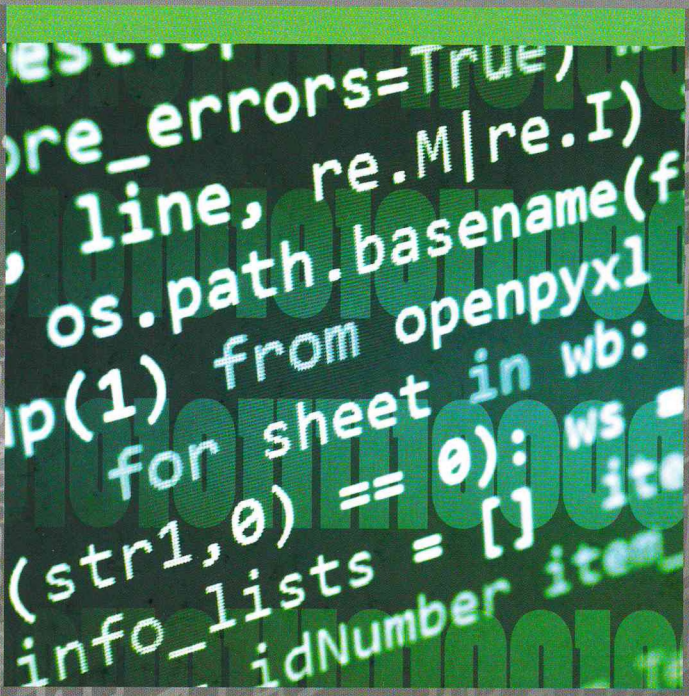
ТОП 50

ПРОФЕССИОНАЛЬНОЕ ОБРАЗОВАНИЕ

Учебник



Е. В. Поколодина, Н. А. Долгова, Д. В. Ананьев

ACADEMIA

РЕВЬЮИРОВАНИЕ ПРОГРАММНЫХ МОДУЛЕЙ

ПРОФЕССИОНАЛЬНОЕ ОБРАЗОВАНИЕ

Е. В. ПОКОЛОДИНА, Н. А. ДОЛГОВА,
Д. В. АНАНЬЕВ

РЕВЬЮИРОВАНИЕ ПРОГРАММНЫХ МОДУЛЕЙ

УЧЕБНИК

Рекомендовано
Федеральным учебно-методическим объединением
в системе среднего профессионального образования
по укрупненной группе профессий и специальностей
09.00.00 «Информатика и вычислительная техника»
в качестве учебника для использования в образовательном процессе
образовательных организаций, реализующих программы
среднего профессионального образования по специальности
09.02.07 «Информационные системы и программирование»

Регистрационный номер рецензии 2
от 13 ноября 2019 г.

ACADEM'A

Москва
Издательский центр «Академия»
2020

УДК 004.05(075.32)
ББК 32.973.202-018.2я723
П485

Рецензенты:
исполнительный директор Центра информационных технологий Республики Татарстан (рецензент ФУМО) *Т. С. Камалетдинова*;
доцент ФГБОУ ВО «Московский авиационный институт (национальный исследовательский университет)» канд. техн. наук *В. М. Новичков*

Поколодина Е. В.

П485 Ревьюирование программных модулей : учебник для студ. учреждений сред. проф. образования / Е. В. Поколодина, Н. А. Долгова, Д. В. Ананьев. — М. : Издательский центр «Академия», 2020. — 208 с.

ISBN 978-5-4468-8609-8

Учебник подготовлен в соответствии с требованиями Федерального государственного образовательного стандарта среднего профессионального образования по специальностям «Информационные системы и программирование» (из списка ТОП-50). Учебное издание предназначено для изучения профессионального модуля «Ревьюирование программных модулей».

Представлено развернутое описание задач и методов моделирования и анализа программных продуктов. Описаны методы организации работы в команде разработчиков, а также механизмы и контроль внесения изменений в код. Рассмотрены вопросы организации ревьюирования и анализ инструментальных средств ревьюирования. Описаны различные современные среды разработки и типовые инструменты и методы анализа программных проектов. Приведены эталоны и методы проверки корректности, а также основные метрики. После каждой главы приведены практические задания, которые помогают закрепить изученный теоретический материал и на практике осуществить ревьюирование кода.

Для студентов учреждений среднего профессионального образования.

УДК 004.05(075.32)
ББК 32.973.202-018.2я723

ISBN 978-5-4468-8609-8

Уважаемый читатель!

Вы держите в руках учебник, который был подготовлен Издательским центром «Академия» в соответствии с Федеральным государственным образовательным стандартом (ФГОС) в рамках реализации комплексного проекта подготовки кадров по 50 наиболее востребованным на рынке труда, новым и перспективным профессиям и специальностям среднего профессионального образования.

Одной из задач проекта является обновление содержания профессионального образования с учетом профессиональных стандартов, современных методик и технологий. При разработке ФГОС также учитывались требования международных конкурсов профессионального мастерства, включая чемпионаты «Молодые профессионалы» (WorldSkills и WorldSkills Russia).

Издательский центр «Академия» является лидером по выпуску учебных материалов для СПО в Российской Федерации. Более двадцати лет наши издания помогают студентам овладевать знаниями, умениями и навыками по рабочим профессиям и специальностям. Стремясь идти в ногу со временем, издательство предлагает не только печатные издания, но и электронные учебники, электронные учебно-методические комплексы и виртуальные практикумы.

Интерактивная форма подачи информации с учетом последних методик и тенденций в преподавании — отличительная особенность и визитная карточка Издательского центра «Академия» на российском рынке.

Мы надеемся, что данный учебник будет полезен студентам, облегчит задачу преподавателей, а также поможет специалистам, которые стремятся расти и развиваться в выбранной ими области, достичь новых профессиональных вершин.

Предисловие

Предлагаемый учебник предназначен для использования в учебном процессе в соответствии с ФГОС СПО по специальности 09.02.07 «Информационные системы и программирование». Учебник охватывает материал профессионального модуля ПМ.03 «Ревьюирование программных продуктов» и может быть использован образовательными учреждениями профессионального образования при реализации учебного процесса указанной специальности как базовой, так и углубленной подготовки. Возможно использование учебника в качестве дополнительной литературы в учебном процессе среднего профессионального и высшего образования укрупненной группы специальностей и направлений 09.00.00 «Информатика и вычислительная техника».

Отдельные разделы учебника могут являться основой для разработки программ переподготовки специалистов в области информационных систем и программирования, а также профессорско-преподавательского состава.

Содержание учебника направлено на формирование следующих профессиональных компетенций:

- осуществлять ревьюирование программного кода в соответствии с технической документацией;
- выполнять измерение характеристик компонент программного продукта для определения соответствия заданным критериям;
- производить исследование созданного программного кода с использованием специализированных программных средств для выявления ошибок и отклонений от алгоритма;
- проводить сравнительный анализ программных продуктов и средств разработки с целью определения наилучшего решения согласно критериям, указанным в техническом задании.

Определяя структуру учебника, его содержание и порядок изложения, авторы опирались на примерную рабочую программу профессионального модуля ПМ.03 «Ревьюирование программных продуктов».

Учебник состоит из трех глав, в основном соответствующих темам примерной рабочей программы:

- задачи и методы моделирования и анализа программных продуктов;
- организация ревьюирования, инструментальные средства ревьюирования;
- менеджмент программного проекта.

В конце глав приведены практические задания для проработки теоретического материала. Следует отметить, что на данный момент теоретические и практические аспекты контроля качества кода и работы с системами контроля версий недостаточно широко представлены в учебной литературе, поэтому рекомендуется изучение интернет-ресурсов и документации, представленной на официальных сайтах компаний — производителей программного обеспечения. Список источников, которые используются в практических работах, представлен в тексте, для получения дополнительной информации также даны ссылки на интернет-ресурсы.

Глава 1

ЗАДАЧИ И МЕТОДЫ МОДЕЛИРОВАНИЯ И АНАЛИЗА ПРОГРАММНЫХ ПРОДУКТОВ

1.1. МЕТОДЫ ОРГАНИЗАЦИИ РАБОТЫ В КОМАНДЕ РАЗРАБОТЧИКОВ. СИСТЕМЫ КОНТРОЛЯ ВЕРСИЙ

1.1.1. Проект

Разработка программного обеспечения (ПО) ведется в рамках проектов. Рассмотрим возможные составляющие проекта.

Как правило, у проекта есть **заказчик**, который формулирует свои требования к содержанию и желаемым результатам проекта.

Бывает, что конкретного заказчика проекта не существует, но есть **потребность рынка** в определенном программном продукте, который будет востребован у потенциальных пользователей.

Какой бы ни был проект, у него существует **бюджет**. При наличии заказчика бюджет определяет он. Затраты на разработку проекта могут быть жестко фиксированными заказчиком или согласовываться с разработчиком, если у заказчика есть определенный финансовый резерв, который может быть выделен на разработку и развитие проекта. В случае отсутствия заказчика бюджет определяется на основе анализа рынка: будет изучена стоимость аналогичных продуктов, составлен прогноз о потенциальных пользователях, которые готовы приобрести разработанный продукт, и определена фиксированная цена за единицу продукта.

Проект имеет **срок реализации**.

В зависимости от бюджета, сроков реализации, содержания и желаемого результата проекта будет формироваться **команда разработчиков**. Для разработки простого проекта бывает достаточно одного разработчика. Чем выше сложность проекта, больше объем работы и меньше срок реализации проекта, тем большее количество работников будет задействовано для его разработки. Формирование штата разработчиков и подбор высококвалифици-

рованного персонала зависят от бюджета проекта и готовности руководства фирмы-разработчика направить определенную часть финансов из бюджета на оплату работы сотрудников.

1.1.2. Команда проекта

Рассмотрим возможные роли членов команды трудоемкого проекта.

Аналитик — специалист, хорошо разбирающийся в предметной области разрабатываемого проекта.

Аналитик исследует проблемы, анализирует потребности заказчиков, планирует, что необходимо сделать при разработке продукта, и доносит эту информацию до команды разработчиков. Аналитик разбирается в запросах клиентов и работе команды программистов в достаточной степени, чтобы написать спецификацию для разработчиков. Также он может исследовать аналогичные продукты конкурентов совместно со специалистом по маркетингу, чтобы способствовать выпуску качественного, конкурентоспособного продукта.

В больших проектах могут присутствовать **бизнес-аналитик**, который занимается анализом бизнес-области и формализует бизнес-процессы данной области, и **системный аналитик**, отвечающий за перевод требований, которые вытекают из бизнес-процессов заказчика, в конкретные функциональные требования к разрабатываемому ПО.

Разработчик архитектуры проекта или команда высококвалифицированных программистов коллегиально разрабатывают архитектуру проекта.

Команда программистов — разработчики ПО (проекта).

В команде могут быть выделены такие роли, как администратор базы данных, который занимается ее проектированием, и проектировщик, который работает на более низком, чем разработчик архитектуры проекта, уровне, проектируя компоненты системы.

Тестировщик — сотрудник, проверяющий корректность работы приложения в соответствии с требованиями и спецификациями для данного программного продукта.

Дизайнер — сотрудник, разрабатывающий внешний вид (интерфейс) программного продукта.

Руководитель (менеджер) проекта несет общую ответственность за проект, контролирует сроки и качество выполнения задач, стимулирует и мотивирует сотрудников, разбивает задачи

на подзадачи, решает ряд важных вопросов, требующих взаимодействия с заказчиком. «Работа руководителя проекта заключается в том, чтобы обеспечить выпуск высококачественного продукта в заданные сроки и при этом уложиться в рамки отпущенного бюджета» [22].

Администратор проекта помогает руководителю решать вопросы функционирования команды.

Специалист по подготовке пользователей разрабатывает учебную документацию о работе с ПО, обучает пользователей.

Сервисный инженер организует установку и сопровождение установленного продукта.

Специалист по маркетингу занимается продвижением продукта на рынке, его развитием и улучшением.

Состав команды проекта зависит от многих факторов. Часть из них мы рассмотрели выше. В зависимости от потребностей проекта будет формироваться и его команда. Представим, что есть заказчик и проект. Что необходимо сделать, чтобы определить состав команды?

Ответы на этот вопрос могут быть различными, но в любом случае необходимо:

- проанализировать потребности заказчика;
- разработать архитектуру и спецификацию для архитектуры проекта;
- сформулировать задачи.

Это позволит понять, какие сотрудники и какие программисты необходимы.

1.1.3. Организация работы в команде разработчиков

Методы организации работы. Методы организации работы в команде разработчиков могут быть совершенно разными и зависят от многих факторов, в частности:

- какие задачи стоят перед командой;
- какие сотрудники входят в команду;
- какова структура команды;
- как происходит распределение обязанностей среди членов команды;
- каким образом взаимодействуют члены команды и какие вопросы требуют совместных решений;
- какие методы руководства и управления выбраны и т. д.

Питер Гудлиф, анализируя ремесло программистов, выделяет несколько типов команд разработчиков и методов организации их работы [9, с. 406—445]. Рассмотрим некоторые из них. (Все возможные методы организации работы команды программистов ими не исчерпываются. Также при организации работы команды может быть задействовано несколько методов.)

Равноправная основа. Есть команда высококвалифицированных специалистов, разбирающихся в нескольких областях разработки и способных выполнить широкий ряд функций. Проект разрабатывается ими на равноправной основе. Каждый специалист отвечает за определенный участок работы, реализуя его полностью: выбор архитектуры, проектирование, разработка, тестирование, создание документации.

Преимущество данного подхода — в хорошей разработке модулей проекта. Сложности в том, что универсальные высококвалифицированные специалисты стоят дорого. Они могут быть компетентны не во всех областях. В этом случае работа над участком, где они менее компетентны, потребует от них большего количества времени и результат может быть не настолько хорошо проработан, как у узконаправленного специалиста.

Высококвалифицированный опытный руководитель. Команда программистов может быть набрана из разработчиков, которые компетентны только в определенных областях, и иметь высокопрофессионального руководителя, разбирающегося во всех областях, в некоторых более, в некоторых менее детально. Руководитель курирует проект в целом и ставит задачи каждому члену команды.

Достоинством работы такой команды будет хорошая проработка отдельных участков проекта. При этом может возникнуть опасность, что при объединении участки будут менее работоспособны или функциональны, так как при разработке определенного модуля было задействовано большое количество специалистов.

Старшие и младшие разработчики. Команда может состоять из групп разработчиков. Более опытные разработчики (старшие) решают более сложные задачи и имеют целостное представление о проекте. Менее опытные и зачастую менее компетентные разработчики (младшие) выполняют более простые задачи, помогая старшим, могут не иметь представления о проекте в целом, занимаясь только отдельными задачами.

Зачастую разработчики ПО участвуют в работе команд разного уровня.

1.1.4. Уровни групповой работы

Рассмотрим работу специалиста в компании по разработке ПО [9, с. 406—408]. Сотрудник создает отдельный программный компонент, входящий в более крупный проект. Его разработку он может вести самостоятельно или в составе группы программистов. Это **первая команда**.

Компонент должен войти в более крупный продукт. Участвующие в его создании (программисты, тестировщики, сервисные инженеры и нетехнический состав, например администрация и служба маркетинга) образуют **вторую команду**.

Кроме того, программист или группа программистов работает в компании, которая зачастую занята несколькими проектами одновременно, и это **третья команда**.

В любой крупной компании, занимающейся разработкой ПО, на практике уровней групповой работы значительно больше.

Уровни групповой работы (рис. 1.1) [9, с. 407]:

- *разработчик*. Разрабатывает ПО. Требуются профессиональные знания и умения, обучаемость, мотивация;

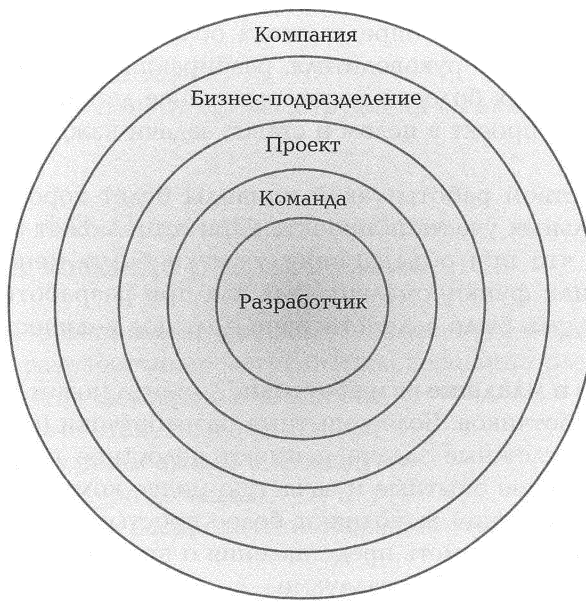


Рис. 1.1. Уровни групповой работы

- *команда*. Организованное взаимодействие между разработчиками. Требуются коммуникативные навыки;
- *проект*. Объединяет несколько команд. Требуется внимательное отношение к обмену данными, планированию и управлению ресурсами;
- *бизнес-подразделение*. Каждому сотруднику необходимо решать задачи бизнес-подразделения. На этом уровне на работу влияют политика компании и корпоративная культура;
- *компания*. На данном уровне идет взаимодействие с другими компаниями, клиентами и поставщиками, возникают бизнес-стратегии, схемы стимулирования рыночной эффективности и эффективной работы сотрудников внутри компании.

На уровнях «разработчик — команда» происходит взаимодействие внутри команды, на уровнях «разработчик — команда — проект» — между командами.

1.1.5. Инструменты команды программистов

Чтобы избежать хаоса и иметь отлаженный механизм разработки, разработчики используют определенный **набор инструментов, помогающих организовать командную работу**:

- *методология разработки* описывает, как будет происходить процесс разработки, устанавливает ответственность каждого сотрудника за различные участки работы и порядок передачи данной ответственности. Каждый разработчик должен иметь представление, что ему необходимо делать и каким образом он будет участвовать в работе команды;
- *планом проекта* определяется последовательность работы, конкретизируется работа каждого участника для каждого этапа разработки, определяются сроки выполнения тех или иных задач различными участниками;
- *средства автоматизации групповой работы* позволяют наладить эффективную коммуникацию между членами команды, особенно в случае, когда они не работают в одном офисе. Эта система доступна группе разработчиков и содержит календарь, адресный справочник, средство планирования совещаний, механизм для совместного доступа и хранения документации. Некоторые из таких систем могут иметь возможность проводить телеконференции, вебинары, делать тематические форумы;
- *система управления версиями*, или *система контроля версий*, — ПО, позволяющее вести совместную разработку кода. Оно дает

возможность проанализировать, кто, чем и когда занимается, управлять модификациями кода, делать откат ошибочных вариантов и получать последнюю версию кода;

- *база данных ошибок*. После разработки кода ошибки в нем могут обнаружить разработчик, его коллеги, тестировщик или отдел контроля качества. Если лица, обнаружившие ошибки, начнут их исправлять одновременно или в разное время, может возникнуть путаница. В связи с этим полезно иметь базу ошибок, где хранится информация об обнаруженных ошибках, приоритетах их исправления. Для каждой ошибки назначают ответственных за ее исправление, обозначают срок исправления. После внесения исправлений об этом делается отметка и сохраняется код с исправленной ошибкой. Такая база должна быть доступна для отделов тестирования и разработки.

Рассмотрим подробнее один из перечисленных инструментов — систему контроля версий.

1.1.6. Системы контроля версий

Система контроля версий (Version Control System, VCS, или Revision Control System) — ПО для облегчения работы с изменяющейся информацией.

Система позволяет хранить несколько версий одного и того же документа, при необходимости возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение, и многое другое [26]. То есть система контроля версий (СКВ) — это система, которая позволяет регистрировать изменения в одном или нескольких файлах проекта. Она предоставляет следующие **возможности**:

- хранение нескольких версий одних и тех же файлов проекта;
- возврат к определенным версиям файлов проекта, сделанным ранее;
- получение последних, актуальных на данный момент версий файлов проекта;
- фиксация, кто и когда сделал изменение;
- синхронизация работы команды по объединению изменений, выполненных разными разработчиками.

Существуют разные типы СКВ.

По **типу хранения информации** и возможностям ее получения выделяют локальные, централизованные и децентрализованные СКВ.

Локальная СКВ имеет базу данных на локальном компьютере пользователя, где хранятся версии разрабатываемого программного продукта (рис. 1.2).

Централизованная СКВ характеризуется наличием центрального сервера, на котором хранятся версии файлов (рис. 1.3). Разработчики и заинтересованные лица могут обращаться с запросами на сервер для получения информации. Основной недостаток — возможность отсутствия доступа к серверу. Еще одним из рисков являются неисправность сервера и возможная потеря информации.

В случае **децентрализованных**, или **распределенных**, **СКВ** в отличие от централизованных разработчики не просто выгружают последние версии файлов, а полностью копируют весь репозиторий. Репозиторий проекта есть у каждого разработчика. При этом можно выделить центральный репозиторий, в который разработчики отправляют изменения своих локальных репозиториев, а последние синхронизируются с центральным (рис. 1.4). Достоинствами подобной СКВ являются более высокая надежность, гибкость и автономность отдельного рабочего места, возможность работать с несколькими удаленными репозиториями, экономия затрат времени при выполнении большинства операций.

По **принципу внесения изменений** выделяют блокирующие и неблокирующие СКВ.

В **блокирующих СКВ** разработчик, изменяющий файл, сообщает системе о желании его отредактировать. Файл открывается в режиме записи только для этого разработчика, а остальным членам

Рис. 1.2. Принцип работы локальной СКВ

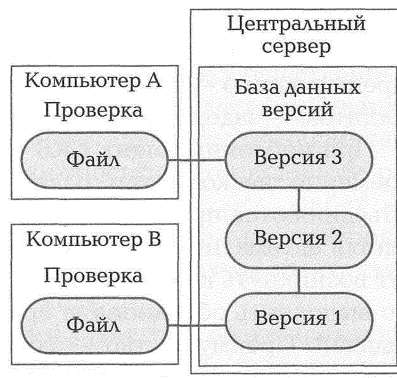


Рис. 1.3. Принцип работы централизованной СКВ

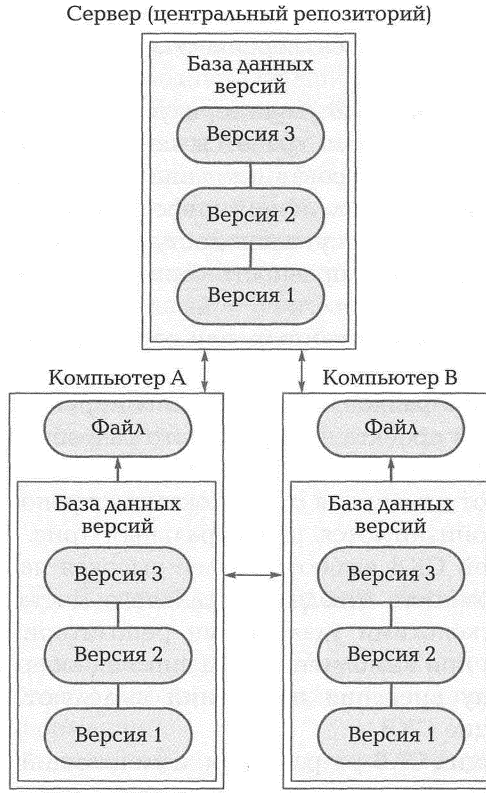


Рис. 1.4. Принцип работы децентрализованной СКВ

команды данный файл будет доступен только в режиме чтения до тех пор, пока изменения не будут внесены и блокировка не снята. Преимущество — отсутствие конфликтов при изменении кода; недостаток — задержка работы над тем или иным файлом проекта.

При **неблокирующих СКВ** один файл может изменяться одновременно несколькими разработчиками. Модификации файлов объединяются при внесении их в систему. По умолчанию выполняется автоматическое объединение. Во время этого процесса могут возникнуть конфликты. В этом случае разработчику необходимо выполнить объединение вручную, что может стать трудоемкой задачей. Преимущество — совместная работа над определенным файлом и экономия времени в случае отсутствия конфликтов; недостатки — возможность возникновения конфликтов и необходимость временных затрат на объединение вручную.

Существует большое количество СКВ. Наиболее **популярные СКВ** по версиям интернет-ресурсов следующие (в скобках указан сайт разработчика):

- `GIT` — распределенная СКВ, разработана в 2005 г. (https://git-scm.com/);
- `SVN (Subversion)` — централизованная СКВ, разработана в 2002 г. (http://subversion.apache.org/);
- `Mercurial` — распределенная СКВ, разработана в 2005 г. (http://mercurial-scm.org/);
- `CVS (Concurrent Versions System)` — централизованная СКВ, разработана в 1990 г. (http://nongnu.org/cvs/);
- `Team Foundation Server` — централизованная СКВ, разработана в 2005 г. (https://visualstudio.microsoft.com/tfs/);
- `Bazaar` — распределенная СКВ, разработана в 2004 г. (https://bazaar.canonical.com).

Сравнительная характеристика СКВ представлена в гл. 2.

1.2. ЦЕЛИ, ЗАДАЧИ, ЭТАПЫ, ОБЪЕКТЫ И ПЛАНИРОВАНИЕ РЕВЬЮИРОВАНИЯ

1.2.1. Определение и цель ревьюирования

В энциклопедии мы можем встретить следующее определение: «***Рецензия*** (англ. *review* — обзор) — анализ, разбор, некоторая оценка публикации, произведения или продукта, жанр газетно-журнальной публицистики и литературной критики. Рецензия может относиться к материальным вещам (приборы, аксессуары, бытовая техника), компьютерным технологиям, художественной литературе, музыке, фильмам, компьютерным играм.

Рецензироваться могут также текущие события, общественные заявления и происшествия. В дополнение к критическому утверждению автор рецензии может выставить предмету рецензирования некоторую оценку для указания относительной ценности рецензируемого предмета» [27].

Таким образом, ревьюирование, или, иначе, рецензирование, — это анализ и оценка чего-либо. В нашем случае мы рассмотрим анализ и оценку программного кода.

Code Review (данный термин переводят как ревьюирование, рецензирование, обзор, инспектирование кода) — один из стандартных методов программной инженерии, направленный на проверку кода, его анализ и составление рецензии о проверенном

коде, основной целью которого является повышение качества программного кода.

Цель ревьюирования — повысить качество программного кода. Помимо этого ревьюирование решает ряд важных задач, которые мы сформулируем, рассмотрев методы ревьюирования и этапы официальной инспекции кода группой разработчиков.

1.2.2. Методы ревьюирования кода

Самый лучший способ избежать ошибок — это не делать их, но мы знаем пословицу «Не ошибается тот, кто ничего не делает». Каким бы опытным ни был программист, в его коде могут возникнуть ошибки и недостатки, поэтому необходимы рекомендации, как улучшить ту или иную часть кода.

Одним из наиболее эффективных методов поиска и устранения недостатков кода является его рецензирование. В ходе этого процесса код подвергается проверке и критике, что помогает находить широкий класс ошибок и недоработок, с трудом детектируемых или недетектируемых компьютерной техникой и соответствующим ПО.

При ревьюировании оцениваются различные **аспекты разработанного кода**, в частности:

- общая конструкция кода. Проверяется выбор алгоритмов и внешних интерфейсов;
- конструкции в коде. Его разбиение на классы и функции. Разработчики будут обсуждать, действительно ни необходим тот или иной класс, или какой-либо из них по функциональным возможностям может быть объединен с другим, или, наоборот, не были ли пропущены потенциально важный класс или функция;
- аспекты архитектуры. Выбор клиентских связей и иерархии наследования;
- код в отдельных семантических блоках. Проверяется корректность каждого класса, функции, цикла. Анализируются используемые структуры синтаксиса языка, чтобы код был структурирован эффективным способом для дальнейшей работы с ним;
- отдельные операторы кода. Проверяется их соответствие стандартам, принятым в мировой практике и проекте;
- комментарии и документация. Оцениваются их проработанность и удобство.

Могут подвергаться оценке и другие аспекты работы [28].

Рассмотрим **методы проведения ревьюирования**.

Самостоятельное ревьюирование работы. Разработчик самостоятельно проверяет свой код.

Достоинство: экономия времени, так как для проверки кода не привлекаются другие сотрудники.

Недостатки:

- нет экспертной оценки кода;
- разработчик может не заметить ошибки или возможности для лучшего проектирования.

Ревьюирование кода одним посторонним лицом. Рассмотрим два варианта.

1. Разработчик показывает свой код для критического анализа другому программисту. Тот вместе с ним проверяет логику кода автора и смотрит, нет ли ошибок.

Достоинства:

- метод оценки более объективен в отличие от ревьюироания автором работы;
- могут быть даны советы стороннего разработчика по доработке кода и исправлению ошибок;
- не требует большого количества времени и дополнительных формальных процедур.

Недостатки:

- проверяющий поверхностно знаком с кодом, следовательно, проверка может быть неэффективной;
- для проверки один из разработчиков отвлекается от своей основной работы.

2. Разработчик отдает код на проверку. Код изучается сторонним программистом без присутствия автора.

Достоинства:

- простота;
- рецензент может проверить код в любое удобное для него время.

Недостатки:

- могут потребоваться дополнительные комментарии к коду автора, которые не всегда просто получить;
- длительность процесса больше, чем при использовании варианта проверки, описанного в подразд. 2.1.

Работа парами. При такой стратегии код разрабатывает пара программистов за одним терминалом: один из них является ведущим — разрабатывает код, другой обдумывает сделанное и осуществляет контроль, исправляя ошибки до того, как код будет полностью написан. Через какое-то время роли в паре меняются и тот, кто разрабатывал код, начинает контролировать, а контролирующий занимает место разработчика.

Иногда этот метод программирования используется спонтанно, когда один разработчик помогает другому решать возникшую сложную задачу.

Достоинство: осуществление контроля и исправление ошибок на этапе разработки.

Недостаток: общая производительность работников будет ниже, так как два программиста заняты решением одной задачи вместо двух.

Официальная инспекция кода группой разработчиков. Это регламентированная процедура проверки кода группой разработчиков, которая планируется заранее в строго отведенные дату и время.

По результатам такой проверки составляется отчет. Согласно результатам отчета автор дорабатывает код, и после доработки разработчик предоставляет отчет об исправлениях и откорректированный вариант кода.

Достоинства:

- привлечение группы программистов, каждый член которой выражает свой взгляд на сделанную работу;
- обмен опытом;
- глубокий анализ кода.

Недостатки:

- трудоемкая процедура, требующая времени;
- отвлечение сотрудников от основной работы;
- психологически сложный процесс для автора кода [29].

Рассмотрим подробнее процедуру ревьюирования.

1.2.3. Этапы и планирование ревьюирования

Процедура ревьюирования содержит следующие этапы:

1) автор сообщает, что код готов к ревьюированию;

2) определяется группа разработчиков для ревьюирования кода. Среди членов группы должны присутствовать:

- автор, который расскажет о проделанной работе и будет участвовать в обсуждении кода;
- председатель — лицо, которое ведет совещание;
- рецензенты — разработчики или сотрудники отдела качества, имеющие достаточные знания и квалификацию, чтобы оценить код;
- секретарь — помогает в организации совещания, ведет его протокол, записывает, какие вопросы были подняты, чтобы ничего не забыть по окончании рецензирования;

3) определяются дата и время работы группы для совместного ревьюирования, о чем оповещаются все члены группы;

4) решается вопрос, в какой форме будет происходить ревьюирование. Например, совещание с присутствием всех участников либо онлайн-совещание с использованием программы Skype или иного ПО;

5) определяется повестка совещания;

6) подготавливаются необходимые ресурсы (компьютер, проектор, ПО, распечатки и т.д.);

7) автор кода предоставляет его заранее всем членам группы. После этого внесение изменений в код нежелательно. Разработчик компании Eiffel Software отмечает, что код желательно предоставлять за неделю до общего собрания [28];

8) члены группы изучают код до начала общего собрания. Для экономии времени на общем собрании каждому разработчику рекомендуется написать личную рецензию на изученный код и выложить ее на какой-либо общий ресурс для ознакомления с ней автора кода и других членов группы [28].

В ходе ревьюирования председатель самостоятельно или совместно с секретарем организует площадку для проведения совещания по ревьюированию кода, чтобы процедура была начата без задержек. Разработчик кода (автор) в течение отведенного ему времени рассказывает о проделанной им работе, далее предлагается высказаться по структуре проекта, отметить общие замечания по коду. «Они могут касаться неправильно выбранного стиля кодирования, неудачных шаблонов приложения и проекта или неправильных идиом языка» [9, с. 496]. Затем последовательно разбирается код по строке или по блокам, ищутся ошибки или места, на которые следует обратить внимание, рассматриваются все возможные сценарии выполнения кода, выделяются недостатки, даются рекомендации по внесению изменений. Если большинство членов группы с ними согласны, они заносятся в протокол и последующий отчет. Секретарь фиксирует изменения, которые необходимо произвести, записывая имя файла и номер строки. Проблемы, которые могут касаться более широкой области кода, регистрируются секретарем для дальнейшего исследования. После ревьюирования обсуждаются последующие действия.

В результате ревьюирования могут быть приняты следующие **решения**:

- хороший код не требует доработки. Возможны небольшие замечания, которые следует учесть при последующей разработке кода;
- требуется доработка кода без организации нового совещания, назначаются проверяющий и срок для доработки кода, про-

граммист в указанные сроки производит доработку кода и отчитывается об этом;

- код требует значительной переработки, после чего его необходимо представить на повторное ревьюирование.

Помимо основной цели ревьюирования выделяют большое количество **задач**, которые оно может решить:

- всесторонняя проверка и улучшение качества кода;
- уменьшение дефектов в коде;
- улучшение коммуникации о содержании кода;
- обмен знаниями и опытом;
- обучение молодых программистов;
- повышение личной ответственности за разработку кода. Разработчик, понимая, что ему необходимо предоставить код для ревьюирования, будет более внимательно и ответственно подходить к его разработке;
- лучшее знакомство с проектом. Сотрудники группы знакомятся не только со своими задачами, но и с задачами проекта, над которыми работают их коллеги;
- постепенная выработка единой стратегии к написанию кода проекта;
- повышение ответственности группы за разрабатываемый код. Если код был проверен группой, то в случае возникновения ошибок она тоже несет за них ответственность;
- совместное инспектирование наиболее сложных участков проекта и его завершающих этапов;
- разработка более простого в сопровождении кода;
- простота последующего тестирования кода;
- удовлетворенность клиентов при работе с ПО, которое работает без ошибок, и др.

1.3. АНАЛИЗ ПРОГРАММНЫХ ПРОДУКТОВ

1.3.1. Цели, корректность и направления анализа

Большое количество программных продуктов на современном рынке зачастую вызывает вопросы пользователя: какое ПО выбрать, какое будет качественным, какое максимально отвечает потребностям? Вопросы о ПО возникают не только у пользователей, но и у программистов: как разработать качественное и востребованное ПО, что для того нужно сделать?

Для ответов на эти вопросы используют два вида мыслительных операций, используемых для исследования объектов, — анализ и синтез. Анализ позволяет выделить и изучить отдельные части объекта исследования, а синтез — объединить результаты исследования по каким-либо критериям и сделать выводы.

Что необходимо для анализа чего-либо, в том числе и ПО? Прежде всего должен быть **предмет исследования** — то, что необходимо изучить. Например, планируется проанализировать ПО, в этом случае предметом исследования будет ПО.

Затем следует определиться с **объектом исследования**. Это очень важно. Есть целые группы ПО, которые можно выделить по типу использования ПО и его функциональному назначению.

Существуют три большие **группы ПО**:

- системное — комплекс программ, предназначенных для управления работой компьютера;
- прикладное — комплекс программ, предназначенных для решения тех или иных задач пользователей;
- инструментальное — совокупность программ, предназначенных для проектирования, разработки и сопровождения ПО.

Среди этих групп в зависимости от функционального назначения можно выделить классы различных программ. При исследовании разных программ могут быть интересны их различные характеристики, т. е. анализ проводится по разным критериям. Таким образом, объект исследования определяет, по каким критериям можно вести анализ данного объекта.

Для каждого объекта можно выделить массу критериев анализа, характеризующих те или иные свойства объекта. Выбор критериев для анализа зависит от **цели исследования**. Именно она определяет направление исследования и необходимые результаты, и от нее будет зависеть набор критериев для анализа того или иного объекта, в нашем случае — программного продукта.

После постановки цели анализа выделяют ряд задач, которые нужно решить для достижения этой цели. И самая важная задача — подбор критериев для анализа выбранного объекта, так как от этого будут зависеть корректность и полнота исследования.

1.3.2. Критерии анализа и оценки программного обеспечения

Критерии анализа любого объекта, в том числе и программного обеспечения, могут быть объективными и субъективными. Например, корректность работы ПО, позволяющая оценить его правиль-

ную работу при вводе корректных данных, будет объективным критерием, а удобство использования интерфейса — субъективным. Субъективные критерии важны не меньше, чем объективные. Но необходимо понимать, что оценка субъективных критериев может различаться у разных пользователей.

Поэтому для **корректной оценки субъективного критерия** необходимо:

- разработать шкалу оценки того или иного критерия;
- опросить группу пользователей в соответствии с разработанной шкалой;
- найти среднее значение оценки.

Для **корректного анализа** критерии должны быть:

- валидными, т. е. оценивать именно то, что необходимо оценить в ходе анализа;
- предметными, т. е. относиться к оценке объекта или предмета исследования в контексте объекта;
- непротиворечивыми;
- интерпритируемыми, т. е. позволять интерпретацию полученной оценки;
- проверяемыми;
- достаточно полными, т. е. максимально охватить все существенные согласно цели исследования характеристики исследуемого объекта.

Отметим еще раз: выбор критериев для анализа ПО зависит от следующих составляющих:

- цель;
- анализ;
- объект исследования.

Это одна из самых важных и сложных задач анализа.

Рассмотрим существующие критерии оценки для анализа качества ПО.

Можно встретить различные **определения понятия «качество ПО»**:

- величина, отражающая, в каком объеме в программный продукт включен набор желаемых функций для повышения эффективности программного продукта в течение его жизненного цикла [25];
- качество — это полнота свойств и характеристик продукта, процесса или услуги, которые обеспечивают способность удовлетворять заявленным или подразумеваемым потребностям [30] (определение ISO);
- степень, в которой оно обладает требуемой комбинацией свойств [30] (определение IEEE).

Существуют **группы стандартов для оценки качества ПО**:

- ISO/IEC 25010:2011 (ГОСТ Р ИСО/МЭК 25010—2015 «Информационные технологии (ИТ). Системная и программная инженерия. Требования и оценка качества систем и программного обеспечения (SQuaRE). Модели качества систем и программных продуктов») — группа стандартов, описывающих модели качества программных продуктов;
- ISO 9001:2015, ISO/IEC 9126 (ГОСТ Р ИСО/МЭК 9126—93 «Информационная технология (ИТ). Оценка программной продукции. Характеристики качества и руководства по их применению») — стандарты качества ПО;
- ISO/IEC 14598 — набор стандартов, регламентирующих способы оценки характеристик качества.

С моделями и критериями оценки качества можно ознакомиться, изучив эти стандарты. Сравнительный анализ критериев качества и их эволюция рассматриваются в источнике [30].

1.3.3. Модели качества программного обеспечения

Чтобы познакомиться с критериями оценки качества ПО, рассмотрим несколько моделей качества.

Модель Мак-Кола. Первая модель качества была предложена Мак-Колом (рис. 1.5). Она имела **три направления анализа**:

- использование (корректность, надежность, эффективность, целостность, практичность);
- модификация (тестируемость, гибкость, сопровождаемость — факторы качества, важные для разработки новой версии ПО);
- переносимость (мобильность, возможность многократного использования, функциональная совместимость — факторы качества, важные для переносимости программного продукта на другие аппаратные и программные платформы).

Модель Боэма. Для определения качества ПО заданным набором показателей и метрик была разработана модель Боэма (рис. 1.6).

Модель FURPS/FURPS+. Одна из моделей анализа качества ПО была предложена Грейди и компанией Hewlett Packard. Она построена аналогично предыдущим, но в отличие от них состоит из двух слоев:

- первый слой определяет характеристики;
- второй — связанные с ними атрибуты.

В модели выделяли две категории требований: функциональные (F) и нефункциональные (URPS). Эти категории могут исполь-

Рис. 1.5. Модель Мак-Кола

зоваться в качестве требований к программному продукту и в оценке качества ПО. Модель широко используется современными программистами. Аббревиатура FURPS описывает требования к качеству ПО (табл. 1.1). Символ «+» используется для рас-

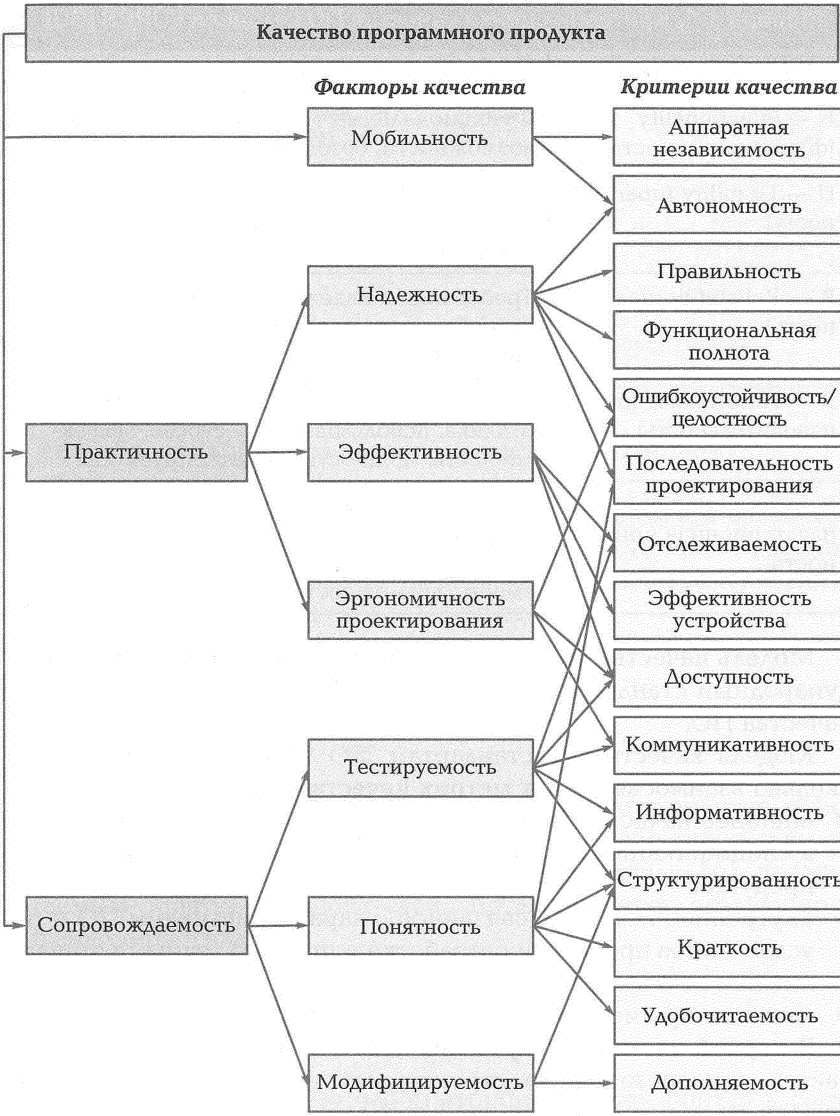


Рис. 1.6. Модель Боэма

ширения модели, добавляя к оценке ограничения проекта, ограничения, накладываемые на взаимодействие с внешними системами, требования к выполнению, физические требования, требования к лицензированию.

Таблица 1.1. Критерии оценки качества ПО в модели FURPS [30]

Критерий оценки	Описание критерия
F — Functionality (функциональность)	Функциональные требования: особенности, возможности, безопасность
U — Usability (практичность)	Требования к удобству использования: человеческий фактор, эстетика, последовательность, документация
R — Reliability (надежность)	Требования к надежности: частота возможных сбоев, отказоустойчивость, восстанавливаемость
P — Performance (производительность)	Требования к производительности: время отклика, использование ресурсов, эффективность, мощность, масштабируемость
S — Supportability (эксплуатационная пригодность)	Требования к поддержке: возможность поддержки, ремонтопригодность, гибкость, модифицируемость, модульность, расширяемость, возможность локализации

Модель качества ПО по стандартам ISO 9126. ISO 9126 — международный стандарт, определяющий оценочные характеристики качества ПО.

Модель качества по стандартам ISO 9126 рассматривает несколько взаимосвязанных **метрик качества** (рис. 1.7):

- внешнее качество, которое задается требованиями заказчика в спецификациях и отражается в характеристиках конечного продукта;
- внутреннее качество, связанное с характеристиками ПО и обусловленное процессом разработки и промежуточными этапами жизненного цикла ПО;
- качество при использовании ПО в процессе его эксплуатации.

В **первой части стандарта ISO 9126-1** качество ПО рассматривается в разрезе шести структурных наборов характеристик, каждая из которых имеет свои субхарактеристики (подхарактеристики).

1. ***Функциональность*** — набор атрибутов, характеризующий, соответствие функциональных возможностей ПО набору требуемой пользователем функциональности. Детализируется следующими субхарактеристиками:

- пригодность для применения;
- корректность (правильность, точность);

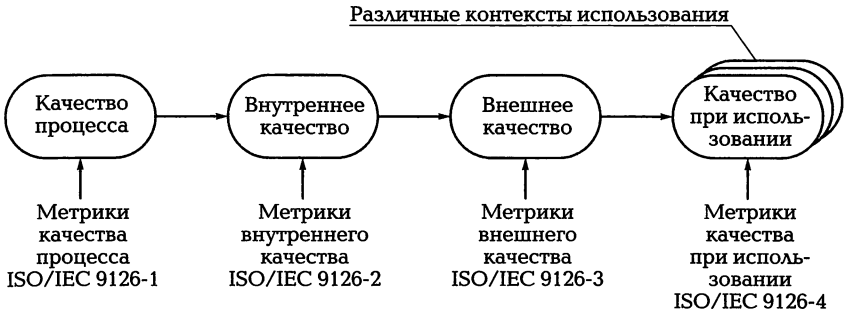


Рис. 1.7. Основные аспекты качества программного обеспечения

- способность к взаимодействию (в частности, сетевому);
- защищенность.

2. ***Надежность*** — набор атрибутов, относящихся к способности ПО сохранять свой уровень качества функционирования в установленных условиях за определенный период времени. Детализируется следующими субхарактеристиками:

- уровень завершенности (отсутствие ошибок);
- устойчивость к дефектам;
- восстанавливаемость;
- доступность;
- готовность.

3. ***Практичность*** (***применимость***) — набор атрибутов, относящихся к объему работ, которые требуют для исполнения и индивидуальной оценки такого исполнения определенным или предполагаемым кругом пользователей. Детализируется следующими субхарактеристиками:

- понятность;
- обучаемость;
- простота использования;
- привлекательность.

4. ***Эффективность*** — набор атрибутов, относящихся к соотношению между объемом используемых ресурсов при установленных условиях и уровнем качества функционирования ПО. Детализируется следующими субхарактеристиками:

- эффективность с течением времени;
- используемость ресурсов.

5. ***Сопровождаемость*** — набор атрибутов, относящихся к объему работ, которые необходимы для проведения конкретных

изменений (модификаций) ПО. Детализируется следующими субхарактеристиками:

- удобство для анализа;
- изменяемость;
- стабильность;
- тестируемость.

6. ***Мобильность*** — набор атрибутов, относящихся к способности ПО быть перенесенным из одного окружения в другое. Детализируется следующими субхарактеристиками:

- адаптируемость;
- простота установки (инсталляции);
- совместимость;
- взаимозаменяемость;
- согласованность мобильности.

Вторая и третья части стандарта ISO 9126 посвящены формализации внешних и внутренних метрик для характеристик качества сложных программных средств. Они содержат унифицированную рубрикацию, где отражены имя и назначение метрики, метод ее применения, способ измерения, тип шкалы метрики, тип измеряемой величины, исходные данные для измерения и сравнения, а также этапы жизненного цикла программного средства, к которым применима метрика.

Четвертая часть стандарта ISO 9126-4 предназначена для покупателей, поставщиков, разработчиков, сопровождающих, пользователей и менеджеров качества программных средств. В ней рассматриваются рекомендуемые виды характеристик программных средств.

Модель оценки ISO 9126, представленная в стандартах оценки качества ПО, показана на рис. 1.8.

Стандарты ISO пересматриваются каждые пять лет. В ходе пересмотра они обновляются и дорабатываются. На данный момент группа стандартов ISO 9126 переработана и переименована. Познакомиться с новыми стандартами оценки качества можно, изучив стандарты ISO/IEC 25010:2011, ISO/IEC 25023:2016, ISO/IEC 25023:2016, ISO/IEC 25022:2016. Несмотря на имеющиеся в новых стандартах изменения, основная идея оценки качества ПО осталась той же. Актуальные версии стандартов см.: https://www.iso.org.

Анализ модели оценки качества ПО свидетельствует, что существуют разные подходы к его оценке и большое количество критериев анализа.

Говоря об анализе ПО, мы подразумевали, что есть некоторое ПО, которое можно проанализировать по определенным крите-

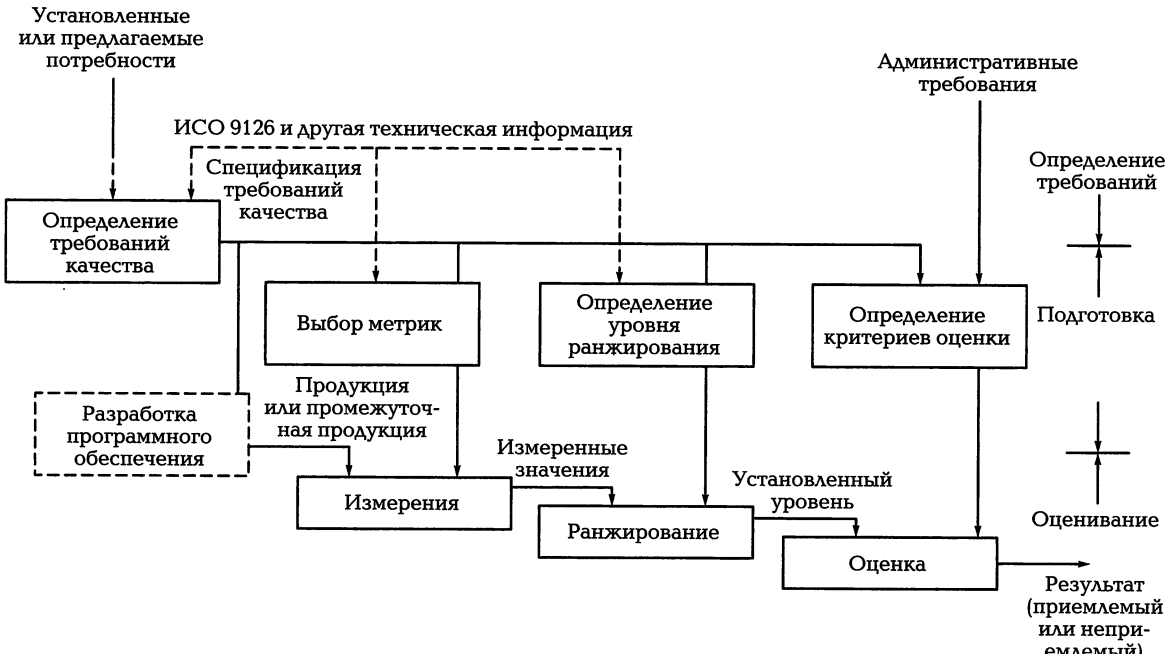


Рис. 1.8. Модель оценки ISO 9126

риям и по результатам анализа сделать выводы. Очень часто перед пользователями и разработчиками встает задача не просто анализа программного продукта, а сравнительного анализа определенных программных продуктов.

1.4. СРАВНИТЕЛЬНЫЙ АНАЛИЗ ПРОГРАММНЫХ ПРОДУКТОВ

1.4.1. Критерии анализа и представление результатов

Стоит отметить, что при сравнительном анализе программных продуктов целесообразно выбирать продукты с аналогичным функционалом. Мы можем сравнить велосипед с автомобилем, так как и велосипед, и автомобиль — это средства передвижения, но гораздо интереснее сравнить разные марки велосипедов или разные марки автомобилей между собой. Если говорить о ПО, то операционная система скорее всего будет сравниваться с операционной системой, а информационная система (ИС) для ведения бухгалтерской деятельности — с аналогичной ИС.

Любой из указанных ранее критериев оценки, а также полный набор критериев из какой-либо методики оценки качества ПО мы можем использовать для сравнительного анализа ПО. Вопрос в том, нужны ли будут все перечисленные критерии для оценки? Ведь оценка по всем критериям какой-либо модели будет трудоемкой и займет большое количество времени.

Чтобы ответить на этот вопрос, вспомним, что у любого анализа должна быть цель. Выбор критериев анализа обусловлен целью и объектом исследования.

Потребность в сравнительном анализе ПО может возникнуть у разработчиков ПО, чтобы создать конкурентноспособный продукт, востребованный у потребителя, а также у пользователя, который задумывается, какое ПО купить или установить на свой компьютер.

Компьютерные технологии широко применяются в различных сферах нашей жизни. Большинство из нас имеют персональные компьютеры, а значит, являются пользователями ПО.

Чаще всего при **выборе ПО** мы рассматриваем:

- его функциональные возможности;
- интерфейс (удобство использования, простоту освоения, дизайн);

- наличие поддержки (руководство пользователя и система помощи);
- требования к аппаратному обеспечению для корректной работы;
- цену ПО;
- надежность — бесперебойное выполнение всех заявленных функций, корректное завершение работы при возникновении различных возможных ошибок (по вине либо не по вине пользователя).

Этот сокращенный список критериев оценки ПО может быть детализирован и расширен.

Результаты анализа могут быть представлены в любой удобной для их изучения форме, например в виде:

- текстового описания, в котором присутствует информация об объекте, целях и задачах исследования, выбранных критериях оценки, полученных результатах с их интерпретацией и выводами;
- табличного представления результатов сравнения;
- графического представления — схемы, рисунки, диаграммы.

В конце исследования (сравнительного анализа) обязательно необходимо сделать выводы.

1.4.2. Примеры сравнительного анализа программных продуктов

Сравнительный анализ текстовых редакторов

Условия. Петр Иванович — экономный директор ООО «Яблони и груши». Мария Ивановна — его секретарь, занимающаяся документацией компании. На компьютере Марии Ивановны установили операционную систему Windows 7 с текстовыми редакторами «Блокнот» и WordPad.

Необходимо произвести сравнительный анализ программных продуктов Microsoft Word, WordPad и «Блокнот» для выбора оптимального продукта в соответствии с рабочими задачами Марии Ивановны.

Для решения данной задачи необходимо узнать у Марии Ивановны, какой функционал текстового редактора ей нужен для работы с документацией. Мы выяснили, что необходимо:

- работать с файлами — создавать их, редактировать и сохранять в различных форматах;
- создавать и редактировать документы с отформатированным шрифтом и абзацами, таблицами, списками, распределением текста по колонкам;

- работать с многостраничными документами — вставлять нумерацию страниц и создавать оглавление;
- создавать схемы, вставлять изображения и формулы;
- вставлять диаграммы из файлов Excel;
- использовать предварительный просмотр документа и выводить его на печать.

Побеседовав с директором, мы выяснили, что он хотел бы получить сделанную документацию и экономно расходовать денежные средства: «Никакого лишнего программного обеспечения!».

Критерии и задачи исследования. Предмет — ПО. Объекты: текстовые редакторы Microsoft Word, WordPad и «Блокнот». Цель — произвести сравнительный анализ текстовых редакторов для выбора редактора с оптимальным функционалом для работы секретаря.

Задачи:

- сформировать критерии оценки программных продуктов в зависимости от потребностей пользователей;
- изучить ПО;
- произвести сравнительный анализ программных продуктов в соответствии с выбранными критериями;
- сделать выводы в зависимости от полученных результатов.

Последовательность проведения исследования. Рассмотрим интерфейс ПО и сформируем таблицу с остальными критериями оценки функциональных возможностей ПО (табл. 1.2). Выявим, какими функциональными возможностями обладает каждый программный продукт.

Интерфейс Microsoft Word (рис. 1.9):

- строка заголовка;
- ленточные панели с пиктограммами команд (9 шт.), закладка со списком команд;
- рабочая область;
- строка состояния с возможностью просмотреть статистику документа, изменить вид и масштаб документа.

Таблица 1.2. Функциональные возможности программного обеспечения для создания и редактирования документов секретаря ООО «Яблони и груши»

Критерий оценки	Программный продукт		
	Microsoft Word	WordPad	«Блокнот»
Работа с файлами (создание, редактирование, сохранение)	Да	Да	Да

Окончание табл. 1.2

Критерий оценки	Программный продукт		
	Microsoft Word	WordPad	«Блокнот»
Форматы файлов при сохранении	rtf, doc, docx, html, txt, wps	rtf, doc, txt	txt
Форматирование шрифта	Тип шрифта, начертание, размер, стиль, заливка, цвет, видоизменение, положение на странице	Тип шрифта, начертание, размер, стиль, заливка, цвет	Тип шрифта, начертание, размер
Форматирование абзацев: установка абзацных отступов и выравнивание	Да	Да	Нет
Создание таблиц	Да	Нет	Нет
Создание списков	Да	Да	Нет
Распределение текста по колонкам	Да	Нет	Нет
Вставка нумерации страниц	Да	Нет	Нет
Создание оглавления	Да	Нет	Нет
Создание схем	Да	Нет	Нет
Вставка изображений	Да	Да	Нет
Вставка формул	Да	Нет	Нет
Предварительный просмотр	Да	Да	Да
Вывод на печать	Да	Да	Да
Поддержка встраивания и связывания объектов	Да	Да	Нет
Финансовые затраты	Необходимы дополнительные затраты на покупку лицензионного ПО	Устанавливается с операционной системой, не требует дополнительных финансовых затрат	

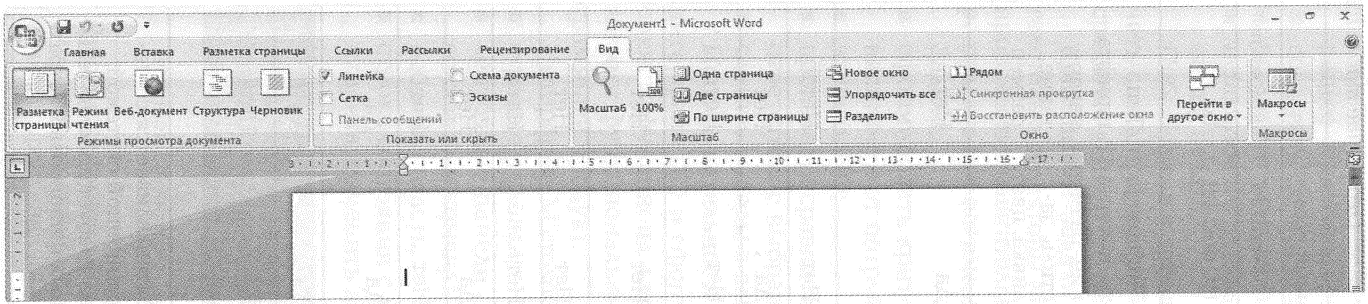


Рис. 1.9. Интерфейс Microsoft Word

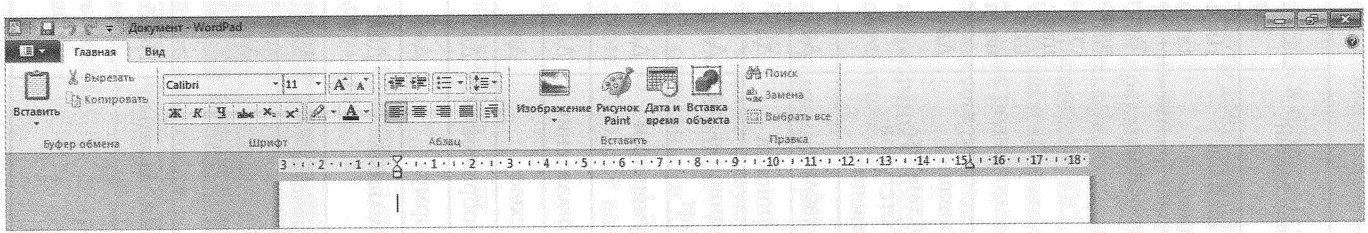


Рис. 1.10. Интерфейс WordPad

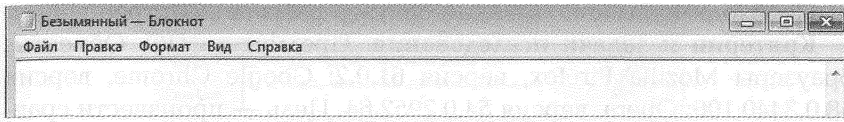


Рис. 1.11. Интерфейс программы «Блокнот»

Интерфейс WordPad (рис. 1.10):

- строка заголовка;
- ленточные панели с пиктограммами команд (2 шт.), закладка со списком команд;
- рабочая область;
- строка состояния с возможностью изменить масштаб документа.

Интерфейс программы «Блокнот» (рис. 1.11):

- строка заголовка;
- закладки со списками команд (5 шт.);
- рабочая область.

Строка состояния в программе «Блокнот» отсутствует.

Выводы. По результатам исследования можно сделать выводы.

1. Анализ интерфейсов ПО свидетельствует, что редактор программы Microsoft Word имеет больше возможностей для работы с текстовыми документами. Пока этот вывод — один из предварительных, поскольку мы еще не оценили функционал ПО в соответствии с потребностями секретаря компании.

2. На основе анализа функциональных возможностей ПО для создания и редактирования документов (см. табл. 1.2) можно сделать вывод, что для работы секретаря подойдет только текстовый редактор Microsoft Word. Другие редакторы не смогут удовлетворить потребности компании для разработки документов.

Заметим, что при анализе текстовых редакторов мы не использовали субъективные критерии. Это будет возможно при выполнении группой практической работы «Сравнительный анализ офисных пакетов». Каждый сможет поставить свою субъективную оценку по определенному критерию, а затем у группы будет возможность найти среднее арифметическое значение выставленных субъективных оценок.

Сравнительный анализ скорости работы браузеров

Условия. У пользователя на компьютере установлены три браузера: Mozilla Firefox, версия 61.0.2; Google Chrome, версия 68.0.3440.106; Opera, версия 54.0.2952.64.

Необходимо выяснить, какой браузер работает быстрее.

Критерии и задачи исследования. Предмет — ПО. Объекты: браузеры Mozilla Firefox, версия 61.0.2; Google Chrome, версия 68.0.3440.106; Opera, версия 54.0.2952.64. Цель — произвести сравнительный анализ скорости работы браузеров, оценив, как быстро браузер открывает различные интернет-ресурсы на компьютере с определенным аппаратным обеспечением и как быстро происходит запуск браузера.

Задачи:

- произвести сравнительный анализ скорости запуска браузеров (первый и повторный запуски);
- осуществить сравнительный анализ общей производительности браузеров, используя сервис browserbench.org.

Последовательность проведения исследования. Рассмотрим основные этапы исследования.

1. Проанализируем ПО. Существует множество утилит для исследования работы браузеров и иного ПО. В нашем исследовании оценим скорость загрузки интернет-страниц и время запуска браузера. Для получения информации о скорости запуска браузеров используем программу `AppTimer`, позволяющую получить информацию о скорости запуска ПО.

Для тестирования ПО с помощью данной программы необходимо указать адрес приложения (`Application`), файл для записи результатов (`Log File`), количество тестирований через определенный промежуток времени (`Execution`), задержку между тестированиями в миллисекундах (`Delay`) (рис. 1.12).

2. Для исследования общей производительности браузеров используем сервис https://browserbench.org. Сервис имеет несколько утилит (рис. 1.13):

- `ARES-6` — измеряет время выполнения различных функций JavaScript. Состоит из четырех подтестов: `Air`, `Basic`, `Babylon` и `ML`;
- `Speedometer 2.0` — измеряет среднюю скорость открытия браузером большого количества интернет-страниц с различным контентом за минуту. Чем выше эта скорость, тем быстрее работает браузер;
- `JetStream` — содержит большое количество тестов, которые позволяют оценить работу браузеров с контентом JavaScript при различных нагрузках и определить средний результат производительности браузеров по результатам выполнения различных тестов;
- `MotionMark` — позволяет оценить производительность браузеров при обработке графики и анимации.

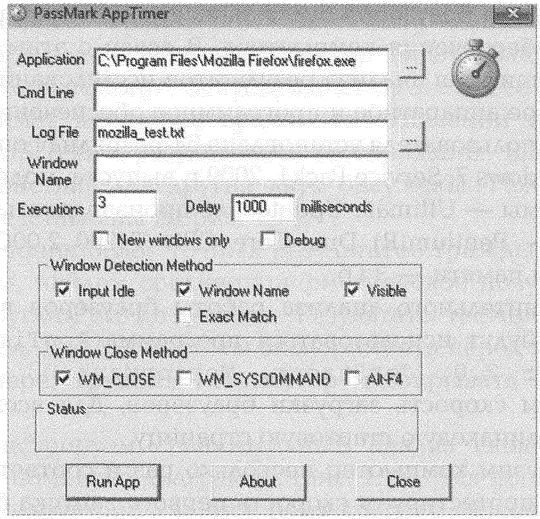


Рис. 1.12. Использование программы `AppTimer` для тестирования

На производительность браузера влияют число установленных расширений, тема оформления и объем кеша. В связи с этим для объективного тестирования необходимо почистить кеш браузера, отключить расширения и убрать тему оформления.

Рис. 1.13. Сервис https://browserbench.org/ — тестирование общей производительности браузеров

Также работа браузеров будет зависеть от аппаратного и программного обеспечения компьютера. В связи с этим для получения подходящих для анализа результатов исследования необходимо одинаковое аппаратное и программное обеспечение.

У нашего пользователя установлена 64-разрядная операционная система Windows 7, Service Pack1, 2009 г. выпуска, модель операционной системы — Ultimate Update3, ее производительность — 3.0, процессор — Pentium(R) Dual-Core CPU T4200 2.00GHz, размер оперативной памяти — 3 Гб.

Для сравнительного анализа работы браузеров в нашем исследовании будут использоваться программа `AppTimer` и тесты `Speedometer 2.0, JetStream, MotionMark`.

3. Оценим скорость загрузки браузеров. Для всех браузеров установим одинаковую стартовую страницу.

Перезагрузим компьютер несколько раз и соответственно несколько раз протестируем скорость первого запуска браузера после запуска операционной системы. Для получения результатов о скорости запуска браузера используем программу `AppTimer` (см. рис. 1.12). В нашем исследовании тестирование производится три раза, а затем находится среднее арифметическое, результаты округляются до сотых.

Запустим несколько приложений и произведем повторные запуски браузеров. С помощью программы `AppTimer` протестируем скорость загрузки браузеров. Полученные результаты представим в виде табл. 1.3.

Таблица 1.3. Тестирование скорости загрузки браузеров

Браузер	Скорость загрузки, %			
	Тест 1	Тест 2	Тест 3	Среднее значение
Первый запуск				
Opera	53,74	54,08	53,78	53,86
Google Chrome	51,21	51,47	51,21	51,30
Mozilla Firefox	54,42	54,08	52,10	53,53
Повторный запуск				
Opera	58,93	59,43	58,95	59,10
Google Chrome	56,54	56,41	55,23	56,06
Mozilla Firefox	58,78	56,64	57,86	57,76

4. С помощью сервиса browserbench.org оценим производительность браузеров. Откроем браузер Opera. Зайдем на сайт https://browserbench.org/. Запустим тест `Speedometer 2.0`. Тестирование займет некоторое время. По окончании тестирования сделаем PrintScreen с результатами тестирования. Выполним аналогичные действия для других двух браузеров. Полученные результаты представлены на рис. 1.14.

Для получения более точных результатов трижды протестируем браузеры, найдем среднее арифметическое полученных значений и округлим его до десятых. Результаты тестирования представим в табл. 1.4. Для чистоты эксперимента перед тестированием будем перезагружать компьютер и закрывать посторонние вкладки.

Продемонстрируем сравнительный анализ скорости работы браузеров с помощью диаграммы (рис. 1.15). Для построения диаграммы используем полученное нами среднее значение скорости.

5. Тест `JetStream` содержит группу тестов, направленных на оценку скорости работы браузеров с Java Script. Тестирование производится трижды, в итоге находится средний коэффициент производительности.

Протестируем каждый из браузеров с помощью группы тестов `JetStream`. Сделаем PrintScreen с результатами тестирования (рис. 1.16).

Результаты сравнительного анализа тестирования, округленные до сотых, представлены на диаграмме (рис. 1.17).

6. Оценить производительность браузеров при обработке графики и анимации можно с помощью теста `MotionMark`. Тестирование для каждого браузера на компьютере пользователя проводилось многократно. Результаты тестирования имели низкую валидность и высокую погрешность (рис. 1.18). В связи с этим объективно оценить производительность браузеров при обработке анимации и графики на компьютере пользователя не удалось.

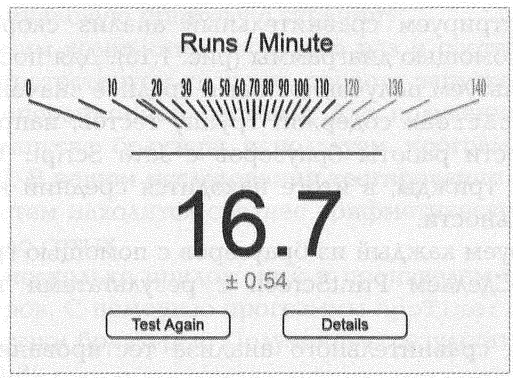
Результаты пользователей, имеющих компьютер с другим аппаратным и программным обеспечением, могут иметь более высокую валидность и более низкую погрешность.

Большое количество сайтов содержит анимацию. Также есть пользователи, играющие в видеоигры. В связи с этим для сравнительного анализа браузеров тест интересен, если возможно получить объективные и корректные результаты тестирования.

Выводы. По результатам исследования можно сделать выводы.

1. Проанализировав данные табл. 1.3, можно сказать, что самый быстрый запуск у браузера Google Chrome, второе место за-

а



б

в

Рис. 1.14. Результаты тестирования скорости работы браузеров с помощью теста `Speedometer 2.0`:

а — Opera; *б* — Mozilla Firefox; *в* — Google Chrome

Таблица 1.4. Тестирование производительности браузеров с помощью теста `Speedometer 2.0`

Браузер	Производительность, %			
	Тест 1	Тест 2	Тест 3	Среднее значение
Opera	19,9	19,3	19,3	19,5
Google Chrome	21,8	21,5	21,7	21,7
Mozilla Firefox	16,7	16,2	16,9	16,6

нимает Mozilla Firefox, Opera запускается немного медленней, чем два предыдущих браузера.

2. Проанализировав результаты тестирования скорости работы с помощью теста `Speedometer 2.0`, можно сказать, что самую высокую скорость работы продемонстрировал браузер Google Chrome, скорость у браузера Opera на 10,1 %, Mozilla Firefox — на 23,5 % ниже.

3. Оценка скорости работы браузеров с Java Script с помощью теста `JetStream` показала, что более высокую производительность при обработке Java Script продемонстрировал браузер Mozilla Firefox, средние показатели тестирования у браузеров Google Chrome и Opera — соответственно на 6,7 и 8,8 % ниже.

4. Объективно оценить производительность браузеров при обработке анимации и графики на компьютере пользователя с помощью теста `MotionMark` не удалось.

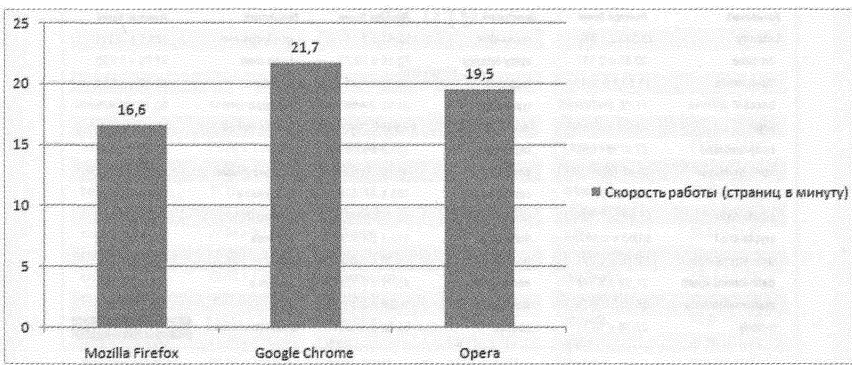


Рис. 1.15. Сравнительный анализ скорости работы браузеров с помощью теста `Speedometer 2.0`

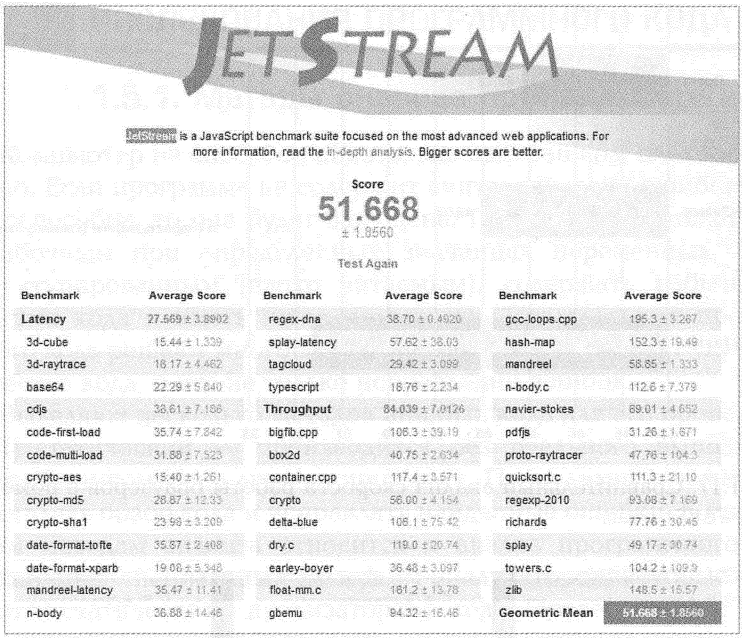
5. В ходе сравнительного анализа скорости работы браузеров нельзя сделать однозначный вывод о том, что какой-то из браузеров является самым быстрым, так как существуют браузеры, которые быстрее загружаются, есть те, которые имеют более высокую скорость загрузки интернет-страниц, но скорость обработки Java Script по результатам тестирования у них ниже.

При этом стоит сказать, что на компьютере пользователя для большинства сайтов быстрее будет работать Google Chrome, так как этот браузер имеет более высокую скорость загрузки интернет-страниц и скорость запуска (см. табл. 1.3, 1.4). Показатель тестирования `JetStream` у данного браузера всего лишь на 6,7 % ниже максимального.

Также для большого количества интернет-ресурсов скорость работы браузера Opera на компьютере пользователя будет выше, чем скорость работы браузера Mozilla Firefox. При этом сложные сайты, содержащие большое количество кода Java Script, браузер Mozilla Firefox обработает быстрее.

а

Рис. 1.16. Результаты тестирования с помощью теста `JetStream` браузеров Opera (*а*), Mozilla Firefox (*б*) и Google Chrome (*в*)



JETSTREAM

JetStream is a JavaScript benchmark suite focused on the most advanced web applications. For more information, read the in-depth analysis. Bigger scores are better.

Score
51.668
± 1.8560

Test Again

Benchmark	Average Score	Benchmark	Average Score	Benchmark	Average Score
Latency	27.569 ± 3.8902	regex-dna	38.70 ± 0.4920	gcc-loops.cpp	195.3 ± 3.267
3d-cube	15.44 ± 1.339	splay-latency	57.62 ± 38.03	hash-map	152.3 ± 19.49
3d-raytrace	16.17 ± 4.462	tagcloud	29.42 ± 3.099	mandreel	58.85 ± 1.333
base64	22.29 ± 5.640	typescript	18.76 ± 2.234	n-body.c	112.6 ± 7.379
cdjs	38.61 ± 7.186	**Throughput**	84.039 ± 7.0126	navier-stokes	89.01 ± 4.652
code-first-load	35.74 ± 7.842	bigfib.cpp	106.3 ± 39.19	pdfjs	31.26 ± 1.671
code-multi-load	31.58 ± 7.523	box2d	40.75 ± 2.634	proto-raytracer	47.76 ± 104.3
crypto-aes	15.40 ± 1.261	container.cpp	117.4 ± 3.571	quicksort.c	111.3 ± 21.10
crypto-md5	28.87 ± 12.33	crypto	56.00 ± 4.154	regexp-2010	93.08 ± 7.169
crypto-sha1	23.98 ± 3.209	delta-blue	108.1 ± 75.42	richards	77.76 ± 30.45
date-format-tofte	35.87 ± 2.408	dry.c	119.0 ± 20.74	splay	49.17 ± 30.74
date-format-xparb	19.06 ± 5.348	earley-boyer	36.48 ± 3.097	towers.c	104.2 ± 109.9
mandreel-latency	35.47 ± 11.41	float-mm.c	161.2 ± 13.78	zlib	148.5 ± 16.57
n-body	36.58 ± 14.46	gbemu	94.32 ± 16.46	**Geometric Mean**	51.668 ± 1.8560

б

JETSTREAM

JetStream is a JavaScript benchmark suite focused on the most advanced web applications. For more information, read the in-depth analysis. Bigger scores are better.

Score
48.019
± 1.1655

Test Again

Benchmark	Average Score	Benchmark	Average Score	Benchmark	Average Score
Latency	24.108 ± 0.17872	regex-dna	60.63 ± 2.906	gcc-loops.cpp	160.1 ± 7.246
3d-cube	21.80 ± 1.583	splay-latency	78.78 ± 14.27	hash-map	90.95 ± 3.397
3d-raytrace	13.30 ± 4.257	tagcloud	22.40 ± 1.809	mandreel	59.79 ± 0.3551
base64	11.75 ± 1.059	typescript	25.06 ± 4.189	n-body.c	60.97 ± 3.076
cdjs	42.58 ± 8.920	**Throughput**	81.787 ± 3.7877	navier-stokes	74.93 ± 0.8494
code-first-load	27.88 ± 2.749	bigfib.cpp	140.7 ± 6.302	pdfjs	35.84 ± 21.27
code-multi-load	26.99 ± 2.341	box2d	57.62 ± 10.19	proto-raytracer	86.51 ± 9.313
crypto-aes	20.53 ± 1.151	container.cpp	102.9 ± 2.686	quicksort.c	129.8 ± 1.089
crypto-md5	16.25 ± 0.9882	crypto	53.20 ± 14.26	regexp-2010	114.7 ± 3.036
crypto-sha1	8.543 ± 0.3093	delta-blue	119.7 ± 1.151	richards	73.80 ± 4.224
date-format-tofte	19.45 ± 4.363	dry.c	57.84 ± 1.705	splay	49.34 ± 15.15
date-format-xparb	23.73 ± 7.330	earley-boyer	44.96 ± 1.523	towers.c	119.8 ± 4.946
mandreel-latency	32.60 ± 1.161	float-mm.c	161.9 ± 1.657	zlib	106.0 ± 4.437
n-body	25.55 ± 2.184	gbemu	64.97 ± 10.68	**Geometric Mean**	48.019 ± 1.1655

в

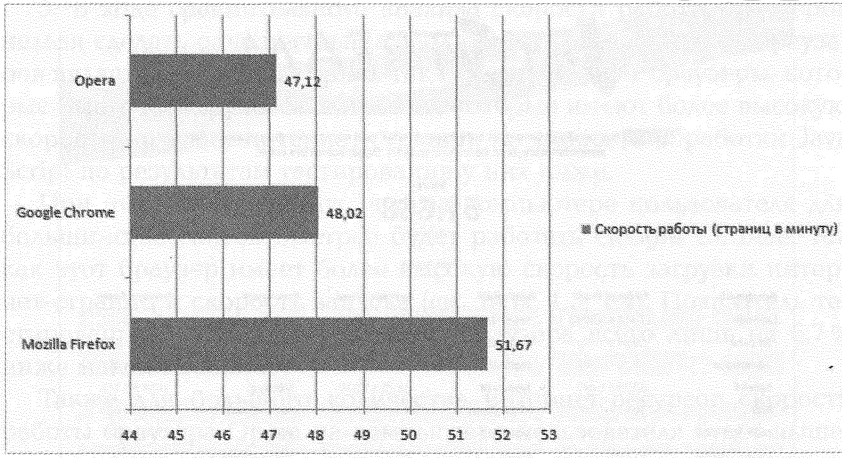


Рис. 1.17. Сравнительный анализ скорости работы браузеров с помощью теста `JetStream`

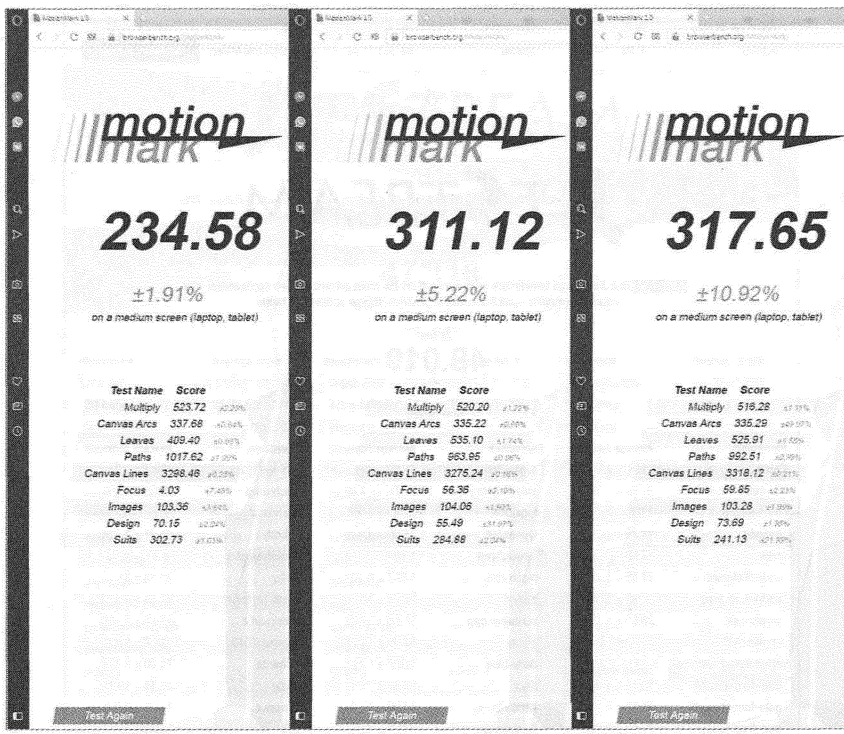


Рис. 1.18. Результаты трех вариантов тестирования скорости работы браузера Opera с помощью `MotionMark`

1.5.1. Методы анализа программного кода

Компьютер не способен понять, как написан код, плохо или хорошо. Если программа не содержит синтаксических ошибок и работоспособна, то она будет запущена. При этом код может быть ошибочным при определенных значениях переменных, плохо структурированным (плохо читаемым), содержать избыточные участки кода и иметь иные недостатки. Исследования программного кода решают много важных задач. Они позволяют улучшить качество кода, и это не только исправление ошибок.

Выделяют несколько **методов анализа кода** для его проверки:

1) статический — производится без реального выполнения разработчиком исследуемых программ. Программист может осуществлять обзор кода и исправлять найденные ошибки. К статическим методам анализа относится и анализ программного кода с помощью специальных программ-анализаторов, которые называются статическими анализаторами кода. Проверив код с помощью такой программы, программист получает информацию о проблемных областях кода, проверяет их и, если необходимо, исправляет ошибки или недостатки;

2) динамический — предполагает выполнение программ на реальном или виртуальном процессоре. Для эффективного анализа кода требуется большое количество входных данных, чтобы проанализировать работу программы во всех возможных ситуациях. Для динамического анализа кода также существуют специальные утилиты;

3) гибридный — предполагает сочетание различных методик анализа кода для его исследования. К гибридным методам анализа можно отнести:

- комбинирование результатов статического анализа и тестирования для повышения точности результатов;
- создание тестов на основе статического анализа;
- статический анализ для автоматического формирования моделей, для которых применяются формальные методы проверки моделей;
- уточнение результатов статического анализа с помощью метода проверки моделей.

Каждая группа методов помогает решить определенные задачи и имеет свои достоинства и недостатки. Рассмотрим статический и динамический анализ кода.

Статический анализ (и статистические анализаторы) решает ряд важных задач, таких как:

- выявление различных типов ошибок и возможных уязвимостей программного кода;
- подсчет метрик. Метрика ПО — мера, позволяющая получить численное значение некоторого свойства ПО или его спецификаций;
- формирование рекомендаций по оформлению кода.

Достоинства методов статического анализа кода:

- полное покрытие кода. Статические анализаторы проверяют все фрагменты кода, что позволяет находить ошибки в коде, используемом крайне редко;
- не зависит от используемого компилятора и среды, в которой будет выполняться скомпилированная программа, что позволяет находить скрытые ошибки, которые могут проявить себя только через несколько лет. Такие ошибки могут возникнуть при смене версии компилятора или использовании других ключей для оптимизации кода;
- можно легко и быстро обнаруживать опечатки, на выявление которых без использования статического анализатора может быть потрачено значительное время;
- своевременная проверка и устранение дефектов в программе способствуют уменьшению стоимости их исправления, так как исправление ошибки на этапе тестирования будет дороже, чем ее исправление на этапе конструирования.

Недостатки методов статического анализа кода:

- недостаточно хорошая диагностика утечек памяти и параллельных ошибок. Для выявления подобных ошибок необходимо виртуальное выполнение части программы, что для анализатора сложно реализовать, так как подобные алгоритмы требуют много памяти и процессорного времени. Обычно статические анализаторы ограничиваются диагностикой простых случаев. Для диагностики таких ошибок используются динамические методы анализа кода;
- корректный код при анализе тоже может попасть в список недостатков. Такие ситуации называют ложнопозитивными событиями. На анализ таких событий разработчик тратит время. Полученная информация о недостатках требует проверки.

Динамический анализ выполняется с помощью наборов данных, которые подаются на вход исследуемой программе. Эффективность анализа зависит от качества и количества входных данных для тестирования, обеспечивающих полноту покрытия кода, которая будет обеспечена результатами тестирования.

Динамический анализ кода решает ряд важных задач, в частности позволяет:

- имитировать поведение пользователя и рассматривать работу программы в различных ситуациях и с различными наборами данных, корректными и некорректными;
- находить программные ошибки, к которым можно отнести не только ошибки кода, но и ошибочные результаты и вычисления, полученные при выполнении программы или ее модулей;
- обнаружить наличие уязвимостей в программе;
- оценить используемые ресурсы, время выполнения программы в целом, время выполнения отдельных модулей программы, количество внешних запросов, количество используемой оперативной памяти и других ресурсов;
- получить определенные наборы метрик.

Динамическое тестирование необходимо в областях, где главными критериями являются надежность программы, время отклика или потребляемые ресурсы.

Достоинства динамического анализа кода:

- обычно не присутствует появление ложных срабатываний;
- обнаруженная ошибка является фактической, а не возможной;
- позволяет протестировать программы с закрытым кодом.

Недостатки динамического анализа кода:

- обнаруживает дефекты только при тестировании с определенными наборами данных, в частях программы, которые тестирование не охватывает, ошибки могут быть не обнаружены;
- требуются значительные вычислительные ресурсы для проведения тестирования;
- только один путь выполнения может быть проверен в каждый конкретный момент времени. Требуется большое количество тестовых запусков для большей полноты тестирования;
- при тестировании на реальном процессоре исполнение некорректного кода может привести к непредсказуемым последствиям.

Так как у каждого из рассмотренных видов анализа есть свои слабые и сильные стороны, наиболее эффективно их совместное использование.

1.5.2. Методы исследования кода

Рассмотрим кратко следующие **методы исследования кода**:

- ***отладка*** — этап разработки программы, на котором обнаруживают, локализуют и устраняют ошибки. В ходе отладки прихо-

дится узнавать текущие значения переменных, по какому пути выполнялась программа;

- ***трассировка*** — процесс пошагового выполнения программы, в ходе которого разработчик видит последовательность выполнения команд и значения переменных при их выполнении. Это позволяет обнаружить ошибки. При трассировке можно задать точки остановки, чтобы начать ее и окончить в необходимых для разработчика участках программного кода;
- ***ревьюирование*** — один из методов программной инженерии, направленный на проверку кода, его анализ и составление рецензии о проверенном коде, основной целью которого является повышение качества программного кода. Подробнее виды ревьюирования и решаемые им задачи рассматривались ранее;
- ***тестирование*** — метод, направленный на обнаружение как можно большего количества ошибок в программном продукте. Существует большое количество методологий и методов тестирования, с которыми можно познакомиться, изучив специальную литературу [28, 29, 32]. Отметим, что тестирование может производиться по принципам «белого», «черного» и «серого» ящика:
  - ✓ «белый ящик» — программный код, информация о внутренней структуре, устройстве и реализации программного продукта доступны для исследования. Использование данного принципа для исследования программного кода требует от исследователя высокой квалификации;
  - ✓ «черный ящик» — функциональность работы ПО исследуется без доступа к программному коду. Тестировщики пишут тест-кейсы, опираясь только на требования и спецификацию программного продукта. Метод имитирует поведение пользователя, который использует программу, но не имеет никаких сведений о ее устройстве;
  - ✓ «серый ящик» — метод тестирования или изучения программного продукта с частичным знанием его внутреннего устройства. Необходимости в доступе к программному коду нет. Тесты пишутся на основе знания архитектуры, алгоритмов и других описаний программы;
- ***профилирование*** — сбор характеристик работы программы, таких как время выполнения отдельных фрагментов, число верно предсказанных условных переходов, число кеш-промахов и т. д. Инструменты, используемые для подобного анализа работы, называют профилировщиками или профайлерами;
- ***обратное проектирование*** (reverse engineering) — процесс исследования и анализа машинного кода, направленный на по-

нимание общих механизмов функционирования программы и на перевод программы в машинных кодах на более высокий уровень абстракции, возможно даже до восстановления текста программы на исходном языке программирования;

- ***дизассемблирование*** — процесс и (или) способ получения исходного кода программы на ассемблере из программы в машинных кодах. Этот метод используется, когда программа скомпилирована и нет возможности изучить программный код без дополнительных действий по его получению в доступном для изучения виде;
- ***использование анализаторов трафика (снифферов)***. Снифферы — это программы или устройства для перехвата и анализа своего и чужого трафика. Они позволяют отследить сетевой трафик, генерируемый программой;
- ***использование анализаторов программного кода***, позволяющих производить анализ программного кода без реального выполнения программ;
- ***использование утилит для динамического анализа кода***;
- ***экспертиза ПО*** — разновидность инженерно-технической экспертизы, при которой независимая группа высококвалифицированных экспертов всесторонне изучает разработанное ПО. Экспертиза — самый дорогой и длительный анализ приложения, по результатам которого заказчик получает полный отчет о работе приложения при его эксплуатации. В отчете содержатся информация о наличии уязвимостей, оценке рисков, рекомендации по устранению уязвимостей. Помимо обозначенного, решается широкий круг задач по исследованию ПО. Полученные результаты также вносятся в отчет.

1.6. МЕХАНИЗМЫ И КОНТРОЛЬ ВНЕСЕНИЯ ИЗМЕНЕНИЙ В КОД

Процесс разработки имеет коллективный характер. При этом на практике зачастую невозможно реализовать линейную стратегию разработки программных средств. В связи с этим для контроля внесений изменений в код группой разработчиков актуально ПО, называемое системой контроля версий (СКВ), о котором мы говорили ранее.

Система контроля версий позволяет:

- команде программистов работать с одним и тем же хранилищем исходного кода, называемым репозиторием проекта;

- хранить несколько версий одних и тех же файлов проекта;
- вести учет и контроль версий;
- получать программисту личную копию хранилища и работать с этой копией на своем локальном компьютере;
- возвращаться к более ранним версиям файлов кода, получать актуальные версии файлов и решать ряд других важных задач.

Существуют разные **типы СКВ** со своими алгоритмами и возможностями работы:

- локальная;
- централизованная;
- децентрализованная (распределенная).

Как уже отмечалось, СКВ может быть блокирующей и неблокирующей и применять одну из моделей доступа.

При **блокирующей модели СКВ** один из разработчиков получает доступ к файлу для редактирования, другие, пока не снята пометка о редактировании и не сохранен отредактированный вариант, могут иметь доступ только для чтения.

При **неблокирующей модели СКВ** один и тот же файл может изменяться одновременно несколькими разработчиками. По завершении редактирования модификации объединяются автоматически.

При **организации репозиториев** соблюдаются определенные принципы:

- репозиторий содержит дерево всех файлов и директорий проекта, хранит главную копию всех файлов исходного кода и вспомогательные файлы и документы;
- при записи файла в репозиторий сохраняется его предыдущее состояние, и только после этого происходит обновление файла;
- внесенные изменения становятся доступными для других разработчиков команды;
- при чтении файла из репозитория по умолчанию предоставляется последняя версия файла со всеми сделанными изменениями.

При **работе с репозиторием** СКВ реализует ряд механизмов:

- контроль за исходным кодом. Это механизм управления файлами, код которых разрабатывается. Он поддерживает файлы, структуру их каталогов и регулирует порядок одновременного доступа к коду и его модификациям;
- управление версиями. Система контроля за исходным кодом, регистрирующая изменения, сделанные в файле, позволяет изучать, извлекать, сравнивать любые версии файлов на протяжении всего времени работы с ними;
- управление конфигурациями. Механизм, реализующий управление конфигурацией программного продукта на протяжении

всего времени существования проекта. Он основан на управлении версиями, обеспечивает надежную среду для управления разработкой ПО и ведения необходимых процессов [9, с. 457]. П. Гудлиф приводит следующее определение управления конфигурациями: «Порядок установления конфигурации системы в дискретные моменты времени с целью систематического контроля за изменениями в этой конфигурации и сохранения целостности и возможности слежения за этой конфигурацией на протяжении всего времени существования системы» [9, с. 456];

- контроль доступа. Разграничение прав доступа к разным частям базового кода. Если права администратора доступны всем раз-

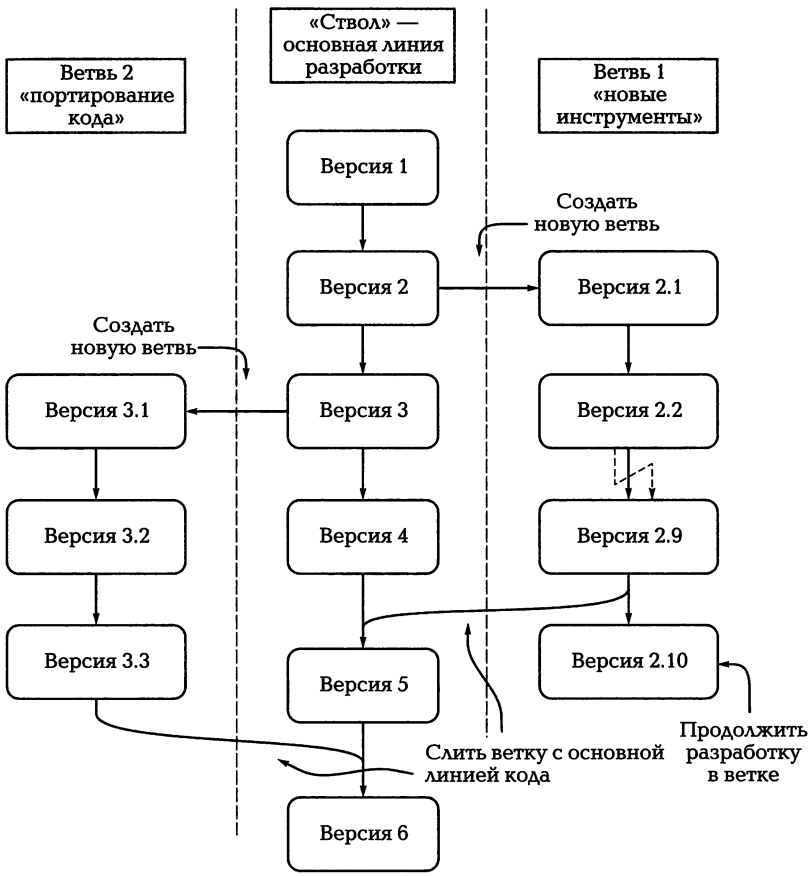


Рис. 1.19. Механизм ветвления проекта в системе контроля версий

работчикам, то это может привести к неприятным последствиям. У СКВ должен быть администратор;

- ветвление. Возможность создания нескольких параллельных ветвей разработки файла или группы файлов.

Рассмотрим механизм ветвления (рис. 1.19). Существует одна основная линия разработки кода, называемая стволом. Для доработки структурного компонента ствола может создаваться несколько веток, при этом во время доработки компонента его код в стволе остается неизменным. Когда код структурного компонента доработан и проверен, результат возвращается в ствол проекта и соответствующий файл или участок кода обновляется.

Механизм обеспечивает возможность создания нескольких параллельных потоков разработки файла или группы файлов. Разработка в отдельной ветке не наносит ущерба базовому коду. Разработчик в своем пространстве дорабатывает код и возвращает только отлаженный, работоспособный вариант. Механизм обеспечивает сопровождение старой версии проекта при параллельной работе над новой версией.

1.7. ОБРАТНОЕ ПРОЕКТИРОВАНИЕ

Обратное проектирование, или **реверс-инжиниринг**, является одним из способов исследования ПО, для которого отсутствуют документация, описания, спецификации и исходный код, т. е. оно представлено в виде исполняемых файлов.

Такая ситуация далеко не уникальна. Программное обеспечение является интеллектуальной собственностью организации-разработчика, и все проприетарное (коммерческое, платное) ПО в настоящее время поставляется с минимальным набором сопровождающих документов.

Возможны ситуации, при которых возникает необходимость понять алгоритм функционирования ПО, выведенного из эксплуатации. Во всех сферах науки и техники сложно начинать разработку чего-то принципиально нового без запаса исследовательского опыта, поэтому возникает необходимость черпать этот опыт из устаревших разработок.

Ежедневно злоумышленники разрабатывают вредоносное ПО и компьютерные вирусы, за которыми не успевают средства антивирусной защиты. К сожалению, некоторые области деятельности человека настолько чувствительны к потрясениям, что оставлять ситуацию в этой сфере без контроля чревато большими финансо-

выми и репутационными потерями, например в банковской деятельности. В некоторых случаях из-за широкого применения цифровых технологий в автоматизации производства и управлении транспортом потери могут быть фатальными, в том числе затрагивающими жизнь человека. Поэтому аналитики вредоносного ПО обращаются к методам реверс-инжиниринга с целью определить характер воздействия вредоносного ПО на ИС, создать сигнатуры для антивирусных баз данных и сформировать правила для систем предупреждения о вторжении.

Естественно, стоит указать и не совсем законные, а зачастую и уголовно наказуемые сферы деятельности, касающиеся пиратства в сфере информационных технологий и ведения промышленного шпионажа как в коммерческих целях, так и в целях специальных служб на уровне государств.

Независимо от вида процессорного устройства, управляющего средством вычислительной техники (ВТ) или интеллектуальным прибором, язык исполняемого процессором набора команд ориентирован на его архитектуру. Если бы разработчикам ПО для написания программы приходилось тщательно изучать принципы аппаратного устройства и функционирования каждого такого изделия, процесс программирования был бы очень сложным. Однако современные операционные системы предоставляют возможность применять инструментарий и языки высокого уровня, позволяющие пользоваться понятной семантикой такого языка программирования. Специальные программы-компиляторы преобразуют исходный код языка высокого уровня в код языка низкого уровня, а по сути — в набор исполняемых процессором машинных команд.

Таким образом, перед аналитиками стоит достаточно сложная задача, обратная процессу проектирования (отсюда и название сферы деятельности). Сложность задачи обусловлена рядом факторов: трудностью разделения между собой команд и данных при отсутствии имен переменных и функций в исполняемом коде, сложностью процесса дизассемблирования, связанной с нелинейностью алгоритмов используемого ПО, усложнением анализа за счет применения разработчиками методов защиты и обфускации разрабатываемых программ.

Методы обратного проектирования разделяются на статическое исследование (без запуска исследуемой программы на выполнение) и динамический анализ (с запуском на выполнение). Соответственно при этом используются специальные инструментальные средства. Для статического анализа применяются различные

виды дизассемблеров, предоставляющих возможности отображения команд ассемблера, анализа дампа памяти, графического представления распознанных участков кода в виде графов и т.д. Для динамического анализа используются программы-отладчики, позволяющие осуществлять процесс трассировки исполняемого кода, т.е. пошаговое выполнение команд языка низкого уровня одну за одной.

Результатом проведения анализа исполняемого кода программы методами обратной разработки становится, как правило, неполное восстановление кода программы, понимание принципов ее функционирования и построения ее алгоритма. Более подробно методы реверс-инжиниринга рассмотрены в гл. 3.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие роли возможны в команде проекта?
2. Нужна ли команда при разработке проекта?
3. Команда программистов задерживает сроки разработки проекта. Чьей задачей будет общение с заказчиком? Назовите роль.
4. Какие инструменты может использовать команда программистов для успешной работы?
5. Для чего нужны системы контроля версий?
6. Назовите типы систем контроля версий.
7. Зачем нужна база данных ошибок?
8. Группа программистов использует распределенную систему контроля версий. Во время работы членов команды над проектом произошла потеря данных в центральном репозитории. Можно ли восстановить данные?
9. Группа программистов использует блокирующую систему контроля версий. Один из них вносит изменения в файл `tt.db`. Будет ли этот файл доступен для записи? Будет ли этот файл доступен для чтения?
10. Программист Иванов Иван сказал, что `GIT` — централизованная система контроля версий. Прав ли он?
11. Что такое ревьюирование кода?
12. Какова цель ревьюирования?
13. Назовите синонимы словосочетания «ревьюирование кода».
14. Кто должен присутствовать в группе по ревьюированию кода?
15. Какие недостатки возможны при самостоятельном ревьюировании кода?
16. Какие задачи может решить ревьюирование кода?

17. Попробуйте найти задачи, которые решает ревьюирование кода и которые не указаны в данном учебнике.
18. Какие методы ревьюирования кода возможны?
19. Как вы считаете, нужно ли делать ревьюирование кода или это потеря времени? Обоснуйте ответ.
20. Как вы считаете, как часто необходимо делать ревьюирование кода? Обоснуйте ответ (для ответа можно изучить специальные источники литературы [26]).

ПРАКТИЧЕСКИЕ ЗАДАНИЯ

Задание 1.1. Создание и изучение возможностей репозитория проекта

Цель и задачи исследования. *Цель* — ознакомиться с возможностями СКВ `Git` и научиться создавать репозиторий.

Задачи. В ходе практического занятия будет изучено создание репозитория с помощью `Git`. `Git` — это набор консольных утилит, которые отслеживают и фиксируют изменения в файлах (чаще всего речь идет об исходном коде программ, но можно использовать его для любых файлов). С его помощью можно откатиться на более старую версию вашего проекта, сравнивать, анализировать, сливать изменения и многое другое, т. е. проводить контроль версий (или управление версиями). Существуют различные системы для контроля версий, например `SVN`, `Mercurial`, `Perforce`, `CVS`, `Bitkeeper` и др.

`Git` является распределенным, т. е. не зависит от одного центрального сервера, на котором хранятся файлы. Вместо этого он работает полностью локально, сохраняя данные в папках на жестком диске, которые называются репозиторием. Тем не менее вы можете хранить копию репозитория онлайн, это облегчает работу над одним проектом для нескольких людей. Для этого используются различные сайты, например github и bitbucket.

Практическая часть. Исследование выполняется в несколько этапов.

1. Первоначально необходимо установить `Git` на компьютер.

Установка `Git`:

- для Linux — нужно открыть терминал и установить приложение при помощи пакетного менеджера вашего дистрибутива;
- Ubuntu — команда будет выглядеть следующим образом:

```
sudo apt-get install git
1

sudo apt-get install git
```

- Windows — рекомендуется `Git for Windows`, так как он содержит и клиент с графическим интерфейсом, и эмулятор `bash`;

- OS X — проще всего воспользоваться `homebrew`. После его установки запустите в терминале:

```
brew install git
1

brew install git
```

Лучше установить `Git` из исходных кодов, поскольку так можно получить самую свежую версию. Каждая новая версия `Git` обычно включает полезные улучшения пользовательского интерфейса. Для установки `Git` понадобятся библиотеки, от которых он зависит: `curl`, `zlib`, `openssl`, `expat` и `libiconv`. Например, если в вашей системе менеджер пакетов — `yum` (`Fedora`) или `apt-get` (`Debian`, `Ubuntu`), можно воспользоваться следующими командами, чтобы разрешить все зависимости:

```
$ yum install curl-devel expat-devel gettext-devel \
openssl-devel zlib-devel

$ apt-get install libcurl4-gnutls-dev libexpat1-dev
gettext \
libz-dev libssl-dev
```

Установив все необходимые библиотеки, можно идти дальше и скачать последнюю версию с сайта `Git`: http://git-scm.com/download.

Теперь скомпилируйте и установите:

```
$ tar -zxf git-1.7.2.2.tar.gz
$ cd git-1.7.2.2
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

После этого вы можете скачать `Git` с помощью самого `Git`, чтобы получить обновления:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

Установка в Linux. Если нужно установить `Git` под Linux как бинарный пакет, это можно сделать, используя обычный менеджер пакетов вашего дистрибутива.

Установка в Ubuntu. Если имеется дистрибутив, основанный на `Debian`, например Ubuntu, воспользуемся `apt-get`:

```
$ apt-get install git
```

Установка в Windows. Скачайте exe-файл инсталлятора со страницы проекта на `GitHub` и запустите его: http://msysgit.github.com/.

После установки у вас появится как консольная версия (включающая SSH-клиент, который пригодится позднее), так и стандартная графическая.

Git необходимо использовать только из командной оболочки, входящей в состав msysGit, потому что так вы сможете запускать сложные команды, приведенные в примерах. Командная оболочка Windows использует иной синтаксис, из-за чего примеры в ней могут работать некорректно.

Для начинающих разработчиков клиент с графическим интерфейсом (например, GitHub Desktop и Sourcetree) будет полезен, но тем не менее знать команды очень важно.

Установка на Mac. Лучше всего использовать графический инсталлятор Git, который можно скачать со страницы на SourceForge: http://sourceforge.net/projects/git-osx-installer/.

Еще один способ установки Git — через MacPorts. Команда для установки:

```
$ sudo port install git-core +svn +doc +bash_completion
+gitweb
```

Для использования вместе с Git репозитариев Subversion понадобится дополнение +svn.

2. После установки нужно добавить настройки. Настроим самые важные опции — имя пользователя и адрес электронной почты.

Откройте терминал и запустите команды:

```
git config --global user.name "My Name"
git config --global user.email myEmail@example.com
1
2
git config --global user.name "My Name"
git config --global user.email myEmail@example.com
```

Теперь каждое действие будет отмечено именем и почтой. Таким образом, пользователи всегда будут в курсе, кто за какие изменения отвечает, это обеспечивает порядок.

3. Создать новый репозиторий. Как мы уже отмечали, Git хранит свои файлы и историю прямо в папке проекта. Чтобы создать новый репозиторий, нужно открыть терминал, зайти в папку проекта и выполнить команду init. Это включит приложение в этой конкретной папке и создаст скрытую директорию .git, где будут храниться история репозитория и настройки.

Создайте на рабочем столе папку под названием git_exercise. Для этого в окне терминала введите:

```
$ mkdir Desktop/git_exercise/
$ cd Desktop/git_exercise/
$ git init
1
2
3
$ mkdir Desktop/git_exercise/
$ cd Desktop/git_exercise/
$ git init
```

Командная строка должна содержать, например, следующее:

```
Initialized empty Git repository in /home/user/Desktop/
git_exercise/.git/
1

Initialized empty Git repository in /home/user/Desktop/
git_exercise/.git/
```

Это значит, что репозиторий был успешно создан, но пока он пуст. Теперь создайте текстовый файл под названием `hello.txt` и сохраните его в директории `git_exercise`.

4. Определить состояние репозитория. `Status` — это еще одна важнейшая команда, которая показывает информацию о текущем состоянии репозитория: актуальна ли информация на нем, нет ли чего-то нового, что поменялось и т. д. Запуск `git status` на нашем недавно созданном репозитории выдаст:

```
$ git status
On branch master
Initial commit
Untracked files:
(use "git add ..." to include in what will be committed)
hello.txt
1
2
3
4
5
6

$ git status
On branch master
Initial commit
Untracked files:
(use "git add ..." to include in what will be committed)
hello.txt
```

Сообщение говорит о том, что файл `hello.txt` неотслеживаемый. Это значит, что файл новый и система еще не знает, нужно ли следить за изменениями в файле или можно просто игнорировать его. Чтобы начать отслеживать новый файл, нужно его специальным образом объявить.

5. Подготовить файлы. В `Git` есть концепция области подготовленных файлов. Можно представить ее как холст, на который наносят изменения, нужные в коммите.

Коммит — состояние репозитория в определенный момент времени. Первоначально он пустой, но затем мы добавляем на него файлы (или части файлов, или одиночные строки) командой `add` и наконец коммитим

все нужное в репозиторий (создаем слепок нужного нам состояния) командой `commit`.

В нашем случае у нас только один файл, так что добавим его:

```
$ git add hello.txt
1
$ git add hello.txt
```

Если нам нужно добавить все, что находится в директории, мы можем использовать:

```
$ git add -A
1

$ git add -A
```

Проверим статус снова, на этот раз мы должны получить другой ответ:

```
$ git status
On branch master
Initial commit
Changes to be committed:
(use "git rm --cached ..." to unstage)
new file: hello.txt
1
2
3
4
5
6

$ git status
On branch master
Initial commit
Changes to be committed:
(use "git rm --cached ..." to unstage)
new file: hello.txt
```

Файл готов к коммиту. Сообщение о состоянии также говорит о том, какие изменения относительно файла были проведены в области подготовки, в данном случае это новый файл, но файлы могут быть модифицированы или удалены.

6. Зафиксировать изменения (коммит). Чтобы зафиксировать изменения, нужно хотя бы одно изменение в области подготовки (мы только что создали его при помощи `git add`), после которого мы можем коммитить:

```
$ git commit -m "Initial commit."
1

$ git commit -m "Initial commit."
```

Эта команда создаст новый коммит со всеми изменениями из области подготовки (добавление файла `hello.txt`). Ключ `-m` и сообщение `"Initial commit."` — это созданное пользователем описание всех изменений, включенных в коммит. Считается хорошей практикой делать коммиты часто и всегда писать содержательные комментарии.

Примечание. После выполнения всех действий следует представить отчет в виде скриншотов всех команд. Результаты этой практической работы будут использованы в качестве данных для задания 2.4 (см. гл. 2).

Задание 1.2. Экспорт настроек в командной среде

Цель и задачи исследования. *Цель* — получение навыков управления параметрами настройки `IDE Visual Studio`.

Задачи. Студенту необходимо изменить настройки среды разработки `Visual Studio`, осуществить их экспорт в файл и импортировать на другую вычислительную машину. В ходе выполнения работы следует делать снимки экрана, после чего оставить отчет в виде инструкции по выполнению необходимых операций для экспорта и импорта настроек среды `Visual Studio`.

Практическая часть. Исследование выполняется в несколько этапов.

1. С учетом того что `IDE Visual Studio` обладает достаточно обширными возможностями в рамках настройки параметров, обеспечивающих удобство работы, студенту предлагается выполнить настройку наиболее заметных из них, отвечающих за внешний вид среды разработки, что, в свою очередь, позволит с легкостью фиксировать результат работы. Для этого необходимо осуществить следующие действия:

- в строке меню необходимо выбрать `Инструменты → Параметры`;
- в списке параметров необходимо выбрать `Окружение → Общие`;
- для изменения цветовой схемы среды в окошке с выпадающим списком `Цветовая схема` необходимо выбрать соответствующую вкусу схему и нажать кнопку `Ок`;
- для изменения настроек применяемых шрифтов необходимо выполнить первые два пункта и выбрать в списке пункт `Шрифты и цвета`, выполнить необходимые настройки и нажать кнопку `Ок`.

2. Для экспорта параметров среды в файл необходимо выбрать пункт главного меню `Средства`, подпункт `Импорт и экспорт параметров`. После запуска мастера импорта и экспорта параметров нужно выбрать, установив соответствующий флажок, вариант действий (импорт или экспорт) и нажать кнопку `Далее`. Выполняя действия, предлагаемые мастером, пометить необходимые для импорта/экспорта параметры и записать их в файл в указанное место.

3. Сохранить файл с параметрами и перенести его на другую машину с установленной средой `Visual Studio`.

4. Вызвав мастер импорта и экспорта параметров, импортировать настройки в новую среду разработки.

5. Оформить отчет, описав свои действия и разместив соответствующие снимки экрана.

Задание 1.3. Сравнительный анализ офисных пакетов

Условия исследования. Представим ситуацию: в компании 100 персональных компьютеров, на 15 из них установлена операционная система Linux, на остальных — Windows 10. Необходимо принять решение об установке офисных пакетов на персональные компьютеры сотрудников для работы с документацией. Наиболее актуальными программами являются текстовые и табличные редакторы, программы создания и просмотра презентаций. В компании есть внутренняя, входящая и исходящая документация. В связи с этим важны возможности обмена данными между офисными пакетами, сохранения данных и их просмотра в различных форматах. Сотрудников компании заинтересовали офисные пакеты Microsoft Office и Open Office.

Программное обеспечение, необходимое для работы: пакет программ Microsoft Office, пакет программ Open Office, доступ в Интернет.

Цель и задачи исследования. *Цель* — произвести сравнительный анализ офисных пакетов Microsoft Office и Open Office.

Задачи. Студенту необходимо:

- предоставить обзорную информацию о ПО;
- произвести сравнительный анализ офисных пакетов, в том числе функциональный анализ их программных продуктов для работы с текстом, таблицами и презентациями;
- на основании полученных результатов анализа сделать выводы и разработать рекомендации по установке офисных пакетов на имеющиеся компьютеры.

Результатом работы являются сравнительный анализ офисных пакетов и рекомендации по установке ПО на персональные компьютеры сотрудников компании, оформленные в файле текстового редактора одного из исследуемых пакетов программ. Предпочтительнее оформить документ средствами редактора Writer пакета программ Open Office. В связи с тем что документ будет многостраничным, необходимо вставить нумерацию страниц и создать автоматическое оглавление.

Практическая часть. Вам будет предложен список заданий, направленных на решение задач исследования.

1. Найдите информацию:

- о фирмах — производителях офисных пакетов;
- дате выхода и названии последней версии офисного пакета;
- стоимости последних версий офисных пакетов;
- требованиях к аппаратному обеспечению последних версий офисных пакетов;
- операционных системах, для которых возможна установка исследуемых офисных пакетов.

2. Укажите данные о версиях офисных пакетов, установленных на вашем компьютере.

3. Проведите сравнительный анализ базовой комплектации офисных пакетов.

4. Проведите сравнительный анализ форматов файлов, с которыми работают соответствующие программы офисных пакетов для работы с текстом, табличными данными и презентациями.

5. Выясните, что такое файлы открытого формата, относятся ли к файлам открытого формата файлы приложений Microsoft Office и Open Office.

6. Определите, возможен ли обмен данными между приложениями Microsoft Office и Open Office. Если да, то каким образом он осуществляется.

7. Изучите и проанализируйте возможность обмена данными между приложениями офисных пакетов и сторонними приложениями. Узнайте, в каких форматах можно экспортировать данные и какие объекты можно импортировать.

8. Проведите сравнительный анализ интерфейсов программ Word и Writer, Excel и Calc, Power и Point Impress.

9. Разработайте критерии для сравнительного функционального анализа программ (см. табл. 1.2) Word и Writer, Excel и Calc, Power и Point Impress.

10. Результаты анализа программ с аналогичным функционалом представьте в табличном виде (см. табл. 1.2). После выполнения заданий 7 и 8 вы получите три таблицы.

11. Сделайте выводы по результатам анализа офисных пакетов.

12. Разработайте рекомендации по установке офисных пакетов для компании.

Этот список не является окончательным и неизменным. В ходе исследования его можно дополнить другими необходимыми пунктами.

Задание 1.4. Сравнительный анализ браузеров

Условия исследования. Программное обеспечение, необходимое для работы: `AppTimer`, браузеры для тестирования, доступ в Интернет.

Цель и задачи исследования. *Цель* — произвести сравнительный анализ трех выбранных браузеров по определенным критериям.

Задачи. Направления анализа:

- поддержка браузерами HTML5 и CSS3;
- скорость запуска браузера;
- анализ производительности браузера с помощью https://browserbench.org или других аналогичных сервисов;
- интерфейс браузеров.

Результат работы должен быть представлен в виде отчета (файл MS Word), содержащего информацию о результатах исследования браузеров, анализ полученных результатов и выводы. Для наглядного представления результатов можно использовать изображения, диаграммы, графики, схемы, таблицы.

Практическая часть. Исследование выполняется в несколько этапов.

1. Ознакомьтесь с анализом браузеров, изучив статью: https://os-chrome.ru/reviews/sravnenie-brauzerov/. Выберите три любых браузера, которые будет оценивать группа из 7—10 человек, чтобы в дальнейшем

была возможность совместной оценки интерфейса программных продуктов. Произведите независимую оценку браузеров.

2. Запустите каждый выбранный браузер. Откройте в каждом из них сайт http://html5test.com. Проанализируйте полученные результаты. Сделайте сравнительный анализ и выводы о поддержке браузерами HTML5. Отразите результаты вашего анализа и полученные выводы в отчете.

3. Запустите каждый из выбранных браузеров. Откройте в каждом из них интернет-ресурс http://css3test.com. Проанализируйте полученные результаты. Сделайте сравнительный анализ и выводы о поддержке браузерами CSS3. Отразите результаты вашего анализа и полученные выводы в отчете.

Получить информацию о поддержке тега или стиля популярными браузерами можно, используя ресурс https://caniuse.com. Для знакомства с ресурсом откройте любой из браузеров, запустите указанный сайт и ответьте на несколько простых вопросов.

В поле после предложения «Can i use» введите `display:flex`. Какие браузеры лучше поддерживают указанный стиль и связанную с ним группу стилей? Введите в поле название одного из семантических тегов `section`. Начиная с какой версии Firefox и Internet Explorer происходит поддержка данного тега? Ответы на вопросы сохраните в отдельном файле.

4. Откройте программу `AppTimer`, оцените скорость запуска каждого из выбранных вами браузеров. Протестируйте скорость первого запуска браузера после загрузки операционной системы и нескольких его повторных запусков. Проанализируйте полученные результаты. Сделайте выводы. Внесите информацию в отчет.

5. Оцените производительность браузеров, используя ресурс https://browserbench.org. Каждый тест потребует определенного количества времени. В связи с тем что время занятия ограничено, протестируйте каждый браузер с помощью соответствующих тестов по одному разу. Для получения более точных результатов тестирования вы можете запустить один и тот же тест браузера несколько раз на своем домашнем компьютере. Поскольку дома и на занятии у вас различные аппаратное и программное обеспечение и скорость загрузки интернет-страниц, ваши результаты тестирования дома и на занятии могут отличаться.

Результаты тестирования браузеров, сравнительный анализ и выводы сохраните в вашем отчете.

6. Опишите ваши впечатления об интерфейсе браузеров, его достоинства и недостатки. Оцените интерфейс каждого браузера по шкале от 1 до 10 по следующим правилам:

- оценку выставляет каждый участник группы, тестировавший три одинаковых браузера;
- количество полученных оценок в зависимости от количественного состава групп будет от 7 до 10 для каждого браузера. Минимальная и максимальная оценки для каждого браузера не учитываются. Среди остальных оценок находится среднее арифметическое значение, которое вы будете использовать в своем отчете;
- в своей оценке постарайтесь оценить именно интерфейс браузера, без учета вашего отношения к нему, обусловленного другими критериями.

Например, подход «поставлю 2 браузеру, так как он медленно работает и не поддерживает семантические теги» будет неверен. Оценивается именно интерфейс браузера.

7. На основе полученных результатов исследования проведите сравнительный анализ браузеров и сделайте выводы.

8. До конца оформите отчет со сравнительным анализом браузеров.

Задание 1.5. Сравнительный анализ средств просмотра видео

Условия исследования. Программное обеспечение, необходимое для работы:

- выбранные обучаемым средства просмотра видео;
- видеофайлы;
- доступ в Интернет.

Цель исследования. *Цель* — произвести сравнительный анализ четырех выбранных средств просмотра видео по определенным критериям.

Практическая часть. Исследование выполняется в два этапа.

1. Предварительно необходимо выработать критерии анализа средств просмотра видео, руководствуясь материалом подразд. 1.4. Рекомендуемые направления анализа:

- функциональные возможности ПО;
- интерфейс (удобство использования, простота освоения, дизайн);
- наличие поддержки ПО (руководство пользователя и система помощи);
- требования к аппаратному обеспечению для корректной работы;
- надежность ПО — бесперебойное выполнение всех заявленных функциональных возможностей, корректное завершение работы при возникновении различных возможных ошибок (по вине или не по вине пользователя).

Рекомендуется выбирать и скачивать бесплатные видеоплейеры, например представленные в https://softprime.net/video/videopleery/.

Примеры выбора параметров функциональных возможностей:

- воспроизведение различных форматов файла;
- скорость загрузки файла;
- возможность корректировки параметров воспроизведения;
- возможность выбора скорости прокрутки;
- настройки по предпочтениям пользователя;
- настройка звука;
- совместимость с различными браузерами и т. д.

2. Просмотреть небольшие видеофайлы и фрагменты видеофайлов с помощью выбранных средств просмотра. Результат работы должен быть представлен в виде отчета (файл MS Word), содержащего информацию о результатах исследования, их анализ и выводы. Для наглядного представления полученных результатов можно использовать изображения, диаграммы, графики, схемы, таблицы и скриншоты.

Задание 1.6. Обратное проектирование алгоритма

Условия исследования. Для выполнения работы необходимо установить свободно распространяемую версию программы IDA PRO. Следует заметить, что данное ПО относится к проприетарному, поэтому бесплатная версия не может использоваться в коммерческих целях и ее функциональные возможности ограничены, она работает только с файлами для 64-разрядных систем и не поддерживает возможность встраивания дополнительных инструментов.

Перед выполнением работы необходимо ознакомиться с основными методами обратного проектирования (базовым статическим и расширенным статическим), а также понимать назначение и отличия между линейным и рекурсивным алгоритмами анализа кода программ при проведении дизассемблирования.

Цель исследования. *Цель* — знакомство с интерфейсом и возможностями программного комплекса IDA PRO, сочетающего в себе свойства отладчика и дизассемблера, работающего с файлами для 32/64-разрядных систем. Выполнение задания демонстрирует возможности программы и формирует общее представление о технологиях обратного проектирования, в частности статического анализа.

Практическая часть. Исследование выполняется в несколько этапов.

1. Выполнить установку необходимого ПО. Запуск программы IDA помимо информации о лицензии начнется с отображения диалога, предлагающего три способа попасть в рабочее окружение.

2. Для начала рекомендуется выбором пункта New запустить помощника, который будет помогать при выборе файла для анализа, так как пользователь должен точно определить тип файла, который он хочет открыть. При помощи стандартного диалога Открыть файл осуществляется выбор файла. Перед анализом и отображением появляются один или более дополнительных диалогов, которые позволяют указать специфические опции для анализа до непосредственной загрузки файла.

3. После загрузки исследуемого файла необходимо ознакомиться со следующими основными компонентами рабочей области:

- область панели инструментов, которая содержит основные элементы меню и кнопки быстрого доступа к главным используемым операциям;
- окно дизассемблирования, предназначенное для вывода данных. Здесь присутствуют два режима вывода информации — в виде графа (по умолчанию) и в форме листинга. В режиме графа IDA отображает схематический граф одной функции. Изучение графов помогает визуально понять процесс выполнения функции. Для переключения между режимами графа и листинга используется клавиша Пробел;
- окно сообщений для предоставления информации, генерируемой IDA;
- окно строк фактически является аналогом утилиты Strings, отображает список строк, извлеченных из двоичного файла, а также их адреса. При использовании перекрестных ссылок окно строк позволяет быстро обнаружить интересующую строку и отследить ссылку на нее;
- окно экспортов предназначено для перечисления входных точек файла. Они включают в себя входную точку исполнения программы, как

указано в заголовочном разделе, а также все функции и переменные, которые файл экспортирует для использования другими файлами;

- окно функций перечисляет все функции, распознанные IDA в базе данных;
- окно структур используется для отображения распознанных IDA сложных структур данных, таких как structs и unions в Си;
- окно импортов перечисляет все функции, импортируемые анализируемым двоичным файлом. Окно импортов необходимо при использовании двоичным файлом общих библиотек. Записи в окне импортов включают в себя имя импортируемой функции или данных и имя библиотеки, их содержащей.

4. Ознакомившись с основными окнами интерфейса, студентам предлагается через окно импорта определить точку входа в исследуемую программу и, используя режим представления в виде графа, осуществить перемещение между вызываемыми функциями, исследуя общий алгоритм функционирования программы через вызов функций и осуществляя переключение в режим листинга.

5. В результате проделанной работы следует написать отчет, разместив в нем скриншоты окон программы IDA PRO, и ответить на следующие контрольные вопросы:

- Для чего предназначено обратное проектирование?
- Для чего предназначены программы-дизассемблеры?
- Для чего применяется программа IDA PRO при обратном проектировании?
- Что отображает окно дизассемблирования программы IDA PRO в режиме графа?
- Что может дать исследование строк как символов в кодировке Unicode при исследовании программ?

Также студенту необходимо:

- объяснить предназначение программ отладчиков;
- описать основные методы обратного проектирования;
- дать определения линейному и рекурсивному методам при статическом анализе кода исполняемых файлов.

6. Сделать выводы о проделанной работе.

Глава 2

ОРГАНИЗАЦИЯ РЕВЬЮИРОВАНИЯ. ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА РЕВЬЮИРОВАНИЯ

2.1. УТИЛИТЫ ДЛЯ REVIEW: ОБЗОР

Как отмечалось в гл. 1, ревьюирование представляет собой просмотр исходного кода ПО с целью обнаружения ошибок и дефектов, возможно, до того, как это ПО заработает. Ревьюирование предназначено для выявления таких ошибок, как неспособность выполнять то или иное требование спецификации или ее неправильное понимание, а также алгоритмических ошибок в реализации.

Тестовый прогон обеспечивает тестирование ПО в процессе выполнения программы. Осуществляя прогон программы, тестировщик стремится определить, способна ли программа вести себя в соответствии со спецификацией. Он должен выбрать наборы входных данных, определить соответствующие им правильные наборы выходных данных и сопоставить их с реально получаемыми выходными данными.

Утилита — компьютерная программа, расширяющая стандартные возможности оборудования и операционных систем и выполняющая узкий круг специфических задач.

Утилиты предоставляют доступ к возможностям (параметрам, настройкам, установкам), недоступным без их применения, либо делают процесс изменения некоторых параметров проще (автоматизируют его). Утилиты зачастую входят в состав операционных систем или идут в комплекте со специализированным оборудованием. В данном разделе приведен обзор перспективных в настоящее время утилит, входящих в состав ПО для анализа кода. Поскольку утилиты входят в состав соответствующего

ПО — анализаторов кода, отладчиков и т.д., рассмотрим также некоторые из них.

Одной из наиболее перспективных разработок является созданная учеными из Цюриха **утилита DeepCode**. Программа применяется для обнаружения и исправления ошибок в коде. Основой системы послужил искусственный интеллект, она включает в себя более 250 тыс. алгоритмических правил.

Утилита работает по принципу плагина `Grammarly`. Напомним, что плагин (англ. *plug in* — подключать) — это независимо компилируемый программный модуль, динамически подключаемый к основной программе и предназначенный для расширения и (или) использования ее возможностей.

Считывание кода осуществляется из любых репозиториев `GitHub`, находящихся в общем или приватном доступе. `GitHub` — это веб-платформа для управления версиями и совместной работы для разработчиков ПО. Ресурс основан на `Git` — системе управления исходным кодом, созданной Линусом Торвальдсом для ускорения разработки ПО. После анализа файлов `DeepCode` дает подсказки, содержащие информацию о предполагаемых исправлениях или улучшениях кода, при этом ПО основывается на собственных алгоритмах. Исправления могут быть, например, разновидностями написания одного и того же присваивания, но могут выявлять и скрытые ошибки в коде. Так, строка `name: String` преобразована в `name: {type: String}`). Подсказки носят довольно точный характер, поскольку основываются на статистических данных, собранных в результате анализа большого количества программных кодов. При этом `DeepCode` представляет собой не только отладчик. Система сравнивает одни и те же строки кода в разных вариантах и предлагает пользователю наиболее эффективную реализацию.

Программное обеспечение анализирует миллионы репозиториев и фиксирует изменения, которые вносят разработчики. Далее происходит обучение системы, основанное на этих данных, чтобы предоставить пользователю предложения по улучшению качества кода.

Еще одной популярной утилитой для выявления ошибок в коде является вышедшая совсем недавно **`Node.js`**. Она предназначена для поиска уязвимостей в модулях.

Преимущества использования утилиты для анализа кода заключаются в следующем:

- улучшается качество кода;
- находятся «глупые» ошибки или опечатки;
- облегчается совместная работа с кодом;
- повышается степень совместного владения кодом;

- стиль написания кода становится единым;
- код приводится к единому стилю написания.

Такого рода ПО хорошо подходит для выработок навыков программирования у начинающих, обмена знаниями с более опытными разработчиками.

Далее следует выделить **PVS-Studio** — инструмент для выявления ошибок и потенциальных уязвимостей в исходном коде программ, написанных на языках C, C++ и C#, поскольку в нем также предусмотрены специальные утилиты для анализа кода.

Прежде всего PVS-Studio предназначен для выполнения статического анализа кода и генерирует отчет, помогающий программисту находить и устранять ошибки. PVS-Studio выявляет различные ошибки кода, но в первую очередь опечатки и последствия неудачного копирования фрагментов. Примеры таких ошибок: V501, V517, V522, V523, V3001.

Отметим, что если регулярно использовать статистический анализ, многие ошибки будут выявлены и устранены на ранних этапах. Анализатор может запускаться ночью на сервере и сообщать о подозрительных местах в новом коде. В идеальном случае ошибки могут быть обнаружены и исправлены еще до попадания в репозиторий. PVS-Studio может автоматически запускаться сразу после компилятора на файлах, в которые только что внесли изменения. Программное решение возможно применять в таких операционных системах, как Windows, Linux и macOS.

PVS-Studio можно интегрировать в среду разработки Visual Studio 2010—2017. При использовании этой среды разработки достаточно зайти в меню плагина PVS-Studio и выбрать команду Проверить проект. Но часто требуется интеграция PVS-Studio в сборочную систему, в том числе какую-либо нестандартную. Это достаточно специфическая проблема, описанная в специальной литературе.

Как уже отмечалось, в PVS-Studio для Windows и Linux предусмотрены специальные утилиты. В первую очередь это утилиты, собирающие информацию о запусках компилятора. Таким образом возможно быстро выполнить анализ проекта, который собран любым способом. Возможности анализатора легко изучить, не тратя время на его интеграцию с makefile или сборочным скриптом, просмотрев описание утилиты Standalone (Windows) и pvs-studio-analyzer (Linux/macOS).

Интересной утилитой является **BlameNotifier**. Инструмент позволяет рассылать письма разработчикам об ошибках, которые PVS-Studio нашел во время ночного прогона.

Остановимся еще немного на технологии работы PVS-Studio. Для поиска мест в исходном коде, которые похожи на известные шаблоны кода с ошибкой, применяется сопоставление с шаблоном (pattern-based analysis) на основе абстрактного синтаксического дерева. Полная информация о всех переменных и выражениях, встречающихся в коде, возможна благодаря выводу типов (type inference) на основе семантической модели программы. За вычисление значений переменных, которые могут приводить к ошибкам, отвечает символьное выполнение (symbolic execution). Также производится проверка диапазонов (range checking) значений.

Для вычисления ограничений значений переменных используется анализ потока данных (data-flow analysis). Примером может служить переменная, принимающая значения внутри блоков if/else.

Для предоставления информации об используемых методах применяется аннотирование методов (method annotations), при этом предоставляется больше информации, чем могло бы быть получено путем анализа их сигнатуры.

Приведем еще некоторые возможности и особенности PVS-Studio:

- удобная и простая интеграция с Visual Studio 2010—2017;
- наличие онлайн-справки по всем диагностикам, доступной в самой программе и на сайте, а также документация в pdf-файле;
- сохранение и загрузка результатов анализа — возможность проверить ночью код, сохранить результаты, а утром загрузить их и просмотреть;
- возможность сохранить результаты анализа в формате HTML с полной навигацией по коду;
- запуск из командной строки для проверки всего решения, позволяет интегрировать PVS-Studio в ночные сборки, чтобы утром у всех разработчиков был свежий лог-файл. Лог-файл (logs, или логи) — это текстовый файл, где ведется запись всех событий, которые происходили на сервере (на котором расположен сайт);
- отличная масштабируемость. Поддержка многоядерных и многопроцессорных систем с настройкой количества используемых ядер, поддержка IncrediBuild — инструмента, который применяется разработчиками для распределенной сборки крупных проектов;
- наличие функции Mark as False Alarm (Пометить как ложное срабатывание), позволяющей разметить диагностические сообщения анализатора PVS-Studio, которые являются

«ложными срабатываниями», чтобы не видеть эти сообщения при следующем запуске анализатора;

- интерактивная фильтрация результатов анализа (лога) в окне PVS-Studio: по коду диагностики, по имени файла, по включению слова в текст диагностики;
- большое количество вариантов интеграции в проекты, разрабатываемые под Linux и macOS;
- функция Mass Suppression — позволяет подавить все старые сообщения, чтобы анализатор выдавал ноль срабатываний. К подавленным сообщениям можно вернуться позже. Возможность легко внедрить PVS-Studio в существующий процесс разработки и сфокусироваться на ошибках только в новом коде;
- статистика ошибок в Excel — можно посмотреть темпы правки ошибок, количество ошибок во времени и т.д.;
- автоматическая проверка на наличие новых версий PVS-Studio (как при работе в IDE, так и при ночных сборках);
- использование относительных путей в файлах отчета для возможности переноса отчета на другую машину;
- наличие системы мониторинга компиляции (PVS-Studio Compiler Monitoring, CLMonitoring), предназначенной для «бесшовной» интеграции статического анализа PVS-Studio в любую сборочную систему на ОС семейства Windows. CLMonitoring — проверка проектов, у которых нет файлов Visual Studio (.sln/.vcxproj); если не хватит функциональности CLMonitoring, то можно интегрировать PVS-Studio в любую Makefile-based-систему сборки вручную;
- наличие pvs-studio-analyzer — утилиты, аналогичной CLMonitoring, но работающей под Linux и macOS;
- возможность исключить из анализа файлы по имени, папке или маске, возможность проверять файлы, модифицированные за последние N дней;
- возможность интеграции с SonarQube — открытой платформой для обеспечения непрерывного контроля качества исходного кода.

В заключение отметим, что PVS-Studio поддерживает множество языков и компиляторов, таких как:

- Windows. Visual Studio 2010—2017 C, C++, C++/CLI, C++/CX (WinRT), C#;
- Windows. IAR Embedded Workbench, C/C++ Compiler for ARM C, C++;
- Windows/Linux. Keil µVision, DS-MDK, ARM Compiler 5/6 C, C++;
- Windows/Linux. Texas Instruments Code Composer Studio, ARM Code Generation Tools C, C++;

- Windows/Linux/macOS. Clang C, C++;
- Linux/macOS. GCC C, C++;
- Windows. MinGW C, C++.

2.2. ПРЕДПРОЦЕССИНГ КОДА. ИНТЕГРАЦИЯ В IDE

Интегрированная среда разработки (Integrated Development Environment, IDE) — комплексное средство, включающее все необходимое программисту для создания ПО: редактор исходного кода с подсветкой, средства автоматизации сборки, отладчик, а также большой набор прочих инструментов, упрощающих и ускоряющих процесс работы с кодом.

У большинства IDE имеются базовые анализаторы кода, которые являются первым фильтром после просмотра вручную разработчиком.

Однако код не всегда пишется в IDE, и не всегда разработчик обращает внимание на некоторые предупреждения, к тому же IDE может не быть настроена, например, на количество пробелов или кодировку файлов, которые определяются в `Code Convection` проекта, — правила, которые нужно соблюдать при написании кода. Прочитать такой код весьма затруднительно, как и привести разработанные ранее проекты к единому стандарту. Для предотвращения такой ситуации существует **предпроцессинг кода**.

Выделяются следующие разновидности такого подхода. Первое — интеграция в среду разработки. В данном случае имеются в виду плагины или уже входящие в комплект IDE хендлеры утилит, описанных выше. Такого рода подход хорош для разработки в одиночку. До начала работы со свежеустановленной IDE необходимо свериться с настройками и прописать в них пути к утилитам, выставить необходимый стандарт оформления кода. В случае командной разработки необходимо сохранить экспорт настроек, наиболее часто используемых участниками проекта IDE, в репозиторий, для того чтобы сэкономить время другим разработчикам на настройку проектных практик оформления кода.

Второй метод — `Pre-Commit` (`client side check`). Запускается в конце транзакции, но до `commit`, часто используется для валидации данных, например для проверки непустых лог-собщений. Преимущество этого метода — возможность централизации процесса контроля качества кода. При этом скрипты, которые выполняют проверки, могут лежать в отдельном общедоступном репозитории — ветке или папке.

Далее нужно ознакомиться с определением перехвата в программировании, так как данный термин достаточно часто будет встречаться в дальнейшем.

Перехват (hooking) — технология, позволяющая изменить стандартное поведение тех или иных компонентов ИС.

Конечно, децентрализованные СКВ имеют определенную специфику и в `hooking`.

Для начала стоит определиться, что уже реализовано в проверках на первом уровне. Проверки на клиентской части на этапе перехвата вполне могут перекрывать проверки на уровне IDE. Поэтому лучшим решением будет проверка на этапе `Pre-Update` (до обновления).

На этом уровне возможны следующие проверки:

- валидация JS, CSS (подробнее о валидации — в следующем разделе). В случае ошибки в коде в JS, CSS не нужно будет тратить время и ресурсы сервера контроля версий на то, чтобы поместить ошибки в нем;
- валидация синтаксиса PHP и проверка кода на наличие устаревшего и дублирующего кода. Для больших объемов кода и в случае использования `Git` лучше применять проверку на `Pre-Push` (сообщение, которое встречает пользователя, впервые зашедшего на сайт);
- валидация формата commit-сообщений (в `Git` есть отдельный hook-тип — `commit-msg`).

Уровень строгости проверок может быть задан. Здесь важно соблюдать «золотую середину», и вообще на данном этапе допускается пропуск проверок, но только посредством специальных вставок в коде, которые поддерживают все описанные выше утилиты, например, в `PHP_CodeSniffer` это будет выглядеть так:

```
// @codingStandardsIgnoreStart class MyClassTest
extends \PHPUnit_Framework_TestCase { //
@odingStandardsIgnoreEnd // ... }
```

Эти комментарии потом все равно будут выявлены на этапе ревьюирования, и тогда, возможно, будет пересмотрен участок кода.

Третий метод — `Pre-Update` (до обновления), `server side check` — добавляется на стороне сервера СКВ. Здесь появляется возможность запускать проверки, которые зависят от окружения (особенно если разработка ведется с участием удаленного веб-сервера). Здесь все зависит от объемов тестов: если времени на их обработку уходит много, стоит перенести проверки в `Build`.

Обязательными являются проверки:

- формата сообщений;
- ACL-права на возможность изменять определенные директории отдельными пользователями;
- возможности записи в отдельно взятые ветки.

`During Build check` — проверка, запускаемая во время сборки. Это наиболее объемная проверка, включающая запуск автотестов (Unit, Functional, UI), анализаторы кода (PHPMD, PHPCPD), автодокументирование и прочие процессы, которые занимают время и ресурсы сервера.

Подводя итоги, следует отметить, что необходимо соизмерять трудозатраты на подготовку предпроцессинга кода и организацию процесса ревьюирования с масштабами задачи, временны́ми и человеческими ресурсами.

Описанные утилиты работы с кодом значительно экономят время и облегчают решение задачи.

2.3. ВАЛИДАЦИЯ КОДА НА СТОРОНЕ СЕРВЕРА И РАЗРАБОТЧИКА

Валидация — процесс проверки данных на соответствие различным критериям.

Термин в первую очередь применяется к веб-разработкам, однако термин «валидность кода», как правило, подразумевает следующее: валидный HTML-код — это HTML-код, написание которого соответствует стандарту, указанному в DOCTYPE. Например, если на сайте в DOCTYPE стоит XHTML1.1, то соответствие должно быть именно стандарту XHTML1.1.

Перечислим основные проверки при валидации кода.

Валидация синтаксиса — проверка на наличие синтаксических ошибок. Проверка вложенности тегов подразумевает, что теги должны быть закрыты в обратном порядке относительно их открытия.

Валидация DTD — проверка соответствия кода указанному в Document Type Definition. Она включает проверку названий тегов, атрибутов и «встраивания» тегов (теги одного типа внутри тегов другого типа).

Проверка на посторонние элементы — проверка выявляет все, что есть в коде, но отсутствует в DTD, например пользовательские теги и атрибуты.

Проверить валидность HTML-кода можно, например, по адресу официального вариатора http://validator.w3.org.

Ниже приведен список некоторых распространенных **валидаторов**:

- W3C MarkUp Validation Service — онлайн HTML и XHTML;
- W3C CSS Validation Service — бесплатный, легкий в использовании CSS-валидатор;
- WDG HTML Validator — HTML-валидатор для Windows;
- WAVE 3.0 Accessibility Evaluator;
- WCAG 1.0 Check (SIDAR);
- профессиональные W3C-валидаторы;
- Nu HTML Checker. Checks HTML Documents — проверка HTML-документов;
- Mobile Checker. Be Mobile Friendly — проверка адаптации «для мобильных» (удобства для просмотра на маленьких экранах смартфонов и планшетов);
- CSS Validator. Checks your Cascading Style Sheets (CSS) — проверка каскадных таблиц стилей;
- Link Checker. Checks your Web Pages for Broken Links — проверка веб-страницы на предмет несуществующих ссылок;
- Internationalization Checker. Checks Level of Internationalization — friendliness — проверка уровня адаптации ПО для целевых рынков;
- Markup Validator. Checks the Markup of your Web Documents. (HTML or XHTML) — проверка разметки документа (HTML или XHTML);
- RDF Validator. Checks and Visualizes RDF Documents — проверка на соответствие разработанной консорциумом Всемирной паутины модели для представления данных, в особенности метаданных;
- RSS Feed Validator. Validator for Syndicated Feeds (RSS and Atom Feeds) — проверка каналов новостей и двух связанных технологий — формата для описания ресурсов на веб-сайтах и протокола для их публикации;
- Unicorn. Unified Validator. HTML, CSS, Links & Mobile — проверка HTML, CSS, ссылок и адаптации под мобильные устройства;
- веб-портал imbf.org.

Преимуществами валидного кода являются:

- повышение скорости загрузки;
- облегчение парсинга сайта;
- лучшая индексация поисковыми системами;
- более высокая кроссбраузерность.

По мнению некоторых практикующих разработчиков, преимущества валидности не столь существенны, поскольку скорость

загрузки повышается незначительно, облегчается возможность несанкционированно скопировать контент с помощью автоматизированных средств.

Кроме того, современные браузеры и поисковые системы достаточно хорошо «разбираются» и в не очень валидном коде. При этом на валидацию может быть затрачено существенное время. Интересно отметить, что многие крупные интернет-ресурсы не являются валидными.

Однако валидность кода желательна, прежде всего для обеспечения кроссбраузерности.

Далее остановимся на клиентской и серверной валидации, рассмотрев этот механизм применительно к веб-разработкам.

Клиентская валидация производится в браузере на стороне клиента. Чаще всего логика валидации на стороне клиента реализуется с помощью сценариев JavaScript, которые запускаются внутри браузера.

Клиентская валидация обычно содержит несложные алгоритмы проверки.

Клиентский код не может обратиться к серверным ресурсам (например, к базе данных), поэтому на стороне клиента проверяются простейшие сценарии (такие как проверка длины строки, проверка на вхождение в диапазон и т. д.).

Клиентская валидация может отсутствовать в приложении. Но ее наличие избавляет от необходимости лишних обращений к серверу в случае невыполнения простых условий проверки.

Серверная валидация работает в рамках программного кода, размещенного на стороне сервера. Здесь проверяются всевозможные случаи, в том числе те, которые уже были проверены на стороне клиента.

На стороне сервера могут работать более сложные алгоритмы. Если в браузере у клиента отключено исполнение сценариев JavaScript, то клиентские проверки могут не сработать. Поэтому существует необходимость дублирования проверки сценариев на стороне сервера. Таким образом, наличие клиентской валидации не может гарантировать успешную проверку определенных там ограничений.

Таким образом, серверная валидация — это необходимый процесс, который должен выполняться всякий раз, когда обрабатывается пользовательский ввод, а клиентская валидация — опциональный компонент, который позволяет избавить пользователя от необходимости лишних обращений к серверу и повысить удобство работы с приложением.

2.4. СОВМЕСТИМОСТЬ И ИСПОЛЬЗОВАНИЕ ИНСТРУМЕНТОВ РЕВЬЮИРОВАНИЯ В РАЗЛИЧНЫХ СИСТЕМАХ КОНТРОЛЯ ВЕРСИЙ

Рейтинг СКВ, основанный на опросе ряда респондентов, использующих ту или иную СКВ, приведен в источнике [33].

Среди СКВ при обработке рассматривались также `Bazaar` и `Perforce`.

По экспертным оценкам, главными мотивами использования СКВ были следующие:

- общее, удобное, безопасное хранение исходных кодов с историей изменений;
- коллективное владение кодом;
- разделение задач и функционала приложения внутри команды;
- автоматизация сборок, развертывания и вообще непрерывная интеграция;
- оптимизация командной разработки.

В ходе разработки код проекта активно меняется. При этом нужно вести учет того, что уже было сделано, и согласовывать действия отдельных участников по одновременному изменению кода так, чтобы доработки участников проекта учитывали все ранее сделанные правки других участников. Система контроля версий позволяет автоматизировать этот процесс.

На выбор СКВ влияют следующие факторы:

- поддержка ядра СКВ и ее конкретной реализации;
- знакомство команды разработчиков с тем или иным ПО;
- популярность той или иной системы;
- соответствие возможностей СКВ принятому в команде процессу разработки.

В заключение отметим, что согласно экспертным оценкам одной из самых перспективных и удобных СКВ, в том числе с точки зрения использования инструментов ревьюирования, была признана `Git`. (Подробнее о СКВ см. гл. 1.)

2.5. ОСОБЕННОСТИ РЕВЬЮИРОВАНИЯ В LINUX. НАСТРОЙКИ ДОСТУПА

Особенностью операционной системы Linux является разграничение полномочий. Одна из самых важных функций безопасности в Linux — система прав доступа к файлам.

Обычно в Linux пользователь регистрируется на разных виртуальных консолях под разными именами. Примером может служить следующее: для выполнения административных функций на консоли 1 пользователь регистрируется как `root`, для выполнения прикладной программы на консоли 2 — под именем `prog11`, и т. д. Таким образом, различные пользователи обладают различными индивидуальностями в системе — правами доступа к ресурсам, начальными установками, файлами и т. д., что и порождает особенности ревьюирования.

Если машина включена в сеть, то возможна удаленная регистрация пользователя с машины 1 в машину 2, даже когда эти машины расположены на разных континентах (связь машин обеспечивает Интернет).

Помимо сетевого подключения машина может иметь десятки терминалов, подключенных через последовательные порты.

Терминал представляет собой экран и клавиатуру, т. е. консоль, на которой пользователь может регистрироваться обычным образом.

Консоль не виртуальная, а представлена специализированным отдельным устройством. Терминалом может служить персональный компьютер, на котором выполняется программа, имитирующая логику работы консоли (прозрачный ввод с местной клавиатуры в Linux-машину и прозрачный вывод на местный дисплей данных от Linux-машины).

Все перечисленные способы регистрации пользователей в системе действуют совместно. Таким образом, в одной машине десятки и сотни пользователей, вошедших в систему по различным каналам связи, могут одновременно выполнять тысячи задач. В этой связи одной из важных задач становится настройка прав доступа к файлам. Ниже приведем список команд, реализующих настройку и изменение прав доступа к файлам.

Основные права доступа к файлам в Linux. Изначально каждый файл имел три параметра доступа:

- чтение — разрешает получать содержимое файла, но на запись разрешения нет, для каталога позволяет получить список файлов и каталогов, расположенных в нем;
- запись — разрешает записывать новые данные в файл или изменять существующие, а также позволяет создавать и изменять файлы и каталоги;
- выполнение — вы не можете выполнить программу, если у нее нет флага выполнения. Этот атрибут устанавливается для всех программ и скриптов, именно с его помощью система может понять, что этот файл нужно запускать как программу.

При этом каждый файл имеет три категории пользователей, для которых можно устанавливать различные **сочетания прав доступа**:

- владелец — набор прав для владельца файла, пользователя, который его создал, или который сейчас установлен его владельцем. Обычно владелец имеет все права — чтение, запись и выполнение;
- группа — любая группа пользователей, существующая в системе и привязанная к файлу. Но это может быть только одна группа, и обычно это группа владельца, хотя для файла можно назначить и другую группу;
- остальные — все пользователи, кроме владельца и пользователей, входящих в группу файла.

Именно с помощью этих наборов полномочий устанавливаются права файлов в Linux. Каждый пользователь может получить полный доступ только к файлам, владельцем которых он является, или к тем, доступ к которым ему разрешен. Только пользователь `root` может работать со всеми файлами независимо от их набора и полномочий.

Но со временем такой системы стало не хватать, и было добавлено еще несколько флагов, которые позволяют делать файлы неизменяемыми или выполнять программы от имени суперпользователя.

Специальные права доступа к файлам в Linux. Чтобы позволить обычным пользователям выполнять программы от имени суперпользователя без знания его пароля, существуют SUID- и SGID-биты. Рассмотрим эти полномочия подробнее.

SUID. Если этот бит установлен, то при выполнении программы ID пользователя, от которого она запущена, заменяется на ID владельца файла. Фактически это позволяет обычным пользователям запускать программы от имени суперпользователя.

SGID. Этот флаг работает аналогично, разница в том, что пользователь считается членом группы, с которой связан файл, а не групп, к которым он действительно принадлежит. Если SGID-флаг установлен на каталог, все файлы, созданные в нем, будут связаны с группой каталога, а не пользователя. Такое поведение используется для организации общих папок.

Sticky-bit. Этот бит тоже используется для создания общих папок. Если он установлен, то пользователи могут только создавать, читать и выполнять файлы, но не могут удалять файлы, принадлежащие другим пользователям.

Рассмотрим, как посмотреть и изменить права на файлы в Linux.

Просмотр прав доступа к файлам в Linux. Можно посмотреть права доступа к файлам в Linux с помощью файлового менеджера, но так можно получить неполную информацию. Для максимально подробной информации обо всех флагах, в том числе специальных, нужно использовать команду `ls` с параметром `-l`. Все файлы из каталога будут выведены в виде списка, и там будут показаны все атрибуты и биты.

Чтобы узнать права на файл Linux, следует выполнить команду `ls -l` в папке, где находится этот файл:

```
sergiy@linux-x1a7:~/test> ls -1
итого 4
-rwxr-xr-x 1 sergiy users 0 окт 6 19:09 test1
---------- 1 sergiy users 0 окт 6 19:10 test10
-rwsr-sr-x 1 sergiy users 0 окт 6 19:09 test2
drwxr-xr-t 2 sergiy users 4096 окт 6 19:34 test3
-rwxrwxrwx 1 sergiy users 0 окт 6 19:09 test4
-rw-r--r-- 1 sergiy users 0 окт 6 19:09 test5
-rw-r--r-- 1 sergiy users 0 окт 6 19:09 test6
-rw-r--r-- 1 sergiy users 0 окт 6 19:09 test7
-rw-r--r-- 1 sergiy users 0 окт 6 19:09 test8
-rw-r--r-- 1 sergiy users 0 окт 6 19:10 test9
sergiy@linux-x1a7:~/test>
```

За права файлов в Linux отвечают «черточки». Первая — это тип файла. Дальше идут группы прав — сначала для владельца, для группы и для всех остальных. Всего девять «черточек» на права и одна на тип. Рассмотрим **условные значения флагов прав**:

- `-` — нет прав;
- `-x` — разрешено только выполнение файла как программы, но не изменение и не чтение;
- `-w-` — разрешены только запись и изменение файла;
- `-wx` — разрешено изменение и выполнение, но в случае с каталогом вы не можете посмотреть его содержимое;
- `r-` — права только на чтение;
- `r-x` — только чтение и выполнение, без права на запись;
- `rw-` — права на чтение и запись, но без выполнения;
- `rwx` — все права;
- `-s` — установлен SUID- или SGID-бит, первый отображается в поле для владельца, второй — для группы;
- `-t` — установлен sticky-bit, а значит, пользователи не могут удалить этот файл.

В приведенном примере файл `test1` имеет типичные разрешения для программ:

- владелец может все;
- группа — только чтение и выполнение;
- все остальные — только выполнение.

Для `test2` дополнительно установлен флаг SUID и SGID, а для папки `test3` установлен sticky-bit. Файл `test4` доступен всем. Теперь вы знаете, как посмотреть права на файл Linux.

Изменение прав файла в Linux. Чтобы изменить права на файл в Linux, необходимо использовать утилиту `Chmod`.

Данная утилита позволяет менять все флаги, включая специальные. Ее синтаксис:

```
$ chmod опции категория действие флаг файл
```

С помощью **опции `-R`** вы можете заставить программу применять изменения ко всем файлам и каталогам рекурсивно.

Категория указывает, для какой группы пользователей нужно применять права. Как вы помните, доступны только три категории:

- `u` — владелец файла;
- `g` — группа файла;
- `o` — другие пользователи.

Действие может быть одно из двух: либо добавить флаг «+», либо убрать флаг «–».

Сами права доступа аналогичны выводу утилиты `ls`:

- `r` — чтение;
- `w` — запись;
- `x` — выполнение;
- `s` — `suid/sgid`, в зависимости от категории, для которой вы его устанавливаете;
- `t` — устанавливает sticky-bit.

Например, всем пользователям предоставляется полный доступ к файлу `test5`:

```
chmod ugo+rwx test5
```

Заберем все права у группы и остальных пользователей:

```
chmod go-rwx test5
```

Дадим группе право на чтение и выполнение:

```
chmod g+rx test5
```

Остальным пользователям — только чтение:

```
chmod o+r test5
```

Для файла `test6` установим SUID:

```
chmod u+s test6
```

А для `test7` — SGID:

```
chmod g+s test7
```

В результате получилось:

```
ls -l
sergiy@linux-x1a7:~/test> ls -1
итого 4
-rwxr-xr-x 1 sergiy users 0 окт 6 19:09 test1
---------- 1 sergiy users 0 окт 6 19:10 test10
-rwsr-sr-x 1 sergiy users 0 окт 6 19:09 test2
drwxr-xr-t 2 sergiy users 4096 окт 6 19:34 test3
-rwxrwxrwx 1 sergiy users 0 окт 6 19:09 test4
-rw-r-xr-- 1 sergiy users 0 окт 6 19:09 test5
-rwSr--r-- 1 sergiy users 0 окт 6 19:09 test6
-rw-r-Sr-- 1 sergiy users 0 окт 6 19:09 test7
-rw-r--r-- 1 sergiy users 0 окт 6 19:09 test8
-rw-r--r-- 1 sergiy users 0 окт 6 19:10 test9
sergiy@linux-x1a7:~/test>
```

Как видно из приведенных примеров, изменить права на файл в Linux довольно просто. Основные права возможно изменить с помощью файлового менеджера.

2.6. ТИПОВЫЕ ИНСТРУМЕНТЫ И МЕТОДЫ АНАЛИЗА ПРОГРАММНЫХ ПРОЕКТОВ

Основными инструментами анализа программных проектов являются отладчики и дизассемблеры.

Отладчики. Используются при разработке компьютерных программ на этапе «отладка». Они облегчают процесс обнаружения, локализации и устранения ошибок в алгоритме программы и в ее программном коде. Чтобы определить место ошибки, надо следить за текущими значениями переменных и отслеживать ход выполнения программы.

Отладчик — специальная программа, которая с помощью пользовательского интерфейса позволяет пошагово выполнять программу с остановками в заданных программистом характерных местах программы (некоторых строках или по достижении заданного условия).

Пользовательский интерфейс отладчика дает возможность наблюдать за ходом выполнения отлаживаемой программы, останавливать и перезапускать ее, прогонять в замедленном темпе, изменять значения в памяти и иногда возвращать назад по времени.

Среди технологий отладки обычно рассматривают два подхода:

1) с помощью специально добавленных в код программы операторов вывода обеспечить для ее критических точек вывод текущего состояния на любое удобное программисту средство отображения информации с обеспечением отображения значений переменных программы;

2) использование отладчиков.

Существует множество отладчиков. К ним относятся DEBUG, SUPER TRACER, CODEVIEW, NTiceSET, TRW и др.

Рассмотрим особенности отладчиков на примере Turbo Debugger. Он позволяет отлаживать программы на уровне исходного кода и представляет собой набор инструментальных средств, облегчающих программисту процесс отладки разработанной пограммы и использующих семейство компиляторов Borland. Этот отладчик содержит набор выполняемых файлов, утилит и справочных файлов.

Turbo Debugger позволяет отлаживать программы для DOS, Windows NT и Microsoft Windows. Наличие перекрывающих друг друга окон, спускающихся и раскрывающихся меню обеспечивает программисту удобную работу. Этот отладчик имеет быстрый интерактивный интерфейс пользователя, а интерактивная, контекстнозависимая справочная система обеспечивает программиста подсказкой на всех этапах его работы.

Turbo Debugger также имеет полный набор отладочных средств, таких как:

- полное управление выполнением программы и все средства проверки данных;
- вычисление любых выражений на языках Assembler, Pascal, C и C++;
- широкие возможности задания точек останова и регистрации;
- обратное выполнение;
- доступ на нижнем уровне к регистрам процессора и к системной памяти;
- наличие макрокоманд в виде последовательностей нажатий клавиш, ускоряющих ввод и выполнение команд отладки;
- контекстно-зависимое меню;
- обработка исключительных ситуаций C, C++ и операционной системы;
- сохранение сеанса, и др.

Запуск отладчика целесообразен после компиляции и компоновки программ с включением отладочной информации. После запуска Turbo Debugger и загрузки под его управлением отлаживаемой программы начнется процесс отладки. Существуют три варианта этого отладчика: TD.EXE (для отладки 16-разрядных приложений DOS), TDW.EXE (для отладки 16-разрядных приложений Windows) и TD32.EXE (для отладки 32-разрядных приложений Windows).

Параметры запуска и режимы отладки Turbo Debugger можно задать при запуске из командной строки. Команда на запуск Turbo Debugger из командной строки имеет следующий синтаксис:

```
TD TDW TD32 [параметры] [имя_программы] [аргументы].
```

При выполнении TDW (или TD32) открывается полноэкранное текстовое окно, но сменить задачи в нем клавишами Alt+Esc или Ctrl+Esc нельзя. Этого ограничения нет в Windows NT, где TD32 активизирует окно с командной подсказкой и где доступны все обычные средства приложения Windows.

В процессе отладки отладчик периодически контролируемо передает управление отлаживаемой программе. Когда управление работой компьютера находится у Turbo Debugger, он может использовать свои средства для поиска по исходному коду и структурам данных программы. Это позволяет выявить причины неправильной работы программы. Процесс поиска причин облегчается путем использования меню и окон отладчика. Есть много способов управления ходом выполнения программы: программу можно выполнять по шагам (по одной машинной инструкции или строке исходного кода), можно выполнить функцию за один шаг или всю программу до заданного точкой останова места либо до появления определенного сообщения Windows и др. С другой стороны, при выполнении программы можно получить доступ к самому отладчику, что полезно, например, когда в программе не установлены точки останова.

Помимо Turbo Debugger стоит обратить внимание на SoftIce, который представляет собой универсальный отладчик, позволяющий отладить любой код, включая драйверы ввода-вывода и прерывания. SoftIce состоит из низкоуровневого отладчика и загрузчика отладочной информации, имеет широкие возможности и среди множества функций способен распознавать функции api с перечислением необходимых параметров.

Дизассемблеры. ***Дизассемблер*** — транслятор машинного кода, преобразующий объектный файл или библиотечные модули в текст программы на языке ассемблера. Дизассемблеры можно разделить на автоматические и интерактивные.

Автоматические дизассемблеры создают готовый текст программы, который затем можно редактировать в любом текстовом редакторе. Примером такого дизассемблера может служить `Sourcer`, а примером **интерактивного дизассемблера** — `IDA PRO`, который позволяет изменять правила дизассемблирования и очень удобен для исследования программ.

Также дизассемблеры можно классифицировать по **количеству проходов** на одно- и многопроходные. Однопроходные дизассемблеры (как и построчные ассемблеры) обычно входят в состав отладчиков.

Основное затруднение при работе дизассемблера состоит в различении данных от машинного кода. Для преодоления этого затруднения на первых проходах собирается информация о границах процедур и функций, а на последнем формируется итоговый текст программы. Интерактивный дизассемблер лучше приспособлен для решения этой задачи, так как программист, просматривая дамп области памяти, может сразу выделить строковые константы, дать содержательные имена известным точкам входа и написать комментарии к разобранным фрагментам программы.

Дизассемблер также может использоваться для анализа машинного кода с неизвестным исходным текстом дизассемблируемой программы. Другое его применение — поиск ошибок в программах и компиляторах, а также анализ степени оптимизации создаваемого компилятором машинного кода. Существуют также дизассемблеры длин, которые позволяют разделять лист машинного кода на отдельные команды.

2.7. ИНСТРУМЕНТАРИЙ РАЗЛИЧНЫХ СРЕД РАЗРАБОТКИ

Подбор удобного инструмента — одна из главных задач программиста. Первым делом это касается среды разработки. IDE (среда разработки) отличается от простого редактора кода следующим **инструментарием**:

- поддержка большого числа языков программирования;
- наличие компилятора, интерпретатора для преобразования в машинный код;

- встроенные утилиты для автоматизации процесса — библиотеки, шаблоны, сниппеты и т. д.;
- наличие отладчика для обнаружения ошибок, опечаток и прочих дефектов программного кода.

Для начинающих программистов может хватить редактора кода, который использует один-два языка, а проверку кода на ошибки лучше осуществлять в ручном режиме. Для разработчиков, имеющих некоторый опыт, IDE уже является необходимостью.

В табл. 2.1 приведены характеристики некоторых популярных в настоящее время IDE.

Подробнее с характеристиками сред разработки можно ознакомиться на официальных сайтах компаний-производителей.

Таблица 2.1. Характеристики IDE

IDE	Платформа	Описание
NetBeans	Windows/ Mac/Linux	Кроссплатформенная open-source IDE, предназначенная для работы на Java, но поддерживающая Python, JavaScript, Ruby, C, C++, Ada, PHP, HTML, CSS, XML, Groovy. Среда позволяет работать во всех трех направлениях разработки
Eclipse	Windows/ Mac/Linux	Популярная IDE, не имеющая привязки к ОС, и с помощью надстроек поддерживающая до 25 языков программирования. Количество подключаемых библиотек делает Eclipse одной из самых функциональных IDE
Microsoft Visual Studio Express	Windows	Усеченная, но бесплатная версия Visual Studio. Система автодополнения IntelliSense, поддержка C, C++, C#, Python, Ruby, VB.NET, JavaScript, HTML, CSS/XML, СКВ Git и TFS, интеграция с Azure
CodeLite	Windows/ Mac/Linux	Кроссплатформенная IDE для разработки ПО на C/C++, PHP и Node.js. Несмотря на свою минималистичность, она содержит все для начинающего разработчика

Окончание табл. 2.1

IDE	Платформа	Описание
`Code::Blocks:`	Windows/ Mac/Linux	Поддержка большого числа компиляторов, среди которых MinGW/GCC, Watcom, Clang, Digital Mars C/C++, и отладчиков GNU GDB и MS CDB. Инструмент `wxSmith` для быстрой разработки приложений (RAD). Основная библиотека `wxWidgets` упрощает разработку графического интерфейса в приложениях
`Qt Creator`	Windows/ Mac/Linux	`Qt Creator` специализируется на работе с C, C++ и QML, поддерживает те же самые компиляторы и отладчики, что и `Code::Blocks:`. Работает со всеми популярными СКВ: Subversion, Mercurial, Git, CVS, Bazaar, Perforce
`Aptana Studio`	Windows/ Mac/Linux	Без подключения библиотек `Aptana` может работать только с JavaScript, HTML и CSS. С ними поддерживает PHP, Python и Ruby. Также `Aptana` подключается к `Eclipse` в качестве плагина, полезна для начинающих
`WebScripter`	Mac	Простая и понятная IDE от Apple. Не развивается уже лет пять, но для начинающих достаточно встроенных редактора, отладчика и дебаггера
`PSPad`	Windows	Редактор кода, поддерживающий более 30 языков программирования, в том числе Cobol и Pascal. Имеет возможность подключения внешнего компилятора и отладчика, что превращает `PSPad` в настоящую IDE

Некоторые из перечисленных инструментов подходят только для начинающих программистов, при этом время на изучение таких сред разработки минимально. Другие обладают расширен-

ными возможностями, но требуют долгого освоения. Анализ мнений разработчиков показывает, что популярны также такие среды разработки, как PHPStorm, IntelliJ IDEA Community (от JetBrains). Некоторые из сред разработки будут подробнее рассмотрены в следующих разделах.

2.8. ИНСТРУМЕНТАРИЙ JAVA DEVELOPMENT KIT

Java Development Toolkit — платформа Java, среда разработки, представляющая собой инструментарий и набор утилит, который позволяет создавать Java-приложения (официальный сайт разработчика Java: http://java.sun.com). Содержит стандартные инструменты и утилиты, а также экспериментальные инструменты и утилиты JDK.

Базовый инструментарий JDK:

- javac — компилятор языка Java;
- java — интерпретатор байт-кода;
- javah — создание заголовочные файлы;
- javadoc — формирование стандартной документации;
- jar — создание дистрибутивов Java;
- javap — дизассемблер;
- apt — обработчик аннотаций.

Другие базовые инструменты — appletviewer, jdb, extcheck.

Javac (компилятор языка Java) преобразует исходный код в промежуточный байт-код.

Пример компиляции:

```
javac myClass.java
```

После выполнения данной команды будет создан файл myClass.class в той же директории, где расположен файл с исходным кодом, myClass.java. Все параметры компиляции (в том числе расположение обоих файлов) можно конфигурировать при помощи опций. Полученный в результате байт-код не может быть выполнен непосредственно (как файл .exe). Его может выполнить только интерпретатор Java. Виртуальная машина может изменять свое поведение в зависимости от переданных параметров.

Параметры компиляции:

- -X — дополнительные опции;
- -J — свойство, передаваемое в JVM. Виртуальная машина может изменять свое поведение в зависимости от переданных параметров;

- `-cp (-classpath)` — указание пути, по которому можно найти классы, необходимые для компиляции (переменная `CLASSPATH`);
- `-bootclasspath` — указание пути, по которому можно найти классы;
- `-help` — перечень разрешенных опций компилятора;
- `-target` — указание версии JVM, для которой создается класс-файл;
- `-version` — вывод версии компилятора;
- `-source` — указание версии исходного кода.

Расширенные параметры компиляции. Дополнительные проверки для JNI-кода:

- `Xbootclasspath/a (/p)` — замена классов, необходимых для запуска компилятора;
- `-Xstdout` — перенаправление вывода программы;
- `-Xlint` — вывод предупреждения о некорректном коде программы;
- `-Xcheck:jni` — дополнительные проверки для JNI-кода;
- `-Xstdout` — перенаправление ввода программы.

Java (интерпретатор байт-кода) запускает среду выполнения Java (Java Runtime Environment, JRE), загружает указанный класс и вызывает метод `main` данного класса. Примеры запуска:

```
java myClass.class
  java -jar myArchive.jar SomeArgument
```

Можно передавать аргументы приложению и указывать опции запуска в командной строке.

Параметры запуска (параметры исполнения можно изменять при помощи ключей интерпретатора, передаваемых Java):

- `-agentlib` — загрузка отладочного агента;
- `-X` — расширенные параметры;
- `-client/-server` — выбор клиентской или серверной модификации JVM;
- `-javaagent` — загрузка Java-агента;
- `-D` — установка системного свойства, указание пути, по которому содержатся классы, необходимые для запуска;
- `-cp(-classpath)` — указание пути, по которому можно найти классы, необходимые для запуска;
- `-client/-server` — выбор клиентской или серверной модификаций JVM.

Расширенные параметры запуска:

- `-Xss` — установка размера стека;
- `-Xloggc` — журнализация сборщика мусора;

- `-Xincgc` — включение инкрементального сборщика мусора;
- `-Xcheck:jni` — дополнительные проверки JNI-вызовов;
- `-Xms/-Xmx` — установка размера heap-области (начальный/максимальный размеры);
- `-Xnoclassgc` — отключение удаления объектов;
- `-Xbootclasspath/a` — замена классов, необходимых для запуска приложения.

Javah генерирует заголовочные файлы и файлы-заглушки на языке C на основе Java-классов. Данные файлы позволяют наладить взаимодействие кода на Java и на C или C++. Название заголовочного файла и структуры, в нем описанной, основывается на имени Java-класса с учетом пакета. Технология JNI (Java Native Interface) позволяет общаться с чужим кодом без заголовочных файлов и заглушек.

Для отладки следует использовать утилиту `javah_g` (неоптимизированную версию `Javah`):

```
javah [options] fully-qualified-classname...
javah_g [options] fully-qualified-classname...
```

Javadoc генерирует документацию к программному интерфейсу приложения.

Для обеспечения поддержки код должен быть хорошо документирован. Найти нужную информацию непосредственно в коде не всегда просто. Для этого нужен инструмент, собирающий разбросанные по коду комментарии и предоставляющий удобную навигацию по ним.

От разработчика требуется придерживаться несложных правил написания комментариев (чтобы `Javadoc` правильно их интерпретировал), а также запускать утилиту `Javadoc` для создания и обновления документации.

По умолчанию `Javadoc` генерирует документацию для пакетов, public-классов и интерфейсов, public- и protected-методов, public- и protected-полей.

При необходимости документировать private-классы/методы/поля можно указать ключ `-private`. Внутри текста комментария можно использовать специальные теги. Стандартные теги HTML допустимы.

Запуск утилиты:

```
javadoc [options] [packagenames] [sourcefiles] [@files]
```

На рис. 2.1 представлен пример `Javadoc`, используется формат HTML.

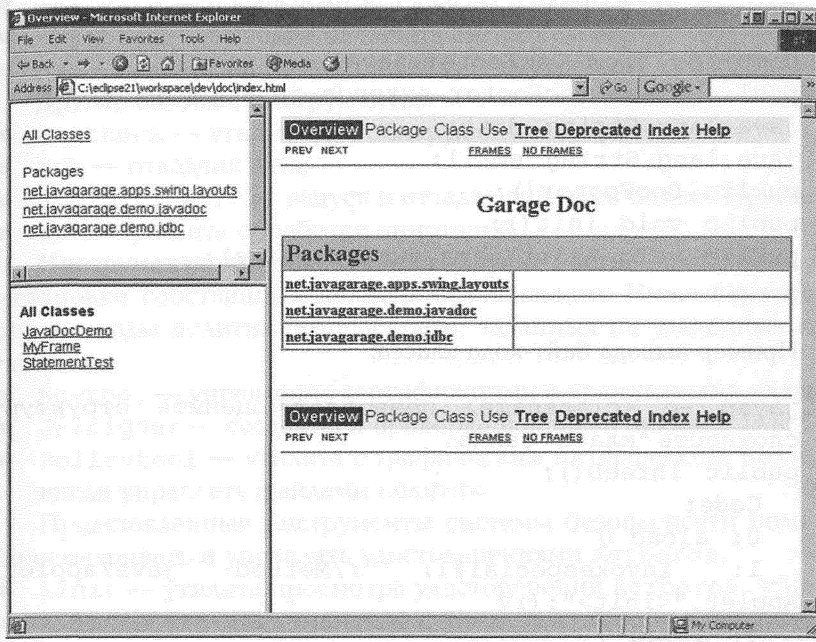


Рис. 2.1. Пример использования `Javadoc`

`Jar` (`Java ARchive`) — утилита для создания дистрибутивов Java-программ.

Пример:

```
% jar cf myApp.jar *.class
```

Все файлы текущей директории с расширением `.class` собираются в архив `myApp.jar`. При этом автоматически создается манифест, содержащий метаинформацию о приложении. Использует алгоритм `Zip`.

Также можно сделать JAR-файл самораспаковывающимся. Запакованное приложение можно запустить двойным щелчком. JAR-файлы могут быть подписаны автором.

`Javap` (дизассемблер) разбирает класс-файл. Выводимая информация варьируется в зависимости от используемых опций. По умолчанию `Javap` выводит название пакета, а также protected- и public-поля и методы анализируемого класса. Запуск утилиты из командной строки:

```
javap [options] class...
```

Пример вывода информации о классе:

```
Compiled from DocFooter.java
public class DocFooter extends java.applet.Applet {
java.lang.String date;
java.lang.String email;
public DocFooter();
public void init();
public void paint(java.awt.Graphics);
}
```

Пример вывода байт-кода класса:

```
javap -c <classname> позволяет оценить структуру
и сложность класс файла.
public ThreeD();
  Code:
  0: aload_0
  1:   invokespecial#1;   //Method   java/applet/
Applet."<init>":()V
  4: aload_0
  5: iconst_1
  6: putfield #2; //Field painted:Z
  9: aload_0
  10: fconst_1
  11: putfield #3; //Field scalefudge:F
  14: aload_0
  15: new #4; //class Matrix3D
  18: dup
  19: invokespecial#5; //Method Matrix3D."<init>":()V
  22: putfield #6; //Field amat:LMatrix3D;
  25: aload_0
  26: new #4; //class Matrix3D
  29: dup
  30: invokespecial#5; //Method Matrix3D."<init>":()V
  33: putfield #7; //Field tmat:LMatrix3D;
```

Javap-опции:

- `-package` — показывает пакет, в котором расположен класс, а также его protected- и public-поля и методы. Данная опция используется по умолчанию;
- `-public (-protected)` — показывает только public- и protected-классы и члены классов;

- `-private` — показывает все классы и члены классов;
- `-verbose` — выводит на печать сигнатуры внутренних типов;
- `-s` — выводит на печать сигнатуры внутренних типов.

Другие базовые инструменты:

- `extcheck` — утилита для обнаружения конфликтов JAR;
- `jdb` — отладчик Java;
- `appletviewerv` — запуск и отладка апплетов без веб-браузера;
- `apt` — утилита обработки аннотаций.

Инструментарий обеспечения безопасности. Необходим для установки собственных политик безопасности. Ниже перечислены команды политик безопасности, заданных на удаленных сайтах:

- `Keytool` — управление сертификатами и хранилищами ключей;
- `Jarsigner` — создание и проверка подписей архивов JAR;
- `Policytool` — утилита с графическим интерфейсом, позволяющая управлять файлами политик.

Представленные инструменты системы безопасности помогут просматривать и управлять удостоверениями `Kerberos`:

- `kinit` — утилита просмотра удостоверений `Kerberos v5`;
- `klist` — утилита командной строки, позволяющая работать со списком записей в кеше удостоверений;
- `ktab` — утилита командной строки, помогающая пользователю управлять записями в таблице ключей.

Инструментарий интернационализации. Данный инструмент помогает создавать локализуемые приложения.

Инструментарий развертывания приложений. Содержит утилиты, используемые для развертывания Java-приложений и апплетов в сети:

- `pack200` — преобразует файл JAR в сжатый файл `pack200` при помощи системы сжатия данных `gzip`;
- `unpack200` — преобразует архив, полученный при использовании `pack200`, в архив JAR.

Пересылаемые архивы представляют собой архивы JAR, которые сжаты сильнее, чем обычно, и при этом могут быть непосредственно развернуты средой исполнения. За счет уменьшения размера файла снижается время загрузки.

Инструментарий подключения плагинов. Содержит утилиты, используемые совместно с Java Plug-in:

- `Unregbean` — снимает регистрацию компонента JavaBeans, использует Active X;
- `Htmlconverter` — преобразует страницу (файл) HTML, содержащую апплет в формате OBJECT/EMBED для Java Plug-in.

Инструментарий удаленного вызова методов:

- `Rmid` — системный демон активации RMI;
- `serialver` — получение класса Serial Version UID;
- `rmiregistry` — сервис регистрации удаленных объектов;
- `rmic` — генерация заглушеяк и каркасов для удаленных объектов.

Консоль контроля выполнения приложений. Включает графический инструмент, удовлетворяющий стандарту JMX, для мониторинга виртуальной машины Java. Позволяет следить за работой как локальной, так и удаленной JVM, а также за работой приложения и управлять им. Запускается следующей командой:

```
jconsole [ options ] [ connection... ]
connection = pid | host:port | jmxUrl
```

`host:port` — имя хоста, на котором запущена JVM, и номер порта; `jmxUrl` — адрес агента JMX, с которым следует установить соединение; `pid` — идентификатор процесса локальной JVM. Идентификатор пользователя должен быть одним и тем же для JVM и jconsole.

Инструментарий веб-сервисов. Обеспечивает легкую интеграцию веб-сервисов и веб-клиентов. Позволяет прозрачно использовать формат XML для передачи информации:

- `xjc` — компилятор связываний в архитектуре XML связываний Java;
- `Wsimport`, `wsgen` — инструменты для генерации переносимых приложений JAX-WS;
- `schemagen` — генератор схем в архитектуре XML связываний Java.

Экспериментальные инструменты:

- инструментарий мониторинга (`jps`, `jstat`, `jstatd`);
- инструменты диагностики ошибок (`jinfo`, `jhat`, `jmap`, `jsadebugd`, `jstack`);
- инструменты работы со скриптами (`jrunscript`).

Инструментарий мониторинга. Может быть использован для учета статистических показателей производительности JVM. Данные инструменты работают на всех современных платформах:

- `jps` — отслеживает состояние процессов JVM (JVM Process Status Tool), выводит список виртуальных машин Java (HotSpot) в интересующей системе;
- `jstat` — утилита для ведения статистики по JVM (JVM Statistics Monitoring Tool), подключается к оборудованной виртуальной

машине Java (HotSpot), собирает и заносит в журнал статистику по производительности;

- `jstatd` — демон утилиты JVM `jstat`, загружает приложение RMI сервера, отслеживающее создание и завершение работы виртуальной машины Java.

Перечисленные далее инструменты работают на всех современных платформах:

- `jinfo` — конфигурационная информация Java (Configuration Info for Java), выводит конфигурационную информацию для заданного процесса, файла памяти или удаленного сервера отладки;
- `jstat` — средство просмотра дампа (снимка информации) кучи (Heap Dump Browser). Стартует веб-сервис для файла, содержащего дамп кучи (например, созданного утилитой `jmap` с опцией `-dump`), позволяя просматривать содержимое кучи. Куча — это специализированная структура данных типа «дерево», которая удовлетворяет свойству кучи: если B является узлом-потомком узла A, то ключ(A) $\geqslant$ ключ(B). Из этого следует, что элемент с наибольшим ключом всегда является корневым узлом кучи, поэтому иногда такие кучи называют max-кучами (если сравнение перевернуть, то наименьший элемент будет всегда корневым узлом, такие кучи называют min-кучами). Не существует ограничений относительно того, сколько узлов-потомков имеет каждый узел кучи, хотя на практике их число обычно не более двух. Куча является максимально эффективной реализацией абстрактного типа данных, который называется очередью с приоритетом. Кучи имеют решающее значение в некоторых эффективных алгоритмах на графах, таких как алгоритм Дейкстры на d-кучах и сортировка методом пирамиды;
- `jmap` — карта памяти Java, выводит карты памяти разделяемых объектов или детали памяти кучи для заданного процесса, файла памяти или удаленного сервера отладки;
- `jstack` — трассировщик стека Java (Stack Trace for Java), выводит историю стека потоков для заданного процесса, файла памяти или удаленного сервера отладки;
- `jsadebugd` — демон отладки агента, занимающегося удобством эксплуатации приложения (Serviceability Agent Debug Daemon for Java), подключается к процессу или файлу памяти и действует в качестве сервера отладки.

Инструменты работы со скриптами. Оболочка для скриптов в Java (Script Shell for Java) — `jrunscript`. Может быть использован для запуска скриптов, взаимодействующих с платформой Java.

2.9. ИНСТРУМЕНТАРИЙ ECLIPSE C/C++ DEVELOPMENT TOOLS

Eclipse — интегрированная среда разработки, предназначенная для написания модульных приложений. Поддерживает Java, JS, C/C++, PHP, Python, 1C v8 и ряд других языков программирования, взаимодействует с любыми операционными системами.

Eclipse — один из лучших инструментов Java, созданных за последние годы. Вначале Eclipse появилась как коммерческий продукт, но в ноябре 2001 г. ее исходные коды были опубликованы. Создателем системы является компания Object Technology International (OTI), которая впоследствии была приобретена корпорацией IBM. С 2001 г. Eclipse была загружена более 50 млн раз и используется десятками тысяч программистов по всему миру. Поддержкой и разработкой Eclipse в настоящее время занимаются организация Eclipse Foundation и сообщество Eclipse, информацию о которых можно найти на официальном сайте в Интернете: http://www.eclipse.org.

Функционал среды разработки. В основе архитектуры Eclipse лежит компонент Runtime, который управляет расширениями, обеспечивает взаимодействие с операционной системой и осуществляет поддержку системы помощи. IDE служит для создания и настройки основных программных элементов, регулирует сборку и отладку проектов, поддерживает совместную работу и поиск файлов.

Основные возможности платформы:

- разработка приложений различных типов;
- компиляция программ;
- интеграция с СКВ;
- командная работа над проектом;
- загрузка из сети системных файлов для стартапов;
- поиск дополнительного ПО для повышения качества разработки.

Eclipse повышает продуктивность разработки программ, ее функциональность зависит от сборки — количества и типа подключенных расширений. Стандартная версия SDK обеспечивает минимальные возможности и используется для знакомства со средой, содержит два плагина (DT и PDE), формирующие IDE для программирования на Java и создания платформенных компонентов. Для получения платформы с требуемой функциональностью следует установить соответствующую сборку либо подключить необходимые модули к базовой версии.

Особенности платформы. Eclipse поставляется с открытым исходным кодом, что позволяет расширять функциональные возможности среды путем подключения диспетчеров баз данных, новых языков программирования, серверов приложений и различных модулей. Отличием Eclipse от NetBeans является использование библиотеки SWT вместо Swing при разработке пользовательского интерфейса приложений.

Преимущества среды разработки:

- кроссплатформенность;
- русифицированный интерфейс;
- поддержка всех языков программирования;
- бесплатная лицензия;
- наличие компилятора;
- расширенный набор функций.

В версии Eclipse 4.7, выпущенной в июне 2017 г., были добавлены новые компоненты и функции, повышена стабильность и производительность работы. Появился модуль для разработки приложений для Android. Для перевода интерфейса на русский язык необходимо добавить языковой пакет в список расширений дистрибутива.

Основные инструментальные средства Eclipse Java включают:

- редактор исходного кода (создание и редактирование исходного текста программ);
- средства отладки и интеграции с Ant.

Кроме того, в Eclipse доступно множество бесплатных и коммерческих дополнений (плагинов), таких как инструментальные средства создания схем UML, разработка баз данных и др.

Сама по себе Eclipse — это только платформа, которая предоставляет возможность разрабатывать дополнения, называемые плагинами, которые естественным образом встраиваются в платформу. В Eclipse доступны дополнения для следующих языков: C/C++, HTML, Cobol, Perl, PHP, Ruby и др. Можно также разработать собственное дополнение для расширения возможностей Eclipse.

Системные требования. Как уже отмечалось, Eclipse разработана для широкого круга операционных систем, таких как Linux, Microsoft Windows и macOS. Для ее запуска требуется JVM (Java Virtual Machine) — виртуальная Java-машина, а также JDK (Java Development Kit) — набор для Java-разработки. Загрузить данные пакеты можно с официального сайта разработчика Java: http://java.sun.com.

При первоначальном знакомстве среда IDE Eclipse может показаться несколько сложной для неподготовленного пользователя.

Ниже будет представлено описание инструментария работы со средой Eclipse.

Рабочее пространство (workspace) — каталог для проектов пользователя, в котором располагаются файлы проекта.

Все, что находится внутри каталога, считается частью рабочего пространства, например /home/user/workspace.

Инструментальные средства Eclipse становятся доступны сразу после запуска приложения. Это, по существу, сама платформа с набором различных функциональных возможностей главного меню, где прежде всего выделяется набор операций по управлению проектом.

Фактическая обработка, как правило, осуществляется дополнениями (плагинами), например, редактирование и просмотр файлов проектов осуществляется JDT, и т. д.

К инструментам (workbench) относится набор соответствующих редакторов и представлений, размещенных в рабочей области Eclipse. Для конкретной задачи определенный набор редакторов и представлений называют перспективой, или компоновкой.

Компоновка (perspective) — набор представлений и редакторов, расположенных в том порядке, который вам требуется.

В каждой компоновке присутствует свой набор инструментов, некоторые компоновки могут иметь общие наборы инструментов. В определенный момент времени активной может быть только одна компоновка. Переключение между различными компоновками осуществляется нажатием клавиш Ctrl+F8. Используя компоновки, можно настроить свое рабочее пространство под определенный тип выполняемой задачи.

В Eclipse также имеется возможность создавать свои компоновки. Открыть компоновку можно командой Window → Open Perspective.

Редакторы представляют собой программные средства, которые выступают дополнениями к операции с файлами (создавать, открывать, редактировать, сохранять и др.).

Представления являются дополнением к редакторам, где выводится информация сопроводительного или дополнительного характера, как правило, о файле, находящемся в редакторе. Открыть представления можно командой Window → Show View.

Наиболее часто используемые представления для различных компоновок приведены в табл. 2.2.

Проект (project) представляет собой набор файлов приложения и сопутствующих дополнений. При работе с Java (рис. 2.2) используются в основном файлы, имеющие расширения .java, .jsp, .xml.

Таблица 2.2. Представления для различных компоновок

Компоновка	Представление
Debug	Breakpoints, Debug, Variables, Expressions, Task, Outline, Console
Java Browsing	Projects, Packages, Types, Members
Java	Package, Explorer, Problems, Hierarchy, Outline, Javadoc, Declaration

Дополнением (plug-in) называют приложение, которое дополнительно может быть установлено в `Eclipse`. Примером дополнения может выступать JDT.

Мастер — программное средство, которое помогает пользователю в настройках и проведении сложной операции.

В `Eclipse` имеется множество различных мастеров, которые делают работу пользователя в системе удобной и эффективной, беря часть рутинных операций на себя. В качестве примера назовем мастер создания нового класса, который помогает пользователю в таких операциях, как создание нового файла в нужной директории.

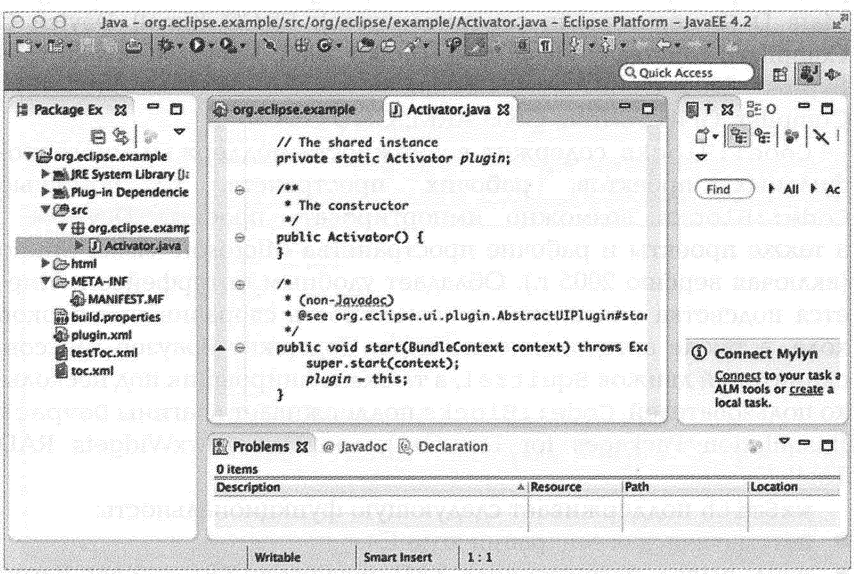


Рис. 2.2. Вид окна `Eclipse` с открытым Java-файлом

На рис. 2.2 представлен экран рабочего пространства Eclipse для Java.

Как видно из представленного рисунка, Eclipse обладает удобным и расширенным интерфейсом. (Для дальнейшего изучения инструментария IDE Eclipse рекомендуется обратиться к интернет-ресурсу http://www.eclipse.org.)

2.10. ИНСТРУМЕНТАРИЙ CODE::BLOCKS

Еще одной из самых востребованных IDE является **Code::Blocks** — свободная кроссплатформенная интегрированная среда разработки. Данная среда популярна как у практикующих разработчиков, так и у начинающих программистов. Code::Blocks достаточно проста в освоении и вполне может быть рекомендована для начального обучения С-подобным языкам программирования. Внешний вид окна приложения представлен на рис. 2.3.

Code::Blocks написана на C++ и использует библиотеку wxWidgets, имея открытую архитектуру, может масштабироваться за счет подключаемых модулей, поддерживает языки программирования C, C++, D (с ограничениями) и множество компиляторов, например: MinGW/GCC C/C++; Digital Mars C/C++; Digital Mars D (с некоторыми ограничениями); SDCC (Small Device C Compiler); Microsoft Visual C++ Toolkit 2003; Microsoft Visual C++ Express 2005 (with limits); Borland C++ 5.5; Watcom Intel C++ Compiler; GNU Fortran; GNU ARM; GNU GDC.

Code::Blocks содержит возможности поддержки многопрофильных проектов, рабочих пространств. С помощью Code::Blocks возможно импортировать проекты Dev-C++ , а также проекты и рабочие пространства Microsoft Visual Studio (включая версию 2005 г.). Обладает удобным интерфейсом. Имеется подсветка синтаксиса и возможность сворачивания блоков кода, а также автодополнение кода. Содержит браузер классов, скриптовой движок Squirrel, а также планировщик под несколько пользователей. Code::Blocks поддерживает плагины Devpack (Installation Packages for Dev-C++), wxSmith (awxWidgets RAD Tool).

wxSmith поддерживает следующую функциональность:

- автоматическая генерация кода C++;
- ввод, вывод, загрузка кода XML-ресурсов (XRC — XML Based Resource System);

- валидаторы;
- создание таблицы событий и обработчиков;
- поддержка импорта изображений различных форматов в проект-специфичный формат XPM и сохранение в коде (inline) или отдельных файлах;
- легкий доступ к указателям окна;
- управление сайзерами — невидимыми элементами формы, которые обозначают местоположение всех компонентов формы, разработку дизайна без сайзеров.

Среди возможностей интерфейса `Code::Blocks`:

- браузер классов;
- скриптовый движок `Squirrel`;
- система проверки правописания (только для комментариев);
- автоформатирование кода `AStyle`, настраиваемый `Code Style`;
- утилита для разработки регулярных выражений (использует `wxWidgets regexp parser`, синтаксис `pcre`);
- переименования в файлах проекта (базовая поддержка `Refactor → Rename` для произвольного идентификатора);
- `DoxyBlocks` — плагин для извлечения документации в формате `DoxyGen`, синтаксис комментариев поддерживается редактором кода;
- `Block Comment` (`Ctrl+C`, `Ctrl+X`);

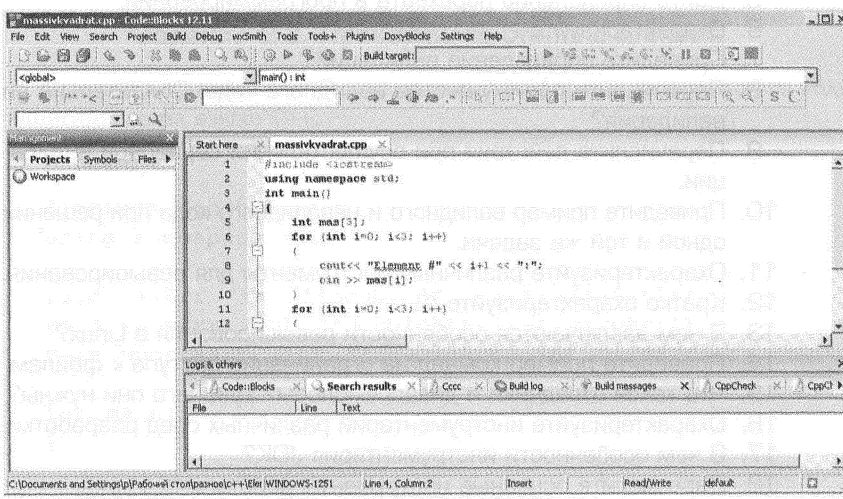


Рис. 2.3. Вид окна приложения `Code::Blocks`

- поиск по проекту с подсветкой найденных совпадений, поддерживает регулярные выражения;
- поиск места декларации или реализации идентификатора (функции, макроса, класса и т.д.), включая подключенные заголовки из внешних библиотек;
- переход `*.h<->*.cpp (F11)`;
- запуск полученной программы с аргументами (включая аргументы для терминала, например `cmd /u` (unicode console), или замену `xterm` на `gnome-terminal`).

`Code::Blocks` поддерживает отладчики GNU GDB и MS CDB.

Отметим, что `Code::Blocks` весьма удобен для начинающих разработчиков, так как отличается простым и понятным интерфейсом.

Подробнее о работе с IDE см. интернет-ресурс http://www.codeblocks.org/.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Дайте определение утилиты. Перечислите известные утилиты для ревьюирования.
2. Какие функции выполняют утилиты для ревьюирования?
3. Расскажите о разновидностях предпроцессинга кода.
4. Дайте определение интегрированной среды разработки и перечислите известные IDE.
5. Дайте определение перехвата в программировании.
6. Охарактеризуйте `PVS-Studio`.
7. Приведите определение валидации.
8. Перечислите преимущества валидного кода. Обязательна ли валидация?
9. Перечислите действия при осуществлении серверной валидации.
10. Приведите пример валидного и невалидного кода при решении одной и той же задачи.
11. Охарактеризуйте различные инструменты для ревьюирования.
12. Кратко охарактеризуйте `Git`.
13. В чем заключаются особенности ревьюирования в Linux?
14. Приведите пример команд на ограничение доступа к файлам.
15. Что такое отладчики и дизассемблеры? Для чего они нужны?
16. Охарактеризуйте инструментарий различных сред разработки.
17. В чем особенности инструментария JDK?
18. Перечислите основные инструменты работы в `Eclipse`.
19. Охарактеризуйте инструментарий `Code::Blocks`.
20. Перечислите этапы проведения работ при создании репозитория.

Задание 2.1. Планирование code-review

Цель исследования. *Цель* — ознакомление с этапами планирования ревьюирования, получение практических навыков работы в команде разработчиков по ревьюированию кода. Предварительно необходимо ознакомиться с теоретическими материалами гл.1.

Практическая часть. Исследование выполняется в несколько этапов.

1. Группа обучаемых распределяет роли в команде. Для рассмотрения предлагается заранее написанный код одного из учащихся, который в дальнейшем именуется «***автор***». За неделю до запланированной работы автор предоставляет код для ознакомления членам команды. В качестве ***председателя*** выступает преподаватель или (на усмотрение преподавателя) один из обучаемых. Также назначаются ***рецензенты*** — обучаемые, которые имеют достаточные знания и квалификацию, чтобы оценить код. ***Секретарь*** помогает в организации совещания, ведет его протокол, записывает, какие вопросы были подняты, чтобы ничего не забыть по окончании рецензирования.
2. Преподаватель подготавливает заранее необходимые ресурсы (компьютер, проектор, ПО, распечатки и т.д.).
3. К моменту начала собрания каждому разработчику рекомендуется написать личную рецензию на изученный код и выложить файл на предоставленный преподавателем ресурс.
4. Преподаватель оценивает результаты работы группы и каждого обучаемого. Определяется рейтинг рецензий. Обучаемым предлагается обменяться впечатлениями и выработать план проработки ошибок в коде.

Задание 2.2. Проверки на стороне клиента

Условия исследования. Представлены три версии программного кода, написанного на языке C++.

Вариант 1:

```
#include <iostream>
using namespace std;

void fillArr (int **doubleArr, int sizeArr1);
void showArr (int **doubleArr, int sizeArr1);
void updateArr (int **doubleArr, int sizeArr1);

int main()
{
   const int sizeArr1=5;
   int **doubleArr= new int *[sizeArr1];
   for (int i = 0; i < sizeArr1; i++)
   {
```

```
      doubleArr[i] = new int [sizeArr1];
   }

   fillArr(doubleArr , sizeArr1);
   cout << "Your Array now looks like: " << endl;
   showArr(doubleArr , sizeArr1);
   updateArr(doubleArr , sizeArr1);
   cout<<"Your updated Array now looks like: "<<endl;
   showArr(doubleArr , sizeArr1);

   for (int i = 0; i < sizeArr1; i++)
   {
      delete [] doubleArr[i];
   }
   delete [] doubleArr;

   return 0;
}

void fillArr (int **doubleArr, int sizeArr1)
{
   srand(time(0));
   for (int i = 0; i < sizeArr1; i++)
   {
      for (int j = 0; j < sizeArr1; j++)
      {
         doubleArr[i][j] = 10 + rand()%90;
      }
   }
}

void showArr (int **doubleArr, int sizeArr1)
{
   cout<<endl;
   for (int i = 0; i < sizeArr1; i++)
   {
      cout<<" | ";
      for (int j = 0; j < sizeArr1; j++)
      {
         cout<<doubleArr[i][j]<<" ";
      }
      cout<<" | "<<endl;
   }
}

void updateArr (int **doubleArr, int sizeArr1)
{
```

```
    for (int i = 0; i < sizeArr1; i++)
    {
       int maxValue = 0;
       int temp = doubleArr[i][0];
       int *ptrElem = &doubleArr[i][0];
       for (int j = 0; j < sizeArr1; j++)
       {
          if (maxValue < doubleArr[i][j])
          {
             maxValue = doubleArr[i][j];
             ptrElem = &doubleArr[i][j];
          }
       }
       doubleArr[i][0]= *ptrElem;
       *ptrElem = temp;
    }
}
```

Вариант 2:

```
#include <iostream>
#include <ctime>
using namespace std;

int main()
{
   setlocale(LC_ALL, "rus");
   srand(time(NULL));

   int size = 5;

   // выделяем память для указателей на строки матрицы
   int** arrWithDigits = new int* [size];
   for (int i = 0; i < size; i++)
   {
   arrWithDigits[i] = new int[size]; // выделение памяти
для каждого элемента строки матрицы
   }

   for (int i = 0; i < size; i++)
   {
   for (int j = 0; j < size; j++)
   {
   arrWithDigits[i][j] = 10 + rand() % 90; // заполнение
массива
   cout << arrWithDigits[i][j] << " | ";  // вывод мас-
сива
   }
```

```
    cout << endl;
    }
    cout << endl << endl;

    // поиск максимального значения в строке матрицы

    int max = 0; // для записи максимального значения
    int buf = 0; // буфер для перестановки местами значений
    int x = 0; // для записи номера строки в котором ищем
максимальное значение
    int y = 0; // для записи номера столбца с максималь-
ным значением

    for (int i = 0; i < size; i++)
    {
    max = arrWithDigits[i][0];
    for (int j = 1; j < size; j++)
    {
    if (arrWithDigits[i][j] > max)
    {
    max = arrWithDigits[i][j];
    x = i; // запоминаем номер строки
    y = j; // запоминаем номер столбца
    }
    }
    // замена максимума на первый элемент в строке
    if (arrWithDigits[i][0] < max)
    {
    buf = arrWithDigits[i][0];
    arrWithDigits[i][0] = max;
    arrWithDigits[x][y] = buf;
    }
    }

    cout << "Матрица после перестановки максимальных зна-
чений в строке: "<< endl;
    for (int i = 0; i < size; i++)
    {
    for (int j = 0; j < size; j++)
    {
    cout << arrWithDigits[i][j] << " | ";
    }
    cout << endl;
    }
    cout << endl << endl;

    // освобождение памяти
```

```
    for (int i = 0; i < size; i++)
    {
    delete[] arrWithDigits[i]; // освобождаем память ячеек
    }
    delete[] arrWithDigits; // память под указатели на
строки матрицы
    return 0;
}
```

Вариант 3:

```
#include <iostream>
using namespace std;

int **func1(int NumberOfLines, int NumberOfColumns);
//Выделение памяти
void func2(int **pointer, int NumberOfLines, int
NumberOfColumns);
    //Заполнение и вывод на экран
void func3(int **pointer, int NumberOfLines, int
NumberOfColumns);
//меняем местами максимальный и первый элемент в строках
int **func4(int **pointer, int NumberOfLines, int
NumberOfColumns); //Освобождение памяти

int main()
{
    setlocale(LC_ALL,"rus");
    srand(time(NULL));

    int n,m;

    cin >> n >> m; // ввести кол-во строк (n) и количе-
ство столбцов (m)

    int **a = NULL;
    a = func1(n,m);
    cout << "Исходная матрица: " << endl;
    func2(a,n,m);
    cout << "Конечная матрица: ";
    func3(a,n,m);
    func4(a,n,m);
}

int **func1(int NumberOfLines, int NumberOfColumns)
{
    int **pointer = new int*[NumberOfLines];
    for(int i(0); i < NumberOfLines; i++)
```

```
    {
    pointer[i] = new int[NumberOfColumns];
    }
    return pointer;
}

void func2(int **pointer, int NumberOfLines, int
NumberOfColumns)
{
    for(int i(0); i < NumberOfLines; i++)
    {
    for(int j(0); j < NumberOfColumns; j++)
    {
    pointer[i][j] = 10 + rand() % 99;
    }
    }
    for(int i(0); i < NumberOfLines; i++)
    {
    for(int j(0); j < NumberOfColumns; j++)
    {
    cout << pointer[i][j] << " ";
    }
    cout << endl;
    }
}
```

Цель и задачи исследования. *Цель* — научиться осуществлять проверку кода на соответствие заданным характеристикам.

Задачи:

- выявить ошибки в коде, в случае отсутствия ошибок оценить скорость выполнение кода, корректность вывода результатов, универсальность для работы с различными форматами данных;
- представить скриншот результатов выполнения кодов в среде разработки;
- оформить в виде списка рейтинг представленных вариантов, а также описать достоинства и недостатки каждого из них;
- представить возможные варианты дальнейшего улучшения качества каждого из вариантов.

Примечание. Анализ возможно провести в любой из имеющихся в наличии сред разработки.

Практическая часть. Исследование выполняется в два этапа.

1. Создать двумерный массив 5×5, заполнить его случайными числами от 10 до 99 и вывести на экран.

2. Поменять местами максимальный элемент каждой строки с первым элементом в соответствующей строке. Задачу решить с помощью указателей.

Задание 2.3. Проверки на стороне сервера

Цель исследования. *Цель* — научиться пользоваться валидаторами кода и плагинами для браузеров.

Валидация сайта позволяет следить за правильным отображением сайта в разных браузерах. Ошибки в тегах, опечатки в коде, стили css могут отображаться некорректно. Кроме того, поисковые системы отдают предпочтение валидным сайтам.

Практическая часть. Исследование выполняется в несколько этапов.

1. Осуществить проверку с помощью сервиса validator.w3.org. Для проверки необходимы веб-ресурсы, доступные на клиентском компьютере, один из которых размещен в Интернете.

Для начала проверки необходимо перейти по адресу: validator.w3.org. Откроется страница, на которой три вкладки (рис. 2.4).

2. Первоначально проверить веб-сайт, размещенный в Интернете. Необходимо выбрать опцию на первой вкладке `Validate by URI`.

3. Далее нажать на кнопку `More options` и выставить следующие значения: `Character Encoding` — кодировка исследуемого сайта. Если она уже есть между тегами <head>, то выполнить действия следующего пункта.

4. На сайте в браузере на клиентском компьютере нажать на сочетание клавиш `CTRL+U` и найти в начале документа строку:

```
<meta charset="UTF-8" />
```

Оставить выбранной опцию `detect automatically`. Если первой строкой указан тип текущего документа — `Document Type`, то оставить опцию — `detect automatically`. В поле `Address` ввести адрес иссле-

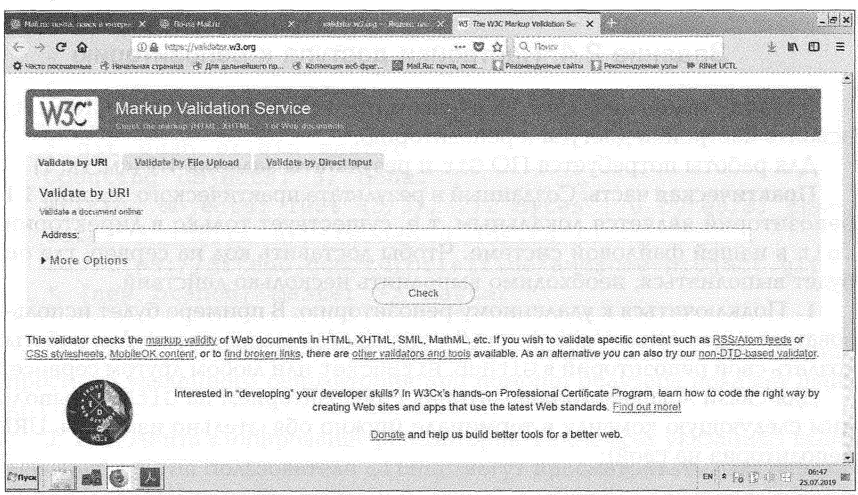


Рис. 2.4. Внешний вид окна валидатора

дуемого сайта. Нажать кнопку `Check`, которая расположена посередине серого блока.

5. Далее идет процесс валидации сайта, и через некоторое время появится результат валидации. Будет выведена страница с сообщением, например: «This document was successfully checked as HTML5!». Это значит, что ваш сайт успешно прошел проверку на валидность определенному типу документа — HTML5.

Если появилась надпись на красном фоне — это значит, что в HTML-документе присутствуют ошибки. Их необходимо исправить. Для этого следует просто выделить название ошибки и попытаться проанализировать ее с помощью поискового сервера, изучить рекомендации, представленные в Интернете, для исправления ошибки либо обсудить пути возможного исправления в группе.

6. Проверить валидность сайта, загруженного с компьютера. На первой вкладке страницы валидатора выбрать опцию `Validate by File Upload` и провести валидацию ресурса. Проверку валидности сайта осуществить с помощью опции `Validate by Direct Input`. При этом содержимое файла вставить непосредственно в форму для ввода.

7. Оформить отчет о результатах валидации, привести скриншоты выполненного задания, результаты проверки и исправления обнаруженных погрешностей.

8. Дополнительно: провести проверку веб-ресурсов с помощью плагинов. Загрузить плагины для соответствующего браузера со следующих ресурсов:

- http://users.skynet.be/mgueury/mozilla/;
- https://chrome.google.com/webstore/detail/html-tidy-browser;
- extensi/gljdonhfjnfdkl1jmfaabfpjlonflfnm;
- https://addons.opera.com/en/extensions/details/validator/.

Задание 2.4. Настройки доступа к репозиторию

Цель исследования. *Цель* — ознакомиться с возможностями СКВ `Git`, освоить настройки доступа к репозиторию.

Для работы потребуется ПО `Git` и результаты задания 1.1 (см. гл. 1).

Практическая часть. Созданный в результате практического задания 1.1 репозиторий является локальным, т. е. существует только в директории `.git` в нашей файловой системе. Чтобы доставить код на сервер, где он будет выполняться, необходимо выполнить несколько действий.

1. Подключиться к удаленному репозиторию. В примере будет использоваться адрес: https://github.com/tutorialzine/awesome-project. Для работы создать свой репозиторий в `GitHub`, `BitBucket` или любом другом сервисе.

Для связи локального репозитория с репозиторием на `GitHub` выполним следующую команду в терминале (нужно обязательно изменить URI репозитория на свой):

```
# This is only an example. Replace the URI with your
own repository address.
```

```
$ git remote add origin https://github.com/tutorialzine/
awesome-project.git
1
2
# This is only an example. Replace the URI with your
own repository address.
$ git remote add origin https://github.com/tutorialzine/
awesome-project.git
```

Проект может иметь несколько удаленных репозиториев одновременно. Для отличия им дают разные имена. Обычно главный репозиторий называется `origin`.

2. Осуществить отправку изменений на сервер. С помощью команды `push` пересылается локальный коммит на сервер. Этот процесс происходит каждый раз, когда мы хотим обновить данные в удаленном репозитории.

Команда принимает два параметра: имя удаленного репозитория (`origin`) и ветку, в которую необходимо внести изменения (`master` — это ветка по умолчанию для всех репозиториев). Введем команды:

```
$ git push origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 212 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/tutorialzine/awesome-project.git
* [new branch] master -> master
1
2
3
4
5
6

$ git push origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 212 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/tutorialzine/awesome-project.git
* [new branch] master -> master
```

Может потребоваться аутентификация. При правильном выполнении просмотр удаленного репозитория при помощи браузера покажет файл `hello.txt`.

3. Выполнить клонирование репозитория. После всех указанных выше действий другие пользователи `GitHub` могут просматривать наш репозиторий, скачать из него данные и получить полностью работоспособную копию проекта при помощи команды `clone`.

Выполним команду:

```
$ git clone https://github.com/tutorialzine/awesome-
project.git
1
$ git clone https://github.com/tutorialzine/awesome-
project.git
```

Новый локальный репозиторий создается автоматически с `GitHub` в качестве удаленного репозитория.

4. Запрос изменений с сервера. При изменении в репозитории другие пользователи могут скачать изменения при помощи команды `pull`:

```
$ git pull origin master
From https://github.com/tutorialzine/awesome-project
* branch master -> FETCH_HEAD
Already up-to-date.
1
2
3
4
$ git pull origin master
From https://github.com/tutorialzine/awesome-project
* branch master -> FETCH_HEAD
Already up-to-date.
```

Поскольку с тех пор, как проект клонирован, новых коммитов не было, никаких изменений, доступных для скачивания, нет.

5. Ветвление. ***Ветка*** — это копия оригинального проекта. Во время разработки новой функциональности считается хорошей практикой работать с веткой. Ветви имеют свою собственную историю и изолированные друг от друга изменения до тех пор, пока не принято решение о слиянии. При работе с веткой:

- рабочая, стабильная версия кода сохраняется;
- различные новые функции могут разрабатываться параллельно разными программистами;
- разработчики могут работать с собственными ветками без риска, что кодовая база поменяется из-за чужих изменений.

В случае сомнений различные реализации одной и той же идеи могут быть разработаны в разных ветках и затем сравниваться.

Создание новой ветки. Основная ветка в каждом репозитории называется `master`. Чтобы создать еще одну ветку, используем команду `branch <name>`:

```
$ git branch amazing_new_feature
1
$ git branch amazing_new_feature
```

Это создаст новую ветку, точную копию ветки `master`.

Переключение между ветками. Запустив `branch`, увидим две доступные опции:

```
$ git branch
amazing_new_feature
* master
1
2
3

$ git branch
amazing_new_feature
* master
```

`master` — активная ветка, она помечена звездочкой. Для переключения на другую ветку воспользуемся командой `checkout`, она принимает один параметр — имя ветки, на которую необходимо переключиться:

```
$ git checkout amazing_new_feature
1
$ git checkout amazing_new_feature
```

Слияние веток. Созданная новая ветка будет еще одним текстовым файлом под названием `feature.txt`. Создадим его, добавим и закоммитим:

```
$ git add feature.txt
$ git commit -m "New feature complete."
1
2
$ git add feature.txt
$ git commit -m "New feature complete."
```

Изменения завершены, теперь мы можем переключиться обратно на ветку `master`:

```
$ git checkout master
1
$ git checkout master
```

Откроем проект в файловом менеджере. Мы не увидим файла `feature.txt`, потому что переключились обратно на ветку `master`, в которой такого файла не существует. Чтобы он появился, необходимо объединить ветки с помощью команды `merge` (применение изменений из ветки `amazing_new_feature` к основной версии проекта):

```
$ git merge amazing_new_feature
1
$ git merge amazing_new_feature
```

Теперь ветка `master` актуальна. Ветка `amazing_new_feature` больше не нужна, и ее можно удалить:

```
$ git branch -d awesome_new_feature
1
$ git branch -d awesome_new_feature
```

6. Воспользовавшись данными задания 1.1, создать репозиторий и произвести настройки доступа, как указано в примерах. Скорректировать имена файлов и директорий применительно к своему проекту. Оформить отчет в виде скриншотов результатов работы команд. Изучить дополнительный материал на сайте `Git`: http://git-scm.com/.

Глава 3

МЕНЕДЖМЕНТ ПРОГРАММНОГО ПРОЕКТА

3.1. ИНСТРУМЕНТЫ ДЛЯ ИЗМЕРЕНИЯ ХАРАКТЕРИСТИК И КОНТРОЛЯ КАЧЕСТВА И БЕЗОПАСНОСТИ КОДА

Стремительное развитие вычислительной техники (ВТ), касающееся возможностей обработки и хранения информации, является движущей силой, которая заставляет постоянно модифицировать и совершенствовать ПО, связанное с управлением самими устройствами ВТ, и прикладное ПО, вовлеченное в процессы обработки, хранения и передачи информации в разрабатываемых информационных (компьютерных) системах (ИС, КС), автоматизированных системах управления (АСУ).

Автоматизированные системы по определению предполагают наличие персонала и средств автоматизации его деятельности, реализующих определенные информационные технологии. Средства автоматизации включают в себя ПО, от качества которого зависят функционирование всей системы в целом, а также оценка и результат деятельности различных заинтересованных групп людей, таких как коллективы разработчиков, заказчики и пользователи ПО, лица, использующие результаты применения ПО. При этом влияние ПО на деятельность различных людей может различаться по степени значимости — от малого и незначительного до критического и фатального. Например, часть бортовых систем самолета управляется программно, естественно, есть коллективы разработчиков, создавших это ПО, есть компания-перевозчик, которая покупает самолет, есть пилоты — пользователи ПО, есть пассажиры, пользующиеся результатами применения ПО. От функционирования этого ПО зависит многое, в том числе и жизнь людей. Таким образом, значимость ПО в человеко-машинной системе сложно переоценить.

Приведенный пример свидетельствует о важности процессов управления программным проектом на всех этапах жизненного цикла с целью создания качественного программного продукта, и это достаточно сложная задача. Все организации, вовлеченные в процессы разработки, закупки, сопровождения или оценки ПО, должны разработать политику и планы относительно действий по заданию требований и оценке качества, которые также предусматривают конкретные обязанности групп оценки.

На уровне этих организаций должны осуществляться следующие виды деятельности:

- управление средой организации — разработка планов, определение обязанностей и значимых показателей, проведение анализа;
- управление ресурсами — управление персоналом, финансами, всесторонняя поддержка деятельности персонала;
- планирование использования и усовершенствование требований к качеству и технологии оценки качества;
- реализация технологии оценки;
- передача технологии, используемой для оценки;
- оценка технологии для задания требований и оценки качества;
- управление опытом, т. е. применение накопленных знаний в проектах, сопровождаемых раньше.

На уровне управления проектом группа оценки, возглавляемая опытным руководителем, должна эффективно управлять своими действиями, а сама оценка должна иметь необходимые ресурсы, утвержденный бюджет, четко разработанный, задокументированный и согласованный план оценки качества и быть обеспечена соответствующими инструментами, стандартами и методиками.

В этой главе рассмотрены проблемы, касающиеся управления качеством ПО, и не затрагиваются вопросы и технологии, связанные с управлением ресурсами проекта, например использование техник экстремального программирования.

3.2. МЕТРИКИ, НАПРАВЛЕНИЯ ИХ ПРИМЕНЕНИЯ

3.2.1. Критерии и характеристики качества программы

Как уже отмечалось, при выполнении разнообразных функций как в коммерческих целях, так и для персонального использования программные вычислительные системы занимают лидирующее положение.

Проблема организации управления программными проектами заключается в том, что ПО не является чем-то осязаемым и по сути больше напоминает произведение искусства (например, как определить, качественно написана картина или нет). В свою очередь, процесс управления чем-либо есть контроль значений некоторых параметров. Именно о таких параметрах, численно характеризующих ПО, и пойдет речь далее.

Согласно ГОСТ Р ИСО/МЭК 25010—2015 «Информационные технологии (ИТ). Системная и программная инженерия. Требования и оценка качества систем и программного обеспечения (SQuaRE). Модели качества систем и программных продуктов» ***качество системы*** — степень удовлетворения системой заявленных и подразумеваемых потребностей различных заинтересованных сторон, которая позволяет оценить ее достоинства.

Эти заявленные и подразумеваемые потребности представлены в международных стандартах серии SQuaRE (System and Software Quality Models) посредством моделей качества, представляющих качество продукта в виде разбивки на классы характеристик, которые в отдельных случаях далее подразделяются на подхарактеристики (а некоторые подхарактеристики разделяются далее на подподхарактеристики). Подобная иерархическая декомпозиция обеспечивает удобную разбивку качества продукта на классы (рис. 3.1). Однако множество подхарактеристик, связанных с характеристикой, которая избрана для представления типичных проблем, необязательно будет исчерпывающим.

Характеристика качества программы — некоторое поддающееся измерению свойство, отражающее отдельные факторы, которые влияют на качество программы.

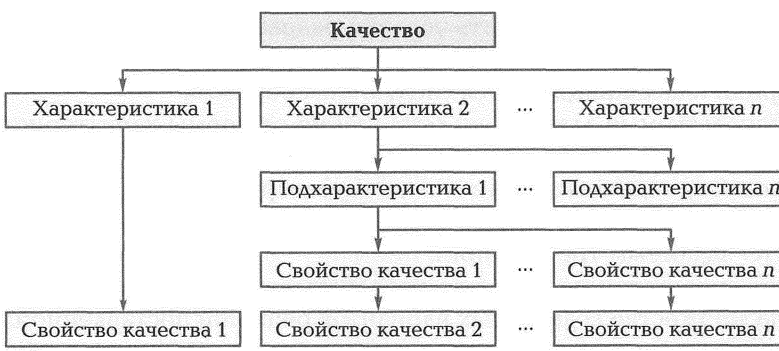


Рис. 3.1. Структура, используемая для моделей качества

Критерий качества — некоторый численный показатель, характеризующий степень, которая присуща оцениваемому свойству программы.

Общая схема взаимодействия критериев качества изображена на рис. 3.2. Критерии качества комплексов программ включают такие характеристики, как экономичность, гибкость, модульность, ясность, надежность, тестируемость, обоснованность, точность, модифицируемость, эффективность, легкость сопровождения, документированность и т. д.

Критерий качества должен соответствовать следующим требованиям:

- давать численную характеристику основной целевой функции программы;
- обеспечивать возможность определить:
  - ✓ затраты, необходимые для достижения требуемого уровня качества;
  - ✓ степень влияния на показатель качества внешних факторов;
- по возможности быть простым, хорошо измеримым и иметь малую дисперсию.

Для измерения критериев и характеристик качества программ используют **метрики**, которые составляют систему измерений качества. Такие измерения могут проводиться на уровне критериев качества программ или на уровне отдельных характеристик качества: система измерений позволяет непосредственно сравнивать программы по качеству, но сами измерения не могут быть проведены без субъективных оценок свойств программ; измерения могут быть выполнены объективно и достоверно, но оценка качества ПО в целом будет связана с субъективной интерпретацией получаемых оценок. При исследовании метрик ПО различают два основных подхода, когда ищутся метрики оценки самого ПО либо используются метрики, помогающие оценить условия разработки программы.

По **виду информации, получаемой при оценке качества ПО**, метрики классифицируются следующим образом:

- используемые для оценки отклонения от нормы характеристик исходных проектных материалов, они позволяют установить полноту заданных технических характеристик исходного кода;
- используемые для принятия решения о соответствии конечного ПО заданным требованиям;
- прогнозирующие качество разрабатываемого ПО, они заданы на множестве возможных вариантов решений поставленной задачи и их реализации и определяют качество ПО, которое будет достигнуто в итоге.

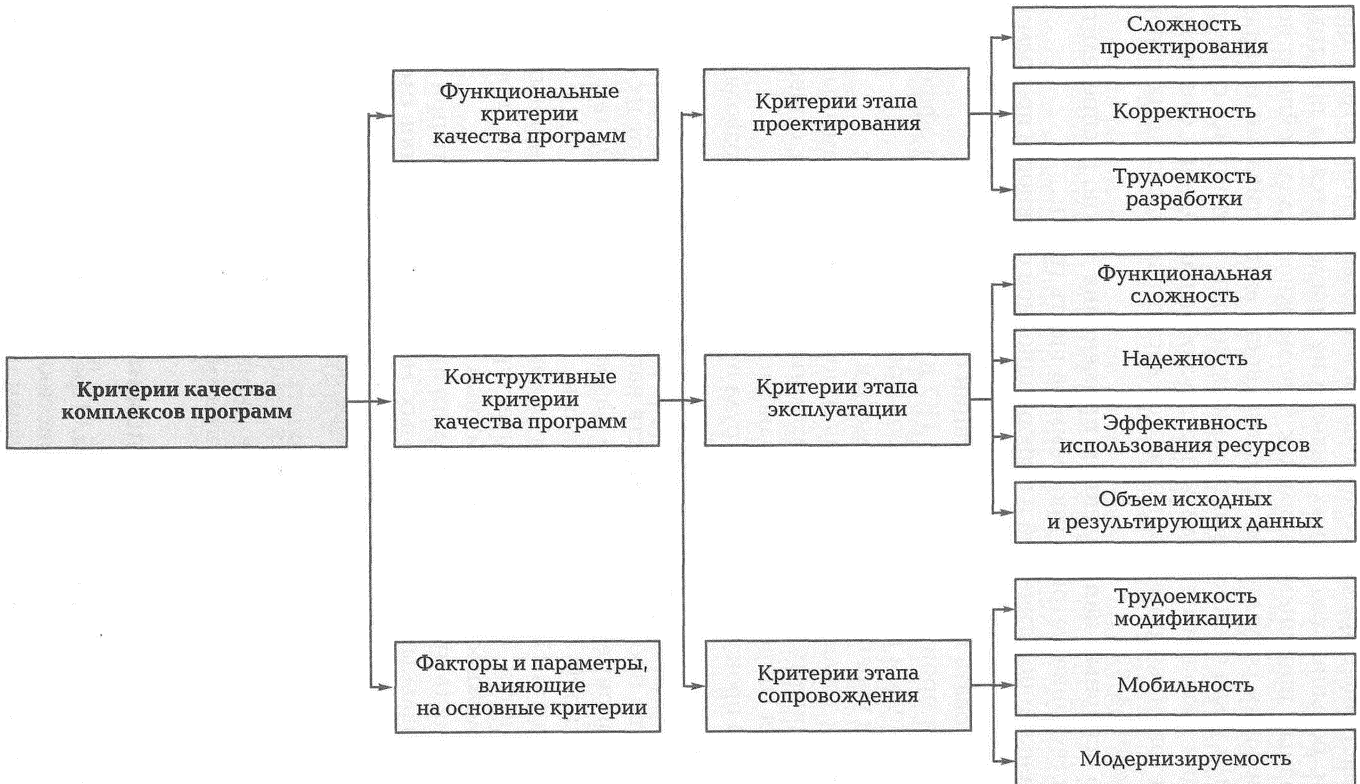


Рис. 3.2. Взаимодействие основных критериев качества программ

В настоящее время насчитывается несколько сотен метрик программ, большинство из них никогда не применяется на практике по разным причинам, таким как невозможность дальнейшего использования результатов, невозможность автоматизации измерений или узкая специализация данных метрик, поэтому все существующие метрики приводить нецелесообразно. Однако существуют метрики, которые применяются достаточно часто, и их обзор приведен далее.

По **основным направлениям применения** можно распределить существующие метрики программ на шесть групп:

- оценки топологической и информационной сложности программ;
- оценки надежности программных систем, позволяющие прогнозировать возможные ситуации отказов;
- оценки производительности ПО и повышения его эффективности путем выявления ошибок проектирования;
- оценки уровня языковых средств и их применения;
- оценки трудности восприятия и понимания программных текстов, ориентированные на психологические факторы, существенные для сопровождения и модификации программ;
- оценки производительности труда программистов, позволяющие прогнозировать сроки разработки программ и планировать работы по созданию программных комплексов.

Качество программных средств напрямую зависит от их сложности. Например, чем сложнее разрабатываемое ПО, тем ниже его надежность и сопровождаемость.

Сложность ПО определяется рядом факторов. Основными из них являются сложность архитектуры программного проекта, а также сложность кода программных модулей.

Сложность программных средств принято оценивать при помощи специальных мер, называемых ***метриками оценки сложности программ***. Согласно отчетности IBM Rational ClearCase среди них, как правило, выделяют четыре основные **группы метрик**:

- размера программ;
- сложности потока управления программ;
- сложности потока данных программ;
- стилистики и понятности программ.

3.2.2. Метрики сложности

Метрики размера программ. Прежде всего следует рассмотреть количественные характеристики исходного кода программ, которые ввиду простоты получили широкое распространение.

Традиционной характеристикой размера программ является количество строк исходного текста (Source Lines of Code, SLOC). Под строкой понимается любой оператор программы, поскольку именно оператор, а не отдельно взятая строка является тем интеллектуальным «кирпичиком» программы, опираясь на который можно строить метрики сложности ее создания.

Несмотря на свою простоту, измерение размера программы дает хорошие результаты. Конечно, метрик размера программы недостаточно для принятия решения о ее сложности, но они вполне пригодны для классификации программ, различающихся по объему.

Уменьшение различий в объеме между программами выдвигают на первый план оценки других факторов, которые оказывают влияние на сложность.

Метрики размера программы — оценки по номинальной шкале (согласно ГОСТ Р ИСО/МЭК 27004—2011 «Информационная технология. Методы и средства обеспечения безопасности. Менеджмент информационной безопасности. Измерения» шкала, которая классифицирует программы на типы по признаку наличия или отсутствия некоторой характеристики без учета градаций), на основе которой определяются только категории программ без уточнения оценки для каждой категории.

К группе оценок размера программ можно отнести **метрику М. Холстеда**.

Основу метрики составляют шесть измеряемых характеристик программы:

- n_1 — число уникальных операторов программы, включая символы-разделители, имена процедур и знаки операций (словарь операторов);
- n_2 — число уникальных операндов программы (словарь операндов);
- N_1 — общее число операторов в программе;
- N_2 — общее число операндов в программе;
- n'_1 — теоретическое число уникальных операторов;
- n'_2 — теоретическое число уникальных операндов.

Опираясь на эти характеристики, получаемые непосредственно при анализе исходных текстов программ, М. Холстед вводит следующие оценки:

- словарь программы:

$$n = n_1 + n_2;$$

- длина программы:

$$N = N_1 + N_2; \qquad (3.1)$$

- измеряемый в битах объем программы:

$$V = N\log_2 n. \tag{3.2}$$

Под битом подразумевается логическая единица информации — символ, оператор, операнд.

Теоретический словарь программы n', т. е. словарный запас, необходимый для ее написания:

$$n' = n'_1 + n'_2.$$

Используя n', Холстед вводит оценку V:

$$V' = n'\log_2 n', \tag{3.3}$$

с помощью которой описывается потенциальный объем программы, соответствующий максимально компактному тексту программы, которая реализует данный алгоритм.

Метрики сложности потока управления программ. Это вторая наиболее крупная группа оценок сложности программ. Она основана не на количественных показателях, а на анализе управляющего графа программы. С помощью этих оценок обычно оперируют либо плотностью управляющих переходов внутри программ, либо взаимосвязями этих переходов.

В случае метрик сложности потока управления для 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 (рис. 3.3). При построении данного графа необходимо условие: каждая вершина достижима из начальной вершины, а конечная вершина — из любой другой вершины. Таким образом, стало традиционным представление программ в виде управляющего ориентированного графа $G = (V, E)$, где V — вершины, соответствующие операторам, а E — дуги, соответствующие переходам.

Первой из оценок сложности потока управления программ, которую мы рассмотрим, будет **метрика Т. Мак-Кейба**. Мак-Кейб пер-

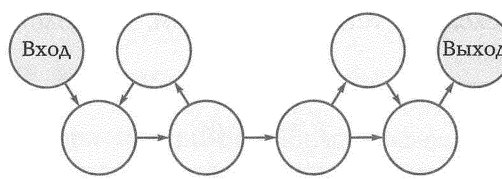


Рис. 3.3. Представление программы в виде ориентированного графа

вым представил программы в графическом виде. Основной метрикой сложности он предложил считать цикломатическую сложность графа программы, или цикломатическое число Мак-Кейба, которое характеризует трудоемкость тестирования программы.

Трудоемкость тестирования программы определяется как

$$V(G) = e - n + 2p,$$

где e — число дуг; n — число вершин; p — число компонентов связности.

Число компонентов 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. На примере графа, изображенного на рис. 3.3, это равнозначно добавлению дуги между узлами выхода и входа. По сути, $V(G)$ определяет число линейно независимых контуров в сильно связном графе. Иначе говоря, цикломатическое число Мак-Кейба показывает требуемое количество проходов для покрытия всех контуров сильно связного графа или количество тестовых прогонов программы, необходимых для исчерпывающего тестирования по критерию «работает каждая ветвь». Так что в корректно написанных программах $p = 1$, и поэтому формула для расчета цикломатической сложности приобретает указанный выше вид.

К сожалению, данная оценка 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 (под предикатом понимается логическое выражение, содержащее хотя бы одну переменную).

Для исправления данного недостатка была разработана новая метрика — **метрика Маерса**. Г. Маерс в качестве оценки предложил взять интервал (эту оценку еще называют интервальной) $[V(G), V(G) + h]$, где h для простых предикатов равно нулю, а для n-местных $h = n - 1$. Таким образом, метрика Маерса позволяет различать программы, представленные одинаковыми графами. К сожалению, информация о результатах использования этого метода отсутствует, поэтому ничего нельзя сказать о его применимости.

Для анализа программ, при создании которых использовалось неструктурное кодирование на таких языках, как язык ассемблера и Фортран, используется **метрика подсчета точек пересечения**, авторами которой являются М. Вудвард, М. Хенел и Д. Хидлей.

В графе программы, где каждому оператору соответствует вершина, т. е. не исключены линейные участки, при передаче управления от вершины a к b номер оператора a равен $\min(a, b)$, а номер оператора b — $\max(a, b)$. Точка пересечения дуг появляется, если:

```
min(a, b) < min(p, q) < max(a, b) & max(p, q) > max(a, b)/
min(a, b) < max(p, q) < max(a, b) & min(p, q) < min(a, b)
```

Другими словами, точка пересечения дуг возникает в случае выхода управления за пределы пары вершин (a, b).

Количество точек пересечения дуг графа программы дает характеристику неструктурированности программы.

Одной из наиболее простых, но, как считают разработчики, достаточно эффективных оценок сложности программ является **метрика Т. Джилба**, в которой 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 n, таким образом:

$$cl = \frac{CL}{n}.$$

Достаточно высокий интерес представляет оценка сложности программ по **методу граничных значений**. В данном случае рассматривается $G = (V, E)$ — ориентированный граф программы с единственной начальной и единственной конечной вершинами. В этом графе число входящих вершин у дуг называется отрицательной степенью вершины, а число исходящих из вершины дуг — положительной. Тогда набор вершин графа можно разбить на две группы — вершины, у которых положительные степени $\leqslant 1$ и $\geqslant 2$. Вершины первой группы назовем 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.

Число вершин, образующих такой подграф, равно скорректированной сложности вершины отбора. Каждая принимающая вершина имеет скорректированную сложность, равную единице, кроме конечной вершины, скорректированная сложность которой равна нулю.

Скорректированные сложности всех вершин графа G суммируются, образуя абсолютную граничную сложность программы. После этого определяется относительная граничная сложность программы:

$$S_o = \frac{1-(v-1)}{S_a},$$

где S_o, S_a — соответственно относительная и абсолютная граничная сложность программы; v — общее число вершин графа программы.

Метрики сложности потока данных. Это следующая крупная группа метрик сложности программ, которые применяют для оценки использования, конфигурации и размещения данных в программах.

Первой оценкой сложности потока данных, которую мы рассмотрим, будет **метрика обращения к глобальным переменным**, которая связывает сложность программы с обращениями к глобальным переменным.

В данном случае 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.

Характеристика A_{Up} 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 P_{Up} определяется по формуле

$$R_{U_p} = \frac{A_{U_p}}{P_{U_p}}.$$

Формула показывает приближенную вероятность ссылки произвольного модуля на произвольную глобальную переменную. Чем выше эта вероятность, тем выше вероятность «несанкционированного» изменения какой-либо переменной, что может существенно осложнить работы, связанные с модификацией программы. Пока нельзя сказать, насколько удобен и точен этот метод на практике, так как соответствующие статистические данные отсутствуют.

Следующей рассматриваемой оценкой является **метрика спена**. Определение спена 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 $n - 1$. При большом спене усложняются тестирование и отладка.

Метрика Чепина предлагает метод, суть которого 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 четыре функциональные группы:

- P — вводимые переменные для расчетов и обеспечения вывода, например, используемая в программах лексического анализатора переменная, содержащая строку исходного текста программы, т. е. сама переменная, не модифицируется, а только содержит исходную информацию;
- M — модифицируемые, или создаваемые внутри программы, переменные;
- C — переменные, участвующие в управлении работой программного модуля (управляющие переменные);
- T — неиспользуемые в программе («паразитные») переменные.

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

Далее вводится значение метрики Чепина:

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

где a_1—a_4 — весовые коэффициенты, которые используются 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. Весовые коэффициенты остальных групп распределяются следующим образом: $a_1 = 1$, $a_2 = 2$, $a_3 = 3$, $a_4 = 0{,}5$. Весовой коэффициент группы T не равен нулю, поскольку «паразитные» переменные не увеличивают сложность потока данных программы, но иногда затрудняют ее понимание. С учетом весовых коэффициентов значение метрики Чепина принимает вид

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

Следует отметить, что рассмотренные метрики сложности программ основаны на анализе исходных текстов программ и графов, что обеспечивает единый подход к автоматизации из расчета.

3.2.3. Метрики стилистики и понятности

Другая группа метрик оценки сложности программ — метрики стилистики и понятности, показывающие полноту комментирования и документирования исходного кода. При этом учитывается не просто число строк комментариев, а то, насколько плотно комментирован исходный код. Например, код сначала был документирован хорошо, затем плохо или так, что шапка функции или класса документирована и комментирована, а код — нет. Суть метрики проста: код разбивается на n равных кусков и для каждого из них определяется коэффициент.

Наиболее простой метрикой стилистики и понятности программ является **оценка уровня комментированности программы** F, вычисляемая как отношение

$$F = \frac{N_{\text{ком}}}{N_{\text{стр}}},$$

где $N_{\text{ком}}$ — число комментариев в программе; $N_{\text{стр}}$ — число строк или операторов исходного текста.

Таким образом, метрика F отражает насыщенность программы комментариями.

Исходя из практического опыта принято считать, что $F \geq 0{,}1$, т. е. на каждые десять строк программы должен приходиться минимум один комментарий. В свою очередь, 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. д. Такая насыщенность компенсирует недостаток комментариев в теле программы, и поэтому отношение количества комментариев к числу строк недостаточно точно отражает комментированность функциональной части текста программы.

Более удачен вариант, когда вся программа разбивается на n равных сегментов и для каждого из них определяется F_i:

$$F_i = \text{sign}(\frac{N_{\text{ком}}}{N_{\text{стр}}} - 0{,}1),$$

тогда

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

Уровень комментированности программы считается нормальным, если $F = n$. В противном случае какой-либо фрагмент программы дополняется комментариями до номинального уровня.

Продолжением метрики М. Холстеда являются указанные ниже характеристики.

1. Для измерения теоретической длины программы $\widehat{N}$ Холстед вводит *аппроксимирующую формулу*

$$\widehat{N} = n_1 \log_2(n_1) + n_2 \log_2(n_2). \qquad (3.4)$$

Вводя эту оценку, Холстед исходит из основных концепций теории информации, по аналогии с которыми частота использования операторов и операндов в программе пропорциональна двоичному логарифму количества их типов. Таким образом, выражение (3.4) представляет собой идеализированную аппроксимацию (3.1), т. е. справедливо для потенциально корректных программ, свободных от избыточности или несовершенств (стилистических ошибок).

Несовершенствами можно считать следующие ситуации:

- последующая операция уничтожает результаты предыдущей без их использования;
- присутствуют тождественные выражения, решающие совершенно одинаковые задачи;
- одной и той же переменной назначаются различные имена, и т. д.

Подобные ситуации приводят к изменению N без изменения n.

Холстед утверждает, что для стилистически корректных программ отклонение в оценке теоретической длины $\widehat{N}$ от реальной N не превышает 10%.

Предлагается использовать $\widehat{N}$ как эталонное значение длины программы со словарем n. Длина корректно составленной программы N, т. е. программы, свободной от избыточности и имеющей словарь n, не должна отклоняться от теоретической длины программы $\widehat{N}$ более чем на 10%. Таким образом, измеряя n_1, n_2, N_1 и N_2 и сопоставляя значения N и $\widehat{N}$ для некоторой программы, при более чем 10%-ном отклонении можно говорить о наличии в программе стилистических ошибок, т. е. несовершенств. На практике N и $\widehat{N}$ часто существенно различаются.

2. Другой характеристикой, принадлежащей к метрикам корректности программ по Холстеду, является *уровень качества программирования* L (уровень программы):

$$L = \frac{V'}{V}, \tag{3.5}$$

где V и V' определяются соответственно выражениями (3.2) и (3.3).

Для введения этой характеристики послужило предположение о том, что при снижении стилистического качества программирования уменьшается содержательная нагрузка на каждый компонент программы и, как следствие, расширяется объем реализации исходного алгоритма. Учитывая это, можно оценить качество программирования на основании степени расширения текста относительно потенциального объема V'. Очевидно, для идеальной программы $L = 1$, а для реальной — всегда $L < 1$.

3. Нередко целесообразно определить уровень программы, не прибегая к оценке ее теоретического объема, поскольку список параметров программы часто зависит от реализации и может быть искусственно расширен. Это приводит к увеличению метрической характеристики качества программирования. Холстед предлагает аппроксимировать эту оценку выражением, включающим только фактические параметры, т. е. параметры реальной программы:

$$L' = \frac{2n_2}{n_2 N_2}.$$

4. Располагая характеристикой L', Холстед вводит характеристику I, которую рассматривает как интеллектуальное содержание

конкретного алгоритма, инвариантное по отношению к используемым языкам реализации:

$$I = L'V. \tag{3.6}$$

По мнению автора, термин «интеллектуальность» не совсем удачен. Преобразуя выражение (3.6) с учетом (3.5), получаем

$$I = L'V = LV = \frac{V'V}{V} = V'.$$

Тогда эквивалентность I и V' свидетельствует о том, что мы имеем дело с характеристикой информативности программы.

Введение характеристики I позволяет определить умственные затраты на создание программы. Процесс создания программы условно можно представить как ряд операций, таких как:

- осмысление предложения известного алгоритма;
- запись предложения алгоритма в терминах используемого языка программирования, т. е. поиск в словаре языка соответствующей инструкции, ее смысловое наполнение и запись.

Используя эту формализацию в методике Холстеда, можно сказать, что написание программы по заранее известному алгоритму есть N'-кратная выборка операторов и операндов из словаря программы n, причем число сравнений (по аналогии с алгоритмами сортировки) составит $\log_2(n)$.

Если учесть, что каждая выборка-сравнение, в свою очередь, содержит ряд мысленных элементарных решений, то можно поставить в соответствие содержательной нагрузке каждой конструкции программы сложность и число этих элементарных решений. Количественно это можно характеризовать с помощью характеристики L, поскольку $\frac{1}{L}$ имеет смысл рассматривать как средний коэффициент сложности, влияющий на скорость выборки для данной программы. Тогда *оценка необходимых интеллектуальных усилий по написанию программы* E может быть измерена как

$$E = N' \log_2\left(\frac{n}{L}\right). \tag{3.7}$$

Таким образом, E характеризует число требуемых элементарных решений при написании программы. Однако следует заметить, что E адекватно характеризует лишь начальные усилия по написанию программ, поскольку при построении E не учитываются отладочные работы, которые требуют интеллектуальных затрат иного характера.

Суть интерпретации этой характеристики состоит в оценке затрат не на разработку программы, а на восприятие готовой программы.

При этом вместо теоретической длины программы $\widehat{N}$ используется ее реальная длина:

$$E' = N \log_2\left(\frac{n}{L}\right).$$

Характеристика E' введена исходя из предположения, что интеллектуальные усилия на написание и восприятие программы очень близки по своей природе. Однако если при написании программы стилистические погрешности в тексте практически не должны отражаться на интеллектуальной трудоемкости процесса, то при попытке понять такую программу их присутствие может привести к серьезным осложнениям. Этот посыл достаточно хорошо согласуется с нашими выводами относительно взаимосвязи N и $\widehat{N}$, изложенными выше.

Преобразуя формулу (3.7) с учетом выражений (3.2) и (3.6), получаем

$$E = \frac{VV}{V'}.$$

Такое представление E', а соответственно, и E, так как $E = E'$, наглядно иллюстрирует целесообразность разбиения программ на отдельные модули, поскольку интеллектуальные затраты оказываются пропорциональными квадрату объема программы, который всегда больше суммы квадратов объемов отдельных модулей.

Рассмотрим еще одну метрику, по своему характеру несколько отличающуюся от предыдущих. Она опирается на принцип оценки, при котором используется **измерение флуктуации длин программной документации**.

Исходным является предположение о том, что чем меньше изменений и корректировок вносится в программную документацию, тем более четко были сформулированы решаемые задачи на всех этапах работ.

Неточности и неясности при создании ПО служат причиной увеличения количества корректировок и изменений в документации. И напротив, демпфированный переходный процесс с немногочисленными изменениями длин документов — естественное следствие хорошо обдуманной идеи, качественно проведенного анализа, проектирования и ясной структуры программ. Эти взаимосвязи и являются основными для данного метода оценки, суть которого состоит в следующем.

Предположим, что документация изменяется в дискретные моменты времени t_i, $i = 1, 2, ..., n$. Тогда в любой момент времени t_i текущая длина документа l_i может быть определена как

$$l_i = l_{i-1} + a_i - b_i l_0 = 0,$$

где l_0, l_{i-1} — соответственно начальная и в предыдущий момент времени длина документа; a_i, b_i — соответственно добавляемая и исключаемая части документа.

Далее вводится d_i, представляющая собой отклонение текущей длины документа l_i от конечного значения l_n:

$$d_i = l_n - l_i.$$

Затем рассчитывается интеграл по модулю этого отклонения на интервале от t_i до $t(n)$, представленный в виде суммы

$$H_n = \sum_{i=1}^{n-1} | d_i | (t_{i+1} - t_i). \qquad (3.8)$$

Значение H_n представляет собой оценку переходного процесса для интервала времени от t_1 до t_n. Однако H_n не учитывает изменений типа $a_i = b_i$, хотя они, бесспорно, влияют на ход дальнейшего процесса.

Чтобы отразить влияние изменений такого рода, называемых в дальнейшем импульсными, вводится экспоненциальная функция, отражающая функцию отклика. Заштрихованная область на рис. 3.4 представляет собой дополнение к оценке H, отражающее влияние импульсного изменения длины документов и вычисляемое как

$$\int_{t_i}^{\infty} a_i e^{-\alpha^{-1}(t-t_i)} dt = \alpha a_i = \alpha b_i, \ \alpha > 0, \qquad (3.9)$$

где α — коэффициент пропорциональности.

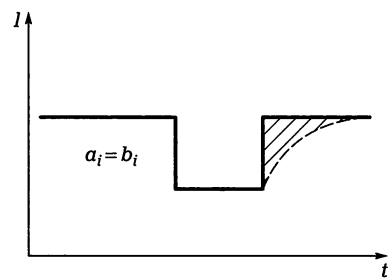


Рис. 3.4. Пример импульсного изменения длины программного документа

Таким образом, оценка длины документа пропорциональна значению импульсного изменения длины $a_i = b_i$ с коэффициентом пропорциональности α.

В принципе, импульсное изменение длины документа присутствует и при $a_i \neq b_i$. Поэтому с учетом (3.9) выражение (3.8) получает вид

$$H'_n = \sum_{i=1}^{n-1} [| d_i | (t_{i+1} - t_i) + \alpha c_i], \qquad (3.10)$$

причем $c_i = \min\{a_i, b_i\}$.

Если в процессе работы значения a_i и b_i неконтролируемы, импульсное изменение длины учесть нельзя. Тогда $c_i = 0$, и выражение (3.10) возвращается к виду (3.8). Используя конечное значение длины документа, можно записать

$$H''_n = \frac{H'_n}{l_n}.$$

Необходимо уточнить некоторые моменты, существенные для применения этой метрики на практике. В действительности программная документация неоднородна по своему содержанию, так как встречаются документы, сильно зависящие от текста программы, но существует и документация, отражающая постановку задачи, имеются спецификации и ведомости, а также материалы, относящиеся к испытаниям программ. Целесообразно использовать данную метрику для оценки качества прежде всего эксплуатационной документации, исключая формуляр и ведомость эксплуатационных документов. Отдельно оцениваются техническое задание с пояснительной запиской и описание программ.

Остальные документы чаще всего отражают изменения в перечисленных документах или неявно вызывают эти изменения.

Представляет интерес сам подход, дающий возможность оценивать динамику характеристик. Ведь если вместо текущей длины документа использовать какую-либо иную характеристику, например длину программы, то все исходные предпосылки останутся в силе. Измеряя в определенные моменты времени значения исследуемой характеристики, мы можем делать выводы об ее динамике. По этим данным, выявляя резкие скачки в переходном процессе, можно судить, например, о потере стабильности используемого ПО. Подводя итог, необходимо отметить, что универсальных метрик не существует. Любые метрические характеристики программы должны контролироваться либо в зависимости друг

от друга, либо в зависимости от конкретной задачи. Таким образом, любая метрика — это лишь показатель, который сильно зависит от языка и стиля программирования, поэтому ни одну меру нельзя возводить в абсолют и принимать какие-либо решения, основываясь только на ней.

3.3. ИЗМЕРИТЕЛЬНЫЕ МЕТОДЫ ОЦЕНКИ ПРОГРАММ: НАЗНАЧЕНИЕ, УСЛОВИЯ ПРИМЕНЕНИЯ

В настоящее время для исследования и оценки параметров программ достаточно активно используются измерительные методы. Из-за высокой достоверности эти методы становятся достаточно привлекательными для проверки имитационных и аналитических методов оценки характеристик программ, основываясь на убеждении, что результат, полученный на практике, является лучшим критерием истины.

Измерительные методы применяются в следующих случаях:

- измерение параметров потребления программами ресурсов вычислительной системы для устранения дефектов производительности;
- предварительное измерение параметров системы для имитационных или аналитических моделей программ перед их последующим использованием. Это связано со сложностью оценки параметров моделей, особенно при использовании новых системных средств с неизвестными динамическими параметрами;
- проверка адекватности имитационных или аналитических моделей и методов расчета характеристик выполнения программ по результатам моделирования.

Для применения измерительных методов необходимо выполнение следующих **условий**:

- наличие готовой программы, подлежащей измерительному исследованию;
- наличие реальной вычислительной системы (а не ее модели) для прогона программы;
- наличие аппаратных или программных средств проведения измерений;
- создание условий снижения искажений, вносимых в функционирование системы в процессе проведения измерений, до уровня, соответствующего требованиям.

При проведении измерений (рис. 3.5) оцениваемая программа может подаваться на вычислительную систему как извне, в виде рабочей нагрузки, так и быть компонентом вычислительной системы.

Исследуемая вычислительная система содержит набор следующих программно-аппаратных средств:

- средства регистрации параметров потребляемых ресурсов при выполнении данной рабочей нагрузки;
- архив для хранения результатов многочисленных измерений.

Результаты измерений обрабатываются некоторой вычислительной системой (отдельная вычислительная система или та же, на которой снимались измерения, но после выполнения сеанса измерений). Рабочая нагрузка — одна или несколько программ или наборов данных для получения статистики проводимых измерений.

Процесс подготовки и проведения измерений включает следующие три этапа:

- выбор рабочей нагрузки, представительной с точки зрения исследования параметров выполнения программы на исследуемой системе;
- выбор (разработка) средств регистрации параметров потребления ресурсов системы;
- выбор (разработка) алгоритмов расчетов характеристик программ по результатам измерений.

Все эти этапы объединяются в единое целое в зависимости от назначения проводимого эксперимента и его планирования.

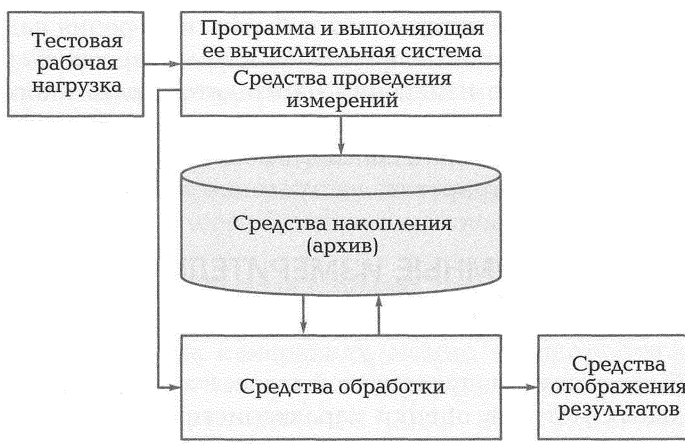


Рис. 3.5. Общая схема проведения измерений

Существуют два основных способа регистрации измеряемых параметров — трассирующий и выборочный.

При **трассирующем способе** измеряемые параметры фиксируются в момент наступления какого-либо события, связанного с изменением управляющих таблиц операционной системы. В результате получаем множество значений r_i потребления i-го ресурса, взятых в моменты времени наступления j-го события: $\{r_i(e_j)\}$, где e — событие (event). В качестве событий обычно рассматриваются обращения к некоторым операторам программы, программным модулям, наборам данных или устройствам системы. Если рассматривается емкостная нагрузка, то берутся наборы данных, а если временная, то выбираются обращения к программным компонентам. Вторая группа событий — это прерывания процессора, как внутренние, так и внешние, наступающие при обмене данными с внешними устройствами.

Трассирующий метод характеризуется меньшим количеством измерений параметров и сильной зависимостью от конкретной рабочей нагрузки.

При **выборочном способе** измерения производятся в моменты времени t_i обычно равноудаленные друг от друга: $t_j = t_{j-1} + \Delta t$, и формируется множество $\{r_i(t_j)\}$, при этом регистрация событий не осуществляется, но в той или иной степени привязывание к событиям осуществляется после.

Выборочный метод имеет на один-два порядка большее число измерений параметров в связи со статистическими методами последующей обработки результатов измерений. Он применяется из-за сложности процесса регистрации событий. При этом, чтобы не пропустить момент наступления события, измерения надо производить часто (через малые промежутки времени), как следствие, возрастает количество измерений (приходится брать много лишних отсчетов измеряемой величины).

Достоинства выборочного способа заключаются в его простоте, так как не надо регистрировать наступление событий.

3.4. ПРОГРАММНЫЕ ИЗМЕРИТЕЛЬНЫЕ МОНИТОРЫ

Как было сказано в предыдущей главе, в ходе реализации измерительных методов оценки характеристик программ производится контроль необходимых параметров в момент наступления определенных событий. Именно для обеспечения регистра-

ции измерений параметров предназначены измерительные мониторы.

К измерительным мониторам предъявляются следующие **требования**:

- минимальные искажения в системе при выполнении программы. Это касается временных искажений, связанных с рассогласованием времени наступления события и времени измерения параметра, и пространственных искажений, связанных с потреблением монитором ресурсов системы, что может осложнить работу с памятью и сказаться на производительности системы в целом;
- достаточная точность измерений (временной интервал между измерениями);
- достаточно высокая разрешающая способность (частота событий, которые возможно отслеживать при помощи монитора);
- независимость от измеряемой системы (программы);
- низкая стоимость;
- простота использования.

Рассмотрим различные типы характеристик, которые обычно получают при помощи измерительных мониторов.

К первому типу относятся **трассировочные записи**. Они представляются множеством пар значений $\{r_i; t_j\}$, где r_i — некоторый i-й параметр потребления ресурсов системы, а t_j — интервал времени между предыдущим и текущим регистрируемым событием либо непосредственно момент регистрации события. Трассировочные записи дают наиболее полную характеристику тестируемой программы, это связано с большим количеством измерений (обычно исчисляемых тысячами) и четкой хронологией их выполнения.

Второй тип характеристик называется **динамическим профилем выполнения программы**. Они делятся на две группы — частотные и временные. Частотные профили определяют количественное распределение потребления рассматриваемых ресурсных параметров, а временные задают распределение времен потребления ресурсов в абсолютном или относительном масштабом (табл. 3.1).

К достоинствам профилей можно отнести выявление узких мест выполнения программ в монопольном режиме. В дальнейшем, основываясь на измерениях, можно улучшить эти участки программы путем изменения алгоритма или заменой программных действий аппаратными.

Недостатком же профиля является отсутствие хронологии потребления ресурсов программой, что не позволяет проводить

Таблица 3.1. Примеры частотного и временного профилей

Выполняемая операция	Частотный профиль, число раз	Масштаб временного профиля	
		абсолютный, мс	относительный, %
Сложение с фиксированной запятой	514	0,5	4,46
Деление	14	2,0	17,86
Операции с плавающей точкой	78	7,0	62,50
Вычитание, сравнение	256	0,3	2,68
Умножение с фиксированной запятой	72	1,4	12,50

анализ для конвейерных систем и взаимодействующих процессов. Также недостатком, характерным как для профилей, так и для трассировочных записей, является зависимость от рабочей нагрузки.

К третьему типу измеряемых характеристик относится **коэффициент загрузки ресурсов** (утилизация) $K_з$, определяемый как отношение $T_{исп}$ — времени потребления (использования) ресурса к $T_{общ}$ — общему времени выполнения программы: $K_з = \frac{T_{исп}}{T_{общ}}$. В качестве ресурса может выступать аппаратный или программный компонент системы.

Ну и наконец, четвертым типом измеряемых характеристик являются **полные**, или **обобщенные**, **характеристики потребления ресурсов программой**, иногда называемые **динамическими смесями** (команд/программ). Они позволяют оценивать для различных совокупностей команд или программ частотные распределения использования ресурсов системы. Для различных программ можно составлять различные смеси (смеси вычислительных задач, смеси для диалоговых сценариев, графических систем и т.д.). Смеси имеют меньшую зависимость от конкретных программ и предметную ориентированность. Позволяют анализировать производительность вычислительной системы в целом.

Измерительные мониторы подразделяют на три группы (рис. 3.6):

- автономные;
- гибридные;
- программные.

Рис. 3.6. Классификация измерительных мониторов

Аппаратные измерительные мониторы легко регистрируют события в аппаратных средствах, но менее ориентированы на измерения параметров, характеризующих выполнение программ. Гибридные измерительные мониторы — более гибкие, специализированные и дорогие измерительные средства, в которых программно фиксируются события, а аппаратным способом осуществляется регистрация параметров при их наступлении. Остановимся подробнее на доступных и широко представленных на рынке ПО программных мониторах.

Программные измерительные мониторы **(*ПИМ*)** — совокупность команд или программ, выполняемых исключительно с целью проведения измерений. Обычно ПИМ — это специальные программные средства, под управлением которых выполняются программы на той же ЭВМ, на которой измеряемая программа и должна выполняться. При этом ПИМ собирает данные о ходе выполнения программ и накапливает их в памяти.

Достаточно распространенной группой являются встроенные ПИМ, которые включаются в состав операционной системы на этапе генерации. Они используются для регистрации фиксированного набора параметров при фиксированном объеме событий. Обычно это журналы регистрации событий. Их задача — определить потребление ресурсов той или иной программой. Например, консоль управления компьютером (Microsoft Management

Console, MMC), которая является компонентом операционной системы Windows 2000 и более поздних версий Windows, позволяет системным администраторам и опытным пользователям с помощью гибкого интерфейса конфигурировать и отслеживать работу системы.

Консоль управления предоставляет более широкие возможности для управления компьютером. Основной принцип действия заключается в оснастках — небольших программах, позволяющих настроить разные аспекты операционной системы. Например, окно `Монитор ресурсов`, изображенное на рис. 3.7, отображает информацию о потребляемых ресурсах системы, таких как загрузка центрального процессора, память, дисковое пространство и сеть, запущенные процессы.

Выполнение команд ПИМ даже самой измеряемой системой приводит к возникновению искажений. Количество искажений зависит от частоты обнаруживаемых событий и от операций, выполняемых измерителем при обнаружении каждого события.

Также далеко не новой является идея использовать измерительные средства в процессе разработки. В настоящее время существует несколько типов таких программных средств. Особое место среди них занимают профилировщики.

Профилировщики (называемые также анализаторами процесса выполнения программ) — программы, позволяющие получить ряд количественных данных о процессе выполнения объекта разработки и на основании этих данных выявить в программе узкие места, отрицательно сказывающиеся на эффективности ее работы.

Профиль программы может содержать, например, следующую информацию о процессе выполнения программы:

- как и на что расходуется время работы программы;
- сколько раз выполняется данная строка программы;
- сколько раз и какими модулями вызывается данный модуль программы.

Для примера можно рассмотреть средства профилирования на этапе отладки приложения в интегрированной среде разработки (IDE) Microsoft Visual Studio Community 2017. Visual Studio предоставляет широкий набор средств профилирования для выявления различных типов проблем с производительностью в зависимости от типа приложения.

Средства профилирования, которыми можно воспользоваться во время сеанса отладки, доступны в окне `Средства диагностики`, которое появится автоматически, если вы не отключали эту функцию.

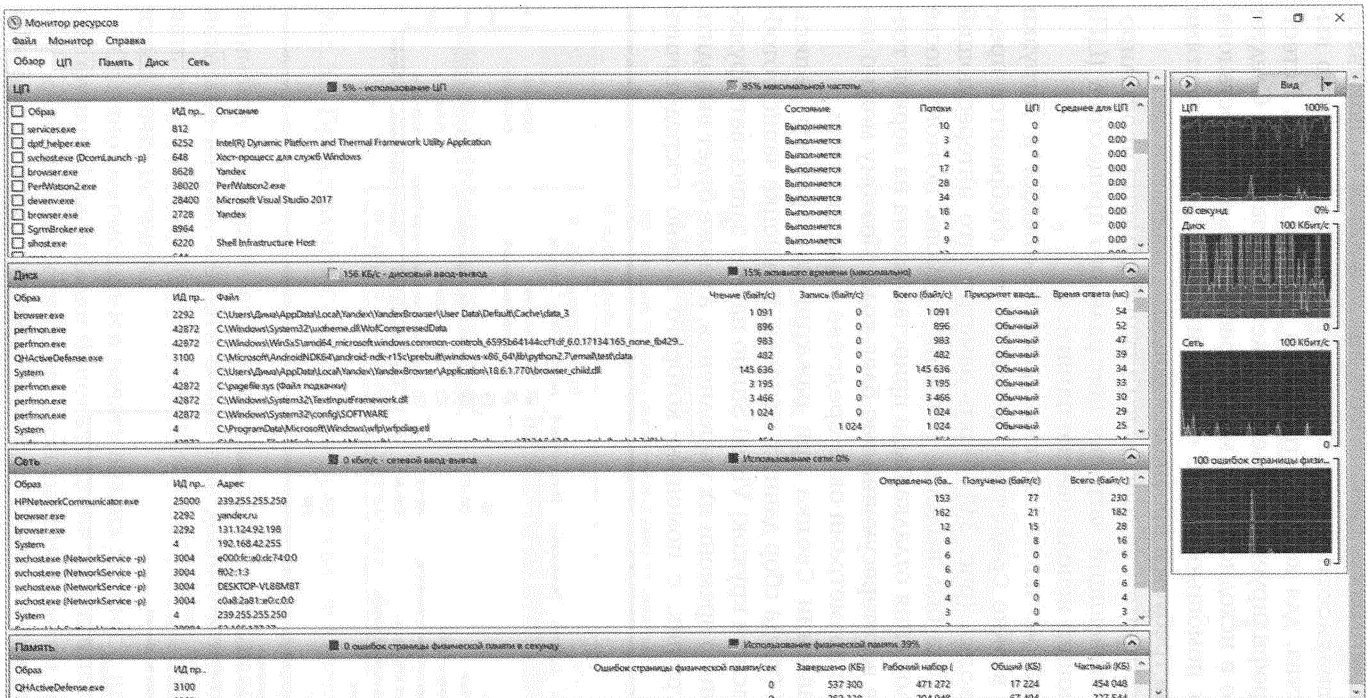


Рис. 3.7. Окно Монитор ресурсов

Чтобы открыть окно, щелкните Отладка → Окна → Показать средства диагностики (рис. 3.8). В открытом окне можно выбирать средства, для которых требуется собрать данные. Естественно, что профилируемый объект должен быть запущен. Следует заметить, что в исходном коде можно расставить точки останова, так как с их помощью можно ограничить сбор данных частями кода, которые требуется проанализировать.

Здесь будут продемонстрированы два инструмента профилирования — оценивающий загрузку центрального процессора (ЦП) и анализирующий использование памяти.

Таким образом, после запуска процесса отладки и загрузки приложения в окне Средства диагностики отобразится представление Сводка (рис. 3.9). Опять заметим, что интересующий нас участок кода был выделен точками останова, поэтому после запуска приложения отладка была приостановлена на первой точке, при входе в интересующий нас блок кода. Поэтому можно начать работу с параметрами окна Средства диагностики.

В представлении Сводка при нажатии на параметр Запись профиля ЦП (когда сбор данных включен, на кнопке записи отображается красный кружок) Visual Studio начнет записывать функции и сведения о времени их выполнения, а также будет выводить временной график, с помощью которого можно сосредоточить

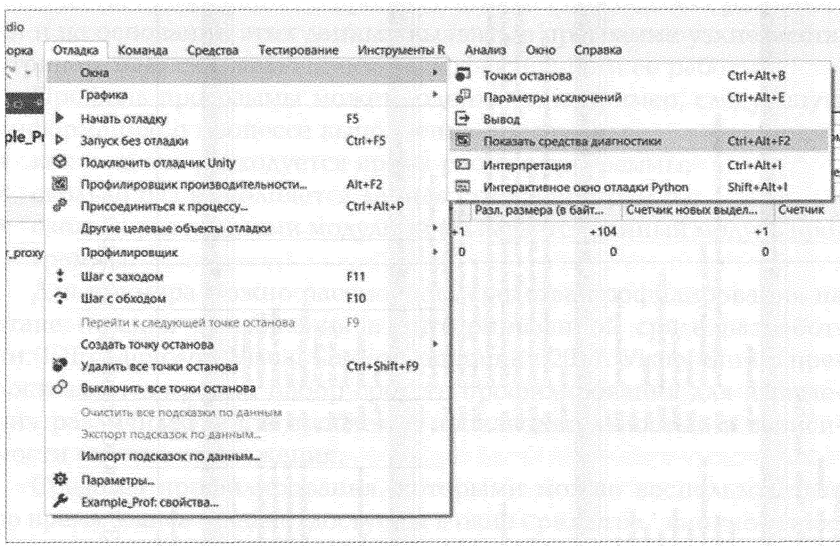


Рис. 3.8. Способ вызова окна Средства диагностики в IDE Visual Studio

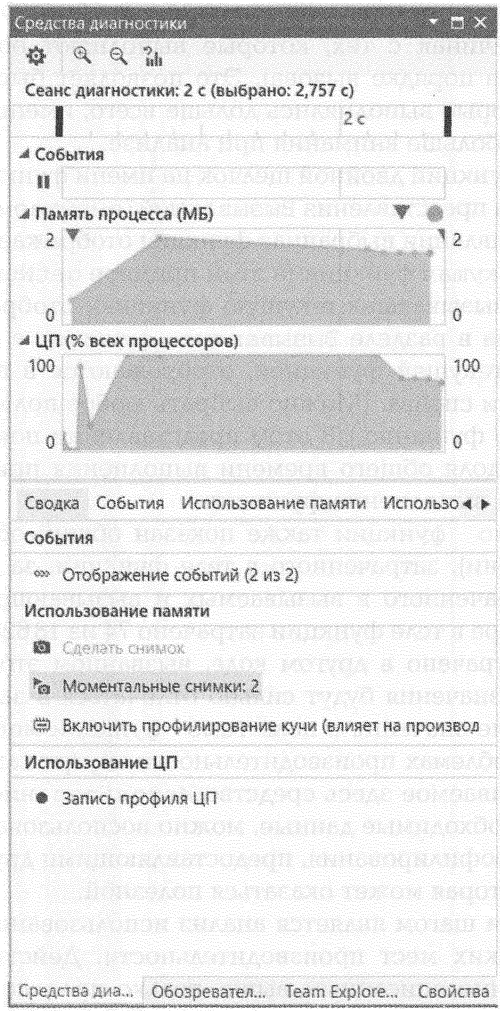


Рис. 3.9. Представление `Сводка` в окне `Средства диагностики`

внимание на определенных частях сеанса профилирования, как показано на рис. 3.9. Эти собранные данные можно просматривать только в том случае, когда приложение остановится в следующей точке останова.

Обычно рекомендуется начать анализ данных с проверки списка функций на вкладке `Использование ЦП` и выявления функций, выполняющих основную часть работы, а затем подробно рассмотреть каждую из этих функций. Список функций отобража-

ется в нижней части окна, как видно на рис. 3.10. Функции перечисляются, начиная с тех, которые выполняют большую часть работы (а не в порядке вызова). Это позволяет быстро находить функции, которые выполнялись дольше всего, именно им необходимо уделить больше внимания при анализе.

В списке функций двойной щелчок на имени функции вызывает открытие окна представления Вызывающий/вызываемый (рис. 3.11). В этом представлении выбранная функция отображается в заголовке и в поле Текущая функция (в этом примере getNumber).

Функция, вызывавшая текущую функцию, отображается в левой части окна в разделе Вызывающие функции, а все функции, вызываемые текущей функцией, отображаются в поле Вызываемые функции справа. (Можно выбрать любое поле, чтобы изменить текущую функцию.) В этом представлении показаны общее время (мс) и доля общего времени выполнения приложения, затраченного на выполнение функции.

В поле Тело функции также показан общий объем времени (и доля времени), затраченного в теле функции, за исключением времени, затраченного в вызываемых и вызывающих функциях. (В этом примере в теле функции затрачено 74 из 18 623 мс, а остальное время затрачено в другом коде, вызванном этой функцией.) Фактические значения будут сильно отличаться в зависимости от среды. Высокие значения в поле Тело функции могут свидетельствовать о проблемах производительности внутри самой функции.

Если описываемое здесь средство Использование ЦП не предоставляет необходимые данные, можно воспользоваться другими средствами профилирования, предоставляющими другие виды информации, которая может оказаться полезной.

Следующим шагом является анализ использования памяти для выявления узких мест производительности. Действия выполняются аналогично описанным выше, только процесс отладки должен быть перезапущен.

С помощью встроенного в отладчик средства диагностики Использование памяти, изображенном на рис. 3.12, можно находить утечки памяти и выявлять ее неэффективное использование. С помощью средства Использование памяти можно сделать один или несколько снимков управляемой и собственной памяти в куче, чтобы понять влияние использования памяти типов объектов.

Существует возможность делать снимки приложений .NET, приложений на основе машинного кода, а также смешанных программ (на основе .NET и машинного кода).

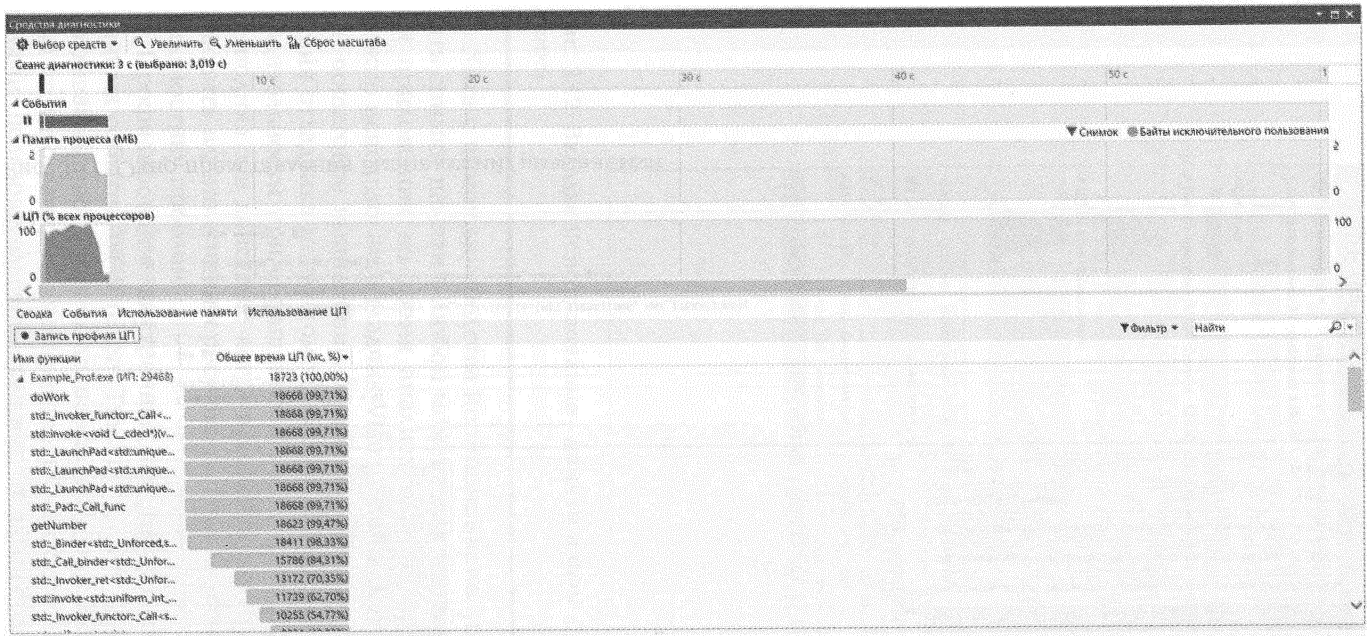


Рис. 3.10. Измерения загрузки центрального процессора, отображаемые в окне `Средства диагностики`

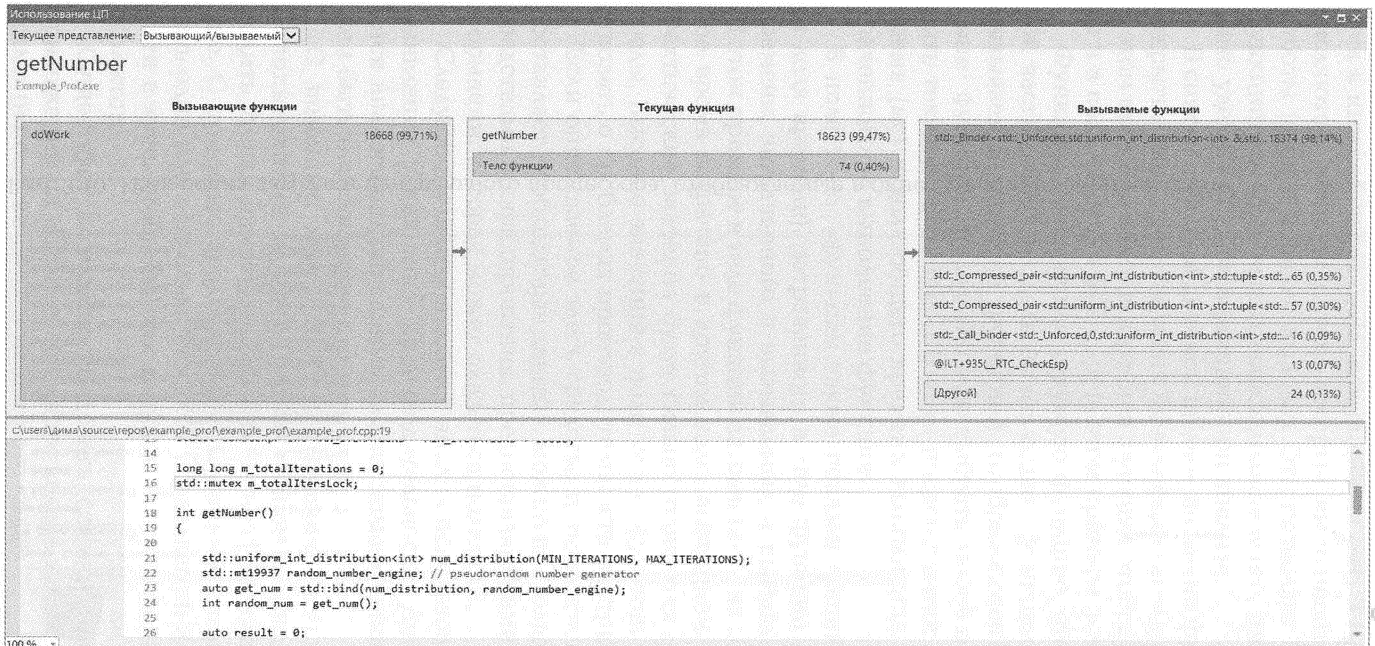


Рис. 3.11. Окно представления Вызывающий/вызываемый

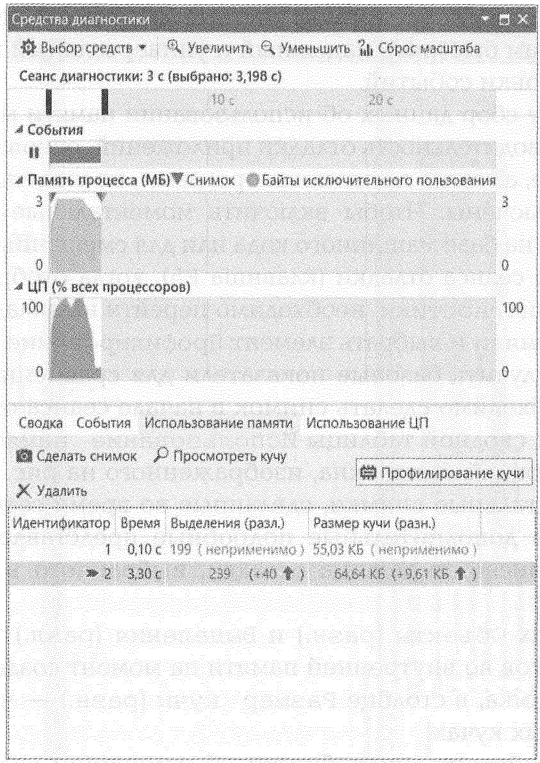


Рис. 3.12. Вкладка Использование памяти в окне Средства диагностики

Хотя с помощью средства Использование памяти можно делать снимки памяти в любой момент, для управления выполнением приложения во время анализа ошибок производительности можно использовать отладчик Visual Studio. Задание точек останова, пошаговое выполнение, всеобщее прерывание и другие действия отладчика могут помочь сосредоточиться на анализе производительности при обращении к наиболее важным ветвям кода. Выполняя эти действия, когда приложение запущено, можно исключить влияние неинтересующего кода и значительно ускорить диагностику проблем.

Профилировщик внутренней памяти работает путем сбора данных событий ETW (Event Tracing Windows) выделения памяти, создаваемых во время выполнения. ETW предоставляет механизм для отслеживания и регистрации событий, вызываемых приложениями пользовательского режима и драйверами режима ядра,

он реализован в операционной системе Windows и предоставляет разработчикам быстрый, надежный и универсальный набор функций трассировки событий.

Поскольку сбор данных об использовании памяти может повлиять на производительность отладки приложений, основанных на машинном коде, а также смешанных программ, по умолчанию снимки памяти выключены. Чтобы включить моментальные снимки для приложений на базе машинного кода или для смешанных программ, после начала сеанса отладки (клавиша F5), когда отобразится окно Средства диагностики, необходимо перейти на вкладку Использование памяти и выбрать элемент Профилирование кучи.

Чтобы получить базовые показатели для сравнения состояния памяти, необходимо сделать снимок в начале сеанса отладки.

В строках сводной таблицы Использование памяти, расположенной в нижней части окна, изображенного на рис. 3.12, приводятся моментальные снимки, сделанные во время сеанса отладки, и ссылки на дополнительные подробные представления. Имена столбцов зависят от режима отладки, выбранного в параметрах проекта.

В столбцах Объекты (разн.) и Выделения (разл.) указывается число объектов во внутренней памяти на момент создания моментального снимка, в столбце Размер кучи (разн.) — число байтов в собственных кучах.

Чтобы отобразить подробности об изменении значения текущего моментального снимка по сравнению с предыдущим, необходимо щелкнуть мышью на разницу в значениях слева от стрелки (Увеличение объема используемой памяти). Красная стрелка обозначает, что объем используемой памяти увеличился, а зеленая — что он снизился. Необходимо заметить, что выводимые в отдельных окнах отчеты будут зависеть от типа проекта. На рис. 3.13 представлен отчет об управляемых типах. Для более быстрого выявления проблемы с памятью типы объектов в отчетах об изменениях можно отсортировать по наибольшему увеличению общего объема.

В Visual Studio 2017 много возможностей по отладке, диагностике и профилированию, которые могут помочь повысить производительность разработчика. Все упомянутые инструменты есть в каждой редакции VS, вплоть до бесплатной Visual Studio Community Edition. При этом профилирование можно осуществлять без выполнения отладки. Таким образом, в Visual Studio 2017 профилирование приложений объединено с процессом удобной отладки, ежедневно применяемой разработчиками.

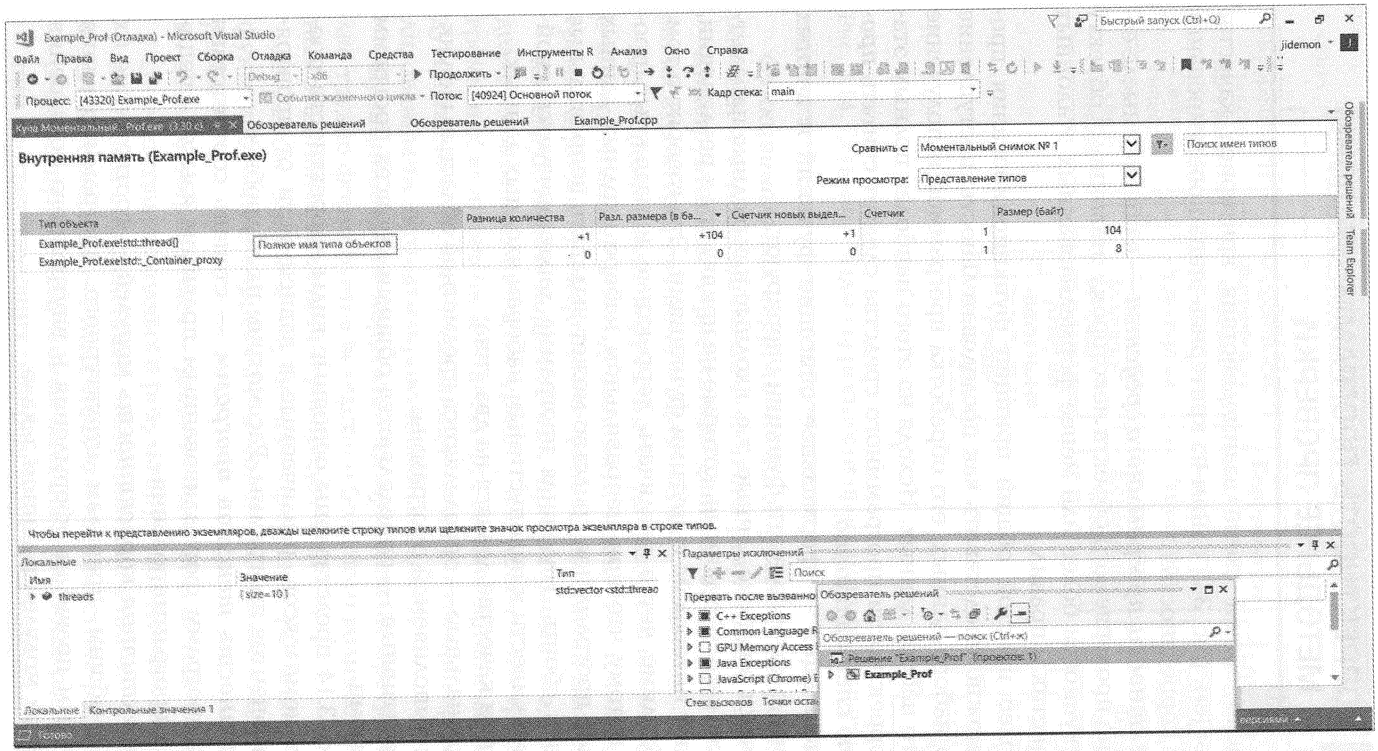


Рис. 3.13. Отчет об управляемых типах моментального снимка № 2

3.5. КОРРЕКТНОСТЬ ПРОГРАММ: ЭТАЛОНЫ И МЕТОДЫ ЕЕ ПРОВЕРКИ

Согласно общей схеме взаимодействия критериев качества программ (см. рис. 3.2) одним из критериев этапа проектирования является корректность.

Корректность программного средства — в общем смысле соответствие проверяемого объекта некоторому эталонному объекту или совокупности более или менее формализованных эталонных характеристик и правил.

Наиболее простой общий принцип функционирования программы можно представить как последовательность: поступление входного набора данных, его обработка программой и получение выходного набора данных. Исходя из этого наиболее полным эталоном корректности программного средства следует считать ***программную спецификацию***.

Спецификация — формальное описание функций и данных программы, с которыми эти функции оперируют.

Различают видимые данные, т. е. входные и выходные параметры, а также скрытые данные, которые не привязаны к реализации, определяют интерфейс с другими функциями. Это говорит о том, что необходимо четко различать корректные и надежные программы, потому что при неправильном наборе входных данных корректное программное средство может выдавать неправильные данные или сбоить. Надежная программа характеризуется свойством устойчивости к недопустимым наборам входных данных.

Критерии качества делятся на два типа:

- функциональные — определяются предметной областью и функциями выполняемой программы;
- конструктивные — определяются общими для всех программ свойствами.

На рис. 3.14 представлены основные виды корректности программных комплексов, различаемые в зависимости от проверяемых компонентов программы. Рассмотрим их подробнее.

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

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

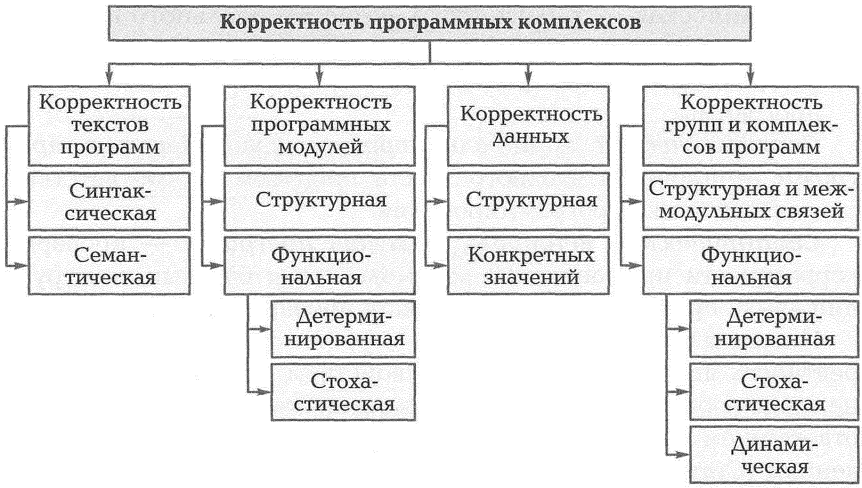


Рис. 3.14. Основные виды корректности программных комплексов

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

Конструктивная корректность данных определяется правилами их структурирования и упорядочения.

Функциональная корректность данных связана в основном с конкретизацией их содержания в процессе исполнения программ, а также при подготовке данных внешними абонентами.

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

Функциональная корректность комплексов программ наиболее трудно формируется вследствие большого количества возможных эталонных значений и распределений.

В самом сложном случае для программ реального времени функциональную корректность можно разделить следующим образом:

- детерминированная — должно быть обеспечено однозначное соответствие исходных и результирующих данных исполняемых программ определенным эталонным значениям;
- стохастическая — статистическое соответствие распределений результирующей случайных величин эталонным распределениям при соответствующих распределениях исходных данных;

- динамическая — характерна для систем реального времени и определяется согласованием во времени порядка поступления входных данных и порядка выдачи результатов выполнения программы.

Синтаксический контроль корректности текстов программ — проверка входного текста программ на соответствие синтаксису языка программирования.

Семантический контроль текстов программ — проверка корректности применения и взаимодействия базовых конструкций языка программирования в тексте проверяемых программ.

Исходя из общего определения корректности и того, что корректность является статическим свойством программы, так как не зависит от времени, необходимо осуществить контроль над соответствием необходимых критериев заданным эталонным значениям. Статическую проверку соответствия системе эталонных правил между структурой программы и последовательностью основных операций обеспечивает формализованный структурный контроль программ, а методом его осуществления является верификация.

Согласно ГОСТ Р ИСО 25010—2015 ***верификация*** (verification) — подтверждение на основе представления объективных свидетельств того, что заданные требования полностью выполнены. Верификация в проектировании и разработке представляет собой процесс анализа результатов конкретных действий для определения соответствия заданным требованиям для этих действий.

Формы эталонов для проверки корректности программ:

- формализованные правила;
- программные спецификации;
- тесты.

На рис. 3.15 изображено взаимодействие компонентов по обнаружению отклонения программ от эталонных значений, а в табл. 3.2 проводится сопоставление методов контроля соответствия эталонам.

Существуют три основных **способа формирования эталонных тестов**:

- *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;

Рис. 3.15. Схема взаимодействия компонентов по обнаружению отклонения программ от эталонных значений

- *использование моделирования на ЭВМ*. Способ универсален. При этом ряд данных моделируется другим способом и по другим алгоритмам, нежели испытываемая программа, и на других ЭВМ. Причем наборы входных данных создаются по случайным законам, что обеспечивает высокую гибкость этого способа;
- *использование результатов испытаний предшествующих вариантов программ*. 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 максимальная 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.

Для полного контроля корректности программы помимо метода верификации следует использовать и метод валидации. Именно верификация и валидация обеспечивают проверку полноты, непротиворечивости и однозначности спецификации и правильности выполнения функций системы в соответствии с требованиями. Если верификация устанавливает соответствия между программой и ее спецификацией, то валидация устанавливает соответствия между тем, что делает программа, и тем, что нужно заказчику.

Таблица 3.2. Типы эталонов и методы проверки корректности программ

Эталон	Метод
Формализованные правила: описания программ; описания данных; структуры модулей; структуры комплекса	Проверка соответствия формализованным правилам: синтаксический контроль; семантический контроль; структурный контроль модулей; контроль структуры межмодульных связей
Программные спецификации: на модули; данные; группы программ; комплекс программ	Проверка программных спецификаций: контроль полноты спецификаций; контроль связи модулей по информации и по управлению; верификация программ
Тесты: детерминированные; стохастические; динамические	Тестирование: 1) детерминированное: планирование тестирования; контроль полноты проверок; корректность корректировок; корректность сравнения с эталонами; 2) стохастическое: полнота исходных данных; корректность обработки результатов; корректность сравнения с эталонами; 3) динамическое: полнота выполнения функций; корректность использования ресурсов ЭВМ; надежность функционирования

3.6. ИССЛЕДОВАНИЕ ПРОГРАММНОГО КОДА НА ПРЕДМЕТ ОШИБОК И ОТКЛОНЕНИЯ ОТ АЛГОРИТМА

Какой бы ценностью ни обладал разрабатываемый программный продукт, гарантировать отсутствие в нем ошибок не может никто. История знает множество достаточно крупных программных проектов, содержащих достаточно большое количество ошибок, приводящих к различным последствиям — от простых зависаний и вылетов из программы до критически важных, приводящих к достаточно ощутимым техногенным авариям. Естественно, что

основной причиной появления ошибок в программах является разработчик. Команды, работающие над большими и сложными проектами, состоят из специалистов с различным опытом и виденьем решения задачи, поэтому появление ошибок в коде неизбежно, а задача по их выявлению становится все более актуальной в связи с ростом сложности разрабатываемого ПО.

Согласно международному стандарту ANSI/IEEE-729-83 все ошибки в разработке программ разделяют на следующие типы:

- ***ошибка*** (error) — состояние программы, при котором выдаются неправильные результаты, причиной которых являются изъяны (flaw) в операторах программы или в технологическом процессе ее разработки, что приводит к неправильной интерпретации исходной информации, следовательно, и к неверному решению;
- ***дефект*** (fault) — следствие ошибок разработчика на любом этапе разработки, может содержаться в исходных или проектных спецификациях, текстах кодов программ, эксплуатационной документации и т. д. В процессе выполнения программы может быть обнаружен дефект или сбой;
- ***отказ*** (failure) — отклонение программы от функционирования или невозможность программы выполнять функции, определенные требованиями и ограничениями, что рассматривается как событие, способствующее переходу программы в неработоспособное состояние из-за ошибок, скрытых в ней дефектов или сбоев в среде функционирования. Отказы, как правило, являются результатами одной или более ошибок в программе, а также наличия разного рода дефектов.

Ранее уже упоминался такой метод оценки корректности ПО, как верификация, целью которого является обнаружение ошибок, уязвимостей, некорректно реализованных свойств и требований. Существующие методы верификации ПО можно разделить на эмпирические (использующие экспертизу), формальные (использующие математический аппарат для верификации ПО) и динамические (проверяющие работу программной реализации с помощью непосредственного запуска), а с точки зрения уровня автоматизации — на ручные, автоматизированные и автоматические.

Так как верификацией ПО является его проверка на соответствие заявленным качественным характеристикам ПО, перечислим наиболее важные из них:

- корректность (соответствие системы своему предназначению);
- безопасность системы;
- устойчивость системы в случае недетерминированного поведения окружения (например, неверные входные данные);

- эффективность использования ресурсов времени и памяти;
- адаптируемость системы к небольшим изменениям окружения;
- переносимость;
- совместимость.

Существует ряд характеристик, позволяющих дать оценку различным методам верификации:

- уровень автоматизации — определяет, в какой степени можно автоматизировать алгоритмы того или иного метода верификации;
- уровень функциональной пригодности — определяет, насколько широкий круг задач покрывает метод верификации ПО;
- точность — характеристика качества получаемых измерений, показывает, насколько погрешность метода (оценка отклонения измеренного значения величины от ее истинного значения) стремится к нулю;
- типы обнаруживаемых ошибок;
- эффективность — определяет продуктивность метода;
- область применимости — определяет, на каком этапе разработки применим тот или иной метод верификации ПО, а также к каким артефактам ПО можно применить метод верификации. Артефактом ПО называется исследуемый участок кода программы;
- время выполнения — определяет время, которое требуется для верификации программы;
- способ достижения результата — определяет методы и алгоритмы, с помощью которых инструменты анализа осуществляют верификацию ПО.

Самым распространенным из эмпирических методов верификации является **экспертиза**. Ее суть заключается в исследовании кода программы наиболее опытным программистом из команды разработчиков, при этом для беспристрастной оценки могут быть приглашены специалисты сторонней организации.

Экспертиза бывает общей и специализированной.

Наиболее интересные виды **общей экспертизы** для разработчиков:

- сквозной контроль — при котором анализ и оценка программы или ее части проводятся группой экспертов, а участники команды последовательно представляют все характеристики программы. Эксперты проводят анализ и вносят замечания, отмечая возможные ошибки и уязвимости;
- инспекция — анализ, при котором поиск ошибок и уязвимостей осуществляется в соответствии с точным планом;
- аудит — анализ программы, выполняемый людьми, не входящими в команду проекта.

Примером **специализированной экспертизы** является экспертиза защищенности, когда контроль разрабатываемого ПО проводится специалистами по информационной безопасности с целью оценки защищенности проекта.

Отличительная особенность экспертизы — отсутствие автоматизации. Стоит отметить, что она применима на всех этапах жизненного цикла ПО, для анализа как всего проекта в целом, так и его отдельных составляющих, а самое главное — имеется возможность отслеживать все виды ошибок.

Формальные методы верификации подразумевают верификацию математической модели программы, а не ее исходный код. В данном случае к программной модели формулируются требования в виде спецификации и проверяется их выполнимость на модели программы.

Формальные методы можно разделить в соответствии со следующими **признаками**:

- дедуктивный анализ;
- проверка моделей;
- проверка согласованности;
- абстрактная интерпретация.

Возможность автоматизации по сравнению с экспертизой является явным преимуществом формальных методов. Тем не менее для построения математической модели всегда необходим квалифицированный специалист.

Формальные методы обладают высокой функциональной пригодностью, а также высокой точностью в случае, если построена адекватная формальная модель.

Построение формальной модели позволяет представить код программы в виде рядов логических выражений, тем самым дает возможность проверить свойства программы, выраженные в виде спецификации. При помощи формальных методов можно выявить следующие **классы ошибок**:

- неопределенное поведение программы;
- неинициализированные переменные;
- обращение к Null-указателям;
- нарушение правил и алгоритмов пользования библиотекой;
- сценарии, приводящие к недокументированному поведению программы;
- переполнение буфера;
- сценарии, мешающие кроссплатформенности;
- ошибки, возникающие в повторяющемся коде;
- ошибки форматных строк.

Основным недостатком методов формальной верификации является сложность (а иногда невозможность) построения наиболее полной и адекватной математической модели. Данный метод применим только к проверяемым участкам, которые можно учесть в формальной модели.

Статический анализ ПО — анализ ПО, производимый (в отличие от динамического анализа) без реального выполнения исследуемых программ.

В большинстве случаев анализ производится над какой-либо версией исходного кода, хотя иногда анализу подвергается какой-нибудь вид объектного кода. Для анализа обычно применяется специальное ПО.

В зависимости от используемого инструмента глубина анализа может варьироваться — от определения поведения отдельных операторов до анализа, включающего весь имеющийся исходный код. Способы использования полученной в ходе анализа информации также различны — от выявления мест, возможно содержащих ошибки, до формальных методов, позволяющих математически доказать какие-либо свойства программы (например, соответствие поведения спецификации).

Со статическим анализом сталкиваются даже начинающие программисты. Примером может служить написание кода в большинстве распространенных IDE. Большинство компиляторов (например, GNU C Compiler) выводят на экран «предупреждения» — сообщения о том, что код, будучи синтаксически правильным, скорее всего, содержит ошибку:

```
int x;
int y=x+2; // переменная x объявлена, но не инициализирована
```

Часто встречаются случаи, когда в процессе разработки используются какие-либо параметры, например передаваемые какому-нибудь методу, а потом разработчик просто забывает его удалить после того, как необходимость в данном параметре и логике, связанной с ним, отпадает:

```
void doExampleSom (int n, bool m) // m всегда равно true
{
     if (m)
{
   ..... // логика оператора if
   } else
```

```
    {
    ..... // код есть, но не задействован
    }
}
doExampleSom (n, true);
...
doExampleSom (10, true);
...
doExampleSom (x.size(), true);
```

Эти два примера наглядно демонстрируют простейший статический анализ.

Процесс статического анализа состоит из двух основных шагов — создания дерева кода (также называемого абстрактным деревом синтаксиса) и анализа этого дерева.

Для того чтобы проанализировать исходный код, анализатор должен сначала «понять» этот код, т. е. разобрать его по составу и создать структуру, описывающую анализируемый код в удобной форме. Эта форма и называется деревом кода. Чтобы проверить, соответствует ли код стандарту кодирования, необходимо построить такое дерево.

В общем случае дерево строится только для анализируемого фрагмента кода (например, для какой-то конкретной функции). Чтобы создать дерево, код обрабатывается сначала лексером (фактически проводится лексический анализ кода), а затем синтаксическим анализатором.

Лексер по определенному алгоритму разбивает входные данные на отдельные лексемы, определяет их тип, а затем последовательно передает их синтаксическому анализатору. Исходный код программы считывается лексером построчно, затем полученные строки разбиваются на зарезервированные слова, идентификаторы и константы, называемые лексемами. После получения лексемы лексер определяет ее тип. На рис. 3.16 изображено приблизительное представление процесса обработки лексером функции `int Func () { return 0 ; }`, написанной на языке C. Строка

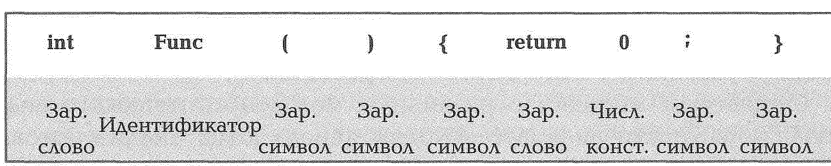


Рис. 3.16. Разбор на лексемы объявления функции лексером

будет распознана как восемь корректных лексем, и эти лексемы переданы синтаксическому анализатору.

Синтаксический анализатор понимает грамматику языка. Он отвечает за обнаружение синтаксических ошибок и преобразование программы, в которой такие ошибки отсутствуют, в структуры данных, называемые деревьями кода. В случае с рассмотренным примером (см. рис. 3.16) он просмотрит контекст и выяснит, что данный набор лексем является объявлением функции, которая не принимает никаких параметров, возвращает целое число, и это число всегда равно нулю.

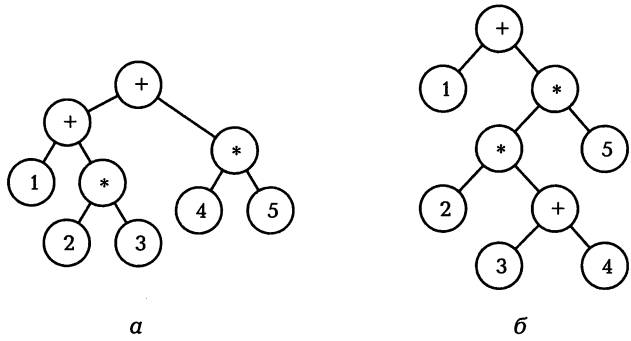
Синтаксический анализатор выяснит это, когда создаст дерево кода из лексем, предоставленных лексером, и проанализирует это дерево. Если лексемы и построенное из них дерево будут сочтены правильными, это дерево будет использовано при статическом анализе. В противном случае синтаксический анализатор выдаст сообщение об ошибке. Эти структуры, в свою очередь, поступают на вход статического анализатора и обрабатываются им.

Дерево кода представляет самую суть поданных на вход данных в форме дерева, опуская несущественные детали синтаксиса. Такие деревья отличаются от конкретных деревьев синтаксиса тем, что в них нет вершин, представляющих знаки препинания вроде точки с запятой, завершающей строку, или запятой, которая ставится между аргументами функции.

При разработке вершин дерева в первую очередь обычно определяется уровень модульности. Иными словами, определяется, будут ли все конструкции языка представлены вершинами одного типа, различаемыми по значениям. В качестве примера рассмотрим представление бинарных арифметических операций. Один вариант — использовать для всех бинарных операций одинаковые вершины, одним из атрибутов которых будет тип операции, например «+». Другой вариант — использовать для разных операций вершины различного типа. В качестве примера разберем два выражения: 1 + 2 * 3 + 4 * 5 и 1 + 2 * (3 + 4) * 5, как показано на рис. 3.17. Как видно из рисунка, оригинальный вид выражения может быть восстановлен при обходе дерева слева направо.

После проверки дерева кода статический анализатор может определить, соответствует ли исходный код правилам и рекомендациям, указанным в стандарте кодирования.

Существует множество различных методов статического анализа, в частности анализ с обходом дерева кода, анализ потока данных, анализ потока данных с выбором пути и т.д. Конкретные реализации этих методов различны в разных анализаторах.



а *б*

Рис. 3.17. Разобранные выражения: 1 + 2 * 3 + 4 * 5 (*а*) и 1 + 2 * (3 + 4) * 5 (*б*)

Тем не менее статические анализаторы для различных языков программирования могут использовать один и тот же базовый код (инфраструктуру). Эти инфраструктуры содержат набор основных алгоритмов, которые могут использоваться в разных анализаторах кода вне зависимости от конкретных задач и анализируемого языка. Набор поддерживаемых методов и конкретная реализация этих методов опять же будут зависеть от конкретной инфраструктуры. Например, инфраструктура может позволять легко создавать анализатор, использующий обход дерева кода, но не поддерживать анализ потока данных.

Хотя все три перечисленные выше методы статического анализа используют дерево кода, построенное синтаксическим анализатором, эти методы различаются по своим задачам и алгоритмам.

Анализ с обходом дерева, как видно из названия, выполняется путем обхода дерева кода и проведения проверок на предмет соответствия кода принятому стандарту кодирования, указанному в виде набора правил и рекомендаций. Именно этот тип анализа проводят компиляторы.

Анализ потока данных можно описать как процесс сбора информации об использовании, определении и зависимостях данных в анализируемой программе. При анализе потока данных используется граф потока команд, генерируемый на основе дерева кода. Этот граф представляет все возможные пути выполнения данной программы: вершины обозначают «прямолинейные», без каких бы то ни было переходов фрагменты кода, а ребра — возможную передачу управления между этими фрагментами. Поскольку анализ выполняется без запуска проверяемой программы, точно определить результат ее выполнения невозможно.

Иными словами, невозможно выяснить, по какому именно пути будет передаваться управление. Поэтому алгоритмы анализа потока данных аппроксимируют возможное поведение, например, рассматривая обе ветви оператора if—then—else или выполняя с определенной точностью тело цикла while. Ограничение точности существует всегда, поскольку уравнения потока данных записываются для некоторого набора переменных, и количество этих переменных должно быть ограничено, так как мы рассматриваем лишь программы с конечным набором операторов. Следовательно, для количества неизвестных всегда существует некий верхний предел, дающий ограничение точности. С точки зрения графа потока команд при статическом анализе все возможные пути выполнения программы считаются действительными. Из-за этого допущения при анализе потока данных можно получать лишь приблизительные решения для ограниченного набора задач.

Описанный выше алгоритм анализа потока данных не различает путей, поскольку все возможные пути вне зависимости от того, реальны они или нет, будут ли они выполняться часто или редко, все равно приводят к решению. На практике, однако, выполняется лишь малая часть потенциально возможных путей. Более того, самый часто выполняемый код, как правило, составляет еще меньшее подмножество всех возможных путей. Логично сократить анализируемый граф потока команд и уменьшить таким образом объем вычислений, анализируя лишь некоторое подмножество возможных путей. Анализ с выбором путей проводится по сокращенному графу потока команд, в котором нет невозможных путей и путей, не содержащих «опасного» кода. Критерии выбора путей различны в различных анализаторах. Например, анализатор может рассматривать лишь пути, содержащие объявления динамических массивов, считая такие объявления «опасными» согласно настройкам анализатора.

Вместе с усложнением ПО растет число как разрабатываемых методов статического анализа, так и создаваемых анализаторов. Это обусловлено отсутствием возможности анализа больших объемов кода вручную.

В рамках **динамических методов** анализ ПО осуществляют при помощи реального выполнения программы. В случае имитационного моделирования выполняется не сама программа, а программа, ее моделирующая.

На вход программы поступают последовательности данных, которые могут вызвать неопределенное поведение, тем самым по-

зволяя обнаружить уязвимости и ошибки. Динамический анализ можно разделить на несколько видов:

- тестирование;
- мониторинг;
- имитационное тестирование;
- профилирование.

Ранее частично уже рассматривались мониторинг и профилирование, поэтому более подробно остановимся на тестировании.

Тестирование ПО — поиск ситуаций в программном коде, в которых поведение программы становится неопределенным, неправильным и не соответствующим спецификации.

Цели тестирования:

- повышение вероятности того, что приложение, предназначенное для тестирования, будет:
  - ✓ работать правильно при любых обстоятельствах;
  - ✓ соответствовать всем описанным требованиям;
- предоставление актуальной информации о состоянии продукта на данный момент.

Обычно тестирование осуществляется в рамках известных, заданных сценариев. Как правило, методы тестирования включают в себя мониторинг, позволяя создать контролируемую среду выполнения программы, опробовать различные наборы тестов и запротоколировать полученные результаты.

Полноценное тестирование характеризуется тем, насколько хорошо определены цели тестирования, обеспечена полнота тестирования и определены критерии полноты тестирования.

Подробнее рассмотрим стратегии тестирования (рис. 3.18).

Начнем с рассмотрения **тестирования по стратегии «черный ящик»**. При тестировании «черного ящика» тестировщик имеет доступ к программе только через те же интерфейсы, что и заказчик или пользователь, либо через внешние интерфейсы, позволяющие другому компьютеру либо другому процессу подключиться к системе для тестирования. Например, тестирующий модуль может виртуально нажимать клавиши или кнопки мыши в тестируемой программе с помощью механизма взаимодействия процессов с уверенностью в том, что все идет правильно и эти события вызывают тот же отклик, что и реальные нажатия клавиш и кнопок мыши.

Как правило, тестирование «черного ящика» ведется с использованием спецификаций или иных документов, описывающих требования к системе. Обычно в данном виде тестирования критерий покрытия складывается из покрытия структуры входных

Рис. 3.18. Виды тестирования

данных, покрытия требований и покрытия модели (в тестировании на основе моделей).

При тестировании методом «черного ящика» используются различные техники тест-дизайна.

Тест-дизайн — этап процесса тестирования ПО, на котором проектируются и создаются тестовые случаи (тест-кейсы) в соответствии с определенными ранее критериями качества и целями тестирования.

Согласно классификации, приведенной на рис. 3.18, мы рассмотрим:

- *эквивалентное разделение* (equivalence partitioning). Например, имеется диапазон допустимых значений от 1 до 10, вы должны выбрать одно верное значение внутри интервала, например 5, и одно неверное значение вне интервала — 0;
- *анализ граничных значений* (boundary value analysis). Если взять пример выше, в качестве значений для позитивного тестирования выберем минимальную и максимальную границы (1 и 10) и значения больше и меньше границ (0 и 11). Анализ граничных значений может быть применен к полям, записям, файлам или к любого рода сущностям, имеющим ограничения;

- *анализ причинно-следственных связей* (cause/effect). Это, как правило, ввод комбинаций условий (причин) для получения ответа от системы (следствие). Например, вы проверяете возможность добавлять клиента, используя определенную экранную форму. Для этого необходимо ввести несколько полей, таких как `Имя`, `Адрес`, `Номер телефона`, а затем нажать кнопку `Добавить` — это «причина». После нажатия кнопки `Добавить` система добавляет клиента в базу данных и показывает его номер на экране — это «следствие»;
- *предположение об ошибке* (error guessing). Тест-аналитик использует свои знания системы и способность к интерпретации спецификации на предмет того, чтобы «предугадать», при каких входных условиях система может выдать ошибку. Например, спецификация говорит: «Пользователь должен ввести код». Тест-аналитик будет думать: «Что если я не введу код?», «Что если я введу неправильный код?» и т. д. Это и есть предугадывание ошибки.

Тестирование по стратегии «белый ящик» — тестирование кода на предмет логики работы программы и корректности ее работы с точки зрения компилятора языка, на котором она писалась. Техника тестирования по принципу «белого ящика» (также называется «техника тестирования, управляемая логикой программы») позволяет проверить внутреннюю структуру программы. Исходя из этой стратегии тестировщик получает тестовые данные путем анализа логики работы программы, т. е. внутреннее устройство программы известно проводящему тестирование.

Рассмотрим следующие **варианты тест-дизайна** (см. рис. 3.18):

- *покрытие операторов* (statement coverage) — процентное отношение операторов, исполняемых набором тестов, к их общему количеству;
- *покрытие решений* (decision coverage) — процент результатов альтернативы, который был проверен набором тестов, 100%-ное покрытие решений подразумевает 100%-ное покрытие ветвей и 100%-ное покрытие операторов;
- *покрытие условий и решений* (condition coverage) — процент исходов условий, которые были проверены набором тестов, 100%-ное покрытие условий требует, чтобы каждое отдельное условие в каждом выражении решения было проверено как «истина» и «ложь»;
- *комбинаторное покрытие условий* (combinatorial decition coverage) — требует подобрать такой набор тестов, чтобы хотя бы один раз выполнялась любая комбинация простых условий.

Также различные виды тестирования можно классифицировать по **функциональности** на функциональные, нефункциональные и связанные с изменениями.

На рис. 3.18 указаны **нефункциональные виды**:

- *нагрузочное тестирование* — автоматизированное тестирование, имитирующее работу определенного количества бизнес-пользователей на каком-либо общем (разделяемом ими) ресурсе;
- *тестирование в предельных режимах*, или *стрессовое тестирование* (stress testing), — позволяет проверить, насколько приложение и система в целом работоспособны в условиях стресса, и оценить способность системы к регенерации, т. е. к возвращению к нормальному состоянию после прекращения воздействия стресса. Стрессом в данном контексте могут быть повышение интенсивности выполнения операций до очень высоких значений или аварийное изменение конфигурации сервера. Также одной из задач при стрессовом тестировании может быть оценка деградации производительности, таким образом, цели стрессового тестирования могут пересекаться с целями тестирования производительности;
- *тестирование работоспособности*, или *тестирование стабильности или надежности* (stability/reliability testing). Задачей тестирования стабильности (надежности) является проверка работоспособности приложения при длительном (многочасовом) тестировании со средним уровнем нагрузки;
- *тестирование на отказ и восстановление* (failover and recovery testing) — проверяет тестируемый продукт с точки зрения способности противостоять и успешно восстанавливаться после возможных сбоев, возникших в связи с ошибками ПО, отказами оборудования или проблемами связи (например, отказ сети). Целью данного вида тестирования является проверка систем восстановления (или дублирующих основной функционал систем), которые в случае возникновения сбоев обеспечат сохранность и целостность данных тестируемого продукта.

Также следует упомянуть о разделении тестирования на **уровни**:

- *модульное* (*компонентное*) *тестирование* (unit testing) проверяет функциональность и ищет дефекты в частях приложения, которые доступны и могут быть протестированы по отдельности (модули программ, объекты, классы, функции и т. д.);
- *интеграционное тестирование* (integration testing) проводится, когда проверяется взаимодействие между компонентами системы после компонентного тестирования;

- *системное тестирование* (system testing). Его основная задача — проверка как функциональных, так и нефункциональных требований в системе в целом. При этом выявляются такие дефекты, как неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство использования и т. д.;
- *операционное тестирование* (release testing). Даже если система удовлетворяет всем требованиям, важно убедиться, что она удовлетворяет нуждам пользователя и играет свою роль в среде своей эксплуатации, как это было определено в бизнес-модели системы. Следует учесть, что и бизнес-модель может содержать ошибки. Поэтому важно провести операционное тестирование как финальный шаг валидации. Кроме того, тестирование в среде эксплуатации позволяет выявить и нефункциональные проблемы, такие как конфликт с другими системами, смежными в области бизнеса или в программных и электронных окружениях, недостаточная производительность системы в среде эксплуатации и др. Очевидно, что нахождение подобных вещей на стадии внедрения — критичная и дорогостоящая проблема. Поэтому так важно проведение не только верификации, но и валидации с самых ранних этапов разработки ПО;
- *приемочное тестирование* (acceptance testing) — формальный процесс тестирования, который проверяет соответствие системы требованиям и проводится с целью определения, удовлетворяет ли система приемочным критериям, вынесения решения заказчиком или другим уполномоченным лицом, принимается приложение или нет.

Подготовка тестов производится вручную, сам процесс тестирования и мониторинга можно автоматизировать. Методы динамического анализа позволяют провести проверку программы в случае отсутствия исходного кода путем создания контролируемой среды выполнения программы, а также обнаружить множество дефектов и получить наиболее точную оценку качества сложной системы. Набор тестов и систему мониторинга можно использовать многократно. В отличие от статического анализа и формальных методов она позволяет выяснить временные и количественные характеристики ПО, такие как время выполнения программы в целом и время выполнения ее отдельных участков, и количество используемых ресурсов (например, занимаемая приложением

оперативная память). Динамический анализ помимо стандартного набора программных ошибок, приведенного выше, позволяет определить виды ошибок, которые возникают при исполнении программы.

3.7. ПРИМЕНЕНИЕ ОТЛАДЧИКОВ И ДИЗАССЕМБЛЕРА

Существуют два основных подхода к изучению (имеется в виду реверс-инжиниринг ПО) программ — трассировка и дизассемблирование. Конечно, методы отладки имеют ряд неоспоримых преимуществ, но совместное с ними применение методов дизассемблирования способно дать хорошо документированный листинг программы, понять механизм взаимодействия различных ее ветвей, возможность внесения изменений и рекомпилирования продукта. Считается, что отладчик и дизассемблер — это инструменты, используемые совместно для решения серьезных задач, и раздельное их использование малоэффективно. Например, отладчиком возможно перехватывать вызовы функций открытия, чтения, позиционирования файлов, после чего дизассемблировать заданные фрагменты. Анализ листинга исследуемого ПО позволяет определить логику работы программы на уровне взаимодействия различных ветвей кода, а детали и тонкости уточняются применением возможностей отладчика. Эти действия повторяются до тех пор, пока структура файлов не становится очевидной. Именно работа связки «дизассемблер — отладчик» позволяет в сжатые сроки анализировать многомегабайтные файлы.

Дизассемблирование — преобразование программы на машинном языке к ее ассемблерному представлению, а ***декомпиляция*** — получение кода языка высокого уровня из программы на машинном языке или ассемблере.

Декомпиляция — достаточно сложный процесс по следующим причинам:

- процесс компиляции происходит с потерями. В машинном языке отсутствуют имена переменных и функций, и только по производимым над ними операциям может быть определен тип данных. Например, в процессе наблюдения пересылки 32 бит данных требуется много усилий и опыта, чтобы определить, являются ли эти данные целым числом, дробью или указателем;
- компиляция — операция типа «множество — множество». Так как компиляция и декомпиляция могут быть выполнены мно-

жеством способов, результат декомпиляции может значительно отличаться от исходного кода;

- декомпиляторы в значительной степени зависят от конкретного языка и библиотек. Например, обрабатывая исполняемый файл, созданный компилятором языка Pascal, декомпилятором, разработанным для C++, можно получить непредсказуемый результат;
- необходимо точное дизассемблирование исполняемого файла. Любая ошибка или упущение на этапе дизассемблирования с большой долей вероятности приведут к увеличению количества ошибок в результирующем коде.

Прогресс средств декомпиляции происходит неторопливо. Наиболее сложен на сегодняшний день декомпилятор IDA Pro (будет рассмотрен ниже), мощная современная IDE MS Visual Studio имеет соответствующие встроенные возможности.

Цель применения инструментов дизассемблирования и отладки заключается в исправлении ошибок в разрабатываемом ПО и исследовании функционирования программ, когда их исходные коды не доступны.

Наиболее распространенные **цели дизассемблирования**:

- анализ вредоносного ПО;
- анализ уязвимостей ПО с закрытым исходным кодом;
- анализ совместимости ПО с закрытым исходным кодом;
- валидация компилятора;
- отображение команд программы в процессе отладки.

Анализ вредоносного ПО. Сами цели применения вредоносного ПО обусловливают максимальную скрытность и отсутствие информации о принципах его функционирования. Без доступа к исходному коду таких программ задача определения принципов их функционирования и способов влияния на ИС в целом довольно сложна.

Анализ уязвимостей. Процесс аудита безопасности можно разделить на три стадии:

- поиск уязвимостей;
- анализ уязвимостей;
- разработка эксплойта (программы-взломщика, использующей возможности, предоставляемые уязвимостью).

Одни и те же шаги предпринимаются вне зависимости от того, имеется ли у вас исходный код, однако уровень трудоемкости резко возрастает, когда в вашем распоряжении есть лишь исполняемый файл. При обнаружении проблемы требуется определить, является ли она уязвимостью, и если является, то при каких условиях.

Для понимания ряда свойств исследуемого вредоносного ПО используется листинг дизассемблирования. Например, он помо-

гает понять, каким образом компилятор расположил переменные в памяти. К примеру, может оказаться полезной информация, что объявленный программистом 70-байтный массив символов при распределении памяти компилятором был округлен в сторону 80 байтов.

Также листинги дизассемблирования являются единственным способом понимания, каким образом объявлены переменные в исследуемой программе — глобально или внутри функций. Понимание реального расположения переменных в памяти жизненно необходимо в процессе разработки эксплойта.

Анализ совместимости. В случае доступности программы только в виде исполняемых файлов сторонним разработчикам крайне сложно обеспечить совместимость и функциональность своих программ с ними. Например, если производитель не предоставил драйвер для аппаратного устройства, то реверс-инжиниринг — практически единственное средство для разработки альтернативных драйверов.

Валидация компилятора. Дизассемблирование может быть средством для проверки соответствия работы компилятора его спецификации. Также исследователя может заинтересовать наличие дополнительных возможностей, оптимизирующих результат компиляции.

С точки зрения безопасности важно быть уверенным, что код, генерируемый компилятором, не содержит так называемых черных ходов, делающих компилируемое ПО уязвимым к различным злонамеренным воздействиям.

Отладка. Обобщение опыта множества разработчиков показывает, что дизассемблеры, встроенные в отладчики, зачастую малоэффективны. Они не зарекомендовали себя как способные к серийному дизассемблированию инструменты, поэтому для лучшего контроля над процессом отладки целесообразно использовать отладчик в сочетании с хорошим дизассемблером.

Типичными задачами дизассемблирования являются определение сегмента декомпиляции из исходного файла, отделение кода от данных и преобразование его с минимальными потерями к языку ассемблера. В этот список можно добавить дополнительные опции, например определение границ функций, распознавание таблиц переходов, выделение локальных переменных, что, в свою очередь, значительно усложнит его работу. Качество результирующих листингов дизассемблирования определяется свойствами применяемых алгоритмов, а также уместностью их применения в конкретной ситуации.

Базовый алгоритм дизассемблирования включает следующие шаги:

- идентификация кодового сегмента. Так как команды обычно смешаны с данными, то дизассемблеру необходимо их разграничить;
- получив адрес первой команды, необходимо прочитать значение, содержащееся по этому адресу (или смещению в файле), и выполнить табличное преобразование двоичного кода операции в соответствующую ему мнемонику языка;
- как только команда была обнаружена и декодирована, ее ассемблерный эквивалент может быть добавлен к результирующему листингу, после этого необходимо выбрать одну из разновидностей синтаксиса языка;
- следует перейти к следующей команде и повторить предыдущие шаги до тех пор, пока каждая команда файла не будет дизассемблирована.

Далее рассмотрим основные алгоритмы, используемые в процессе дизассемблирования.

Алгоритм линейной развертки использует прямолинейный подход при выборе очередной команды для дизассемблирования. При завершении кода одной команды начинается код новой. Поэтому наиболее сложной задачей становится определение первой команды. Дизассемблирование начинается с первого байта в сегменте кода и последовательно продвигается, обрабатывая команды одну за другой, пока не будет достигнут конец сегмента. При этом анализ логики передачи управления в программе посредством распознавания команд перехода, таких, например, как условия, не производится. Основным преимуществом алгоритма линейной развертки является полное покрытие кода в сегменте. Один из основных недостатков — невозможность распознать совмещенные с кодом данные.

Алгоритм рекурсивного спуска применяет концепцию передачи управления, которая определяет необходимость в дизассемблировании команды исходя из определения наличия или отсутствия ссылок на нее от других команд.

Классифиция команд по влиянию на счетчик команд центрального процессора упрощает понимание алгоритма. Рассмотрим основные из них.

Команды, не влияющие на счетчик команд. После выполнения такой команды управление переходит непосредственно к следующей команде, например, арифметические команды, такие как `add`. Для подобных команд процесс дизассемблирования такой же, как при использовании алгоритма линейной развертки.

Команды условного перехода. Команды условного перехода, например `x86 jnz`, образуют две возможные ветви исполнения. Поскольку при статическом анализе часто невозможно определить исход проверки условия, при использовании алгоритма рекурсивного спуска дизассемблируются обе ветви. Адрес целевой ветви добавляется в список адресов для последующего дизассемблирования. Дизассемблирование продолжается последовательно, так, как если бы условие было ложно.

Команды безусловного перехода. Безусловные переходы приводят к нарушению последовательного порядка исполнения команд. После выполнения безусловного перехода расположение команды, получившей управление, может оказаться на значительном расстоянии. Кроме того, команды, непосредственно следующие за командой безусловного перехода, не исполняются вообще. Поэтому необходимость в их дизассемблировании отпадает. Алгоритм рекурсивного спуска пытается определить адрес назначения безусловного перехода и занести его в список адресов для последующего дизассемблирования. Не все безусловные переходы могут быть корректно обработаны данным алгоритмом. В случае когда адрес назначения перехода зависит от параметра, получаемого в процессе исполнения, его определение методами статического анализа становится невозможным.

Команды вызова функции. Команды вызова функции работают сходным образом с командами безусловных переходов (включая невозможность определить адрес назначения команды, такой как `call eax`), за исключением того, что после выполнения функции управление обычно возвращается команде, следующей за ее вызовом. Как и в случае с условными переходами, образуются две ветви исполнения. Адрес назначения команды `call` добавляется в список адресов для последующего дизассемблирования, в то время как команда, следующая за `call`, дизассемблируется с использованием алгоритма линейной развертки.

Алгоритм рекурсивного спуска может оказаться неэффективным, если при возвращении из вызываемой функции поведение программы отклоняется от ожидаемого. Например, в коде функции может преднамеренно модифицироваться адрес возврата.

Команды возврата. Команда возврата из функции (например, `ret x86`) не предоставляет информацию о том, какая команда будет выполнена далее, в этих случаях алгоритм рекурсивного спуска терпит неудачу. Если бы программа была на самом деле запущена, управление было бы передано по адресу, расположенному на вершине стека. У дизассемблера нет возможности доступа

к стеку. Вместо этого дизассемблирование внезапно останавливается. В этом случае алгоритм рекурсивного спуска обращается к списку отложенных адресов и процесс дизассемблирования возобновляется. Этот рекурсивный процесс отражает смысл названия алгоритма.

Алгоритм рекурсивного спуска разделяет код и данные. Главным недостатком такого подхода является неспособность распознавать ветви, образуемые таким командами, как `jump` и `call`, использующими для адресации таблицы поиска. Несмотря на это, в сочетании с эвристиками для распознавания указателей на код алгоритм рекурсивного спуска способен обеспечить хорошее покрытие кода в сочетании с прекрасным разделением кода и данных.

Некоторые средства дизассемблирования и отладки предлагают достаточно широкие возможности для анализа, но они, как и любой набор инструментов, несовершенны. Это проявляется при анализе вредоносного ПО (вирусов, червей, троянских коней) или специально разработанных программ. Обычно их разработчики делают все возможное для затруднения их анализа и снижения эффекта от применения отладчиков, дизассемблеров и других подобных инструментальных средств. Например, вирус RST для Linux в случае обнаружения, что он работает под управлением отладчика, завершает свою работу. Этот же вирус при инфицировании исполняемых и компонуемых файлов ELF (Executable and Linkable Format — формат исполняемых и компонуемых модулей) модифицирует их заголовки таким образом, что некоторые дизассемблеры не могут дизассемблировать двоичный код вируса (обычно это достигается путем размещения кода вируса в необъявленном программном сегменте). Часто злоумышленный код шифруется или сжимается, что защищает его от экспертизы. Черви Code Red распространялись половинками своих программ, маскируясь под строки переполнения, и, таким образом, формат их двоичных данных не подходил ни под один из стандартных заголовков файла. Поэтому при анализе двоичных данных нужно не только знать о том, какие инструментальные средства и как применять, но и как действовать без них. При выполнении анализа заголовка файла возможно определить, какие данные, как и с какой целью модифицированы. Вероятно, потребуется проанализировать зашифрованные данные и процедуру их расшифровки для восстановления алгоритма работы программы и выяснения результатов ее работы.

Конечно, для выполнения анализа необходимо знать ассемблер в объеме, достаточном не только для чтения ассемблерных про-

грамм, но и для написания программ расшифровки или восстановления данных. Писать на ассемблере намного сложнее, чем читать программы, написанные на этом языке. Неплохим приемом для обучения будет анализ собственных программ с использованием отладчиков и дизассемблеров.

3.8. ЗАЩИТА ПРОГРАММ ОТ ИССЛЕДОВАНИЯ

Вопросы защиты ПО от исследования возникают в основном в двух случаях: при желании защитить разработанный программный продукт как объект интеллектуальной собственности или при разработке вредоносного ПО с целью усложнить его анализ и скрыть механизмы его воздействия на ИС в целом.

Анализ логики работы программы может выполняться в одном из двух режимов: статическом и динамическом.

Сущность статического режима заключается в изучении исходного текста программы без запуска ее на выполнение. Для получения листингов исходного текста, как было описано ранее, выполняемый программный модуль подвергается дизассемблированию.

Динамический режим изучения алгоритма программы предполагает выполнение трассировки программы, под которой понимается выполнение программы на ЭВМ с использованием специальных средств, позволяющих выполнять программу в пошаговом режиме, получать доступ к регистрам, областям памяти, производить остановку программы по определенным адресам и т.д. В динамическом режиме изучение алгоритма работы программы осуществляется либо в процессе трассировки, либо по данным трассировки, которые записаны в запоминающем устройстве.

Средства противодействия дизассемблированию не могут защитить программу от трассировки, и наоборот: программы, защищенные только от трассировки, могут быть дизассемблированы. Поэтому для защиты программ от изучения необходимо иметь средства противодействия как дизассемблированию, так и трассировке.

Для **противодействия дизассемблированию** обычно применяют комбинацию из следующих методов:

- архивация;
- шифрование;
- использование самогенерирующихся кодов;
- «обман» дизассемблера.

Метод архивации можно рассматривать как простейшее шифрование. Как уже упоминалось, архивация может быть объединена с шифрованием. Комбинация таких методов позволяет получать надежно закрытые компактные программы. Зашифрованную с использованием современных криптографических методов программу невозможно дизассемблировать без дешифрования.

Шифрование (дешифрование) программ может осуществляться аппаратными средствами или отдельными программами. Такое шифрование используется перед передачей программы по каналам связи или при хранении ее на внешних запоминающих устройствах. Дизассемблирование программ в этом случае возможно только при получении доступа к дешифрованной программе, находящейся в оперативной памяти перед ее выполнением. Другой подход к защите от дизассемблирования связан с совмещением процесса дешифровки с процессом выполнения программ. Если дешифрование всей программы осуществляется блоком, получающим управление первым, то такую программу расшифровать довольно просто. Гораздо сложнее дешифровать и дизассемблировать программу, которая поэтапно расшифровывает информацию, и этапы разнесены по ходу выполнения программы. Задача становится еще более сложной, если процесс дешифровки разнесен по тексту программы.

Метод самогенерирующихся кодов основан на генерации исполняемых кодов программы самой программой в процессе ее выполнения. Самогенерируемые коды получаются в результате ряда определенных действий над специально выбранным массивом данных. В качестве исходных данных могут использоваться исполняемые коды самой программы или специально подготовленный массив данных. Указанный метод показал свою высокую эффективность, но он сложен в реализации.

Под **«обманом» дизассемблера** понимают применение техник программирования, которые вызывают нарушение правильной работы стандартного дизассемблера за счет нестандартных приемов использования отдельных команд, за счет нарушения общепринятых соглашений. «Обман» дизассемблера осуществляется несколькими способами:

- нестандартная структура программы;
- скрытые переходы, вызовы процедур, возвраты из них и из прерываний;
- переходы и вызовы подпрограмм по динамически изменяемым адресам;
- модификация исполняемых кодов.

Ранее говорилось о сложности анализа дизассемблером команд скрытых переходов, вызовов и возвратов. Естественно, осуществление указанных действий выполняется с применением нестандартных возможностей команд. Маскировка скрытых действий часто осуществляется с применением стеков.

Трассировка программ обычно проводится с помощью программных продуктов, называемых отладчиками.

При анализе алгоритмов программ используются такие возможности отладчиков, как пошаговое (покомандное) выполнение программ, возможность останова в контрольной точке. Покомандное выполнение осуществляется процессором при установке пошагового режима работы.

Контрольной точкой называют любое место в программе, на котором обычное выполнение программы приостанавливается, и осуществляется переход в особый режим, например в режим покомандного выполнения. Для реализации механизма контрольной точки обычно используется прерывание по соответствующей команде ЭВМ.

В современных процессорах существует возможность использовать специальные регистры для установки нескольких контрольных точек при выполнении определенных операций: обращение к участку памяти, изменение участка памяти, выборка по определенному адресу, обращение к определенному порту ввода-вывода. Современные средства отладки программ не могут полностью исключить возможность изучения алгоритма программы, но существенно затруднить процесс трассировки возможно. Основной задачей противодействия трассировке является увеличение числа и сложности ручных операций, которые необходимо выполнить программисту-аналитику.

Для **противодействия трассировке программы** в ее состав вводятся следующие механизмы:

- изменение среды функционирования;
- модификация кодов программы;
- «случайные» переходы.

Под **изменением среды функционирования** понимается запрет или переопределение прерываний, изменение режимов работы, состояния управляющих регистров, триггеров и т. д. Такие изменения заставляют отслеживать изменения и вручную восстанавливать среду функционирования.

Изменяющиеся коды программ могут приводить к тому, что каждое выполнение процедуры, к примеру, осуществляется по различным ветвям алгоритма.

«Случайные» переходы выполняются за счет вычисления адресов переходов. Исходными данными для этого служат характеристики среды функционирования, контрольные суммы модифицируемых процедур. Включение таких механизмов в текст программ существенно усложняет изучение алгоритмов программ путем их трассировки.

Для защиты программ от исследования необходимо применять методы защиты от исследования как самого файла с его исполняемым кодом, хранящимся на внешнем носителе, так и методы защиты исполняемого кода, загружаемого в оперативную память для выполнения этой программы. В первом случае защита может быть основана на шифровании секретной части программы, а во втором — на блокировании доступа отладчиков к исполняемому коду программы, находящемуся в оперативной памяти.

Кроме того, необходимо предусмотреть обнуление всего кода программы в оперативной памяти перед завершением работы. Это предотвратит возможность несанкционированного копирования из оперативной памяти дешифрованного исполняемого кода после выполнения.

Исходя из сказанного, **структура защищаемой от исследования программы** должна включать следующие модули:

- инициализатор;
- зашифрованная секретная часть;
- деструктор (деинициализатор).

Инициализатор обеспечивает выполнение следующих функций:

- сохранение параметров операционной среды функционирования (векторов прерываний, содержимого регистров процессора);
- запрет всех внутренних и внешних прерываний, обработка которых не может быть запротоколирована в защищаемой программе;
- загрузка в оперативную память и дешифрование кода секретной части программы;
- передача управления секретной части программы.

Секретная часть программы, защищаемая шифрованием для предупреждения внесения в нее программной закладки, предназначена для выполнения основных целевых функций программы.

Деструктор принимает управление после выполнения секретной части программы для осуществления следующих действий:

- обнуление секретного кода программы в оперативной памяти;
- восстановление параметров операционной системы, которые были установлены до запрета неконтролируемых прерываний;

- выполнение операций, которые невозможно выполнить при запрете неконтролируемых прерываний;
- освобождение всех незадействованных ресурсов компьютера и завершение работы программы.

Обобщим порядок применения методов защиты программ от исследования. Для большей надежности применяется частичное шифрование инициализатора: он может дешифровать сам себя по мере выполнения программы, также по мере выполнения может дешифроваться и секретная часть программы. Такое дешифрование называется динамическим дешифрованием исполняемого кода — очередные участки программ перед непосредственным исполнением дешифруются, а после исполнения сразу уничтожаются.

Для повышения эффективности защиты программ от исследования с использованием отладчиков необходимо внесение в них **дополнительных функций безопасности**, таких как:

- периодический подсчет контрольной суммы области оперативной памяти, занимаемой защищаемым исходным кодом;
- сравнение текущей контрольной суммы с предварительно сформированной эталонной суммой и принятие необходимых мер в случае несовпадения;
- проверка занимаемой защищаемой программой оперативной памяти — сравнение с объемом, к которому программа адаптирована, и принятие необходимых мер в случае несоответствия;
- контроль времени выполнения отдельных частей программы;
- блокировка клавиатуры на время отработки особо секретных алгоритмов.

Также следует упомянуть **метод обфускации** (запутывания) для защиты программ от исследования с помощью дизассемблеров. Суть этого метода заключается в усложнении структуры самой программы с целью запутывания злоумышленника, который дизассемблирует эту программу. Например, возможно использование разных сегментов адреса для обращения к одной и той же области памяти, что приводит к усложнению определения факта работы программы с одной и той же областью памяти.

3.9. ИССЛЕДОВАНИЕ КОДА ВРЕДОНОСНЫХ ПРОГРАММ

Вредоносное ПО, или **вредоносные программы (ВП)**, являются основными виновниками большинства компьютерных вторжений и инцидентов в области компьютерной безопасности. Любое ПО,

которое делает что-то, что причиняет вред пользователю, компьютеру или сети, может считаться вредоносным, это могут быть вирусы, троянские кони, черви, руткиты, пугалки и шпионские программы.

Для нейтрализации влияния ВП у аналитиков ВП есть в распоряжении ряд инструментов и методов для их анализа.

Анализ вредоносных программ — деятельность по разбору вредоносного ПО с целью определения принципов его работы, поиска методов его определения и способов противодействия воздействию ВП на функционирование ИС.

Прежде чем рассматривать особенности анализа ВП, необходимо определиться с терминологией, чтобы охватить распространенные типы ВП и найти основные подходы к его исследованию. В то время как ВП фактически ежедневно появляются в различных формах и с различными принципами воздействия на ИС, для их анализа зачастую используются общие методы. Выбор метода анализа будет зависеть от поставленных целей.

Согласно ГОСТ Р 51188—98 «Защита информации. Испытания программных средств на наличие компьютерных вирусов. Типовое руководство» ***компьютерный вирус*** — программа, способная создавать свои копии (необязательно совпадающие с оригиналом) и внедрять их в файлы, системные области компьютера, компьютерных сетей, а также осуществлять иные деструктивные действия. При этом копии сохраняют способность дальнейшего распространения.

Компьютерный вирус относится к ВП.

Естественно, воздействие ВП подразумевает несанкционированный доступ к информации в ИС.

Несанкционированный доступ к программным средствам — доступ к программам, записанным в памяти ЭВМ или на машинном носителе, а также отраженным в документации на эти программы, осуществленный с нарушением установленных правил.

Защита программных средств — организационные, правовые, технические и технологические меры, направленные на предотвращение возможных несанкционированных действий по отношению к программным средствам и устранение последствий этих действий.

Профилактика — систематические действия эксплуатационного персонала, цель которых — выявить и устранить неблагоприятные изменения в свойствах и характеристиках используемых программных средств, в частности проверить эксплуатируемые,

хранимые и (или) вновь полученные программные средства на наличие компьютерных вирусов.

Вакцинирование — обработка файлов, дисков, каталогов, проводимая с применением специальных программ, создающих условия, подобные тем, которые создаются определенным компьютерным вирусом, и затрудняющих повторное его появление.

Следует отметить, что деятельность по созданию и распространению компьютерных вирусов и ВП уголовно наказуема. В Уголовном кодексе Российской Федерации содержатся следующие правовые нормы:

- ст. 272 «Неправомерный доступ к компьютерной информации»;
- ст. 273 «Создание, использование и распространение вредоносных компьютерных программ»;
- ст. 274 «Нарушение правил эксплуатации средств хранения, обработки или передачи компьютерной информации и информационно-телекоммуникационных сетей»;
- ст. 274.1 «Неправомерное воздействие на критическую информационную инфраструктуру Российской Федерации».

Данные статьи обязательны к изучению каждым, кто сталкивается с вопросами информационной безопасности, потому что некоторые действия пользователей, даже производимые без злого умысла, могут привести к достаточно серьезным процессуальным последствиям.

Перейдем к **классификации самого вредоносного ПО**. Здесь необходимо четкое понимание, что ежедневно в мире появляется около двух десятков ВП, именно поэтому антивирусные программы не являются панацеей, обеспечивающей 100%-ную защиту от их воздействия.

Вопрос классификации вредоносного ВП также сложен и неоднозначен, потому что злоумышленники, создавая ВП, для достижения своих целей используют различные механизмы воздействия, включающие признаки различных видов. Мы рассмотрим основные из них:

- `backdoor` — вредоносный код, который устанавливается на компьютер, чтобы предоставить злоумышленнику доступ к системе. `Backdoor` обычно позволяют злоумышленнику подключаться к компьютеру практически без проверки подлинности и выполнять команды в локальной системе;
- `botnet` — похож на `backdoor` в том, что позволяет злоумышленнику получить доступ к системе, но все компьютеры, зараженные тем же `botnet`, получают те же инструкции от одного сервера команд и управления;

- `downloader` — загрузчик вредоносного кода, который существует только для загрузки другого вредоносного кода. Загрузчики обычно устанавливаются злоумышленниками при первом получении доступа к системе. Программа-загрузчик загрузит и установит дополнительный вредоносный код;
- `information-stealing malware` — ВП для кражи информации, которая собирает информацию с компьютера жертвы и обычно отправляет ее злоумышленнику, это, например, могут быть снифферы, хеш-грабберы паролей и кейлоггеры. Эта ВП обычно используется для получения доступа к онлайн-аккаунтам, таким как электронная почта или интернет-банкинг;
- `launcher` — ВП запуска, используются для запуска других ВП. Как правило, `launcher` используют нетрадиционные методы для запуска других ВП, чтобы обеспечить скрытность или больший доступ к системе;
- `rootkit` — вредоносный код, предназначенный для сокрытия существования другого кода. `Rootkit` обычно сопряжен с другими ВП, такими как `backdoor`, чтобы разрешить удаленный доступ злоумышленнику и затруднить обнаружение кода для жертвы;
- `scareware` (антивирусы) — ВП, предназначенные для запугивания зараженных пользователей, заставляющих что-то покупать. Обычно он имеет пользовательский интерфейс, который делает его похожим на антивирус или другую программу безопасности. Он информирует пользователей о том, что в их системе есть вредоносный код и единственный способ избавиться от него — купить их «ПО», когда на самом деле «ПО», которое он продает, не делает ничего, кроме удаления `scareware`;
- `spam-sending malware` — спам-отправка ВП, заражающая компьютер пользователя, а затем использующая его для отправки спама. Это вредоносное ПО приносит доход злоумышленникам, позволяя им продавать услуги рассылки спама;
- `worm` (вирус) — вредоносный код, который может копировать себя и заражать другие компьютеры.

Цели, которые необходимо достигнуть в результате анализа ВП, частично определяют действия по определению масштабов вреда, наносимого таким ПО.

Основной целью анализа ВП обычно является предоставление информации, необходимой для ответа на характер вторжения в сеть. Она, как правило, состоит в том, чтобы точно определить, что произошло, и убедиться, что вы обнаружили все зараженные машины и файлы. При анализе подозрительного вредоносного ПО целью обычно является точное определение, что может сде-

лать конкретный подозрительный двоичный файл, как обнаружить его в вашей сети и как ему противодействовать.

После определения файлов, требующих полного анализа, нужно разработать инструкции, которые будут использоваться системой предотвращения вторжений (IPS) сети. Анализ ВП может использоваться для разработки правил на основе хоста и сети. Индикаторы на основе узлов используются для обнаружения вредоносного кода на компьютерах-жертвах. Эти индикаторы часто идентифицируют файлы, созданные или измененные ВП или конкретными изменениями, которые она вносит в реестр. В отличие от сигнатур антивируса индикаторы вредоносного ПО фокусируются на том, что вредоносное ПО делает с системой, а не на характеристиках самого вредоносного ПО, что делает их более эффективными в обнаружении ВП, которые изменяют форму или были удалены с жесткого диска.

Сетевые правила для обнаружения вредоносного кода путем мониторинга сетевого трафика могут быть созданы без анализа ВП, но созданные с помощью анализа, как правило, гораздо более эффективны, предлагают более высокую скорость обнаружения и меньше ложных срабатываний.

После получения правил для IPS конечная цель — выяснить, как именно работает ВП. Это наиболее часто задаваемый вопрос руководством, которое хочет получить сведения о вторжении в сеть.

Углубленные приемы анализа позволяют определить назначение и возможности ВП.

Существуют два основных **подхода к анализу ВП**: статический и динамический. Статический анализ включает в себя изучение вредоносного ПО без его запуска, динамический подразумевает запуск вредоносного ПО. Оба метода далее классифицируются как основные, или «продвинутые».

Базовый статический анализ состоит в изучении исполняемого файла без просмотра фактических инструкций. Он может подтвердить, является ли файл вредоносным, предоставить данные о его функциональности, а иногда информацию, которая позволит создавать простые сетевые правила. Базовый статический анализ прост и не занимает много времени, но он неэффективен против сложных ВП и может пропустить важные свойства вредоносного ПО.

Базовый динамический анализ включает запуск вредоносного ПО и наблюдение за его поведением в системе для удаления вредоносного ПО, создания эффективных сигнатур или решения обеих этих задач. Тем не менее чтобы запустить вредоносное ПО

безопасно, необходимо настроить среду, которая позволит изучить его в динамике.

Расширенный статический анализ состоит из реверс-инжиниринга внутреннего устройства вредоносного ПО: загрузив исполняемый файл в дизассемблер и глядя на инструкции программы, можно узнать, что делает программа. Инструкции выполняются процессором, поэтому расширенный статический анализ точно скажет, что делает программа.

Расширенный динамический анализ использует отладчик для проверки внутреннего состояния выполняющегося вредоносного файла. Усовершенствованные методы динамического анализа обеспечивают еще один способ извлечения подробной информации из исполняемого файла.

Эти методы наиболее полезны, когда вы пытаетесь получить информацию, которую трудно собрать с помощью других методов.

Для извлечения полезной информации об исполняемых файлах рассмотрим следующие методы:

- использование антивирусного ПО для подтверждения возможного заражения;
- использование хешей для идентификации ВП;
- извлечение информации из строк, функций и заголовков файла.

В зависимости от решаемых задач каждый метод может предоставлять различную информацию, но, как правило, используют несколько методов сразу, чтобы собрать как можно больше информации.

Антивирусное сканирование является первым шагом при начале анализа потенциальных ВП. Неплохо запустить его через несколько антивирусных программ, которые, возможно, уже идентифицировали его. Но антивирусные инструменты, конечно, неидеальны. Они полагаются главным образом на базу данных идентифицируемых фрагментов подозрительного кода (сигнатур файлов), а также на поведенческий анализ и анализ соответствия шаблону (эвристика) для идентификации подозрительных файлов. Одна из проблем заключается в том, что разработчики ВП могут легко изменять свой код, тем самым изменяя сигнатуру своей программы и уклоняясь от антивирусных сканеров. Кроме того, редкие ВП часто остаются незамеченными антивирусным ПО, потому что их просто нет в базе данных. Наконец, эвристика, хотя и часто успешна в идентификации неизвестного вредоносного кода, может быть обойдена новыми и уникальными ВП.

Поскольку различные антивирусные программы используют разные сигнатуры и эвристики, полезно запускать несколько раз-

личных антивирусных программ против одной и той же части подозрительного вредоносного ПО. Веб-сайты, такие как VirusTotal (http://www.virustotal.com/), позволяют загружать файл для сканирования несколькими антивирусными ядрами. Сервис генерирует отчет, содержащий общее количество инструментов, которые пометили файл как вредоносный, указывает, если имеется, имя ВП и дополнительную информацию о нем.

Следующий рассматриваемый метод анализа — **хеширование**. Это распространенный метод, используемый для уникальной идентификации ВП. Вредоносное ПО пропускается через программу хеширования, которая вычисляет уникальный хеш-код, идентифицирующий эту ВП (как отпечаток пальца). Применяя популярные в своем роде алгоритмы MD5 и SHA-1, посредством вычисления хеш-функции получают сообщение-дайджест. Например, использование свободно доступной программы `md5deep` для вычисления хеша программы `Solitaire`, которая поставляется с Windows, будет генерировать выходные данные — сообщение дайджест:

```
373e7a863a1a345c60edb9e20ec3231
```

На рис. 3.19 изображено окно калькулятора WinMD5Free на основе графического интерфейса пользователя, который обладает интуитивно понятным интерфейсом и может вычислять и отображать хеши для нескольких файлов одновременно. Получив уникальный хеш для анализируемой программы, его можно использовать следующим образом:

- использовать в качестве метки;
- поделиться хешем с другими аналитиками, чтобы помочь им определить ВП;
- найти по хешу в Интернете, чтобы увидеть, если файл уже был идентифицирован.

Следующий метод — **поиск строк**. Строка в программе — это последовательность символов. Программа содержит строки, если она выдает сообщения, подключается к URL-адресу или копирует файл в определенное месторасположение.

Поиск по строкам может быть использован как простой способ получения подсказок о функциональности программы. Например, если программа обращается к URL-адресу, то вы увидите URL-адрес, доступ к которому хранится в виде строки в программе. Вы можете использовать программу `Strings`, расположенную по адресу: https://docs.microsoft.com/ru-ru/sysinternals/downloads/strings, чтобы найти строки в исполняемых файлах, которые обычно хранятся в формате ASCII или формате Unicode.

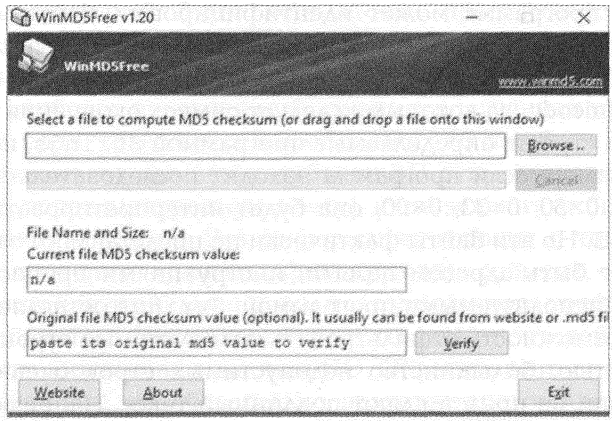


Рис. 3.19. Общий вид окна приложения WinMD5Free для вычисления хеш-кода анализируемых файлов

Стоит заметить, что Microsoft использует термин «строка расширенных символов» для описания собственной реализации строк в формате Unicode, который немного отличается от общих стандартов Unicode. (В этой книге, ссылаясь на Unicode, мы имеем в виду реализацию Microsoft.)

Оба формата — ASCII и Unicode — хранят символы в последовательностях, которые заканчиваются символом конца строки Null, чтобы указать, что строка завершена. Строки ASCII используют 1 байт, а Unicode — 2 байта на символ.

На рис. 3.20 показана строка BAD, представленная в форматах ASCII и Unicode. В ASCII строка хранится как байт 0×42, 0×41, 0×44 и 0×00, где 0×42 является представлением ASCII буквы Б, 0×41 — буквы А и т.д., 0×00 в конце является нулевым терминатором.

При поиске строк, представленных форматами ASCII и Unicode, в исполняемых файлах программа игнорирует контекст и форматирование, так что может анализироваться любой тип файла и по всему файлу могут быть обнаружены строки (хотя это также озна-

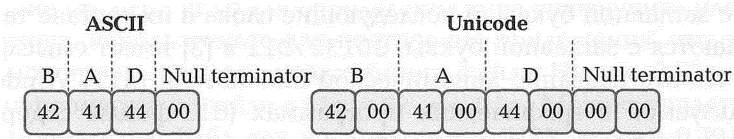


Рис. 3.20. Строка BAD, представленная в форматах ASCII и Unicode

чает, что программа может идентифицировать байты символов как строки, когда они ими не являются). `Strings` выполняет поиск трехбуквенной или большей последовательности символов ASCII и Unicode, за которыми следует символ окончания строки.

Иногда строки, определяемые программой `Strings`, неактуальны. Например, если программа находит последовательность байтов 0×56, 0×50, 0×33, 0×00, она будет интерпретировать это как строку `Vp3`. Но эти байты фактически не представляют эту строку, они могут быть адресом памяти, инструкциями процессора или данными, используемыми программой. `Strings` оставляет пользователю возможность отфильтровать недопустимые строки.

К счастью, большинство недопустимых строк очевидны, поскольку они не представляют подлинный текст. Например, в следующем фрагменте показан результат выполнения утилиты `Strings`:

```
C:>strings bp6.ex_
VP3
VW3
t$@
D$4
99.124.22.1 [4]
e-@
GetLayout [1]
GDI32.DLL [3]
SetLayout [2]
M}C
Mail system DLL is invalid.!Send Mail failed to
send message.[5]
```

В этом примере некоторые строки можно игнорировать. Как правило, если строка короткая и не соответствует осмысленным словам, она, вероятнее всего, бессмысленна.

С другой стороны, строки [1] `GetLayout` и [2] `SetLayout` являются функциями Windows, использующими графические библиотеки Windows. Мы можем легко идентифицировать их как значимые строки, потому что имена функций Windows обычно начинаются с заглавной буквы, и последующие слова в их составе также начинаются с заглавной буквы. `GDI32.Dll` в [3] имеет смысл, потому что это название динамической библиотеки (`dll`) Windows, используемой в графических программах (`dll`-файлы содержат исполняемый код, который совместно используется несколькими приложениями). Число `99.124.22.1`, вероятнее всего, является

IP-адресом, который будет использован программой. Наконец в строке [5] указывается, что не удалось отправить сообщение об ошибке из-за неверной системной `dll`.

Часто наиболее полезная информация, полученная при выполнении `Strings`, содержится в сообщениях об ошибках. Это сообщение показывает две вещи: субъект ВП отправляет сообщения (вероятно, через электронную почту), и это зависит от библиотеки `dll` почтовой системы. Эта информация говорит о том, что мы могли бы проверить журналы, фиксирующие события, связанные с электронной почтой, для оценки подозрительного трафика, и что другая библиотека (системная библиотека `dll` почтовой системы) может быть соотнесена с данным ПО. Обратите внимание, что отсутствующая `dll` не обязательно является вредоносной. Вредоносное ПО часто использует имеющиеся системные библиотеки для достижения своих целей.

Разработчики ВП обычно используют упаковку или запутывание, чтобы затруднить обнаружение и анализ их файлов. Упакованные программы представляют собой подмножество запутанных программ, в которых ВП сжимается и не может быть проанализирована. Оба метода серьезно ограничивают статический анализ ВП.

Вредоносное ПО, которое упаковано или запутано, содержит очень мало строк. Если при анализе программы обнаружится, что она имеет только несколько строк, вероятнее всего, вы столкнулись именно с обфускацией (запутыванием) или упакованным ПО, конечно, если есть подозрение, что данная программа является вредоносной. Скорее всего, не удастся обойтись только методами статического анализа, чтобы продолжить исследование дальше.

При запуске упакованной программы также запускается небольшая программа-оболочка для распаковки упакованного файла (все строки и другая информация упакованного файла сжаты и не видны для большинства инструментов статического анализа), а затем выполняется распакованный файл, как показано на рис. 3.21. При статическом анализе упакованной программы исследуется только небольшая программа-оболочка.

Один из способов обнаружить упакованные файлы — использование утилиты `PEiD` для определения типа упаковщика или компилятора, используемого для построения приложения, что значительно упрощает анализ упакованного файла. На рис. 3.22 показана информация о файле `orig_af2.ex_file`. Можно увидеть, что PEiD определил файл как упакованный с UPX версии 0.89.6-1.02 или 1.05-2.90. Когда программа упакована, вы должны распаковать

Рис. 3.21. Исходный исполняемый файл (слева) и упакованный исполняемый файл (справа)

ее для выполнения любого анализа. Процесс распаковки часто сложен, но программа упаковки UPX настолько популярна и проста в использовании, что для распаковки необходимо просто скачать UPX (http://upx.sourceforge.net/) и запустить ее, используя исследуемый файл в качестве входных данных.

Следует заметить, что программа `PEiD` может без предупреждения запустить исследуемый файл, поэтому необходимо проводить анализ, настроив безопасную среду (окружение, напоминающее так называемую песочницу в антивирусных программах).

До сих пор мы обсуждали инструменты, которые сканируют исполняемые файлы независимо от их формата. Однако формат файла может многое рассказать о функциональности программы.

Формат портативных исполняемых файлов (Portable Executable, PE) используется исполняемыми файлами Windows, объектным кодом и библиотеками `dll`. Формат файла PE — это структура данных, которая содержит информацию, необходимую для загрузчика ОС Windows для управления обернутым исполняемым кодом. Почти каждый файл с исполняемым кодом, который за-

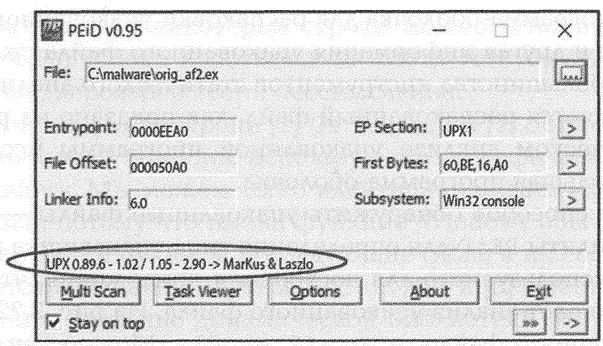


Рис. 3.22. Окно программы `PEiD`, отображающее информацию о программе-упаковщике

гружен Windows, находится в формате PE-файла, хотя некоторые устаревшие форматы файлов появляются в редких случаях в ВП.

PE-файлы начинаются с заголовка, который содержит информацию о коде, типе приложения, требуемых функциях библиотеки и требованиях к пространству. Информация в заголовке PE имеет большое значение для аналитиков ВП.

Статическая компоновка является наименее часто используемым методом компоновки библиотек, хотя она распространена в программах UNIX и Linux. Когда библиотека статически связана с исполняемым файлом, весь код из этой библиотеки копируется в исполняемый файл, что приводит к увеличению его размера. При анализе кода трудно провести различие между статически связанным кодом и собственным кодом исполняемого файла, поскольку ничто в заголовке PE-файла не указывает на то, что файл содержит связанный код.

Непопулярное в реализации обычных программ динамическое связывание широко используется в ВП, особенно когда он упакован или обфусцирован. Такие исполняемые файлы используют подключение библиотек, только когда необходима какая-либо функция, а не при старте программы.

Несколько функций Microsoft Windows предоставляют программистам возможность импортировать связанные функции, не перечисленные в заголовке файла программы. Из них наиболее часто используются `LoadLibrary` и `GetProcAddress`, а также `LdrGetProcAddress` и `LdrLoadDll`. `LoadLibrary` и `GetProcAddress` позволяют программе получить доступ к любой функции в любой библиотеке в системе. Это означает, что при использовании этих функций вы не можете сказать статически, какие функции связаны с подозрительной программой.

Из всех методов динамическое связывание является наиболее распространенным и интересным для аналитиков ВП. Когда библиотеки динамически связаны, операционная система узла ищет необходимые библиотеки при загрузке программы. Когда программа вызывает функцию связанной библиотеки, эта функция выполняется в библиотеке.

В заголовке PE-файла хранится информация о каждой загружаемой библиотеке и каждой функции, которая будет использоваться программой. Используемые библиотеки и вызываемые функции часто являются наиболее значимыми частями программы, и их определение особенно важно, потому что позволяет определить, что делает программа. Например, если программа импортирует функцию `URLDownloadToFile`, можно предположить, что она

подключается к Интернету для загрузки содержимого, которое затем хранится в локальном файле.

Исследовать динамически подключаемые функции можно с помощью утилиты `Dependency Walker` — иллюстратора функциональных зависимостей. Программа `Dependency Walker` (http://www.dependencywalker.com/), распространяемая с некоторыми версиями Microsoft Visual Studio и других пакетов разработки Microsoft, перечисляет только динамически связанные функции в исполняемом файле.

Используя набор относительно простых инструментов, можно выполнить статический анализ ВП, чтобы получить определенное представление о его функции. Но статический анализ, как правило, только первый шаг, и обычно необходим дальнейший анализ, включающий методы реверс-инжиниринга и применение более сложных средств, обзор которых выходит за рамки этой книги.

Современный инструментарий разработки сокращает время конструирования, позволяет повысить производительность более чем на 50 %, также уменьшить объем однообразной, монотонной работы. Хорошие инструменты — лучшие друзья программиста. По статистике 20 % инструментария используется приблизительно в 80 % случаев, и отсутствие знаний о каком-то наиболее полезном инструменте осложняет разработчику жизнь.

Современные среды разработки предоставляют достаточно обширный набор инструментария, при этом с развитием возможностей сети появляется довольно много средств для верификации и валидации программного кода, особенно при разработке веб-приложений. В рамках данного раздела мы рассмотрим возможности средств измерения характеристик качества и безопасности программного кода.

Анализаторы качества кода относятся к категории, исследующей статический исходный код с целью оценки его качества. Перечислим их.

Программы углубленного контроля синтаксиса и семантики, предназначенные для осуществления более тщательной проверки кода, чем это обычно делает компилятор. Компилятор может проверять наличие только элементарных синтаксических ошибок. При углубленном контроле могут учитываться нюансы языка, что позволяет проверить наличие более коварных ошибок — тех, что не выглядят таковыми с точки зрения компилятора. Например, в C++ выражение

```
while ( i = 0 ) ...
```

абсолютно законно, но обычно имеется в виду

```
while ( i == 0 ) ...
```

Первая строка синтаксически корректна, но путаница со знаками «=» и «= =» является распространенной ошибкой, и данная строка, возможно, неправильна.

`Lint` — углубленный анализатор синтаксиса и семантики, используемый во многих средах C/C++, предупреждает о наличии неинициализированных переменных (переменных, которым присвоено значение), которые никогда не используются, выходных параметрах метода, которым не было присвоено значение внутри метода, подозрительных операциях с указателями и логических сравнениях, недостижимом коде и прочих распространенных проблемах. Другие языки предлагают похожие инструменты.

Генераторы отчетов о метриках, которые составляют отчет о качестве кода. Например, средства, сообщающие о сложности каждого метода, позволяют направить наиболее сложные функции на дополнительное рецензирование, тестирование или перепроектирование. Некоторые средства подсчитывают количество строк кода, объявлений данных, комментариев и пустых строк как для всей программы, так и для отдельных методов. Они отслеживают дефекты, внесенные конкретными программистами, затем фиксируют изменения, сделанные для исправления дефектов, и программистов, внесших эти правки. Они подсчитывают количество модификаций ПО и выделяют процедуры, которые исправляются чаще всего. Установлено, что средства анализа сложности положительно влияют на производительность сопровождения, увеличивая ее примерно на 20 %.

Примером может послужить инструментарий, встроенный в IDE Visual Studio. Получить оценку вашего кода можно, нажав правой кнопкой на проекте и выбрав пункт `Calculate Code Metrics`, в результате появится окно, представленное на рис. 3.23.

Данный инструмент позволяет **вычислять определенные метрики программного кода**.

1. *Индекс удобства поддержки* — комплексный показатель качества кода. Этот показатель разработан специалистами из Carnegie Mellon Software Engineering Institute. Рассчитывается метрика по следующей формуле:

$$MI = MAX(0{,}171 - 5{,}2 \cdot \ln(HV) - 0{,}23 \cdot CC - 16{,}2 \cdot \ln(LoC)) \cdot 100/171,$$

где HV (Halstead Volume) — вычислительная сложность (чем больше операторов, тем больше значение этой метрики); CC (Cyclomatic

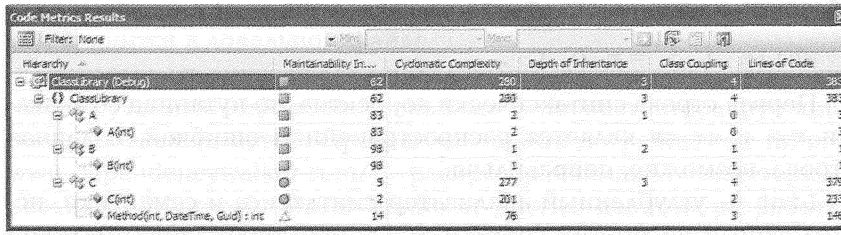


Рис. 3.23. Окно результатов расчета метрик кода программы

Complexity) показывает структурную сложность кода, т. е. количество различных ветвей в коде (чем больше этот показатель, тем больше тестов должно быть написано для полного покрытия кода); LoC — число строк кода.

Эта метрика может принимать значения от 0 до 100 и показывает относительную сложность поддержки кода. Чем больше значение этой метрики, тем легче поддерживать код. Visual Studio помечает методы/классы зеленым цветом, если значение метрики находится в пределах от 20 до 100, желтым — в пределах от 10 до 20 и красным, когда значение меньше 10.

2. *Сложность* показывает структурную сложность кода, т. е. количество различных ветвей в коде. Чем больше этот показатель, тем больше тестов должно быть написано для полного покрытия кода.

3. *Глубина наследования*. Эта метрика показывает для каждого класса, какой он по счету в цепочке наследования. Например, есть три класса *A*, *B*, *C*; если *B* унаследован от *A*, а *C* унаследован от *B*, то значение этой метрики для классов *A*, *B* и *C* будет равно соответственно 1, 2 и 3.

4. *Связанность классов* показывает степень зависимости классов друг с другом. В расчет берутся уникальные классы из параметров, локальных переменных, возвращаемого типа, базового класса, атрибутов (полный список можно найти в MSDN). Хороший дизайн ПО предполагает небольшое количество связанных классов. Чем их больше, тем сложнее в дальнейшем использовать и поддерживать этот класс, так как существует очень много зависимостей.

5. *Строки кода* указывают приблизительное количество строк кода. Этот показатель показывает неточное количество строк в вашем файле, так как подсчет основан на IL-коде. В расчет не берутся пустые строчки, комментарии, строчки со скобками, объявление

типов и пространств имен. Большое количество строк в методе/классе может показывать на ошибки в проектировании и на то, что этот код можно разделить на несколько частей.

Для проведения измерений и анализа безопасности кода приложений используется специализированный класс анализаторов.

Для всех классов анализаторов характерны общие принципы работы: как для анализаторов исходного кода веб-приложений, так и анализаторов встраиваемого кода. Отличие между этими типами продуктов — только в возможности определить особенности выполнения и взаимодействия кода с внешним миром, что отражается в базах уязвимостей анализаторов. Большая часть анализаторов, представленных на рынке, выполняет функции обоих классов, одинаково хорошо проверяя как встраиваемый в бизнес-приложения код, так и код веб-приложений.

Входными данными для анализатора исходного кода является массив исходных текстов программ и его зависимостей (подгружаемых модулей, используемого стороннего ПО и т. д.). В качестве результатов работы все анализаторы выдают отчет об обнаруженных уязвимостях и ошибках программирования, дополнительно некоторые анализаторы предоставляют функции по автоматическому исправлению ошибок.

Стоит отметить, что автоматическое исправление ошибок не всегда работает корректно, поэтому данный функционал предназначен только для разработчиков веб-приложений и встраиваемых модулей.

Заказчик продукта должен опираться только на финальный отчет анализатора и использовать полученные данные для принятия решения по приемке и внедрению разработанного кода или отправки его на доработку.

При проведении оценки исходных текстов анализаторы используют различные базы данных, содержащие описание уязвимостей и ошибок программирования.

За последние десятилетия программисты видели массу инструментов, которые предположительно должны были устранить необходимость программирования, например:

- языки третьего и четвертого поколений;
- автоматическое программирование;
- CASE-средства;
- визуальное программирование.

Каждое из этих достижений привносило значительные улучшения, и общими усилиями они сделали программирование абсолютно неузнаваемым для тех, кто изучал его до этих нововведе-

ний. Но ни одна из этих инноваций не устранила и не устранит программирования как такового.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Дайте определение качества системы.
2. Что понимают под характеристикой качества?
3. Что понимают под критерием качества?
4. Перечислите требования, которым должен соответствовать критерий качества.
5. Объясните, для чего предназначены метрики программного кода.
6. По каким направлениям можно распределить метрики программ?
7. Перечислите четыре основные группы метрик оценки сложности программ.
8. Объясните предназначение метрик размера программ.
9. На чем основана оценка метрик сложности потока управления программ?
10. Объясните предназначение метрик стилистики и понятности программы.
11. Объясните предназначение измерительных методов оценки программ.
12. Для чего применяют измерительные мониторы?
13. Дайте определение трассировочной записи.
14. Что такое динамический профиль выполнения программы?
15. Что понимают под корректностью программного средства?
16. Что такое верификация программного средства?
17. Что называют валидацией программного средства?
18. В чем суть тестирования методом «черного ящика»?
19. В чем суть тестирования методом «белого ящика»?
20. Для чего предназначены дизассемблеры?
21. Как применяют программы-отладчики при обратном проектировании?
22. Какие методы обратного проектирования вы знаете?
23. Объясните суть статического анализа при обратном проектировании.
24. В чем заключается смысл динамического анализа при обратном проектировании?
25. Перечислите известные вам виды компьютерных вирусов.
26. Назовите методы анализа вредоносного ПО.
27. Объясните смысл защиты и запутывания программных средств.
28. Какие инструментальные средства используются для оценки качества ПО?

Задание 3.1. Использование метрик программного продукта

Цель исследования. *Цель* — получить навыки в расчете базовых метрик Холстеда, относящихся к метрикам размера программ.

Пример выполнения работы. Рассмотрим выполнение данной работы на примере программы вычисления выражения $y = \sin x$ листинг которой представлен далее:

```
# include <iostream>
#include <stdio.h>
using namespase std;
main( )
{
const double cst=0.0001;
double x, y;
int n;
double bs;
cin<<x; // Инициализация x путём консольного ввода
y=x;
n=2;
bs=x;
do
{
bs= -bs*x*x/(2*n-1/2*n-2);// формирование слагаемого
n++;
y+=bs;
}
while(abs(bs)>=cst);// При выполнении условия выход из
цикла
cout>>x>>y>>cst
}
```

В табл. 3.3 представлен расчет метрик Холстеда по формулам, приведенным в подразд. 3.2.

Таблица 3.3. Расчет базовых метрик Холстеда

№	Оператор	Число вхождений	№	Операнд	Число вхождений
1	;	9	1	*x*	6
2	=	4	2	*bs*	5
3	*	4	3	*n*	4

Окончание табл. 3.3

№	Оператор	Число вхождений	№	Операнд	Число вхождений
4	,	4	4	*2*	4
5	-	3	5	*cst*	3
6	/	2	6	*y*	2
7	( )	2			
8	{ }	2			
9	*cin*	1			
10	*cout*	1			
11	*do while( )*	1			
12	*abs( )*	1			
13	>=	1			
14	++	1			
15	+=	1			
$n_1 = 15$		$N_1 = 37$	$n_2 = 6$		$N_2 = 24$

Словарь программы: $n = n_1 + n_2 = 15 + 6 = 21$.

Длина программы: $N = N_1 + N_2 = 37 + 24 = 61$.

Объем программы: $V = N\log_2 n = 61 \cdot \log_2 21 \approx 276$.

Практическая часть. Исследование выполняется в несколько этапов.

1. На основании выданного преподавателем задания написать программу на заданном языке программирования. В ходе написания программы реализовать вывод всех входных и выходных данных.

Программа должна быть хорошо прокомментирована и описана. В описании должны быть четко указаны назначение и состав всех используемых входных, выходных и внутренних переменных, а также блоков программы.

2. Используя исходный текст программы, необходимо рассчитать шесть базовых метрик Холстеда.

3. Составить отчет о проделанной работе.

Задание 3.2. Проверка целостности программного кода

Цель и задачи исследования. *Цель* — получить навыки в обнаружении фактов изменения данных, проведения контроля целостности данных с помощью использования механизма хеш-функций; научиться вычислять хеш-свертки для файлов и, используя их, выполнять контроль неизменности данных.

Задачи. Необходимо выполнить задание и предоставить отчет, содержащий описание выполняемых действий, снимки экрана, подтверждающие факт выполнения работы.

Практическая часть. Исследование выполняется в несколько этапов.

1. В первой части работы студенту предлагается написать простое консольное приложение на языке программирования C++, в котором будет реализовано обращение к собственному методу, например вычисляющему несложное арифметическое выражение, создание объекта, и осуществляться вывод текстовых сообщений. (Возможно использование текстов программ, разработанных студентами в рамках других дисциплин или примеров программ, распространяющихся вместе с учебной литературой.)

2. Для вычисления хеш-сверток предлагается использовать программы `WinMD5deep` или `WinSHA-1Sum`, которые можно бесплатно скачать из Интернета.

3. В текстовом редакторе Microsoft Word необходимо создать таблицу для занесения результатов экспериментов, в которую в первый столбец заносится описание указываемого действия, во второй — хеш-свертка, в третий — выводы (табл. 3.4).

Таблица 3.4. Результаты эксперимента

№ п/п	Действие	Значение хеш-суммы	Вывод
1	Исходная программа	`ed1cf0dea20831fb26661c10ca653 40e3a3ea616`	Не влияет

4. Студент формирует исполняемый файл при помощи указанного ПО, вычисляет его хеш-сумму, заносит результат в таблицу. Для продолжения выполняет следующие действия:

- изменяет текст выводимых сообщений, формирует исполняемый файл, вычисляет его хеш-сумму, заносит результат в таблицу, проводит сравнение хешей полученного результата и исходной программы и заносит результат сравнения в таблицу;
- выполняет действия, аналогичные предыдущему пункту, изменив имя вызываемого метода;
- выполняет действия, аналогичные предыдущему пункту, изменив имя класса;
- выполняет действия, аналогичные предыдущему пункту, изменив числовые значения переменных в вычисляемом выражении;
- выполняет действия, аналогичные предыдущему пункту, изменив идентификаторы переменных;
- выполняет действия, аналогичные предыдущему пункту, изменив типы используемых в вычисляемом выражении переменных.

5. Во второй части задания студенту предлагается скачать с официального сайта дистрибутив ОС Linux `Debian` для установки по сети (из-за меньшего объема скачиваемого файла). Перед скачиванием сохранить

рассчитанную по алгоритму SHA-1 хеш-сумму. Затем вычислить хеш-сумму скачанного файла и выполнить сравнение хеш-сумм, полученных в результате эксперимента и предоставленной на сайте поставщика дистрибутива.

6. Оформить отчет, сделав выводы о проделанной работе.

Задание 3.3. Анализ потоков данных

Цель исследования. *Цель* — получить навыки в расчете спена идентификатора и метрики Чепина, относящейся к метрикам анализа потока данных.

Пример выполнения работы. Спеном индентификатора называется число повторных появлений идентификатора в тексте программы. Например, если идентификатор встречается в программе n раз, то его спен равен $n - 1$. Значение спена идентификатора связано со сложностью тестирования программы, так как при трассировке программы по этому идентификатору придется ввести 10 контрольных точек.

В данном примере мы рассмотрим вычисление метрик потока данных на примере программы, вычисляющей выражение $y = \sin x$ (табл. 3.5).

Таблица 3.5. Подсчет суммарного спена программы

Идентификатор	*x*	*n*	*s*	*st*	*y*	Суммарный спен программы
Спен	6	5	4	3	2	20

Используя теоретический материал подразд. 3.2, проводим подсчет переменных, входящих в группы, необходимые для вычисления метрики Чепина.

Для расчета метрики ввода-вывода Чепина используются только переменные ввода-вывода, поэтому относительно нашего примера используются только переменные *x*, *y* и константа *cst*. Все результаты заносятся в табл. 3.6, представленную ниже.

Таблица 3.6. Результаты выполнения работы

Переменные	Полная метрика Чепина				Метрика ввода-вывода Чепина			
Группы переменных	*P*	*M*	*C*	*T*	*P*	*M*	*C*	*T*
Переменные в группе	*x*	*y*, *n*	*bs*, *cns*	—	*x*	*y*, *n*	*cns*	—
Значения переменных	1	2	2	0	1	1	1	0
Значение метрики	$Q = 1 \cdot 1 + 2 \cdot 2 + 3 \cdot 2 + + 0{,}5 \cdot 0 = 11$				$Q = 1 \cdot 1 + 2 \cdot 1 + + 3 \cdot 1 = 6$			

Практическая часть. Исследование выполняется в несколько этапов.

1. На основании выданного преподавателем задания написать программу на заданном языке программирования. В ходе написания программы реализовать вывод всех входных и выходных данных.

Программа должна быть хорошо прокомментирована и описана. В описании должны быть четко указаны назначение и состав всех используемых входных, выходных и внутренних переменных, а также блоков программы.

2. Используя исходный текст программы, необходимо рассчитать и занести в таблицу аналогично приведенному примеру:

- полный спен программы;
- полную метрику Чепина;
- метрику Чепина ввода-вывода.

3. Сделать выводы по проделанной работе и оформить отчет.

Задание 3.4. Использование метрик стилистики

Цель исследования. *Цель* — изучить метрики стилистики и понятности программ.

Практическая часть. Исследование выполняется в несколько этапов.

1. Студенту предлагается написать простое консольное приложение на языке программирования C++, в котором будет реализовано обращение к собственному методу, например вычисляющему несложное арифметическое выражение, создание объекта, и осуществляться вывод текстовых сообщений. (Возможно использование текстов программ, разработанных студентами в рамках других дисциплин или примеров программ, распространяющихся вместе с учебной литературой.)

Программа должна быть хорошо прокомментирована и описана. В описании должны быть четко указаны назначение и состав всех используемых входных, выходных и внутренних переменных, а также блоков программы.

2. Далее, используя теоретический материал подразд. 3.2, рассчитать метрику уровня комментированности программы, разбив исходный код на равные части.

3. Сделать выводы об уровне комментированности программы и составить отчет о проделанной работе.

Задание 3.5. Выполнение измерений характеристик кода в среде Visual Studio

Цель и задачи исследования. *Цель* — получить навыки в получении метрик оценки качества кода при выполнении работ в IDE Visul Studio.

Задачи исследования. Студенту необходимо выполнить задание и предоставить отчет, содержащий описание выполняемых действий, снимки экрана, подтверждающие факт выполнения работы.

Практическая часть. Исследование выполняется в несколько этапов.

1. Студенту необходимо по заданию преподавателя создать приложение на языке программирования C#.

2. Выбрав пункт меню Анализ, выбрать подпункт Вычислить метрики кода для решения. Альтернативным способом может быть вызов контекстного меню нажатием правой клавиши мыши на названии решения в обозревателе решений и выбор пункта Рассчитать метрики кода.

3. Дождавшись появления окна Результаты метрик кода, провести анализ разработанного приложения на основе предоставленных метрик:

- индекс удобства поддержки;
- цикломатическая сложность программы;
- глубина наследования;
- взаимозависимость классов;
- количество строк кода.

4. Оценив рассчитанные метрики, сделать выводы и выполнить отчет по проделанной работе.

Задание 3.6. Выполнение измерений характеристик кода в среде Eclipse

Цель и задачи исследования. *Цель* — получить навыки в получении метрик оценки качества кода при выполнении работ в IDE Eclipse.

Задачи. Студенту необходимо выполнить задание и предоставить отчет, содержащий описание выполняемых действий, снимки экрана, подтверждающие факт выполнения работы.

Практическая часть. Исследование выполняется в несколько этапов.

1. Студенту необходимо по заданию преподавателя создать приложение на языке программирования Java.

2. Для оценки метрик кода, написанного на языке программирования Java, в IDE Eclipse потребуется загрузка и установка плагина Eclipse Metrics. Для этого необходимо:

- загрузить JAR-файл Eclipse Metrics;
- сохранить загруженный JAR-файл в каталоге плагинов Eclipse;
- перезапустить Eclipse.

3. Подключаемый модуль Eclipse Metrics можно настраивать (включить или отключить) для каждого Java-проекта. Чтобы подключить плагин, необходимо открыть пункт меню File и выбрать подпункт Properties проекта, в открывшемся окне выбрать вкладку Metrics и установить флажок Enable Metrics Gathering. Нажмите OK для повторной компиляции проекта (если, конечно, в предпочтениях проекта настроена автоматическая сборка) и вычисления метрик. При выходе метрик за пределы указанного диапазона в представлении Problems View генерируются и отображаются предупреждения, а блок кода помечается.

4. Для установки предпочтений необходимо выбрать пункт меню Windows, выбрать подпункт Preferences, а в нем вкладку Metrics для выполнения следующих действий:

- включение или отключение метрик;
- установка максимальных значений для включенных метрик;
- настройка метрик в соответствии с требованиями проекта.

5. Дождавшись появления окна `Результаты метрик кода`, провести анализ разработанного приложения на основе предоставленных метрик:

- число строк в методе;
- цикломатическая сложность программы;
- число операторов;
- максимальное число уровней вложенности в методе.

6. Оценив рассчитанные метрики, сделать выводы и выполнить отчет по проделанной работе.

Словарь терминов

Валидация — процесс проверки данных на соответствие различным критериям.

Валидация DTD — проверка соответствия вашего кода указанному в Document Type Definition.

Валидация синтаксиса — проверка на наличие синтаксических ошибок.

Ветка — копия оригинального проекта.

Декомпиляция — получение кода языка высокого уровня из программы на машинном языке или ассемблере.

Дизассемблирование — процесс и (или) способ получения исходного кода программы на ассемблере из программы в машинных кодах.

Интегрированная среда разработки (Integrated Development Environment, IDE) — комплексное средство, включающее все необходимое программисту для создания ПО: редактор исходного кода с подсветкой, средства автоматизации сборки, отладчик, а также большой набор прочих инструментов, упрощающих и ускоряющих процесс работы с кодом.

Корректность программного средства — соответствие проверяемого объекта некоторому эталонному объекту или совокупности более или менее формализованных эталонных характеристик и правил.

Метрика ПО — мера, позволяющая получить численное значение некоторого свойства ПО или его спецификаций.

Обратное проектирование (reverse ingeneering) — процесс исследования и анализа машинного кода, направленный на понимание общих механизмов функционирования программы и на перевод программы в машинных кодах на более высокий уровень абстракции.

Отладка — этап разработки программы, на котором обнаруживают, локализуют и устраняют ошибки.

Представления — дополнение к редакторам, где выводится информация сопроводительного или дополнительного характера, как правило, о файле, находящемся в редакторе.

Проверка на посторонние элементы — проверка, которая выявляет все, что есть в коде, но отсутствует в DTD.

Программные измерительные мониторы (ПИМ) — совокупность команд или программ, выполняемых исключительно с целью проведения измерений.

Профилирование — сбор характеристик работы программы, таких как время выполнения отдельных фрагментов, число верно предсказанных условных переходов, число кеш-промахов и т. д.

Профилировщик, или **профайлер**, — инструмент, используемый для профилирования.

Ревьюирование (рецензирование, обзор, инспектирование кода) — один из стандартных методов программной инженерии, направленный на проверку кода, его анализ и составление рецензии о проверенном коде, основной целью которого является повышение качества программного кода.

Серверная валидация — работает в рамках программного кода, размещенного на стороне сервера.

Система контроля версий (Version Control System, VCS, или Revision Control System) (СКВ) — ПО для облегчения работы с изменяющейся информацией. Система управления версиями позволяет хранить несколько версий одного и того же документа, при необходимости возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение, и многое другое.

Снифферы — программы, позволяющие отследить сетевой трафик, генерируемый программой.

Спецификация — формальное описание функций и данных программы, с которыми эти функции оперируют. Различают видимые данные, т. е. входные и выходные параметры, а также скрытые данные, которые не привязаны к реализации и определяют интерфейс с другими функциями.

Статический анализ ПО — анализ ПО, производимый (в отличие от динамического анализа) без реального выполнения исследуемых программ. В большинстве случаев анализ производится над какой-либо версией исходного кода, хотя иногда анализу подвергается какой-то вид объектного кода. Для анализа обычно применяется специальное ПО.

Тестирование — метод, направленный на обнаружение как можно большего количества ошибок в программном продукте.

Трассировка — процесс пошагового выполнения программы, в ходе которого разработчик видит последовательность выполнения команд и значения переменных при их выполнении.

Утилита — компьютерная программа, расширяющая стандартные возможности оборудования и операционных систем, выполняющая узкий круг специфических задач.

Список литературы

Нормативные документы

1. ГОСТ Р ИСО/МЭК 25010—2015. Информационные технологии (ИТ). Системная и программная инженерия. Требования и оценка качества систем и программного обеспечения (SQuaRE). Модели качества систем и программных продуктов.

2. ГОСТ Р ИСО/МЭК 25021—2014. Информационные технологии (ИТ). Системная и программная инженерия. Требования и оценка качества систем и программного обеспечения (SQuaRE). Элементы показателя качества.

3. ГОСТ Р ИСО/МЭК 25040—2014. Информационные технологии (ИТ). Системная и программная инженерия. Требования и оценка качества систем и программного обеспечения (SQuaRE). Процесс оценки.

4. ГОСТ Р ИСО/МЭК 25045—2015. Информационные технологии (ИТ). Системная и программная инженерия. Требования и оценка качества систем и программного обеспечения (SQuaRE). Модуль оценки восстанавливаемости.

5. ГОСТ Р ИСО/МЭК 25051—2017. Информационные технологии (ИТ). Системная и программная инженерия. Требования и оценка качества систем и программного обеспечения (SQuaRE). Требования к качеству готового к использованию программного продукта (RUSP) и инструкции по тестированию.

Основная и дополнительная литература

6. *Аблязов Р.З.* Программирование на ассемблере на платформе x86-64 / Р.З.Аблязов. — М. : ДМК-Пресс, 2011.

7. *Бейзер Б.* Тестирование черного ящика. Технология функционального тестирования программного обеспечения и систем / Б.Бейзер. — СПб. : Питер, 2004.

8. *Генельт А.Е.* Учебно-методическое пособие по дисциплине «Управление качеством разработки ПО» / А.Е.Генельт. — СПб. : Изд-во СПГУ ИТМО, 2007.

9. *Гудлиф П.* Ремесло программиста. Практика написания хорошего кода / П.Гудлиф. — СПб. : Символ-Плюс, 2009.

10. Идеальный код / под ред. Э.Орама, Г.Уилсона. — СПб. : Питер, 2011.

11. *Касперский К.* Искусство дизассемблирования / К.Касперский, Е.Рокко. — СПб. : БХВ-Петербург, 2008.

12. *Ковалевская Е.В.* Метрология, качество и сертификация программного обеспечения : учеб. программа, руководство по изучению дисциплины,

учебное пособие, практикум по курсу, тестовые задания по дисциплине / Е. В. Ковалевская. — М. : МГУЭСИ, 2004.

13. *Куликов С. С.* Тестирование программного обеспечения. Базовый курс : практ. пособие / С. С. Куликов. — Минск : Четыре четверти, 2015.

14. *Лаврищева Е. М.* Методы и средства инженерии программного обеспечения : учебник / Е. М. Лаврищева, В. А. Петрухин. — М. : МФТИ (ГУ), 2006.

15. *Лаврищева Е. М., Слабоспитская О. А.* Технология моделирования изменяемых программных продуктов и систем // XII Международ. научно-практ. конф. «Теоретические и прикладные аспекты построения программных систем» TAAPSD—2015, Киев, 23—26 ноября 2015 г.

16. *Макконнелл С.* Совершенный код. Мастер-класс: Практическое руководство по разработке программного обеспечения : пер. с англ. / С. Макконнелл. — М. : Русская редакция, 2010.

17. Методы верификации программного обеспечения / Р. Е. Гурин, И. В. Рудаков, А. В. Ребриков // Наука и образование. МГТУ им. Баумана [Электрон. журн.]. — 2015. — № 10. — С. 235—251.

18. *Орлов С. А.* Программная инженерия / С. А. Орлов. — СПб. : Питер, 2016.

19. Основные методы тестирования программного обеспечения : учеб. пособие / А. М. Дворянкин, А. А. Ерофеев, А. В. Аникин. — Волгоград : ВолгГТУ, 2015.

20. *Патрыка Т. Л.* Операционные системы, среды и оболочки / Т. Л. Патрыка, И. И. Попов. — М. : Форум, 2011.

21. *Спинеллис Д.* Анализ программного кода на примере проектов Open Source / Д. Спинеллис. — М. : Вильямс, 2004.

22. Тестирование программного обеспечения. Фундаментальные концепции менеджмента бизнес-приложений / С. Канер, Д. Фолк, Енг Кек Нуген. — М. : ДиаСофт, 2001.

23. Управление проектами / И. И. Мазур, В. Д. Шапиро, Н. Г. Ольдерогге. — М. : Омега-Л, 2004.

24. *Sikorski M., Honig A.* Practical malware analysis. — San Francisco : No Starch Press Inc., 2012.

Интернет-ресурсы

25. *Жарко Е. Ф.* Сравнение моделей качества. — URL : http://vspu2014.ipu.ru/proceedings/prcdngs/4585.pdf.

26. https://biz30.timedoctor.com/ru/

27. https://dic.academic.ru/dic.nsf/ruwiki/1118556

28. http://se.inf.ethz.ch/~meyer/publications/acm/code_review.pdf

29. https://habr.com/post/151450/

30. https://vuzlit.ru/990798/standarty_kachestva_programmnogo_obespecheniya

31. https://ru.wikipedia.org/wiki/FURPS

32. https://habr.com/company/pt/blog/149303/

33. http://tagline.ru/version-contrd-systems-rating/

Оглавление

Учебное издание

Поколодина Елена Владиславна,
Долгова Наталья Александровна,
Ананьев Дмитрий Вячеславович

Ревьюирование программных модулей

Учебник

Редактор *Т. Г. Парканова*
Компьютерная верстка: *Р. Ю. Волкова*
Корректор *А. М. Дэймос*

Изд. № 101119733. Подписано в печать 31.01.2020. Формат 60 × 90/16.
Гарнитура «Балтика». Печать офсетная. Бумага офс. № 1. Усл. печ. л. 13,0.
Тираж 1 500 экз. Заказ №1387.

ООО «Издательский центр «Академия». www.academia-moscow.ru
129085, г. Москва, пр-т Мира, д. 101В, стр. 1.
Тел./факс: 8 (495) 648-05-07, 616-00-29.
Сертификат соответствия № РОСС RU.АМ05.Н01493 от 30.05.2019.

Отпечатано по заказу АО «ПолиграфТрейд»
в АО «Первая Образцовая типография»,
филиал «УЛЬЯНОВСКИЙ ДОМ ПЕЧАТИ»
432980, Россия, г. Ульяновск, ул. Гончарова, 14