (dn/dt);

- интенсивностью искажений результатов в единицу времени (λ) на выходе программы (вследствие невыявленных первичных ошибок) при функционировании системы в типовых условиях.

В результате может быть построена экспоненциальная математическая модель распределения ошибок в программах и установлена связь между интенсивностью обнаружения вторичных ошибок при тестировании dn/dt, интенсивностью λ проявления ошибок при нормальном функционировании ПО и числом выявленных первичных ошибок n. При этом учитываются все виды ошибок независимо от источников их происхождения (технологические, программные, алгоритмические, системные).

4.7. Модели распределения ошибок в программах

В настоящее время для оценивания распределения ошибок предложен ряд математических моделей, основными из которых являются:

- экспоненциальная модель распределения ошибок в зависимости от времени отладки;

- модель, учитывающая дискретно понижающуюся частоту появления ошибок как линейную функцию времени тестирования и испытаний;

- модель, базирующаяся на распределении Вейбулла.

При обосновании математических моделей выдвигаются некоторые гипотезы о характере проявления ошибок в комплексе программ. Наиболее

обоснованными представляются предположения, на которых базируется первая экспоненциальная модель распределения ошибок в процессе отладки и которые заключаются в следующем:

1. Любые ошибки в программе являются независимыми и проявляются в случайные моменты времени.

2. Время работы между ошибками определяется средним временем выполнения команды на данной ЭВМ и средним числом команд, исполняемым между ошибками. Это означает, что интенсивность проявления ошибок при реальном функционировании программы зависит от среднего быстродействия ЭВМ.

3. Выбор отладочных тестов должен быть представительным и случайным, с тем чтобы исключить концентрацию необнаруженных ошибок для некоторых реальных условий функционирования программы.

4. Ошибка, являющаяся причиной искажения результатов, фиксируется и исправляется после завершения тестирования, либо вообще не обнаруживается.

Из этих свойств следует, что при нормальных условиях эксплуатации количество ошибок, проявляющихся в некотором интервале времени, распределено по закону Пуассона. В результате длительность непрерывной работы между искажениями распределена экспоненциально.

Предположим, что в начале отладки комплекса программ при $\tau = 0$ в нём содержалось N_0 ошибок. После отладки в течение времени τ осталось n_0 ошибок и устранено n ошибок ($n_0 + n = N_0$). При этом время τ соответствует длительности исполнения программ на вычислительной системе для обнаружения ошибок и не учитывает простои машины, необходимые для анализа результатов и проведения корректировок.

Интенсивность обнаружения ошибок в программе $dn/d\tau$ и абсолютное количество устранённых ошибок связываются уравнением

$$\frac{dn}{d\tau} + kn = kN_0,$$

где k — коэффициент изменения темпа проявления искажений при переходе от функционирования программ на специальных тестах к функционированию при нормальных исходных данных.

Считается, что значение интервалов времени между проявлениями ошибок, которые исправляются, изменяется по экспоненциальному закону. Если предположить, что в начале отладки при $\tau = 0$ отсутствуют обнаруженные ошибки, то решение этого уравнения имеет вид

$$n = N_0\left[1 - \exp\left(-k\tau\right)\right].$$

Количество оставшихся ошибок в комплексе программ

$$n_0 = N_0 - n = N_0 \exp(-k\tau)$$

пропорционально интенсивности обнаружения $dn/d\tau$ с точностью до коэффициента k.

Время безотказной работы программ до отказа T или наработка на отказ, который рассматривается как обнаруживаемое искажение программ, данных или вычислительного процесса, нарушающее работоспособность, равно величине, обратной интенсивности обнаружения отказов (ошибок):

$$T = \frac{1}{dn/d\tau} = \frac{1}{kN_0}\exp(k\tau).$$

Если учесть, что до начала тестирования в комплексе программ содержалось N_0 ошибок и этому соответствовала наработка на отказ T_0 , т.е.

$$T_0 = \frac{1}{dn/d\tau_0} = \frac{1}{kN_0};\ k = \frac{1}{T_0 N_0},$$

то функцию наработки на отказ от длительности проверок можно представить в следующем виде:

$$T = T_0 \exp\left(\frac{\tau}{T_0 N_0}\right).$$

Если известны моменты обнаружения ошибок t_i и каждый раз в эти моменты обнаруживается и достоверно устраняется одна ошибка, то, используя метод максимального правдоподобия, можно получить уравнение для определения значения начального количества первичных ошибок N_0 :

$$\sum_{i=1}^{n}\frac{1}{N_0 - (i-1)} = \frac{n\sum_{i=1}^{n} t_i}{N_0\sum_{i=1}^{n} t_i - \sum_{i=1}^{n}(i-1)t_i},$$

а также выражение для расчёта коэффициента пропорциональности

$$K = \frac{n}{N_0\sum_{i=1}^{n} t_i - \sum_{i=1}^{n}(i-1)t_i}.$$

В результате можно рассчитать число оставшихся в программе ошибок и среднюю наработку на отказ $T_{\text{ср}} = \frac{1}{\lambda}$, т.е. получить оценку времени до обнаружения следующей ошибки.

В процессе отладки и испытаний программ для повышения наработки на отказ от T_1 до T_2 необходимо обнаружить и устранить Δn ошибок. Величина Δn определяется соотношением

$$\Delta n = N_0 T_0 \left[\frac{1}{T_1} - \frac{1}{T_2} \right].$$

Выражение для определения затрат времени $\Delta\tau$ на проведение отладки, которые позволяют устранить Δn ошибок и, соответственно, повысить наработку на отказ от значения T_1 до T_2 , имеет вид

$$\Delta\tau = \frac{N_0 T_0}{K} \ln\left(\frac{T_2}{T_1} \right).$$

Вторая модель, учитывающая дискретно понижающуюся частоту появления ошибок как линейную функцию времени тестирования и испытаний, построена на основе гипотезы о том, что частота проявления ошибок (интенсивность отказов) линейно зависит от времени испытания t_i между моментами обнаружения последовательных i-й и (i-1)-й ошибок, т.е.

$$\lambda(t_i) = K\left[N_0 - (i-1) \right] t_i,$$

где N_0 — начальное количество ошибок; K — коэффициент пропорциональности, обеспечивающий равенство единице площади под кривой вероятности обнаружения ошибок.

Для оценки наработки на отказ получается выражение, соответствующее распределению Релея:

$$P(t_i) = \exp\left\{ -K\left[N_0 - (i-1) \right] \frac{t_i^2}{2} \right\},$$

где $P(t_i) = P(T \ge t_i)$.

Отсюда плотность распределения времени наработки на отказ определяется по формуле

$$f(t_i) = K\left[N_0 - (i-1) \right] t_i \exp\left\{ -K\left[N_0 - (i-1) \right] \frac{t_i^2}{2} \right\}.$$

Используя функцию максимального правдоподобия, получим оценки для общего количества ошибок N_0 и коэффициента K:

$$N_0 = \left[\frac{2n}{K} + \sum_{i=1}^{n} (i-1) t_i^2 \right] \frac{1}{\sum_{i=1}^{n} t_i^2};$$

$$K = \left[\sum_{i=1}^{n} \frac{2}{N_0 - (i-1)} \right] \frac{1}{\sum_{i=1}^{n} t_i^2}.$$

Особенностью **третьей модели**, базирующейся на распределении Вейбулла, является учёт ступенчатого характера изменения надёжности при устранении очередной ошибки. В качестве основной функции рассматривается распределение времени наработки на отказ $P(t)$. Если ошибки не устраняются, то интенсивность отказов является постоянной, что приводит к экспоненциальной модели для распределения:

$$P(t) = \exp(-\lambda t).$$

Отсюда плотность распределения времени наработки на отказ T определяется выражением

$$f(t) = \lambda \exp(-\lambda t),$$

где $t > 0, \lambda > 0$ и $\frac{1}{\lambda} = T_{ср}$ – среднее время наработки на отказ.

Для аппроксимации изменения интенсивности отказов от времени при обнаружении и устранении ошибок используется функция следующего вида:

$$\lambda(t) = \lambda \beta t^{\beta-1}.$$

Если $0 < \beta < 1$, то интенсивность отказов снижается по мере отладки или в процессе эксплуатации. При таком виде функции $\lambda(t)$ плотность функции распределения наработки на отказ описывается двухпараметрическим распределением Вейбулла:

$$f(t) = \lambda \beta t^{\beta-1} \exp\left(-\lambda t^{\beta}\right).$$

Распределение Вейбулла достаточно хорошо отражает реальные зависимости при расчёте функции наработки на отказ.

ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ К РАЗДЕЛУ 4

1. Сформулируйте основные понятия надежности программного обеспечения.
2. Охарактеризуйте показатели надежности программного обеспечения.
3. Приведите классификацию моделей надежности программного обеспечения.
4. Опишите модель Шумана.
5. Опишите модель La Padula.
6. Опишите модель Джелинского-Моранды.
7. Опишите модель Миллса.
8. Опишите простую интуитивную модель.
9. Охарактеризуйте дефекты и ошибки в комплексах программ.
10. Приведите классификацию первичных ошибок программ.
11. Опишите основные свойства вторичных ошибок.
12. Охарактеризуйте экспоненциальную модель распределения ошибок.
13. Охарактеризуйте модель, учитывающую дискретно понижающуюся частоту появления ошибок.
14. Охарактеризуйте модель, базирующуюся на распределении Вейбулла.

ЗАКЛЮЧЕНИЕ

Программное обеспечение является важной составляющей многих сфер жизни, повсеместно используется в промышленности, медицине, в образовании (дистанционное образование, открытое образование). От программного обеспечения зависит не только эффективность производственных процессов, но и жизнь людей (медицина, военная, космическая сфера). По этой причине задача обеспечения качества программного обеспечения выходит на одно из первых мест в процессе разработки сложных программных продуктов.

Однородной картины в области контроля качества и действий по его улучшению в связи с разработкой программного обеспечения нет. Качество программного обеспечения является комплексной проблемой и тесно связано с жизненным циклом программного обеспечения и с тестированием.

Сейчас остро стоит задача измерения качества программного обеспечения с целью оперативного воздействия на процесс производства программного продукта.

При построении системы качества могут быть использованы 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.

СПИСОК ЛИТЕРАТУРЫ

1. Андерсон Р. Доказательство правильности программ: Пер.с англ.– М.: Мир, 1982.– 163 с.
2. Антошина И.В., Домрачеев В.Г., Репинская И.В. Средняя тяжесть ошибок – новый показатель надежности программного обеспечения.– М.; Мытищи: – ЦНИТ Московского гос. ун-та леса, 2005.– 3 с.
3. Бейзер Б. Тестирование черного ящика. Технологии функционального тестирования программного обеспечения и систем / Б. Бейзер.– СПб.: Питер, 2004.– 292 с.
4. Благодатских В.А., Волнин В.А., Поскакалов К.Ф. Стандартизация разработки программных средств: Учеб. пособие под ред. О.С.Разумова.– М.: Финансы и статистика, 2003.– 284 с.
5. ГОСТ Р ИСО/МЭК 12207-2010. Информационная технология. Системная и программная инженерия. Процессы жизненного цикла программных средств.– URL: http://docs.cntd.ru/document/gost-r-iso-mek-12207-2010
6. ГОСТ Р ИСО/МЭК 15288-2005. Информационная технология. Системная инженерия. Процессы жизненного цикла систем.– URL: http://www.internet-law.ru/gosts/gost/2011/
7. ГОСТ Р ИСО/МЭК 15504-1-2009 Информационные технологии. Оценка процессов. Часть 1. Концепция и словарь.– URL: http://www.internet-law.ru/gosts/gost/48644/
8. ГОСТ Р ИСО/МЭК 15504-2-2009 Информационная технология. Оценка процесса. Часть 2. Проведение оценки.– URL: http://www.internet-law.ru/gosts/gost/48767/
9. ГОСТ Р ИСО/МЭК 15504-3-2009 Информационная технология. Оценка процесса. Часть 3. Руководство по проведению оценки.– URL: http://www.internet-law.ru/gosts/gost/49052/
10. ГОСТ Р ИСО/МЭК 15504-4-2012 Информационная технология. Оценка процесса. Часть 4. Руководство по применению для улучшения и оценки возможностей процесса.– URL: http://www.internet-law.ru/gosts/gost/56970/
11. ГОСТ Р ИСО/МЭК 15504-5-2016 Информационные технологии. Оценка процессов. Часть 5. Образец модели оценки процессов жизненного цикла

программного обеспечения.– URL: http://www.internet-law.ru/gosts/gost/63092/
12. ГОСТ Р ИСО/МЭК 25010-2015 Информационные технологии. Системная и программная инженерия. Требования и оценка качества систем и программного обеспечения (SQuaRE). Модели качества систем и программных продуктов.– URL: http://docs.cntd.ru/document/1200121069
13. ГОСТ Р ИСО/МЭК 25021-2014 Информационные технологии. Системная и программная инженерия. Требования и оценка качества систем и программного обеспечения (SQuaRE). Элементы показателя качества.– URL: http://standartgost.ru/g/ГОСТ_Р_ИСО/МЭК_25021-2014
14. ГОСТ Р ИСО/МЭК 25040-2014 Информационные технологии. Системная и программная инженерия. Требования и оценка качества систем и программного обеспечения (SQuaRE). Процесс оценки.– URL: http://docs.cntd.ru/document/1200111327
15. ГОСТ 27.002-89 Надежность в технике. Основные понятия. Термины и определения.– М.: ИПК Издательство стандартов, 2002.
16. ГОСТ Р ИСО 9000-2015 Системы менеджмента качества. Основные положения и словарь. – URL: http://internet-law.ru/gosts/gost/52164/
17. ГОСТ Р ИСО 9001-2015 Системы менеджмента качества. Требования.– URL:http://protect.gost.ru/
18. Изосимов А.В., Рыжко А.Л. Метрическая оценка качества программ.– М.: Изд-во МАИ, 1989.– 96 с.
19. Карповский Е.Я., Чижов С.А. Надежность программной продукции.– Киев: Техника, 1990.– 160 с.
20. Кларк Э.М., Гамбург О., Пелед Д. Верификация моделей программ: Model Checking. // Пер. с англ./ Под ред. Р. Смелянского. - М.: МЦНМО, 2002. – 416 с.
21. Липаев В.В. Надежность программных средств.– М.: СИНТЕГ, 1998.– 232 с.
22. Липаев В. В. Тестирование компонентов и комплексов программ. Учебник.– М.: СИНТЕГ, 2010.– 400 с.
23. Липаев В. В. Программная инженерия сложных заказных программных продуктов: Учебное пособие. – М.: МАКС Пресс, 2014. – 312 с.
24. Майерс Г. Надежность программного обеспечения: пер. с англ./ под ред. В.Ш. Кауфмана.– М.: Мир, 1980.– 360 с.
25. Майерс Г. Искусство тестирования программ: пер. с англ.– М.: Финансы и статистика, 1982.– 273 с.
26. Муса Дж.Д. Измерение и обеспечение надежности программных средств // ТИИЭР.– 1980.– Т. 68, № 9.– С. 113-117.

27. Смагин В.А. Основы теории надежности программного обеспечения: учеб. пособие / В.А. Смагин.– СПб.: ВКА имени А.Ф.Можайского, 2009.– 355 с.
28. Хоар Ч. Взаимодействующие последовательные процессы.– М.: Мир, 1989.– 264 с.
29. Холстед М.Х. Начала науки о программах: пер. с англ.– М.: Финансы и статистика, 1981.– 128 с.
30. Floyd R. W. Assigning meanings to programs, Proc. Symp. Appl. Math., 19; in: J.T.Schwartz (ed.), Mathematical Aspects of Computer Science, pp. 19-32, American Mathematical Society, Providence, R.I., 1967.
31. Kazman R., Bass L., Abowd G., Webb M. SAAM: A Method for Analyzing the Properties of Software Architectures. // Proc. Of 16-th International Conference on Software Engineering.– 1994 – P. 81-90.
32. Kazman R., Barbacci M., Klein M., Carriere S. J., Woods S.G. Experience with Performing Architecture Tradeoff Analysis. // Proc. of International Conference of Software Engineering.– May 1999 – P. 54-63.
33. Kazman R., Klein M., Barbacci M., Lipson H., Longstaff T., Carriere S. J. The Architecture Tradeoff Analysis Method. // Proc. of 4-th International Conference on Engineering of Complex Computer Systems. – August 1998.– P. 68-78.
34. McCabe T. A Complexity Measure // IEEE Transactions on Software Engineering.– 1976.– V. SE 2, № 4.– P. 308-320.
35. Moranda P. B., Jelinski J. Final Report of Software Reliability Study. − McDonnell Douglas Astronautic Company. MDC Report № 63921. Dec. 1972.
36. IEEE 1074-1997 Developing a software project life cycle processes.– URL: http://sistemas.unla.edu.ar/sistemas/sls/ls-3-Proyecto-de-software/pdf/IEEE1074.pdf
37. IEEE Std. 1061-1998 IEEE Computer Society: Standard for Software Quality Metrics Methodology, 1998. 20 p.
38. ISO/IEC/IEEE 15288:2015 Systems and software engineering. System life cycle processes.– URL: http: // www.pqm-online.com/assets/files/lib/std/iso_mek_15288-2002(r).pdf
39. Shick C.J., Wolverton R. W. Assessment of Software Reliability // Proc. 11-th Annual Meeting of the German Operation Research Society. Hamburg, Germany, 6-8 Sept., 1972.
40. Shooman M. L. Software engineering: Reliability, Development and Management. − McGraw-Hill, International Book Co, 1983.

41. Vardi M.Y., Wolper P. An Automata-Theoretic Approach to Automatic Program Verification. // In LICS86 Journal of the ACM. – Vol. 20, №1. – P.332-334.