ЛЕКЦИЯ 1

ТЕМА: Тестирование программ. Определения. Типы тестов и их роль в процессе разработки программного обеспечения. Основные понятия и принципы тестирования программных продуктов.

Что такое тестирование и откуда оно появилось

На протяжении десятилетий развития разработки ПО к вопросам тестирования и обеспечения качества подходили очень и очень по-разному. Можно выделить несколько основных «эпох тестирования».

В **50–60-х годах** прошлого века процесс тестирования был предельно формализован, отделён от процесса непосредственной разработки ПО и «математизирован». Фактически тестирование представляло собой скорее **отладку** программ.

В **70-х годах** фактически родились две фундаментальные идеи тестирования: тестирование сначала рассматривалось как **процесс доказательства работоспособности программы** в некоторых заданных условиях (positive testing), а затем — строго наоборот: **как процесс доказательства неработоспособности программы** в некоторых заданных условиях (negative testing). Это внутреннее противоречие не только не исчезло со временем, но и в наши дни многими авторами совершенно справедливо отмечается как две взаимодополняющие цели тестирования.

Тестирование «приобрело» в 70-е годы:

- тестирование позволяет удостовериться, что программа соответствует требованиям;
- тестирование позволяет определить условия, при которых программа ведёт себя некорректно.

В **80-х годах** произошло ключевое изменение места тестирования в разработке ПО: вместо одной из финальных стадий создания проекта тестирование стало **применяться на протяжении всего цикла разработки**, что позволило в очень многих случаях не только быстро обнаруживать и устранять проблемы, но даже предсказывать и предотвращать их появление.

В этот же период времени отмечено бурное развитие и **формализация методологий тестирования** и появление **первых элементарных попыток автоматизировать тестирование**.

В **90-х годах** произошёл переход от тестирования как такового к более всеобъемлющему процессу, который называется «**обеспечение качества**», охватывает весь цикл разработки ПО и затрагивает процессы планирования, проектирования, создания и выполнения тест-кейсов, поддержку имеющихся тест-кейсов и тестовых окружений. Тестирование вышло на качественно новый уровень, который естественным образом привёл к дальнейшему развитию методологий, появлению достаточно мощных **инструментов управления процессом тестирования** и **инструментальных средств автоматизации тестирования**, уже вполне похожих на своих нынешних потомков.

В **нулевые годы** нынешнего века развитие тестирования продолжалось в контексте поиска всё новых и новых путей, методологий, техник и подходов к обеспечению качества. Серьёзное влияние на понимание тестирования оказало появление гибких методологий разработки и таких подходов, как «**разработка под управлением тестированием** (test-driven development, TDD)». Автоматизация тестирования уже воспринималась как обычная неотъемлемая часть большинства проектов, а также стали популярны идеи о том, что во главу процесса тестирования следует ставить не соответствие программы требованиям, а её способность **предоставить конечному пользователю возможность эффективно решать свои задачи**.

Основные характеристики **современного этапа** развития тестирования:

- ✓ гибкие методологии и гибкое тестирование;
- ✓ глубокая интеграция с процессом разработки;
- ✓ широкое использование автоматизации;
- ✓ колоссальный набор технологий и инструментальных средств;
- ✓ кросс-функциональность команды (когда тестировщик и программист во многом могут выполнять работу друг друга).

Почему важен процесс тестирования?

Процесс разработки ПО невозможен без контроля качества разрабатываемого продукта.

Процесс тестирования ПО представляет собой столь же неотъемлемую часть процесса разработки, как и проектирование.

Тестирование позволяет оценить качество разрабатываемого продукта.

Цель тестирования состоит в получении объективной информации о качестве продукта.

Тестирование ПО – процесс исследования ПО с целью получения информации о качестве продукта (ISO 9126) с учетом следующих составляющих:

- функциональность;
- надёжность;
- удобство использования (практичность);
- производительность (эффективность);
- удобство сопровождения;
- переносимость (мобильность).

✅ **Тестирование ПО** – процесс проверки соответствия заявленных к продукту требований и реально реализованной функциональности, осуществляемый путем наблюдения за его работой в искусственно созданных ситуациях и на ограниченном наборе тестов, выбранных определенном образом.

✅ **Тестирование** – это процесс, который требует анализа, планирования и тщательной подготовки.

✅ **Тестирование** – это один из видов оценки системы с целью найти различия между тем, какой она должна быть, и тем, какая она есть.

✅ **Тестирование ПО** – это процесс, который осуществляется специально обученными специалистами.

✓ **Тестирование программного обеспечения** — процесс анализа программного средства и сопутствующей документации с целью выявления дефектов и повышения качества продукта.

✓ **Тестирование** – это:

- набор большого количества активностей таких как планирование и управление;
- выбор тестовых условий;
- разработка и выполнение тестовых сценариев;
- проверка результатов;
- оценка критериев выхода;
- создание отчетов о процессе тестирование и об испытываемой системе и закрытие или завершающие действия после того, как фаза тестирования была выполнена;
- рецензирование документации (включая исходный код).

Основные процессы тестирования:

- Планирование и контроль (планирование, стратегия, контроль).
- Анализ и дизайн (подготовка и анализ, составление, определение результатов).
- Реализация и выполнение (разработка, приоритезация, тестовое окружение, тестовые результаты, исполнение).
- Оценка критериев выхода и составление отчетности.
- Принятие решения о достаточности тестов (критерии завершенности тестов).

Терминология

✓ **Тестирование –** это обнаружение (доказательство наличия) ошибок (несоответствий, неполноты, двусмысленностей и т.д.) в текущем состоянии.

✓ **Валидация** – доказательство того, что в результате разработки ПО достигнуты те цели, которые планировали достичь благодаря её использованию.

✓ **Валидация программной системы** – проверка соответствия системы ожиданиям заказчика.

✓ **Верификация** – достижение гарантии того, что верифицируемый объект:

- соответствует требованиям,
- реализован без непредусмотренных функций,
- удовлетворяет проектным спецификациям и стандартам.

Процесс верификации проводится сверху вниз: от общих требований, заданных в техническом задании и/или спецификации на всю информационную систему до детальных требований на программные модули и их взаимодействие.

Процесс верификации включает в себя:

- инспекции;
- тестирование кода;
- анализ результатов тестирования;
- формирование и анализ отчетов о проблемах.

Философия тестирования

Тестирование ПО охватывает целый ряд видов деятельности:

– постановка задачи для теста;

– проектирование, написание тестов;

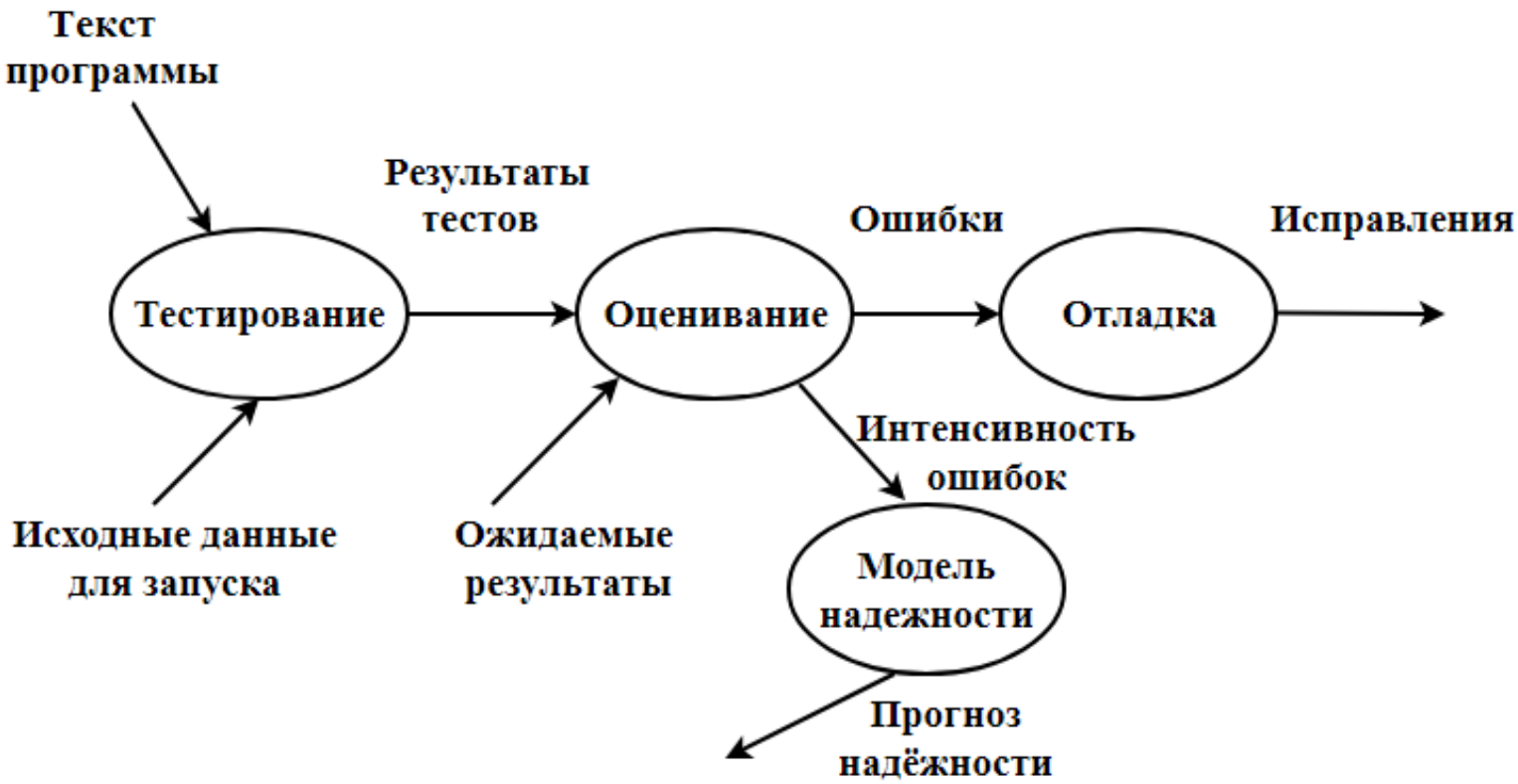
– тестирование тестов;

– выполнение тестов;

– изучение результатов тестирования.

Основная проблема тестирования – определение достаточности множества тестов для истинности вывода о правильности реализации программы. А также нахождения множества тестов, обладающих этим свойством.

Информационные потоки процесса тестирования

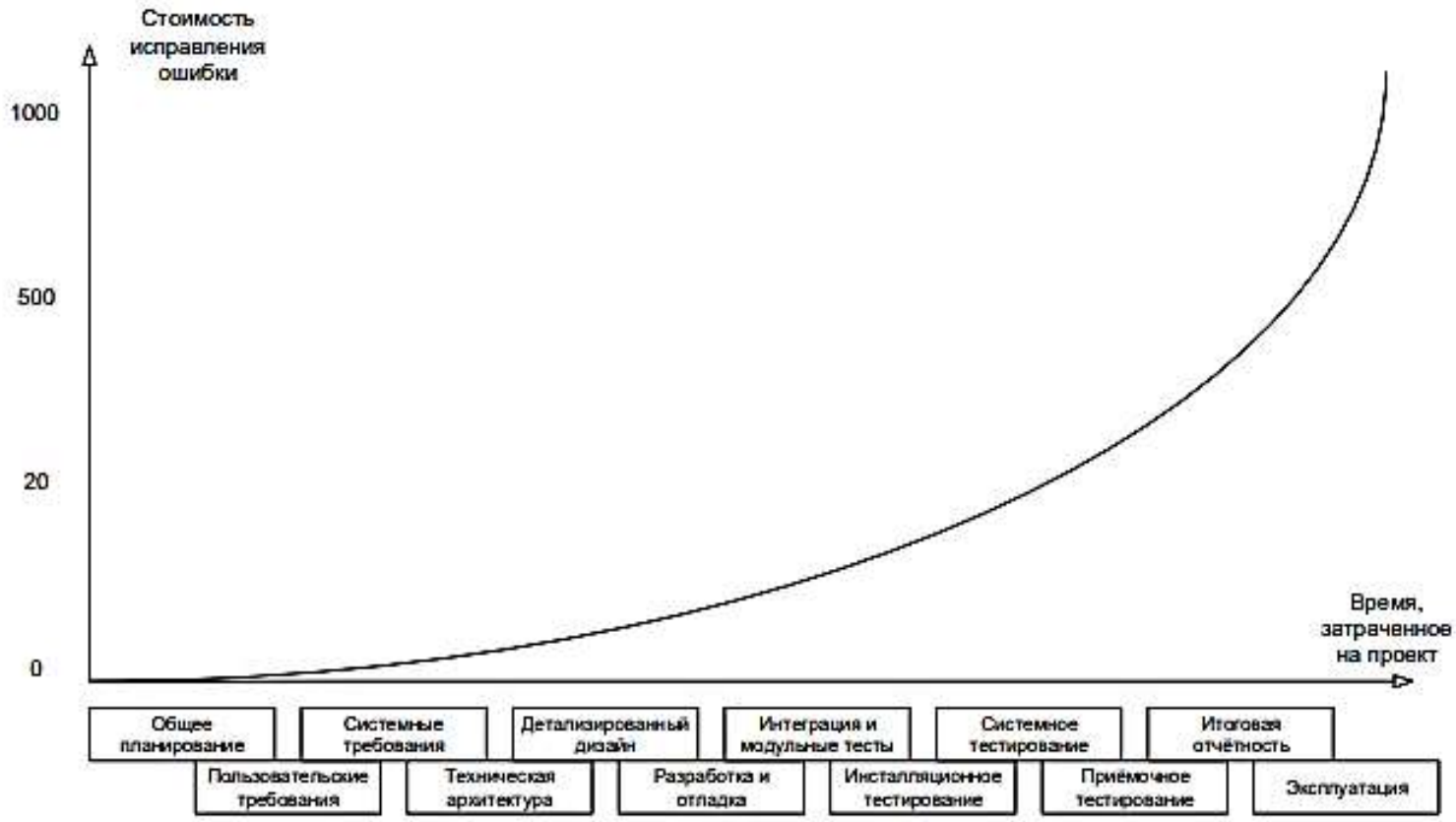


На входе процесса тестирования три потока:

- текст программы;
- исходные данные для запуска программы;
- ожидаемые результаты.

Стоимость ошибки

Ошибки в ПО – все возможные несоответствия между демонстрируемыми характеристиками его качества и сформулированными или подразумеваемыми требованиями и ожиданиями пользователей.



Чем раньше ошибка была найдена, тем стоимость ниже. Чем позднее – стоимость выше и растет экспоненциально, тем сложнее и дороже будет её решение.

Если проблема в требованиях будет выяснена на стадии анализа требований, её решение может свестись к исправлению пары слов в тексте, в то время как недоработка, вызванная пропущенной проблемой в требованиях и обнаруженная на стадии эксплуатации, может даже полностью уничтожить проект.

Рисунок 1.1 – Стоимость исправления ошибки в зависимости от момента её обнаружения (Брайан Хэнкс)

Оценка стоимости ошибок

Некоторое время назад ряд компаний провел исследование оценки стоимости ошибок, возникающих на разных этапах создания программ. Каждая фирма действовала независимо, тем не менее результаты получены примерно одинаковые: если стоимость усилий, необходимых для обнаружения и устранения ошибок на стадии написания кода, принять за единицу, то стоимость выявления и устранения ошибки на стадии выработки требований будет в 5—10 раз меньше, а стоимость обнаружения и устранения ошибки на стадии сопровождения — в 20 раз больше (рисунок 1.2).

/ 0,1—0,2	Этап Время выработки требований
/ 0,5	Проектирование
/ 1	Кодирование
/ 2	Тестирование компонентов
/ 5	Приемка
20	Поддержка и обслуживание

Рисунок 1.2 - Оценка стоимости ошибок на разных этапах создания ПО

Откуда берется такая высокая стоимость ошибки? Ко времени обнаружения ошибки в требованиях группа разработчиков уже могла потратить время и усилия на создание проекта по этим ошибочным требованиям. В результате проект, вероятно, придется отбросить или пересмотреть.

Истинная природа ошибки может быть замаскирована; при проведении тестирования и проверок на данной стадии все думают, что имеют дело с ошибками проектирования, и значительное время и усилия могут быть потрачены впустую.

В зависимости от того, где и когда при работе над проектом разработки программного приложения был обнаружен дефект, цена его может разниться в 50—100 раз. Причина состоит в том, что для его исправления придется затратить средства на некоторые (или все) нижеперечисленные действия.

- Повторная спецификация.
- Повторное проектирование.
- Повторное кодирование.
- Повторное тестирование.
- Замена заказа — сообщить клиентам и операторам о необходимости заменить дефектную версию исправленной.

- Внесение исправлений — выявить и устранить все неточности, вызванные неправильным функционированием ошибочно специфицированной системы, что может потребовать выплаты определенных сумм возмущенным клиентам, повторного выполнения определенных вычислительных задач на ЭВМ и т. п.
- Списание той части работы (кода, части проектов и т. п.), которая выполнялась с наилучшими побуждениями, но оказалась ненужной, когда обнаружилось, что все это создавалось на основе неверных требований.
- Отозвание дефектных версий встроенного программного обеспечения и соответствующих руководств. Если принять во внимание, что программное обеспечение сегодня встраивается в различные изделия — от наручных часов и микроволновых печей до автомобилей, — такая замена может коснуться как этих изделий, так и встроенного в них программного обеспечения.
- Выплаты по гарантийным обязательствам.
- Ответственность за изделие, если клиент через суд требует возмещение убытка, причиненного некачественным программным продуктом.
- Затраты на обслуживание — представитель компании должен посетить клиента, чтобы установить новую версию программного обеспечения.
- Создание документации.

Одним аргументом в пользу тестирования требований является то, что, по разным оценкам, в них зарождается от ½ до ¾ всех проблем с программным обеспечением. В итоге есть риск, что получится так, как показано на рисунке 1.3. Поскольку постоянно говорят «документация и требования», а не просто «требования», то стоит рассмотреть перечень документации, которая должна подвергаться тестированию в процессе разработки ПО.

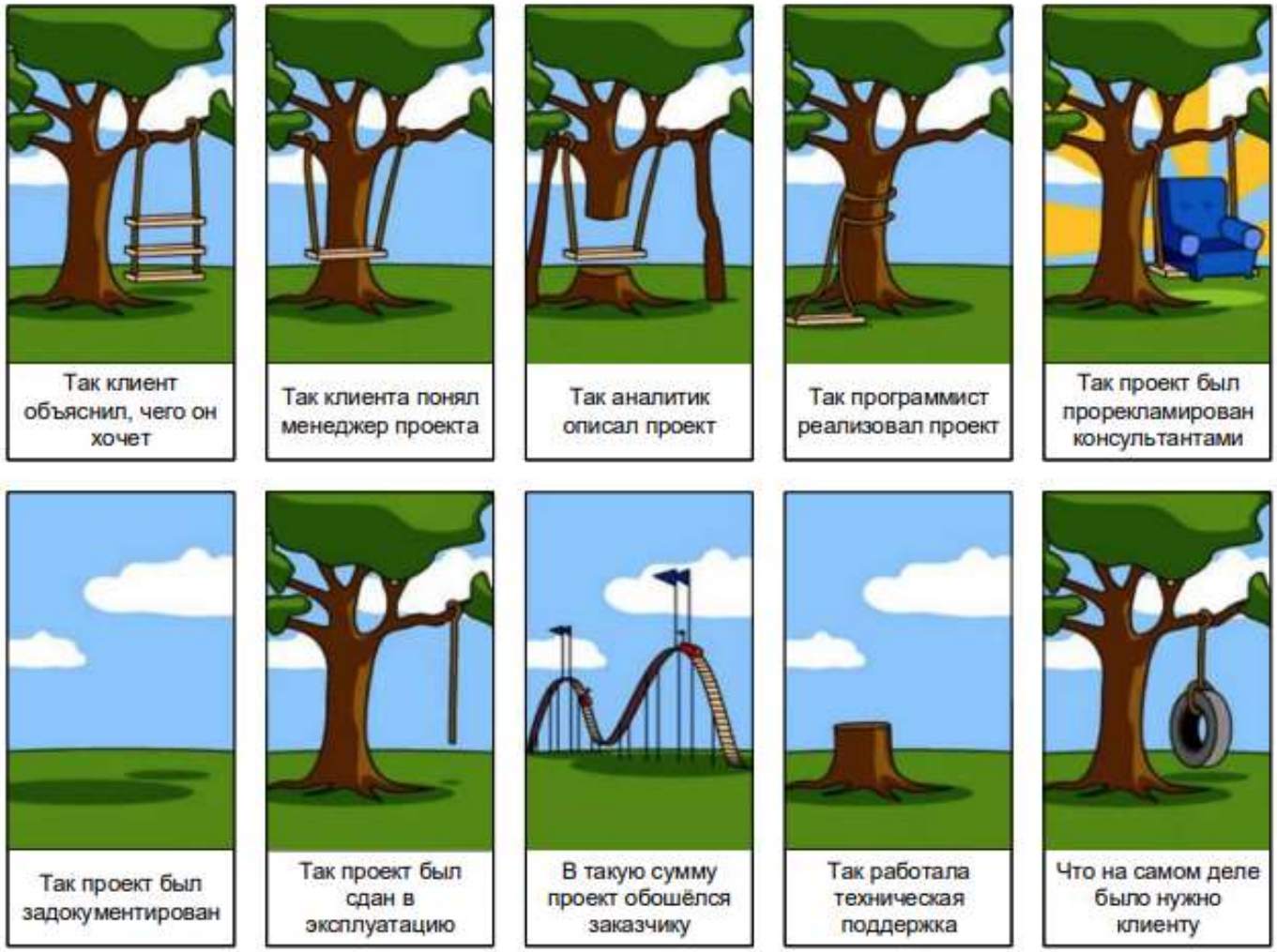


Рисунок 1.3 — Типичный проект с плохими требованиями

Каждый **тест** определяет:

– свой набор исходных данных и условий для запуска программы;

– набор ожидаемых результатов работы программы.

Другое название теста — **тестовый вариант (ТВ)**. Полную проверку программы гарантирует исчерпывающее тестирование. Оно требует проверить все наборы исходных данных, все варианты их обработки и включает большое количество тестовых вариантов. Исчерпывающее тестирование во многих случаях остается только мечтой — срабатывают ресурсные ограничения (прежде всего, ограничения по времени).

Хорошим считают тестовый вариант с высокой вероятностью обнаружения еще не раскрытой ошибки.

Успешным называют тест, который обнаруживает до сих пор не раскрытую ошибку.

Целью проектирования тестовых вариантов является систематическое обнаружение различных классов ошибок при минимальных затратах времени и стоимости.

Каждый **тестовый вариант** формируется в следующем виде:

ТВ:

Исходные данные (ИД):

Ожидаемые результаты (ОЖ.РЕЗ.):

Пример:

Тестовый вариант 1 (четный массив, найден промежуточный элемент)

ТВ1:

ИД: M = (15, 20, 25, 30, 35, 40); Key=25.

ОЖ.РЕЗ.: Result=True; I=3.

Тестовый сценарий (тест)

Тестовый сценарий — это *описание начальных условий*, входных данных, *действий пользователя* и ожидаемого результата.

Хорошая практика — написание тестовых сценариев на основании вариантов использования (Use cases).

Варианты названий **тестового варианта**:

- Атомарный тест.
- Тестовый вариант.
- Вариант тестирования.
- Тестовый случай.
- Тест-кейс.

Тест кейс Тестовый случай (Test Case) / Тестовый набор (Test Suite)

- **Тестовый случай (Test Case)** — это артефакт, описывающий совокупность шагов, конкретных условий и параметров, необходимых для проверки реализации тестируемой функции или её части.
- **Тестовый набор (Test Suite)** — это набор тестов, реализующих бизнес-задачу, выполняемую тестируемой системой.

Тестовый набор включает в себя, кроме тестовых сценариев, ещё и тестовые данные и *правила их генерации*.

На примере тестового набора можно рассмотреть так:

тестовый набор — это кирпичная стена,

тестовый случай – это один кирпич из стены.

Тестовый сценарий — это *описание начальных условий*, входных данных, *действий пользователя* и ожидаемого результата.

Тест дизайн (Test Design) – это этап процесса тестирования ПО, на котором проектируются и создаются тестовые случаи (Test Case), в соответствии с определёнными ранее критериями качества и целями тестирования.

Существуют системы для описания тест-кейсов, например: **TestLink, TestRail, QaTraq**.

Тест кейсы разделяют по ожидаемому результату на **позитивные** и **негативные**:

- **Позитивный тест кейс** использует только корректные данные и проверяет, что приложение правильно выполнило вызываемую функцию.
- **Негативный тест кейс** оперирует как корректными, так и некорректными данными (минимум 1 некорректный параметр) и ставит целью проверку исключительных ситуаций, а также проверяет, что вызываемая приложением функция не выполняется при срабатывании валидатора.

Внимание! Очень частая ошибка! Негативные тесты НЕ предполагают возникновения в приложении ошибки. Напротив — они предполагают, что верно работающее приложение даже в критической ситуации поведёт себя правильным образом (в примере с делением на ноль, например, отобразит сообщение «Делить на ноль запрещено»).

- **Высокоуровневый тест-кейс (high level test case)** — тест-кейс без конкретных входных данных и ожидаемых результатов.
- **Низкоуровневый тест-кейс (low level test case)** — тест-кейс с конкретными входными данными и ожидаемыми результатами. Представляет собой «полностью готовый к выполнению» тест-кейс и вообще является наиболее классическим видом тест-кейсов.

- **Спецификация тест-кейса (test case specification)** — документ, описывающий набор тест-кейсов (включая их цели, входные данные, условия и шаги выполнения, ожидаемые результаты) для тестируемого элемента (test item, test object).
- **Спецификация теста (test specification)** — документ, состоящий из:
    - спецификации тест-дизайна (test design specification),
    - спецификации тест-кейса (test case specification)
    - и/или спецификации тест-процедуры (test procedure specification).
- **Тест-сценарий (test scenario, test procedure specification, test script)** — документ, описывающий последовательность действий по выполнению теста (также известен как «тест-скрипт»), а не набор тест-кейсов.

Наборы тестовых сценариев

Это тестовые сценарии, сгруппированные по некоему признаку (например, тестируемой функциональности). Они могут быть как зависящими от последовательности выполнения (результат выполнения предыдущего является предварительным условием для следующего (Test script)), так и независимыми (Test suite).

Наиболее часто выделяемыми наборами являются: Набор тестовых сценариев для Smoke-test и План приёмо-сдаточных испытаний.

Примеры

Тестовый случай (Test Case) — это артефакт, описывающий совокупность шагов, конкретных условий и параметров, необходимых для проверки реализации тестируемой функции или её части.

Ниже приведены несколько примеров сценария тестирования.

Тестовый сценарий для интернет-магазина

Сценарии тестирования, которые можно принять во внимание при проверке приложения для онлайн-покупок, следующие:

Тестовый сценарий 1: проверка работоспособности входа

Тестовые случаи, которые можно рассмотреть для создания:

- Поведение приложения при вводе действительного идентификатора входа и действительного пароля можно проверить.
- Поведение приложения при вводе действительного логина и неверного пароля может быть проверено.
- Поведение приложения при вводе неверного логина и действительного пароля можно проверить.
- Поведение приложения при вводе неверного логина и неверного пароля можно проверить.
- Поведение приложения при входе путем ввода идентификатора входа без пароля можно проверить.
- Поведение приложения при входе путем ввода пароля без идентификатора входа можно проверить.
- Поведение приложения при входе в систему без ввода логина и пароля можно проверить.
- Поведение приложения при выборе забытого пароля.

Тестовый сценарий 2. Проверка функциональности поиска

Тестовые случаи, которые можно рассмотреть для создания:

- Поведение приложения при поиске действующего товара.
- Поведение приложения при поиске недействительного продукта.

Тестовый сценарий 3: проверка деталей продукта

Тестовые случаи, которые можно рассмотреть для создания:

- Поведение приложения при выборе продукта.
- Поведение приложения продукта в списке пожеланий.
- Поведение приложения при добавлении товара в корзину.
- Поведение приложения при выборе опции «Купить сейчас».
- Поведение приложения при вводе неверного адреса.
- Поведение приложения при вводе действительного адреса.
- Поведение приложения при проверке нескольких вариантов оплаты.

Тестовый сценарий 4: проверка функциональности платежа

Тестовые случаи, которые можно рассмотреть для создания:

- Поведение приложения при выборе каждого варианта оплаты.
- Поведение приложения, когда выбран правильный способ оплаты.
- Поведение приложения при выборе неверного варианта оплаты.
- Поведение приложения при успешной оплате.
- Поведение приложения при отклонении платежа.

Тестовый сценарий 5: проверка функциональности деталей заказа

Тестовые случаи, которые можно рассмотреть для создания:

- Поведение приложения при выборе каждого заказа.
- Поведение приложения при выборе опции «Возврат товара».
- Поведение приложения при выборе опции отслеживания товара.
- Поведение приложения при выборе опции «Обзор продукта».

Стандартный **шаблон для тест - кейса**:

Окружение

Предусловия

Шаги

Ожидаемый результат

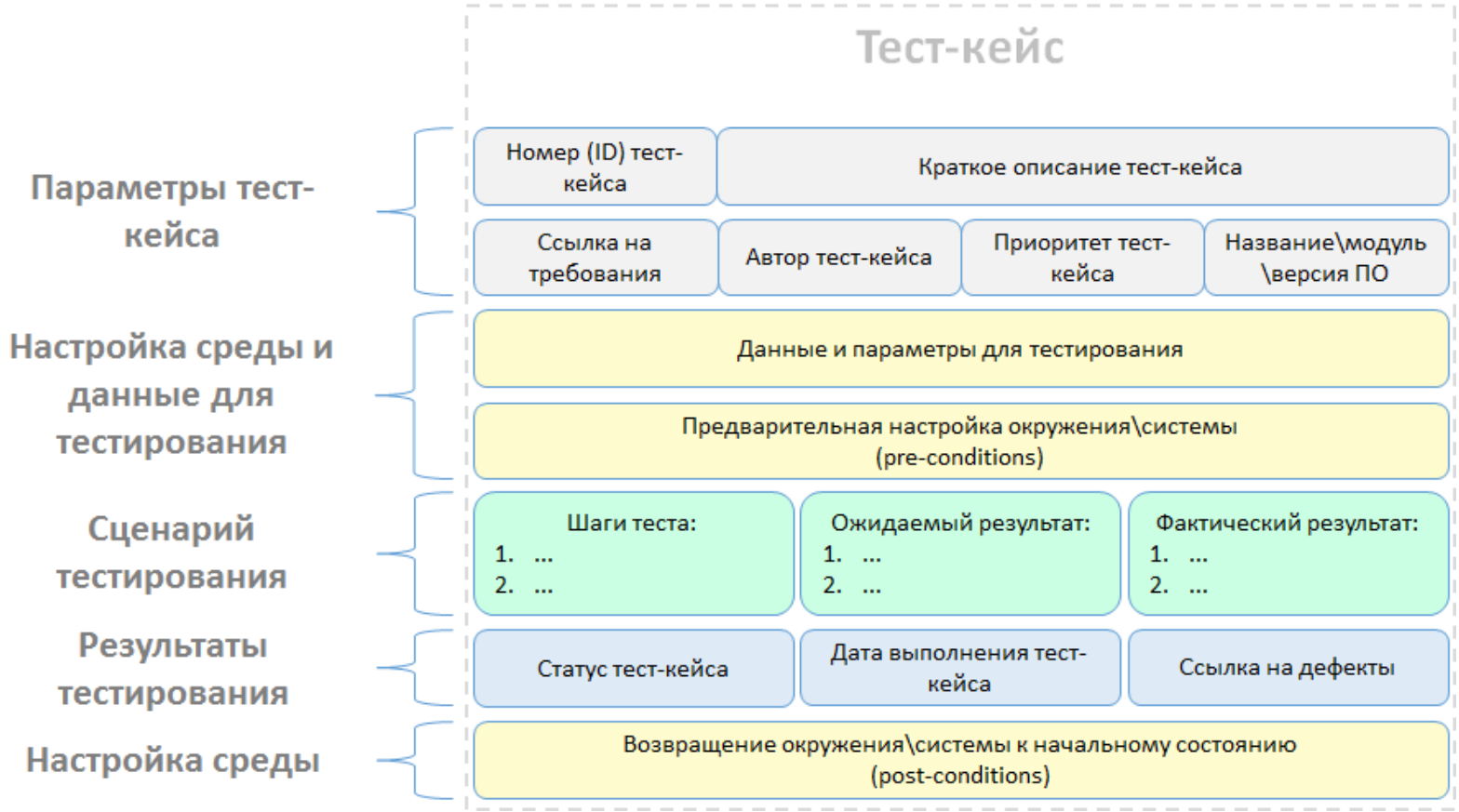
Фактический результат

В зависимости от ситуации поле Окружение можно опустить.

Шаблон тестового кейса

Рассмотрим типовой шаблон теста, где будут показаны основные составные части тест-кейса, приведены важные параметры и атрибуты, которые должен содержать любой хорошо описанный тестовый сценарий.

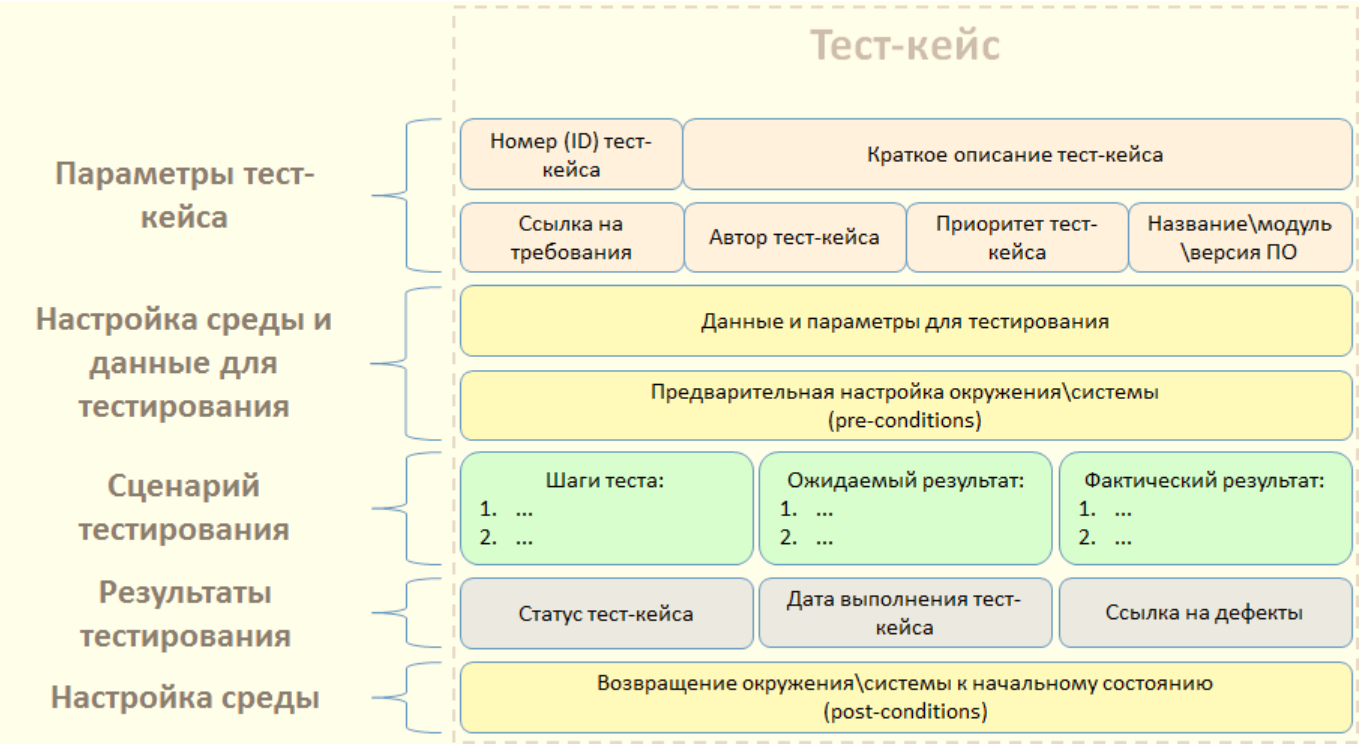
Структура реального тест-кейса может и отличаться от данного шаблона (схема ниже), но тут показан наиболее краткий и, в тоже время, содержащий достаточно информации пример оформления.



Рассмотрим основные поля, необходимые при описании дефекта.

Параметры тест-кейса

- Номер (ID) тест-кейса – уникальный (например, числовой) идентификатор.
- Краткое описание тест-кейса – одна или несколько фраз, из которых ясно, что проверяется данным сценарием.
- Ссылка на требования – прямая ссылка или указание названия и версии документа с требованиями.
- Автор тест-кейса – тестировщик, разработавший тест-кейс.
- Приоритет тест-кейса – насколько важен тест-кейс для проверки данного функционала.
- Название\модуль\версия ПО – точное описание тестируемого ПО.

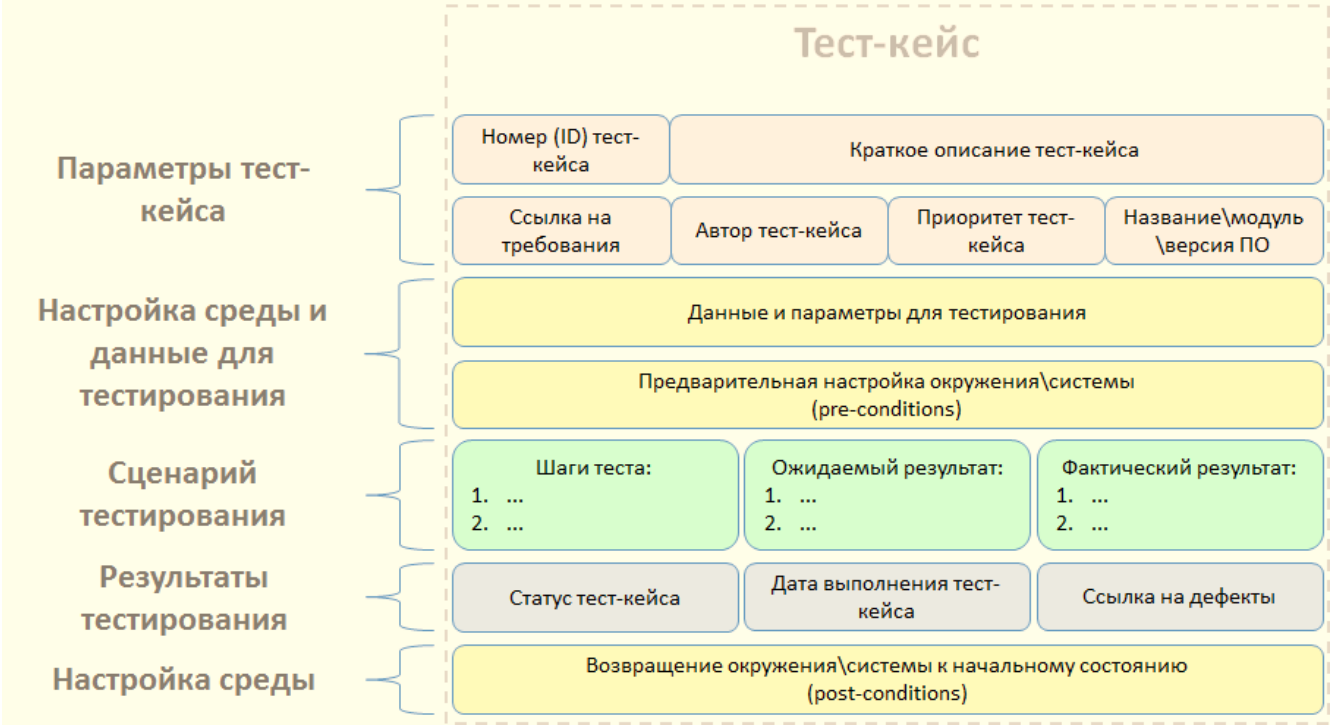


Настройка среды и данные для тестирования

- Данные и параметры для тестирования – исходные данные, необходимые для выполнения проверок.
- Предварительная настройка окружения\системы (pre-conditions) – подготовка необходимой аппаратной части и\или выполнение программных настроек.

Сценарий тестирования

- Шаги теста – кратко и четко описанное атомарное действие, необходимое для проверки.
- Ожидаемый результат – что ждем после этого действия.
- Фактический результат – что получаем в реальности (совпадает или нет с ожиданием).

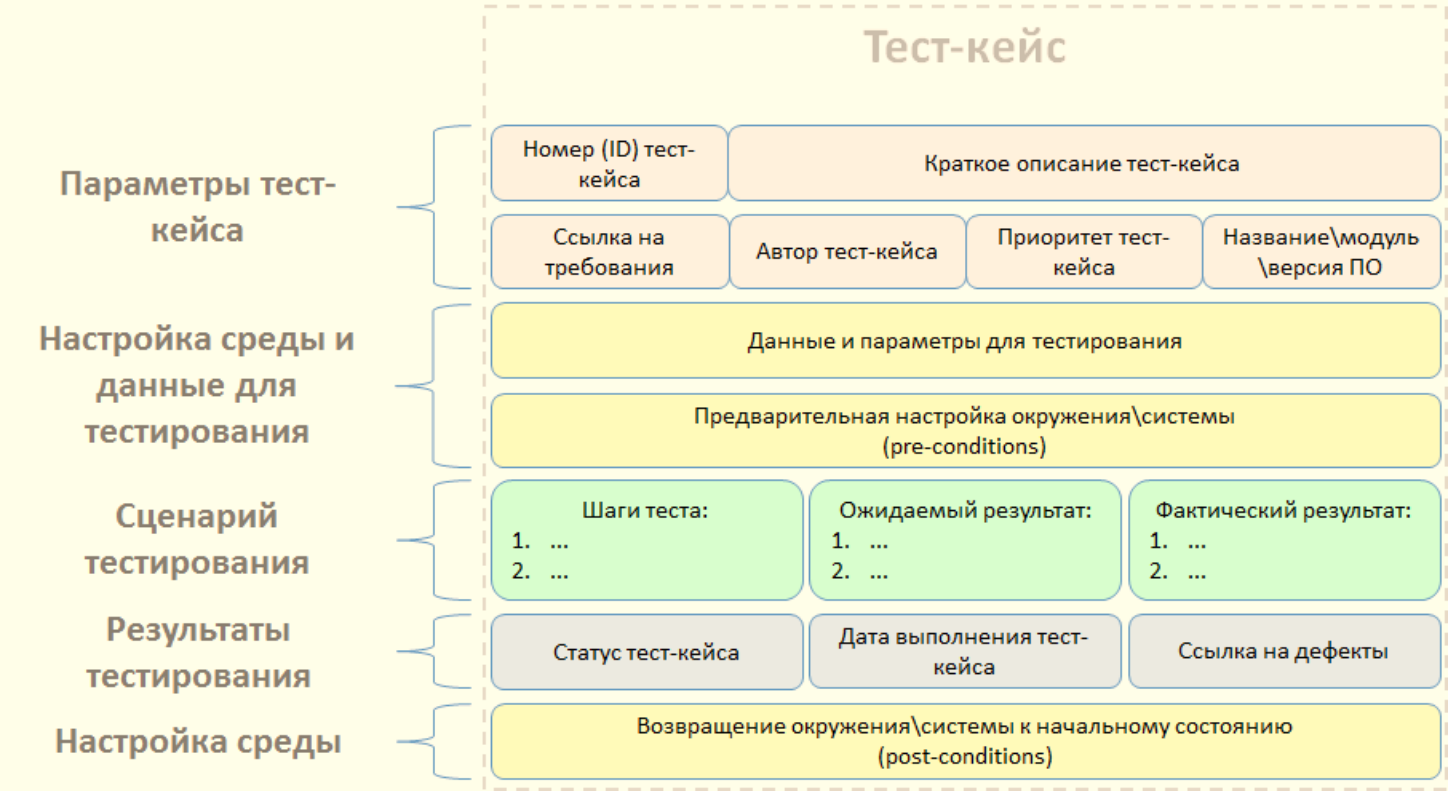


Результаты тестирования

- Статус тест-кейса – текущие состояние теста (например, «разработан», «отправлен в архив»).
- Дата выполнения тест-кейса – дата, когда тест проходили последний раз.
- Фактический результат - результат выполнения тест-кейса (например, «пройден», «заблокирован»).

Настройка среды

- Возвращение окружения\системы к начальному состоянию (post-conditions) – отмена всех сделанных ранее настроек.



Примеры тест-кейсов:

Примеры проверок для тестирования формы регистрации.

Цель настоящего документа - познакомить с основными проверками, проводимыми тестировщиками на примере простой формы регистрации. Стоит помнить, что это лишь базовый список проверок.

Пример формы регистрации.

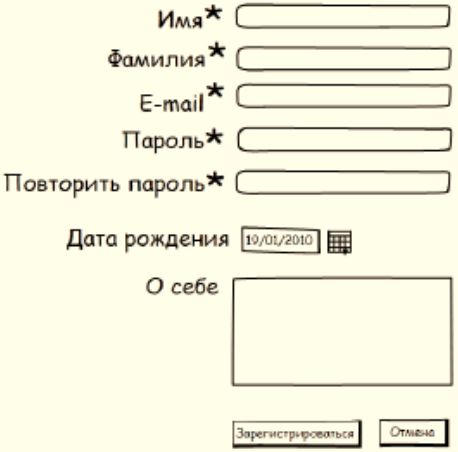
Проверка	Ожидаемый результат
Заполнить все поля корректной информацией и нажать кнопку 'Зарегистрироваться'.	1. Появляется сообщение об успешной регистрации с указанием дальнейших действий (необходимость перейти на форму логина/необходимость подтверждения регистрации). 2. Пользователь получает соответствующее уведомление на почтовый ящик с указанием дальнейших действий.
В уведомлении об успешной регистрации, полученной на почтовый ящик, перейти по предоставленной ссылке для подтверждения регистрации.	1. Пользователь перенаправлен на соответствующую форму с указанием успешной активации аккаунта. 2. Пользователь может войти в систему с указанными при регистрации данными.
Не заполнить обязательные поля (отмеченные звездочкой) и нажать кнопку 'Зарегистрироваться'.	Появляется сообщения об шибке для полей, отмеченых как обязательные для заполнения.
Ввести пробелы во все поля, отмеченные как обязательные, и нажать кнопку 'Зарегистрироваться'.	Появляется сообщения об шибке для полей, отмеченых как обязательные для заполнения.

Ввести пробелы до и после основного текста во все текстовые поля. К примеру, '_Имя_' и нажать кнопку 'Зарегистрироваться'. После подтверждения аккаунта войти в систему и перейти на форму редактирования личных данных.	Введенные пробелы до и после основного текста 'обрезаны'. Правильное отображение данных: 'Имя'.
Заполнить все поля корректной либо некорректной информацией и нажать кнопку 'Отмена'.	Введенные значения сбрасываются/пользователь перенаправляется на предыдущую форму.
Ввести в текстовые поля больше допустимого максимального количество символов и нажать кнопку 'Зарегистрироваться'.	Появляется сообщения об ошибке о том, что превышено максимально допустимое количество символов.
Ввести html tags/wildcards/специальные символы в текстовые поля и нажать кнопку 'Зарегистрироваться'. После подтверждения аккаунта войти в систему и перейти на форму просмотра личных данных (профиль пользователя). *Пример html тега: text* *Пример wildcards: &copy* *Пример специальных символов: !@#$%^&*()_+*	Информация отображается в том же виде, в котором она была введена на форме регистрации.
В поле E-mail ввести некорректный формат почтового ящика. Например, • не содержащий @/содержащий несколько символов @ • не содержащий домен верхнего уровня (.com/.org и др.) После ввода нажать кнопку 'Зарегистрироваться'.	Появляется сообщения об ошибке о некорректном формате E-mail адреса.
Предусловие: зарегистрировать пользователя. В поле E-mail ввести E-mail только что зарегистрированного пользователя.	Появляется сообщения об ошибке о том, что пользователь с указанным E-mail уже зарегистрирован в системе.
В поле 'О себе' ввести текст в несколько строк.	1. Информация введена и расположена в несколько строк. 2. Информация расположена в несколько строк в профиле пользователя.
В поле 'Дата рождения' выбрать дату в будущем. После заполнения остальных обязательных полей нажать кнопку 'Зарегистрироваться'.	Появляется сообщения об ошибке о том, что выбранная дата рождения некорректна.
В поле 'Дата рождения' ввести текст вручную. После заполнения остальных обязательных полей нажать кнопку 'Зарегистрироваться'.	Появляется сообщения об ошибке о том, что выбранная дата рождения некорректна.
В поле 'Пароль' и 'Повторить пароль' ввести разные данные. После заполнения остальных обязательных полей нажать кнопку 'Зарегистрироваться'.	Появляется сообщения об шибке о том, что данные в полях 'Пароль' и 'Повторить пароль' не соответствуют друг другу.

Примеры проверок для тестирования формы регистрации.

Цель настоящего документа - познакомить с основными проверками, проводимыми тестировщиками на примере простой формы регистрации. Стоит помнить, что это лишь базовый список проверок.

Пример формы регистрации.



Что такое спецификация программы?

- **Спецификация**, определение требований к программе — один из важнейших этапов, на котором подробно описывается исходная информация, формулируются требования к результату, поведение программы в особых случаях (например, при вводе неверных данных), разрабатываются диалоговые окна, обеспечивающие взаимодействие пользователя и программы.
- **Спецификация программы** — это средство для точного описания того, что должно быть совершено в результате выполнения программы.
- **Спецификация программы —** точная и полная формулировка задачи, содержащая информацию, необходимую для построения алгоритма (программы) решения этой задачи.

При разработке модуля необходимо ответить на вопросы:

- какие данные доступны каждому модулю при исполнении;
- в каких условиях можно выполнять данный модуль;
- какие действия выполняет модуль;
- как изменяются данные после завершения его работы.

Спецификация модуля:

- какие данные доступны каждому модулю при исполнении;
- в каких условиях можно выполнять данный модуль;
- какие действия выполняет модуль;
- как изменяются данные после завершения его работы;
- спецификация модуля не должна описывать метод решения задачи 

Спецификация включает в себя:

предусловия и

постусловия функции.

Примеры спецификаций (например, для метода сортировки массива в возрастающем порядке

//сортировка массива

//предусловие: переменная A является массивом

//состоящим из N целых чисел, N0

// постусловие: целые числа в массиве A упорядочены.

Достаточно ли этих пред- и постусловий?

Очевидно, что нет:

Во-первых, не указано в каком порядке необходимо упорядочить числа.

Во-вторых, не указано, насколько большим может быть число элементов в массиве N.

Пересмотренная спецификация: Sort(A,N)

//сортировка массива в возрастающем порядке

//предусловие: переменная A является массивом

//состоящим из N целых чисел,1

// где MAX_N – глобальная константа

//задающая максимальный размер массива A

// постусловие: целые числа в массиве A упорядочены

// по возрастанию: A[1]

// число N не изменилось

В предусловии описываются входные аргументы функции, указываются все глобальные именованные константы, использующиеся в ней, перечисляются все ограничения, которые накладываются функцией.

В постусловии описываются результаты работы функции (либо возвращаемое функцией значение) и все последствия ее работы.

Внешняя спецификация

Назначение: Программа сортировки чисел методом полного перебора.

Входные данные:

$a_1, a_2, \ldots a_n$ - неупорядоченный набор чисел [вещественные]

n - количество чисел [целое]

Выходные данные:

Получить на экране монитора результаты в следующем виде:

где:

$b_1, b_2, \ldots b_n$ - числа [вещественные]

Аномалии входных данных:

1. 1. n <= 0

Вывести сообщение “Ошибка” и завершить работу программы

2. 2. n > nmax {nmax – константа, будет определена в программе}

Вывести сообщение “Объем массива превышает <nmax>”.

Присвоить переменной n значение nmax и продолжить работу.

Сортировка чисел.

Алгоритм “Полный перебор”

Всего чисел = <n>

Исходный массив:

$<a_1> <a_2> \ldots <a_5>$

$<a_6> \ldots \ldots <a_{10}>$

. . .

$<a_n>$

Упорядоченный массив:

$<b_1> <b_2> \ldots <b_5>$

$<b_6> \ldots \ldots <b_{10}>$

. . .

$<b_n>$

Пример спецификации программы:

На вход программа принимает два параметра: x - число, n – степень. Результат вычисления выводится на консоль.

Значения числа и степени должны быть целыми.

Значения числа, возводимого в степень, должны лежать в диапазоне – [0..999].

Значения степени должны лежать в диапазоне – [1..100].

Если числа, подаваемые на вход, лежат за пределами указанных диапазонов, то должно выдаваться сообщение об ошибке.

Возьмём для демонстрации процесса тестирования программу, спецификация которой приведена ниже:

“*Назначение программы: сложить два введённых вами числа. В каждом из чисел должна быть одна или две цифры. Программа выполняет эхо-отображение вводимых чисел, а затем выводит их сумму. Ввод каждого числа завершается нажатием клавиши <Enter>. Запускается программа с помощью команды ADDER.*”

Задание к семинарскому заданию:

1. Составить спецификацию.
2. Привести примеры позитивных и негативных тестовых вариантов.
3. Привести примеры тест кейсов.
4. Привести примеры тестовых сценариев.

Путь развития в тестировании: http://svyatoslav.biz/wp-pics/testing_career_path.png

Объекты тестирования

Тестировать можно всё:

- работу программы;
- качество её кода и понятность комментариев;
- быстродействие;
- устойчивость под большой нагрузкой;
- расход ресурсов (памяти, диска, потери этих ресурсов);
- взаимодействие с другими программами;
- удобство интерфейса;
- документацию к программе (смысловые и грамматические ошибки, понятность и полноту);
- работу через сеть, работу аппаратного обеспечения и т.п.

Участники тестирования

Группа обеспечения качества ПО (Quality Assurance Team, QA team).

Специалист по контролю качества – участник проектной группы:

– осуществляет взаимодействие с разработчиками, менеджером программы и специалистами по безопасности и сертификации,

– отслеживает общее качество ПО,

– соответствие ПО стандартам и спецификациям.

Специалисты по тестированию (тестировщик) – участник команды разработчиков:

– определяет стратегию тестирования, тест-требования и тест-планы для каждого этапа проекта;

– выполняет тестирование системы, собирает и анализирует отчеты о прохождении тестирования.

Разработчики (программисты) – выполняют юнит-тестирование.

Принципы тестирования программного обеспечения. «Искусство тестирования» Г. Майерса

1. Обязательная часть тестирования – определение ожидаемого результата.
2. Программистам следует избегать тестирования их собственных программ (и участков кода).
3. Организациям, создающие программы, следует избегать тестирования их собственных программ.
4. Процесс тестирования должен включать в себя тщательную проверку результатов каждого теста.
5. Тест-кейсы должны быть составлены как для корректных и ожидаемых входных условий, так и для некорректных и неожидаемых.
6. Исследование Системы на предмет того, что она не делает того, что должна, — лишь пол дела. Вторая часть – разобраться в том, чего недолжного она делает.
7. Избегайте одноразовых тест-кейсов, только если сама программа не является одноразовой. Одноразовые тест-кейсы для одноразовых программ. В остальных случаях следует избегать таковых.
8. Не занимайтесь процессом тестирования с предустановкой, что вы не найдете ошибок.
9. Вероятность наличия ошибок в определенной части Системы пропорционально количеству уже найденных здесь ошибок.
10. Тестирование – это вызов вашим творческим и интеллектуальным способностям. Тестирование – это невероятно творческое и интеллектуальное занятие.

Документирование результатов тестирования

Баг или дефект репорт (Bug Report) (отчет об ошибке) — это документ, описывающий ситуацию или последовательность действий, приведшую к некорректной работе объекта тестирования, с указанием причин и ожидаемого результата.

Рассмотрим последовательно структуру отчета об ошибке, понятия "важность и приоритет дефекта", и кратко опишем процедуру написания такого отчета.

Структура Отчета об ошибке

Шапка

Короткое описание (Summary)	Короткое описание проблемы, явно указывающее на причину и тип ошибочной ситуации.
Проект (Project)	Название тестируемого проекта
Компонент приложения (Component)	Название части или функции тестируемого продукта
Номер версии (Version)	Версия, на которой была найдена ошибка

Важность (Severity)	Наиболее распространена пятиуровневая система градации важности дефекта: • S1 Блокирующий (Blocker) • S2 Критический (Critical) • S3 Значительный (Major) • S4 Незначительный (Minor) • S5 Тривиальный (Trivial)
Приоритет (Priority)	Приоритет дефекта: • P1 Высокий (High) • P2 Средний (Medium) • P3 Низкий (Low)
Статус (Status)	Статус бага. Зависит от используемой процедуры и жизненного цикла бага (bug workflow and life cycle)
Автор (Author)	Создатель баг репорта
Назначен на (Assigned To)	Имя сотрудника, назначенного на решение проблемы

Окружение

Версия ОС и т. д. / Версия браузера /	Информация об окружении, на котором был найден баг: операционная система, для WEB тестирования - имя и версия браузера и т. д.

Описание

Шаги воспроизведения (Steps to Reproduce)	Шаги, по которым можно легко воспроизвести ситуацию, приведшую к ошибке.
Фактический Результат (Result)	Результат, полученный после прохождения шагов к воспроизведению
Ожидаемый результат (Expected Result)	Ожидаемый правильный результат

Дополнения

Прикрепленный файл (Attachment)	Файл с логами, скриншот или любой другой документ, который может помочь прояснить причину ошибки или указать на способ решения проблемы

Важность и приоритет дефекта

Разные системы баг трекинга предлагают разные пути описания важности и приоритета баг репорта, неизменным остается лишь смысл, вкладываемый эти поля. Все знают такой баг-трекер, как Atlassian JIRA. В нем, начиная с какой-то версии вместо одновременного использования полей Severity и Priority, оставили только Priority, которое собрало в себе свойства обоих полей: Originally, JIRA did have both a Priority and a Severity field. The Severity field was removed for a number of reasons... Таким образом, те кто привык работать с JIRA не всегда понимают разницу между этими понятиями, так как не имели опыта их совместного использования.

Важность (Severity) — это атрибут, характеризующий влияние дефекта на работоспособность приложения.

Приоритет (Priority) — это атрибут, указывающий на очередность выполнения задачи или устранения дефекта. Можно сказать, что это инструмент менеджера по планированию работ. Чем выше приоритет, тем быстрее нужно исправить дефект.

Градация Важности дефекта (Severity)

S1 **Блокирующая** (Blocker)

Блокирующая ошибка, приводящая приложение в нерабочее состояние, в результате которого дальнейшая работа с тестируемой системой или ее ключевыми функциями становится невозможна. Решение проблемы необходимо для дальнейшего функционирования системы.

S2 **Критическая** (Critical)

Критическая ошибка, неправильно работающая ключевая бизнес-логика, дыра в системе безопасности, проблема, приведшая к временному падению сервера или приводящая в нерабочее состояние некоторую часть системы, без возможности решения проблемы, используя другие входные точки. Решение проблемы необходимо для дальнейшей работы с ключевыми функциями тестируемой системой.

S3 **Значительная** (Major)

Значительная ошибка, часть основной бизнес логики работает некорректно. Ошибка не критична или есть возможность для работы с тестируемой функцией, используя другие входные точки.

S4 **Незначительная** (Minor)

Незначительная ошибка, не нарушающая бизнес логику тестируемой части приложения, очевидная проблема пользовательского интерфейса.

S5**Тривиальная** (Trivial)

Тривиальная ошибка, не касающаяся бизнес-логики приложения, плохо воспроизводимая проблема, малозаметная по средствам пользовательского интерфейса, проблема сторонних библиотек или сервисов, проблема, не оказывающая никакого влияния на общее качество продукта.

Градация приоритета дефекта (Priority)

P1 **Высокий** (High)

Ошибка должна быть исправлена как можно быстрее, т.к. ее наличие является критической для проекта.

P2 **Средний** (Medium)

Ошибка должна быть исправлена, ее наличие не является критичной, но требует обязательного решения.

P3 **Низкий** (Low)

Ошибка должна быть исправлена, ее наличие не является критичной, и не требует срочного решения.

Порядок исправления ошибок по их приоритетам:

High -> Medium -> Low

Требования к количеству открытых багов

Наличие открытых дефектов P1 (высокий приоритет), P2 (средний приоритет) и S1 (блокирующая ошибка), S2 (критическая ошибка), считается неприемлемым для проекта. Все подобные ситуации требуют срочного решения и идут под контроль к менеджерам проекта.

Наличие строго ограниченного количества открытых ошибок P3 (низкий приоритет) и S3 (значительная ошибка), S4 (незначительная ошибка), S5 (тривиальная ошибка) не является критичным для проекта и допускается в выдаваемом приложении. Количество же открытых ошибок зависит от размера проекта и установленных критериев качества.

Все требования к открытым ошибкам оговариваются и документируются на этапе принятия решения о качестве разрабатываемого продукта. Как пример документирования подобных требований — это пункт Критерии окончания тестирования в плане тестирования.

Написание отчета об ошибке

Отчет об ошибке — это технический документ и в связи с этим язык описания проблемы должен быть техническим. Должна использоваться правильная терминология при использовании названий элементов пользовательского интерфейса (editbox, listbox, combobox, link, text area, button, menu, popup menu, title bar, system tray и т.д.), действий пользователя (click link, press the button, select menu item и т.д.) и полученных результатах (window is opened, error message is displayed, system crashedи т.д.).

Требования к обязательным полям баг репорта

Отметим, что обязательными полями баг репорта являются: короткое описание (Bug Summary), важность (Severity), шаги к воспроизведению (Steps to reproduce), результат (Actual Result), ожидаемый результат (Expected Result). Ниже приведены требования и примеры по заполнению этих полей.

Короткое описание

Название говорит само за себя. В одном предложение вам надо уместить смысл всего баг репорта, а именно: коротко и ясно, используя правильную терминологию сказать, что и где не работает.

Например:

Приложение зависает, при попытке сохранения текстового файла размером больше50Мб.

Данные на форме "Профайл" не сохраняются после нажатия кнопки "Сохранить".

Важность

Если проблема найдена в ключевой функциональности приложения и после ее возникновения приложение становится полностью недоступно, и дальнейшая работа с ним невозможна, то она блокирующая. Обычно все блокирующие проблемы находятся во время первичной проверки новой версии продукта (Build Verification Test, Smoke Test), так как их наличие не позволяет полноценно проводить тестирование. Если же тестирование может быть продолжено, то важность данного дефекта будет критическая. На счет значительных, незначительных и тривиальных ошибок вопрос достаточно прозрачный.

Шаги к воспроизведению/ Результат /Ожидаемый результат

Очень важно четко описать все шаги, с упоминаем всех вводимых данных (имени пользователя, данных для заполнения формы) и промежуточных результатов.

Например:

Шаги к воспроизведению

1. Войдите в системы:

Пользователь Тестер1, пароль xxxXXX

--> Вход в систему осуществлен

2. Кликните линк Профайл

--> Страница Профайл открылась

3. Введите Новое имя пользователя: Тестер2

4. Нажмите кнопку Сохранить

Результат

На экране появилась ошибка. Новое имя пользователя не было сохранено

Ожидаемый результат

Упрощённая классификация тестирования

Тестирование можно классифицировать по очень большому количеству признаков, и практически в каждой серьёзной книге о тестировании автор показывает свой (безусловно имеющий право на существование) взгляд на этот вопрос.

Соответствующий материал достаточно объёмен и сложен, а глубокое понимание каждого пункта в классификации требует определённого опыта, потому разделим данную тему на две: сейчас рассмотрим самый простой, минимальный набор информации, необходимый начинающему тестировщику, а в следующей лекции приведём подробную классификацию.

Используйте нижеприведённый список как очень краткую «шпаргалку для запоминания». Итак, тестирование можно классифицировать (смотри рисунок 1.3):

Рисунок 1.3 — Упрощённая классификация тестирования

- По запуску кода на исполнение:
  - **Статическое тестирование** — без запуска.
  - **Динамическое тестирование** — с запуском.
- По доступу к коду и архитектуре приложения:
  - **Метод белого ящика** — доступ к коду есть.
  - **Метод чёрного ящика** — доступа к коду нет.
  - **Метод серого ящика** — к части кода доступ есть, к части — нет.

- По степени автоматизации:
  - **Ручное тестирование** — тест-кейсы выполняет человек.
  - **Автоматизированное тестирование** — тест-кейсы частично или полностью выполняет специальное инструментальное средство.
- По уровню детализации приложения (по уровню тестирования):
  - **Модульное (компонентное) тестирование** — проверяются отдельные небольшие части приложения.
  - **Интеграционное тестирование** — проверяется взаимодействие между несколькими частями приложения.
  - **Системное тестирование** — приложение проверяется как единое целое.

- По (убыванию) степени важности тестируемых функций (по уровню функционального тестирования):
  - **Дымовое тестирование** — проверка самой важной, самой ключевой функциональности, неработоспособность которой делает бессмысленной саму идею использования приложения.
  - **Тестирование критического пути** — проверка функциональности, используемой типичными пользователями в типичной повседневной деятельности.
  - **Расширенное тестирование** — проверка всей (остальной) функциональности, заявленной в требованиях.
- По принципам работы с приложением:
  - **Позитивное тестирование** — все действия с приложением выполняются строго по инструкции без никаких недопустимых действий, некорректных данных и т.д. Можно образно сказать, что приложение исследуется в «тепличных условиях».
  - **Негативное тестирование** — в работе с приложением выполняются (некорректные) операции и используются данные, потенциально приводящие к ошибкам (классика жанра — деление на ноль).

Что такое тип тестирования?

В настоящее время нет общепринятого определения «типа тестирования программного обеспечения». Это не редкость, когда методы, уровни или даже тестовая дизайн техника определяются как тип тестирования. Например, иногда white box тестирование, интеграционное тестирование или даже тестирование граничный значений рассматриваются как типы тестирования.

Согласно определению Международной квалификационной коллегия по тестированию программного обеспечения ISTQB (International Software Testing Qualifications Board), которая предлагает стандарты в тестировании программного обеспечения, которые признаны во всем мире, типы тестирования являются **«средством четкого определения цели определенного уровня для программы или проекта»**. Тестер фокусируется на конкретной цели тестирования во время выполнения тестового случая. В зависимости от его целей существует четыре типа тестирования программного обеспечения:

1. Структурное тестирование.
2. Тестирование изменений.
3. Функциональное тестирование.
4. Нефункциональное тестирование.

Конечно, существует множество подтипов тестирования, которые можно увидеть на рисунке ниже:

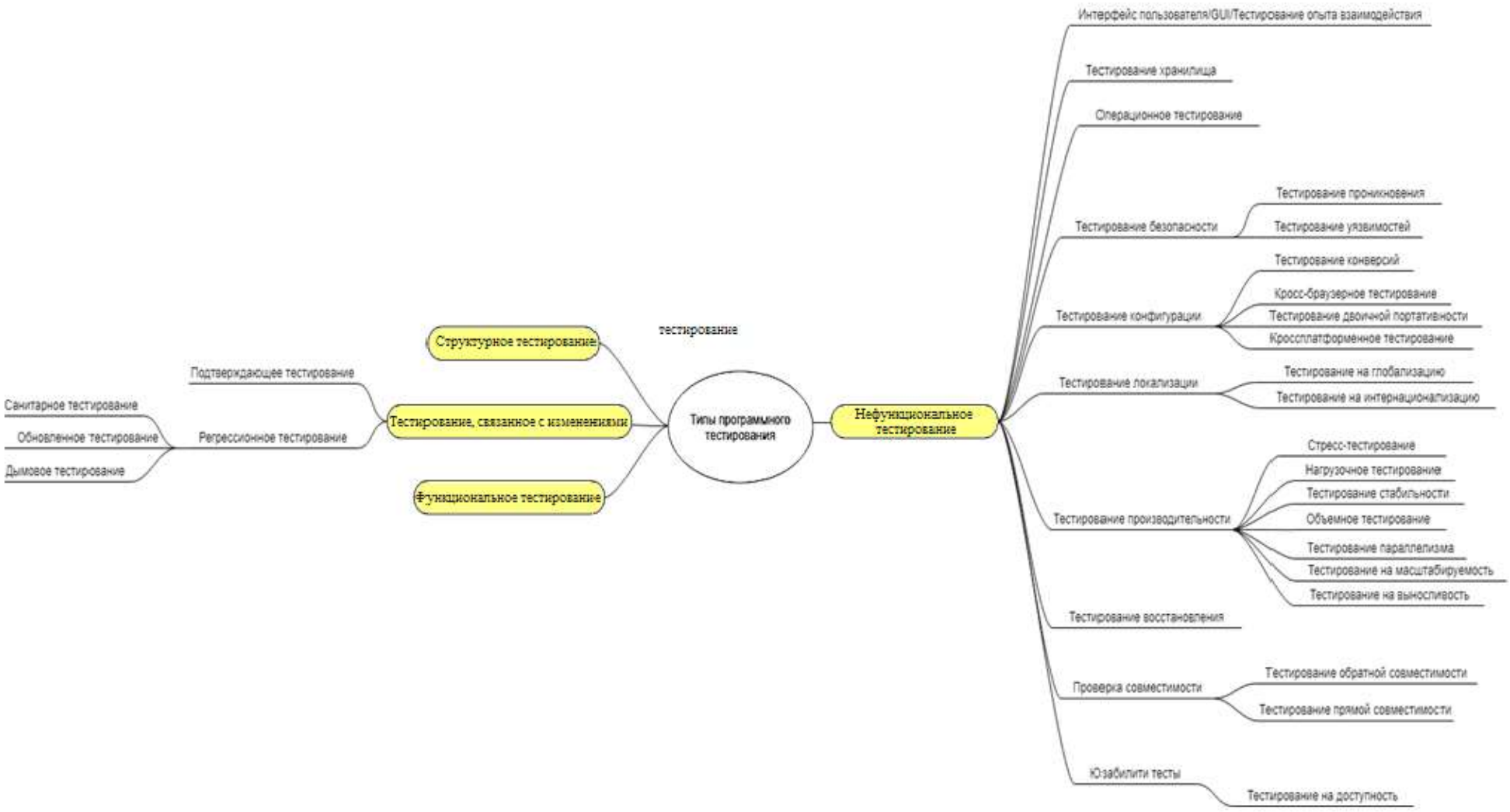


Рисунок 1.4 – Типы и подтипы программного обеспечения

Функциональное тестирование

Функциональное тестирование подтверждает, что каждая функция программного приложения работает в соответствии с требованиями спецификации. Функциональное тестирование показывает **«Что делает система»**. Цель этого тестирования - проверить, является ли система функционально совершенной.

В функциональном тестировании необходимо учитывать пять шагов:

- Подготовка тестовых данных на основе функций в спецификации.
- Бизнес-требования являются вкладом в функциональное тестирование.
- Вывод результатов работы на основе функциональных спецификаций.
- Выполнение тестовых случаев.
- Соответствие фактических и ожидаемых результатов.

Существуют две перспективы, в которых функциональность тестирования может быть выполнена на основе тестирования на **основе требований** и тестирование на **основе бизнес-процессов**.

Тестирование на основе требований выполняется в строгом соответствии с установленными требованиями.

Тестирование на основе бизнес-процесса выполняется в соответствии с знаниями, основанными на повседневном использовании системы в бизнесе.

Функциональное тестирование в основном включает тестирование «черного ящика» и не касается исходного кода приложения. Это тестирование проверяет:

- пользовательский интерфейс,
- API,
- базу данных,
- безопасность,
- связь клиент / сервер,
- другие функциональные возможности тестируемого приложения.

Тестирование может проводиться либо вручную, либо с использованием автоматизации.

Преимущества функционального тестирования:

- Функциональное тестирование имитирует фактическое использование системы.
- Оно выполняется в условиях, близких к клиентским.
- Легко провести ручное тестирование.

Ограничения функционального тестирования:

- Существует высокая вероятность избыточного тестирования.
- Логические ошибки в программном обеспечении могут быть пропущены при обеспечении функционального тестирования.

Легко найти и использовать инструменты для функционального тестирования. Наиболее известными из них являются: **Selenium** (как веб-приложения, так и настольные приложения), **Robotium** (приложение для Android), **Linux Test Project**, **JUnit**, **Sprinter by Hewlett Packard Entreprise** (ручное тестирование), **Browserstack** (как автоматическое, так и ручное тестирование), **Usersnap** (ручное тестирование).

В следующих лекциях будут рассмотрены следующие способы функционального тестирования:

– Способ разбиения по классам эквивалентности.

– Способ анализа граничных значений.

– Способ диаграмм причин-следствий.

Структурное тестирование

Структурное тестирование проверяет реализацию программы или кода посредством тестирования структуры программной системы или ее компонентов. Тестер концентрируется на работе программного обеспечения во время структурных испытаний. Он может использоваться на всех уровнях тестирования.

Основные **цели** структурного тестирования:

- Очевидная неадекватность идентификации.
- Функциональное тестирование.
- Чтобы понять, не хватает ли чего в нашем наборе тестов.

Преимущества структурного тестирования:

- Удаление мертвого кода.
- Существует возможность обнаруживать ошибки на ранней стадии.
- Это обеспечивает более тщательное тестирование программного обеспечения.
- Структурное тестирование не является трудоемким процессом.

Недостатки структурного тестирования:

- Структурное тестирование затратное.
- Оно требует знания кода.
- Оно требует глубокого знания инструмента, используемого для тестирования.

Методы структурного тестирования:

- **Способ тестирования базового пути** проверяет, что каждый оператор в программе выполняется хотя бы один раз во время тестирования программы.
- **Способ тестирование ветвей и операторов отношений** предназначен для проверки логических условий программы. При этом желательно обеспечить охват операторов из всех ветвей программы.
- **Способ тестирование потоков данных**, подвергается анализу информационная структура программы.
- **Тестирование циклов**, основное внимание обращается на правильность конструкций циклов.

Специальные инструменты для структурного тестирования: JBehave, Cucumber, JUnit, Cfix.

Тестирование изменений

Тестирование изменений предоставляется для обеспечения исправления ранее исправленных ошибок и устранения ошибок, которые могут быть случайно отображены в новой версии. В соответствии с этими целями существуют два подтипа тестирования, связанного с изменением:

- **тестирование подтверждения** (повторное тестирование),
- **регрессионное тестирование**.

Очень часто путают тестирование подтверждения с регрессионным тестированием. Как правило, их нужно использовать один за другим. Выясним разницу между этими подтипами тестирования.

Тестирование подтверждения выполняется, чтобы убедиться, что ошибка действительно успешно удалена. Проще говоря, тестовый сценарий, который первоначально обнаружил ошибку, выполняется снова, и на этот раз он должен пройти без проблем.

Повторное тестирование — проверяется исправление багов.

Регрессионное тестирование — это тип тестирования программного обеспечения, выполняемый для проверки того, не повлияло ли изменение кода на текущие функции и функции приложения.

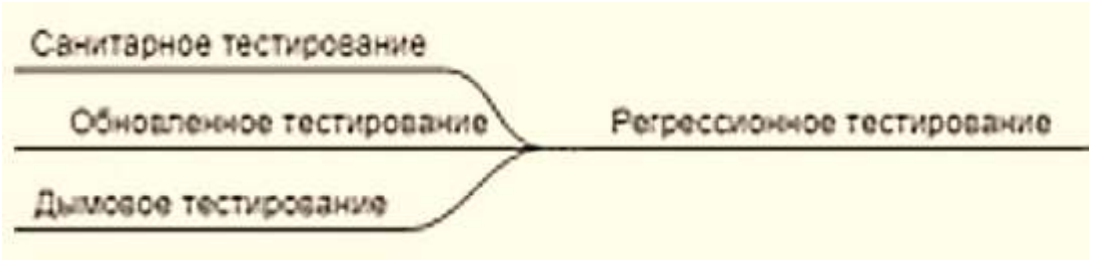
Регрессионное тестирование состоит не только из выявленных случаев проверки ошибок, но также для обеспечения того, чтобы новые дефекты не появились или не были обнаружены после изменений.

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

Сэм Канер описал 3 основных типа регрессионного тестирования:

– **Регрессия багов (Bug regression)** - попытка доказать, что исправленная ошибка на самом деле не исправлена.
– **Регрессия старых багов (Old bugs regression)** - попытка доказать, что недавнее изменение кода или данных сломало исправление старых ошибок, то есть старые баги стали снова воспроизводиться.
– **Регрессия побочного эффекта (Side effect regression)** - попытка доказать, что недавнее изменение кода или данных сломало другие части разрабатываемого приложения.

Инструменты для тестирования, связанные с изменением: Selenium, HP Quick Test Professional, TestComplete, TestDrive, SoapUI.



Санитарное тестирование (тестирование работоспособности) (Sanity testing) — это своего рода тестирование программного обеспечения, выполняемое после получения сборки программного обеспечения с незначительными изменениями в коде или функциональности, чтобы убедиться, что ошибки были исправлены и из-за этих изменений не возникло никаких дополнительных проблем. Цель состоит в том, чтобы определить, что предлагаемая функциональность работает примерно так, как ожидалось. Если тест на работоспособность не проходит, сборка отклоняется, чтобы сэкономить время и деньги, связанные с более строгим тестированием.

Целью является «не» тщательная проверка новой функциональности, а определение того, что разработчик применил некоторую рациональность (здравомыслие) при создании программного обеспечения. Например, если ваш научный калькулятор дает результат 2 + 2 = 5! Тогда нет смысла тестировать расширенные функции, такие как sin 30 + cos 50.

Санитарное тестирование — это узконаправленное тестирование достаточное для доказательства того, что конкретная функция работает согласно заявленным в спецификации требованиям. Является подмножеством **регрессионного тестирования.** Используется для определения работоспособности определенной части приложения после изменений, произведенных в ней или окружающей среде. Обычно выполняется вручную.

Понятие дымовое тестирование пошло из инженерной среды:

"При вводе в эксплуатацию нового оборудования ("железа") считалось, что тестирование прошло удачно, если из установки не пошел дым."

В области же программного обеспечения, *дымовое тестирование рассматривается как короткий цикл тестов, выполняемый для подтверждения того, что после сборки кода (нового или исправленного) устанавливаемое приложение, стартует и выполняет основные функции*.

Вывод о работоспособности основных функций делается на основании результатов поверхностного тестирования наиболее важных модулей приложения на предмет возможности выполнения требуемых задач и наличия быстро находимых критических и блокирующих дефектов. В случае отсутствия таковых дефектов дымовое тестирование объявляется пройденным, и приложение передается для проведения полного цикла тестирования, в противном случае, дымовое тестирование объявляется проваленным, и приложение уходит на доработку.

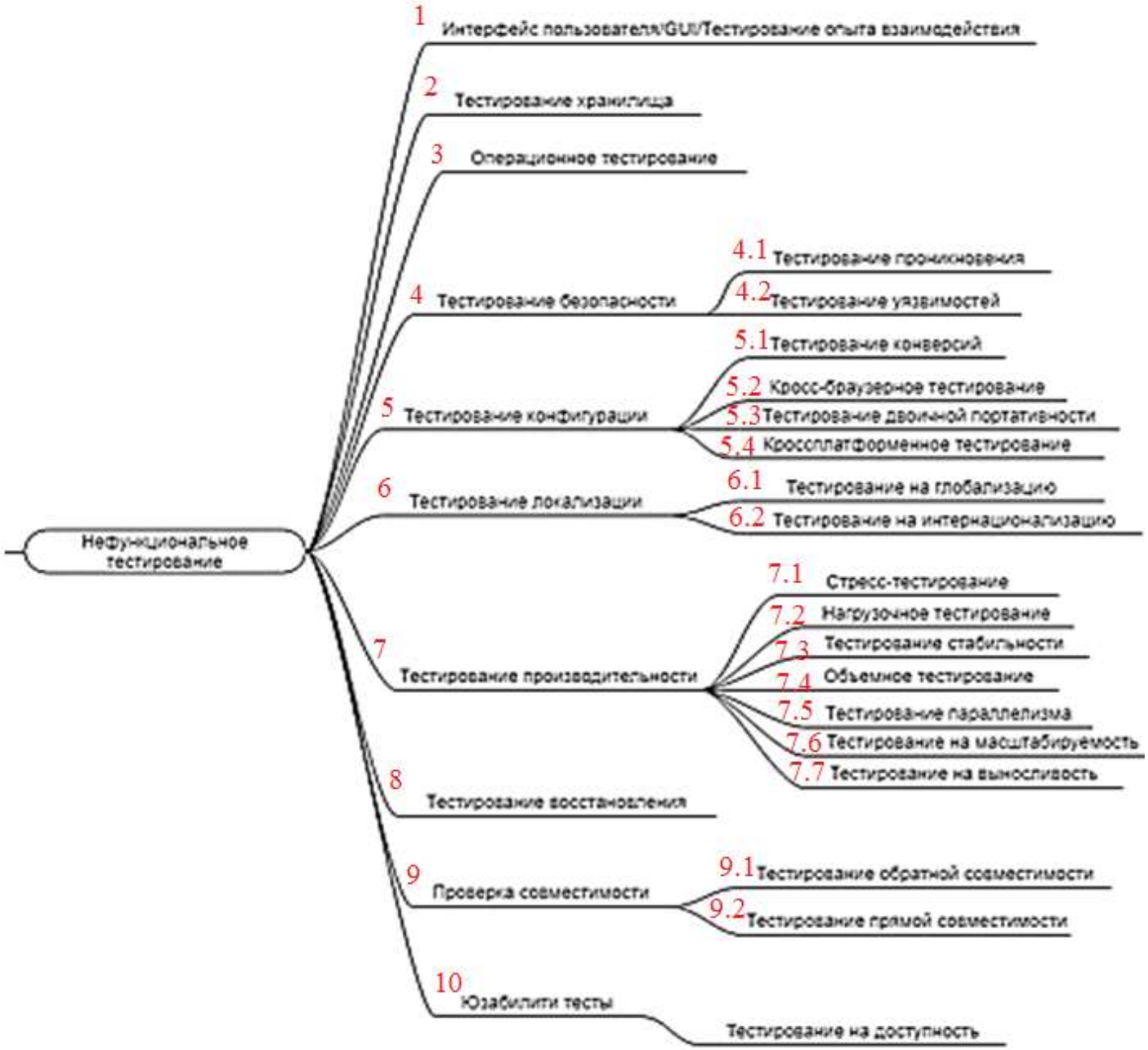
Аналогами дымового тестирования являются **Build Verification Testing** и **Acceptance Testing**, выполняемые на функциональном уровне командой тестирования, по результатам которых делается вывод о том, принимается или нет установленная версия программного обеспечения в тестирование, эксплуатацию или на поставку заказчику.

Для облегчения работы, экономии времени и людских ресурсов рекомендуется внедрить **автоматизацию тестовых сценариев** для дымового тестирования.

Отличие санитарного тестирования от дымового (Sanity vs Smoke testing)

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

Дымовые (Smoke)	Санитарные (Sanity)	Регрессионные (Regression)	Повторные (Re-test)
Исполняются с целью проверить что критически важные функциональные части системы работают как положено	Нацелено на установление факта того, что определённые части системы всё так же работают как положено после минорных изменений или исправлений багов	Подтверждают, что свежие изменения в коде или приложении в целом не оказали негативного влияния на уже существующую функциональность/набор функций	Перепроверяет и подтверждает факт того, что ранее заваленные тест-кейсы проходят после того, как дефекты исправлены
Цель — проверить «стабильность» системы в целом, чтобы дать зелёный свет проведению более тщательного тестирования	Целью является проверить общее состояние системы в деталях, чтобы приступить к более тщательному тестированию	Цель — убедиться, что свежие изменения в коде не оказали побочных эффектов на устоявшуюся работающую функциональность	Проверяет что дефект исправлен
Перепроверка дефектов не является целью Smoke	Перепроверка дефектов не является целью Sanity	Перепроверка дефектов не является целью Regression	Факт того, что дефект исправлен подтверждает Re-Test
выполняется перед регрессионным	выполняется перед регрессионным и после smoke-тестов	Проводится на основании требований проекта и доступности ресурсов (закрывается автотестами), «регресс» может проводиться в параллели с ре-тестами	—выполняется перед sanity-тестированием — Так же, приоритет выше регрессионных проверок, поэтому должно выполняться перед ними
Может выполняться автоматизировано или вручную	Чаще выполняется вручную	Желательна автоматизация, т.к. ручное может быть крайне затратным по ресурсам или времени	Не поддаётся автоматизации
Является подмножеством регрессионного тестирования	Подмножество приёмочного тестирования	Выполняется при любой модификации или изменениях в уже существующем проекте	проводится на исправленной сборке с использованием тех же данных, на том же окружении, но с различным набором входных данных
Тест-кейсы часть регрессионных тест-кейсов, но покрывающие крайне критичную функциональность	может выполняться без тест-кейсов, но знание тестируемой системы обязательно	могут быть получены из функциональных требований или спецификаций, пользовательских мануалов, и проводятся вне зависимости от того, что исправили разработчики	Используется тот же самый тест-кейс, который выявил дефект


Нефункциональное тестирование
1 Интерфейс пользователя/GUI/Тестирование опыта взаимодействия
2 Тестирование хранилища
3 Операционное тестирование
4 Тестирование безопасности
4.1 Тестирование проникновения
4.2 Тестирование уязвимостей
5 Тестирование конфигурации
5.1 Тестирование конверсий
5.2 Кросс-браузерное тестирование
5.3 Тестирование двоичной портативности
5.4 Кроссплатформенное тестирование
6 Тестирование локализации
6.1 Тестирование на глобализацию
6.2 Тестирование на интернационализацию
7 Тестирование производительности
7.1 Стресс-тестирование
7.2 Нагрузочное тестирование
7.3 Тестирование стабильности
7.4 Объемное тестирование
7.5 Тестирование параллелизма
7.6 Тестирование на масштабируемость
7.7 Тестирование на выносливость
8 Тестирование восстановления
9 Проверка совместимости
9.1 Тестирование обратной совместимости
9.2 Тестирование прямой совместимости
10 Юзабилити тесты
Тестирование на доступность

Нефункциональные типы тестирования

Нефункциональные типы тестирования связаны с нефункциональными требованиями. Нефункциональное тестирование помогает оценить готовность системы в соответствии с различными критериями, которые не охватываются функциональным тестированием. В отличие от функционального тестирования, он показывает «**Как хорошо работает система**».

Давайте рассмотрим вариации многотипных подтипов нефункционального тестирования.

1. **Тестирование пользовательского интерфейса** (User Interface - UI) предназначено для обеспечения соответствия графического пользовательского интерфейса приложения требованиям. Это помогает оценить элементы дизайна, такие как макет, цвета, шрифты, размеры шрифта, метки, текстовые поля, форматирование текста, титры, кнопки, списки, значки, ссылки и контент.

   Подходы к тестированию пользовательского интерфейса: руководство, запись и ответ, основанные на модели.

Вот некоторые проверки для **тестирования интерфейса веб-сайта**:

- Соответствие стандартам графических интерфейсов
- Оценка элементов дизайна: макет, цвета, шрифты, размеры шрифтов, ярлыки, текстовые поля, форматирование текста, титры, кнопки, списки, значки, ссылки
- Тестирование с различными разрешениями экрана
- Тестирование локализированных версий: точность перевода, проверка длины имен элементов интерфейса и т. п.
- Тестирование графического интерфейса пользователя на целевых устройствах: смартфоны и планшеты.

Самые популярные специальные инструменты и рамки для тестирования пользовательского интерфейса: **FitNesse**, **iMacros**, **Coded UI**, **Jubula**, **LoadUI**.

Тестирование опыта взаимодействия (User Experience - UX) нацелено на проверку взаимодействия с продуктами и услугами компании. Собственно, UX намного больше, чем визуальный интерфейс вашего продукта. Он содержит:

- Впечатления, которые клиент отвлекает от взаимодействия с продуктом или услугой.
- Процесс, который должен пройти клиент, чтобы узнать продукт или услугу компании.
- Последовательность действий клиента, когда он взаимодействует с интерфейсом.

Специальные инструменты и рамки для тестирования UX: **Usabilla**, **Omnigraffle**, **Visual Web Optimizer**, **UXPin**, **Crazy Egg**.

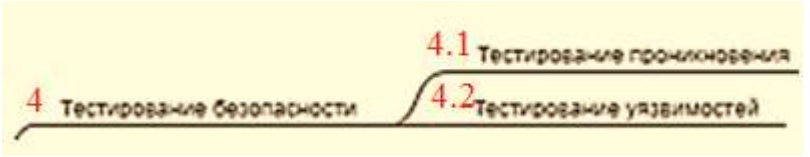
2. **Тестирование хранилища** проверяет тестируемое приложение, сохраняет соответствующие данные в правильных каталогах и имеет достаточное пространство для предотвращения неожиданного завершения из-за недостаточного дискового пространства. Это помогает определить, использует ли приложение больше памяти, чем предполагалось, поскольку заполнение дискового пространства может привести к значительным простоям.

   Инструменты часто используются для тестирования хранилища: **HCIbench**, **Iometer**, **Diskspd Utility**.

3. **Операционное тестирование** направлено на оценку системы или компонента в его операционной среде. Используя его, можем обеспечить соответствие системы и компонентов в стандартной операционной среде приложения.

   Это тестирование в основном фокусируется на оперативной готовности системы, которая, как предполагается, имитирует производственную среду.

4. **Тестирование безопасности** направлено на то, чтобы обеспечить информационную систему



защитой данных и поддерживать функциональность по назначению. Тестирование на проникновение и тестирование уязвимостей являются видами типов тестирования безопасности.

4.1. **Тестирование проникновения** — это симуляция атаки вредоносного источника, которое позволяет оценить безопасность компьютерной системы или сети.

4.2. **Тестирование уязвимостей** направлено на оценку кванта рисков, связанных с системой, чтобы уменьшить вероятность. Это помогает предотвратить проблемы, которые могут повлиять на целостность и стабильность приложения.

Некоторые проверки для **тестирования безопасности**:

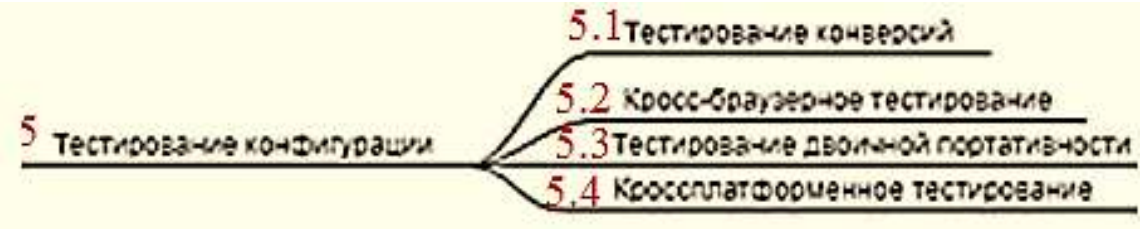
- Обеспечить невозможность несанкционированного доступа к защищенным страницам.
- Автоматическое прекращение проверки сеансов после длительного простоя пользователя.
- Тестирование функций безопасности SSL.
- Все попытки взлома, сообщения об ошибках и т. п. должны регистрироваться и сохраняться в отдельном файле для дальнейшего анализа.
- Проверьте работу captcha с помощью автоматических скриптов.
- Убедитесь, что файлы с ограниченным доступом не загружаются без соответствующего разрешения.
- Убедитесь, что при вводе неправильного пароля или имени пользователя нет возможности входа в систему.

Такие инструменты, как **Retina CS Community**, **OWASP Zed Attack Proxy**, **Veracode,** **Google Nogotofail,** и **SQL Map** часто используются для тестирования безопасности.

5. **Тестирование конфигурации** выполняется для проверки системы с каждой из поддерживаемых программных и аппаратных конфигураций:

- Конфигурация ОС - Win 7 32 bit/64 bit, Win 8 32 bit/64 bit, Win 7 32 bit/64 bit, Win 8 32 bit/64 bit.
- Конфигурация базы данных - Oracle, DB2, MySql, MSSQL Server, Sybase.
- Конфигурация браузера - IE 10, IE 11, Mozilla Firefox, Google Chrome.

Виды тестирования конфигурации:



- тестирование конверсий**,**
- кросс-браузерное тестирование**,**
- тестирование двоичной портативности,
- кроссплатформенное тестирование**.**

5.1.Тестирование конверсий обеспечивает правильное преобразование данных из существующих систем для использования в системах замещения.

5.2.Кросс-браузерное тестирование выполняется для проверки правильной работы приложения или системы в разных конфигурациях браузера: Mozilla Firefox, Google Chrome, Internet Explorer и Opera и т. Д.

5.3.Тестирование двоичной портативности позволяет оценить переносимость программного обеспечения, выполнив программное обеспечение на разных платформах и в среде. Он используется для подтверждения спецификации прикладного двоичного интерфейса (Application Binary Interface - ABI).

5.4.В отличие от кросс-браузерного тестирования, **кроссплатформенное тестирование** нацелено на оценку работы приложения в разных ОС: Windows, iOS / Mac OS, Linux, Android и BlackBerry и т. д.

Основные инструменты, которые часто используются для тестирования всех конфигураций: **BrowserStack,CrossBrowserTesting by Smart Bear**, **Litmus, Browsera**, **Rational Clearcase by IBM**, **Ghostlab**.

Что такое кроссплатформенность?

Прежде, чем приступить к рассмотрению кроссплатформенного тестирования, давайте разберемся с понятием кроссплатформенности в целом.

Кроссплатформенность или межплатформенность – это способность программного обеспечения работать с несколькими платформами или операционными системами. Кроссплатформенность достигается с помощью высокоуровневых языков программирования, сред разработки и выполнения, поддерживающих условную компиляцию, компоновку и выполнение кода для различных платформ.

Самым распространенным и простым примером кроссплатформенного приложения является веб-браузер. Они отображают веб-сайты практически одинаково, вне зависимости от того, на какой платформе или операционной системе их запустили.

Кроссплатформенность будет реализована по-разному в зависимости от того какое приложение у нас – традиционное или веб-приложение.

Обычно, все веб-приложения позиционируются как кроссплатформенные из-за того, что доступны из любого веб-браузера и в разных ОС. Веб-приложения используют клиент-серверную архитектуру, но могут существенно отличаться по функционалу и сложности.

Простые веб-приложения обрабатывают данные с сервера без сохранения состояния и направляют результат клиенту. То есть, взаимодействие пользователя с приложением осуществляется путем обмена запросами данных и ответами сервера.

Примерами более сложных веб-приложений могут быть:

– веб-интерфейс Gmail;

– веб-сайт Google Maps;

– служба Bing от Microsoft.

Такие расширенные приложения обычно зависят от дополнительных функций, которые можно найти только в более свежих версиях популярных веб-браузеров. Эти зависимости включают Ajax, JavaScript, Dynamic HTML, SVG и другие компоненты многофункциональных интернет-приложений. Старые версии популярных веб-браузеров, как правило, не поддерживают некоторые функции.

Но, хотя популярность веб-приложений растет, многие продолжают пользоваться традиционными приложениями, не основанными на клиент-серверной архитектуре.

Такое программное обеспечение распространяется в виде двоичных файлов, которые поддерживают только ОС и компьютерную архитектуру, для которой они были построены. Поэтому, если приложение должно быть кроссплатформенным, для него создают несколько исполнительных установочных файлов – для каждой поддерживаемой платформы отдельно.

Но иногда, с помощью различных инструментов, нельзя создать сборки для всех поддерживаемых платформ. В этом случае разработчик ПО должен изменить код таким образом, чтобы он подходил для новой архитектуры компьютера или операционной системы.

Понятие и важность кроссплатформенного тестирования

Кроссплатформенное тестирование – это процесс проверки работы приложения или сайта на различных платформах.

Кроссплатформенное тестирование является очень важным этапом обеспечения качества ПО. Когда продукт разрабатывается для нескольких платформ, важно выделить все наиболее популярные и протестировать работу приложения на них. Для веб-приложений можно просто выбрать наиболее популярные версии браузеров и проводить тестирование на них. Для декстопных приложений количество комбинаций ОС/конфигураций системы слишком велико, поэтому необходимо проводить более глубокий анализ аудитории и выделять наиболее важные комбинации для тестирования.

Кроссплатформенное тестирование проводится для определения поведения приложения и веб-сайта в различных средах. Межплатформенное тестирование помогает в выявлении проблем, которые могут различаться в зависимости от платформы или конфигурации, таких как согласованность, пользовательский интерфейс, проблемы с юзабилити и производительность.

В приложении, которое предполагается запускать на нескольких платформах, важно, чтобы интерфейс разработки был согласованным. Дизайн должен быть разработан таким образом, чтобы не вызывать проблем с выравниванием или ненужным переносом текста в пользовательском интерфейсе.

Если приложение использует много данных, это может вызвать проблемы для устройств с низким кэшем и оперативной памятью. Также важно учитывать удобство использования устройства. Например, пользователи телефонов Android предпочитают приложения небольшого размера, которые не занимают много места на своих устройствах.

GUI со слишком большим количеством кнопок может быть сложным и недружественным для конечного пользователя. Очень важно знать, как конечные пользователи будут взаимодействовать с системой и какой тип ввода они предпочтут использовать. Например, в финансовых приложениях предпочтительной комбинацией ввода являются текстовые поля, но, если пользователю предоставляются выпадающие списки и кнопки, это вызовет трудности.

Подобные проблемы вызывают у пользователей плохое впечатление и интерес быстро теряется.

Особенности, методы кроссплатформенного тестирования

Ниже приведен краткий список советов и методов для проведения кроссплатформенного тестирования:

Определить границу

Во избежание путаницы при тестировании важно определить четкие границы для кроссплатформенного тестирования. Выделить наиболее важные платформы, которые будут использовать реальные пользователи.

Тестовая матрица

Матрица всегда полезна, когда речь идет о разработке тестовых сценариев. Подготовьте тестовую матрицу и используйте ее для выполнения тестовых случаев и для связи с конечными пользователями.

Тепловые карты

Выделите элементы, которые имеют высокий риск в тестовой матрице. Тепловая карта может быть полезна для идеального определения зон риска. Это помогает в правильной расстановке приоритетов, когда время и ресурсы ограничены.

Многомерная матрица вместе с тепловой картой будут полезны для определения того, что необходимо протестировать в первую очередь, а что может подождать. Желтые клетки в таблице – то, что нужно протестировать при наличии достаточного количества времени, а красные – «условия тестирования высокого риска», то, что нужно протестировать в первую очередь.

	OS		Safari	Chrome	IE	Firefox	Android native
Devices	Android	Samsung Galaxy J7					
		Samsung Galaxy S9					
		Google Pixel					
		Samsung Galaxy Tab A					
		Nexus 7					
	iOS	iPhone 6S					
		iPhone 7					
		iPhone X					
		iPad Air 2					
		iPad 4 Wi-Fi					
	Windows	Nokia Lumia 930					

Наиболее популярные ОС, которые используются для кроссплатформенных приложений

Самыми распространенным операционными системами являются Windows, Android и iOS.

Android – операционная система, которая может использоваться на различных девайсах: смартфонах, планшетах, электронных книгах, цифровых проигрывателях, наручных часах, фитнес-браслетах, игровых приставках, ноутбуках, нетбуках, смартбуках, очках Google Glass, телевизорах и других устройствах.

Это наиболее распространенная операционная система для мобильных устройств в данный момент. При планировании кросс-браузерного тестирования необходимо учитывать отличия между версиями операционной системы и их распространенность.

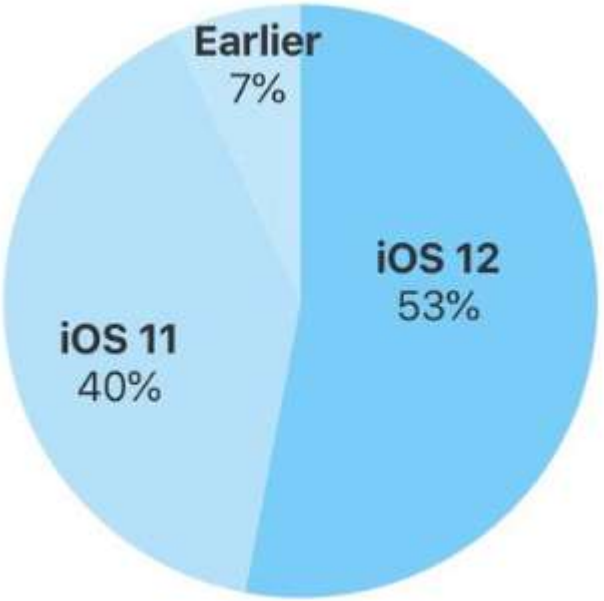
Ниже размещена таблица популярности версий Android OS в 2019 году:

Версия системы	Текущее распределение
2.3 Gingerbread	0,3%
4.0 Ice Cream Sandwich	0,3%
4.1 4.2 4.3 Jelly Bean	3,2%
4.4 Kitkat	6,9%
5.0 Lollipop	3%
5.1 Lollipop	11,5%.
6.0 Marshmallow	16,9%
7.0 Nougat	11,4%
7.1 Nougat	7,8%
8.0 Oreo	12,9%
8.1 Oreo	15,4%
9.0 Pie	10,4%

iOS (до 24 июня 2010 года – iPhone OS) – операционная система, которая также может использоваться на различных девайсах: смартфонах, электронных планшетах, носимых проигрывателях и некоторых других устройствах. Эта операционная система разрабатывается и выпускается американской компанией Apple. Была выпущена в 2007 году; изначально – для iPhone и iPod touch, позже – для iPad. В 2014 году появилась поддержка автомобильных мультимедийных систем Apple CarPlay. В отличие от Android (Google), выпускается только для устройств, производимых фирмой Apple.

53% устройств, представленных в последние 4 года, используют iOS 12. Статистика популярности версий iOS:

По данным App Store на 10 октября 2018 года, на устройствах, представленных с сентября 2014 года.



Windows – семейство коммерческих операционных систем корпорации Microsoft, которые ориентированы на управление с помощью графического интерфейса. Windows была установлена не менее чем на 88,5% персональных компьютеров и рабочих станций, согласно данным за июнь 2019 года. По данным компании Net Applications, на июнь 2019 года рыночная доля Windows составила 88,33 %. Падение доли связано, в первую очередь, с тенденцией к сокращению продаж ПК в мире, а также с увеличением популярности конкурентов – macOS и Linux. Самой популярной версией Windows по данным W3Schools с июля 2017 года считается Windows 10 (около 37 %).

При планировании тестирования на Windows необходимо учитывать особенности различных версий операционной системы, а также взаимодействия их с различными браузерами при кросс-браузерном тестировании.

Инструменты для проведения кроссплатформенного тестирования

Инструменты всегда полезны при любом типе тестирования. Особенно, когда необходимо проверить один и тот же функционал на различных платформах, на помощь придут инструменты для автоматизированного тестирования и эмуляции выполнения программы на различных платформах.

Некоторые из распространенных инструментов для кроссплатформенного тестирования:

1. Appium

Доступен как на iOS, так и на Android. Это HTTP-сервер, управляющий сеансами WebDriver. Он поддерживает тестирование в любой среде и на любом языке, который может создать запрос HTTP. Это инструмент с открытым исходным кодом для автоматизации нативных, мобильных и веб-приложений, а также гибридных приложений на платформах iOS и Android.

2. MonkeyTalk

Также работает с Android и iOS, это инструмент с открытым исходным кодом, состоящий из трех компонентов, таких как IDE, сценарии и агенты. MonkeyTalk автоматизирует функциональные интерактивные тесты для приложений Android и iOS. Скрипты MonkeyTalk используют простой синтаксис ключевых слов и механизмы исполнения Ant of Java. Тесты могут быть обработаны данными из электронной таблицы в формате CVS.

3. Katalon

Katalon – это бесплатный инструмент для мобильного тестирования, веб-тестирования и автоматизации, который совместим с последними версиями iOS и Android. Katalon предлагает полнофункциональное управление тестами, интерфейсы с двумя сценариями для начинающих и опытных пользователей. Выполнение сценариев на эмуляторах, реальных устройствах или облачное тестирование. Встроенные ключевые слова поддерживают GUI, API и тестирование данных, а также имеют встроенную интеграцию с JIRA, GIT и Kobiton.

4. EggPLANT

EggPLANT – это коммерческий продукт для автоматизированного тестирования с графическим интерфейсом, разработанный TestPlant для iOS и Android, а также для тестирования веб-приложений. Полезен для автоматизации пользовательского интерфейса и функциональности, тестирования на основе изображений, мобильного тестирования, сетевого тестирования, веб-тестирования и кросс-браузерного тестирования. Можно использовать один скрипт для всех устройств и платформ, а также создавать дополнительные для конкретного устройства/платформы.

5. Browsershots

Широко используемый инструмент кросс-браузерного тестирования из-за его функций и доступных настроек. Он поддерживает IE, Firefox, Google Chrome, Opera, Safari, Minefield, Netscape и многие другие со всеми их версиями. Вы можете выбрать размер экрана, глубину цвета и параметр, позволяющий проверить состояние JavaScript, а также варианты включения или отключения Flash. Одна из проблем здесь заключается в том, что для получения результатов требуется много времени, и нет мобильных браузеров.

6. BrowserStack

Обеспечивает тестирование веб-браузера в режиме реального времени с мгновенным доступом ко всем настольным и мобильным устройствам. Нет необходимости устанавливать приложение, поскольку оно основано на облаке. Для отладки и кросс-браузерного тестирования используются предустановленные инструменты разработчика.

7. KeepItFunctional (KIF)

Платформа с открытым исходным кодом, разработанная для приложений iOS, предназначенная для тестирования пользовательского интерфейса мобильного приложения и простого автоматизированного тестирования. Это интегрированная среда тестирования iOS, используемая для функционального тестирования, которая создает и выполняет контрольные тесты.

8. Xamarin

Xamarin имеет инструменты тестирования для кроссплатформенных мобильных приложений на iOS, Android, Blackberry и Windows. У Xamarin есть собственный инструмент разработки интерфейса и программа Xamarin University, где вы можете пройти онлайн-курсы; он позволяет запускать облачный автоматический приемочный тест пользовательского интерфейса.

9. Browserling

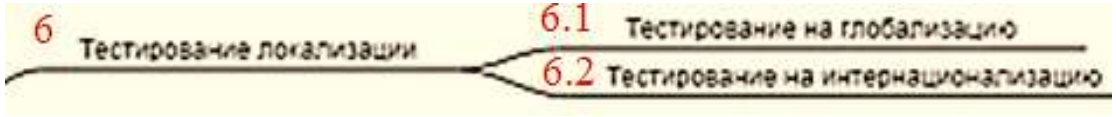
Запущенный в 2010 году Browserling — это интерактивный кросс браузерный инструмент для тестирования в режиме реального времени. Поддерживает тестирование на большинстве версий Internet Explorer, Chrome, Opera, Safari и Firefox. Поддерживаются операционные системы Windows XP, Windows Vista и Windows 8.1, а также Android Mobile.

10. Litmus

Известный инструмент для тестирования почтовых ящиков, Litmus можно использовать и для тестирования веб-приложений. Litmus позволяет видеть, как страница отображается в известных веб-браузерах на мобильных и настольных платформах. Поддерживает браузеры Firefox, Chrome, Safari и мобильные версии, а также операционные системы Windows, iOS и Android.

Кроссплатформенное тестирование – важный вид тестирования, который необходимо проводить для понимания того, будет ли должным образом отображаться тестируемый продукт на различных платформах, используемых целевой аудиторией. Зачастую, для проведения этого вида тестирования, используются инструменты автоматизированного тестирования, которые доступны в большом количестве в магазинах приложений.

6. **Тестирование локализации** выполняется для адаптации глобализированного приложения к определенной культуре / языку. Этот процесс включает перевод всех строк родного языка на целевой язык и настройку графического интерфейса, чтобы он соответствовал целевому рынку. **Тестирование на глобализацию** и **тестирование на интернационализацию** являются одними из его видов.



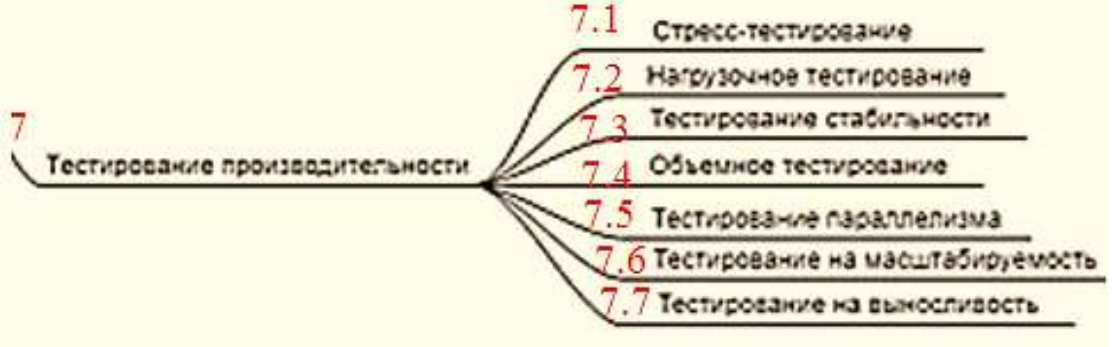
6.1. Проверка на предмет **глобализации** проверяет правильную функциональность продукта с любыми настройками культуры / локализации с использованием любого типа международного ввода.

6.2. Тестирование на **интернационализацию** проверяет правильность содержимого контента на разных языках и местах.

Локализация обычно осуществляется с использованием некоторой комбинации внутренних ресурсов, независимых подрядчиков и полномасштабных услуг компании локализации. Вот некоторые инструменты для обеспечения тестирования локализации – **eggPlant**, **Babylon.NET by Redpin**, и **smartCAT.**

7. **Тестирование производительности** (Performance testing) намеревается определить, как система работает с точки зрения быстроты реагирования и стабильности при определенной нагрузке.

Виды тестирования производительности:



7.1. стресс-тестирование (Stress testing),

7.2. нагрузочное тестирование (Load testing),

7.3. тестирование стабильности (Stability testing),

7.4. объемное тестирование (Volume testing),

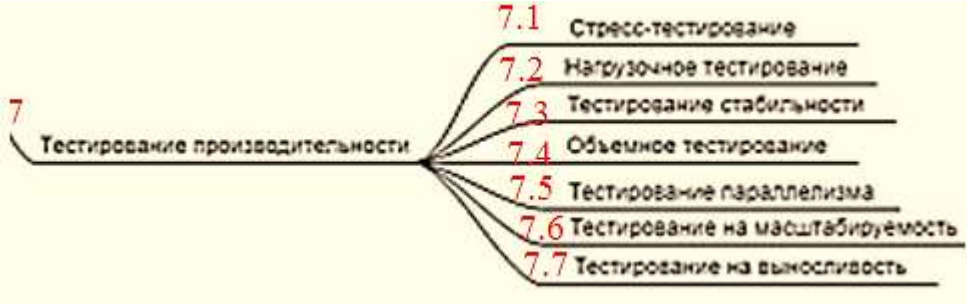
7.5. тестирование параллелизма (Concurrency testing),

7.6. тестирование на масштабируемость (Scalability testing),

7.7 тестирование на выносливость (Endurance testing) и т. д.

7.1. Стресс-тестирование оценивает поведение системы в пределах ее ожидаемой рабочей нагрузки или выходит за ее пределы.

7.2. Проведено **тестирование нагрузки** для оценки поведения системы при увеличении рабочей нагрузки.

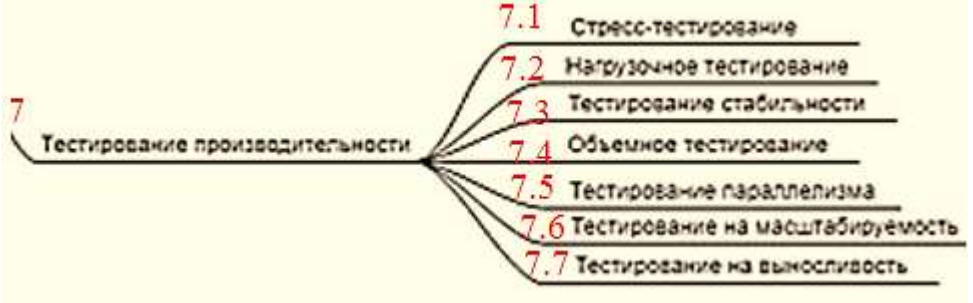


7.3. Тестирование на стабильность направлено на то, чтобы проверить, может ли приложение непрерывно работать в пределах или чуть выше допустимого периода.

7.4. Тестирование объема позволяет анализировать производительность системы за счет увеличения объема данных в базе данных. Оно проверяет, что любые значения могут стать большими с течением времени (например, накопленные счета, журналы и файлы данных), могут быть адаптированы программой и не заставят программу прекратить работу или ухудшить ее работу.

7.5. Тестирование параллелизма выполняется для выявления дефектов в приложении, когда несколько пользователей входят в приложение. Используя его, мы можем идентифицировать и измерить проблемы в системных параметрах, такие как время отклика, пропускная способность, блокировки / взаимоблокировки или любые другие проблемы, связанные с параллелизмом.

7.6. Масштабируемость оценивает способность системы расти, увеличивая различные показатели, такие как рабочая нагрузка на каждого пользователя или количество одновременных пользователей или размер базы данных.



7.7 Испытание на выносливость определяет проблемы, которые могут возникать при длительном выполнении. Он оценивает поведение системы, когда значительная рабочая нагрузка предоставляется непрерывно.

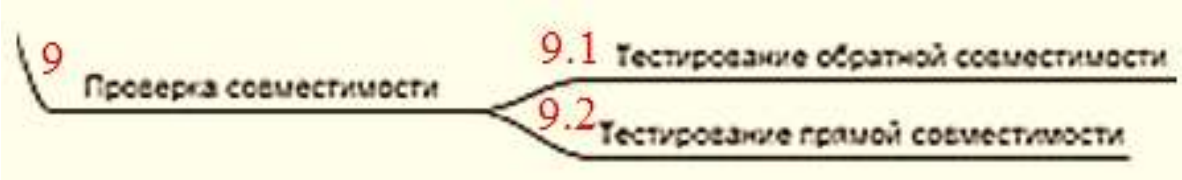
Тестирование Ramp — это испытание на выносливость, которое заключается в непрерывном повышении входного сигнала до тех пор, пока система не сломается.

Apache JMeter, **HP LoadRunner**, **Silk Performer from Micro Focus**, **WebLOAD**, и **Gatling** часто используются для выполнения различных видов тестирования производительности.

8. **Тестирование восстановления** предназначено для оценки способности системы восстанавливаться после сбоев, сбоев оборудования или других катастрофических проблем. Он выполняется группами тестирования.

TestDisk, Recuva by Piriform, Wise Data Recovery by WiseCleaner и Restoration by Softonic являются основными специальными инструментами для тестирования восстановления.

9. **Проверка совместимости** проверяет совместимость приложения в разных средах: аппаратное обеспечение, программное обеспечение, операционная система, сетевая среда.



Существует два типа такого типа тестирования:

- **обратная совместимость,**
- **тестирование прямой совместимости**.

9.1. Тестирование обратной совместимости гарантирует, что новая версия продукта продолжит работу с более старым продуктом.

9.2. Тестирование прямой совместимости обеспечивает подключение к будущей версии продукта.

Для тестирования совместимости могут использоваться бесплатно Browsershots и **MultiBrowser**

10. **Юзабилити тесты**, выполненные для оценки продукта или услуги путем тестирования его с репрезентативными пользователями. Это помогает определить способность пользователя учиться работать, готовить входные данные и интерпретировать выходы системы или компонента.

Тестирование на доступность — это **тип тестирования юзабилити**, который определяет удобный для пользователя уровень продукта, для людей с ограниченными возможностями (глухих, слепых, умственно отключенных).

Популярные инструменты для тестирования юзабилити: **User Zoom**, **Reflector,** **Loop**.

Требования к требованиям:

• Корректность.

• Недвусмысленность.

• Полнота набора требований.

• Непротиворечивость набора требований.

• Проверяемость (тестопригодность).

• Трассируемость.

• Понимаемость.

В книге Сэм Канера, Джека Фолка и др. Тестирование программного обеспечения приведены распространенные программные ошибки (около 400 ошибок, 13 категорий ошибок).

1 категория: **Ошибки пользовательского интерфейса**

Понятие пользовательского интерфейса объединяет все аспекты продукта, с которыми имеет дело его пользователь. Разработчики пользовательского интерфейса пытаются достичь компромисса между такими факторами, как.

- функциональность;
- быстрота изучения программы новым пользователем;
- легкость запоминания необходимых приемов работы;
- производительность;
- вероятность возникновения ошибок пользователя;
- удовлетворенность пользователя программой.

Например, рассмотрим **функциональность**:

- избыточная функциональность;
- пропущенная функция;
- неверно работающая функция;
- программа не делает того, что ожидает от нее пользователь;
- пропущенная информация;
- неверная или смущающая пользователя информация:
  - простая фактическая ошибка;
  - синтаксические ошибки;
  - неаккуратное упрощение;
  - неточные названия команд и функций;
  - несколько названий одной функции;
  - избыточность информации;
- справочная система и сообщения об ошибках:
  - *высокий уровень сложности* (максимально допустимым уровнем сложности текста на экране является уровень 5);
  - *многословность;*
  - *неуместная эмоциональность;*

…

- ошибки отображения:
  - два *курсора;*
  - *исчезновение курсора;*
  - курсор *отображается не в том месте;*
  - курсор вне *области ввода данных;*
  - *отображение ввода не в том месте экрана;*
  - *неочищенная часть экрана;*
  - не *выделены активные элементы экрана;*
  - *отображена неверная или неполная строка;*
  - *сообщение остается на экране слишком* долго или исчезает *слишком быстро;*
- организация экрана:
  - *неэстетическое оформление экрана;*
  - *неудачная организация меню;*
  - *труднонаходимые инструкции;*
  - *неуместное использование* мигания;
  - *пестрые цветовые сочетания;*
- потери времени

Программа не должна тратить зря ни одной секунды времени пользователя.

Программирование «с защитой от ошибок»

Любая из ошибок программирования, которая не обнаруживается на этапах компиляции и компоновки программы, в конечном счете может проявиться тремя способами: привести к выдаче системного сообщения об ошибке, «зависанию» компьютера и получению неверных результатов.

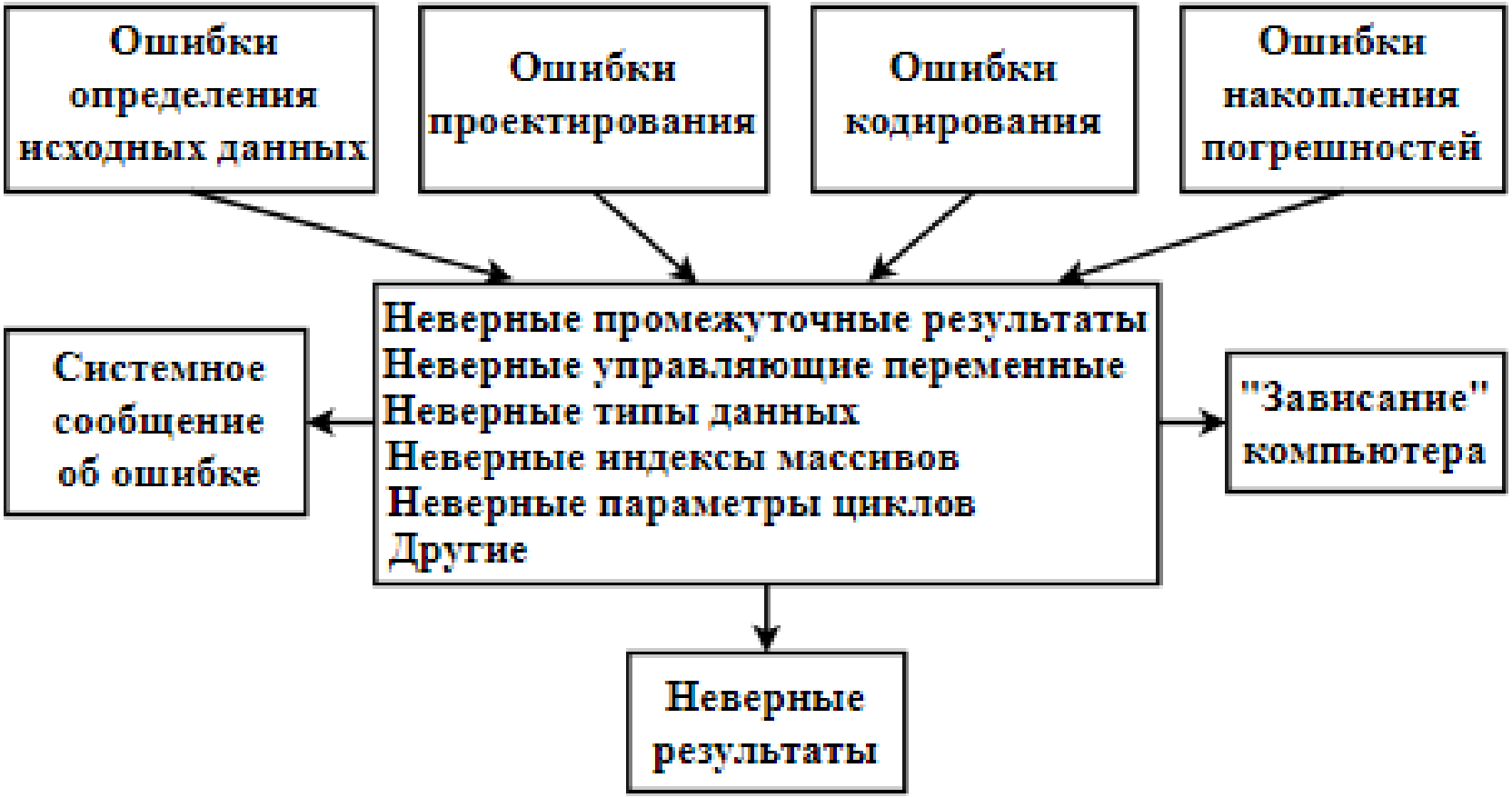


Рисунок 1.5 – Способы проявления ошибок

Однако до того, как результат работы программы становится фатальным, ошибки обычно много раз проявляются в виде неверных промежуточных результатов, неверных управляющих переменных, неверных типах данных, индексах структур данных и т. п. (рисунок 1.5). А это значит, что часть ошибок можно попытаться обнаружить и нейтрализовать, пока они еще не привели к тяжелым последствиям.

Программирование, при котором применяют специальные приемы раннего обнаружения и нейтрализации ошибок, было названо защитным или программированием с защитой от ошибок. При его использовании существенно уменьшается вероятность получения неверных результатов.

Детальный анализ ошибок и их возможных ранних проявлений показывает, что целесообразно проверять:

- правильность выполнения операций ввода-вывода;
- допустимость промежуточных результатов (значений управляющих переменных, значений индексов, типов данных, значений числовых аргументов и т. д.).

Проверки правильности выполнения операций ввода-вывода

Причинами неверного определения исходных данных могут являться, как внутренние ошибки-ошибки устройств ввода-вывода или программного обеспечения, так и внешние ошибки - ошибки пользователя. При этом принято различать:

- ошибки передачи - аппаратные средства, например, вследствие неисправности, искажают данные;
- ошибки преобразования - программа неверно преобразует исходные данные из входного формата во внутренний;
- ошибки перезаписи - пользователь ошибается при вводе данных, например, вводит лишний или другой символ;
- ошибки данных - пользователь вводит неверные данные. Ошибки передачи обычно контролируются аппаратно.

Для защиты от ошибок преобразования данные после ввода обычно сразу демонстрируют пользователю («эхо»). При этом выполняют сначала преобразование во внутренний формат, а затем обратно. Однако предотвратить все ошибки преобразования на данном этапе обычно крайне сложно, поэтому соответствующие фрагменты программы тщательно тестируют, используя методы эквивалентного разбиения и граничных значений.

Обнаружить и устранить ошибки перезаписи можно только, если пользователь вводит избыточные данные, например контрольные суммы. Если ввод избыточных данных по каким-либо причинам нежелателен, то следует по возможности проверять вводимые данные, хотя бы контролировать интервалы возможных значений, которые обычно определены в техническом задании, и выводить введенные данные для проверки пользователю. Неверные данные обычно может обнаружить только пользователь.

Проверка допустимости промежуточных результатов

Проверки промежуточных результатов позволяют снизить вероятность позднего проявления не только ошибок неверного определения данных, но и некоторых ошибок кодирования и проектирования. Для того, чтобы такая проверка была возможной, необходимо, чтобы в программе использовались переменные, для которых существуют ограничения любого происхождения, например, связанные с сущностью моделируемых процессов.

Однако следует также иметь в виду, что любые дополнительные операции в программе требуют использования дополнительных ресурсов (времени, памяти и т. п.) и могут также содержать ошибки. Поэтому имеет смысл проверять не все промежуточные результаты, а только те, проверка которых целесообразна, т. е. возможно позволит обнаружить ошибку, и не сложна. Например:

- если каким-либо образом вычисляется индекс элемента массива, то следует проверить, что этот индекс является допустимым;
- если строится цикл, количество повторений которого определяется значением переменной, то целесообразно убедиться, что значение этой переменной не отрицательно;
- если определяется вероятность какого-либо события, то целесообразно проверить, что полученное значение не более 1, а сумма вероятностей всех возможных независимых событий равна 1 и т. д.

Предотвращение накопления погрешностей. Чтобы снизить погрешности результатов вычислений, необходимо соблюдать следующие рекомендации:

- избегать вычитания близких чисел (машинный ноль);
- избегать деления больших чисел на малые;
- сложение длинной последовательности чисел начинать с меньших по абсолютной величине;
- стремиться по возможности уменьшать количество операций;
- использовать методы с известными оценками погрешностей;
- не использовать условие равенства вещественных чисел;
- вычисления производить с двойной точностью, а результат выдавать с одинарной.

Обработка исключений

Поскольку полный контроль данных на входе и в процессе вычислений, как правило, невозможен, следует предусматривать перехват обработки аварийных ситуаций.

Для перехвата и обработки аппаратно и программно фиксируемых ошибок в некоторых языках программирования, например, Delphi Pascal, C++, Java, предусмотрены средства обработки исключений. Использование эти средств позволяет не допустить выдачи пользователю сообщения об аварийном завершении программы, ничего ему не говорящего. Вместо этого программист получает возможность предусмотреть действия, которые позволяют исправить эту ошибку или, если это невозможно, выдать пользователю сообщение с точным описанием ситуации и продолжить работу.

Сквозной структурный контроль

Сквозной структурный контроль представляет собой совокупность технологических операций контроля, позволяющих обеспечить как можно более раннее обнаружение ошибок в процессе разработки. Термин «сквозной в названии отражает выполнение контроля на всех этапах разработки. Термин «структурный» означает наличие четких рекомендаций по выполнена контролирующих операций на каждом этапе. Сквозной структурный контроль должен выполняться на специальных контрольных сессиях, в которых, помимо разработчиков, могут участвовал специально приглашенные эксперты. Время между сессиями определяет объем материала, который выносится на сессию: при частых сессиях материал рассматривают небольшими порциями, при редких - существенным фрагментами.

Материалы для очередной сессии должны выдаваться участникам заранее, чтобы они могли их обдумать.

Одна из первых сессий должна быть организована на этапе определения спецификаций. На этой сессии проверяют полноту и точность спецификаций, при этом целесообразно присутствие заказчика или специалиста по предметной области, которые смогут определить, насколько правильно полно определены спецификации программного обеспечения.

На этапе проектирования вручную по частям проверяют алгоритмы разрабатываемого программного обеспечения на конкретных наборах данных и сверяют полученные результаты с соответствующими спецификациями. Основная задача - убедиться в правильности понимания спецификаций и проанализировать достоинства и недостатки концептуальных решений, закладываемых в проект.

На этапе реализации проверяют план (последовательность) реализации модулей, набор тестов, а также тексты отдельных модулей.

Для всех этапов целесообразно иметь списки наиболее часто встречающихся ошибок, которые формируют по литературным источникам и исходя из опыта предыдущих разработок. Такие списки позволяют сконцентрировать усилия на конкретных моментах, а не проверять все подряд. При этом все найденные ошибки фиксируют в специальном документе, но не исправляют их.

Помимо раннего обнаружения ошибок, сквозной структурный контроль обеспечивает своевременную подготовку качественной документации по проекту.

https://studfile.net/preview/7249089/

ЛЕКЦИЯ 2

ТЕМА 2: Классификация тестирования. Статическое и динамическое тестирование. Категории программных ошибок. Распространённые программные ошибки. Документирование и анализ ошибок

Тестирование программного обеспечения — процесс исследования, испытания программного продукта, имеющий две различные цели:

- продемонстрировать разработчикам и заказчикам, что программа соответствует требованиям;
- выявить ситуации, в которых поведение программы является неправильным, нежелательным или не соответствующим спецификации.

Существующие на сегодня методы тестирования программного обеспечения не позволяют однозначно и полностью выявить все дефекты и установить корректность функционирования анализируемой программы, поэтому все существующие методы тестирования действуют в рамках формального процесса проверки исследуемого или разрабатываемого программного обеспечения.

Такой процесс формальной проверки, или верификации, может доказать, что дефекты отсутствуют с точки зрения используемого метода. (То есть, нет никакой возможности точно установить или гарантировать отсутствие дефектов в программном продукте с учётом человеческого фактора, присутствующего на всех этапах жизненного цикла программного обеспечения.)

Существует множество подходов к решению задачи тестирования и верификации программного обеспечения, но эффективное тестирование сложных программных продуктов — это процесс в высшей степени творческий, не сводящийся к следованию строгим и чётким процедурам или созданию таковых.

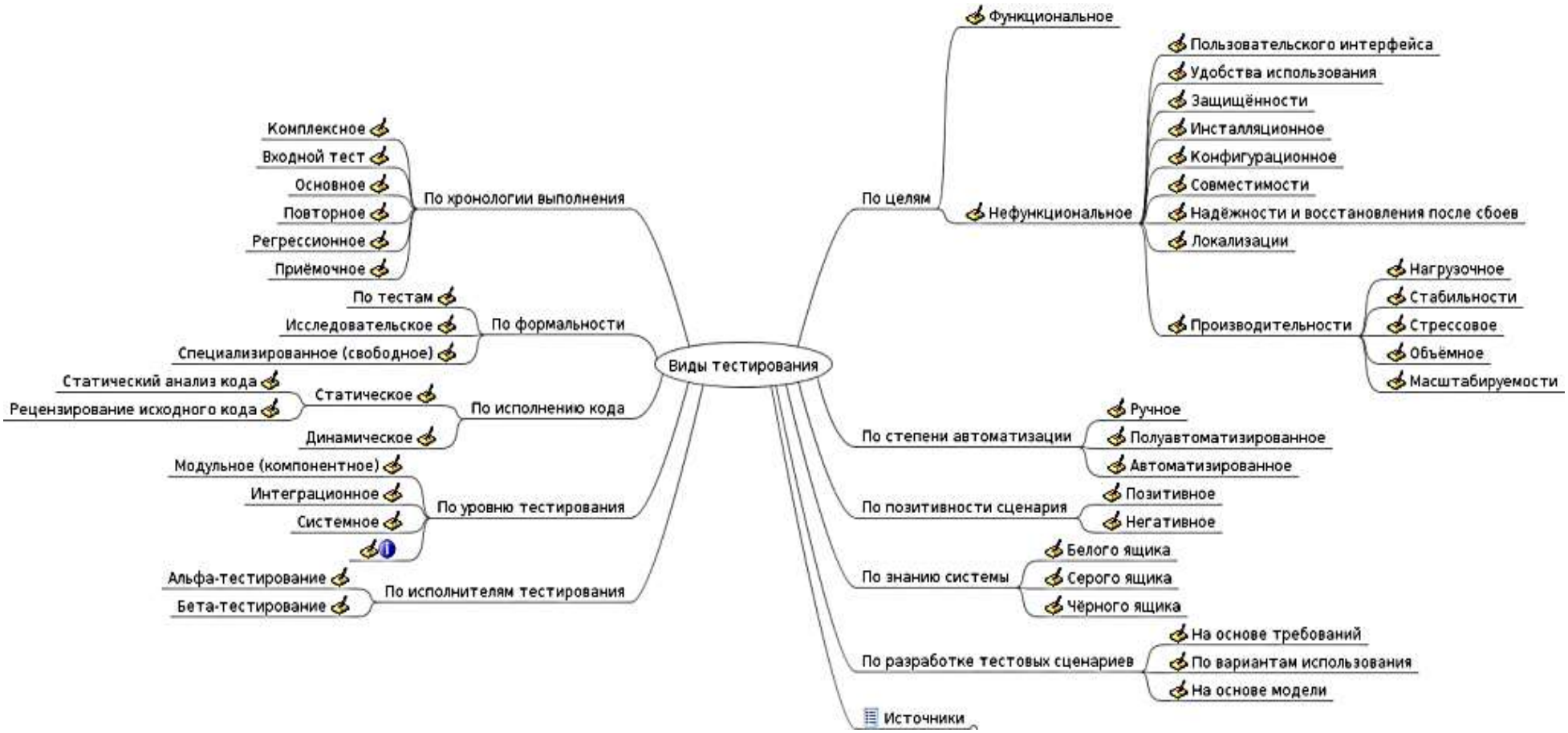
Качество программного обеспечения можно определить как совокупную характеристику исследуемого ПО с учётом следующих составляющих:

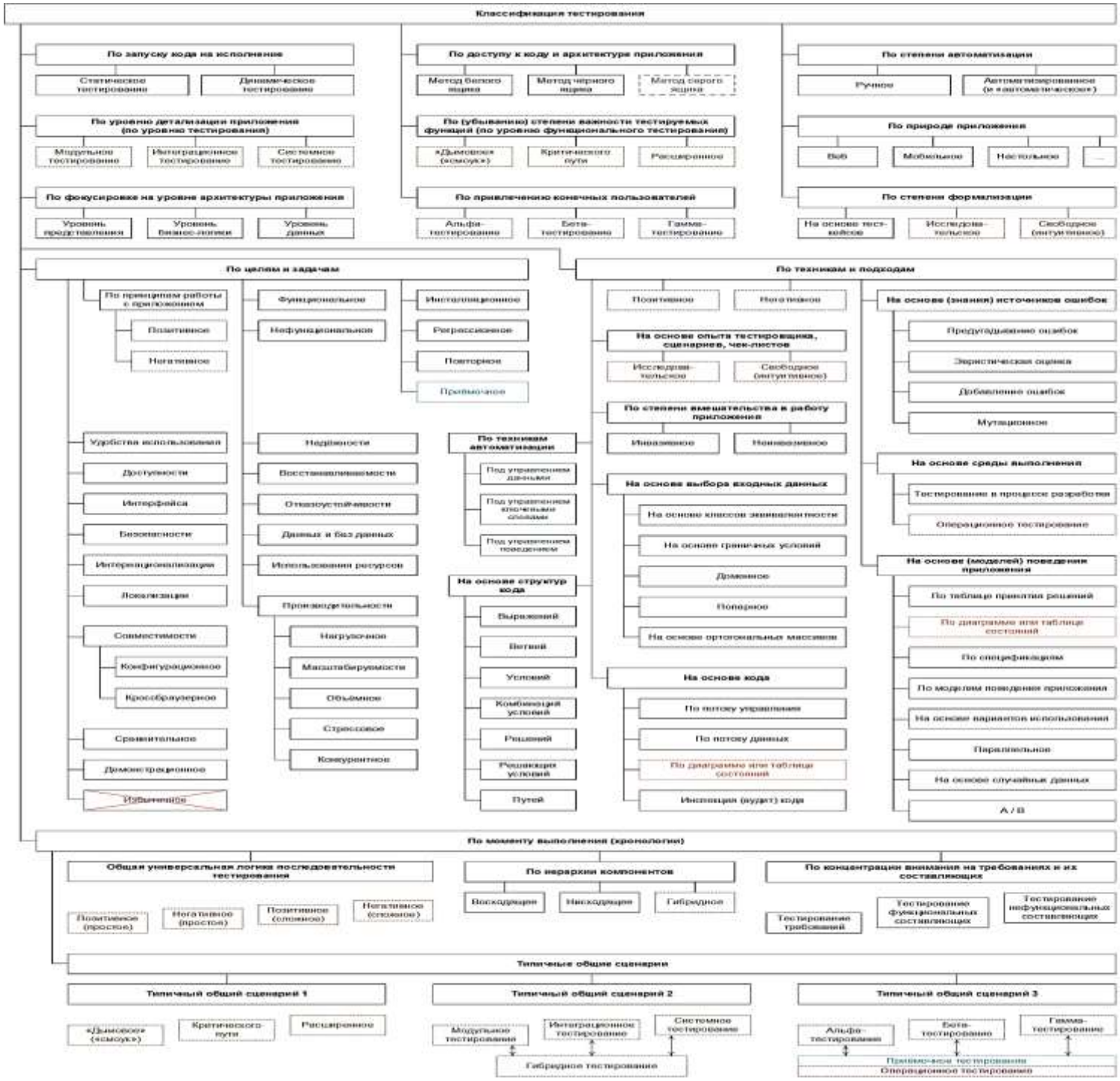
- надёжность;
- сопровождаемость;
- практичность;
- эффективность;
- мобильность;
- функциональность.

Виды тестирования сгруппированы на mind-карте по:

- целям;
- хронологии выполнения;
- формальности;
- позитивности;
- по исполнения кода;
- по уровню тестированию;
- по исполнителям тестирования;
- по степени автоматизации;
- по знанию системы;
- по разработке тестовых сценариев.

Существует несколько признаков, по которым принято производить классификацию видов тестирования. Обычно выделяют следующие:




Классификация тестирования
По запуску кода на исполнение
Статическое тестирование
Динамическое тестирование
По уровню детализации приложения (по уровню тестирования)
Модульное тестирование
Интеграционное тестирование
Системное тестирование
По фокусировке на уровне архитектуры приложения
Уровень представления
Уровень бизнес-логики
Уровень данных
По доступу к коду и архитектуре приложения
Метод белого ящика
Метод черного ящика
Метод серого ящика
По (убыванию) степени важности тестируемых функций (по уровню функционального тестирования)
«Дымовое» («смоук»)
Критического пути
Расширенное
По привлечению конечных пользователей
Альфа-тестирование
Бета-тестирование
Гамма-тестирование
По степени автоматизации
Ручное
Автоматизированное (и «автоматическое»)
По природе приложения
Веб
Мобильное
Настольное
—
По степени формализации
На основе тест-кейсов
Исследовательское
Свободное (интуитивное)
По целям и задачам
По принципам работы с приложением
Позитивное
Негативное
Функциональное
Нефункциональное
Инсталляционное
Регрессионное
Повторное
Приемочное
Удобства использования
Доступности
Интерфейса
Безопасности
Интернационализации
Локализации
Совместимости
Конфигурационное
Кроссбраузерное
Сравнительное
Дымо-трещинное
Избыточное
Надёжности
Восстанавливаемости
Отказоустойчивости
Данных и баз данных
Использования ресурсов
Производительности
Нагрузочное
Масштабируемости
Объёмное
Стрессовое
Конкурентное
По техникам и подходам
Позитивное
Негативное
На основе опыта тестировщика, сценариев, чек-листов
Исследовательское
Свободное (интуитивное)
По степени вмешательства в работу приложения
Инвазивное
Неинвазивное
На основе выбора входных данных
На основе классов эквивалентности
На основе граничных условий
Доменное
Попарное
На основе ортогональных массивов
На основе кода
По потоку управления
По потоку данных
По диаграмме или таблице состояний
Инспекция (аудит) кода
По техникам автоматизации
Под управлением данными
Под управлением ключевыми словами
Под управлением поведением
На основе структур кода
Выражений
Ветвей
Условий
Комбинаций условий
Решений
Решающих условий
Путей
На основе (знания) источников ошибок
Предугадывание ошибок
Эвристическая оценка
Добавление ошибок
Мутационное
На основе среды выполнения
Тестирование в процессе разработки
Операционное тестирование
На основе (моделей) поведения приложения
По таблице принятия решений
По диаграмме или таблице состояний
По спецификациям
По моделям поведения приложения
На основе вариантов использования
Параллельное
На основе случайных данных
A / B
По моменту выполнения (хронологии)
Общая универсальная логика последовательности тестирования
Позитивное (простое)
Негативное (простое)
Позитивное (сложное)
Негативное (сложное)
По иерархии компонентов
Восходящее
Нисходящее
Гибридное
По концентрации внимания на требованиях и их составляющих
Тестирование требований
Тестирование функциональных составляющих
Тестирование нефункциональных составляющих
Типичные общие сценарии
Типичный общий сценарий 1
«Дымовое» («смоук»)
Критического пути
Расширенное
Типичный общий сценарий 2
Модульное тестирование
Интеграционное тестирование
Системное тестирование
Гибридное тестирование
Типичный общий сценарий 3
Альфа-тестирование
Бета-тестирование
Гамма-тестирование
Приемочное тестирование
Операционное тестирование

По объекту тестирования

- Функциональное тестирование.
- Тестирование производительности.
- Нагрузочное тестирование.
- Стресс-тестирование.
- Тестирование стабильности.
- Конфигурационное тестирование.
- Юзабилити-тестирование.
- Тестирование интерфейса пользователя.
- Тестирование безопасности.
- Тестирование локализации.
- Тестирование совместимости.

По степени автоматизации

- Ручное тестирование.
- Автоматизированное тестирование.
- Полуавтоматизированное тестирование.

По знанию системы

- Тестирование чёрного ящика.
- Тестирование белого ящика.
- Тестирование серого ящика.

По степени изолированности компонентов

- Модульное тестирование.
- Интеграционное тестирование.
- Системное тестирование.

По времени проведения тестирования

- Альфа-тестирование.
- Дымовое тестирование (англ. smoke testing).
- Тестирование новой функции (new feature testing).
- Подтверждающее тестирование.
- Регрессионное тестирование.
- Приёмочное тестирование.
- Бета-тестирование.
- Гамма-тестирование.

По признаку позитивности сценариев

- Позитивное тестирование.
- Негативное тестирование.

По степени подготовленности к тестированию

- Тестирование по документации (формальное тестирование).
- Интуитивное тестирование (англ. ad hoc testing).

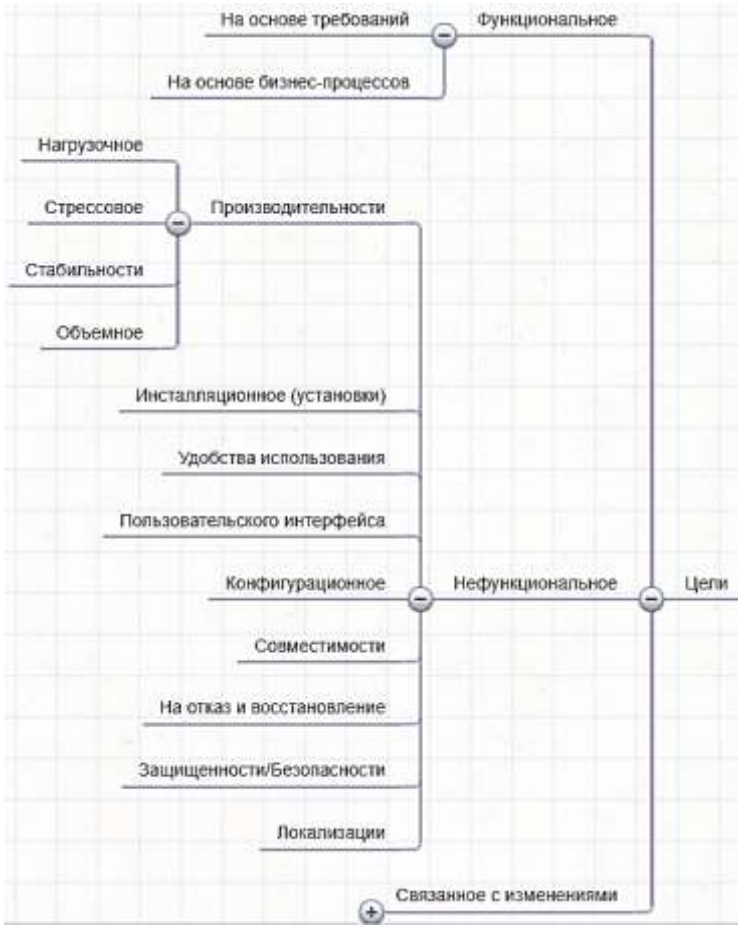
Итак, на сегодняшний момент наши знания о видах тестирования выглядят следующим образом.

Виды тестирования по целям

Виды тестирования по целям: виды нефункционального тестирования

Нефункциональное тестирование направлено на определение характеристик программного обеспечения, которые могут быть измерены различными величинами. То есть это проверка того, «насколько хорошо работает система».

Рассмотрим виды тестирования, которые относятся к нефункциональному.



Тестирование производительности

Проверяет, как программный продукт работает под определенной нагрузкой.

Задачей тестирования производительности является определение масштабируемости приложения под нагрузкой, при этом происходит:

- измерение времени выполнения выбранных операций при определенных интенсивностях выполнения этих операций;
- определение количества пользователей, одновременно работающих с приложением;
- определение границ приемлемой производительности при увеличении нагрузки (при увеличении интенсивности выполнения этих операций);
- исследование производительности на высоких, предельных, стрессовых нагрузках.

Тестирование производительности включает в себя:

Нагрузочное тестирование (Performance and Load Testing)

Определяет, как программа работает под ожидаемой нагрузкой.

При этом происходит:

- измерение времени выполнения выбранных операций при определенных интенсивностях выполнения этих операций,

- определение количества пользователей, одновременно работающих с приложением,

- определение границ приемлемой производительности при увеличении нагрузки (при увеличении интенсивности выполнения этих операций),

- исследование производительности на высоких, предельных, стрессовых нагрузках.

Стрессовое тестирование (Stress Testing)

Определяет работоспособность программного продукта при максимальной нагрузке.

Стрессом в данном контексте может быть повышение интенсивности выполнения операций до очень высоких значений или аварийное изменение конфигурации сервера. Также одной из задач при стрессовом тестировании может быть оценка деградации производительности.

*Важно **не путать стрессовое и нагрузочное** тестирование. Нагрузочное тестирование — это все же тестирование средних нагрузок. А стрессовое — это тестирование запредельных нагрузок*

Тестирование стабильности/надежности (Stability / Reliability Testing)

Проверяет, выдержит ли программный продукт длительную нагрузку.

Время на выполнения операций может играть в данном виде тестирования второстепенную роль. При этом на первое место выходит отсутствие утечек памяти, перезапусков серверов под нагрузкой и другие аспекты, влияющие именно на стабильность работы.

Объемное тестирование (Volume Testing)

Для получения оценки производительности при увеличении объёмов данных в базе данных приложения. При этом происходит:

- измерение времени выполнения выбранных операций при определенных интенсивностях выполнения этих операций,
- может производиться определение количества пользователей, одновременно работающих с приложением.

Итак, закрепим. Какую же работоспособность проверяем в каждом из видов тестирования производительности?

Нагрузочное тестирование — при ***нормальных*** условиях.

Стрессовое тестирование — при ***экстремальных*** нагрузках.

Тестирование стабильности — при ***длительной*** работе.

Объемное тестирование — при ***увеличенных объемах*** обрабатываемых данных.

Инсталляционное тестирование/Тестирование установки (Installation Testing)

Проверяет, не возникает ли проблем при установке, удалении, а также обновлении программного продукта.

Обычно пользователь самостоятельно выполняет установку программного обеспечения, используя документацию в виде инструкций или readme файлов, шаг за шагом описывающих все необходимые действия и проверки.

В распределенных системах, где приложение разворачивается на уже работающем окружении, простого набора инструкций может быть мало. Для этого, зачастую, пишется план установки (Deployment Plan), включающий не только шаги по инсталляции приложения, но и шаги отката (roll–back) к предыдущей версии, в случае неудачи.

Сам по себе план установки также должен пройти процедуру тестирования для избежания проблем при выдаче в реальную эксплуатацию. Особенно это актуально, если установка выполняется на системы, где каждая минута простоя – это потеря репутации и большого количества средств, например: банки, финансовые компании или даже баннерные сети.

Поэтому тестирование установки можно назвать одной из важнейших задач по обеспечению качества программного обеспечения.

Тестирование удобства использования/Юзабилити тестирование (usability testing)

Это метод тестирования, направленный на установление степени удобства использования, «обучаемости», понятности и привлекательности для пользователей разрабатываемого продукта в контексте заданных условий. [ISO 9126]

По сути, проверяет, удобен ли программный продукт в использовании.

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

- **производительность, эффективность (efficiency)** — сколько времени и шагов понадобится пользователю для завершения основных задач приложения, например размещение новости, регистрации, покупка и т.д.? (меньше - лучше)
- **правильность (accuracy)** — сколько ошибок сделал пользователь во время работы с приложением? (меньше - лучше)
- **активизация в памяти (recall)** — как много пользователь помнит о работе приложения после приостановки работы с ним на длительный период времени? (повторное выполнение операций после перерыва должно проходить быстрее чем у нового пользователя)
- **эмоциональная реакция (emotional response)** — как пользователь себя чувствует после завершения задачи - растерян, испытал стресс? Порекомендует ли пользователь систему своим друзьям? (положительная реакция - лучше)

Стоит обратить внимание, что данный метод не имеет ничего общего с тестированием пользовательского интерфейса.

Тестирование пользовательского интерфейса (Graphical user interface, GUI)

Определяет удобство пользовательского интерфейса (кнопки, цвета, выравнивание и т.д.).

Основные моменты при проверке:

- расположение, размер, цвет, ширина, длина элементов; возможность ввода букв или цифр;
- реализуется ли функционал приложения с помощью графических элементов;
- размещение всех сообщений об ошибках, уведомлением (а также шрифт, цвет, размер, расположение и орфография текста);
- читабелен ли использованный шрифт;
- переходит ли курсор из текстового в поинтер при наведении на активные элементы, выделяются ли выбранные элементы;
- выравнивание текста и форм;
- качество изображений;
- проверить расположение и отображение всех элементов при различных разрешениях экрана, а также при изменении размера окна браузера (проверить, появляется ли скролл);
- проверить текст на орфографические, пунктуационные ошибки;
- появляются ли тултипы (если есть необходимость);
- унификация дизайна (цвета, шрифты, текст сообщений, названия кнопок и т. д.).

Конфигурационное тестирование (Configuration Testing)

Определяет, как программный продукт будет работать в условиях смены конфигурации (платформы, драйверов, компьютеров). Данный вид тестирования применяется, если известно, что продукт будет использоваться на разных платформах, в различных браузерах, будет поддерживать разные версии драйверов и т. п.

Исходя из определения, можно выделить 2 **цели** конфигурационного тестирования:

- Определить ***оптимальную конфигурацию*** оборудования, обеспечивающую требуемые характеристики производительности и времени реакции тестируемой системы.

- Проверить объект тестирования на ***совместимость*** с объявленным в спецификации оборудованием, операционными системами и программными продуктами третьих фирм.

Можно выделить 2 **уровня** проведения тестирования конфигурации:

1. **Клиентский уровень**

Приложение тестируется с позиции рабочего окружения конечного пользователя. А именно:

- Кроссплатформенное тестирование (типы и версии ОС).
- Кроссбраузерное тестирование (используется, при тестировании веб-приложения).
- Тестирование работы при различных версиях драйверов.
- При тестировании игровых приложений – тестирование видеоадаптера.

2. **Клиент-серверный уровень**

Если приложение клиент-серверное, необходимо протестировать взаимодействие приложение с окружением:

- Аппаратным (тип и количество процессоров, объем памяти, характеристики сети / сетевых адаптеров и т. д.).
- Программным (ОС, драйвера и библиотеки, стороннее ПО, влияющее на работу приложения и т.д.).

Непосредственно, само **тестирование проводиться** таким образом:

– Определяются все возможные конфигурации, которые необходимо протестировать.

– Данные конфигурации распределяются в очередь по приоритету, так как их количество может быть огромным.

– В соответствии с установленными приоритетами проводится само тестирование.

Тестирование совместимости (Compatibility testing)

Это тестирование работы программного продукта в определенном окружении. Окружение может включать в себя следующие элементы:

- аппаратная платформа;
- сетевые устройства;
- периферия (принтеры, CD/DVD-приводы, веб-камеры и пр.);
- операционная система (Unix, Windows, MacOS, ...);
- базы данных (Oracle, MS SQL, MySQL, ...);
- системное программное обеспечение (веб-сервер, файрвол, антивирус, ...);
- браузеры (Internet Explorer, Firefox, Opera, Chrome, Safari).

В чем же его отличие от конфигурационного тестирования?

При конфигурационном тестировании тестируют различные конфигурации железа, софта, которые поддерживает система. Например, сайт по ТЗ должен работать на веб-браузерах Google Chrome, Mozilla Firefox и Apple Safari. Или приложение должно запускаться с 500 мб оперативной памяти.

Тестирование совместимости проверяет совместимость с объектами вне зависимости от того поддерживает ли их система.

Пример. Приложение разработано для работы с семейством Windows. При тестировании его на различных видах Windows (XP, 7, 10, ...), это будет конфигурационное тестирование. Но когда мы начнем тестировать его на Linux, то это уже будет тестирование совместимости.

Тестирование на отказ и восстановление (Failover and Recovery Testing)

Исследование программной системы на предмет восстановления после ошибок, сбоев. Оценивание реакции защитных свойств приложения.

Данный вид тестирования имеет весьма специфический (сравнительно с другими видами) подход к выполнению тестов, так как объектами исследования являются:

- Поведение ПО при прерывании обработки данных.
- При потере сети.
- При отключении электричества (на стороне клиента или сервера).
- Потеря подключения носителей данных.

Следовательно, и различные сценарии тестирования разрабатываются, опираясь на вышеупомянутые факторы влияния на способность ПО к восстановлению после сбоя.

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

- Симулировать внезапный отказ электричества на компьютере (обесточить компьютер).
- Симулировать потерю связи с сетью (выключить сетевой кабель, обесточить сетевое устройство).
- Симулировать отказ носителей (обесточить внешний носитель данных).
- Симулировать ситуацию наличия в системе неверных данных (специальный тестовый набор или база данных).

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

- Потеря или порча данных в допустимых пределах.
- Отчет или система отчетов с указанием процессов или транзакций, которые не были завершены в результате сбоя.

Тестирование на отказ и восстановление **особенно актуально** при разработке систем, которые должны работать на протяжении длительного времени, вплоть до 24x7. От способности такого ПО возобновлять работоспособность после непредвиденной ситуации, а также минимизировать потери данных будет зависеть не только репутация компании-разработчика, а порой и нечто больше, чем деньги.

При моделировании ситуации сбоя, оценивается как степень потери данных (находится ли она в пределах допустимого), так и способность системы журналировать все транзакции и их статус выполнения.

Тестирование защищенности/безопасности (Security testing)

Определяет, насколько безопасно использование программного продукта, т. е. защищен ли программный продукт от атак хакеров, несанкционированного доступа к данным и т.д.

Общая стратегия безопасности основывается на трех основных **принципах**:

- конфиденциальность,
- целостность,
- доступность.

Конфиденциальность

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

Целостность

Существует два основных критерия при определении понятия целостности:

1. **Доверие**. Ожидается, что ресурс будет изменен только соответствующим способом определенной группой пользователей.
2. **Повреждение и восстановление**. В случае, когда данные повреждаются или неправильно меняются авторизованным или не авторизованным пользователем, необходимо определить на сколько важной является процедура восстановления данных.

Доступность

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

Тестирование локализации (Localization testing)

Тестирование локализованной версии программного продукта. Локализация — это процесс адаптации интерфейса ПО под разные регионы, культуры, языки.

Многие полагают, что локализация – это просто перевод на другой язык, но это не так. Вот примеры локализации, которая может использоваться:

Язык. В разных странах говорят на разных языках. Это верно не только для стран, но и для регионов.

Правописание. Даже когда две области говорят на одном языке, они могут писать по-разному. К примеру, цвет пишется как "color" в США, но как "colour" в Канаде и Великобритании.

Слова и идиомы. Они могут различаться даже в пределах одного языка. В США глагол "table" применительно к теме разговора означает, что этот разговор нужно отложить, а в Великобритании и Канаде – что на эту тему нужно немедленно начать говорить.

Валюта. Использование разных значков перед суммой ($ или £).

Форматы даты и времени. В США даты пишутся как месяц/день/год, а в Великобритании – как день/месяц/год. В США, как правило, используют AM и PM, обозначая время, но во многих странах используется 24-часовое время, и 1:00 PM в США превратится в 13:00.

Системы мер и весов. Например, большая часть стран измеряет температуру по Цельсию, а в США это делается по Фаренгейту.

Индексы и номера телефонов.

Изображения. Картинки, возможно, придется менять в зависимости от страны, принимая во внимание определенные моменты.

Виды тестирования по запуску кода

Можем ли протестировать только работающий продукт или можем начать его тестирование еще до запуска?

Тестирование не всегда предполагает взаимодействие с работающим приложением. Отсюда и классификация тестирования по запуску кода на исполнителя.

По критерию запуска программы (исполняется ли программный код) выделяют 2 вида тестирования:

– статическое тестирование;

– динамическое тестирование.

Статическое тестирование

Метод статического тестирования (static testing) – это тип тестирования ПО, где программное обеспечение проверяется без запуска кода; является процессом или инструментом, направленным на обнаружение возможных багов в ПО. Кроме этого, он находит и устраняет ошибки в разного рода сопроводительных документах, например спецификации требований к ПО.

Статическое тестирование ***начинается на ранних этапах*** жизненного цикла ПО и является, соответственно, частью процесса верификации.

Можно поделить статическое тестирование на 2 типа:

1. Обзоры (Review).
2. Статический анализ (Static Analysis).

1. Обзоры

Обзоры (Review) – проверка обычно используется для поиска и устранения ошибок или неясностей в документах. Это могут быть требования, дизайн, тестовые случаи и так далее.

В свою очередь обзоры делятся на:

- **Неформальные.** При неофициальном рассмотрении создатель документов показывает содержание документов аудитории. Каждый присутствующий высказывает свое мнение, что позволяет выявить недостатки на ранней стадии.
- **Сквозные просмотры (Walkthroughs).** Выполняются опытным человеком или экспертом для проверки отсутствия дефектов, с целью предупреждения возникновения проблем на этапе разработки или тестирования.
- **Экспертная оценка.** Означает проверку документов для выявления и исправления дефектов. В основном это делается в команде.
- **Инспектирование ПО.** Это, в большинстве, проверка документа вышестоящим органом, например, проверка требований к программному обеспечению.

2. Статический анализ

Статический анализ (Static Analysis**)** – код, написанный разработчиками, анализируется на наличие структурных дефектов, которые могут привести к ошибкам.

Статический анализ включает оценку качества кода, написанного разработчиками. Для анализа кода и сравнения его со стандартом используются разные инструменты. Статический анализ хорошо помогает найти такие ошибки, как:

- неиспользуемые переменные;
- мертвый код;
- бесконечные циклы;
- переменные с неопределенными значениями;
- неправильный синтаксис.

Статический анализ состоит из 3-х частей:

1. Поток данных.
2. Контроль потока (как выполняются операторы или инструкции).
3. Цикломатическая сложность (измерение сложности программы, которое в основном связано с количеством независимых путей в графе потоков управления программы).

Анализ может производиться как вручную, так и с помощью специальных инструментов. Например, можно использовать автоматические средства проверки синтаксиса программного кода.

Целью анализа является ***наиболее раннее выявление ошибок*** и потенциальных проблем в программном продукте. Как правило, код ревью выполняется самим разработчиком.

Примерами ошибок, которые потенциально можно выявить с помощью автоматического статического тестирования, могут быть:

– утечки ресурсов (утечки памяти, неосвобождаемые файловые дескрипторы и т. д.),
– возможность переполнения буфера (buffer overflows),
– ситуации частичной (неполной) обработки ошибок.

Как правило, результатом автоматического анализа кода является список рекомендаций для ручного review некоторых участков кода, потенциально содержащих ошибки.

В рамках этого подхода тестированию могут подвергаться:

- Документы (требования, тест-кейсы, описания архитектуры приложения, схемы баз данных и т.д.).
- Графические прототипы (например, эскизы пользовательского интерфейса).
- Код приложения (что часто выполняется самими программистами в рамках аудита кода (code review), являющегося специфической вариацией взаимного просмотра в применении к исходному коду). Код приложения также можно проверять с использованием техник тестирования на основе структур кода.
- Параметры (настройки) среды исполнения приложения.
- Подготовленные тестовые данные.

Плюсы и минусы

Преимущества статического тестирования:

- Снижает стоимость исправления найденных багов, поскольку выявляет баги на ранних этапах цикла разработки программного обеспечения.
- Отзывы, полученные в ходе этого тестирования, помогают улучшить функционирование процесса, что также помогает команде избежать подобных дефектов и багов.
- Повышает информированность о различных проблемах качества программного обеспечения.
- Улучшает обмен критической и важной информацией между членами команды.
- Существенно сокращаются усилия по исправлению ошибок, что еще больше способствует продуктивности разработки.

Недостатки статического тестирования:

- Процесс статического тестирования может занимать много времени, так как в основном он выполняется вручную.
- Препятствует обнаружению уязвимостей, представленных в среде выполнения.

Динамическое тестирование

Динамическое тестирование (dynamic testing) – это методика, направленная на проверку функционала программы, во время выполнения кода. Запускаться на исполнение может как код всего приложения целиком (системное тестирование), так и код нескольких взаимосвязанных частей (интеграционное тестирование), отдельных частей (модульное или компонентное тестирование) и даже отдельные участки кода.

То есть, данный тип тестирования подразумевает фактическую эксплуатацию программы и определение того, как работает ее функционал, в соответствии с ожиданиями или нет.

Динамическое тестирование включает в себя тестирование ПО в режиме реального времени путем предоставления входных данных и изучения результата поведения программы. Проверка осуществляется с помощью ручного или автоматического выполнения заранее подготовленного набора тестов. Оно является частью процесса валидации программного обеспечения.

То есть любое тестирование, в котором начинают взаимодействовать с приложением, является динамическим. Например,

проверка авторизации на сайте,

запуск приложения,

посадка деревьев,

смена оружия,

многое другое.

Задача — посмотреть, как продукт реагирует на действия. Для этого вводятся все необходимые условия и просматривается результат.

Если рассмотреть функции, предлагаемые динамическим тестированием, можно легко понять причины его выполнения в течение жизненного цикла тестирования программного обеспечения. С помощью этого тестирования можно проверить различные критические аспекты программного обеспечения. Если оставить их без какой-либо оценки, они могут повлиять на производительность, функционирование, а также надежность программного продукта.

Плюсы и минусы

Преимущества динамического тестирования

- Это тщательное исследование, которое рассматривает всю функциональность приложения, поэтому качество соответствует самым высоким стандартам.
- Процесс динамического тестирования хорошо налажен, приложение тестируется с точки зрения пользователя, что повышает качество ПО.
- Обнаружение сложных ошибок, которые могли ускользнуть на этапе анализа кода.
- Динамическое тестирование может быть автоматизировано с помощью специальных инструментов.

Недостатки динамического тестирования

- Поскольку динамическое тестирование представляет собой сложный процесс, оно занимает много времени.
- Высокая стоимость проведения тестирования.
- Динамическое тестирование обычно выполняется после завершения кодирования, и найденные баги обнаруживаются позже в жизненном цикле разработки.

Сравнение

	Динамическое тестирование	Статическое тестирование
Этап	Валидация	Верификация
Выполнение программного кода	Требует	Не требует
Направленность	Обеспечивает функциональность продукта	Ориентировано на предотвращение дефектов
Этап разработки ПО	Более поздние этапы	Ранние этапы
Стоимость исправления багов	Высокая	Более низкая
Покрытие	Покрывает ограниченную область кода, динамическое тестирование требует меньшего охвата кода	Обеспечивает более широкий охват кода, чем динамическое тестирование, за более короткий промежуток времени
Цель	Сравнение реального поведения программы с ожидаемым	Предотвращение дефектов программного обеспечения
Число дефектов	Обнаруживается меньше дефектов, чем при статическом тестировании	Комплексное тестирование кода, которое помогает найти больше дефектов в системе
Когда выполняется	После развертывания приложения (подготовки к использованию)	Перед развертыванием приложения

Статическое тестирование требует много времени на бурные дискуссии и встречи. Однако оно помогает предотвращать появления дефектов на последних этапах разработки продукта. Поэтому статическое тестирование по праву считается важным шагом на пути к разработке ПО без ошибок.

Динамическое тестирование не менее важно. Благодаря непосредственному выполнению тестов программного обеспечения (проверки функционального поведения, производительности, надежности и других важных аспектов) команда может проверить и подтвердить качество и эффективность ПО.

На текущий момент существует множество **видов тестирования** также существует большое количество классификаций эти видов. Основная классификация видов тестирования происходит по целям. На рисунке ниже представленная **классификация видов тестирования**.

Виды тестирования по позитивности сценария

Многие знают, что тестирование бывает негативным и позитивным. Но для чего же нам разделять его на негативное и позитивное? Есть ли в этом смысл? Разберем в статье.

Позитивное тестирование – это тестирование с применением сценариев, которые соответствуют нормальному (штатному, ожидаемому) поведению системы. С его помощью можно определить, что система делает то, для чего и была создана.

Примеры:

- умножить на калькуляторе цифр 3 и 5,
- в игре посадить морковь на грядку для овощей,
- оплатить покупку действующей картой.

Негативным называют тестирование, в рамках которого применяются сценарии, которые соответствуют внештатному поведению тестируемой системы. Это могут быть исключительные ситуации или неверные данные.

Пример:

- умножить на калькуляторе числа 3 на грушу (значение «груша» не является валидным для калькулятора),
- в игре посадить морковь на реку,
- оплатить покупку несуществующей картой.

Прежде всего негативное тестирование направлено на проверку устойчивости системы к различным воздействиям, валидации неверных данных, обработку исключительных ситуаций. Сценарии позитивного тестирования, в свою очередь, направлены на проверку работы системы с теми типами данных для которых, она разрабатывалась.

Реакция продукта на тесты

Какой результат ожидаем от позитивных и негативных тестов?

Позитивное тестирование должно всегда давать результат в виде отсутствия багов.

Негативные проверки могут дать 2 результата:

1. На данный ввод у продукта есть ответ в виде сообщения/контроля.

2. Система не знает, как реагировать на введенные данные.

Получается, есть три реакции на действия по вводу данных:

- Действие: создание сущности, переход на новый шаг и т. п.
- Контроль: сообщение с контролем, блокировка дальнейших действий и т. п.
- Отказ: возникает исключение Exception, 404-й ошибки и т.п.

Для чего важно различать негативное и позитивное тестирование?

Чтобы верно расставлять приоритеты в тестировании в зависимости от ситуации.

Создание позитивных сценариев (тест-кейсов), как правило, предшествует созданию негативных.

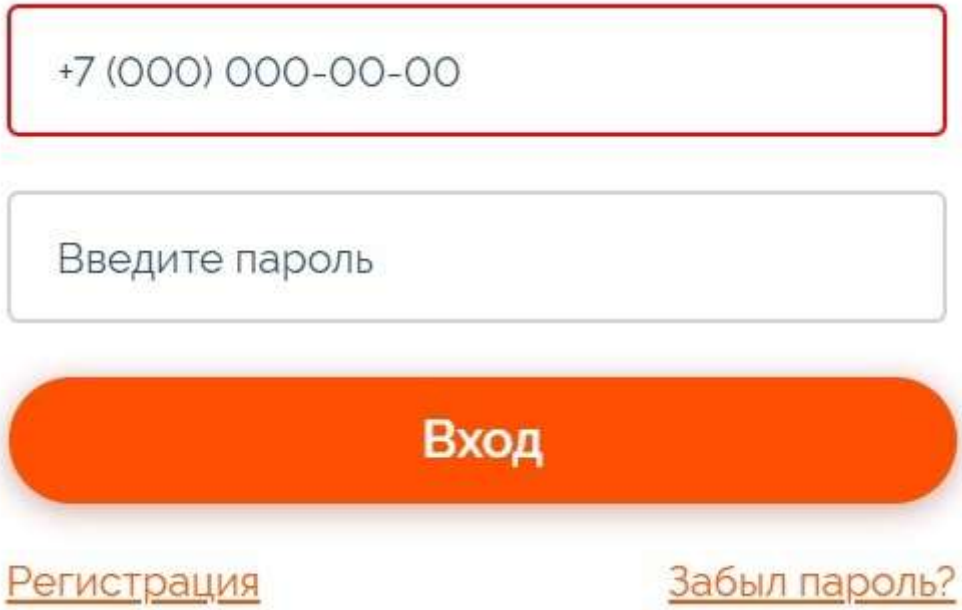
Сначала проверяют работу системы, когда условный пользователь работает с системой «правильно». А уже потом приступаем к проверке отклика системы на пользователя, который допускает различные ошибки (ввод неверных данных, например). И система должна быть готова ответить на неверный запрос. Это и есть цель негативного тестирования.

Почему важно сначала провести позитивное тестирование? Большинство пользователей использует тестируемый продукт так, как необходимо. То есть, если в поле ввода просят указать «Имя», то большинство пользователей напишут в него именно имя, а не набор цифр. Если не проверим верно ли распознаются корректные данные, то в случае ошибки большинство пользователей не смогут воспользоваться данным продуктом.

Именно поэтому делят все тесты на позитивные и негативные и начинаем тестировать с позитивных. Именно с позитивных, так как они приоритетней. Лучше не останется времени на негативные тесты, чем не будет проверен основной функционал продукта на способность корректно отвечать пользователю на корректные запросы.

Пример позитивных и негативных тестов

Давайте рассмотрим эти виды тестирования немного подробнее на примере формы авторизации на сайте.



Авторизация на сайте

С помощью **позитивных тестов** проверим, что данная форма регистрации работает верно. Для этого проведем следующие тесты:

- Авторизация с правильным логином и паролем

К **негативным тестам** можно отнести следующие сценарии:

- Авторизация с неправильным логином.
- Авторизация с неправильным паролем.
- Авторизация с неправильным логином и паролем.
- Авторизация с пустым логином.
- Авторизация с пустым паролем.
- Авторизация с пустым логином и паролем.

Отметим, что при тестировании очень важно соблюдать приоритет!

Сначала выполняем позитивные тесты, а потом негативные.

Это связано с тем, что для продукта и пользователей важно, чтобы весь функционал работал правильно, так как задумывались изначально.

К примеру, ошибка при авторизации с правильным логином и паролем гораздо опаснее, чем проблема возникающая, когда пользователь вводит неправильный пароль. А критичные ошибки лучше всегда находить как можно раньше, чтобы было время их исправить и внимательно проверить.

Виды тестирования по доступу к коду

Мы, как тестировщики, при тестировании продукта можем иметь доступ к его коду или не иметь. Почему оба подхода до сих пор существуют? Разве тестирование с доступом к коду не эффективнее? Может существует золотая середина? Давайте разбираться.

В терминологии тестирования фразы «тестирование белого ящика» и «тестирование чёрного ящика» относятся к тому, имеет ли тестировщик доступ к исходному коду тестируемого ПО или нет.

Черный ящик

Разработка тестов методом **черного ящика** (black box test design technique) — процедура создания и/или выбора тестовых сценариев, основанная на анализе функциональной или нефункциональной спецификации компонента или системы без знания внутренней структуры. (ISTQB)

Тестируемая программа для тестировщика – как черный непрозрачный ящик, содержания которого он не видит.

Основной посыл такого тестирования в том, что **мы не знаем, как устроена тестируемая система.** Имеется в виду изнутри. При таком тестировании тестировщик очень похож на обычного пользователя: тест анализ и исследование продукта он проводит, опираясь на ТЗ (техническое задание), спецификации и прочую документацию, которая описывает этот продукт.

Получается, что идеи для тестирования идут от предполагаемых паттернов (pattern — образец) поведения пользователей. Поэтому такой подход еще называют поведенческим.

Пример. Заходим в приложение вызова такси и видим возможность привязать карту для автоматической оплаты. Начинаем думать, как пользователь:

— Что, если привязать заблокированную карту?

— А если забыть/неверно указать срок действия карты?

— Долларовая карта привяжется?

— А может можно после вызова такси и подачи машины быстро отвязать ее до списания оплаты… Что тогда будет? Спишутся средства? Или водителю придет уведомление, что оплата изменилась с безналичной на наличную?

По сути, мы тестируем и строим предположения **на основе того, что видим** и рисуем себе в голове. То есть мы только предполагаем, что элементы должны работать таким образом и на основании этого подбираем тесты, но точно не уверены, что это именно так.

Либо **открываем спецификацию** и смотрим, как система должна работать. Потом запускаем продукт и сверяем его с тем, что указано в спецификации.

Таким образом, мы не имеем представления о структуре и внутреннем устройстве системы. Нужно концентрироваться на том, что программа делает, а не на том, как она это делает.

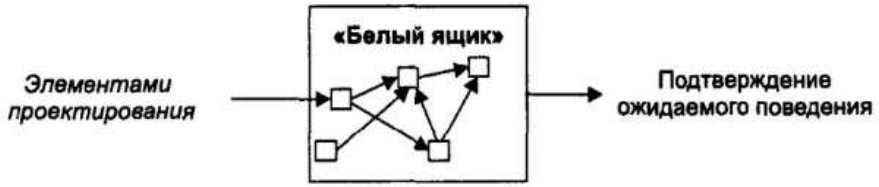
Преимущества:

- Тестировщик не обязан обладать (глубокими) знаниями в области программирования.
- Поведение приложения исследуется в контексте реальной среды выполнения и учитывает её влияние.
- Поведение приложения исследуется в контексте реальных пользовательских сценариев.
- Тест-кейсы можно создавать уже на стадии появления стабильных требований.
- Процесс создания тест-кейсов позволяет выявить дефекты в требованиях.
- Допускает создание тест-кейсов, которые можно многократно использовать на разных проектах.

Недостатки:

- Возможно повторение части тест-кейсов, уже выполненных разработчиками.
- Высока вероятность того, что часть возможных вариантов поведения приложения останется не протестированной.
- Для разработки высокоэффективных тест-кейсов необходима качественная документация.
- Диагностика обнаруженных дефектов более сложна в сравнении с техниками метода белого ящика.
- В связи с широким выбором техник и подходов затрудняется планирование и оценка трудозатрат.
- В случае автоматизации могут потребоваться сложные дорогостоящие инструментальные средства.

Белый ящик



Разработка тестов методом **белого ящика** (white-box test design technique): Процедура разработки или выбора тестовых сценариев на основании анализа внутренней структуры компонента или системы. (ISTQB)

Белый ящик является **полной противоположностью черному ящику**. При тестировании черного ящика нам необходимо запускать программу и смотреть, что она делает. А в белом этого не требуется. Достаточно смотреть на код программы.

Основной посыл этого типа тестирования — нам **известны все детали** реализации тестируемой программы.

Тестирование методом белого ящика (прозрачного, открытого, стеклянного ящика, основанное на коде или структурное тестирование) – метод тестирования программного обеспечения, который предполагает, что внутренняя структура/устройство/реализация системы известны тестировщику.

Мы выбираем входные значения, основываясь на знании кода, который будет их обрабатывать. Так же мы знаем, каким должен быть результат этой обработки. Знание всех особенностей тестируемой программы и ее реализации – обязательны для этой техники. Тестирование белого ящика – углубление во внутреннее устройство системы, за пределы ее внешних интерфейсов.

Профессиональные тестировщики, которые тестируют методами белого ящика, имеют большую экспертизу в программировании, так как должны уметь читать код и находить в нем проблемы.

Преимущества:

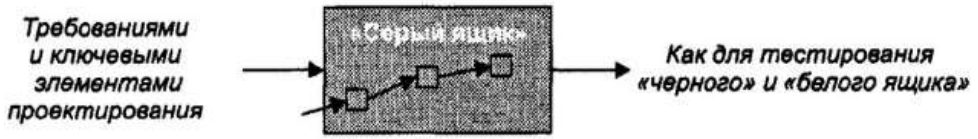
- Показывает скрытые проблемы и упрощает их диагностику.
- Допускает достаточно простую автоматизацию тест-кейсов и их выполнение на самых ранних стадиях развития проекта.
- Обладает развитой системой метрик, сбор и анализ которых легко автоматизируется.
- Стимулирует разработчиков к написанию качественного кода.
- Многие техники этого метода являются проверенными, хорошо себя зарекомендовавшими решениями, базирующимися на строгом техническом подходе.

Недостатки:

- Не может выполняться тестировщиками, не обладающими достаточными знаниями в области программирования.
- Тестирование сфокусировано на реализованной функциональности, что повышает вероятность пропуска нереализованных требований.
- Поведение приложения исследуется в отрыве от реальной среды выполнения и не учитывает её влияние.
- Поведение приложения исследуется в отрыве от реальных пользовательских сценариев.

Серый ящик. Отдельный вид или миф?

Его основной посыл в том, что нам **известны только некоторые особенности** реализации тестируемой системы.



То есть, внутреннее устройство программы нам известно лишь частично. Предполагается, например, доступ к внутренней структуре и алгоритмам работы ПО для написания максимально эффективных тест-кейсов, но само тестирование проводится с помощью техники черного ящика, то есть, с позиции пользователя.

Эту технику тестирования также называют методом полупрозрачного ящика: что-то мы видим, а что-то – нет.

Кто-то говорит, что этот вид тестирования — это **симбиоз белого и черного ящика**. Кто-то **противопоставляет его белому и черному**, опираясь на то, что внутренняя структура тестируемого объекта изначально известна частично и выясняется по мере исследования.

ISTQB относит тестирование методами белого и черного ящика к **методам проектирования тестов**. Поэтому, ни о каком «среднем» или «промежуточном» методе в этом случае конечно и речи быть не может. Мы либо разрабатываем тесты, зная код, либо не зная его. То есть в классификации ISTQB такого вида тестирования не существует.

Почему все ящики эффективны?

Методы белого и черного ящика не являются конкурирующими или взаимоисключающими. Наоборот, они гармонично дополняют друг друга, компенсируя имеющиеся недостатки.

Параллельное использование черного и белого ящиков увеличивает покрытие возможных сценариев:

- количественно, потому что появляется большее количество тест-кейсов,
- качественно, потому что ПО тестируется принципиально разными подходами: с точки зрения пользователя ("Черный ящик") и с точки зрения “внутренностей”.

То есть у каждого из методов тестирования свои неоспоримые плюсы, которые помогают выпустить качественный продукт.

Напомню, что для каждого ящика тестирования есть свой набор техник тест-дизайна. Ведь это не просто виды тестирования, а методы проектирования тестов. Почитать об этих техниках можно тут https://vk.com/@zapiskisedogotestera-obzor-tehnik-test-dizaina

Тестирование восстановления

Тестирование на отказ и восстановление проверяет тестируемый продукт на возможность противостоять и успешно восстанавливаться после возможных сбоев, возникших в связи с ошибками программного обеспечения, отказами оборудования или проблемами связи.

Целью данного тестирования является проверка возможности продолжить выполнение операций после происшествия или после нарушения целостности системы. Оно включает в себя откат до точки, где целостность системы была определена.

Пример

Допустим, приложение работает с сетью (принимает данные). В это время мы прервем соединение с сетью. После определенного времени восстановим соединение и проанализируем возможность приложения получать данные с момента, когда соединение было прервано.

Другим примером является перезапуск системы во время работы с браузером. Можно проверить, сможет ли браузер восстановить своё предыдущее состояние(вкладки) до экстренного закрытия.

Тестирование восстановления является нефункциональным тестированием. Под нефункциональным тестированием понимается тестирование аспектов ПО, которые могут быть не связаны с определенной функцией или действием пользователя.

Время, затрачиваемое на тестирование, зависит от:

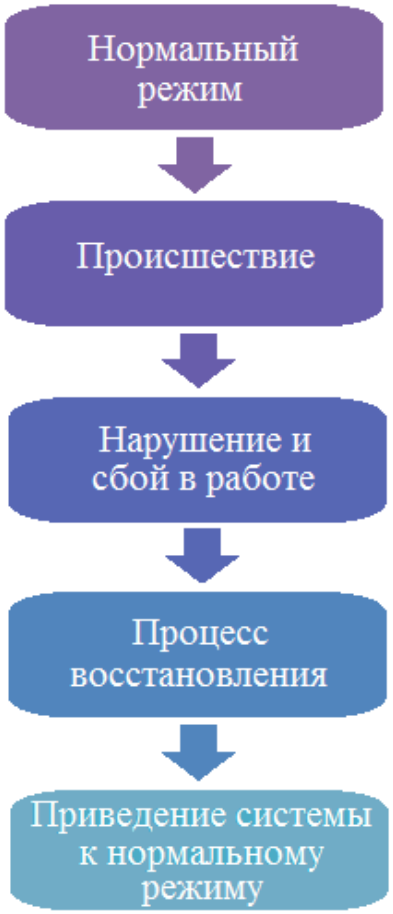
- количества точек восстановления;
- объем системы;
- навыков и умений персонала, ответственного за восстановление;
- доступных инструментов.

В случае, если в заданный момент времени присутствует множество ошибок, тестирование восстановления должно быть выполнено в структурированном виде, т.е. восстанавливаться должен сначала один сегмент, затем второй и т.д.

Цикл жизни

Цикл жизни процесса восстановления может быть описан следующими пятью шагам:

1. Нормальный режим работы.
2. Происшествие.
3. Нарушение и сбой в работе.
4. Процесс восстановления.
5. Приведение системы к нормальному режиму.



Цикл жизни процесса восстановления

Обсудим эти 5 шагов цикла жизни процесса восстановления более детально:

1. Система, состоящая из программного обеспечения, аппаратного обеспечения и встроенного программного обеспечения, объединенных для достижения общей цели, функционирует ради выполнения явно определенной цели. Система работает в нормальном режиме, т. е. выполняет запланированную работу без каких-либо нарушений в течение определенного периода времени.
2. Под происшествием понимается неисправность ПО, ошибочные входные данные, аппаратные ошибки, физические нарушения (огонь, удар).
3. Ошибка выполнения является наиболее тяжелым этапом, по причине которого происходят бизнес потери, разрыв отношений, потеря возможностей, потеря человеко-часов, а также неизбежные финансовые и гудвилл потери.
4. Если бекап и план снижения рисков корректно определенны вышеописанные последствия можно свести к минимуму.

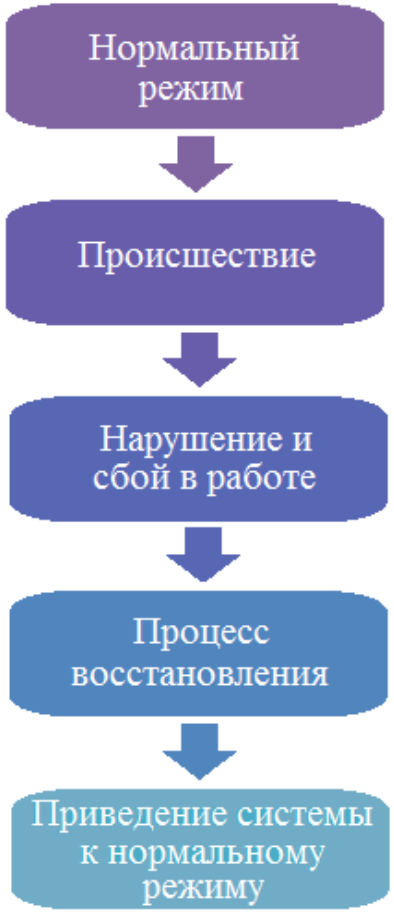


Рисунок 1 – Цикл жизни

5. Восстановление может включать в себя множество сессий для реконструирования всех папок и конфигурационных файлов.

Стратегия восстановления

Лицо, ответственное за восстановление, должно разработать свою стратегию (план) восстановления, которая обеспечит извлечение релевантной информации и данных для последующего приведения системы к нормальному состоянию.

Примеры существующих стратегий:

- Создать один или более бэкапов (резервных копий).
- Хранить бэкапы на разных носителях.
- Онлайн или офлайн бэкап.
- Создание бэкапов автоматически, либо вручную.
- Создание независимой ответственной команды, или переложить ответственность на команду разработчиков.

Выбор конкретной стратегии восстановления связан в первую очередь со стоимостью, а также критичностью информации.

Как проводить тестирование восстановления

При проведении тестирования следующие факторы должны быть учтены:

- Необходимо максимально точно приблизить условия тестирования к реальным. Изменения в интерфейсе, протоколах, прошивке, аппаратном обеспечении, программном обеспечении должны быть приближены к реальным примерам.
- Несмотря на то, что исчерпывающее тестирование является крайне затратным тестированием, оно все равно должно быть выполнено.
- Если возможно, следует провести тестирование на оборудовании, которые мы собираемся восстановить. Это особенно важно, в случае если создание копий и восстановление происходит на разных компьютерах.
- Некоторые системы ожидают неизменность размера жесткого диска.
- Облачные системы резервного копирования также должны быть протестированы.

Тестирование надежности

Тестирование надежности является одним из видов нефункционального тестирования, целью которого является проверка работоспособности программы в течение длительного периода времени, используя весь спектр своих возможностей, работая без сбоев и не вызывая сбоев.

Тестирование надежности проводится для проверки эффективности разработанного продукта при условиях, превышающих нормальные, часто доходящие до точки остановка (breakpoint). При таком тестировании следует обращать внимание на обработку ошибок, надежность продукта, масштабируемость при большой нагрузке, а не на поведение системы при нормальных обстоятельствах.

Цикл жизни

1. Планирование. На этом создается методология тестирования, которая включает в себя следующие элементы:
   - Сбор и анализ статистики среды.
   - Согласование требований к производительности.
   - Определение сценариев нагрузки.
   - Определение тестируемых компонентов.
   - Описание взаимодействия с внешними системами.
   - Описание требований к БД.
2. Создание тестовой модели подразумевает:
   - Разработка нагрузочных скриптов.
   - Разработка эмуляторов.
3. Создание сценариев нагрузки:
   - Создание сценариев для генерации БД.
   - Руководство по проведению теста.

Рисунок 2 – Цикл жизни тестирования

4. Обзор и подготовка к тестированию:
    - Проверка работоспособности среды.
    - Установка инструментов тестирования.
    - Настройка инструментов мониторинга.
    - Проведение пробных испытаний.
5. Проведение тестирования:
    - Запуск тестов в соответствии со сценарием тестирования.
    - Выключение/перезапуск выбранных компонентов.
    - Проверка отказоустойчивости системы.
    - Анализ результатов.
6. Создание отчета:
    - Анализ узких мест.
    - Анализ влияния работоспособности компонентов системы.
    - Анализ времени восстановления системы.
    - Подготовка рекомендаций.

Необходимость тестирования надежности

Данный вид тестирования помогает понять, как система будет работать в реальных условиях.

Плюсы проведения тестирования надежности:

- Показывает объем данных, которые система может обработать.
- Обеспечивает уверенность в производительности системы.
- Определяет стабильность системы под нагрузкой.
- Тестирование стабильности помогает улучшить взаимодействие с конечным пользователем.

В то же время, если не проводить тестирование стабильности, это может привести к следующим последствиям:

- Система рушится неожиданно.
- Система работает медленней с большими объемами данных.
- Неопределенность работы системы в различных условиях.

Как проводить тестирование надежности

Рассмотрим данное тестирование на примере смартфона. Когда в телефон загружено последнее разработанное программное обеспечение, сначала проверяется загрузка, а затем проводится регрессионное или дымовое тестирование. После прохождения первого уровня тестирования проводится функциональное и нефункциональное тестирование. Функциональное включает выполнение всех тестовых случаев, связанных с функциональностью, а нефункциональное тестирование или тестирование производительности включает нагрузочное тестирование, стресс-тестирование, тестирование стабильности и тестирование надежности.

Следует обратить внимание на следующие аспекты:

- Система проверяется на производительность при заполнении памяти на 60% данными.
- Память заполняется на 80% и проводится повторное тестирование.
- С помощью инструмента Load Runner проверяется загрузка системы и её стабильность.
- Написание автоматических сценариев для использования всех ресурсов системы и проверки её работоспособности в данных условиях. К примеру, щелчок по конкретной кнопке 100 раз, отправка 1000 запросов системе, нажатие случайных кнопок, открытие и закрытие приложений и т. д.
- Также проверяется время автономной работы и работоспособность батареи. Иногда, загруженная прошивка потребляет неожиданно много энергии.

Отчетность

Для эффективного тестирования, во время выполнения теста собираются и измеряются статистические данные, которые позднее анализируются с целью создания отчета и выявления возможных проблем.

Как правило собираются данные по следующим параметрам:

- Время отклика: среднее время, необходимое для выполнения транзакций. По этой статистике может оценить находится ли производительность сервера в пределах допустимых значений.
- Число посещений в секунду: количество обращений к серверу. Полезно при определении генерируемой нагрузки, относительно количества обращений.
- Пропускная способность: объем информации, обрабатываемой сервером, измеряется в байтах. Помогает оценить объем генерируемой нагрузки в любой момент времени.
- Количество транзакций в секунду: общее число завершенных транзакций (успешных и неудачных), выполненных во время теста. Показывает фактическую нагрузку системы.
- CPU: процент использования процессора.
- RAM: использование памяти во время теста.
- Память: использование дискового пространства.

Конечный отчет включает в себя следующие пункты:

- Информацию о количестве найденных дефектов, обнаруженных в системе, после выхода из строя определенного компонента.
- Список и описание дефектов после выхода определенного компонента.
- Информация о необходимом времени восстановления системы.
- Рекомендации по изменению архитектуры системы.
- Описание нагрузочных профилей.
- Тип тестовых данных.
- Скрипт тестирования.
- Эмулятор внешней среды.
- Скрипт для генерации БД.
- Руководство по проведению тестирования.

Заключение

Таким образом, эти два тестирования являются крайне важным этапом в процессе улучшения качества опыта конечного пользователя. Критически важно обеспечить стабильность и надежность работы системы, иначе большинство пользователей перейдет к более стабильным альтернативам. Другим важным аспектом является возможность обеспечить возможность продолжить работу, даже после краха системы.

ЛЕКЦИЯ 3

ТЕМА 3: Тестирование “белого ящика”. Особенности тестирования “белого ящика”. Способы структурного тестирования. Способ тестирования базового пути. Потоковый граф. Цикломатическая сложность. Примеры

Известна: внутренняя структура программы.

Исследуются: внутренние элементы программы и связи между ними (рисунке 3.1).

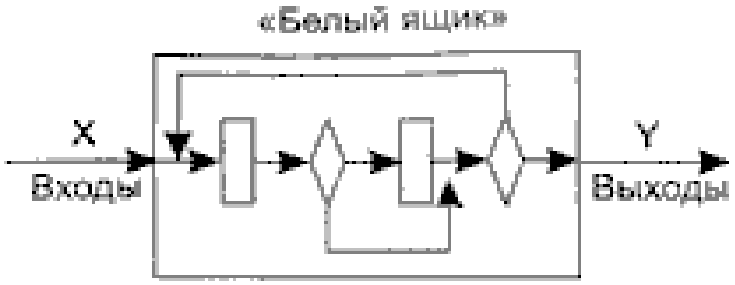


Рисунок 3.1 - Тестирование «белого ящика»

Объектом тестирования здесь является не внешнее, а внутреннее поведение программы. Проверяется корректность построения всех элементов программы и правильность их взаимодействия друг с другом. Обычно анализируются управляющие связи элементов, реже — информационные связи. Тестирование по принципу «белого ящика» характеризуется степенью, в какой тесты выполняют или покрывают логику (исходный текст) программы. Исчерпывающее тестирование также затруднительно. Особенности этого принципа тестирования рассмотрим отдельно.

Особенности тестирования «белого ящика»

Обычно тестирование «белого ящика» основано на анализе управляющей структуры программы. Программа считается полностью проверенной, если проведено исчерпывающее тестирование маршрутов (путей) ее графа управления.

В этом случае формируются тестовые варианты, в которых:

- гарантируется проверка всех независимых маршрутов программы;
- проходятся ветви True, False для всех логических решений;
- выполняются все циклы (в пределах их границ и диапазонов);
- анализируется правильность внутренних структур данных.

Недостатки тестирования «белого ящика»:

– Количество независимых маршрутов может быть очень велико. Например, если цикл в программе выполняется *k* раз, а внутри цикла имеется *n* ветвлений, то количество маршрутов вычисляется по формуле:

$$m = \sum_{i=1}^{k} n^i = 10^{14} = O(5^{20})$$

При $n = 5$ и $k = 20$ количество маршрутов $m = 10^{14}$. Примем, что на разработку, выполнение и оценку теста по одному маршруту расходуется 1 мс. Тогда при работе 24 часа в сутки 365 дней в году на тестирование уйдет 3170 лет.

2. Исчерпывающее тестирование маршрутов не гарантирует соответствия программы исходным требованиям к ней.

3. В программе могут быть пропущены некоторые маршруты.

4. Нельзя обнаружить ошибки, появление которых зависит от обрабатываемых данных (это ошибки, обусловленные выражениями типа if abs (a-b) < eps..., if (a+b+c)/3=a...)-.

Достоинства тестирования «белого ящика» связаны с тем, что принцип «белого ящика» позволяет учесть особенности программных ошибок:

1. Количество ошибок минимально в «центре» и максимально на «периферии» программы.

2. Предварительные предположения о вероятности потока управления или данных в программе часто бывают некорректны. В результате типовым может стать маршрут, модель вычислений по которому проработана слабо.

3. При записи алгоритма ПО в виде текста на языке программирования возможно внесение типовых ошибок трансляции (синтаксических и семантических).

4. Некоторые результаты в программе зависят не от исходных данных, а от внутренних состояний программы.

Каждая из этих причин является аргументом для проведения тестирования по принципу «белого ящика». Тесты «черного ящика» не смогут реагировать на ошибки таких типов.

Методы структурного тестирования:

- **Способ тестирования базового пути** проверяет, что каждый оператор в программе выполняется хотя бы один раз во время тестирования программы.
- **Способ тестирование ветвей и операторов отношений** предназначен для проверки логических условий программы. При этом желательно обеспечить охват операторов из всех ветвей программы.
- **Способ тестирование потоков данных**, подвергается анализу информационная структура программы.
- **Тестирование циклов**, основное внимание обращается на правильность конструкций циклов.

Способ тестирования базового пути

Тестирование базового пути — это способ, который основан на принципе «белого ящика». Автор этого способа — Том МакКейб (1976).

Способ тестирования базового пути даёт возможность:

- получить оценку комплексной сложности программы;
- использовать эту оценку для определения необходимого количества тестовых вариантов.

Тестовые варианты разрабатываются для проверки базового множества путей (маршрутов) в программе. Они гарантируют однократное выполнение каждого оператора программы при тестировании.

Потоковый граф

Для представления программы используется потоковый граф. Перечислим его особенности.

1. Граф строится отображением управляющей структуры программы. В ходе отображения закрывающие скобки условных операторов и операторов циклов (end if; end loop) рассматриваются как отдельные (фиктивные) операторы.
2. Узлы (вершины) потокового графа соответствуют линейным участкам программы, включают один или несколько операторов программы.
3. Дуги потокового графа отображают поток управления в программе (передачи управления между операторами). Дуга — это ориентированное ребро.
4. Различают операторные и предикатные узлы. Из операторного узла выходит одна дуга, а из предикатного — две дуги.
5. Предикатные узлы соответствуют простым условиям в программе. Составное условие программы отображается в несколько предикатных узлов. Составным называют условие, в котором используется одна или несколько булевых операций (OR, AND).

Например,

фрагмент программы

```
if a OR b
   then x
   else y
end if;
```

вместо прямого отображения в потоковый граф вида, показанного на рисунке 3.2, отображается в преобразованный потоковый граф (рисунке 3.3).

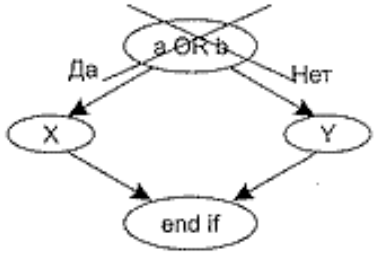


Рисунок 3.2 - Прямое отображение в потоковый граф

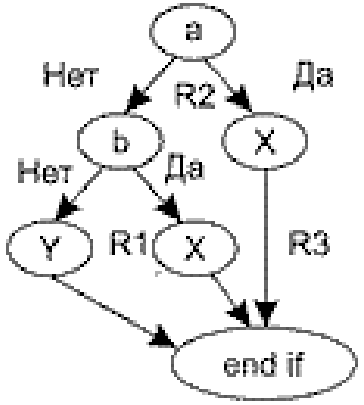


Рисунок 3.3 - Преобразованный потоковый граф

6. Замкнутые области, образованные дугами и узлами, называют регионами.
7. Окружающая граф среда рассматривается как дополнительный регион. Например, показанный здесь граф имеет три региона — Rl, R2, R3.

Пример 1. Рассмотрим процедуру сжатия:

```
процедура сжатие
1           выполнять пока нет EOF
1               читать запись;
2               если запись пуста
3                   то удалить запись:
4                   иначе если поле a ≥ поля b
5                           то удалить b;
6                           иначе удалить a;
7a                      конец если;
7a              конец если;
7b          конец выполнять;
8       конец сжатие;
```

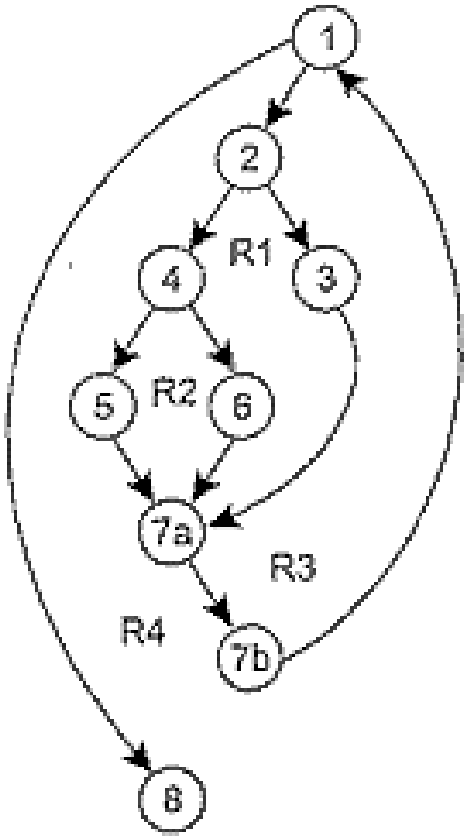


Рисунок 3.4 - Преобразованный потоковый граф процедуры сжатия

Она отображается в потоковый граф, представленный на рисунке 3.4. Видим, что этот потоковый граф имеет четыре региона.

Цикломатическая сложность

Цикломатическая сложность — метрика ПО, которая обеспечивает количественную оценку логической сложности программы. В способе тестирования базового пути Цикломатическая сложность определяет:

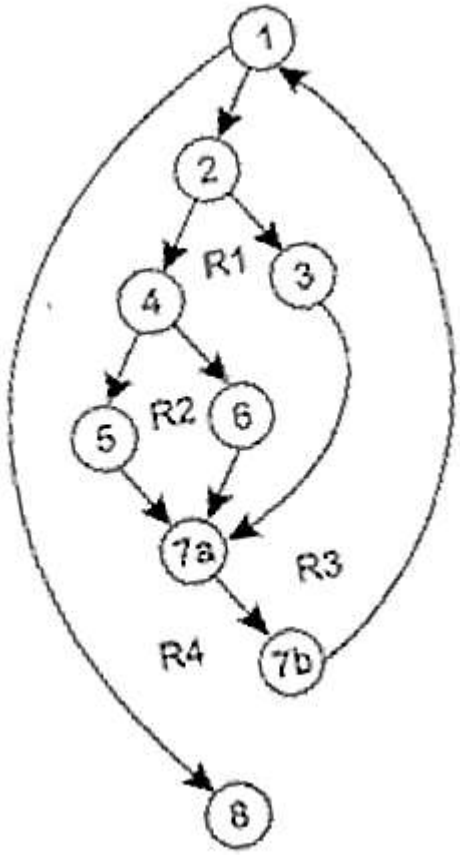
- количество независимых путей в базовом множестве программы;
- верхнюю оценку количества тестов, которое гарантирует однократное выполнение всех операторов.

Независимым называется любой путь, который вводит новый оператор обработки или новое условие. В терминах потокового графа независимый путь должен содержать дугу, не входящую в ранее определенные пути.

ПРИМЕЧАНИЕ

Путь начинается в начальном узле, а заканчивается в конечном узле графа. Независимые пути формируются в порядке от самого короткого к самому длинному.

Перечислим независимые пути для потокового графа из примера 1:



Путь 1:1-8.

Путь 2: 1-2-3-7a-7b-1-8.

Путь 3: 1-2-4-5-7a-7b-1-8.

Путь 4: 1-2-4-6-7a-7b-1-8.

Заметим, что каждый новый путь включает новую дугу.

Все независимые пути графа образуют базовое множество.

Свойства базового множества:

- тесты, обеспечивающие его проверку, гарантируют:
  - однократное выполнение каждого оператора;
  - выполнение каждого условия по True-ветви и по False-ветви;
- мощность базового множества равна цикломатической сложности потокового графа.

Значение 2-го свойства трудно переоценить — оно дает априорную оценку количества независимых путей, которое имеет смысл искать в графе.

Цикломатическая сложность вычисляется одним из трех способов:

1) цикломатическая сложность равна количеству регионов потокового графа;

2) цикломатическая сложность определяется по формуле:

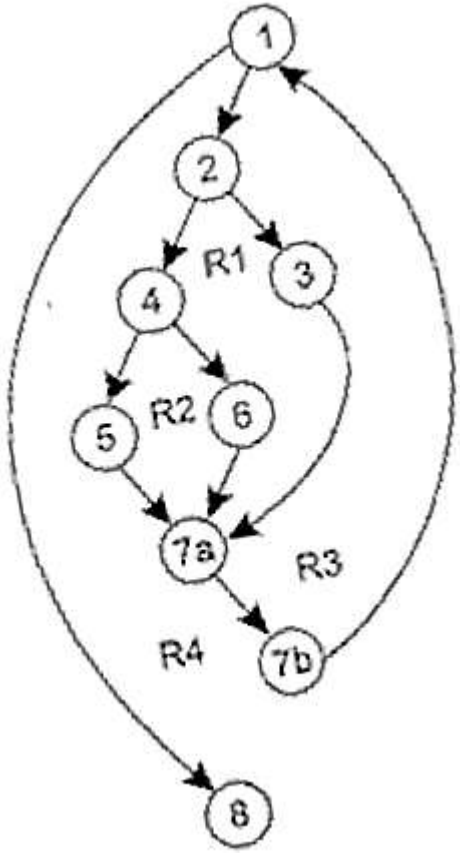
$$V(G) = E - N + 2,$$

где E — количество дуг, N — количество узлов потокового графа;

3) цикломатическая сложность формируется по выражению

$$V(G) = p + 1,$$

где p — количество предикатных узлов в потоковом графе G.



Вычислим цикломатическую сложность графа из примера 1 каждым из трех способов:

1) потоковый граф имеет 4 региона;

2) V(G) = 11 дуг - 9 узлов + 2 = 4;

3) V(G) = 3 предикатных узла +1 = 4.

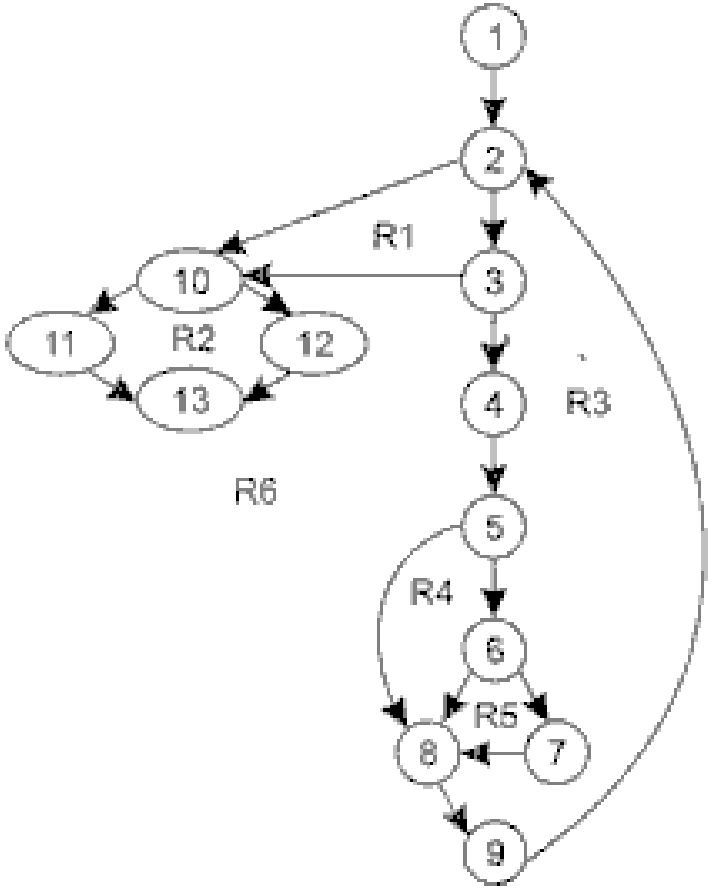
Таким образом, цикломатическая сложность потокового графа из примера 1 равна четырем.

Шаги способа тестирования базового пути

Для иллюстрации шагов данного способа используем конкретную программу:

ПРОЦЕДУРА СРЕД:

```
1  i ← 1
1  введено ← 0
1  колич ← 0
1  сум ← 0
   While  [2]ВЕЛ(i) <> stop & [3]введено ≤ 500
4              do  введено ← введено + 1
                   if  [5]ВЕЛ(i) ≥ мин & [6]ВЕЛ(i) ≤ макс
7                       then колич ← колич + 1
7                       сум ← сум + ВЕЛ(i)
8                  end if
8                  i := i +1
9  end loop
10 if колич > 0
11      then сред ← сум / колич
12      else сред ← stop
13 end if
13 end сред
```



Заметим, что процедура содержит составные условия (в заголовке цикла и условном операторе). Элементы составных условий для наглядности помещены в рамки.

Шаг 1. На основе текста программы формируется потоковый граф:

- нумеруются операторы текста (номера операторов показаны в тексте процедуры);
- производится отображение пронумерованного текста программы в узлы и вершины потокового графа (рисунок 3.5).

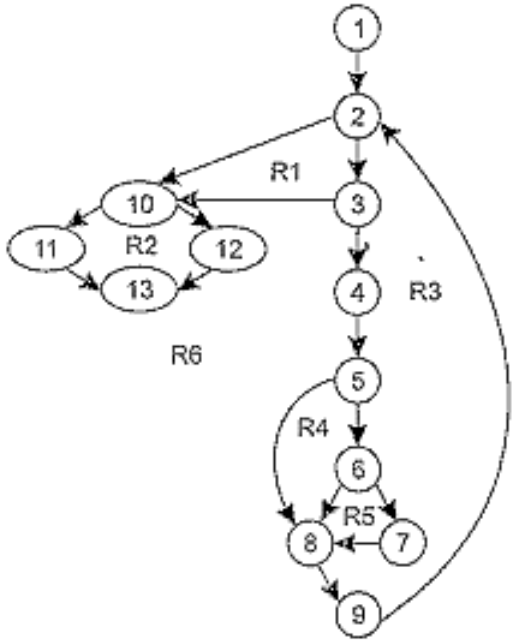


Рисунок 3.5 - Потоковый граф процедуры вычисления среднего значения

Шаг 2. Определяется цикломатическая сложность потокового графа — по каждой из трех формул:

1) *V(G)* = 6 регионов;
2) *V(G)* = 17 дуг - 13 узлов + 2 = 6;
3) *V(G)* = 5 предикатных узлов +1=6.

```
ПРОЦЕДУРА СРЕД:
1  i ← 1
1  введено ← 0
1  колич ← 0
1  сум ← 0
   While ²ВЕЛ(i) <> stop & ³введено ≤ 500
4            do  введено ← введено + 1
                 if ⁵ВЕЛ(i) ≥ мин & ⁶ВЕЛ(i) ≤ макс
7                    then колич ← колич + 1
7                    сум ← сум + ВЕЛ(i)
8                end if
8                i := i +1
9  end loop
10 if колич > 0
11    then сред ← сум / колич
12    else сред ← stop
13 end if
13 end сред
```

Шаг 3. Определяется базовое множество независимых линейных путей:

Путь 1: 1-2-10-11-13; /вел=stop, колич>0.

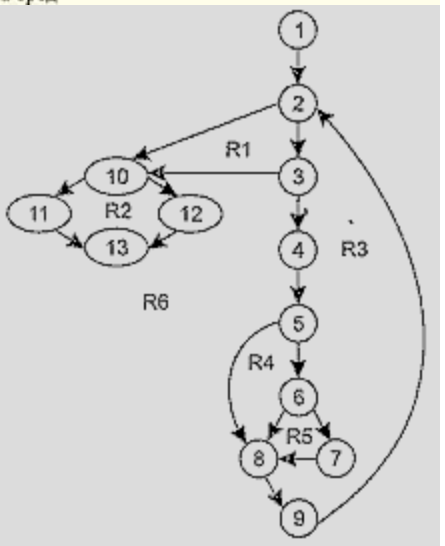
Путь 2: 1-2-10-12-13; /вел=stop, колич=0.

Путь 3: 1-2-3-10-11-13; /попытка обработки 501-й величины.

Путь 4: 1-2-3-4-5-8-9-2-... /вел<мин.

Путь 5: 1-2-3-4-5-6-8-9-2-... /вел>макс.

Путь 6: 1-2-3-4-5-6-7-8-9-2-... /режим нормальной обработки.



ПРИМЕЧАНИЕ

Для удобства дальнейшего анализа по каждому пути указаны условия запуска. Точки в конце путей 4,5,6 указывают, что допускается любое продолжение через остаток управляющей структуры графа.

Шаг 4. Подготавливаются тестовые варианты, инициирующие выполнение каждого пути.

```
ПРОЦЕДУРА СРЕД:
1  i ← 1
1  введено ← 0
1  колич ← 0
1  сум ← 0
   While ²ВЕЛ(i) <> stop & ³введено ≤ 500
4            do введено ← введено + 1
                 if ⁵ВЕЛ(i) ≥ мин & ⁶ВЕЛ(i) ≤ макс
7                   then колич ← колич + 1
7                   сум ← сум + ВЕЛ(i)
8                end if
8                i := i +1
9  end loop
10 if колич > 0
11    then сред ← сум / колич
12    else сред ← stop
13 end if
13 end сред
```

Каждый тестовый вариант формируется в следующем виде:

Исходные данные (ИД):

Ожидаемые результаты (ОЖ.РЕЗ.):

Исходные данные должны выбираться так, чтобы предикатные вершины обеспечивали нужные переключения — запуск только тех операторов, которые перечислены в конкретном пути, причем в требуемом порядке.

Определим тестовые варианты, удовлетворяющие выявленному множеству независимых путей.

Тестовый вариант для пути 1 **ТВ1**:

Путь 1: 1-2-10-11-13; /вел=stop, колич>0.

ИД: вел(k) = допустимое значение, где $k < i$; вел(k) = stop, где $2 < i < 500$.

ОЖ.РЕЗ.: корректное усреднение основывается на k величинах и правильном подсчете.

ПРИМЕЧАНИЕ

Путь не может тестироваться самостоятельно, а должен тестироваться как часть путей 4, 5, 6 (трудности проверки 11 -го оператора).

Тестовый вариант для пути 2

```
ПРОЦЕДУРА СРЕД:
1  i ← 1
1  введено ← 0
1  колич ← 0
1  сум ← 0
   While  ²ВЕЛ(i) <> stop & ³введено ≤ 500
4             do  введено ← введено + 1
                  if  ⁵ВЕЛ(i) ≥ мин & ⁶ВЕЛ(i) ≤ макс
7                     then колич ← колич + 1
7                     сум ← сум + ВЕЛ(i)
8                 end if
8                 i := i +1
9  end loop
10 if колич > 0
11    then сред ← сум / колич
12    else сред ← stop
13 end if
13 end сред
```

ТВ2:

Путь 2: 1-2-10-12-13; /вел=stop, колич=0.

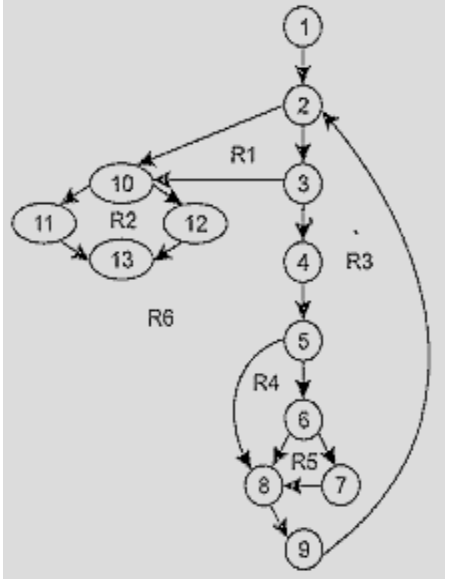
ИД: вел (1) = stop.

ОЖ.РЕЗ.: сред = stop, другие величины имеют начальные значения.

Тестовый вариант для пути 3

ТВ3:

Путь 3: 1-2-3-10-11-13; /попытка обработки 501-й величины



ИД: попытка обработки 501-й величины, первые 500 величин должны быть правильными.

ОЖ.РЕЗ.: корректное усреднение основывается на *k* величинах и правильном подсчете.

Тестовый вариант для пути 4

```
ПРОЦЕДУРА СРЕД:
1  i ← 1
1  введено ← 0
1  колич ← 0
1  сум ← 0
   While ²ВЕЛ(i) <> stop & ³введено ≤ 500
4            do  введено ← введено + 1
                 if ⁴ВЕЛ(i) ≥ мин & ⁵ВЕЛ(i) ≤ макс
7                    then колич ← колич + 1
7                    сум ← сум + ВЕЛ(i)
8                end if
8                i := i +1
9  end loop
10 if колич > 0
11    then сред ← сум / колич
12    else сред ← stop
13 end if
13 end сред
```

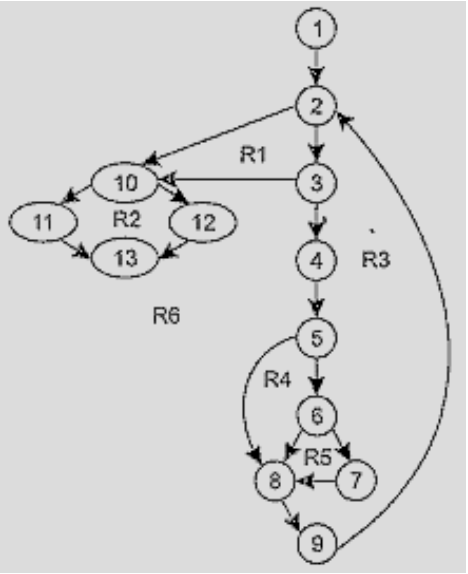
ТВ4:

Путь 4: 1-2-3-4-5-8-9-2-… /вел<мин.

ИД: вел(i) = допустимое значение, где $i \leq 500$; вел(k) < мин, где $k < i$.

ОЖ.РЕЗ.: корректное усреднение основывается на *k* величинах и правильном подсчете.

Тестовый вариант для пути 5



ТВ5:

Путь 5: 1-2-3-4-5-6-8-9-2-… /вел>макс.

ИД: вел(i) = допустимое значение, где $i \leq 500$; вел(k) > макс, где $k < i$.

ОЖ.РЕЗ.: корректное усреднение основывается на *n* величинах и правильном подсчете.

Тестовый вариант для пути 6

```
ПРОЦЕДУРА СРЕД:
1  i ← 1
1  введено ← 0
1  колич ← 0
1  сум ← 0
   While ²ВЕЛ(i) <> stop & ³введено ≤ 500
4          do введено ← введено + 1
              if ⁵ВЕЛ(i) ≥ мин & ⁶ВЕЛ(i) ≤ макс
7                then колич ← колич + 1
7                сум ← сум + ВЕЛ(i)
8             end if
8             i := i +1
9  end loop
10 if колич > 0
11    then сред ← сум / колич
12    else сред ← stop
13 end if
13 end сред
```

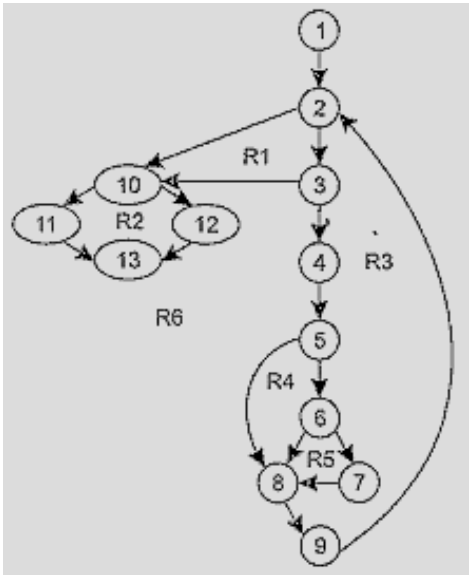
ТВ6:

Путь 6: 1-2-3-4-5-6-7-8-9-2-... /режим нормальной обработки

ИД: вел(i) = допустимое значение, где $i \leq 500$.

ОЖ.РЕЗ.: корректное усреднение основывается на *n* величинах и правильном подсчете.

Реальные результаты каждого тестового варианта сравниваются с ожидаемыми результатами. После выполнения всех тестовых вариантов гарантируется, что все операторы программы выполнены по меньшей мере один раз.



Важно отметить, что некоторые независимые пути не могут проверяться изолированно. Такие пути должны проверяться при тестировании другого пути (как часть другого тестового варианта).

Задания

1. Построить потоковый граф. Используя способ тестирования базового пути, составить тестовые варианты.

```
b=N
while (b ≠ 0)
      t=0
      for j=1 to b-1 do
        if (k[j] > k[j+1])
           k[j]<->k[j+1]
           t=j
        end if
      end for
      b=t
end loop
```

2. Построить потоковый граф. Используя способ тестирования базового пути, составить тестовые варианты.

```
нач
    цел i
    i := n
    нц пока (i >= 1) и (F[i] > D)
        F[i + 1] := F[i] |сдвиг очередного элемента вправо на одну позицию
        i := i − 1
    кц
    F[i + 1] := D
кон
```

ЛЕКЦИЯ 4

ТЕМА 4: Структурное тестирование программного продукта. Способы тестирования условий. Тестирование ветвей и операторов отношений. Примеры

Способы тестирования условий

Цель этого семейства способов тестирования — строить тестовые варианты для проверки логических условий программы. При этом желательно обеспечить охват операторов из всех ветвей программы.

Рассмотрим используемую терминологию.

Различают три типа условий:

- простое условие;
- составное условие;
- булевы выражения.

Простое условие — булева переменная или выражение отношения.

Выражение отношения имеет вид

E1 <оператор отношения> E2,

где El, E2 — арифметические выражения, а в качестве оператора отношения используется один из следующих операторов: <, >, =, != , <=, >=.

Составное условие состоит из:

- нескольких простых условий;
- булевых операторов (OR, AND (&), NOT);
- круглых скобок (заключающая простое или составное условие).

Булевы выражения - условия, не содержащие выражений отношения.

Таким образом, **элементами условия** являются:

- булев оператор (OR, AND (&), NOT);
- булева переменная (TRUE, FALSE);
- пара скобок;
- оператор отношения (<, >, =, != , <=, >=);
- арифметическое выражение (E1, E2, …, En).

Элементы условия определяют типы ошибок в условиях.

Если условие некорректно, то некорректен по меньшей мере один из элементов условия. Следовательно, в условии возможны следующие **типы ошибок:**

– ошибка булева оператора (наличие некорректных / отсутствующих / избыточных булевых операторов);

– ошибка булевой переменной;

– ошибка булевой скобки;

– ошибка оператора отношения;

– ошибка арифметического выражения.

Способ тестирования условий ориентирован на тестирование каждого условия в программе.

Методики тестирования условий имеют следующие **достоинства:**

- ✓ достаточно просто выполнить измерение тестового покрытия условия.
- ✓ тестовое покрытие условий в программе — это фундамент для генерации дополнительных тестов программы.
- ✓ эффективна не только для обнаружения ошибок в условии, но и для обнаружения ошибок в программе.

Целью тестирования условий является определение не только ошибок в условиях, но и других ошибок в программах.

Методики тестирования условий

Тестирование ветвей (простейшая методика)

Здесь для составного условия ***C*** проверяется:

– каждое простое условие (входящее в него);

– True-ветвь;

– False-ветвь.

Тестирование области определения

Для выражения отношения требуется генерация 3–4 тестов. Выражение вида:

$$E_1 \text{ <оператор отношения> } E_2$$

проверяется тремя тестами, которые формируют:

– значение E_1 большим, чем E_2 ($E_1 > E_2$);
– значение E_1 равным E_2 ($E_1 = E_2$);
– значение E_1 меньшим, чем E_2 ($E_1 < E_2$).

Если оператор отношения неправилен, а E_1 и E_2 корректны, то эти три теста гарантируют обнаружение ошибки оператора отношения.

Для определения ошибок в E_1 и E_2 тест должен сформировать значение E_1 большим или меньшим, чем E_2, причем обеспечить как можно меньшую разницу между этими значениями.

Для булевых выражений с ***n*** переменными требуется набор из $\mathbf{2^n}$ тестов. Этот набор позволяет обнаружить ошибки булевых операторов, переменных и скобок, но практичен только при малом ***n***. Впрочем, если в булево выражение каждая булева переменная входит только один раз, то количество тестов легко уменьшается.

Рассмотрим **способ тестирования условий**, базирующийся на приведенных выше методиках.

Тестирование ветвей и операторов отношений

Способ тестирования ветвей и операторов отношений (автор К. Таи, 1989) обнаруживает ошибки ветвления и операторов отношения в условии, для которого выполняются следующие ограничения:

– все булевы переменные и операторы отношения входят в условие только по одному разу;
– в условии нет общих переменных.

В данном способе используются естественные ограничения условий (ограничения на результат). Для составного условия ***C***, включающего ***n*** простых условий, формируется **ограничение условия**:

$$OY_C = (d_1, d_2, d_3, ..., d_n),$$

где *di* — ограничение на результат i-го простого условия.

Ограничение на результат фиксирует возможные значения аргумента (переменной) простого условия (если он один) или соотношения между значениями аргументов (если их несколько).

Если i-е простое условие является **булевой переменной**, то его ограничение на результат состоит из двух значений и имеет вид:

$$di := (\text{true, false}).$$

Если j-е простое условие является **выражением отношения**, то его ограничение на результат состоит из трех значений и имеет следующий вид:

$$d_j := (>, <, =).$$

Говорят, что ограничение условия $\mathbf{ОУ_C}$ (для условия **C**) покрывается выполнением условия ***C***, если в ходе этого выполнения результат каждого простого условия в ***C*** удовлетворяет соответствующему ограничению в $\mathbf{ОУ_C}$.

На основе ограничения условия **ОУ** создается **ограничивающее множество ОМ**, элементы которого являются сочетаниями всех возможных значений $d_1, d_2, d_3, ..., d_n$.

Ограничивающее множество — удобный инструмент для записи задания на тестирование, ведь оно составляется из сведений о значениях переменных, которые влияют на значение проверяемого условия.

Пример. Положим, надо проверить условие, составленное из трех простых условий:

$$b \ \& \ (x > y) \ \& \ a.$$

Условие принимает истинное значение, если все простые условия истинны. В **терминах значений простых условий** это соответствует записи:

(true, true, true),

а в **терминах ограничений** на значения аргументов простых условий — записи:

(true, >, true).

Вторая запись является прямым руководством для написания теста. Она указывает, что переменная ***b*** должна иметь истинное значение, значение переменной ***x*** должно быть больше значения переменной ***y,*** и переменная ***a*** должна иметь истинное значение.

Вывод: каждый элемент **ОМ** задает отдельный тестовый вариант. Исходные данные тестового варианта должны обеспечить соответствующую комбинацию значений простых условий, а ожидаемый результат равен значению составного условия.

Пример 1. Рассмотрим два типовых составных условия:

$$C_{\&} = a \ \& \ b,$$

$$C_{OR} = a \ OR \ b,$$

где a и b — булевы переменные.

Соответствующие ограничения условий принимают вид:

$$ОУ_{\&} = (d_1, d_2),$$

$$ОУ_{OR} = (d_1, d_2),$$

где $d_1 = d_2$ = (true, false).

Ограничивающие множества удобно строить с помощью таблицы истинности (таблица 4.1).

Видим, что таблица задает в **ОМ** четыре элемента (и соответственно, четыре тестовых варианта). Зададим вопрос — каковы возможности минимизации? Можно ли уменьшить количество элементов в **ОМ**?

С точки зрения тестирования, необходимо оценивать влияние составного условия на программу. Составное условие может принимать только два значения, но каждое из значений зависит от большого количества простых условий. Стоит задача — избавиться от влияния избыточных сочетаний значений простых условий.

Воспользуемся идеей сокращенной схемы вычисления — элементы выражения вычисляются до тех пор, пока они влияют на значение выражения.

Таблица 4.1 - Таблица истинности логических операций

Вариант	**Исходные данные**		**Ожидаемый результат**	
	a	**b**	**a & b**	**a OR b**
1	false	false	false	false
2	false	true	false	true
3	true	false	false	true
4	true	true	true	true

При тестировании необходимо выявить ошибки переключения, то есть ошибки из-за булева оператора, оперируя значениями простых условий (булевых переменных). При таком инженерном подходе справедливы следующие выводы:

– для условия типа И *(a & b)* варианты 2 и 3 поглощают вариант 1. Поэтому ограничивающее множество имеет вид:

$OM_{\&}$ = {(false, true), (true, false), (true, true)};

– для условия типа ИЛИ *(a OR b)* варианты 2 и 3 поглощают вариант 4. Поэтому ограничивающее множество имеет вид:

OM_{OR} = {(false, false), (false, true), (true, false)}.

Вариант	**Исходные данные**		**Ожидаемый результат**	
	a	**b**	**a & b**	**a OR b**
1	false	false	false	false
2	false	true	false	true
3	true	false	false	true
4	true	true	true	true

Рассмотрим **шаги способа тестирования ветвей и операторов отношений**.

Для каждого условия в программе выполняются следующие действия:

Шаг 1. Строится ограничение условий **ОУ**.

Шаг 2. Выявляются ограничения результата по каждому простому условию.

Шаг 3. Строится ограничивающее множество **ОМ**. Построение выполняется путем подстановки в константные формулы $\mathbf{ОМ_{\&}}$ или $\mathbf{ОМ_{OR}}$ выявленных ограничений результата.

Шаг 4. Для каждого элемента **ОМ** разрабатывается тестовый вариант.

Пример 2. Рассмотрим составное условие C_1 вида:

$$B_1 \ \& \ (E_1 = E_2),$$

где B_1 — булево выражение, E_1, E_2 — арифметические выражения.

Ограничение составного условия имеет вид:

$$ОУ_{C1} = (d_1, d_2),$$

где ограничения простых условий равны:

$$d_1 = (\text{true, false}),\ d_2 = (=, <, >).$$

Проводя аналогию между C_1 и $C_{\&}$ (разница лишь в том, что в C_1 второе простое условие — это выражение отношения), можно построить ограничивающее множество для C_1 модификацией:

$$ОМ_{\&} = \{(\text{false, true}), (\text{true, false}), (\text{true, true})\}.$$

Заметим, что *true* для $(E_1 = E_2)$ означает =, а false для $(E_1 = E_2)$ означает или <, или >. Заменяя (true, true) и (false, true), ограничениями (true, =) и (false, =) соответственно, а (true, false) — ограничениями (true, <) и (true, >), получаем ограничивающее множество для C_1:

$$ОМ_{C1} = \{(\text{false}, =), (\text{true}, <), (\text{true}, >), (\text{true}, =)\}.$$

Покрытие этого множества гарантирует обнаружение ошибок булевых операторов и операторов отношения в C_1.

Пример 3. Рассмотрим составное условие C_2 вида:

$$(E_3 > E_4) \ \& \ (E_1 = E_2),$$

где E_1, E_2, E_3, E_4 — арифметические выражения.

Ограничение составного условия имеет вид:

$$ОУ_{C2} = (d_1, d_2),$$

где ограничения простых условий равны:

$$d_1 = (=, <, >), \ d_2 = (=, <, >).$$

Проводя аналогию между C_2 и C_1 (разница лишь в том, что в C_2 первое простое условие — это выражение отношения), можно построить ограничивающее множество для C_2 модификацией $ОМ_{C1}$:

$$ОМ_{C2} = \{(=, =), (<, =), (>, <), (>, >), (>, =)\}.$$
$$0\ \ 1 \quad 0\ 1 \quad 1\ 0 \quad 1\ 0 \quad 1\ \ 1$$

Покрытие этого ограничивающего множества гарантирует обнаружение ошибок операторов отношения в C_2.

```
using System;

public class Program
{
    private static bool A()
    {
        Console.WriteLine("A");
        return true;
    }

    private static bool B()
    {
        Console.WriteLine("B");
        return true;
    }

    private static bool C()
    {
        Console.WriteLine("C");
        return true;
    }

    private static bool D()
    {
        Console.WriteLine("D");
        return false;
    }

    public static void Main()
    {
        Console.WriteLine("Test 1");
        var x = A() || B() && C();

        Console.WriteLine("Test 2");
        var y = A() | B() && C();

        Console.WriteLine("Test 3");
        var z = A() | B() & C();

        Console.WriteLine("Test 4");
        var v = A() | D() && C();

        Console.WriteLine("Test 5");
        var w = A() | D() & C();
    }
}
```

```
Test 1
A
Test 2
A
B
C
Test 3
A
B
C
Test 4
A
D
C
Test 5
A
D
C
```

Задания

1. Построить потоковый граф. Используя способ тестирования ветвей и операторов отношений, составить тестовые варианты.

```
b=N
while (b ≠ 0)
        t=0
        for j=1 to b-1 do
                    if (k[j] > k[j+1])
                    k[j]<->k[j+1]
                    t=j
                end if
        end for
        b=t
end loop
```

2. Построить потоковый граф. Используя способ тестирования ветвей и операторов отношений, составить тестовые варианты.

```
function HillClimbing(Cost)
    X_best ← randomly create initial solution;
    while true do
        neighbors ← GetAllNeighborSolutions( X_best );
        X_cur ← select solution with highest cost from neighbors ;
        if (Cost(X_cur) ≥ Cost(X_best))
            X_best ← X_cur ;
        else
            return X_best ;
        end if
    end while
end function
```

3. Построить потоковый граф. Используя способ тестирования ветвей и операторов отношений, составить тестовые варианты.

```
нач
    цел i
    i := n
    нц пока (i >= 1) и (F[i] > D)
        F[i + 1] := F[i] |сдвиг очередного элемента вправо на одну позицию
        i := i − 1
    кц
    F[i + 1] := D
кон
```

4. Составить тестовые варианты для следующего условия: (B_1 <> B_2) OR (E_1 > = E_2).

5. Составить тестовые варианты для следующего условия: ($B_1 = B_2$) OR ($E_1 < E_2$).

6. Составить тестовые варианты для следующего условия: ($B_1 <= B_2$) & ($E_1 > E_2$).

7. Составить тестовые варианты для следующего условия: ($B_1 <> B_2$) OR ($E_1 > = E_2$) & ($C_1 = C_2$).

8. Составить тестовые варианты для следующего условия: ($E_1 = E_2$) &($B_1 = B_2$) OR ($C_1 < C_2$).

ЛЕКЦИЯ 5

ТЕМА 5: Структурное тестирование программного продукта. Способ тестирования потоков данных. Примеры. Тестирование циклов. Простые циклы. Вложенные циклы. Объединённые циклы. Неструктурированные циклы

Способ тестирования потоков данных

В предыдущих способах тесты строились на основе анализа управляющей структуры программы. В данном способе анализу подвергается **информационная структура программы.**

Работу любой программы можно рассматривать как обработку потока данных, передаваемых от входа в программу к ее выходу.

Пример

Пусть потоковый граф программы имеет вид, представленный на рисунке 5.1. В нем сплошные дуги — это связи по управлению между операторами в программе. Пунктирные дуги отмечают информационные связи (связи по потокам данных). Обозначенные здесь информационные связи соответствуют следующим допущениям:

- в вершине **1** определяются значения переменных ***a***, ***b***;
- значение переменной ***a*** используется в вершине **4**;
- значение переменной ***b*** используется в вершинах **3**, **6**;
- в вершине **4** определяется значение переменной **c**, которая используется в вершине **6**.

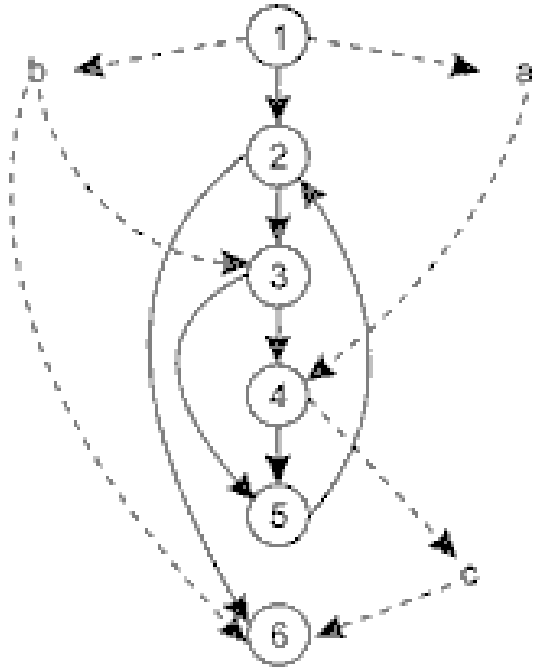


Рисунок 5.1 - Граф программы с управляющими и информационными связями

В общем случае для каждой вершины графа можно записать:

– множество определений данных:

DEF(i) = {x | i -я вершина содержит определение x}; DEF (1) = {a, b}, DEF (4) = {c},

– множество использований данных:

USE (i) = {x | i -я вершина использует x}, USE (3) = {b}, USE (4) = {a}, USE (3) = {b, c}.

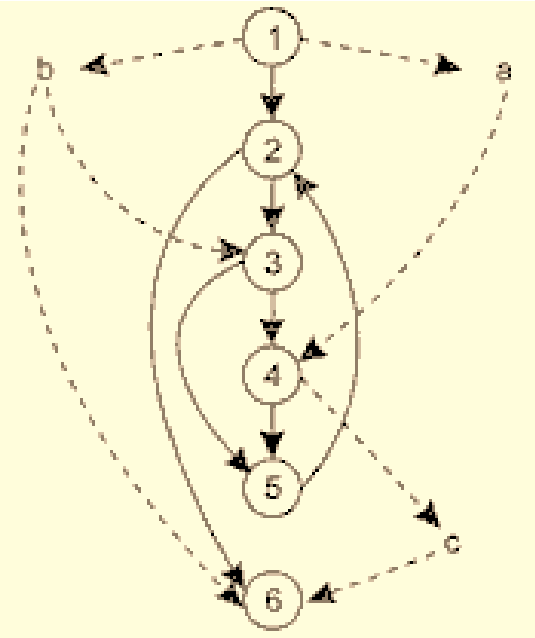
Под ***определением данных*** понимают действия, изменяющие элемент данных. Признак определения — имя элемента стоит в левой части оператора присваивания:

$$x := f(\ldots).$$

Использование данных — это применение элемента в выражении, где происходит обращение к элементу данных, но не изменение элемента. Признак использования — имя элемента стоит в правой части оператора присваивания:

$$[] := f(x).$$

Здесь место подстановки другого имени отмечено прямоугольником (прямоугольник играет роль метки-заполнителя).



Назовём ***DU-цепочкой (цепочкой определения-использования)*** конструкцию [*x, i*, j],

где *i, j* — имена вершин,

x определена в i-й вершине (*x* ∈ DEF(i)),

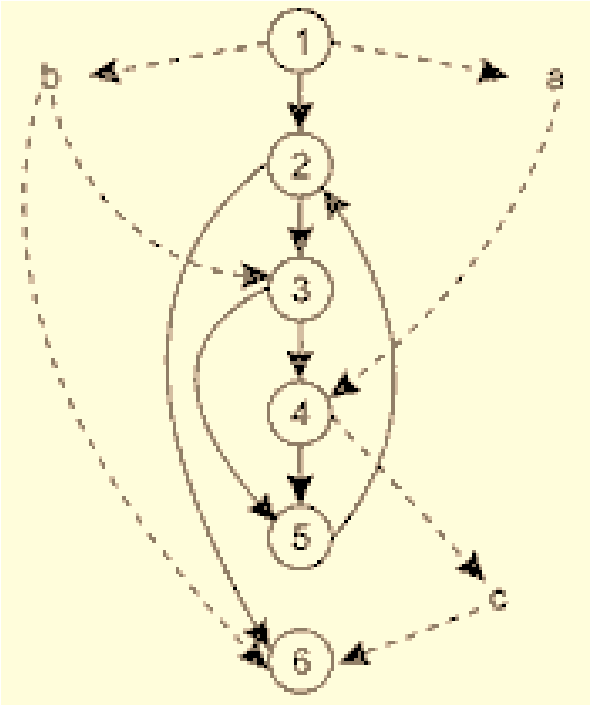
DEF (1) = {a, b}, DEF (4) = {c},

x используется в *j* -и вершине (*x* ∈ USE(j)),

USE (3) = {b}, USE (4) = {a}, USE (3) = {b, c}.

В данном примере существуют следующие DU-цепочки:

[*a,* 1, 4]; [b, 1, 3]; [b, 1, 6]; [c, 4, 6].



Способ *DU-тестирования* требует охвата всех DU-цепочек программы. Таким образом, разработка тестов здесь проводится на основе анализа жизни всех данных программы.

Для подготовки тестов требуется выделение маршрутов — путей выполнения программы на управляющем графе. **Критерий для выбора пути** — покрытие максимального количества DU-цепочек.

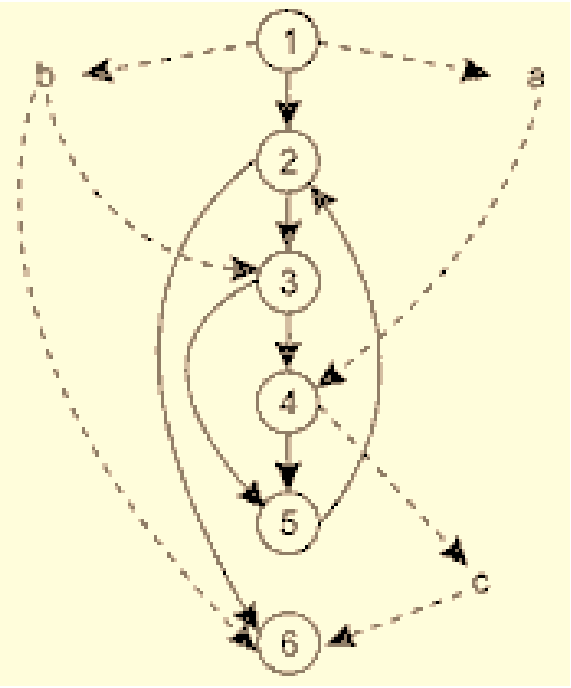
Шаги способа DU-тестирования:

Шаг 1. Построение управляющего графа (УГ) программы.

Шаг 2. Построение информационного графа (ИГ).

Шаг 3. Формирование полного набора DU-цепочек.

Шаг 4. Формирование полного набора отрезков путей в управляющем графе (отображением набора DU-цепочек информационного графа, рисунок 5.2).



[a, 1, 4]:

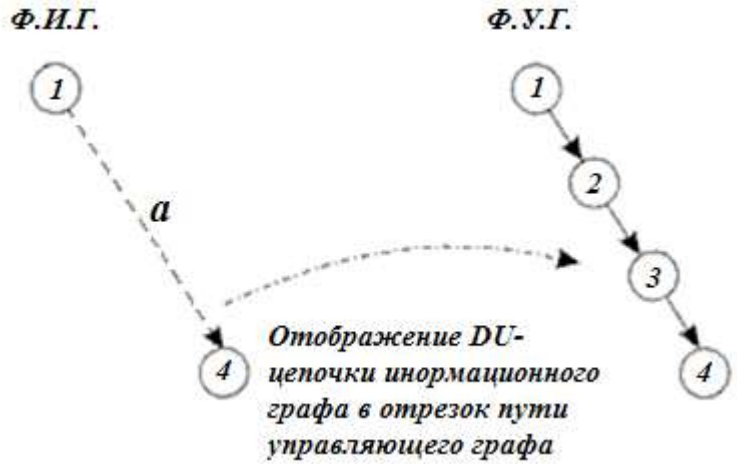


Рисунок 5.2 - Отображение DU-цепочки в отрезок пути

Шаг 5. Построение маршрутов — полных путей на управляющем графе, покрывающих набор отрезков путей управляющего графа.

Шаг 6. Подготовка тестовых вариантов.

Достоинства DU-тестирования:

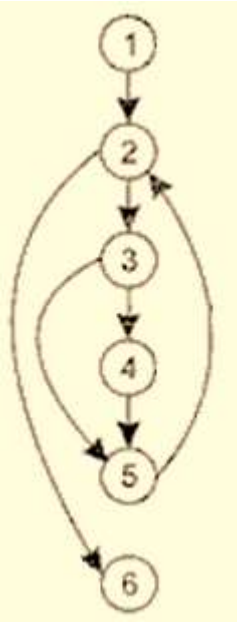
- простота необходимого анализа операционно-управляющей структуры программы;
- простота автоматизации.

Недостаток DU-тестирования: трудности в выборе минимального количества максимально эффективных тестов.

Область использования DU-тестирования: программы с вложенными условными операторами и операторами цикла.

Пример. Способ *DU-тестирования:*

УГ:



Шаг 1. Построение управляющего графа (УГ) программы.

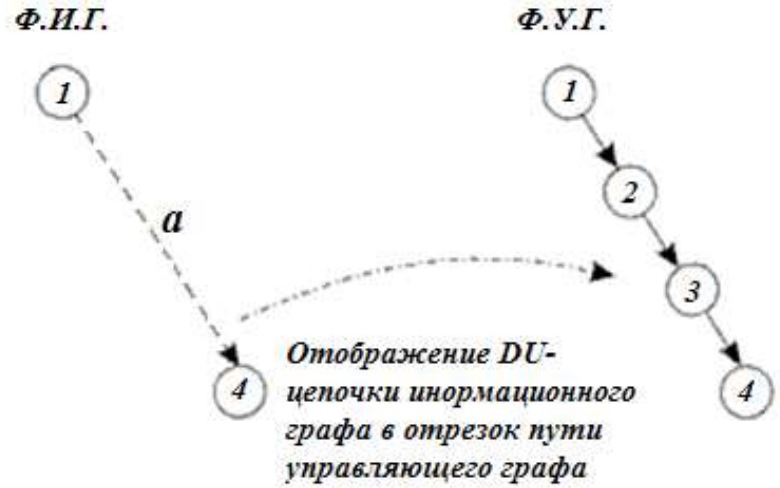
Шаг 2. Построение информационного графа (ИГ).

Шаг 3. Формирование полного набора DU-цепочек.

[*a,* 1, 4]; [b, 1, 3]; [b, 1, 6]; [c, 4, 6].

Шаг 4. Формирование полного набора отрезков путей в управляющем графе (Ф.У.Г.) (отображением набора DU-цепочек информационного графа (Ф.И.Г.), рисунок 5.2).

[*a,* 1, 4]:



ИГ:

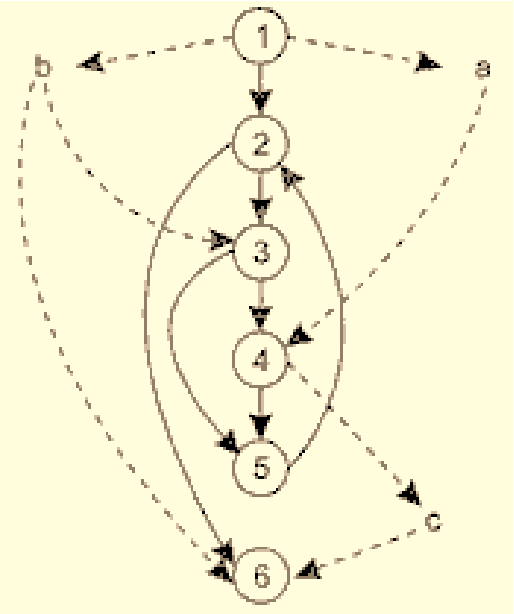
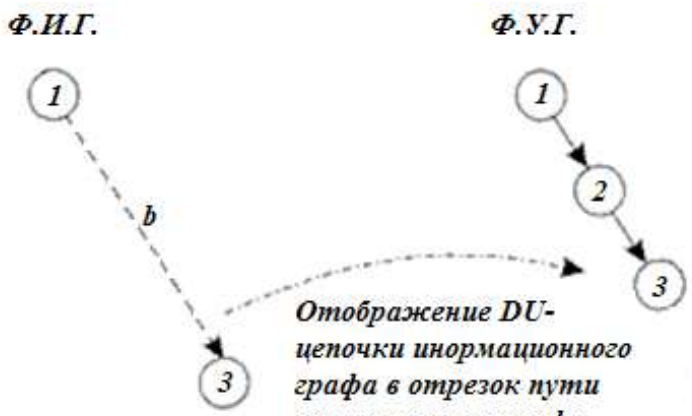
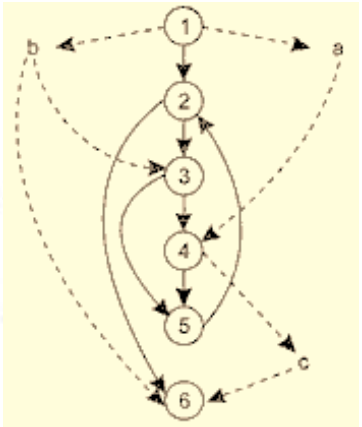


Рисунок 5.2 - Отображение цепочки [*a,* 1, 4] в отрезок пути

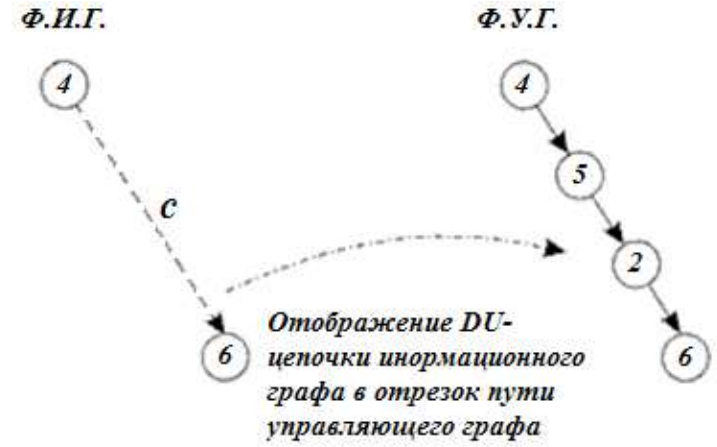
[b, 1, 3]:

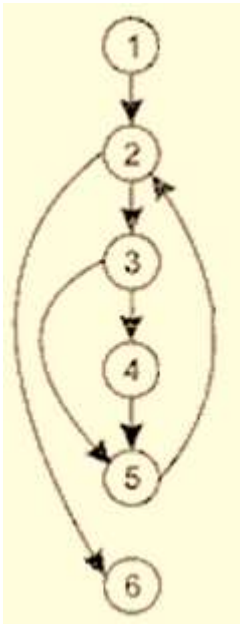




[b, 1, 6]:

[c, 4, 6]:





Шаг 5. Построение маршрутов — полных путей на управляющем графе, покрывающих набор отрезков путей управляющего графа.

Путь 1: 1 – 2 – 3 – 4 – 5 – 2 – 6.

Путь 2: 1 – 2 – 3 – 5 – 2 – 6.

Путь 3: 1 – 2 – 6.

Путь 4: 1 – 2 – 3 – 4 – 5 – 2 – 6.

Итоговые пути: 1 (4), 2, 3

Шаг 6. Подготовка тестовых вариантов.

ТВ1, ТВ2, ТВ3.

Тестирование циклов

Цикл — наиболее распространенная конструкция алгоритмов, реализуемых в ПО. Тестирование циклов производится по принципу «белого ящика», при проверке циклов основное внимание обращается на правильность конструкций циклов. Различают **4 типа циклов**:

- простые;
- вложенные;
- объединенные;
- неструктурированные.

Структура циклов приведена на рисунке 5.3.

Рисунок 5.3 - Типовые структуры циклов

Простые циклы

Для проверки простых циклов с количеством повторений *n* может использоваться один из следующих наборов тестов:

1) прогон всего цикла;
2) только один проход цикла;
3) два прохода цикла;
4) *m* проходов цикла, где *m* < *n;*
5) *n* - 1, *n, n* + 1 проходов цикла.

Вложенные циклы

С увеличением уровня вложенности циклов количество возможных путей резко возрастает. Это приводит к нереализуемому количеству тестов. Для сокращения количества тестов применяется специальная методика, в которой используются такие понятия, как объемлющий и вложенный циклы (рисунок 5.4).

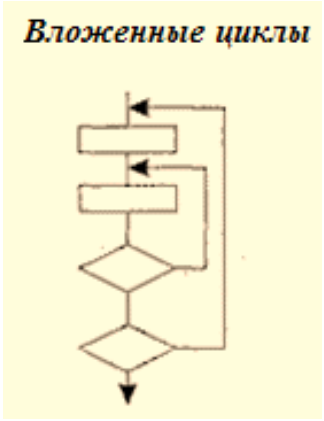


Рисунок 5.4 - Объемлющий и вложенный циклы

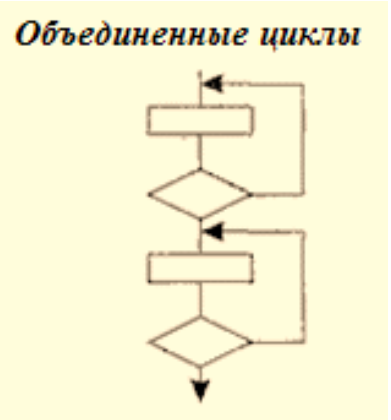
Порядок тестирования вложенных циклов иллюстрирует рисунок 5.5.

Рисунок 5.5 - Шаги тестирования вложенных циклов

Шаги тестирования:

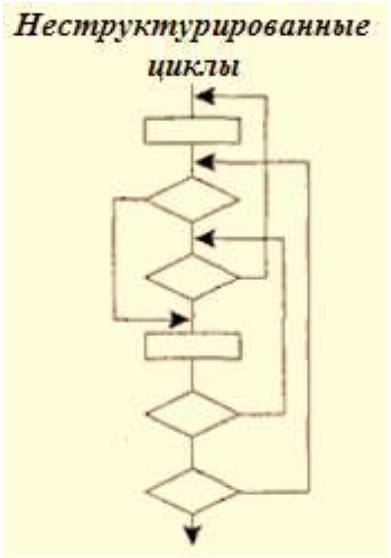
1. Выбирается самый внутренний цикл. Устанавливаются минимальные значения параметров всех остальных циклов.
2. Для внутреннего цикла проводятся тесты простого цикла. Добавляются тесты для исключенных значений и значений, выходящих за пределы рабочего диапазона.
3. Переходят в следующий по порядку объемлющий цикл. Выполняют его тестирование. При этом сохраняются минимальные значения параметров для всех внешних (объемлющих) циклов и типовые значения для всех вложенных циклов.
4. Работа продолжается до тех пор, пока не будут протестированы все циклы.

Объединенные циклы



Если каждый из циклов независим от других, то используется техника тестирования простых циклов. При наличии зависимости (например, конечное значение счетчика первого цикла используется как начальное значение счетчика второго цикла) используется методика для вложенных циклов.

Неструктурированные циклы



Неструктурированные циклы тестированию не подлежат. Этот тип циклов должен быть переделан с помощью структурированных программных конструкций.

ЗАДАНИЯ

Способ тестирования потоков данных

1. Построить потоковый граф. Используя способ тестирования потоков данных, составить тестовые варианты.

```
нач
    цел i
    i := n
    нц пока (i >= 1) и (F[i] > D)
        F[i + 1] := F[i] сдвиг очередного элемента вправо на одну позицию
        i := i − 1
    кц
    F[i + 1] := D
кон
```

2. Построить потоковый граф. Используя способ тестирования потоков данных, составить тестовые варианты.

```
 readln(x)
 L := x; M:= 111
 if L mod 2 <> 0 then
    M := 66
 while L <> M do
    if L > M then
      L := L - M
    else
      M := M - L
 writeln(M)
end
```

3. Построить потоковый граф. Используя способ тестирования потоков данных, составить тестовые варианты.

```
  readln(x)
  a := 0
  b:= 0
  while x > 0 do
    a := a + 1
    b := b + 2 * (x mod 10)
    x := x div 10
  end
  writeln(a); writeln(b)
end
```

Тестирование циклов

1. Построить потоковый граф. Составить тестовые варианты, используя способ тестирования циклов.

```
нач
    цел i, j, k
    вещ Tmp
    нц для i от 2 до n |i – начало неотсортированного массива
        Tmp := A[i];  j := 1
        нц пока  Tmp > A[j]
            j := j + 1 |j — фиксирует место вставки
            нц для k от i – 1 до j
                A[k+1] := A[k] |сдвиг вправо неотсортированной части
            кц
            A[j] := Tmp |вставка выбранного элемента
        кц
    кц
кон
```

2. Построить потоковый граф. Составить тестовые варианты, используя способ тестирования циклов.

```
    нач
        цел i, j
        вещ Tmp
нц для i от 2 до n
    нц для j от n до 1
        если A[j] < A[j – 1]
            то Tmp := A[j]; A[j] := A[j – 1]; A[j – 1] := Tmp | элементы меняются местами
        всё
    кц
кц
кон
```

3. Построить потоковый граф. Составить тестовые варианты, используя способ тестирования циклов.

```
нач
    цел i, j, Amax
    Amax := A[1, 1]
    нц для i от 1 до n |Поиск максимального элемента матрицы
        нц для j от 1 до m
            если A[i, j] > Amax то Amax := A[i, j]
            всё
        кц
    кц
    K := 0 |подсчет количества вхождений Amax
    нц для i от 1 до n
        нц для j от 1 до m
            если A[i, j] = Amax то K := K + 1
            всё
        кц
    кц
кон
```

ЛЕКЦИЯ 6

ТЕМА 6: Функциональное тестирование программного продукта. Особенности тестирования “чёрного ящика”. Способы тестирования “чёрного ящика”. Способ разбиения по эквивалентности. Способ анализа граничных значений. Способ диаграмм причин-следствий. Пример

ФУНКЦИОНАЛЬНОЕ ТЕСТИРОВАНИЕ — это тип тестирования программного обеспечения, который проверяет программную систему на соответствие функциональным требованиям / спецификациям. Цель функциональных тестов состоит в том, чтобы проверить каждую функцию программного приложения, предоставляя соответствующий ввод, проверяя выход в соответствии с функциональными требованиями.

Функциональное тестирование в основном включает тестирование «черного ящика» и не касается исходного кода приложения. Это тестирование проверяет:

- пользовательский интерфейс;
- API;
- базу данных;
- безопасность;
- связь клиент / сервер;
- и другие функциональные возможности тестируемого приложения.

Тестирование может проводиться либо вручную, либо с использованием автоматизации.

Основной целью функционального тестирования является проверка функциональности системы программного обеспечения. Она в основном концентрируется на:

- основных **функциях**: тестирование основных функций приложения;
- **базовом удобстве использования**: включает базовое юзабилити-тестирование системы. Он проверяет, может ли пользователь свободно перемещаться по экранам без каких-либо затруднений;
- **доступности**: проверяет доступность системы для пользователя;
- **условиях ошибки**: использование методов тестирования для проверки ошибок. Он проверяет, отображаются ли подходящие сообщения об ошибках.

Чтобы функционально протестировать приложение, необходимо соблюдать следующие шаги:

Шаг 1. Определение вводных данных для теста (тестовые данные).

Шаг 2. Рассчитать ожидаемые результаты для вводимых данных.

Шаг 3. Выполнить тестовые варианты.

Шаг 4. Сравнение фактических и ожидаемых рассчитанных результатов.

Особенности тестирования «черного ящика»

Тестирование «черного ящика» (функциональное тестирование) позволяет получить комбинации входных данных, обеспечивающих полную проверку всех функциональных требований к программе. Программное изделие здесь рассматривается как «черный ящик», чье поведение можно определить только исследованием его входов и соответствующих выходов. При таком подходе желательно иметь:

– набор, образуемый такими входными данными, которые приводят к аномалиям поведения программы (назовем его *IT);*
– набор, образуемый такими выходными данными, которые демонстрируют дефекты программы (назовем его *OT).*

Как показано на рисунке 6.1, любой способ тестирования «черного ящика» должен:

– выявить такие входные данные, которые с высокой вероятностью принадлежат набору IT;
– сформулировать такие ожидаемые результаты, которые с высокой вероятностью являются элементами набора OT.

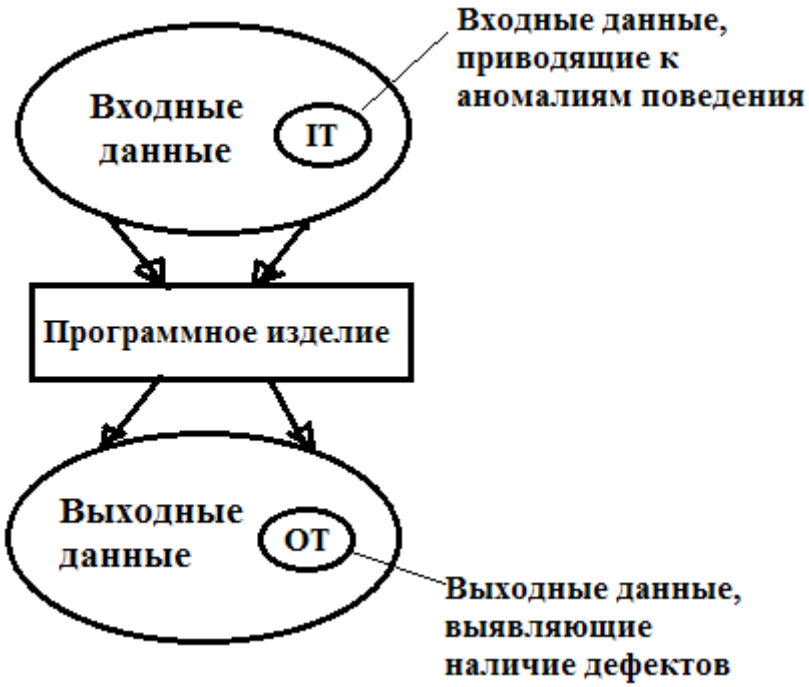


Рисунок 6.1 - Тестирование «чёрного ящика»

Во многих случаях определение таких тестовых вариантов основывается на предыдущем опыте инженеров тестирования. Они используют свое знание и понимание области определения для идентификации тестовых вариантов, которые эффективно обнаруживают дефекты. Тем не менее систематический подход к выявлению тестовых данных, обсуждаемый в данной лекции, может использоваться как полезное дополнение к эвристическому знанию. Принцип «черного ящика» не альтернативен принципу «белого ящика». Скорее это дополняющий подход, который обнаруживает другой класс ошибок.

Тестирование «черного ящика» обеспечивает поиск следующих **категорий ошибок**:

- некорректных или отсутствующих функций;
- ошибок интерфейса;
- ошибок во внешних структурах данных или в доступе к внешней базе данных;
- ошибок характеристик (необходимая емкость памяти и т. д.);
- ошибок инициализации и завершения.

Подобные категории ошибок способами «белого ящика» не выявляются.

В отличие от тестирования «белого ящика», которое выполняется на ранней стадии процесса тестирования, тестирование «черного ящика» применяют на поздних стадиях тестирования. При тестировании «черного ящика» пренебрегают управляющей структурой программы. Здесь внимание концентрируется на информационной области определения программной системы.

Техника «черного ящика» ориентирована на решение следующих задач:

– сокращение необходимого количества тестовых вариантов (из-за проверки не, статических, а динамических аспектов системы);
– выявление классов ошибок, а не отдельных ошибок.

Рассмотрим следующие способы функционального тестирования:

- Способ разбиения по классам эквивалентности.
- Способ анализа граничных значений.
- Способ диаграмм причин-следствий.

Способ разбиения по эквивалентности

Разбиение по эквивалентности — самый популярный способ тестирования «черного ящика».

В этом способе входная область данных программы делится на классы эквивалентности. Для каждого класса эквивалентности разрабатывается один тестовый вариант.

Класс эквивалентности — набор данных с общими свойствами. Обрабатывая разные элементы класса, программа должна вести себя одинаково. Иначе говоря, при обработке любого набора из класса эквивалентности в программе задействуется один и тот же набор операторов (и связей между ними).

Рисунок 6.2 - Разбиение по эквивалентности

На рисунке 6.2 каждый класс эквивалентности показан эллипсом. Здесь выделены входные классы эквивалентности допустимых и недопустимых исходных данных, а также классы результатов. Классы эквивалентности могут быть определены по спецификации на программу.

Например, если спецификация задает в качестве допустимых входных величин 5-разрядные целые числа в диапазоне 20 000÷60 000, то класс эквивалентности допустимых ИД (исходных данных) включает величины от 20 000 до 60 000, а два класса эквивалентности недопустимых ИД составляют:

– числа меньшие, чем 20 000 (19 999);
– числа большие, чем 60 000 (60 001).

Класс эквивалентности включает множество значений данных, допустимых или недопустимых по условиям ввода.

Условие ввода может задавать:

1) диапазон значений;
2) определенное значение;
3) множество конкретных величин;
4) булево условие.

Правильно выбранный тест этого подмножества должен обладать двумя **свойствами**:

- уменьшать, причем более чем на единицу, число других тестов, которые должны быть разработаны для достижения заранее определенной цели «приемлемого» тестирования;
- покрывать значительную часть других возможных тестов, что в не которой степени свидетельствует о наличии или отсутствии ошибок до и после применения этого ограниченного множества значений входных данных.

Указанные свойства, несмотря на их кажущееся подобие, описывают два различных положения:

Во-первых, каждый тест должен включать столько различных входных условий, сколько это возможно, с тем чтобы **минимизировать** общее число необходимых тестов.

Во-вторых, необходимо пытаться разбить входную область программы на конечное число классов эквивалентности так, чтобы можно было предположить (конечно, не абсолютно уверенно), что каждый тест, являющийся представителем некоторого класса, эквивалентен любому другому тесту этого класса.

Иными словами, если один тест класса эквивалентности обнаруживает ошибку, то следует ожидать, что и все другие тесты этого класса эквивалентности будут обнаруживать ту же самую ошибку. Наоборот, если тест не обнаруживает ошибки, то следует ожидать, что ни один тест этого класса эквивалентности не будет обнаруживать ошибки (в том случае, когда некоторое подмножество класса эквивалентности не попадает в пределы любого другого класса эквивалентности, так как классы эквивалентности могут пересекаться).

Эти два положения составляют основу методологии тестирования по принципу черного ящика, известной как эквивалентное разбиение. Второе положение используется для разработки набора «интересных» условий, которые должны быть протестированы, а первое – для разработки минимального набора тестов, покрывающих эти условия.

Примером класса эквивалентности для программы о треугольнике является набор «трех равных чисел, имеющих целые значения, большие нуля». Определяя этот набор как класс эквивалентности, устанавливают, что если ошибка не обнаружена некоторым тестом данного набора, то маловероятно, что она будет обнаружена другим тестом набора. Иными словами, в этом случае время тестирования лучше затратить на что-нибудь другое (на тестирование других классов эквивалентности).

Разработка тестов методом эквивалентного разбиения осуществляется в два этапа:

Шаг 1. Выделение классов эквивалентности;

Шаг 2. Построение тестов.

Построение тестов

Этот процесс включает в себя:

1. Назначение каждому классу эквивалентности уникального номера.
2. Проектирование новых тестов, каждый из которых покрывает как можно большее число непокрытых правильных классов эквивалентности, до тех пор, пока все правильные классы эквивалентности не будут покрыты (только не общими) тестами.
3. Запись тестов, каждый из которых покрывает один и только один из непокрытых неправильных классов эквивалентности, до тех пор, пока все неправильные классы эквивалентности не будут покрыты тестами.

Правила формирования классов эквивалентности*:*

1. Если условие ввода задает диапазон *n...m,* то определяются один допустимый и два недопустимых класса эквивалентности:

   – V_Class = {n...*m*} — допустимый класс эквивалентности;

   – Inv_C1ass1 = {x| для любого *x: x < n*} — первый недопустимый класс эквивалентности;

   – Ivn_C1ass2 = {y| для любого *y: y > m*} — второй недопустимый класс эквивалентности.

Пример 1. Целое данное может принимать значения от 1 до 99 ([1, 99])

V_Class = {1...*99*} — допустимый класс эквивалентности;

Inv_C1ass1 = {x| для любого *x: x < 1*} — первый недопустимый класс эквивалентности;

Ivn_C1ass2 = {y| для любого *y: y > 99*} — второй недопустимый класс эквивалентности.

Пример 2. В автомобиле могут ехать от одного до пяти человек, то определяются один правильный класс эквивалентности и два неправильных (ни одного и более пяти человек).

V_Class = {1...5};

Inv_C1ass1 = {x| для *любого x: x < 1};*

Ivn_Class2 = {y| для любого *y: y > 5*}.

Пример 3. Если программа настроена на работу с памятью объемом от 64 Мб до 256 Мб, то этот диапазон считается классом эквивалентности. Любой другой объем, больше, чем 256 Мб или меньше, чем 64 Мб, считается недопустимым классом.

Правила формирования классов эквивалентности*:*

2. Если условие ввода задает конкретное значение *a,* то определяется один допустимый и два недопустимых класса эквивалентности:

– V_Class = {a};

– Inv_C1ass1 = {x| для *любого x: x < a*};

– Ivn_Class2 = {y| для любого *y: y > a*}.

3. Если условие ввода задает множество значений *{a, b, c},* то определяются один допустимый и один недопустимый класс эквивалентности:

– V_Class = {a, *b, c*};

– Inv_C1ass = {x| для любого *x: (x ≠ a)&(x ≠ b)&(x ≠*c)}.

Пример 1. Известны должности АДМИНИСТРАТОР, РАЗРАБОТЧИК, ТЕСТИРОВЩИК, ДИРЕКТОР, то определяется правильный класс эквивалентности для каждого значения и один неправильный класс эквивалентности (например, «БУХГАЛТЕР»).

Пример 2. Если входное условие описывает ситуацию «должно быть» (например, «первым символом идентификатора должна быть буква»), то определяется один правильный класс эквивалентности (первый символ – буква) и один неправильный (первый символ – не буква)

Пример 3. Если входное условие содержит выражение долженствования (например, "строка ввода должна содержать прописные символы"), то определяются один допустимый класс (прописные символы) и один недопустимый класс (все остальные варианты ввода кроме прописных символов).

Правила формирования классов эквивалентности:

4. Если условие ввода задает булево значение, например true, то определяются один допустимый и один недопустимый класс эквивалентности:

   – V_Class = {true};

   – Inv_Class = {false}.

Пример. Если входное условие описывает ситуацию «должно быть», то определяется один правильный класс эквивалентности и один неправильный.

После построения классов эквивалентности разрабатываются тестовые варианты. Тестовый вариант выбирается так, чтобы проверить сразу наибольшее количество свойств класса эквивалентности.

Рекомендации по определению классов эквивалентности:

1. Все действия, выполненные "задолго" до выполнения задачи, считаются классом эквивалентности. Все действия, выполненные незадолго до завершения программы, считаются еще одним классом. Все действия, выполненные непосредственно перед запуском программой другой операции, также считаются отдельным классом.

2. Разбиение выходных событий на классы связано с входными данными программы. Несмотря на то, что некоторым входным классам эквивалентности могут соответствовать выходные события того же типа, рекомендуется входные классы рассматривать отдельно.

Способ анализа граничных значений

Как правило, большая часть ошибок происходит на границах области ввода, а не в центре. Анализ граничных значений заключается в получении тестовых вариантов, которые анализируют граничные значения. Данный способ тестирования дополняет способ разбиения по эквивалентности.

Основные отличия анализа граничных значений от разбиения по эквивалентности:

1) тестовые варианты создаются для проверки только ребер классов эквивалентности;

2) при создании тестовых вариантов учитывают не только условия ввода, но и область вывода.

Анализ граничных значений

Как показывает опыт, тесты, исследующие граничные условия, приносят большую пользу, чем тесты, которые их не исследуют. Граничные условия – это ситуации, возникающие непосредственно на, выше или ниже границ входных и выходных классов эквивалентности. Анализ граничных значений отличается от эквивалентного разбиения в двух отношениях:

1. Выбор любого элемента в классе эквивалентности в качестве представительного при анализе граничных значений осуществляется таким образом, чтобы проверить тестом каждую границу этого класса.

2. При разработке тестов рассматривают не только входные условия (пространство входов), но и пространство результатов (т. е. выходные классы эквивалентности).

Достаточно трудно описать принимаемые решения при анализе граничных значений, так как это требует определенной степени творчества и специализации в рассматриваемой проблеме. (Следовательно, анализ граничных значений, как и многие другие аспекты тестирования, в значительной мере основывается на способностях человеческого интеллекта.)

Тем не менее существует несколько **общих правил этого метода**.

Правила анализа граничных значений:

1. Если условие ввода задает диапазон *n...m, то* тестовые варианты должны быть построены:

   – для значений *n* и *m;*

   – для значений чуть левее *n* и чуть правее *m* на числовой оси.

   Например, если задан входной диапазон -1,0. ..+1,0, то создаются тесты для значений -1,0, +1,0, -1,001, +1,001.

2. Если условие ввода задает дискретное множество значений, то создаются тестовые варианты:

   – для проверки минимального и максимального из значений;

   – для значений чуть меньше минимума и чуть больше максимума.

   Так, если входной файл может содержать от 1 до 255 записей, то создаются тесты для 0,1, 255, 256 записей.

3. Правила 1 и 2 применяются к условиям области вывода.

   Если вход или выход программы есть упорядоченное множество (например, последовательный файл, линейный список, таблица), то сосредоточить внимание на первом и последнем элементах этого множества

   Рассмотрим пример, когда в программе требуется выводить таблицу значений. Количество строк и столбцов в таблице меняется. Задается тестовый вариант для минимального вывода (по объему таблицы), а также тестовый вариант для максимального вывода (по объему таблицы).

4. Использовать первое правило для каждого выходного условия. Например, если программа вычисляет ежемесячный расход и если минимум расхода составляет $0.00, а максимум – $1165.25, то построить тесты, которые вызывают расходы с $0.00 и $1165.25. Кроме того, построить, если это возможно, тесты, которые вызывают отрицательный расход и расход больше 1165.25 дол. Заметим, что важно проверить границы пространства результатов, поскольку не всегда границы входных областей представляют такой же набор условий, как и границы выходных областей (например, при рассмотрении подпрограммы вычисления синуса). Не всегда также можно получить результат вне выходной области, но тем не менее стоит рассмотреть эту возможность.

5. Использовать второе правило для каждого выходного условия. Например, если система информационного поиска отображает на экране наиболее релевантные статьи в зависимости от входного запроса, но никак не более четырех рефератов, то построить тесты, такие, чтобы программа отображала нуль, один и четыре реферата, и тест, который мог бы вызвать выполнение программы с ошибочным отображением пяти рефератов.

6. Если внутренние структуры данных программы имеют предписанные границы, то разрабатываются тестовые варианты, проверяющие эти структуры на их границах.

7. Попробовать свои силы в поиске других граничных условий. Чтобы проиллюстрировать необходимость анализа граничных значений, можно использовать программу анализа треугольника. Для задания треугольника входные значения должны быть целыми положительными числами, и сумма любых двух из них должна быть больше третьего. Если определены эквивалентные разбиения, то целесообразно определить одно разбиение, в котором это условие выполняется, и другое, в котором сумма двух целых не больше третьего. Следовательно, двумя возможными тестами являются 3–4–5 и 1–2–4. Тем не менее, здесь есть вероятность пропуска ошибки. Иными словами, если выражение в программе было закодировано как A + B ≥ C вместо A + B > C, то программа ошибочно сообщала бы нам, что числа 1–2–3 представляют правильный равносторонний треугольник. Таким образом, существенное различие между анализом граничных значений и эквивалентным разбиением заключается в том, что анализ граничных значений исследует ситуации, возникающие на и вблизи границ эквивалентных разбиений.

Рекомендации по выбору граничных значений:

1. Если допустимое значение плавающей переменной составляет от -1.0 до 1.0, протестируйте значения -1.0, 1.0, -1.001 и 1.001.
2. Если диапазон допустимых значений входных данных составляет целое число от 10 до 100, протестируйте 9, 10, 100, 101.
3. Если программа настроена на работу с прописными символами, протестируйте граничные значения для A и Z. Протестируйте @ и [, потому что в коде ASCII символ @ предшествует A, а символ [следует сразу за Z.
4. Если входные или выходные данные программы представляют собой упорядоченный набор, протестируйте первый и последний элементы набора.
5. Если сумма входных данных выражена числом (n), протестируйте программу, где сумма равна n-1, n или n+1.
6. Если программа поддерживает список, протестируйте значения из этого списка. Все остальные значения считаются недопустимыми.
7. При считывании из файла или записи в файл протестируйте первый и последний символы файла.
8. Наименьшая денежная единица - один цент или его эквивалент. Если программа поддерживает некий диапазон от a до b, протестируйте a -0.01 и b +0.01.
9. Если для переменной указано несколько диапазонов значений, каждый диапазон считается отдельным классом. Если подмножества значений из этих диапазонов не пересекаются, протестируйте крайние значения, а также граничные значения над верхней границей и под нижней границей.

Специальные значения

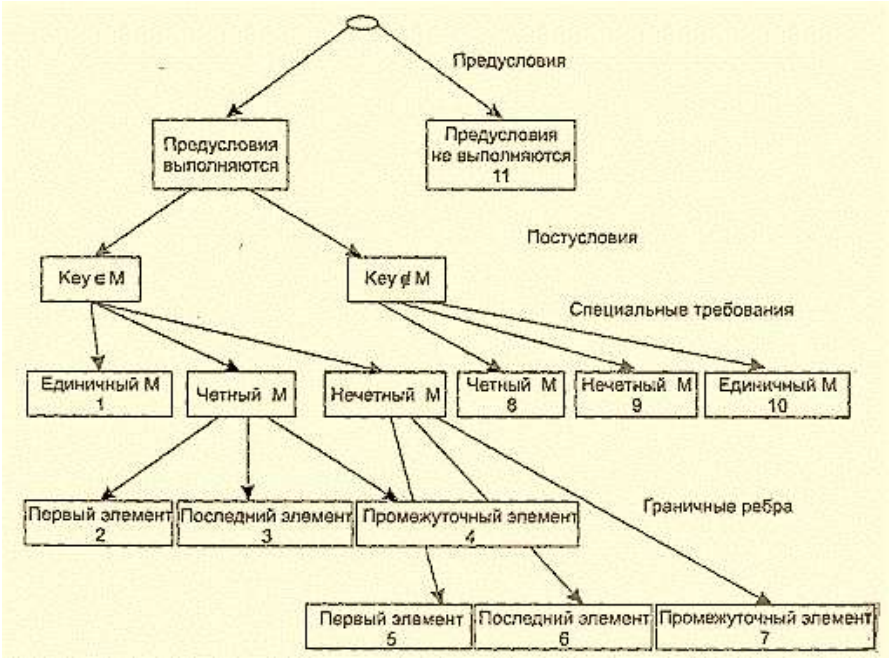
После применения указанных выше стратегий анализа граничных значений рекомендуется исследовать программу на наличие "специальных значений", с помощью которых можно обнаружить множество ошибок. Некоторые примеры приведены ниже:

1. В случае с целыми числами нужно обязательно протестировать ноль, если он входит в допустимый класс эквивалентности.
2. При тестировании времени необходимо для каждого элемента (час, минута и секунда) протестировать значения 59 и 0, независимо от ограничения, установленного для входной переменной. Помимо граничных значений входной переменной необходимо всегда тестировать значения -1, 0, 59 и 60.
3. При тестировании даты (год, месяц и день) необходимо добавить тестовые сценарии, например, для количества дней в конкретном месяце, количества дней в феврале в високосном году или количества дней в не високосном году.

Пример. Применение способов разбиения по эквивалентности и анализа граничных значений. Положим, что нужно протестировать программу бинарного поиска.

Спецификация программы бинарного поиска:

Поиск выполняется в массиве элементов *M,* возвращается индекс I элемента массива, значение которого соответствует ключу поиска *Key.*



Предусловия:

1) массив должен быть упорядочен;
2) массив должен иметь не менее одного элемента;
3) нижняя граница массива (индекс) должна быть меньше или равна его верхней границе.

Постусловия:

1) если элемент найден, то флаг Result=True, значение I — номер элемента;
2) если элемент не найден, то флаг Result=False, значение I не определено.

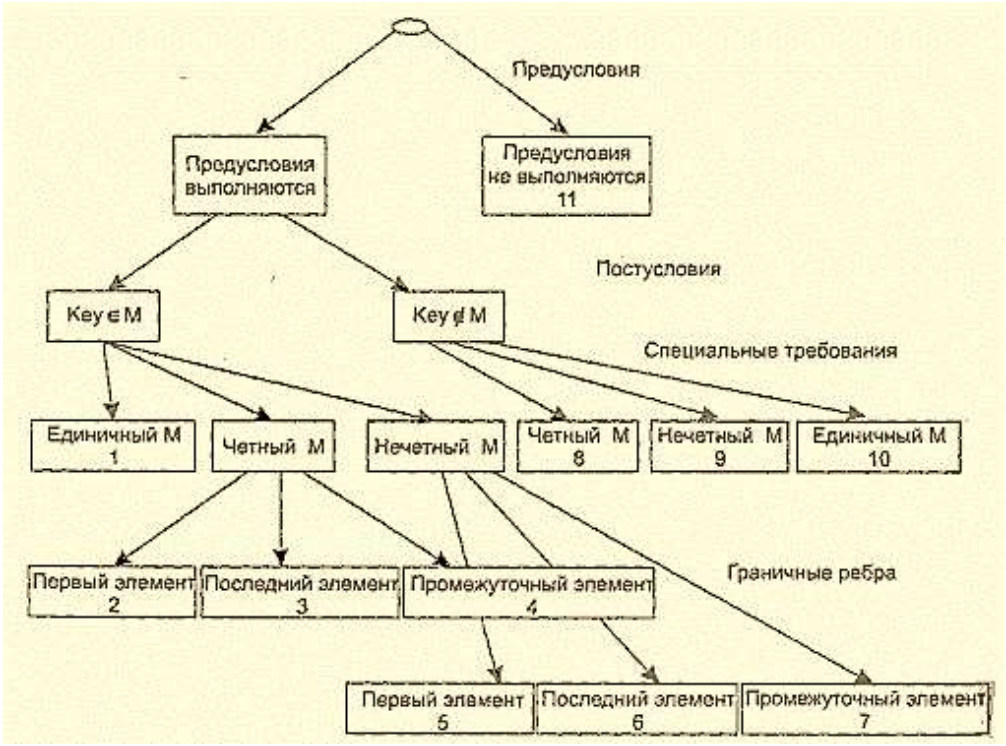
Для формирования классов эквивалентности (и их ребер) надо произвести разбиение области ИД — построить дерево разбиений. Листья дерева разбиений дадут искомые классы эквивалентности.

Стратегия разбиения

Первый уровень анализ выполнимости предусловий.

Второй уровень — выполнимость постусловий.

Третий уровень анализ специальных требований, полученные из практики разработчика. В данном примере — это входной массив должен быть упорядочен. Обработка упорядоченных наборов из четного и нечетного количества элементов может выполняться по-разному. Кроме того, принято выделять специальный случай одноэлементного массива.



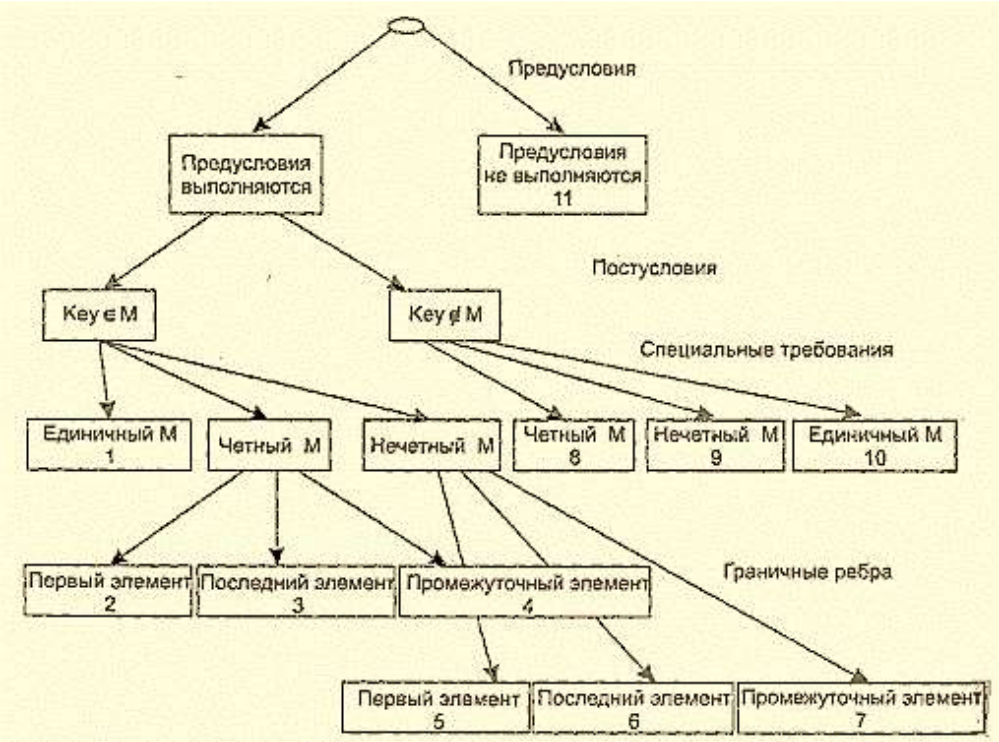
На уровне специальных требований возможны следующие эквивалентные разбиения:

1) массив из одного элемента;
2) массив из четного количества элементов;
3) массив из нечетного количества элементов, большего единицы.

4-м уровень - критерием разбиения может быть анализ ребер классов эквивалентности. Очевидно, возможны следующие варианты:

1) работа с первым элементом массива;
2) работа с последним элементом массива;
3) работа с промежуточным (ни с первым, ни с последним) элементом массива.

Структура дерева разбиений приведена на рисунке 6.3.



Это дерево имеет 11 листьев. Каждый лист задает отдельный тестовый вариант. Покажем тестовые варианты, основанные на проведенных разбиениях.

Рисунок 6.3 - Дерево разбиений области исходных данных бинарного поиска

ТВ1 *(единичный массив, элемент найден)***:**

ИД: M = (15); Key = 15.

ОЖ. РЕЗ.: Result = True; I = 1.

ТВ2 *(четный массив, найден 1-й элемент)*:

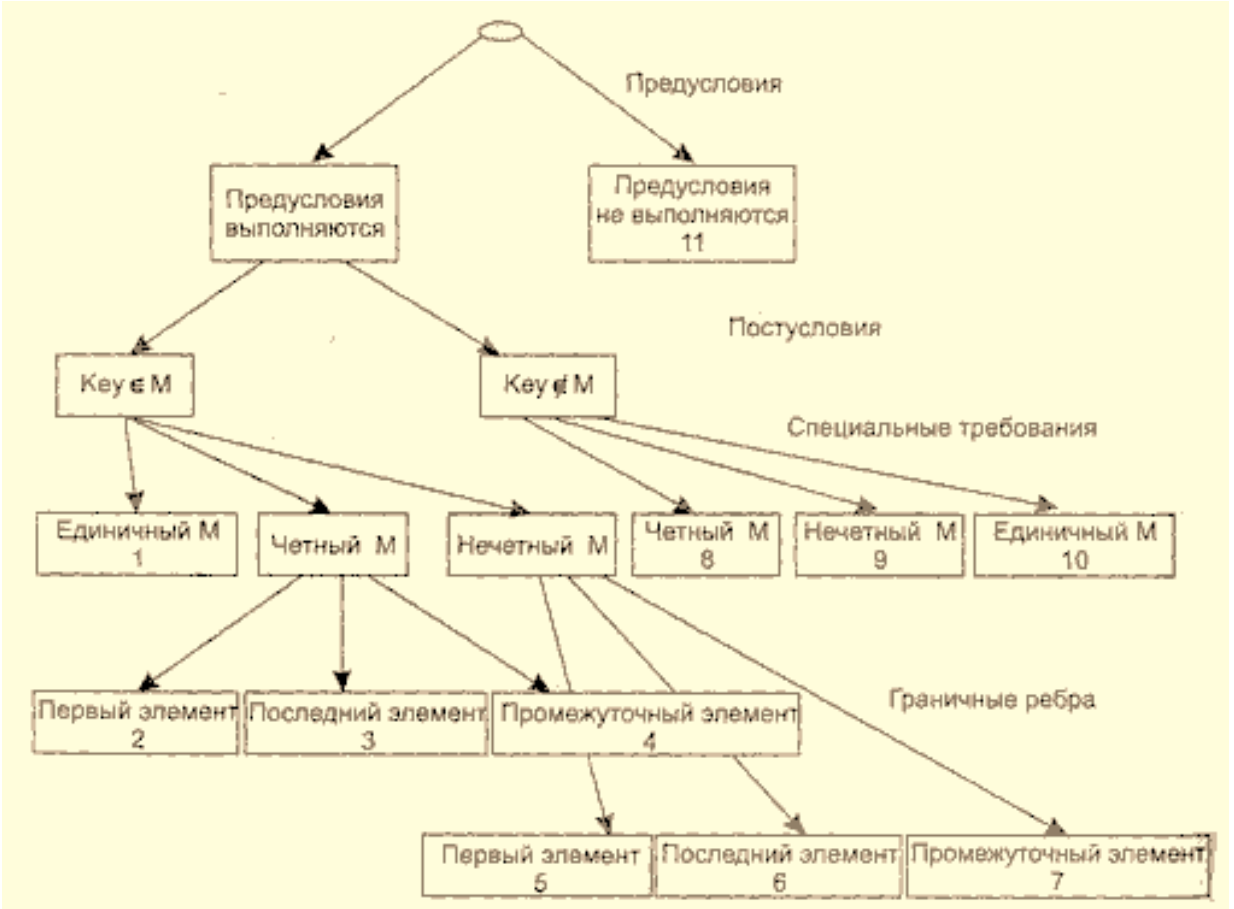
ИД: M = (15, 20, 25, 30, 35, 40); Key = 15.

ОЖ.РЕЗ.: Result = True; 1=1.

ТВ3 *(четный массив, найден последний элемент)***:**

ИД: M = (15, 20, 25, 30, 35,40); Key = 40.

ОЖ.РЕЗ.: Result = True; I = 6.



ТВ4 *(четный массив, найден промежуточный элемент)***:**

ИД: M = (15, 20, 25, 30, 35, 40); Key = 25.

ОЖ.РЕЗ.: Result = True; I = 3.

ТВ5 *(нечетный массив, найден 1-й элемент)***:**

ИД: M = (15, 20, 25, 30, 35, 40, 45); Key = 15.

ОЖ.РЕЗ.: Result = True; I = 1.

ТВ6 *(нечетный массив, найден последний элемент)***:**

ИД: M = (15, 20, 25, 30, 35, 40, 45); Key = 45.

ОЖ.РЕЗ.: Result = True; I = 7.

ТВ7 *(нечетный массив, найден промежуточный элемент)***:**

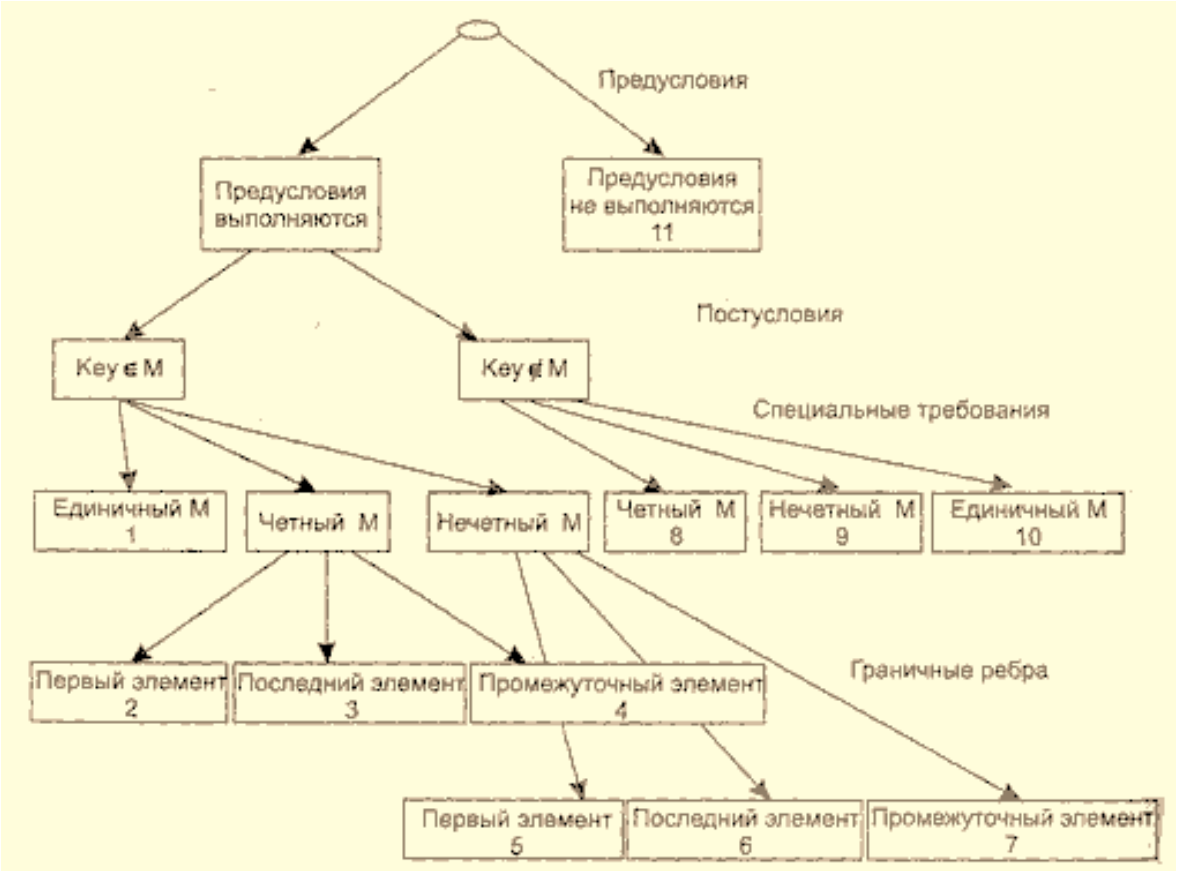
ИД: M = (15, 20, 25, 30, 35, 40, 45); Key = 30.

ОЖ.РЕЗ.: Result = True; I = 4.

ТВ8 *(четный массив, не найден элемент)***:**

ИД: M = (15, 20, 25, 30, 35, 40); Key = 23.

ОЖ.РЕЗ.: Result = False; I - ?



ТВ9 *(нечетный массив, не найден элемент):*

ИД: M = (15, 20, 25, 30, 35, 40, 45); Key = 24.

ОЖ.РЕЗ.: Result = False; I -?

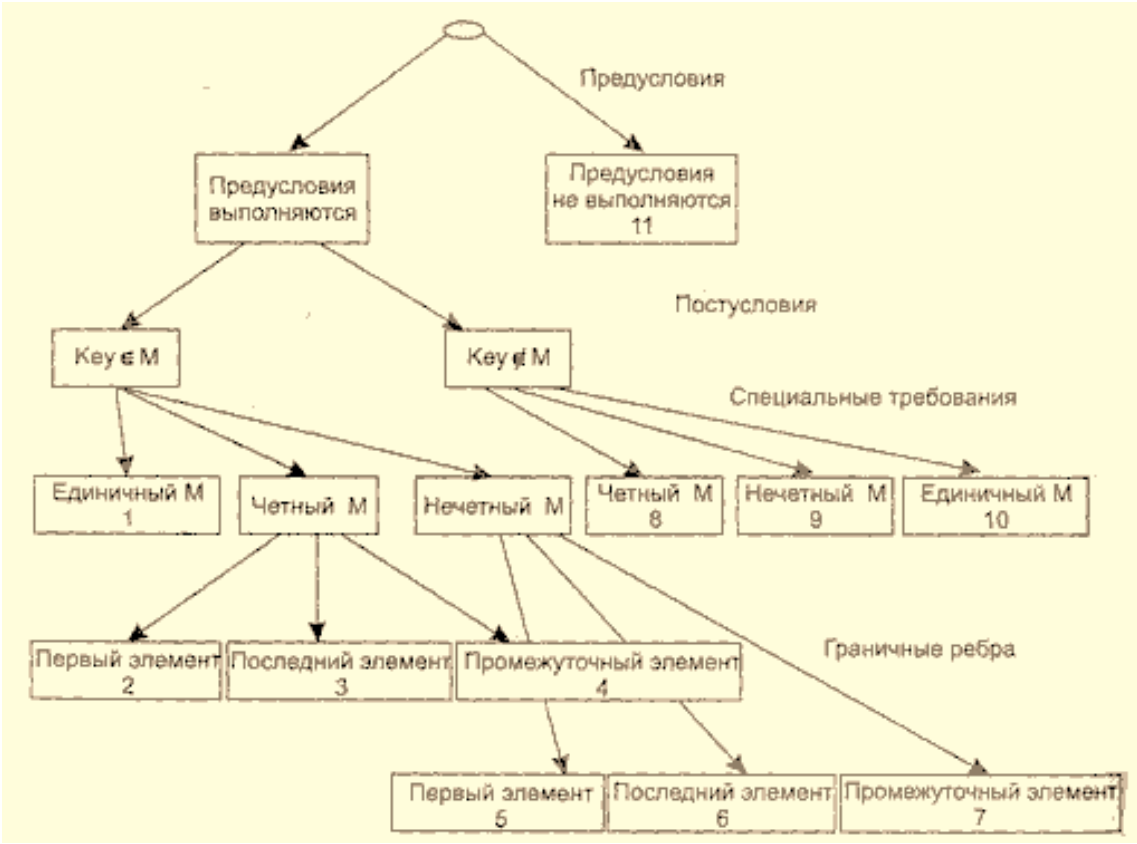
ТВ10 *(единичный массив, не найден элемент)***:**

ИД: M = (15); Key = 0.

ОЖ.РЕЗ.: Result = False; I -?

ТВ11 *(нарушены предусловия)***:**

ИД: M = (15, 10, 5, 25, 20, 40, 35); Key = 35.



ОЖ.РЕЗ.: Аварийное донесение: Массив не упорядочен.

ЛЕКЦИЯ 7

ТЕМА 6: Функциональное тестирование программного продукта. Особенности тестирования “чёрного ящика”. Способы тестирования “чёрного ящика”. Способ разбиения по эквивалентности. Способ анализа граничных значений. Способ диаграмм причин-следствий. Пример. Метод "Разбиение на категории"

Способ диаграмм причин-следствий

Диаграммы причинно-следственных связей — способ проектирования тестовых вариантов, который обеспечивает формальную запись логических условий и соответствующих действий. Используется автоматный подход к решению задачи.

Шаги способа диаграмм причин-следствий:

Шаг 1. Для каждого модуля перечисляются причины (условия ввода или классы эквивалентности условий ввода) и следствия (действия или условия вывода). Каждой причине и следствию присваивается свой идентификатор.

Шаг 2. Разрабатывается граф причинно-следственных связей.

Шаг 3. Граф преобразуется в таблицу решений.

Шаг 4. Столбцы таблицы решений преобразуются в тестовые варианты.

Базовые символы для записи графов причин и следствий (cause-effect graphs)

Сделаем предварительные *замечания:*

1) **причины** будем обозначать символами $\mathbf{c_j}$, а **следствия** — символами $\boldsymbol{e_i}$,
2) каждый узел графа может находиться в состоянии **0** или **1** (0 — состояние отсутствует, 1 — состояние присутствует).

Функция ***тождество*** (рисунок 6.4) устанавливает, что если значение c_1 есть 1, то и значение e_1 есть 1; в противном случае значение e_1 есть 0.

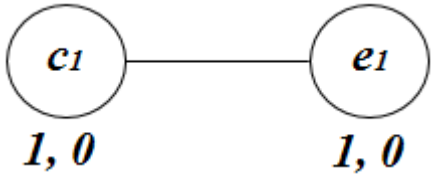


Рисунок 6.4 - Функция ***тождество***

Функция ***не*** (рисунок 6.5) устанавливает, что если значение $\mathbf{c_1}$ есть **1**, то значение $\boldsymbol{e_1}$ есть **0**; в противном случае значение $\boldsymbol{e_1}$ есть **1**.

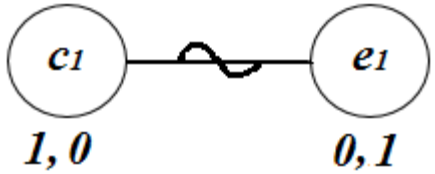


Рисунок 6.5 - Функция ***не***

Функция ***или*** (рисунок 6.6) устанавливает, что если значение $\mathbf{c_1}$ или $\mathbf{c_2}$ есть **1**, то $\boldsymbol{e_1}$ есть **1**, в противном случае $\boldsymbol{e_1}$ есть **0**.

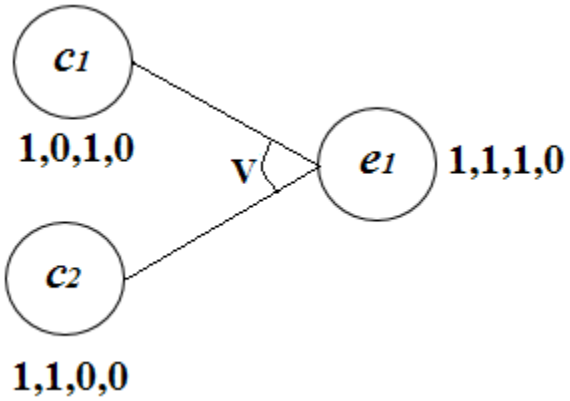


Рисунок 6.6 - Функция ***или***

Функция ***и*** (рисунок 6.7) устанавливает, что если $\mathbf{c_1}$ и $\mathbf{c_2}$ есть **1**, то $\boldsymbol{e_1}$ есть **1**, в противном случае $\boldsymbol{e_1}$ есть **0**.

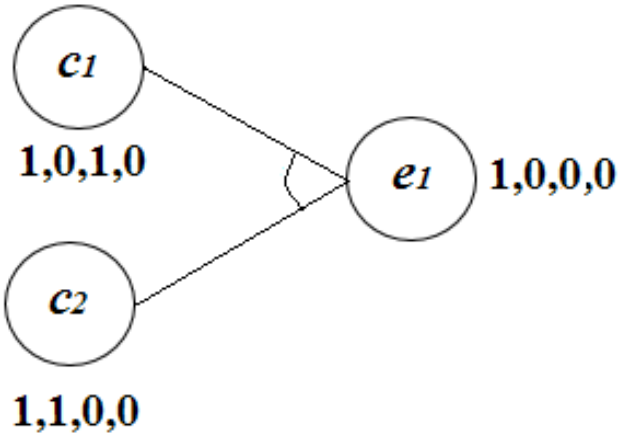


Рисунок 6.7 - Функция ***и***

Часто определенные комбинации причин невозможны из-за синтаксических или внешних ограничений. Используются перечисленные ниже обозначения **ограничений**.

Ограничение **E** (исключает, Exclusive, рисунок 2.8) устанавливает, что **E** должно быть истинным, если хотя бы одна из причин — ***a*** или **b** — принимает значение **1** (**a** и ***b*** не могут принимать значение **1** одновременно).

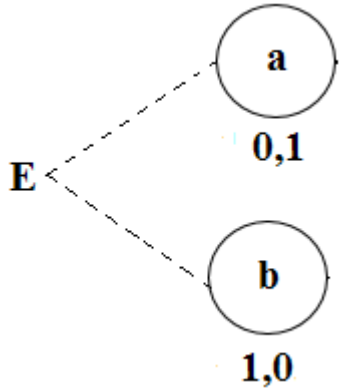


Рисунок 2.8 - Ограничение **E** (исключает, Exclusive)

Ограничение **I** (включает, Inclusive, рисунок 6.9) устанавливает, что по крайней мере одна из величин ***a***, ***b***, или ***c***, всегда должна быть равной **1** (***a***, ***b*** и ***c*** не могут принимать значение **0** одновременно).

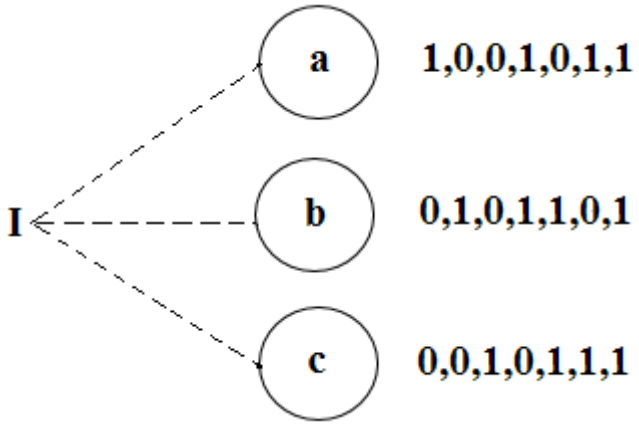


Рисунок 6.9 - Ограничение I (включает, Inclusive)

Ограничение **O** (одно и только одно, Only one, рисунок 6.10) устанавливает, что одна и только одна из величин ***a*** или ***b***, или ***c*** должна быть равна **1**.

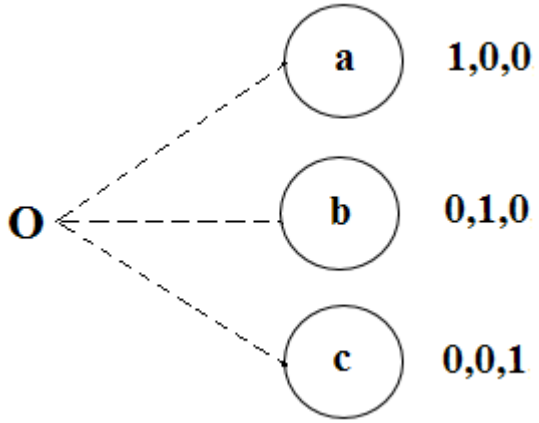


Рисунок 6.10 - Ограничение **O** (одно и только одно, Only one)

Ограничение **R** (требует, Requires, рисунок 6.11) устанавливает, что если ***a*** принимает значение **1**, то и **b** должна принимать значение **1** (нельзя, чтобы ***a*** было равно **1**, а ***b*** — **0**).

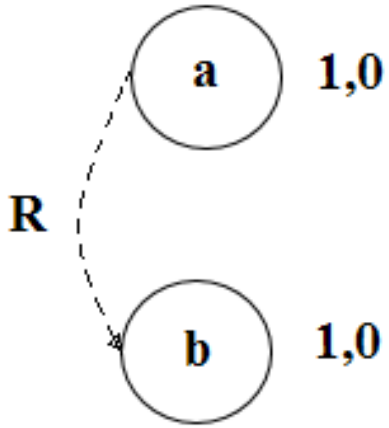


Рисунок 6.11 - Ограничение **R** (требует, Requires)

Часто возникает необходимость в **ограничениях для следствий**.

Ограничение **M** (скрывает, Masks, рисунок 6.12) устанавливает, что если следствие ***a*** имеет значение **1**, то следствие **b** должно принять значение **0**.

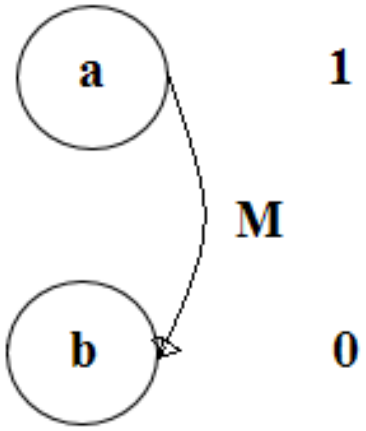


Рисунок 6.12 - Ограничение **M** (скрывает, Masks)

Пример

Иллюстрация использования способа, когда программа выполняет расчёт оплаты за электричество по среднему или переменному тарифу.

При расчёте **по среднему тарифу**:

– при месячном потреблении энергии меньшем, чем 100 кВт/ч, выставляется фиксированная сумма;

– при потреблении энергии большем или равном 100 кВт/ч применяется процедура ***A*** планирования расчёта.

При расчёте **по переменному тарифу**:

– при месячном потреблении энергии меньшем, чем 100 кВт/ч, применяется процедура ***A*** планирования расчёта;

– при потреблении энергии большем или равном 100 кВт/ч применяется процедура ***B*** планирования расчёта.

Шаг 1. **Причинами** являются:

1 расчет по среднему тарифу;

2 расчет по переменному тарифу;

3 месячное потребление электроэнергии меньшее, чем 100 кВт/ч;

4 месячное потребление электроэнергии большее или равное 100 кВт/ч.

На основе различных комбинаций причин можно перечислить следующие **следствия:**

101 — минимальная месячная стоимость;

102 — процедура *A* планирования расчета;

103 — процедура *B* планирования расчета.

Шаг 2. Разработка графа причинно-следственных связей (рисунок 6.13).

Узлы причин перечислим по вертикали у левого края рисунка, а узлы следствий — у правого края рисунка. Для следствия 102 возникает необходимость введения вторичных причин: **11** и **12**, их размещаем в центральной части рисунка.

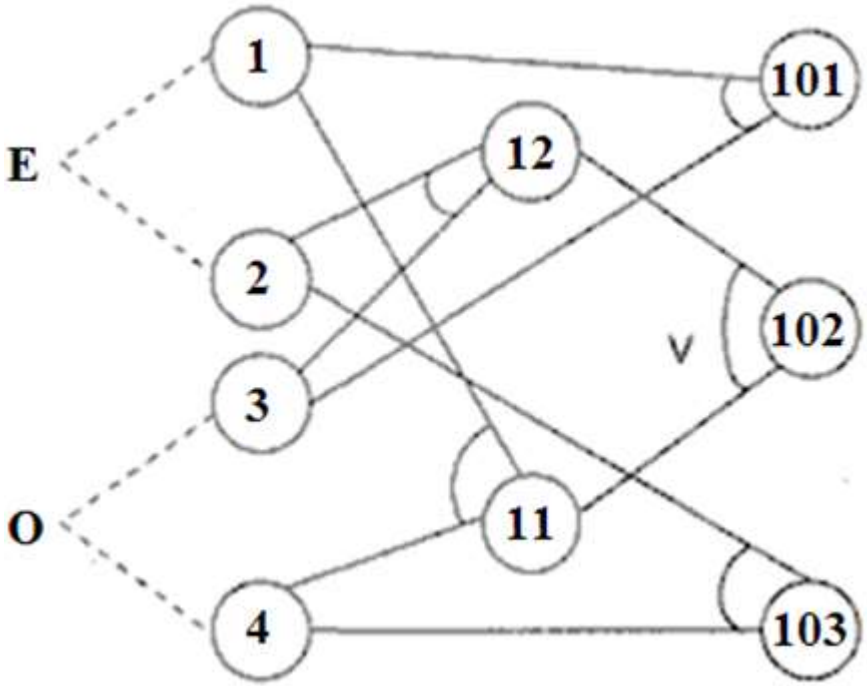
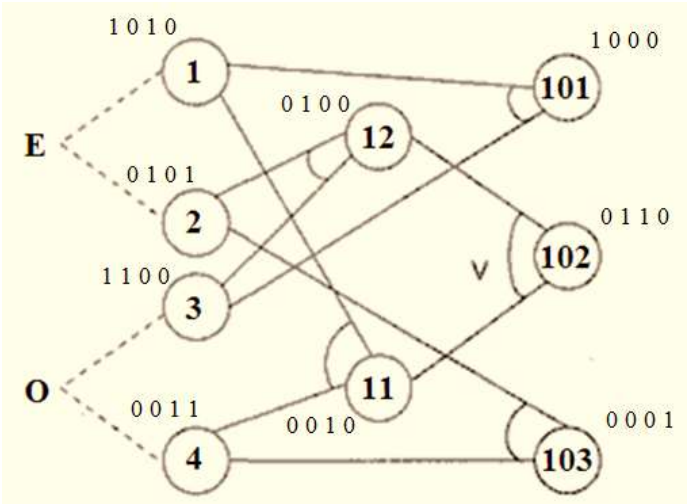


Рисунок 6.13 - Граф причинно-следственных связей

Шаг 3. Генерация таблицы решений. При генерации причины рассматриваются как условия, а следствия — как действия.

Порядок генерации:

1) Выбирается некоторое следствие, которое должно быть в состоянии «**1**».
2) Находятся все комбинации причин (с учетом ограничений), которые устанавливают это следствие в состояние «**1**». Для этого из следствия прокладывается обратная трасса через граф.
3) Для каждой комбинации причин, приводящих следствие в состояние «**1**», строится один столбец.



4) Для каждой комбинации причин доопределяются состояния всех других следствий. Они помещаются в тот же столбец таблицы решений.
5) Действия 1÷ 4 повторяются для всех следствий графа.

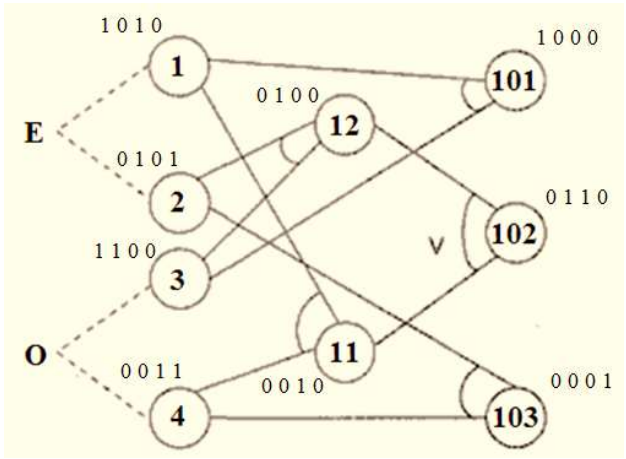
Таблице 6.1 - Таблица решений для расчёта оплаты на электричество

Номера столбцов →			1	2	3	4
Условия	**Причины**	1	1	0	1	0
		2	0	1	0	1
		3	1	1	0	0
		4	0	0	1	1
	Вторичные причины	11	0	0	1	0
		12	0	1	0	0
Действия	**Следствия**	101	1	0	0	0
		102	0	1	1	0
		103	0	0	0	1

Таблица решений показана в таблице.

Номера столбцов →			1	2	3	4
Условия	**Причины**	1	1	0	1	0
		2	0	1	0	1
		3	1	1	0	0
		4	0	0	1	1
	Вторичные причины	11	0	0	1	0
		12	0	1	0	0
Действия	**Следствия**	101	1	0	0	0
		102	0	1	1	0
		103	0	0	0	1

Шаг 4. Преобразование каждого столбца таблицы в тестовый вариант. В данном примере таких вариантов четыре.



ТВ1 *(столбец 1)***:**

ИД*:* расчёт по среднему тарифу; месячное потребление электроэнергии 75 кВт/ч.

ОЖ.РЕЗ*.:* минимальная месячная стоимость.

ТВ2 *(столбец 2)***:**

ИД*:* расчёт по переменному тарифу; месячное потребление электроэнергии 90 кВт/ч.

ОЖ.РЕЗ*.:* процедура *A* планирования расчёта.

ТВ3 *(столбец 3)***:**

ИД*:* расчёт по среднему тарифу; месячное потребление электроэнергии 100 кВт/ч.

ОЖ.РЕЗ*.:* процедура *A* планирования расчета.

ТВ4 *(столбец 4)***:**

ИД*:* расчёт по переменному тарифу; месячное потребление электроэнергии 100 кВт/ч.

ОЖ.РЕЗ*.:* процедура *B* планирования расчёта.

Метод "Разбиение на категории"

Остранд и Бальцер (Ostrand and Balcer) разработали метод разбиения, который позволяет испытателям анализировать спецификацию системы, создавать тестовые сценарии и управлять ими. Если большинство стратегий сфокусированы на работе с исходным кодом, метод Остранда и Бальцера также предполагает использование данных спецификации и проектирования.

Главное преимущество этого метода в том, что он позволяет обнаруживать ошибки еще до создания кода, потому что источником ввода является спецификация, и тестирование основано на ее анализе. Недоработки в спецификациях можно обнаружить на ранних стадиях, часто до ее реализации в коде.

Шаги для применения **метода "разбиение на категории":**

1. Проанализировать спецификацию: разложить набор функций системы на функциональные единицы, которые можно протестировать независимо друг от друга как с помощью спецификации, так и реализации:

    1) Определить параметры и условия среды, от которых зависит выполнение функции. Параметры — это входные данные функциональной единицы. Условия среды — это состояния системы, от которых зависит выполнение функциональной единицы.
    2) Определить характеристики параметров и условий среды.
    3) Разбить характеристики на категории, от которых зависит поведение системы.

    На этой стадии будут обнаружены неоднозначные, противоречащие и недостающие элементы описания поведения системы.

2. Разделить категории на варианты. Варианты — это ситуации, которые могут неожиданно возникнуть. В каждом варианте содержится тот же тип данных, что и в категории.

3. Определить ограничения и отношения между вариантами. Варианты из разных категорий взаимосвязаны, и это нужно учитывать при составлении тестового сценария. Ограничения предотвращают возникновение противоречий между вариантами с разными параметрами или из разных сред.

4. Опираясь на информацию о категориях, вариантах и ограничениях, разработать тестовые сценарии. Если при использовании какого-то варианта возникает ошибка, не добавлять его к другим вариантам при составлении сценария. Если вариант можно полноценно протестировать за один тест, значит это частный случай варианта или специальное значение.

Функциональное и нефункциональное тестирование:

Функциональное тестирование	Нефункциональное тестирование
Функциональное тестирование выполняется с использованием функциональной спецификации, предоставленной клиентом, и проверяет систему на соответствие	Нефункциональное тестирование проверяет производительность, надежность, масштабируемость и другие нефункциональные аспекты системы программного обеспечения.
Функциональное тестирование выполняется первым	Нефункциональное тестирование должно выполняться после функционального
Для функционального тестирования могут использоваться инструменты ручного тестирования или автоматизации.	Использование инструментов будет эффективным для этого тестирования
Бизнес-требования являются входными данными для функционального тестирования	Параметры производительности, такие как скорость, масштабируемость, являются входными данными для нефункционального
Функциональное тестирование описывает, что делает продукт	Нефункциональное тестирование описывает, насколько хорошо работает продукт

Функциональное тестирование	Нефункциональное тестирование
Легко сделать ручное тестирование	Трудно сделать ручное тестирование
Примеры функционального тестирования: • Модульное тестирование • Тестирование дыма • Тестирование в здравом уме • Интеграционное тестирование • Тестирование белого ящика • Тестирование черного ящика • Пользовательское тестирование • Регрессионное тестирование	Примеры нефункционального тестирования: • Тестирование производительности • Нагрузочное тестирование • Объемное тестирование • Нагрузочное тестирование • Тестирование безопасности • Тестирование установки • Проверка на проницаемость • Тестирование совместимости • Миграционное тестирование

Инструменты функционального тестирования

На рынке доступно несколько инструментов для проведения функционального тестирования. Они объясняются следующим образом:

- **Ranorex Studio** — многофункциональная автоматизированная система тестирования для настольных, веб-и мобильных приложений со встроенным Selenium WebDriver.
- **Selenium** — популярный инструмент функционального тестирования с открытым исходным кодом.
- **QTP** — Очень удобный инструмент функционального тестирования от HP.
- **JUnit** — используется в основном для приложений Java и может использоваться в модульном и системном тестировании.
- **soapUI** — это инструмент функционального тестирования с открытым исходным кодом, в основном используемый для тестирования веб-сервисов. Он поддерживает несколько протоколов, таких как HTTP, SOAP и JDBC.
- **Watir** — это функциональный инструмент тестирования для веб-приложений. Он поддерживает тесты, выполняемые в веб-браузере, и использует язык сценариев ruby.

Вывод

В тестировании программного обеспечения функциональное тестирование представляет собой процесс тестирования функциональных возможностей системы и гарантирует, что система работает в соответствии с функциональными возможностями, указанными в деловом документе. Цель этого тестирования — проверить, является ли система функционально совершенной!!!

ЛЕКЦИЯ 8

ТЕМА 7: Организация процесса тестирования программного продукта. Методика тестирования программных систем. Тестирование элементов

В этой лекции будут рассмотрены вопросы, связанные с проведением тестирования на всех этапах конструирования программной системы. Классический процесс тестирования обеспечивает проверку результатов, полученных на каждом этапе разработки. Как правило, он начинается с тестирования в малом, когда проверяются программные модули, продолжается при проверке объединения модулей в систему и завершается тестированием в большом, при котором проверяются соответствие программного продукта требованиям заказчика и его взаимодействие с другими компонентами компьютерной системы.

Методика тестирования программных систем

Процесс тестирования объединяет различные способы тестирования в спланированную последовательность шагов, которые приводят к успешному построению программной системы (ПС). Методика тестирования ПС может быть представлена в виде разворачивающейся спирали (рисунок 7.1).

Рисунок 7.1- Спираль процесса тестирования ПС

В начале осуществляется ***тестирование элементов (модулей)***, проверяющее результаты этапа *кодирования* ПС. На втором шаге выполняется ***тестирование интеграции***, ориентированное на выявление ошибок этапа *проектирования* ПС. На третьем обороте спирали производится ***тестирование правильности***, проверяющее корректность этапа *анализа требований* к ПС. На заключительном витке спирали проводится ***системное тестирование***, выявляющее дефекты этапа *системного анализа* ПС.

Охарактеризуем каждый **шаг процесса тестирования**.

- ***Тестирование элементов***. Цель — индивидуальная проверка каждого модуля. Используются способы тестирования «белого ящика».
- ***Тестирование интеграции***. Цель — тестирование сборки модулей в программную систему. В основном применяют способы тестирования «черного ящика».
- ***Тестирование правильности***. Цель — проверить реализацию в программной системе всех функциональных и поведенческих требований, а также требования эффективности. Используются исключительно способы тестирования «черного ящика».
- ***Системное тестирование***. Цель — проверка правильности объединения и взаимодействия всех элементов компьютерной системы, реализации всех системных функций.

Организация процесса тестирования в виде эволюционной разворачивающейся спирали обеспечивает максимальную эффективность поиска ошибок. Однако возникает вопрос — когда заканчивать тестирование?

Ответ практика обычно основан на статистическом критерии: «Можно с 95%-ной уверенностью сказать, что провели достаточное тестирование, если вероятность безотказной работы ЦП с программным изделием в течение 1000 часов составляет по меньшей мере 0,995».

Научный подход при ответе на вопрос (когда заканчивать тестирование?) состоит в применении математической модели отказов. Например, для логарифмической модели Пуассона формула расчета текущей интенсивности отказов имеет вид:

$$\lambda(t) = \frac{\lambda_0}{\lambda_0 \times p \times t + 1}, \qquad (7.1)$$

где λ(t) — текущая интенсивность программных отказов (количество отказов в единицу времени);

λ_0 — начальная интенсивность отказов (в начале тестирования);

p — экспоненциальное уменьшение интенсивности отказов за счет обнаруживаемых и устраняемых ошибок;

t — время тестирования.

С помощью уравнения (7.1) можно предсказать снижение ошибок в ходе тестирования, а также время, требующееся для достижения допустимо низкой интенсивности отказов.

Тестирование элементов

Объектом тестирования элементов является наименьшая единица проектирования ПС — **модуль**. Для обнаружения ошибок в рамках модуля тестируются его важнейшие управляющие пути. Относительная сложность тестов и ошибок определяется как результат ограничений области тестирования элементов. Принцип тестирования — «белый ящик», шаг может выполняться для набора модулей параллельно.

Тестированию подвергаются:

a) интерфейс модуля;
b) внутренние структуры данных;
c) независимые пути;
d) пути обработки ошибок;
e) граничные условия.

Интерфейс модуля (a) тестируется для проверки правильности ввода-вывода тестовой информации. Если нет уверенности в правильном вводе-выводе данных, нет смысла проводить другие тесты.

Исследование **внутренних структур данных** (b) гарантирует целостность сохраняемых данных.

Тестирование **независимых путей** (c) гарантирует однократное выполнение всех операторов модуля. При тестировании путей выполнения обнаруживаются следующие категории ошибок: ошибочные вычисления, некорректные сравнения, неправильный поток управления.

Наиболее **общими ошибками** вычислений являются:

1) неправильный или непонятый приоритет арифметических операций;
2) смешанная форма операций;
3) некорректная инициализация;
4) несогласованность в представлении точности;
5) некорректное символическое представление выражений.

Источниками ошибок сравнения и неправильных потоков управления являются:

1) сравнение различных типов данных;
2) некорректные логические операции и приоритетность;
3) ожидание эквивалентности в условиях, когда ошибки точности делают эквивалентность невозможной;
4) некорректное сравнение переменных;
5) неправильное прекращение цикла;
6) отказ в выходе при отклонении итерации;
7) неправильное изменение переменных цикла.

Обычно при проектировании модуля предвидят некоторые ошибочные условия. Для защиты от ошибочных условий в модуль вводят **пути обработки ошибок** (d). Такие пути тоже должны тестироваться. Тестирование путей обработки ошибок можно ориентировать на следующие ситуации:

1) донесение об ошибке невразумительно;
2) текст донесения не соответствует обнаруженной ошибке;
3) вмешательство системных средств регистрации аварии произошло до обработки ошибки в модуле;
4) обработка исключительного условия некорректна;
5) описание ошибки не позволяет определить ее причину.

Перейдем к **граничному тестированию** (e). Модули часто отказывают на «границах». Это означает, что ошибки часто происходят:

1) при обработке n - го элемента n - элементного массива;
2) при выполнении *m* - *й* итерации цикла с *m* проходами;
3) при появлении минимального (максимального) значения.

Тестовые варианты, ориентированные на данные ситуации, имеют высокую вероятность обнаружения ошибок.

Тестирование элементов обычно рассматривается как дополнение к этапу кодирования. Оно начинается после разработки текста модуля. Так как модуль не является автономной системой, то для реализации тестирования требуются дополнительные средства, представленные на рисунке 7.2.

Дополнительными средствами являются **драйвер тестирования** и **заглушки**.

Драйвер — управляющая программа, которая принимает исходные данные (InData) и ожидаемые результаты (ExpRes) тестовых вариантов, запускает в работу тестируемый модуль, получает из модуля реальные результаты (OutData) и формирует донесения о тестировании. Алгоритм работы тестового драйвера приведен на рисунке 7.3.

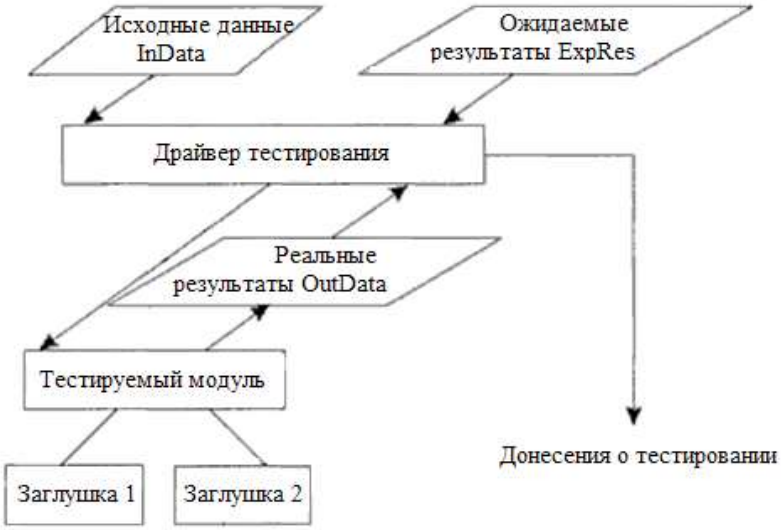


Рисунок 7.2 - Программная среда для тестирования модуля

Заглушки замещают модули, которые вызываются тестируемым модулем. Заглушка, или «фиктивная подпрограмма», реализует интерфейс подчиненного модуля, может выполнять минимальную обработку данных, имитирует прием и возврат данных.

Создание драйвера и заглушек подразумевает дополнительные затраты, так как они не поставляются с конечным программным продуктом.

Если эти средства просты, то дополнительные затраты невелики. Увы, многие модули не могут быть адекватно протестированы с помощью простых дополнительных средств. В этих случаях полное тестирование может быть отложено до шага тестирования интеграции (где драйверы или заглушки также используются).

Тестирование элемента просто осуществить, если модуль имеет высокую связность. При реализации модулем только одной функции количество тестовых вариантов уменьшается, а ошибки легко предсказываются и обнаруживаются.

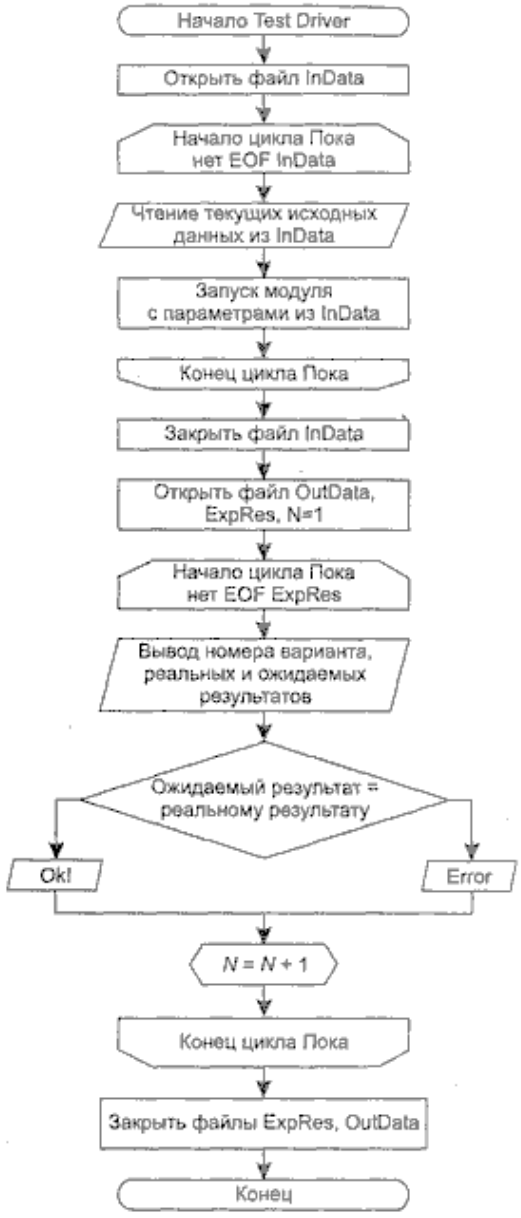


Рисунок 7.3 - Алгоритм работы драйвера тестирования

ЛЕКЦИЯ 9

ТЕМА 8: Тестирование интеграции. Монолитное и пошаговое тестирования. Нисходящее тестирование интеграции. Восходящее тестирование интеграции. Сравнительный анализ

Тестирование интеграции

Тестирование интеграции поддерживает сборку цельной программной системы, состоящей из двух и более модулей.

Интеграционное тестирование — это тестирование корректности взаимодействия нескольких модулей, объединенных в единое целое.

Цель сборки и тестирования интеграции:

- взять модули, протестированные как элементы, и построить программную структуру, требуемую проектом;
- удостовериться в корректности совместной работы компонентов системы;
- проверить соответствия проектируемых единиц функциональным, приёмным и требованиям надежности.

Тесты проводятся для обнаружения ошибок интерфейса.

Некоторые **категории ошибок интерфейса:**

a) потеря данных при прохождении через интерфейс;

b) отсутствие в модуле необходимой ссылки;

c) неблагоприятное влияние одного модуля на другой;

d) подфункции при объединении не образуют требуемую главную функцию;

e) отдельные (допустимые) неточности при интеграции выходят за допустимый уровень;

f) проблемы при работе с глобальными структурами данных.

Реализация процесса тестирования модулей опирается на два ключевых положения:

• построение эффективного набора тестов;

• выбор способа комбинирования модулей при построении из них рабочей программы — задает форму написания тестов модуля, типы средств, используемых при тестировании, порядок кодирования и тестирования модулей, стоимость генерации тестов и стоимость отладки.

Подходы к комбинированию модулей:

- монолитное тестирование;
- пошаговое тестирование.

- **Метод монолитного тестирования** (монолитный метод «большого удара») предполагает выполнение по отдельности тестирования каждого модуля, их комбинирования и формирование рабочей программы.

Монолитное тестирование предполагает, что отдельные компоненты системы серьезного тестирования не проходили.

Основное преимущество – отсутствие необходимости в разработке тестового окружения, драйверов и заглушек.

После разработки всех модулей выполняется их интеграция, затем система проверяется вся в целом, как она есть.

Недостатки:

- очень трудно выявить источник ошибки (идентифицировать ошибочный фрагмент кода);
- трудно организовать исправление ошибок;
- процесс тестирования плохо автоматизируется.

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

Сравнительный анализ данных подходов выявил следующие **особенности**.

1. **Монолитное тестирование** требует больших затрат труда. (+)

2. Расход машинного времени при **монолитном тестировании** меньше. (-)

3. Использование **монолитного метода** предоставляет большие возможности для параллельной организации работы на начальной фазе тестирования (тестирования всех модулей одновременно). Это имеет важное значение при выполнении больших проектов, в которых много модулей. (-)

4. При **пошаговом тестировании** раньше обнаруживаются ошибки в интерфейсах между модулями, поскольку раньше начинается сборка программы. При монолитном тестировании модули «не видят друг друга» до последней фазы процесса тестирования. (+)

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

6. Результаты **пошагового тестирования** более совершенны. (+)

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

Существует шесть основных методов проведения интеграционного тестирования:

- нисходящее тестирование;
- модифицированное нисходящее тестирование;
- метод «большого скачка»;
- восходящее тестирование;
- метод сэндвича.

Рассмотрим каждый из них.

Нисходящее тестирование интеграции

В данном подходе модули объединяются движением сверху вниз по управляющей иерархии, начиная от главного управляющего модуля. Подчиненные модули добавляются в структуру или в результате **поиска в глубину**, или в результате **поиска в ширину**.

Рассмотрим пример (рисунок 8.1). Интеграция **поиском в глубину** будет подключать все модули, находящиеся на главном управляющем пути структуры (по вертикали). Выбор главного управляющего пути отчасти произволен и зависит от характеристик, определяемых приложением. Например, при выборе левого пути прежде всего будут подключены модули M1, M2, M5. Следующим подключается модуль M8 или M6 (если это необходимо для правильного функционирования M2). Затем строится центральный или правый управляющий путь.

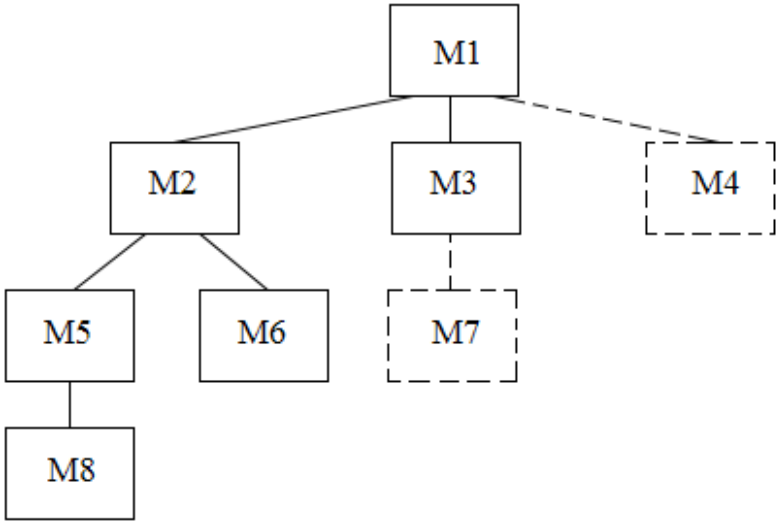


Рисунок 8.1 - Нисходящая интеграция системы

При **интеграции поиском в ширину** структура последовательно проходится по уровням-горизонталям. На каждом уровне подключаются модули, непосредственно подчиненные управляющему модулю — начальнику. В этом случае прежде всего подключаются модули М2, М3, М4. На следующем уровне — модули М5, М6 и т. д.

Возможные **шаги процесса нисходящей интеграции**:

Шаг 1. Главный управляющий модуль (находится на вершине иерархии) используется как тестовый драйвер. Все непосредственно подчиненные ему модули временно замещаются заглушками.

Шаг 2. Одна из заглушек заменяется реальным модулем. Модуль выбирается поиском в ширину или в глубину.

Шаг 3. После подключения каждого модуля (и установки на нем заглушек) проводится набор тестов, проверяющих полученную структуру.

Шаг 4. Если в модуле-драйвере уже нет заглушек, производится смена модуля-драйвера (поиском в ширину или в глубину).

Шаг 5. Выполняется возврат на шаг 2 (до тех пор, пока не будет построена целая структура).

Достоинство нисходящей интеграции: ошибки в главной, управляющей части системы выявляются в первую очередь.

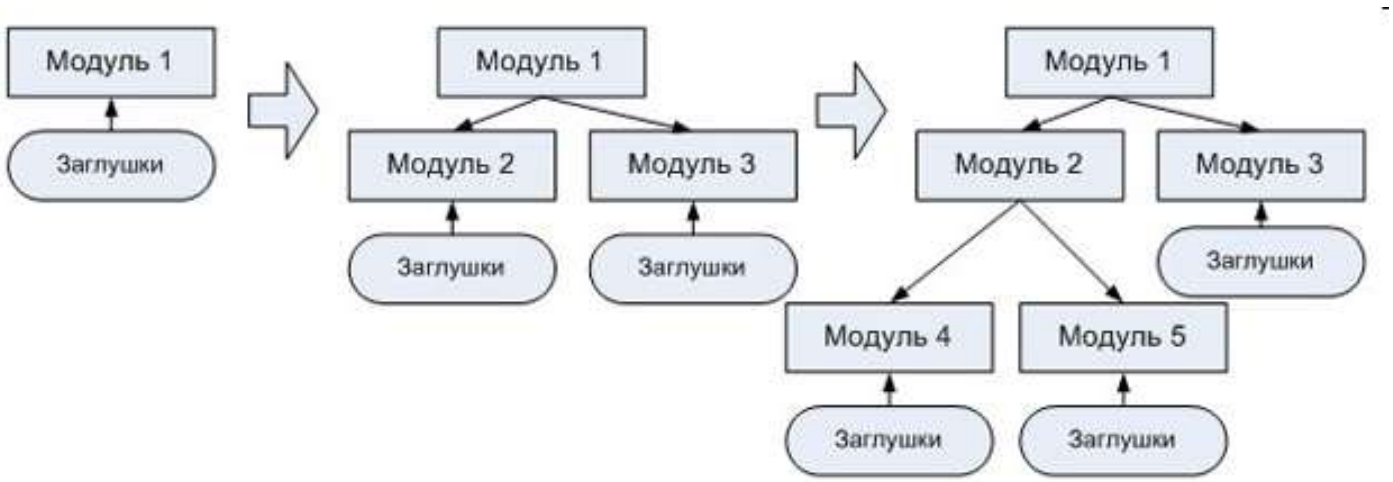
Недостаток: трудности в ситуациях, когда для полного тестирования на верхних уровнях нужны результаты обработки с нижних уровней иерархии. В результате применения такого метода отпадает необходимость в драйверах (роль драйвера выполняет более высокоуровневый модуль системы), однако сохраняется нужда в заглушках.

Существуют 3 возможности борьбы с этим недостатком:

a) откладывать некоторые тесты до замещения заглушек модулями;

b) разрабатывать заглушки, частично выполняющие функции модулей;

c) подключать модули движением снизу вверх.



Первая возможность вызывает сложности в оценке результатов тестирования.

Для реализации *второй возможности* выбирается одна из следующих **категорий заглушек**:

a) заглушка A — отображает трассируемое сообщение;

b) заглушка B — отображает проходящий параметр;

c) заглушка C — возвращает величину из таблицы;

d) заглушка D — выполняет табличный поиск по ключу (входному параметру) и возвращает связанный с ним выходной параметр.

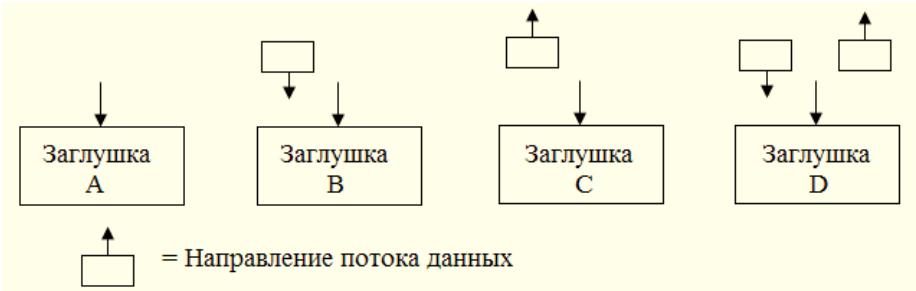


Рисунок 8.2- Категории заглушек

Категории заглушек представлены на рисунке 8.2.

Очевидно, что заглушка A наиболее проста, а заглушка D наиболее сложна в реализации.

Этот подход работоспособен, но может привести к существенным затратам, так как заглушки становятся все более сложными.

Модифицированное нисходящее тестирование

Нисходящий подход имеет еще один существенный недостаток, касающийся полноты тестирования. Предположим, что есть большая программа и где-то ближе к нижнему ее уровню находится модуль, предназначенный для вычисления корней квадратного уравнения. Для заданных входных переменных *A, B* и C он решает уравнение $AX^2 + BX + C = 0$.

При проектировании и программировании модуля с такой функцией всегда следует понимать, что квадратное уравнение может иметь как действительные, так и комплексные корни. Для полной реализации этой функции необходимо, чтобы результаты могли быть действительными или комплексными числами (или, если дополнительные затраты на нахождение комплексных корней неоправданные, модуль должен по крайней мере возвращать код ошибки в случае, когда входные коэффициенты задают уравнение с комплексными корнями). Предположим, что конкретный контекст, в котором используется модуль, исключает комплексные корни (т. е. вызывающие модули никогда не задают входных параметров, которые привели бы к комплексным корням). При строго нисходящем методе иногда бывает невозможно тестировать модуль для случая комплексных корней (или тестировать ошибочные условия). Можно попытаться оправдывать это тем, что, поскольку такое уравнение никогда не будет дано модулю, никого не должно заботить, работает ли он и в этих случаях. Да, это безразлично сейчас, но окажется важным в будущем, когда кто-то попытается использовать модуль в новой программе или модифицировать старую программу так, что станут возможными и комплексные корни.

Эта проблема проявляется в разнообразных формах. Применяя **нисходящее тестирование** в точном соответствии с предыдущим разделом, часто *невозможно тестировать определенные логические условия*, например ошибочные ситуации или защитные проверки. **Нисходящий метод**, кроме того, *делает сложной или вообще невозможной проверку исключительных ситуаций в некотором модуле, если программа работает с ним лишь в ограниченном контексте* (это означает, что модуль никогда не получит достаточно полный набор входных значений). Даже если тестирование такой ситуации в принципе осуществимо, часто бывает трудно определить, какие именно нужны тесты, если они вводятся в точке программы, удаленной от места проверки соответствующего условия.

Метод, называемый **модифицированным нисходящим подходом**, решает эти проблемы: требуется, чтобы каждый модуль прошел автономное тестирование перед подключением к программе. Хотя это действительно решает все перечисленные проблемы, здесь требуются и драйверы, и «заглушки» для каждого модуля.

Метод «большого скачка»

В соответствии с этим методом каждый модуль тестируется **автономно**. По окончании тестирования модулей они интегрируются в систему все сразу.

Недостатки:

- «заглушки» и драйверы необходимы для каждого модуля;
- модули не интегрируются до самого последнего момента, в течение долгого времени серьезные ошибки в сопряжениях могут остаться необнаруженными;
- для крупных программ метод «большого скачка» обычно губителен.

Достоинства:

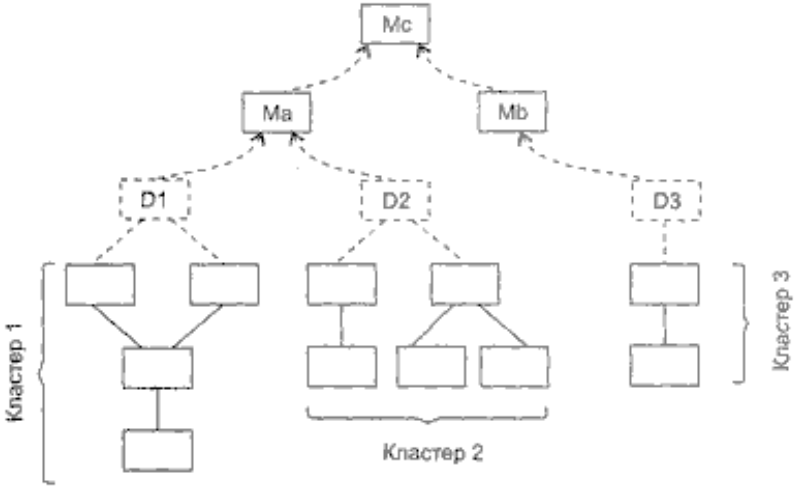
- если программа мала (как, например, программа загрузчика) и хорошо спроектирована, метод может оказаться приемлемым.

Третью возможность обсудим отдельно (подключать модули движением снизу вверх).

Восходящее тестирование интеграции

При восходящем тестировании интеграции сборка и тестирование системы начинаются с модулей-атомов, располагаемых на нижних уровнях иерархии. Модули подключаются движением снизу вверх. Подчиненные модули всегда доступны, и нет необходимости в заглушках.

Шаги методики восходящей интеграции.



Шаг 1. Модули нижнего уровня объединяются в кластеры (группы, блоки), выполняющие определенную программную подфункцию.

Шаг 2. Для координации вводов-выводов тестового варианта пишется драйвер, управляющий тестированием кластеров.

Шаг 3. Тестируется кластер.

Шаг 4. Драйверы удаляются, а кластеры объединяются в структуру движением вверх.

Пример восходящей интеграции системы приведен на рисунке 8.3.

Модули объединяются в кластеры 1, 2, 3. Каждый кластер тестируется драйвером. Модули в кластерах 1 и 2 подчинены модулю Ma, поэтому драйверы D1 и D2 удаляются и кластеры подключают прямо к Ma. Аналогично драйвер D3 удаляется перед подключением кластера 3 к модулю Mb. В последнюю очередь к модулю Mc подключаются модули Ma и Mb.

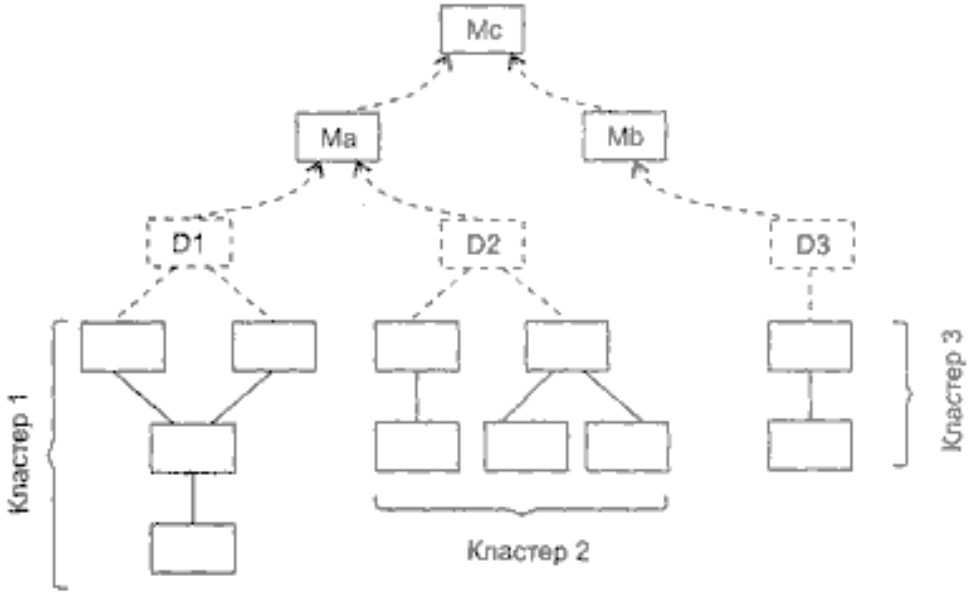


Рисунок 8.3 - Восходящая интеграция системы

Рассмотрим различные **типы драйверов**:

a) драйвер A — вызывает подчиненный модуль;

b) драйвер B — посылает элемент данных (параметр) из внутренней таблицы;

c) драйвер C — отображает параметр из подчиненного модуля;

d) драйвер D — является комбинацией драйверов B и C.

Очевидно, что драйвер A наиболее прост, а драйвер D наиболее сложен в реализации. Различные типы драйверов представлены на рисунке 8.4.

По мере продвижения интеграции вверх необходимость в выделении драйверов уменьшается. Как правило, в двухуровневой структуре драйверы не нужны.

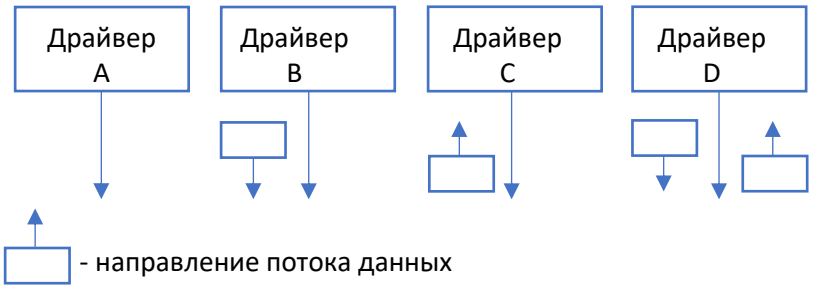
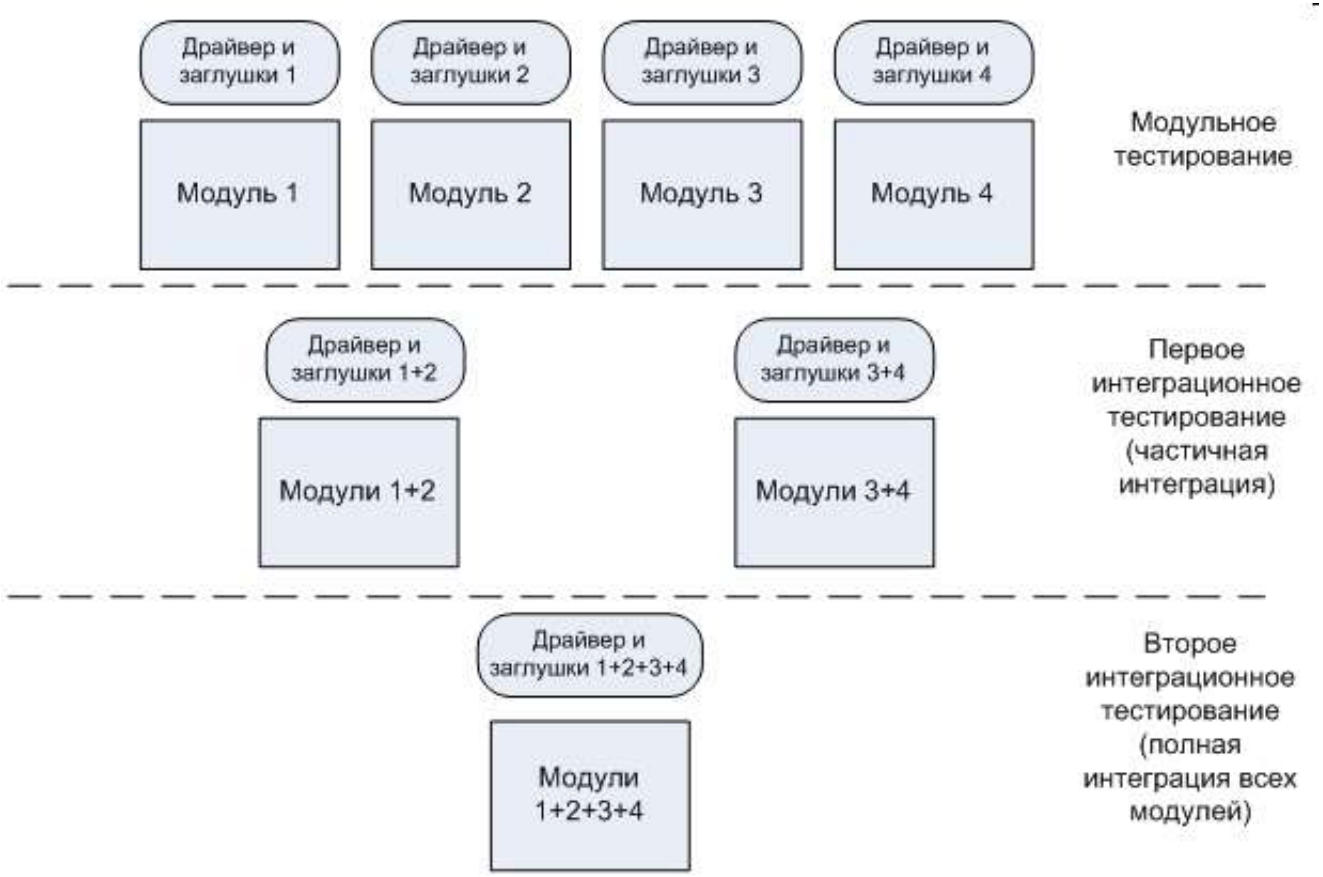


Рисунок 8.4 - Различные типы драйверов

Сначала тестируются все программные модули, входящие в состав системы и только затем они объединяются для интеграционного тестирования.



Метод сэндвича

Представляет собой *компромисс* между восходящим и нисходящим подходами.

Одновременно начинают восходящее и нисходящее тестирование, собирая программу как снизу, так и сверху и встречаясь в конце концов где-то в середине. Точка встречи зависит от конкретной тестируемой программы и должна быть заранее определена при изучении ее структуры. Метод сохраняет такое достоинство нисходящего и восходящего подходов, как начало интеграции системы на самом раннем этапе. Поскольку вершина программы вступает в строй рано как в нисходящем методе, уже на раннем этапе получаем работающий каркас программы. Так как нижние уровни программы создаются восходящим методом, снимаются проблемы нисходящего метода, связанные с невозможностью тестировать некоторые условия в глубине программы.

Проблема заключается в невозможности досконального тестирования отдельных модулей. Восходящий этап тестирования по методу сэндвича решает эту проблему для модулей нижних уровней, но она может по-прежнему оставаться открытой для нижней половины верхней части программы.

Модифицированный метод сэндвича

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

Сравнительная характеристика методов тестирования

С точки зрения надежности программного обеспечения эти стратегии можно оценить по семи критериям, как показано в таблице 8.1:

Первый критерий — время до момента сборки модулей, поскольку это важно для обнаружения ошибок в сопряжениях и предположениях модулей о свойствах друг друга.

Второй критерий — время до момента создания первых работающих «скелетных» версий программы, поскольку здесь могут проявиться главные дефекты проектирования.

Третий и четвертый критерии касаются вопроса о том, необходимы ли «заглушки», драйверы и другие инструменты тестирования.

Пятый критерий — мера параллелизма, который возможен в начале или на ранних стадиях тестирования. Это интересный вопрос, поскольку необходимость в ресурсах (т. е. программистах) обычно достигает пика на этапах проектирования и программирования модулей.

Поэтому важно, чтобы возможность параллельного тестирования появилась ближе к началу, а не к концу цикла тестирования.

Шестой критерий связан с ответом на обсуждавшийся ранее вопрос: возможно ли проверить любой конкретный путь и любое условие в программе? Седьмой критерий характеризует сложность планирования, надзора и управления в процессе тестирования.

Таблица 8.1 - Критериальная оценка методов тестирования

№	Критерий	Восходящий	Нисходящий	Модифицированный нисходящий	Метод «большого скачка»	Метод сэндвича	Модифицированный метод сэндвича
1	**Сборка**	Рано	Рано	Рано	Поздно	Рано	Рано
2	**Время до появления работающего варианта программы**	Поздно	Рано	Рано	Поздно	Рано	Рано
3	**Нужны ли драйверы (новые программы или готовые инструменты)?**	Да	Нет	Да	Да	Частично	Да
4	**Нужны ли «заглушки»**	Нет	Да	Да	Да	Частично	Частично
5	**Параллелизм в начале работы**	Средний	Слабый	Средний	Высокий	Средний	Высокий

6	**Возможность тестировать отдельные пути**	Легко	Трудно	Легко	Трудно	Средне	Легко
7	**Возможность планировать и контролировать последовательность**	Легко	Трудно	Трудно	Легко	Трудно	Трудно

Это связано с осознанием того факта, что тестирование, которым трудно управлять, часто ведет к недосмотрам и упущениям. Время от времени раздаются возражения против нисходящего подхода в связи с тем, что тестирование нижних модулей требует многократных лишних прогонов головных модулей. Однако этот критерий отмечен как несущественный. Хотя лишние прогоны действительно бывают необходимы, возможно также, что во многих случаях, которые кажутся лишними, в действительности воссоздаются несколько разные условия. Эти прогоны могут вскрыть новые ошибки, превращая таким образом недостаток в достоинство. Поскольку этот эффект недостаточно осознан, мы им пренебрегаем.

Теперь оценим наши шесть подходов с помощью перечисленных семи критериев. Как уже говорилось, такая оценка зависит от конкретного проекта. В качестве исходного приближения для выполнения ваших собственных оценок приведен вариант очень грубой оценки. Прежде всего следует взвесить относительное влияние каждого из семи критериев на надежность программного обеспечения. Ранняя сборка и раннее получение работающего каркаса программы, а также возможность тестировать любые конкретные условия представляются наиболее важными, поэтому им дается коэффициент 3. Сложность подготовки «заглушек», а также сложность планирования и управления последовательностью тестов также важны, поэтому они получают вес 2. Третий критерий, необходимость драйверов, имеет вес 1 ввиду доступности общих

инструментов тестирования. Критерий, связанный с параллелизмом работы, также имеет вес 1, потому что, хотя он, может быть, и важен по другим причинам, на надежность сильно не влияет. Седьмой критерий получает коэффициент нуль.

В таблице 8.2 показаны результаты этой оценки. В каждой графе таблицы вес берется со знаком плюс или минус либо не учитывается, в зависимости от того, благоприятно, неблагоприятно или безразлично проявляется соответствующий фактор при рассматриваемом подходе. Модифицированный метод сэндвича и восходящий метод оказываются наилучшими подходами, а метод большого скачка — наихудшим. Если способ оценки оказывается близким к вашей конкретной ситуации, следует рекомендовать модифицированный метод сэндвича для тестирования больших систем или программ и восходящий подход для тестирования программ малых и средних.

Таблица 8.2 - Весовая оценка методов тестирования

Вес	Метод Критерий	Восходящий	Нисходящий	Модифицированный нисходящий	Метод «большого скачка»	Метод сэндвича	Модифицированный метод сэндвича
3	Сборка	+	+	+	-	+	+
3	Время до появления работающего варианта программы	-	+	+	-	+	+

Вес	Метод Критерий	Восхо-дящий	Нисходящий	Модифицированный нисходящий	Метод «большого скачка»	Метод сэндвича	Модифицированный метод сэндвича
1	**Нужны ли драйверы (новые программы и/или готовые инструменты)**	—	+	—	—		—
2	**Нужны «заглушки»**	+	—	—	—		
1	**Параллелизм в начале работы**		—		+		+
3	**Возможность тестировать отдельные пути**	+	—	+	+		+
2	**Возможность планировать и контролировать последовательность действий**	+	—	—	+	—	—
0	**Неэффективность**						
Всего		+6	-1	+4	-3	+4	+7

Сравнение нисходящего и восходящего тестирования интеграции

Нисходящее тестирование:

a) *основной недостаток* — необходимость заглушек и связанные с ними трудности тестирования;

b) *основное достоинство* — возможность раннего тестирования главных управляющих функций.

Восходящее тестирование:

a) *основной недостаток* — система не существует как объект до тех пор, пока не будет добавлен последний модуль;

b) *основное достоинство* — упрощается разработка тестовых вариантов, отсутствуют заглушки.

Возможен комбинированный подход. В нем для верхних уровней иерархии применяют нисходящую стратегию, а для нижних уровней — восходящую стратегию тестирования**.**

Нисходящая и восходящая разработка программного обеспечения

При проектировании, реализации и тестировании компонентов структурной иерархии, полученной при декомпозиции, применяют два подхода:

- восходящий;

- нисходящий.

В литературе встречается еще один подход, получивший название «расширение ядра». Он предполагает, что в первую очередь проектируют и разрабатывают некоторую основу - ядро программного обеспечения, например, структуры данных и процедуры, связанные с ними. В дальнейшем ядро наращивают, комбинируя восходящий и нисходящий методы. На практике данный подход в зависимости от уровня ядра практически сводится либо к нисходящему, либо к восходящему подходам.

Восходящий подход. При использовании восходящего подхода сначала проектируют и реализуют компоненты нижнего уровня, затем предыдущего и т. д. По мере завершения тестирования и отладки компонентов осуществляют их сборку, причем компоненты нижнего уровня при таком подходе часто помещают в библиотеки компонентов.

Для тестирования и отладки компонентов проектируют и реализуют специальные тестирующие программы. Подход имеет следующие недостатки:

- увеличение вероятности несогласованности компонентов вследствие неполноты спецификаций;

- наличие издержек на проектирование и реализацию тестирующих программ, которые нельзя преобразовать в компоненты;

- позднее проектирование интерфейса, а соответственно невозможность продемонстрировать его заказчику для уточнения спецификаций и т. д.

Исторически восходящий подход появился раньше, что связано с особенностью мышления программистов, которые в процессе обучения привыкают при написании небольших программ сначала детализировать компоненты нижних уровней (подпрограммы, классы). Это позволяет им лучше осознавать процессы верхних уровней. При промышленном изготовлении программного обеспечения восходящий подход в настоящее время практически не используют.

Нисходящий подход. Нисходящий подход предполагает, что проектирование и последующая реализация компонентов выполняется «сверху-вниз», т. е. вначале проектируют компоненты верхних уровней иерархии, затем следующих и так далее до самых нижних уровней. В той же последовательности выполняют и реализацию компонентов. При этом в процессе программирования компоненты нижних, еще не реализованных уровней заменяют специально разработанными отладочными модулями - «заглушками», что позволяет тестировать и отлаживать уже реализованную часть.

При использовании нисходящего подхода применяют иерархический, операционный и комбинированный методы определения последовательности проектирования и реализации компонентов.

Иерархический метод предполагает выполнение разработки строго по уровням. Исключения допускаются при наличии зависимости по данным, т. е. если обнаруживается, что некоторый модуль использует результаты другого, то его рекомендуется программировать после этого модуля. Основной проблемой данного метода является большое количество достаточно сложных заглушек. Кроме того, при использовании данного метода основная масса модулей разрабатывается и реализуется в конце работы над проектом, что затрудняет распределение человеческих ресурсов.

Операционный метод связывает последовательность выполнения при запуске программы. Применение метода усложняется тем, что порядок выполнения модулей может зависеть от данных. Кроме того, модули вывода результатов несмотря на то, что они вызываются последними, должны разрабатываться одними из первых, чтобы не проектировать сложную заглушку, обеспечивающую вывод результатов при тестировании. С точки зрения распределения человеческих ресурсов сложным является начало работ, пока не закончены все модули, находящиеся на так называемом критическом пути.

Комбинированный метод учитывает следующие факторы, влияющие на последовательность разработки:

- достижимость модуля - наличие всех модулей в цепочке вызова данного модуля;

- зависимость по данным - модули, формирующие некоторые данные, должны создаваться раньше обрабатывающих;

- обеспечение возможности выдачи результатов - модули вывода результатов должны создаваться раньше обрабатывающих;

- готовность вспомогательных модулей - вспомогательные модули, например, модули закрытия файлов, завершения программы, должны создаваться раньше обрабатывающих;

- наличие необходимых ресурсов.

Кроме того, при прочих равных условиях сложные модули должны разрабатываться прежде простых, так как при их проектировании могут выявиться неточности в спецификациях, а чем раньше это произойдет, тем лучше.

Нисходящий подход допускает нарушение нисходящей последовательности разработки компонентов в специально оговоренных случаях. Так, если некоторый компонент нижнего уровня используется многими компонентами более высоких уровней, то его рекомендуют проектировать и разрабатывать раньше, чем вызывающие его компоненты. И, наконец, в первую очередь проектируют и реализуют компоненты, обеспечивающие обработку правильных данных, оставляя компоненты обработки неправильных данных напоследок.

Нисходящий подход обычно используют и при объектно-ориентированном программировании. В соответствии с рекомендациями подхода вначале проектируют и реализуют пользовательский интерфейс программного обеспечения, затем разрабатывают классы некоторых базовых объектов предметной области, а уже потом, используя эти объекты, проектируют и реализуют остальные компоненты. Нисходящий подход обеспечивает:

- максимально полное определение спецификаций проектируемого компонента и согласованность компонентов между собой;

- раннее определение интерфейса пользователя, демонстрация которого заказчику позволяет уточнить требования к создаваемому программному обеспечению:

- возможность нисходящего тестирования и комплексной отладки.

При проведении тестирования интеграции очень важно выявить **критические модули**.

Признаки критического модуля:

a) реализует несколько требований к программной системе;

b) имеет высокий уровень управления (находится достаточно высоко в программной структуре);

c) имеет высокую сложность или склонность к ошибкам (как индикатор может использоваться цикломатическая сложность — ее верхний разумный предел составляет 10);

d) имеет определенные требования к производительности обработки.

Критические модули должны тестироваться как можно раньше. Кроме того, к ним должно применяться регрессионное тестирование (повторение уже выполненных тестов в полном или частичном объеме).

На практике чаще всего в различных частях проекта применяются все рассмотренные в предыдущем разделе методы в совокупности. Каждый модуль тестируют по мере готовности отдельно, а потом включают в уже готовую композицию. Для одних частей тестирование получается нисходящим, для других – восходящим. В связи с этим представляется полезным рассмотреть еще один *тип классификации типов интеграционного тестирования* – **классификацию по частоте интеграции**:

- тестирование с поздней интеграцией;
- тестирование с постоянной интеграцией;
- тестирование с регулярной или послойной интеграцией.

Тестирование с поздней интеграцией – практически полный аналог монолитного тестирования. Интеграционное тестирование при такой схеме откладывается на как можно более поздние сроки проекта. Этот подход оправдывает себя в том случае, если система представляет собой конгломерат слабо связанных между собой модулей, которые взаимодействуют по какому-либо стандартному интерфейсу, определенному вне проекта (например, в случае если система состоит из отдельных Web-сервисов).

Схематично тестирование **с поздней интеграцией** может быть изображено в виде цепочки:

R-C-V-R-C-V-R-C-V-I-R-C-V-R-C-V-I,

где R – разработка требований на отдельный модуль, C – разработка программного кода, V – тестирование модуля, I – интеграционное тестирование всего, что было сделано раньше.

Тестирование с постоянной интеграцией подразумевает, что как только разрабатывается новый модуль системы, он сразу же интегрируется со всей остальной системой.

Тесты для этого модуля проверяют как сугубо его *внутреннюю функциональность*, так и *его взаимодействие с остальными модулями системы*. Таким образом, *этот подход совмещает в себе модульное тестирование и интеграционное*.

Разработки заглушек при таком подходе не требуется, но может потребоваться разработка **драйверов**. В настоящее время именно этот подход называют ***unit testing***, несмотря на то что в отличие от классического модульного тестирования здесь не проверяется функциональность изолированного модуля. Локализация ошибок межмодульных интерфейсов при таком подходе несколько затруднена, но все же значительно ниже, чем при монолитном тестировании. Большая часть таких ошибок выявляется достаточно рано именно за счет частоты интеграции и за счет того, что за одну итерацию тестирования проверяется сравнительно небольшое число межмодульных интерфейсов.

Схематично тестирование с **постоянной интеграцией** может быть изображено в виде цепочки:

R-C-I-R-C-I-R-C-I,

где R – разработка требований на отдельный модуль, C – разработка программного кода, I – интеграционное тестирование,

в которой *фаза тестирования* модуля намеренно опущена и заменена на тестирование интеграции.

При **тестировании с регулярной или послойной интеграцией** интеграционному тестированию подлежат сильно связанные между собой группы модулей (слои), которые затем также интегрируются между собой. Такой вид интеграционного тестирования называют также **иерархическим интеграционным тестированием**, поскольку укрупнение интегрированных частей системы, как правило, происходит по иерархическому принципу. Однако, в отличие от нисходящего или восходящего тестирования, направление прохода по иерархии в этом подходе не задано.

Таблица 8.3 представляет основные характеристики рассмотренных выше видов интеграционного тестирования. Время интеграции характеризует момент времени, когда проводится первое интеграционное тестирование и все последующие, частота интеграции – насколько часто при разработке выполняется интеграция. Необходимость в драйверах и заглушках определена в последних двух строках таблицы.

Таблица 8.3 - Основные характеристики различных видов интеграционного тестирования

	Восходящее	**Нисходящее**	**Монолитное**	**Поздняя интеграция**	**Постоянная интеграция**	**Регулярная интеграция**
Время интеграции	поздно (после тестирования модулей)	рано (параллельно с разработкой)	поздно (после разработки всех модулей)	поздно (после разработки всех модулей)	рано (параллельно с разработкой)	рано (параллельно с разработкой)
Частота интеграции	Редко	часто	редко	редко	часто	часто
Нужны ли драйверы	Да	нет	нет	нет	да	да
Нужны ли заглушки	Да	да	нет	нет	нет	да

ЛЕКЦИЯ 10

ТЕМА 8: Тестирование интеграции. Нисходящее тестирование интеграции. Восходящее тестирование интеграции. Монолитное тестирование. Сравнительный анализ

Декомпозиция подсистем на модули

Известны два типа моделей модульной декомпозиции:

- модель потока данных;
- модель объектов.

В основе **модели потока данных** лежит разбиение **по функциям**.

Модель объектов основана на слабо сцепленных сущностях, имеющих собственные наборы данных, состояния и наборы операций.

Очевидно, что выбор типа декомпозиции должен определяться сложностью разбиваемой подсистемы.

Модульность

Модуль — фрагмент программного текста, являющийся строительным блоком для физической структуры системы. Как правило, модуль состоит из интерфейсной части и части-реализации.

Модульность — свойство системы, которая может подвергаться декомпозиции на ряд внутренне связанных и слабо зависящих друг от друга модулей.

По определению Г. Майерса, модульность — свойство ПО, обеспечивающее интеллектуальную возможность создания сколь угодно сложной программы. Проиллюстрируем эту точку зрения.

Пусть $C(x)$ — функция сложности решения проблемы x, $T(x)$ — функция затрат времени на решение проблемы x. Для двух проблем p_1 и p_2 из соотношения $C(p_1) > C(p_2)$ следует, что

$$T(p_1) > T(p_2). \tag{8.1}$$

Этот вывод интуитивно ясен: решение сложной проблемы требует большего времени.

Далее. Из практики решения проблем человеком следует:

$$C(p_1 + p_2) > C(p_1) + C(p_2).$$

Отсюда с учетом соотношения (8.1) запишем:

$$T(p_1+p_2) > T(p_1) + T(p_2). \tag{8.2}$$

Соотношение $T(p_1+p_2) > T(p_1) + T(p_2)$. (8.2) — это обоснование модульности. Оно приводит к заключению «разделяй и властвуй» — сложную проблему легче решить, разделив ее на управляемые части. Результат, выраженный неравенством (8.2), имеет важное значение для модульности и ПО. Фактически, это аргумент в пользу модульности.

Однако здесь отражена лишь часть реальности, ведь здесь не учитываются затраты на межмодульный интерфейс. Как показано на рисунке 8.1, с увеличением количества модулей (и уменьшением их размера) эти затраты также растут.

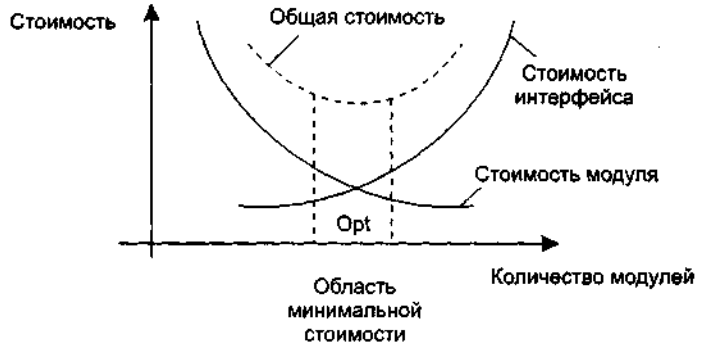


Рисунок 8.1 - Затраты на модульность

Таким образом, существует оптимальное количество модулей Opt, которое приводит к минимальной стоимости разработки. Увы, нет необходимого опыта для гарантированного предсказания Opt.

Разработчики знают, что **оптимальный модуль** должен удовлетворять двум критериям:

- снаружи он проще, чем внутри;
- его проще использовать, чем построить.

Информационная закрытость

Принцип информационной закрытости (автор — Д. Парнас, 1972) утверждает: содержание модулей должно быть скрыто друг от друга. Как показано на рисунке 8.2, модуль должен определяться и проектироваться так, чтобы его содержимое (процедуры и данные) было недоступно тем модулям, которые не нуждаются в такой информации (клиентам).

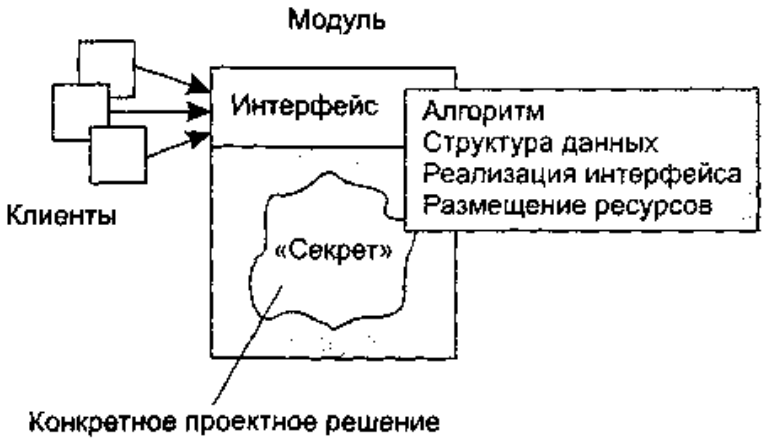


Рисунок 8. 2 - Информационная закрытость модуля

Информационная закрытость означает следующее:

a) все модули независимы, обмениваются только информацией, необходимой для работы;
b) доступ к операциям и структурам данных модуля ограничен.

Достоинства информационной закрытости:

- обеспечивается возможность разработки модулей различными, независимыми коллективами;
- обеспечивается легкая модификация системы (вероятность распространения ошибок очень мала, так как большинство данных и процедур скрыто от других частей системы).

Идеальный модуль играет роль «черного ящика», содержимое которого невидимо клиентам. Он прост в использовании — количество «ручек и органов управления» им невелико (аналогия с эксплуатацией телевизора). Его легко развивать и корректировать в процессе сопровождения программной системы. Для обеспечения таких возможностей система внутренних и внешних связей модуля должна отвечать особым требованиям.

Обсудим характеристики внутренних и внешних связей модуля.

Связность модуля

Связность модуля (Cohesion) — это мера зависимости его частей. Связность — внутренняя характеристика модуля. Чем выше связность модуля, тем лучше результат проектирования, то есть тем «черней» его ящик (капсула, защитная оболочка модуля), тем меньше «ручек управления» на нем находится и тем проще эти «ручки».

Для измерения связности используют понятие силы связности (СС). Существует **7 типов связности**:

a) **Связность по совпадению** (СС = 0). В модуле отсутствуют явно выраженные внутренние связи.

b) **Логическая связность** (СС = 1). Части модуля объединены по принципу функционального подобия.

Например, модуль состоит из разных подпрограмм обработки ошибок. При использовании такого модуля клиент выбирает только одну из подпрограмм.

Недостатки:

- сложное сопряжение;
- большая вероятность внесения ошибок при изменении сопряжения ради одной из функций.

3. **Временная связность** (CC = 3). Части модуля не связаны, но необходимы в один и тот же период работы системы.

   Недостаток: сильная взаимная связь с другими модулями, отсюда — сильная чувствительность внесению изменений.

4. **Процедурная связность** (CC = 5). Части модуля связаны порядком выполняемых ими действий, реализующих некоторый сценарий поведения.

5. **Коммуникативная связность** (CC = 7). Части модуля связаны по данным (работают с одной и той же структурой данных).

6. **Информационная (последовательная) связность** (CC = 9). Выходные данные одной части используются как входные данные в другой части модуля.

7. **Функциональная связность** (CC = 10). Части модуля вместе реализуют одну функцию.

Отметим, что типы связности 1,2,3 — результат неправильного планирования архитектуры, а тип связности 4 — результат небрежного планирования архитектуры приложения.

Общая характеристика типов связности представлена в таблице 8.1.

Таблица 8.1 - Характеристика связности модуля

Тип связности	**Сопровождаемость**	**Роль модуля**
Функциональная	Лучшая сопровождаемость	«Черный ящик»
Информационная (последовательная)		Не совсем «черный ящик»
Коммуникативная		«Серый ящик»
Процедурная	Худшая сопровождаемость	«Белый» или «просвечивающий ящик»
Временная		«Белый ящик»
Логическая		
По совпадению		

Функциональная связность (CC = 10)

Функционально связный модуль содержит элементы, участвующие в выполнении одной и только одной проблемной задачи. Модуль с функциональной связностью не может быть разбит на 2 других модуля, имеющих связность того же типа. Он выполняет единственную функцию и реализуется последовательностью операций в виде единичного цикла. Примеры функционально связных модулей:

- Вычислить синус угла;
- Проверить орфографию;
- Читать запись файла;
- Вычислять координаты цели;
- Вычислить зарплату сотрудника;
- Вычислить логарифм функции;
- Определять место пассажира.

Каждый из этих модулей имеет единичное назначение. Когда клиент вызывает модуль, выполняется только одна работа, без привлечения внешних обработчиков. Например, модуль Определять место пассажира должен делать только это; он не должен распечатывать заголовки страницы.

Некоторые из функционально связных модулей очень просты (например, Вычислять синус угла или Читать запись файла), другие сложны (например, Вычислять координаты цели). Модуль Вычислять синус угла, очевидно, реализует единичную функцию, но как может модуль Вычислять зарплату сотрудника выполнять только одно действие? Ведь каждый знает, что приходится определять начисленную сумму, вычеты по рассрочкам, подоходный налог, социальный налог, алименты и т. д.! Дело в том, что, несмотря на сложность модуля и на то, что его обязанность исполняют несколько подфункций, если его действия можно представить как *единую проблемную функцию* (с точки зрения клиента), тогда считают, что **модуль функционально связен**.

Приложения, построенные из функционально связных модулей, *легче всего сопровождать*. Соблазнительно думать, что любой модуль можно рассматривать как однофункциональный, но не надо заблуждаться. Существует много разновидностей модулей, которые выполняют для клиентов перечень различных работ, и этот перечень нельзя рассматривать как единую проблемную функцию. *Критерий при определении уровня связности этих нефункциональных модулей — как связаны друг с другом различные действия, которые они исполняют*.

Информационная связность (CC = 9)

При информационной (последовательной) связности элементы-обработчики модуля образуют конвейер для обработки данных — результаты одного обработчика используются как исходные данные для следующего обработчика. Модуль с функциональной связностью не может быть разбит на 2 других модуля, имеющих связность того же типа. Он выполняет единственную функцию и реализуется последовательностью операций в виде единичного цикла. Приведем **пример:**

Модуль Прием и проверка записи

прочитать запись из файла

проверить контрольные данные в записи

удалить контрольные поля в записи

вернуть обработанную запись

Конец модуля

В этом модуле 3 элемента. Результаты первого элемента (прочитать запись из файла) используются как входные данные для второго элемента (проверить контрольные данные в записи) и т. д.

Сопровождать модули с информационной связностью почти так же легко, как и функционально связные модули. Правда, возможности повторного использования здесь ниже, чем в случае функциональной связности. Причина — совместное применение действий модуля с информационной связностью полезно далеко не всегда.

Коммуникативная связность (CC = 7)

При коммуникативной связности элементы-обработчики модуля используют одни и те же данные, например внешние данные. Модуль, имеющий коммуникативную связь, может быть разбит на независимые модули, разделяющие общую структуру данных. В основе данного модуля лежит общая структура данных.

Модули высшего уровня иерархической структуры программы должны иметь функциональную или последовательную связь. Для модуля обслуживания предпочтительнее коммуникативная связь.

Пример коммуникативно связного модуля:

Модуль Отчет и средняя зарплата

используется Таблица зарплаты служащих
сгенерировать Отчет по зарплате
вычислить параметр Средняя зарплата
вернуть Отчет по зарплате. Средняя зарплата
Конец модуля

Здесь все элементы модуля работают со структурой Таблица зарплаты служащих.

С точки зрения клиента проблема применения коммуникативно связного модуля состоит в *избыточности получаемых результатов*. Например, клиенту требуется только отчет по зарплате, он не нуждается в значении средней зарплаты. Такой клиент будет вынужден выполнять избыточную работу — выделение в полученных данных материала отчета. Почти всегда разбиение коммуникативно связного модуля на отдельные функционально связные модули улучшает сопровождаемость системы.

Пример 2: по ISBN книги, можно узнать ее название, автора и год издания. Эти три процедуры (определить название, определить автора, определить год издания) связаны между собой тем, что все они работают с одним и тем же информационным объектом – ISBN.

Попытаемся провести аналогию между **информационной** и **коммуникативной** связностью.

Модули с коммуникативной и информационной связностью подобны в том, что содержат *элементы, связанные по данным*. Их удобно использовать, потому что лишь немногие элементы в этих модулях связаны с внешней средой. Главное различие между ними — **информационно связный модуль** работает подобно сборочной линии; его обработчики действуют *в определенном порядке*; в **коммуникативно связном модуле** *порядок выполнения действий безразличен*. В данном примере не имеет значения, когда генерируется отчет (до, после или одновременно с вычислением средней зарплаты).

Процедурная связность (CC = 5)

При достижении процедурной связности попадаем в пограничную область между хорошей сопровождаемостью (для модулей с более высокими уровнями связности) и плохой сопровождаемостью (для модулей с более низкими уровнями связности). **Процедурно связный модуль** состоит из элементов, реализующих *независимые действия*, для которых задан порядок работы, то есть порядок передачи управления. Зависимости по данным между элементами нет. Например:

сделать зарядку,
принять душ,
позавтракать,
одеться,
отправится на учёбу.

Процедурная связность характерна для модуля, управляющие конструкции которого организованы так, как изображены на схеме программы. Такая структура модуля может возникнуть при расчленении длинной программы на части в соответствии с передачами управления, но без определения функций при выборе разделительных точек.

Пример 2.

Модуль Вычисление средних значений

используется Таблица-А. Таблица-В

вычислить среднее по Таблица-А

вычислить среднее по Таблица-В

вернуть среднееТабл-А. среднееТабл-В

Конец модуля

Этот модуль вычисляет средние значения для двух полностью несвязанных таблиц Таблица-А и Таблица-В, каждая из которых имеет по 300 элементов.

Теперь представим себе программиста, которому поручили реализовать данный модуль. Соблазнившись возможностью минимизации кода (использовать один цикл в интересах двух обработчиков, ведь они находятся внутри единого модуля!), программист пишет:

Модуль Вычисление средних значений

используется Таблица-А. Таблица-В
суммаТабл-А := 0
суммаТабл-В := 0
для i := 1 до 300
    суммаТабл-А := суммаТабл-А + Таблица-А(i)
    суммаТабл-В :- суммаТабл-В + Таблица-В(i)
конец для
среднееТабл-А := суммаТабл-А / 300
среднееТабл-В := суммаТабл-В / 300
вернуть среднееТабл-А, среднееТабл-В
Конец модуля

Для процедурной связности этот случай типичен — независимый (на уровне проблемы) код стал зависимым (на уровне реализации). Прошли годы, продукт сдали заказчику. И вдруг возникла задача сопровождения — модифицировать модуль под уменьшение размера таблицы В. Оцените, насколько удобно ее решать.

Временная связность (CC = 3)

Модуль, содержащий функционально несвязные части, но необходимые в один и то же момент обработки имеет временную связность или связность по классу. При связности по времени элементы-обработчики модуля привязаны к конкретному периоду времени (из жизни программной системы).

Классическим примером временной связности является модуль инициализации:

Модуль Инициализировать Систему

Таблица текущих записей : = пробел..пробел

Таблица количества записей := 0..0

Переключатель 1 : = выкл

Переключатель 2 := вкл

Конец модуля

Элементы данного модуля почти не связаны друг с другом (за исключением того, что должны выполняться в определенное время). Они все — часть программы запуска системы. Зато элементы более тесно взаимодействуют с другими модулями, что приводит к сложным внешним связям.

Модуль со связностью по времени испытывает те же трудности, что и процедурно связный модуль. Программист соблазняется возможностью совместного использования кода (действиями, которые связаны только по времени), модуль становится трудно использовать повторно.

Так, при желании инициализировать *Переключатель 1* в другое время вы столкнетесь с неудобствами. Чтобы не сбрасывать всю систему, придется или ввести флажки, указывающие инициализируемую часть, или написать другой код для работы с *Переключателем 1*. Оба решения ухудшают сопровождаемость.

Процедурно связные модули и **модули с временной связностью** очень похожи. Степень их непрозрачности изменяется от темного серого до светло-серого цвета, так как трудно объявить функцию такого модуля без перечисления ее внутренних деталей. *Различие* между ними подобно различию между информационной и коммуникативной связностью. *Порядок выполнения действий более важен* в **процедурно связных модулях**. Кроме того, процедурные модули имеют тенденцию к совместному использованию циклов и ветвлений, а модули с временной связностью чаще содержат более линейный код.

Логическая связность (CC = 1)

Если в модуле объединены операторы только по принципу их функционального подобия, а для настройки модуля применяется алгоритм переключения, то модуль имеет логическую связность. Элементы логически связного модуля принадлежат к действиям одной категории, и из этой категории клиент выбирает выполняемое действие. Рассмотрим следующий пример:

Модуль Пересылка сообщения

переслать по электронной почте

переслать по факсу

послать в телеконференцию

переслать по ftp-протоколу

Конец модуля

Как видим, логически связный модуль — мешок доступных действий. Действия вынуждены совместно использовать один и тот же интерфейс модуля. В строке вызова модуля значение каждого параметра зависит от используемого действия. При вызове отдельных действий некоторые параметры должны иметь значение пробела, нулевые значения и т. д. (хотя клиент все же должен использовать их и знать их типы).

Пример 2, альтернативы: поехать на автомобиле, на метро, на автобусе – являются средством достижения цели: добраться в како-то определенное место, из которых нужно выбрать одну.

Действия в логически связном модуле попадают в одну категорию, хотя имеют не только сходства, но и различия. К сожалению, это заставляет программиста «завязывать код действий в узел», ориентируясь на то, что действия совместно используют общие строки кода.

Логически связный модуль имеет:

- уродливый внешний вид с различными параметрами, обеспечивающими, например, четыре вида доступа;
- запутанную внутреннюю структуру со множеством переходов, похожую на волшебный лабиринт.

В итоге модуль становится сложным как для понимания, так и для сопровождения.

Связность по совпадению (CC = 0)

Модуль имеет связность по совпадению, если его операторы объединяются произвольным образом. Элементы связного по совпадению модуля вообще не имеют никаких отношений друг с другом:

Модуль Разные функции (какие-то параметры)

поздравить с Новым годом (...)

проверить исправность аппаратуры (...)

заполнить анкету героя (...)

измерить температуру (...)

вывести собаку на прогулку (...)

запастись продуктами (...)

приобрести «ягуар» (...)

поужинать,

посмотреть телевизор,

проверить электронную почту

Конец модуля

Связный по совпадению модуль похож на логически связный модуль. Его элементы-действия не связаны ни потоком данных, ни потоком управления. Но в логически связном модуле действия, по крайней мере, относятся к одной категории; в связном по совпадению модуле даже это не так. Словом, связные по совпадению модули имеют все недостатки логически связных модулей и даже усиливают их. Применение таких модулей вселяет ужас, поскольку один параметр используется для разных целей.

Чтобы клиент мог воспользоваться модулем Разные функции, этот модуль (подобно всем связным по совпадению модулям) должен быть «белым ящиком», чья реализация полностью видима. Такие модули делают системы менее понятными и труднее сопровождаемыми, чем системы без модульности вообще!

К счастью, **связность по совпадению** встречается редко. Среди ее причин можно назвать:

- бездумный перевод существующего монолитного кода в модули;
- необоснованные изменения модулей с плохой (обычно временной) связностью, приводящие к добавлению флажков.

Определение связности модуля

Приведем алгоритм определения уровня связности модуля.

1. Если модуль — единичная проблемно-ориентированная функция, то уровень связности — **функциональный**; конец алгоритма. В противном случае перейти к пункту 2.

2. Если *действия внутри модуля связаны*, то перейти к пункту 3. Если *действия внутри модуля никак не связаны*, то перейти к пункту 6.

3. Если *действия внутри модуля связаны данными*, то перейти к пункту 4. Если *действия внутри модуля связаны потоком управления*, перейти к пункту 5.

4. Если порядок действий внутри модуля важен, то уровень связности — **информационный**. В противном случае уровень связности — **коммуникативный**. Конец алгоритма.

5. Если *порядок действий внутри модуля важен*, то уровень связности — **процедурный**. В противном случае уровень связности — **временной**. Конец алгоритма.

6. Если *действия внутри модуля принадлежат к одной категории*, то уровень связности — **логический**. Если *действия внутри модуля не принадлежат к одной категории*, то уровень связности — **по совпадению**. Конец алгоритма.

Возможны более сложные случаи, когда с модулем ассоциируются несколько уровней связности. В этих случаях следует применять одно из двух правил:

- *правило параллельной цепи*. Если все действия модуля имеют несколько уровней связности, то модулю присваивают *самый сильный уровень связности*;
- *правило последовательной цепи*. Если действия в модуле имеют разные уровни связности, то модулю присваивают *самый слабый уровень связности*.

Например, модуль может содержать некоторые действия, которые связаны процедурно, а также другие действия, связные по совпадению. В этом случае применяют правило последовательной цепи и в целом модуль считают связным по совпадению.

В программных системах должны присутствовать модули, имеющие следующие три меры связности: *функциональная*, *последовательная* и *информационная*, так как другие типы связности являются крайне нежелательными и осложняют понимание и сопровождение системы.

Связность модулей. Связность - мера прочности соединения функциональных и информационных объектов внутри одного модуля. Если сцепление характеризует качество отделения модулей, то связность характеризует степень взаимосвязи элементов, реализуемых одним модулем. Размещение сильно связанных элементов в одном модуле уменьшает межмодульные связи и, соответственно, взаимовлияние модулей. В то же время помещение сильно связанных элементов в разные модули не только усиливает межмодульные связи, но и усложняет понимание их взаимодействия. Объединение слабо связанных элементов также уменьшает технологичность модулей, так как такими элементами сложнее мысленно манипулировать.

Различают следующие **виды связности** (в порядке убывания уровня):

- функциональную;
- последовательную;
- информационную (коммуникативную);
- процедурную;
- временную;
- логическую;
- случайную.

При функциональной связности все объекты модуля предназначены для выполнения одной функции (рисунок 8.2, а): операции, объединяемые для выполнения одной функции, или данные, связанные с одной функцией. Модуль, элементы которого связаны функционально, имеет четко определенную цель, при его вызове выполняется одна задача, например подпрограмма поиска минимального элемента массива. Такой модуль имеет максимальную связность, следствием которой являются его хорошие технологические качества: простота тестирования, модификации и сопровождения. Именно с этим связано одно из требований структурной декомпозиции «один модуль - одна между модулями - библиотеками ресурсов. Например, если при проектировании текстового редактора предполагается функция редактирования, то лучше организовать модуль - библиотеку функций редактирования, чем поместить часть функций в один модуль, а часть в другой.

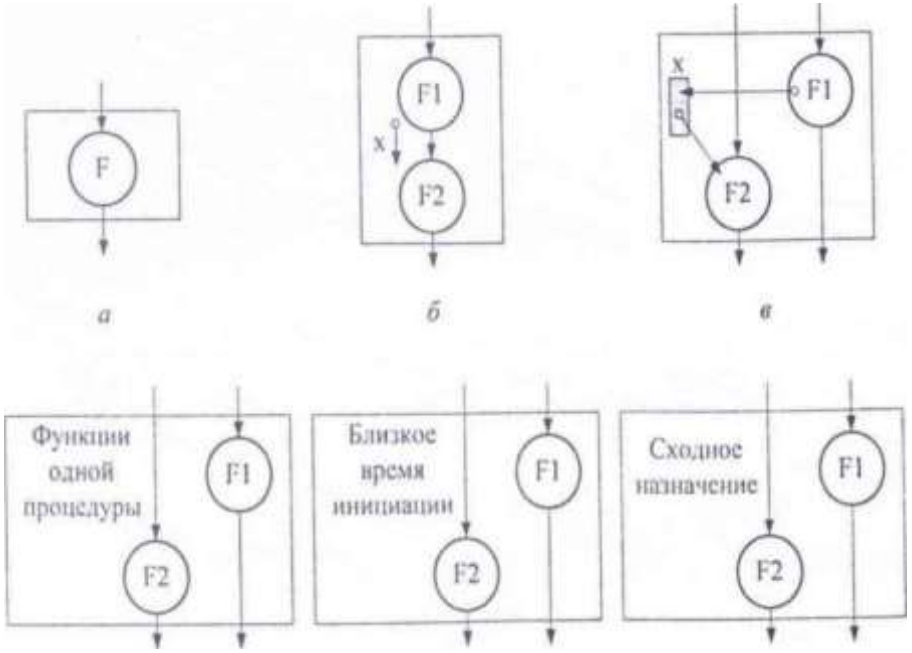


Рисунок 8.2 - Связность модулей:

а - функциональная, б - последовательная, в - информационная;

г - процедурная; б - временная; е - логическая

При последовательной связности функций выход одной функции служит исходными данными для другой функции (рисунок 8.2, б). Как правило, такой модуль имеет одну точку входа, т. е. реализует одну подпрограмму, выполняющую две функции. Считают, что данные, используемые последовательными функциями, также связаны последовательно. Модуль с последовательной связностью функций можно разбить на два или более модулей, как с последовательной, так и с

функциональной связностью. Такой модуль выполняет несколько функций, и, следовательно, его технологичность хуже: сложнее организовать тестирование, а при выполнении модификации мысленно приходится разделять функции модуля.

Информационно связанными считают функции, обрабатывающие одни и те желанные (рисунок 8.2, в). При использовании структурных языков программирования раздельное выполнение функций можно осуществить только, если каждая функция реализуется своей подпрограммой.

Несмотря на объединение нескольких функций, информационно связанный модуль имеет неплохие показатели технологичности. Это объясняется тем, что все функции, работающие с некоторыми данными, собраны в одно место, что позволяет при изменении формата данных корректировать только один модуль. Информационно связанными также считают данные, которые обрабатываются одной функцией.

Процедурно связаны функции или данные, которые являются частями одного процесса (рисунок 8.2, г). Обычно модули с процедурной связностью функций получают, если в модуле объединены функции альтернативных частей программы. При процедурной связности отдельные элементы модуля связаны крайне слабо, так как реализуемые ими действия связаны лишь общим процессом, следовательно, технологичность данного вида связи ниже, чем предыдущего.

Временная связность функций подразумевает, что эти функции выполняются параллельно или в течение некоторого периода времени (рисунок 8.2, д). Временная связность данных означает, что они используются в некотором временном интервале. Например, временную связность имеют функции, выполняемые при инициализации некоторого процесса. Отличительной особенностью временной связности является то, что действия, реализуемые такими функциями, обычно могут выполняться в любом порядке. Содержание модуля с временной связностью функций имеет тенденцию меняться: в

него могут включаться новые действия и/или исключаться старые. Большая вероятность модификации функции еще больше уменьшает показатели технологичности модулей данного вида по сравнению с предыдущим.

Логическая связь базируется на объединении данных или функций в одну логическую группу (рисунок 8.2, е). В качестве примера можно привести функции обработки текстовой информации или данные одного и того же типа. Модуль с логической связностью функций часто реализует альтернативные варианты одной операции, например сложение целых чисел и сложение вещественных чисел. Из такого модуля всегда будет вызываться одна какая-либо его часть, при этом вызывающий и вызываемый модули будут связаны по управлению. Понять логику работы модулей, содержащих логически связанные компоненты, как правило, сложнее, чем модулей, использующих временную связность, следовательно, их показатели технологичности еще ниже.

В том случае, если связь между элементами мала или отсутствует, считают, что они имеют случайную связность. Модуль, элементы которого связаны случайно, имеет самые низкие показатели технологичности, так как элементы, объединенные в нем, вообще не связаны.

В трех предпоследних случаях связь между несколькими подпрограммами в модуле обусловлена внешними причинами. А в последнем - вообще отсутствует. Это соответствующим образом проецируется на технологические характеристики модулей. В таблице 8.2 представлены характеристики различных видов связности по экспертным оценкам.

Анализ таблице 8.2 показывает, что на практике целесообразно использовать функциональную, последовательную и информационную связности.

Как правило, при хорошо продуманной декомпозиции модули верхних уровней иерархии имеют функциональную или последовательную связность функций и данных. Для модулей обслуживания данных характерна информационная связность функций. Данные таких модулей могут быть связаны по-разному. Так, модули, содержащие описание классов при объектно-ориентированном подходе, характеризуются информационной связностью методов и функциональной связностью данных. Получение в процессе декомпозиции модулей с другими видами связности, скорее всего, означает недостаточно продуманное проектирование. Исключением являются лишь библиотеки ресурсов.

Таблица 8.2

Вид связности	**Сцепление, балл**	**Наглядность (понятность)**	**Возможность изменения**	**Сопровождаемость**
Функциональная	10	Хорошая	Хорошая	Хорошая
Последовательная	9	Хорошая	Хорошая	Хорошая
Информационная	8	Средняя	Средняя	Средняя
Процедурная	5	Средняя	Средняя	Плохая
Временная	3	Средняя	Средняя	Плохая
Логическая	1	Плохая	Плохая	Плохая
Случайная	0	Плохая	Плохая	Плохая

Библиотеки ресурсов. Различают библиотеки ресурсов двух типов: библиотеки подпрограмм и библиотеки классов.

Библиотеки подпрограмм реализуют функции, близкие по назначению, например, библиотека графического вывода информации. Связность подпрограмм между собой в такой библиотеке - логическая, а связность самих подпрограмм - функциональная, так как каждая из них обычно реализует одну функцию.

Библиотеки классов реализуют близкие по назначению классы. Связность элементов класса - информационная, связность классов между собой может быть функциональной - для родственных или ассоциированных классов и логической - для остальных.

В качестве средства улучшения технологических характеристик библиотек ресурсов в настоящее время широко используют разделение тела модуля на интерфейсную часть и область реализации (секции Interface и Implementation - в Pascal, h и cpp-файлы в C++ и в Java).

Интерфейсная часть в данном случае содержит совокупность объявлений ресурсов (заголовков подпрограмм, имен переменных, типов, классов и т. п.), которые данная библиотека предоставляет другим модулям. Ресурсы, объявление которых в интерфейсной части отсутствует, извне не доступны. Область реализации содержит тела подпрограмм и, возможно, внутренние ресурсы (подпрограммы, переменные, типы), используемые этими подпрограммами. При такой организации любые изменения реализации библиотеки, не затрагивающие ее интерфейс, не требуют пересмотра модулей, связанных с библиотекой, что улучшает технологические характеристики модулей-библиотек. Кроме того, подобные библиотеки, как правило, хорошо отлажены и продуманы, так как часто используются разными программами.

Виды связности модуля

1.	Связность по совпадению (CC=0)	Части модуля вообще не имеют никаких отношений друг с другом.
2.	Логическая связность (CC=1)	Части модуля объединены по принципу функционального подобия, принадлежат к действиям одной категории.
3.	Временная связность (CC=3)	Части модуля не связаны, но необходимы в один и тот же период работы системы.
4.	Процедурная связность (CC=5)	Части модуля связаны порядком выполняемых ими действий, реализующих некоторый сценарий поведения.
5.	Коммуникативная связность (CC=7)	Части модуля связаны по данным – используют одни и те же данные.
6.	Информационная (последовательная) связность (CC=9)	Выходные данные одной части используются как входные данные в другой части модуля.
7.	Функциональная связность (CC=10)	Части модуля вместе реализуют одну функцию.

Сцепление модулей

Сцепление (Coupling) — мера взаимозависимости модулей поданным. Это мера относительной независимости модулей, которая определяет их читабельность и сохранность.

Модули являются полностью независимыми, если каждый из них не содержит о другом никакой информации. Чем больше информации в них используется, тем менее они независимы и тем более сцеплены. Чем очевиднее взаимодействие двух связных друг с другом модулей, тем проще определить необходимость корректировки одного модуля, зависящего от другого модуля.

Сцепление — внешняя характеристика модуля, которую желательно уменьшать.

№ п.п.	Сцепление	Степень сцепления
1	Независимые	0
2	Сцепление по данным	**1**
3	По образу	**3**
4	По общей области	**4**
5	По управлению	**5**
6	По внешним ссылкам	**7**
7	По кодам	**9** (сильное сцепление)

Количественно сцепление измеряется степенью сцепления (СЦ). Выделяют 7 типов сцепления.

1. **Независимое сцепление** (СЦ=0). возможно только в том случае, если модули не вызывают друг друга и не обрабатывают одну и ту же информацию.
2. **Сцепление по данным** (СЦ=1). Модули сцеплены по данным, если они имеют общие простые элементы данных, которые передаются от одного модуля к другому, как параметры. Вызывающий модуль знает только имя вызываемого модуля, а также типы и значения переменных, передаваемых как параметры. в этом случае изменение в структуре данных не влияет на другой модуль. Модули со сцеплением по данным не имеет общей области данных или неявных параметров. Модуль А вызывает модуль В.

   Все входные и выходные параметры вызываемого модуля — простые элементы данных (рисунок 8.3).

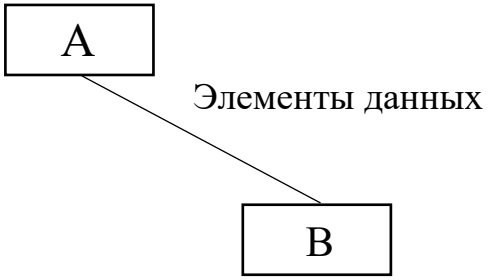


Рисунок 8. 3 - Сцепление поданным

3. **Сцепление по образцу** (СЦ=3). Модули сцеплены по образцу, если в качестве параметров используются структуры данных. Недостаток такого сцепления в том, что оба модуля должны знать о внутренней структуре данных, и если модифицируется структура в одном модуле, то необходимо модифицировать и в другом, значит, увеличивается вероятность появления ошибок при сопровождении программы. В качестве параметров используются структуры данных (рисунок 8.4).

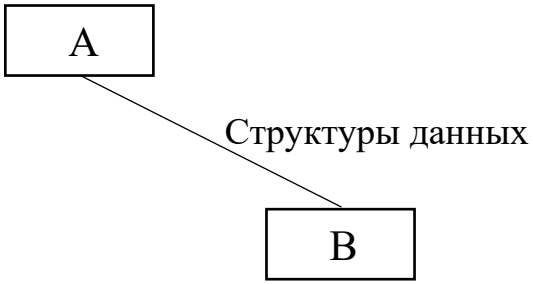


Рисунок 8.4 - Сцепление по образцу

4. **Сцепление по управлению** (СЦ=4). Модули сцеплены по управлению, если какой-то из них управляет решениями внутри другого с помощью передачи флагов, переключателей, то есть один из модулей знает о внутренней структуре другого. Например, модуль имеет логическую связь и при его вызове используется переключатель, то оба модуля сцеплены по умолчанию. Модуль А явно управляет функционированием модуля В (с помощью флагов или переключателей), посылая ему управляющие данные (рисунок 8.5).

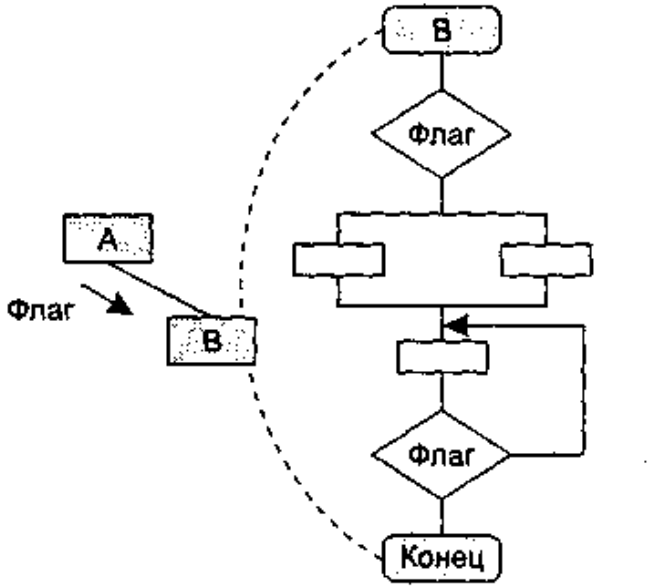


Рисунок 8.5 - Сцепление по управлению

5. Сцепление по внешним ссылкам (СЦ=5). Модули сцеплены по внешним ссылкам, если у 1 модуля есть доступ к данным другого модуля через внешнюю точку входа. Сцепление этого типа возникает, когда внутренние структуры оперируют с глобальными переменными, с которыми оперирует второй модуль. Модули А и В ссылаются на один и тот же глобальный элемент данных.

6. **Сцепление по общей области** (СЦ=7). Модули сцеплены по общей области, если они разделяют одну и ту де глобальную структуру данных. В этом случае возможность появления ошибок при модификации намного больше. Модули разделяют одну и ту же глобальную структуру данных (рисунок 8.6).

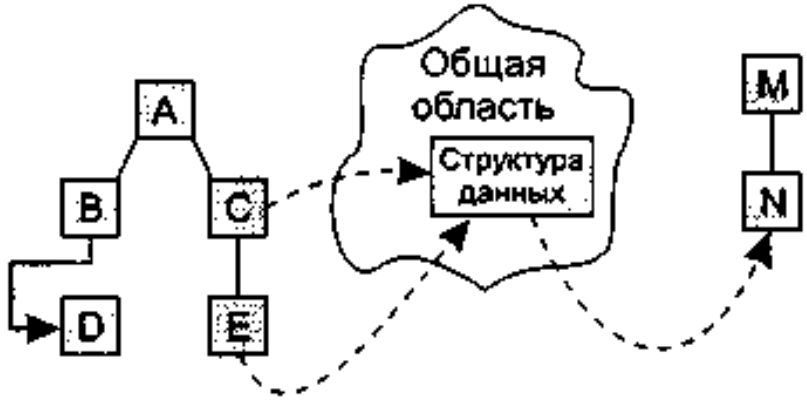


Рисунок 8.6 - Сцепление по общей области и содержанию

7. **Сцепление по содержанию** (СЦ=9). Модули сцеплены по кодам, если коды их команд перемешаны друг с другом. Например, для одного из модулей доступны внутренние области другого модуля, то есть модули используют один участок памяти. Один модуль прямо ссылается на содержание другого модуля (не через его точку входа). Например, коды их команд перемежаются друг с другом (рисунок 8.6).

На рисунке 8.6 видим, что модули B и D сцеплены по содержанию, а модули C, E и N сцеплены по общей области.

Если модули косвенно обращаются друг к другу, между ними также существует сцепление.

Вывод: сцепление модулей зависит от спроектированной структуры данных и способов взаимодействия между модулями. Необходимо использовать простые параметры и не применять глобальных данных.

Виды сцепления модулей

1.	Сцепление по данным (СЦ=1)	Модуль А вызывает модуль В. Все входные и выходные параметры вызываемого модуля – простые элементы данных.
2.	Сцепление по образцу (СЦ=3)	В качестве входных параметров используются структуры данных.
3.	Сцепление по управлению (СЦ=4)	Модуль А явно управляет функционированием модуля В, посылая ему управляющие данные.
4.	Сцепление по внешним ссылкам (СЦ=5)	Модули А и В ссылаются на один и тот же глобальный элемент данных.
5.	Сцепление по общей области (СЦ=7)	Модули разделяют одну и ту же глобальную структуру данных.
6.	Сцепление по содержанию (СЦ=9)	Один модуль прямо ссылается на содержание другого модуля. Например, коды их команд перемежаются друг с другом.

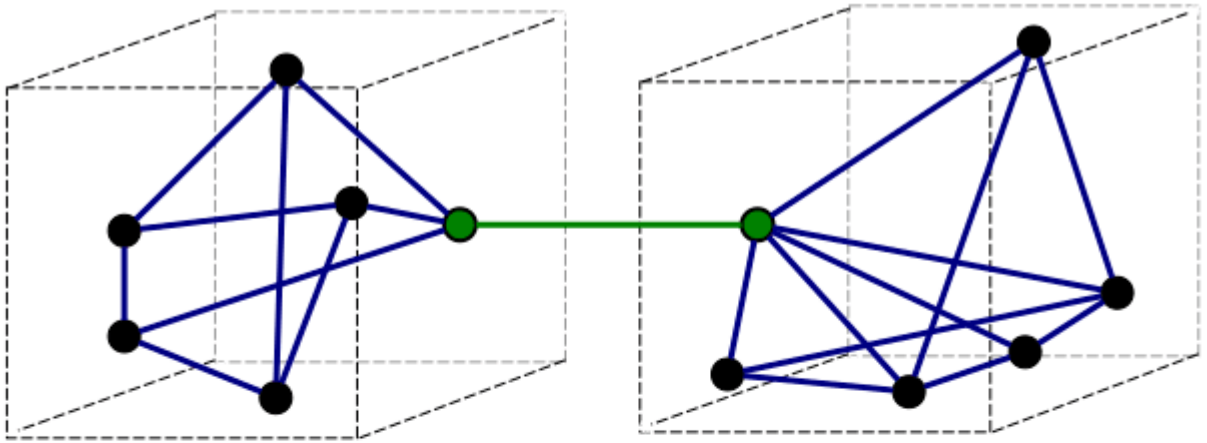
В таблице 8.3 приведены характеристики различных типов сцепления по экспертным оценкам. Допустимыми считают первые три типа сцепления, так как использование остальных приводит к резкому ухудшению технологичности программ.

Таблица 8.3

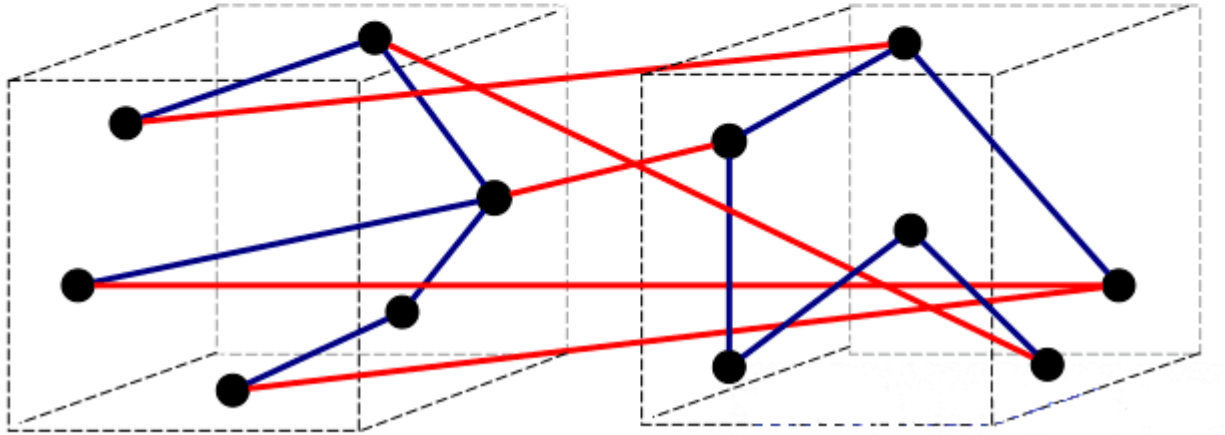
Тип сцепления	Сцепление, балл	Устойчивость к ошибкам других модулей	Наглядность (понятность)	Возможность изменения	Вероятность повторного использования
По данным	1	Хорошая *	Хорошая	Хорошая	Большая
По образцу	3	Средняя	Хорошая *	Средняя	Средняя
По управлению	4	Средняя	Плохая	Плохая	Малая
По общей области	6	Плохая	Плохая	Средняя	Малая
По содержимому	10	Плохая	Плохая	Плохая	Малая

Как правило, модули сцепляются между собой несколькими способами. Учитывая это, качество программного обеспечения принято определять по типу сцепления с худшими характеристиками. Так, если использовано сцепление по данным и сцепление по управлению, то определяющим считают сцепление по управлению.

В некоторых случаях сцепление модулей можно уменьшить, удалив необязательные связи и структурировав необходимые связи. Примером может служить объектно-ориентированное программирование, в котором вместо большого количества параметров метод неявно получает адрес области (структуры), в которой расположены поля объекта, и явно-дополнительные параметры. В результате модули оказываются сцепленными по образцу.

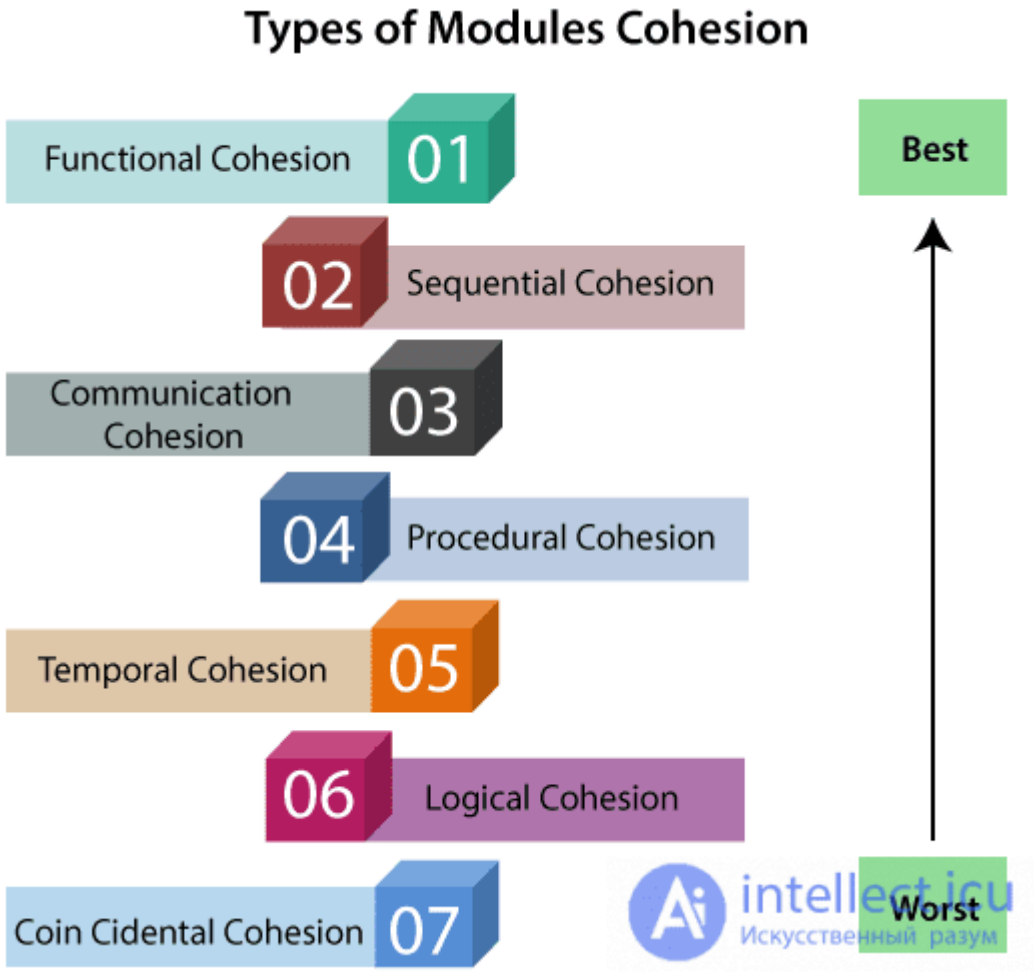


a) Слабое сцепление, сильная связность



b) Сильное сцепление, слабая связность

Типы связанности



Виды связанности в порядке от худшего к лучшему:

Случайная связанность (худшая)

Случайная связанность возникает тогда, когда части модуля добавляются в него произвольным образом; единственным их связывает то, что они входят в один модуль (например класс " Utilities»).

Логическая связанность

Части модуля группируются вместе, потому что они по логике выполняют одну функцию, даже если они разные по природе. Например, группировки всех подпрограмм обработки ввода с клавиатуры и мыши.

Временная связанность

Части модуля группируются в зависимости от того, в какой момент времени возникает необходимость их применения. Например, функции вызываемых после обнаружения ошибки, которые закрывают открытые файлы, записывают ошибку в журнал, и сообщают пользователя.

Процедурная связанность

Части модуля группируются, потому что они все придерживаются определенной последовательности действий. Например, функции, проверяющих разрешения для файла, и после этого его открывают.

Коммуникационная связанность

Части модуля группируются, потому что они работают с одними и теми же данными.

Последовательная связанность

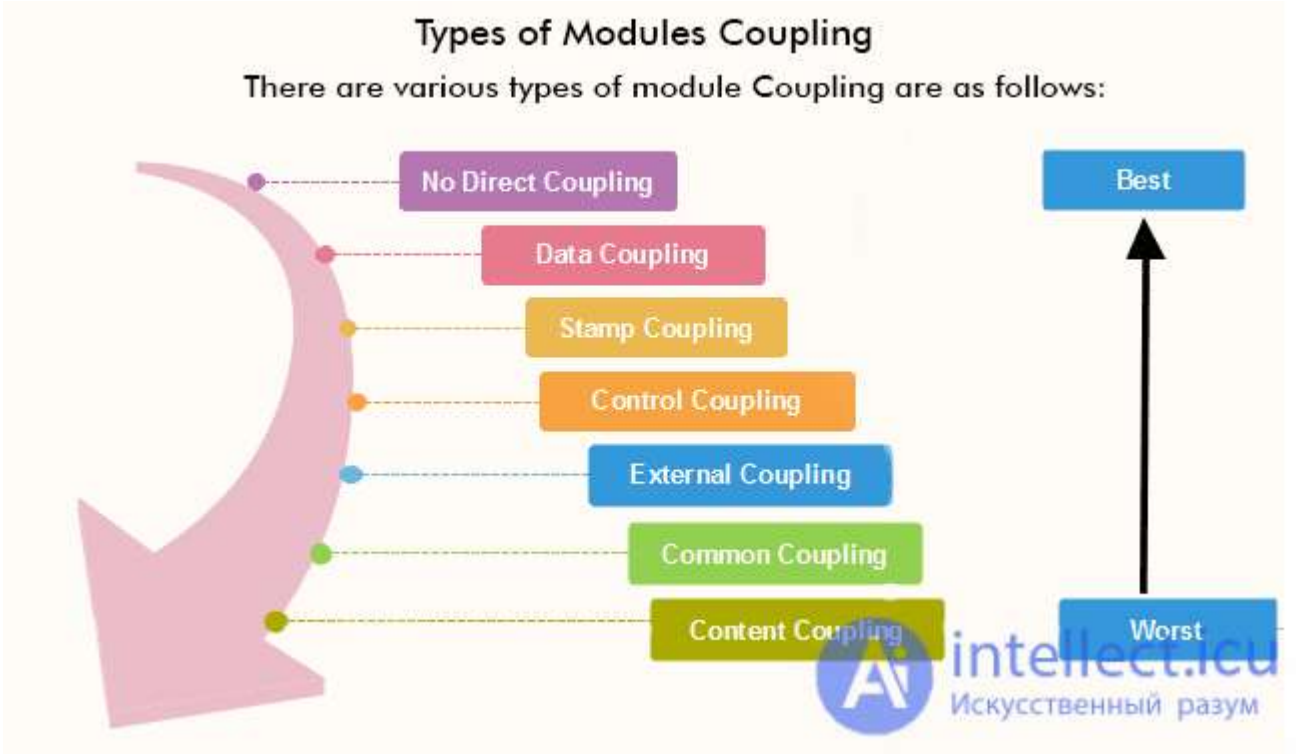
Части модуля группируются, потому что вывод одной части передается на вход другой, как в конвейере. Например, функции, читающих данные из файла, обрабатывают их и пишут назад.

Функциональная связанность (лучшая)

Части модуля группируются, потому что они вместе работают над одной четко обозначенной задачей для этого модуля. Например, лексический анализ текста или XML.

Исследования многих людей, таких как Ларри Константин, Edward Yourdon, и Steve McConnell показывают, что два первых вида связанности плохие, коммуникационная и последовательная связанности достаточно хорошие и функциональная - лучшая, хотя не всегда достижима. Бывают случаи, когда коммуникационная связанность — это лучшее, что может быть достигнуто при данных обстоятельствах

Виды сцепления



Связанность содержимого (content coupling)

- Один модуль изменяет или полагается на внутренние особенности другого модуля (например, использует локальные данные другого модуля).
- Изменение работы второго модуля приведет к переписыванию первого.

Связанность через общее (common coupling)

- Два модуля работают с общими данными (например, глобальной переменной).
- Изменение разделяемого ресурса приведет к изменению всех работающих с ним модулей.

Связанность через внешнее (external coupling)

- Два модуля используют навязанный извне формат данных, протокол связи и т. д.
- Обычно возникает из-за внешних сущностей (инструментов, устройств и т. д.).

Связанность по управлению (control coupling)

- Один модуль управляет поведением другого.
- Присутствует передача информации о том, что и как делать.

Связанность по структурированным данным (data-structured coupling, stamp couplig)

- Модули используют одну и ту же структуру, но каждый использует только ее части.
- Изменение структуры может привести к изменению модуля, который измененную часть даже не использует.

Связанность через данных (data coupling)

- Модули совместно используют данные, например, через параметры.
- Элементарные фрагменты маленькие и только они используются модулями совместно.

Связанность по сообщениям (message coupling)

- Модули общаются только через передачу параметров или сообщений.
- Состояние децентрализовано.

Отсутствие связанности (no coupling)

- Модули вообще никак не взаимодействуют.

Типы зацепления

Связность и зацепление модулей

Типы зацепления, согласно стандарту ISO/IEC/IEEE 24765-2010, включают:

- зацепление по общей области (англ. *common-environment coupling*, *common coupling*);
- зацепление по содержимому (англ. *content coupling*);
- зацепление по управлению (англ. *control coupling*);
- зацепление по данным (англ. *data coupling*, *input-output coupling*);
- смешанное зацепление (англ. *hybrid coupling*);
- патологическое зацепление (англ. *pathological coupling*)[2].

Зацепление по общей области

Тип зацепления, при котором два программных модуля совместно используют общую область данных.

Зацепление по содержимому

Тип зацепления, при котором некоторые или все программные модули включены в некоторый модуль как составные части.

Зацепление по управлению

Тип зацепления, при котором один программный модуль обменивается данными с другим модулем с явной целью повлиять на его последующее выполнение.

Зацепление по данным

Тип зацепления, при котором выходные данные одного программного модуля служат входными данными другого модуля.

Смешанное зацепление

Тип зацепления, при котором различные подмножества значений некоторого элемента данных используются в нескольких программных модулях для разных и несвязанных целей.

Патологическое зацепление

Тип зацепления, при котором один программный модуль зависит от деталей внутренней реализации другого модуля или влияет на них.

Мера связности	Сцепление	Модифицируемость	Понятность	Сопровождаемость
Функциональная	хорошее	хорошая	хорошая	хорошая
Последовательная	хорошее	хорошая	близкая к хорошей	хорошая
Информационная	среднее	средняя	средняя	средняя
Процедурная	приемлемое	приемлемая	приемлемая	плохая
Временная	плохое	плохая	средняя	плохая
Логическая	плохое	плохая	плохая	плохая
Случайная	плохое	плохая	плохая	плохая

Практика показала, что чем выше степень независимости модулей, тем:

- легче разобраться в отдельном модуле и всей программе и, соответственно, тестировать, отлаживать и модифицировать ее;
- меньше вероятность появления новых ошибок при исправлении старых или внесении изменений в программу, т. е. вероятность появления «волнового» эффекта;
- проще организовать разработку программного обеспечения группой программистов и легче его сопровождать.

Таким образом, уменьшение зависимости модулей улучшает технологичность проекта.

Сложность программной системы

В простейшем случае **сложность системы** определяется как **сумма мер сложности ее модулей**.

Сложность модуля может вычисляться различными способами:

1. М. Холстед (1977) предложил меру **длины N модуля**:

$$N \approx n_1 log_2 (n_1) + n_2 log_2 (n_2),$$

где n_1 — число различных операторов, n_2 — число различных операндов.

В качестве второй метрики М. Холстед рассматривал **объем *V* модуля** (количество символов для записи всех операторов и операндов текста программы):

$$V = N \log_2 (n_1 + n_2).$$

2. Вместе с тем известно, что любая сложная система состоит из элементов и системы связей между элементами и что игнорировать внутрисистемные связи неразумно.

Том МакКейб (1976) при оценке сложности ПС предложил исходить из топологии внутренних связей. Для этой цели он разработал **метрику цикломатической сложности**:

$$V(G) = E - N + 2,$$

где E — количество дуг, N — количество вершин в управляющем графе ПС.

Дальнейшее уточнение оценок сложности потребовало, чтобы каждый модуль мог представляться как локальная структура, состоящая из элементов и связей между ними.

При комплексной оценке сложности ПС необходимо рассматривать:

– меру сложности модулей,

– меру сложности внешних связей (между модулями)

– меру сложности внутренних связей (внутри модулей).

Традиционно со внешними связями сопоставляют характеристику «сцепление», а с внутренними связями — характеристику «связность».

Комплексная оценка сложности ПС будет рассмотрена ниже.

Характеристики иерархической структуры программной системы

Иерархическая структура программной системы — основной результат предварительного проектирования. Она определяет состав модулей ПС и управляющие отношения между модулями. В этой структуре модуль более высокого уровня (начальник) управляет модулем нижнего уровня (подчиненным).

Иерархическая структура не отражает процедурные особенности программной системы, то есть последовательность операций, их повторение, ветвления и т. д. Рассмотрим основные характеристики иерархической структуры, представленной на рисунок 8.7.

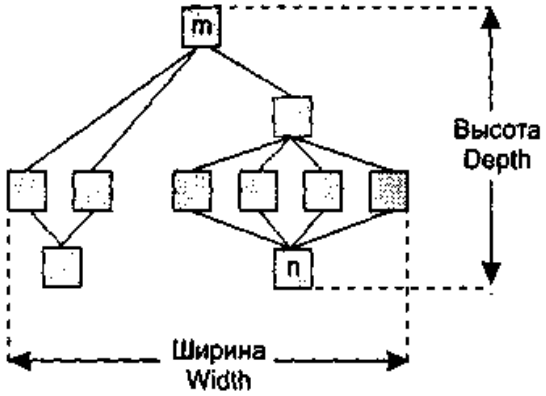


Рисунок 8.7 - Иерархическая структура программной системы

Первичными характеристиками являются *количество вершин (модулей)* и *количество ребер (связей между модулями)*. К ним добавляются две **глобальные характеристики** — высота и ширина:

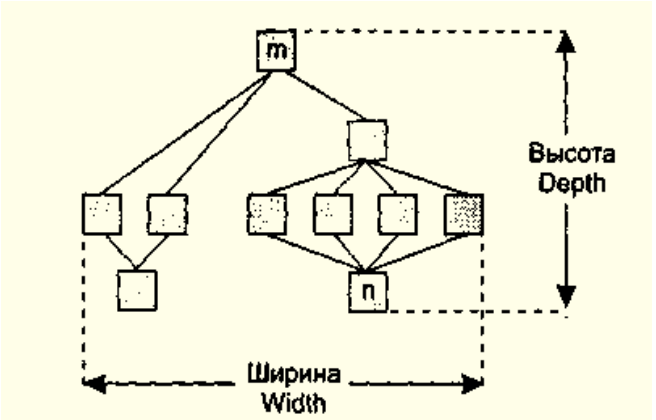
- **высота** — количество уровней управления;
- **ширина** — максимальное из количеств модулей, размещенных на уровнях управления.

В данном примере высота = 4, ширина = 6.

Локальными характеристиками модулей структуры являются:

– коэффициент объединения по входу;

– коэффициент разветвления по выходу.



Коэффициент объединения по входу ***Fan_in(i)*** — это количество модулей, которые прямо управляют ***i***-м модулем.

В примере для модуля **n**: Fan_in(***n***) = 4.

Коэффициент разветвления по выходу ***Fan_out(i)*** — это количество модулей, которыми прямо управляет ***i***-й модуль.

В примере для модуля **m**: Fan_out(***m***) = 3.

Возникает вопрос: как оценить *качество структуры*? Из практики проектирования известно, что *лучшее решение обеспечивается иерархической структурой в виде дерева*.

Степень отличия реальной проектной структуры от дерева характеризуется **невязкой структуры**.

Определение невязки структуры программной системы

Полный граф (complete graph) с ***n*** вершинами имеет количество ребер:

$$e_c = n\,(n - 1) / 2,$$

а дерево (tree) с таким же количеством вершин — существенно меньшее количество ребер:

$$e_t = n - 1.$$

Тогда формулу невязки можно построить, сравнивая количество ребер полного графа, реального графа и дерева.

Для проектной структуры с ***n*** вершинами и ***e*** ребрами **невязка** определяется по выражению:

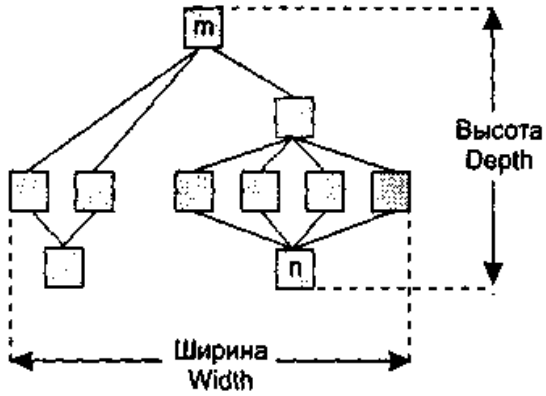
$$Nev = \frac{e - e_t}{e_c - e_t} = \frac{(e - n + 1) \times 2}{n \times (n-1) - 2 \times (n-1)} = \frac{2 \times (e - n + 1)}{(n-1) \times (n-2)}$$

Значение невязки лежит в диапазоне от 0 до 1. Если Nev = 0, то проектная структура является деревом, если Nev = 1, то проектная структура — полный граф.

Невязка дает грубую оценку структуры. Для увеличения точности оценки следует применить характеристики связности и сцепления.

Хорошая структура должна иметь низкое сцепление и высокую связность.

Л. Констентайн и Э. Йордан (1979) предложили оценивать структуру с помощью коэффициентов *локальных характеристик модулей*: **Fan_in(*i*)** и **Fan_out(*i*)** модулей.



Большое значение Fan_in(*i*) — свидетельство высокого сцепления, так как является мерой зависимости модуля.

Большое значение Fan_out(*i*) говорит о высокой сложности вызывающего модуля. Причиной является то, что для координации подчиненных модулей требуется сложная логика управления.

Основной недостаток коэффициентов Fan_in(*i*) и Fan_out(*i*) состоит в игнорировании веса связи. Здесь рассматриваются только управляющие потоки (вызовы модулей). В то же время информационные потоки, нагружающие ребра структуры, могут существенно изменяться, поэтому нужна мера, которая учитывает не только количество ребер, но и количество информации, проходящей через них.

С. Генри и Д. Кафура (1981) ввели информационные коэффициенты **ifan_in(*i*)** и **ifan_out(*j*)**. Они учитывают количество элементов и структур данных, из которых *i*-й модуль берет информацию и которые обновляются *j*-м модулем соответственно.

Информационные коэффициенты суммируются со структурными коэффициентами **sfan_in(*i*)** и **sfan_out(*j*)**, которые учитывают только вызовы модулей.

В результате формируются полные значения коэффициентов:

$$\text{Fan\_in}\ (i) = \text{sfan\_in}\ (i) + \text{ifan\_in}\ (i),$$

$$\text{Fan\_out}\ (j) = \text{sfan\_out}\ (j) + \text{ifan\_out}\ (j).$$

На основе полных коэффициентов модулей вычисляется **метрика общей сложности структуры**:

$$S = \sum_{i=1}^{n} \text{length}(i) \times (\text{Fan\_in}(i) + \text{Fan\_out}(i))^2,$$

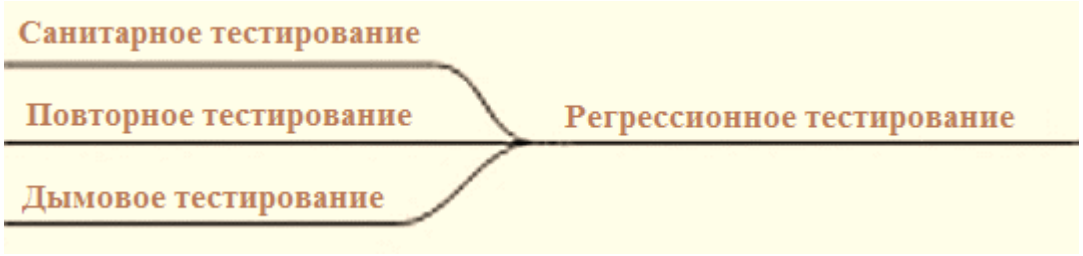
где length(*i*) — оценка размера *i*-го модуля (в виде LOC- или FP-оценки).

ЛЕКЦИЯ 11

ТЕМА 9: Регрессионное тестирование. Тестирование правильности. Альфа-тестирование. Бета-тестирование. Системное тестирование. Тестирование восстановления. Тестирование безопасности

Регрессионное тестирование

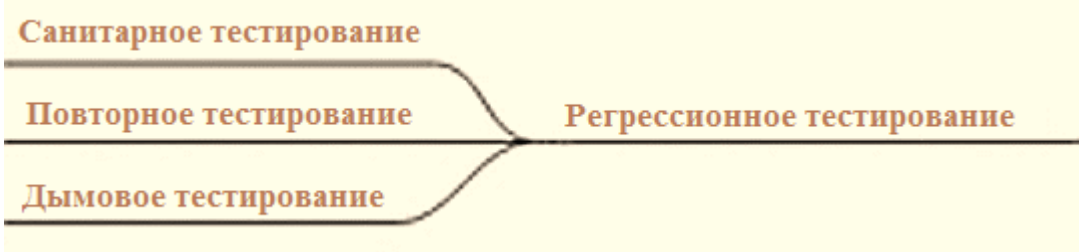
Регрессионное тестирование - собирательное название для всех видов тестирования программного обеспечения, направленных на обнаружение ошибок в уже протестированных участках исходного кода. Обычно используемые методы регрессионного тестирования включают повторные прогоны предыдущих тестов, а также проверки, не попали ли регрессионные ошибки в очередную версию в результате слияния кода.



Санитарное тестирование — это узконаправленное тестирование достаточное для доказательства того, что конкретная функция работает согласно заявленным в спецификации требованиям. Используется для определения работоспособности определенной части приложения после изменений, произведенных в ней или окружающей среде. Обычно выполняется вручную.

Повторное тестирование – это тестирование, во время которого исполняются тестовые сценарии, выявившие ошибки во время последнего запуска, для подтверждения успешности исправления этих ошибок.

Дымовое тестирование рассматривается как короткий цикл тестов, выполняемый для подтверждения того, что после сборки кода (нового или исправленного) устанавливаемое приложение, стартует и выполняет основные функции.



Дымовые (Smoke)	Санитарные (Sanity)	Регрессионные (Regression)	Повторные (Re-test)
Исполняются с целью проверить что критически важные функциональные части системы работают как положено	Нацелено на установление факта того, что определённые части системы всё так же работают как положено после минорных изменений или исправлений багов	Подтверждают, что свежие изменения в коде или приложении в целом не оказали негативного влияния на существующую функциональность/набор функций	Перепроверяет и подтверждает факт того, что ранее заваленные тест-кейсы проходят после того, как дефекты исправлены
Цель — проверить «стабильность» системы в целом, чтобы дать зелёный свет проведению более тщательного тестирования	Целью является проверить общее состояние системы в деталях, чтобы приступить к более тщательному тестированию	Цель — убедиться, что свежие изменения в коде не оказали побочных эффектов на устоявшуюся работающую функциональность	Проверяет что дефект исправлен
Перепроверка дефектов не является целью Smoke	Перепроверка дефектов не является целью Sanity	Перепроверка дефектов не является целью Regression	Факт того, что дефект исправлен подтверждает Re-Test
выполняется перед регрессионным	выполняется перед регрессионным и после smoke-тестов	Проводится на основании требований проекта и доступности ресурсов (закрывается автотестами), «регресс» может проводиться в параллели с ре-тестами	—выполняется перед sanity-тестированием — Так же, приоритет выше регрессионных проверок, поэтому должно выполняться перед ними
Может выполняться автоматизировано или вручную	Чаще выполняется вручную	Желательна автоматизация, т. к. ручное может быть крайне затратным по ресурсам или времени	Не поддаётся автоматизации
Является подмножеством регрессионного тестирования	Подмножество приёмочного тестирования	Выполняется при любой модификации или изменениях в существующем проекте	проводится на исправленной сборке с использованием тех же данных, на том же окружении, но с различным набором входных данных
Тест-кейсы часть регрессионных тест-кейсов, но покрывающие крайне критичную функциональность	может выполняться без тест-кейсов, но знание тестируемой системы обязательно	могут быть получены из функциональных требований или спецификаций, пользовательских мануалов, и проводятся вне зависимости от того, что исправили разработчики	Используется тот же самый тест-кейс, который выявил дефект

Регрессионное тестирование — это тип тестирования программного обеспечения, выполняемый для проверки того, не повлияло ли изменение кода на текущие функции и функции приложения.

Регрессионное тестирование включает:

- **Регрессия новых багов** (new bug-fix) — проверка исправления вновь найденного дефекта, то есть попытка доказать, что исправленная ошибка на самом деле не исправлена.
- **Регрессия старых багов** (old bug-fix) — проверка, что исправленный ранее и верифицированный дефект не воспроизводится в системе снова.
- **Регрессия побочного эффекта** (side-effect) — проверка того, что не нарушилась работоспособность работающей ранее функциональности, если её код мог быть затронут при исправлении некоторых дефектов в другой функциональности.

Из опыта разработки ПО известно, что повторное появление одних и тех же ошибок — случай достаточно частый. Иногда это происходит из-за слабой техники управления версиями или по причине человеческой ошибки при работе с системой управления версиями. Но настолько же часто решение проблемы бывает «недолго живущим»: после следующего изменения в программе решение перестаёт работать. И наконец, при переписывании какой-либо части кода часто всплывают те же ошибки, что были в предыдущей реализации.

Поэтому считается хорошей практикой при исправлении ошибки создать тест на неё и регулярно прогонять его при последующих изменениях программы. Хотя регрессионное тестирование может быть выполнено и вручную, но чаще всего это делается с помощью специализированных программ, позволяющих выполнять все регрессионные тесты автоматически. В некоторых проектах даже используются инструменты для автоматического прогона регрессионных тестов через заданный интервал времени. Обычно это выполняется после каждой удачной компиляции (в небольших проектах) либо каждую ночь или каждую неделю.

Регрессионное тестирование является неотъемлемой частью экстремального программирования. В этой методологии проектная документация заменяется на расширяемое, повторяемое и автоматизированное тестирование всего программного пакета на каждой стадии процесса разработки программного обеспечения.

Необходимость регрессионного тестирования

Как правило, регрессионное тестирование применяется в следующих случаях:

- Новое требование добавлено к существующей функции.
- Новая функция или функциональность добавлена.
- Кодовая база исправлена для устранения дефектов.
- Исходный код оптимизирован для повышения производительности.
- Исправление багов.
- Изменения в конфигурации.

Благодаря быстрому процессу регрессионного тестирования продуктовые группы могут получать более информативную обратную связь и мгновенно реагировать. Регрессионное тестирование обнаруживает новые ошибки на ранних этапах цикла развертывания, поэтому предприятиям не нужно вкладывать средства в затраты и техническое обслуживание для устранения накопленных дефектов. Иногда легкая модификация может вызвать эффект домино на ключевые функции продукта. Именно поэтому любые изменения, даже самые маленькие, должны быть проконтролированы и протестированы.

Функциональные тесты только проверяют поведение новых функций и возможностей, но не учитывают, насколько они совместимы с существующими. Поэтому без регрессионного тестирования более сложно и трудоемко исследовать основную причину и архитектуру продукта.

Другими словами, если проект подвергается частым изменениям, регрессионное тестирование будет фильтром, обеспечивающим качество по мере улучшения продукта.

На рисунке 9.1 показан пример регрессионного тестирования. V1, V2, V3, V4 обозначают версии продукта. Как видим, с каждой новой версией добавляется также и новая функциональность – F1, F2, F3. Регрессионное тестирование позволяет удостовериться, что добавление новых функциональности не сломало ни одну из предыдущих, и не создало каких-либо новых багов.

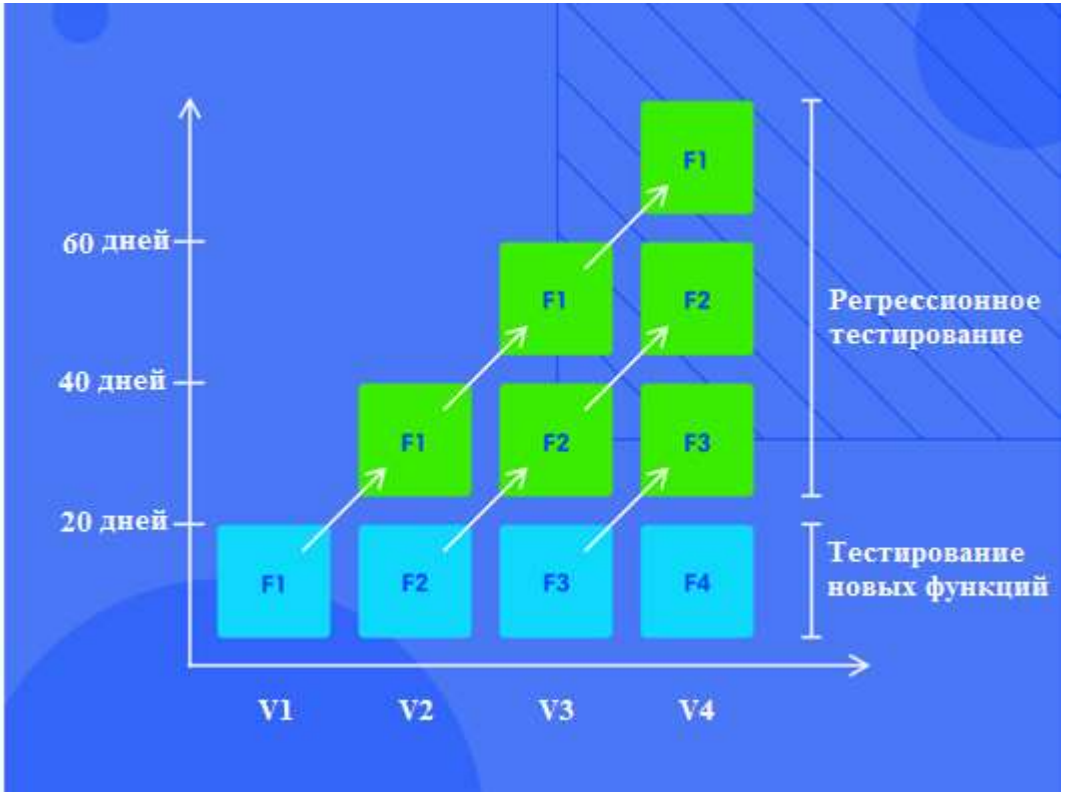


Рисунок 9.1 – Регрессионное тестирование в производстве

Процесс тестирования

Конкретная реализация регрессионного тестирования отличается в разных организациях, однако можно выделить несколько основных шагов (рисунок 9.2):

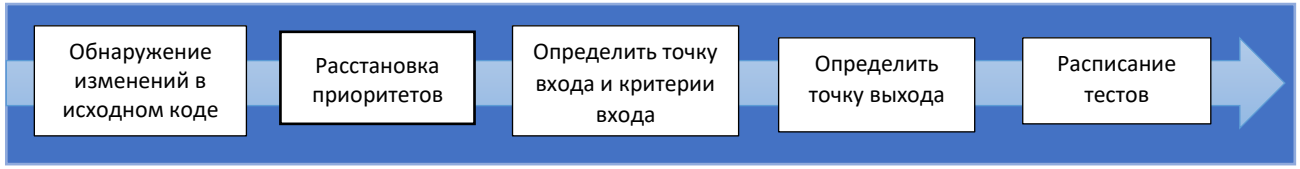


Рисунок 9.2 – Шаги регрессионного тестирования

Шаг 1. *Обнаружение изменений в исходном коде*.

Первоначально необходимо определить какая часть кода была изменена/оптимизирована.

Далее необходимо найти компоненты/модули, которые были затронуты, а также их влияние на существующие функции.

Шаг 2. *Расстановка приоритетов*

Затем необходимо указать приоритет этих изменений и требований к продукту, чтобы упростить процесс тестирования с помощью соответствующих тестовых примеров и инструментов тестирования.

Шаг 3. *Определить точку входа и критерии входа*

Перед выполнением регрессионного теста убедитесь, что приложение соответствует установленным критериям.

Шаг 4. *Определить точку выхода*

Определите точку выхода или конечную точку для требуемого права или минимальных условий, установленных в шаге три.

Шаг 5. *Расписание тестов*. Необходимо определить дату и время для проведения соответствующего теста.

Техники регрессионного тестирования

Существует 3 основных **методики проведения регрессионного тестирования**:

- **Полное тестирование**

В этом методе регрессионное тестирование применяется ко всем существующим комплектам тестов. Хотя это самый безопасный способ обеспечить обнаружение и устранение всех ошибок, этот метод требует значительного времени и ресурсов.

Подход полной регрессии лучше подходит в определенных контекстах - например, когда приложение настраивается на новую платформу или язык или когда операционная система получает серьезное обновление.

Методики проведения регрессионного тестирования

- **Выбор регрессионного теста**

Зачем выполнять весь комплектов тестов, если можно найти модули, которые были затронуты изменениями и протестировать лишь их. Это позволит уменьшить время и усилия вложенные в регрессионное тестирование

- **Приоритезация тестов**

Можно выбрать приоритетные тестовые случаи, которые должны быть включены и выполнены первыми в процессе регрессионного тестирования. Приоритет должен расставляться по следующим критериям: частота отказов, влияние на бизнес и часто используемые функции.

Эффективные регрессионные тесты должны выбираться из следующих тест кейсов:

- Тест кейсы с частыми дефектами.
- Компоненты, которые более заметны для пользователей.
- Тест кейсы, которые проверяют основную функциональность продукта.
- Тест кейсы компонентов, которые претерпели недавние изменения.
- Все интеграционные тесты.
- Все сложные тестовые случаи.
- Контрольные примеры граничных значений.
- Образец успешных тестовых случаев.
- Образец ошибочных тестовых случаев.

Инструменты регрессионного тестирования

Если ПО подвергается частым изменениям, затраты на регрессионное тестирование будут расти. В таком случае, ручное тестирование не самым лучшим выбором, так как увеличит временные и денежные затраты на выполнение тестирование. Автоматизация регрессионных тестов будет лучшим выбором, нежели ручное тестирование.

Степень автоматизации зависит от количества тест кейсов, которые можно повторно использовать в последующих циклах тестирования.

Следующие **инструменты** являются лучшим выбором для автоматизации данного типа тестирования:

- **Ranorex Studio**

Представляет собой многофункциональный фреймворк для тестирования настольных, веб и мобильных приложений, с встроенным Selenium WebDriver. Также включает в себя полноценную IDE, и инструменты для “codeless” тестирования. Является мощным инструментом для GUI – тестирования

- **Selenium**

Selenium - инструмент для автоматизации действий веб-браузера. В большинстве случаев используется для тестирования Web-приложений, но этим не ограничивается. В частности, он может быть использован для решения рутинных задач администрирования сайта или регулярного получения данных из различных источников (сайтов).

Инструменты регрессионного тестирования

- **Rational Functional Tester (RFT)**

Rational Functional Tester — это инструмент автоматизации тестирования программного обеспечения, используемый отделом обеспечения качества для проведения автоматического регрессионного тестирования. Тестировщики создают сценарии с помощью регистратора тестов, который фиксирует действия пользователя в отношении тестируемого приложения. Механизм записи создает тестовый скрипт из действий. Тестовый сценарий создается как приложение Java или Visual Basic.net и представлен в виде серии снимков экрана, которые образуют визуальную раскадровку. Тестировщики могут редактировать сценарий, используя стандартные команды и синтаксис этих языков, или действуя против снимков экрана в раскадровке. Затем Rational Functional Tester может выполнить тестовые сценарии для проверки функциональности приложения. Как правило, тестовые сценарии запускаются в пакетном режиме, когда несколько сценариев группируются и запускаются без присмотра.

Вывод

Регрессионное тестирование является ключом к улучшению общего качества продукта и пользовательского опыта. Правильные инструменты регрессионного тестирования могут значительно идентифицировать все обнаруженные дефекты и устранить их на ранних этапах разработки.

Кроме того, регрессионное тестирование предлагает множество технических и бизнес-преимуществ. Следовательно, чем больше организации инвестируют в планирование и проведение регрессионного тестирования, тем больше у них будет контроль над бюджетом, процессом и уменьшением ошибок своего продукта.

Эффективная регрессионная стратегия, экономит организации время и деньги. Согласно одному из тематических исследований в банковской сфере, регрессия экономит до 60% времени в исправлениях ошибок (которые были бы обнаружены регрессионными тестами) и до 40% в деньгах.

Тестирование правильности

После окончания тестирования интеграции программная система собрана в единый корпус, интерфейсные ошибки обнаружены и откорректированы. Теперь начинается последний шаг программного тестирования — ***тестирование правильности***. **Цель** — подтвердить, что функции, описанные в спецификации требований к ПС, соответствуют ожиданиям заказчика.

Подтверждение правильности ПС выполняется с помощью тестов «черного ящика», демонстрирующих соответствие требованиям. При обнаружении отклонений от спецификации требований создается список недостатков. Как правило, отклонения и ошибки, выявленные при подтверждении правильности, требуют изменения сроков разработки продукта.

Важным элементом подтверждения правильности является проверка конфигурации ПС.

Конфигурацией ПС называют совокупность всех элементов информации, вырабатываемых в процессе конструирования ПС. Минимальная конфигурация ПС включает следующие **базовые элементы**:

a) системную спецификация;
b) план программного проекта;
c) спецификацию требований к ПС; работающий или бумажный макет;
d) предварительное руководство пользователя;
e) спецификация проектирования;
f) листинги исходных текстов программ;
g) план и методику тестирования; тестовые варианты и полученные результаты;
h) руководства по работе и инсталляции;
i) exe-код выполняемой программы;
j) описание базы данных; руководство пользователя по настройке;
k) документы сопровождения; отчеты о проблемах ПС; запросы сопровождения; отчеты о конструкторских изменениях;
l) стандарты и методики конструирования ПС.

Проверка конфигурации гарантирует, что все элементы конфигурации ПС правильно разработаны, учтены и достаточно детализированы для поддержки этапа сопровождения в жизненном цикле ПС.

Разработчик не может предугадать, как заказчик будет реально использовать ПС. Для обнаружения ошибок, которые способен найти только конечный пользователь, используют процесс, включающий альфа- и бета-тестирование.

Альфа - тестирование: преимущества и недостатки

Разработчик не может предугадать, как заказчик будет реально использовать ПС. Для обнаружения ошибок, которые способен найти только конечный пользователь, используют процесс, включающий альфа- и бета-тестирование.

Альфа-тестирование проводится заказчиком в организации разработчика. Разработчик фиксирует все выявленные заказчиком ошибки и проблемы использования. После того как отдельные программные модули готовы, они объединяются в некое единое целое. Это еще не полнофункциональная программа, но она уже способна работать и выполнять, хотя бы частично, свои главные задачи. Такой вариант программы и называют *альфа-версией.*

Преимущества альфа-тестирования:

- Обеспечивает лучшее представление о надежности программного обеспечения на ранней стадии.
- Помогает моделировать поведение пользователя и окружающую среду в режиме реального времени.
- Обнаруживает много серьезных ошибок.
- Дает возможность раннего обнаружения ошибок в отношении дизайна и функциональности.

Недостатки альфа-тестирования:

Функциональность не может быть проверена на всю глубину, поскольку программное обеспечение все еще находится на стадии разработки. Иногда разработчики и тестировщики недовольны результатами альфа-тестирования.

На этапе альфа-тестирования основное внимание уделяется:

- обнаружению ошибок;
- вопросам по юзабилити;
- различию в характеристиках;
- проблемам совместимости / взаимодействия.

И главный вопрос, который возникает в Alpha Testing, - «Работает ли продукт?».

Альфа-тест обычно происходит в циклах, каждый из которых будет составлять примерно 1–2 недели. Количество циклов зависит от функций, включенных для тестирования, и количества проблем, обнаруженных на этом этапе тестирования. Могут быть применены методы тестирования White-box или Black-box или же их комбинация.

Бета-тестирование: преимущества и недостатки

Бета-тестирование *(beta testing)* – интенсивное использование почти готовой версии продукта с целью выявления максимального числа ошибок в его работе для их последующего устранения перед окончательным выходом (релизом) продукта на рынок, к массовому потребителю. Бета-тестирование представляет собой реально работающую версию программы с полным функционалом. И задача бета-тестов – оценить возможности и стабильность работы программы с точки зрения ее будущих пользователей.

Бета-тестирование предполагает привлечение добровольцев из числа обычных будущих пользователей продукта, которым доступна упомянутая предварительная версия продукта (так называемая бета-версия).

Такими добровольцами часто движет любопытство к новому продукту – любопытство, ради удовлетворения которого они вполне согласны мириться с возможностью испытать последствия еще не найденных (а потому и не исправленных) ошибок. Кроме любопытства, мотивация может быть обусловлена желанием повлиять на процесс разработки и в итоге получать более удовлетворяющий их нужды продукт и многим другим. Очень хорошо, если это люди, которые уже имеют опыт работы с программами такого типа, а еще лучше – с предыдущей версией этой же программы. Обычно у компаний уже есть определенный круг лиц, с которыми они постоянно сотрудничают.

Бета-тестирование проводится конечным пользователем в организации заказчика. Разработчик в этом процессе участия не принимает. Фактически, бета-тестирование — это реальное применение ПС в среде, которая не управляется разработчиком. Заказчик сам записывает все обнаруженные проблемы и сообщает о них разработчику. Бета-тестирование проводится в течение фиксированного срока (около года). По результатам выявленных проблем разработчик изменяет ПС и тем самым подготавливает продукт полностью на базе заказчика.

Бета-тестирование может быть:

- *Закрытым:* Программа тестируется в небольшой группе пользователей по приглашениям.
- *Открытым:* Этот вариант позволяет протестировать приложение в большей группе и получить большой объем обратной связи. Любой пользователь сможет присоединиться к открытому бета-тестированию и отправить личный отзыв.

Открытое бета-тестирования, например, может использоваться как часть стратегии продвижения продукта на рынок (например, бесплатная раздача бета-версий позволяет привлечь широкое внимание потребителей к окончательной дорогостоящей версии продукта), а также для получения предварительных отзывов о нём от широкого круга будущих пользователей.

Разработчики не испытывают недостатка в желающих принять участие в такой работе. Такого рода сотрудничество приносит пользу обеим сторонам, ведь исправления проще сделать в процессе работы, а не когда она уже завершена, к тому же замечания и пожелания пользователей позволяют сделать программное обеспечение лучше и качественней.

Преимущества бета-тестирования:

- Снижает риск выхода продукта из строя посредством валидации клиента.
- Бета-тестирование позволяет компании тестировать инфраструктуру после запуска.
- Повышает качество продукции благодаря обратной связи с клиентами.
- Является экономичным методом сбора данных по сравнению с аналогичными методами.
- Создает доброжелательность с клиентами и повышает удовлетворенность клиентов.

Недостатки бета-тестирования:

- Управление тестированием – проблема. По сравнению с другими типами тестирования, которые обычно выполняются внутри компании в контролируемой среде, бета-тестирование выполняется в реальном мире, где у компании редко есть контроль.
- Поиск правильных пользователей бета-версии и поддержание их участия может вызвать трудности.

Поскольку бета-тестирование происходит на стороне конечного пользователя, это не может быть контролируемым действием.

Цель бета-тестирования

Описанные ниже моменты можно рассматривать как цели для бета-теста, и они необходимы для получения гораздо лучших результатов:

a) Бета-тест предоставляет полный обзор истинного опыта, накопленного конечными пользователями при работе с продуктом.

b) Он выполняется широким кругом пользователей, и причины, по которым продукт используется, сильно различаются. Менеджеры по маркетингу ориентируются на мнение целевого рынка о каждой функции, а вот обычные пользователи - на простоту использования продукта, а технические сосредотачиваются на опыте по установке, деинсталляции и т. д. Но фактическое восприятие конечных пользователей ясно показывает, почему они нуждаются в этом продукте и как они будут его использовать.

c) В реальном мире совместимость продукта может быть обеспечена в большей степени благодаря этому тестированию, поскольку здесь используется большое сочетание реальных платформ для тестирования на широком спектре устройств, ОС, браузеров и т. д.

d) Поскольку широкий диапазон платформ, которые фактически используют конечные пользователи, может быть недоступен для внутренней группы тестировщиков, это тестирование также помогает выявить скрытые ошибки и пробелы в конечном продукте.

Когда выполняется бета-тестирование?

Бета-тестирование всегда выполняется сразу после завершения Альфа-тестирования, но до выхода продукта на рынок. Здесь ожидается, что продукт будет готов не менее чем на 90%–95% (достаточно стабильно работать на любой платформе, все функции почти или полностью завершены). В идеале все технические продукты должны пройти стадию бета-тестирования, поскольку они в основном зависят от платформ и процессов.

Обычно продолжительность бета-теста это один или два тестовых цикла с 4 ÷ 6 неделями в цикле. Он расширяется только при добавлении новой функции или при изменении основного компонента.

Бета-тест-план может быть написан многими способами в зависимости от степени его выполнения.

Общие элементы для любого плана тестирования бета-версии:

- Цель: указать цель проекта, чтобы он прошел бета-тестирование даже после тщательных внутренних тестов.
- Сфера применения: четко укажите, какие области необходимо тестировать, а какие нет. Также укажите конкретные данные, которые будут использоваться для определенной функции.
- Подход к тестированию: четко укажите, на чем стоит сосредоточиться пользователю - на функциональности, пользовательском интерфейсе, и т. д. Не забывайте о процедуре регистрации ошибок, а также всем, что необходимо для доказательства (скриншоты / видео).
- График: четко указывайте начальную и конечную даты тестирования, количество циклов и их продолжительность.
- Инструменты: у вас должен быть инструмент регистрации ошибок.
- Обратная связь: сбор отзывов и методов оценки.
- Определите и просмотрите критерии входа и выхода.

Эффективное выполнение плана приведет к успеху фазы тестирования.

Как выполняется бета-тестирование

Этот тип тестирования может быть выполнен несколькими способами, но в целом существует пять разных этапов.

a) Планирование

Определите цели заранее. Это помогает в планировании количества пользователей, участвующих в тестировании, и продолжительности, требуемой для завершения и достижения целей.

b) Набор участников

В идеале любое количество пользователей может участвовать в тестировании, но из-за бюджетных ограничений проект должен установить минимальное и максимальное ограничение количества участвующих пользователей. Как правило это 50–250 пользователей в среднем.

c) Запуск продукта

1) Инсталляционные пакеты должны быть распространены среди участников. В идеале, поделитесь ссылкой, откуда они могут скачать и установить.
2) Поделитесь руководствам пользователя, известными проблемами, объемом тестирования и т. д.
3) Ознакомьте участников со способами регистрации ошибок.

Этапы бета-тестирование

d) Сбор и оценка отзывов

1) Ошибки, отзывы и предложения участников обрабатываются.
2) Отзывы анализируются и оцениваются потребности клиента при использовании данного продукта.
3) Считается, что рекомендуется улучшить продукт в следующих версиях.

e) Закрытие

1) Как только достигнут определенный момент и когда все функции работают, ошибок не возникает, критерии выхода выполняются, тогда можно завершить этап бета-тестирования.
2) Распределите вознаграждения / поощрения участникам в соответствии с планом и официально поблагодарите их (это помогает в дальнейшем бета-тестировании продукта, вы получаете гораздо больше отзывов, предложений и т. д.).

Управление этапом тестирования

Управление всей бета-фазой — это не что иное, как вызов, так как ее нельзя контролировать сразу после запуска. Таким образом, это всегда хорошая практика для создания дискуссий на форуме и включения всех участников для участия в ней. Проводите опросы по опыту использования продукта и просите участников писать отзывы.

Как компании могут успешно выполнить бета-тест:

a) Сначала решите, сколько дней вы хотите сохранять бета-версию для тестировщиков.

b) Определите идеальные группы пользователей для выполнения этого теста. Это будет либо ограниченная группа пользователей, либо проект будет выставлен на широкую публику.

c) Предоставьте четкие инструкции по тестированию (руководство пользователя).

d) Сделайте бета-версию доступной для этих групп - собирайте обратную связь и дефекты.

e) На основе анализа обратной связи решайте, какие вопросы необходимо исправлять до окончательной версии.

f) После исправления предложений и дефектов снова дайте измененную версию для проверки в те же группы.

g) Как только все тесты будут завершены, не принимайте никаких дополнительных запросов на изменение этой версии.

h) Удалите бета-метку и выпустите окончательную версию программного обеспечения.

Отличия между альфа- и бета-тестированием

Альфа-тестирование	Бета-тестирование
Что делает	
Повышает качество продукта и обеспечивает готовность к бета-тестированию.	Повышает качество продукта, интегрирует данные о клиенте в готовый продукт и обеспечивает готовность к выпуску.
Когда проводится	
Ближе к концу процесса разработки, когда продукт находится в почти полностью работоспособном состоянии.	Непосредственно перед запуском.
Как долго проводится	
Обычно очень долго в течении многих итераций. Альфа-тестирование нередко длится в 3–5 раз больше длительности бета-тестирования.	Обычно только несколько недель (иногда до нескольких месяцев) с небольшим количеством основных итераций.
Что включает в себя	
Почти исключительно включает проверку качества ПО (баги, баги, баги).	Обычно включает в себя маркетинг, поддержку, документацию, качество и инжиниринг (в основном, всю группу продуктов).

Альфа-тестирование	Бета-тестирование
Кто проводит	
Обычно выполняется тестировщиками, разработчиками, а иногда и «друзьями и семьей». Фокусируется на тестировании, которое будет эмулировать ~ 80% клиентов.	Испытано в «реальном мире» с «реальными клиентами», и обратная связь может охватывать каждый элемент продукта.
Что получают тестировщики	
Множество ошибок, сбоев, недостающих документов и функций.	Несколько ошибок, меньше сбоев, большинство документации и функций завершены.
Отладка	
Большинство известных критических проблем исправлены, некоторые функции могут быть изменены или добавлены в результате ранней обратной связи.	Большая часть собранной обратной связи рассматривается и/или применяется в будущих версиях продукта. Выполняются только важные/критические изменения.
Как тестируют	
Тестируют по методологии, показателям эффективности. Хороший альфа-тест задает четко определенные критерии и измеряет продукт по отношению к этим ориентирам.	Тестируют с применением реальности и воображения. Бета-тесты исследуют пределы продукта, позволяя клиентам исследовать каждый элемент продукта в своей родной среде.

Альфа-тестирование	Бета-тестирование
Результат	
Есть отличное представление о том, как работает продукт и соответствует ли он критериям дизайна (и «бета-готов» ли он).	Есть представление о том, что ваш клиент думает о продукте и о том, что он может испытывать, когда покупает его.
Что дальше	
Бета-тестирование.	Выпуск.

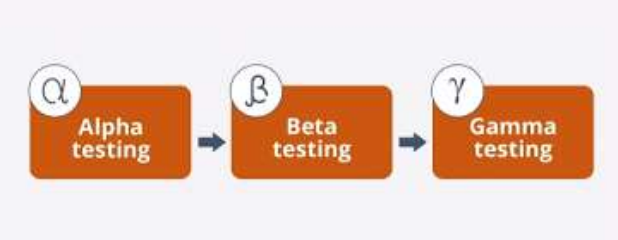
Вывод

Приемочное тестирование помогает определить эффективность команд разработчиков и тестировщиков. Существует несколько инструментов для выполнения этого действия, но обычно это предпочтительно делать вручную, поскольку в нем участвуют реальные пользователи и различные заинтересованные стороны, которые не имеют технической подготовки, и это может быть нецелесообразно для них.

Использование Alpha Testing на ранней стадии разработки программного обеспечения обеспечивает лучшее понимание продукта, точку зрения целевых пользователей и опыт при использовании продукта, а также выполнение альфа-тестирования помогает успешно решать свои задачи и ведет к успешному использованию клиентом.

До тех пор, пока пользователям не понравятся продукт, его никогда нельзя считать успешным. Бета-тестирование - одна из таких методологий, которая позволяет пользователям испытать продукт до его выхода на рынок. Тщательное тестирование на разных платформах и ценная обратная связь в итоге приводит к успешному релизу и гарантирует, что Клиент будет удовлетворен его использованием.

Гамма-тестирование



Часть авторов (например, Святослав Куликов в своем учебнике “Тестирование программного обеспечения. Базовый курс”) еще выделяют третий вид тестирования - гамма-тестирование.

Гамма-тестирование (gamma testing) — финальная стадия тестирования перед выпуском продукта, направленная на исправление незначительных дефектов, обнаруженных в бета-тестировании.

Как правило, также выполняется с максимальным привлечением конечных пользователей/заказчиков. Суть этого вида вкратце: продукт уже почти готов, и сейчас обратная связь от реальных пользователей используется для устранения последних недоработок.

Стоит отметить, что, обычно, тестирование все же делят на альфа и бета без упоминания о гамма-тестировании.

На этапе гамма-тестирования не в окончательном виде могут быть только документация и упаковка.

Системное тестирование

Системное тестирование подразумевает выход за рамки области действия программного проекта и проводится не только программным разработчиком. Классическая проблема системного тестирования — указание причины. Она возникает, когда разработчик одного системного элемента обвиняет разработчика другого элемента в причине возникновения дефекта. Для защиты от подобного обвинения разработчик программного элемента должен:

a) предусмотреть средства обработки ошибки, которые тестируют все вводы информации от других элементов системы;

b) провести тесты, моделирующие неудачные данные или другие потенциальные ошибки интерфейса ПС;

c) записать результаты тестов, чтобы использовать их как доказательство невиновности в случае «указания причины»;

d) принять участие в планировании и проектировании системных тестов, чтобы гарантировать адекватное тестирование ПС.

В конечном счете системные тесты должны проверять, что все системные элементы правильно объединены и выполняют назначенные функции. Рассмотрим основные типы системных тестов.

Системное тестирование качественно отличается от интеграционного и модульного уровней. Системное тестирование рассматривает тестируемую систему в целом и оперирует на уровне пользовательских интерфейсов, в отличие от последних фаз интеграционного тестирования, которое оперирует на уровне интерфейсов модулей. Различны и цели этих уровней тестирования. На уровне системы часто сложно и малоэффективно анализировать прохождение тестовых траекторий внутри программы или отслеживать правильность работы конкретных функций.

Основная задача системного тестирования в выявлении дефектов, связанных с работой системы в целом, таких как:

– неверное использование ресурсов системы;

– непредусмотренные комбинации данных пользовательского уровня;

– несовместимость с окружением;

– непредусмотренные сценарии использования;

– отсутствующая или неверная функциональность;

– неудобство в применении и тому подобное.

Системное тестирование производится над проектом в целом с помощью метода «черного ящика». Структура программы не имеет никакого значения, для проверки доступны только входы и выходы, видимые пользователю. Тестированию подлежат коды и пользовательская документация.

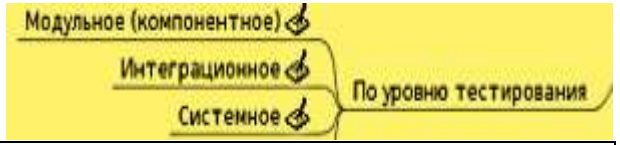
Категории тестов системного тестирования

Выделяют следующие виды системных тестов:

1. Полнота решения функциональных задач.
2. Стрессовое тестирование - на предельных объемах нагрузки входного потока.
3. Корректность использования ресурсов (утечка памяти, возврат ресурсов).
4. Оценка производительности.
5. Эффективность защиты от искажения данных и некорректных действий.
6. Проверка инсталляции и конфигурации на разных платформах.
7. Корректность документации.

Поскольку системное тестирование проводится на пользовательских интерфейсах, создается иллюзия того, что построение специальной системы автоматизации тестирования не всегда необходимо. Однако объемы данных на этом уровне таковы, что обычно более эффективным подходом является полная или частичная автоматизация тестирования, что приводит к созданию тестовой системы гораздо более сложной, чем система тестирования, применяемая на уровне тестирования модулей или их комбинаций. Системную фазу тестирования, как наиболее сложную и критичную для процесса разработки следует рассматривать в рамках индустриального подхода.

Характеристики модульного, интеграционного и *системного тестирования*



	Модульное	**Интеграционное**	**Системное**
Типы дефектов	Локальные дефекты, такие как опечатки в реализации алгоритма, неверные операции, логические и математические выражения, циклы, ошибки в использовании локальных ресурсов, рекурсия и т.п.	Интерфейсные дефекты, такие как неверная трактовка параметров и их формат, неверное использование системных ресурсов и средств коммуникации, и т.п.	Отсутствующая или некорректная функциональность, неудобство использования, непредусмотренные данные и их комбинации, непредусмотренные или неподдерживаемые сценарии работы, ошибки совместимости, ошибки пользовательской документации, ошибки переносимости продукта на различные платформы, проблемы производительности, инсталляции и т.п.
Необходимость в системе тестирования	Да	Да	Нет
Цена разработки системы тестирования	Низкая	Низкая до умеренной	Умеренная до высокой или неприемлемой
Цена процесса тестирования, т.е разработки, прогона и анализа тестов	Низкая	Низкая	Высокая

Тестирование восстановления

Многие компьютерные системы:

должны восстанавливаться после отказов и возобновлять обработку в пределах заданного времени;

должны быть отказоустойчивыми, то есть отказы обработки не должны быть причиной прекращения работы системы;

в случаи системного отказа устранить его в пределах заданного кванта времени, иначе заказчику наносится серьезный экономический ущерб.

Тестирование восстановления использует самые разные пути для того, чтобы заставить ПС отказать, и проверяет полноту выполненного восстановления.

При **автоматическом восстановлении** оцениваются:

- правильность повторной инициализации;
- механизмы копирования контрольных точек;
- восстановление данных;
- перезапуск.

При **ручном восстановлении** оценивается, находится ли среднее время восстановления в допустимых пределах.

Тестирование безопасности

Компьютерные системы очень часто являются мишенью незаконного проникновения. Под проникновением понимается широкий диапазон действий: попытки хакеров проникнуть в систему из спортивного интереса, месть рассерженных служащих, взлом мошенниками для незаконной наживы.

Тестирование безопасности проверяет фактическую реакцию защитных механизмов, встроенных в систему, на проникновение.

В ходе тестирования безопасности испытатель играет роль взломщика. Ему разрешено все:

a) попытки узнать пароль с помощью внешних средств;

b) атака системы с помощью специальных утилит, анализирующих защиты;

c) подавление, ошеломление системы (в надежде, что она откажется обслуживать других клиентов);

d) целенаправленное введение ошибок в надежде проникнуть в систему в ходе восстановления;

e) просмотр несекретных данных в надежде найти ключ для входа в систему.

Конечно, при неограниченном времени и ресурсах хорошее тестирование безопасности взломает любую систему. **Задача проектировщика системы** — сделать цену проникновения более высокой, чем цена получаемой в результате информации.

ЛЕКЦИЯ 12

ТЕМА 9: Регрессионное тестирование. Тестирование правильности. Альфа-тестирование. Бета-тестирование. Гамма-тестирование. Системное тестирование. Тестирование восстановления. Тестирование безопасности

Виды тестирования по времени проведения

Наш продукт может быть не полностью "собран", но это не мешает ему быть объектом тестирования. Кроме того, он может быть в принципе готов "выйти в свет", но было бы неплохо посмотреть на него глазами конечных потребителей.

Именно это и послужило причиной деления тестирования на альфа и бета тестирование. По тому же принципу делятся и пользователи, которые проводят данные виды тестирования. Подробнее обо всем в статье.

Альфа и бета тестирование являются одной из форм приемочного тестирования. То есть ошибок на этом этапе тестирования уже быть не должно. Скорее наоборот, программа должна быть максимально рабочей и пригодной для использования.

Альфа-тестирование

После подготовки отдельных модулей продукта они объединяются в единое целое. Это еще не готовая версия, но она уже способна работать и выполняет свои основные задачи (иногда частично). Этот вариант программы и называют альфа-версией.

Альфа-тестирование (alpha testing) – это вид приемочного тестирования, которое обычно проводится на поздней стадии разработки продукта и включает имитацию реального использования продукта штатными разработчиками либо командой тестировщиков.

Обычно альфа тестирование заключается в систематической проверке всех функций программы с использованием техник тестирования «белого ящика» и «черного ящика».

Альфа-тестирование является методологией оценки качества и стабильности тестируемого продукта в тестовой среде.

На этом этапе **основное внимание** уделяется:

- обнаружению ошибок,
- вопросам по юзабилити,
- различию в характеристиках,
- проблемам совместимости/взаимодействия.
- **фазы** альфа-тестирования:

  - Предварительное альфа-тестирование: пользовательский интерфейс продукта готов, но функции еще не реализованы полностью. На этом этапе обычно принимается решение о том, какие функции следует вводить в продукт. Прототип продукта постоянно пересматривается и анализируется для большего улучшения.
  - Альфа-тестирование: рабочий продукт готов к тестированию.

Альфа-тест обычно происходит в циклах, каждый из которых будет составлять примерно 1–2 недели. Количество циклов зависит от функций, включенных для тестирования, и количества проблем, обнаруженных на этом этапе тестирования.

Критерии входа:

- Тестирование системы должно быть завершено и подписано (если альфа-тестирование начнется после завершения тестирования системы).
- Продукт готов на 70%–90%.
- Участники должны быть идентифицированы и знать особенности продукта.
- Альфа-тесты разработаны и рассмотрены.
- Настроена среда тестирования и подтверждена стабильность.
- Билд для альфа-версии продукта готов и запущен с полными примечаниями к выпуску.

Критерии выхода:

- Функциональные ошибки исправлены.
- Все тестовые циклы полные.
- Все запланированные тесты выполнены и пройдены.
- Функции замораживаются (то есть никаких дополнительных функций, никаких изменений в существующих функциях).

Преимущества альфа-тестирования:

- Обеспечивает лучшее представление о надежности программного обеспечения на ранней стадии.
- Помогает моделировать поведение пользователя и окружающую среду в режиме реального времени.
- Комплексный подход позволяет обнаружить много **серьезных и дорогостоящих ошибок**.
- Опытные специалисты **пропустят меньше ошибок**, чем специалист без опыта. Также, не стоит забывать, что команда тестировщиков несет полную ответственность за свою работу, чего никак нельзя сказать об участниках бета-тестирований. Это во многом сказывается на качестве тестирования в положительную сторону.
- Дает возможность **раннего обнаружения ошибок** в отношении дизайна и функциональности.
- Помогает понять **факторы, влияющие на успешный выпуск** продукта.
- Развернутый отчет о результатах тестирования, в котором все излагается так, что **разработчики могут сразу приступать** к исправлению багов.

Недостатки альфа-тестирования:

- Функциональность **не может быть проверена на всю глубину**, поскольку программное обеспечение все еще находится на стадии разработки. Иногда разработчики и тестировщики недовольны результатами альфа-тестирования.
- **Дорого**. Далеко не каждая компания может себе позволить расширять штат и нанимать сотрудников-тестировщиков на постоянную основу, особенно если речь идет о стартапах. Многие выбирают аутсорс. Тем не менее, это все равно финансовые затраты.
- Может быть **недостаточно глубоким** для нахождения всех багов.
- **Долго**. По сравнению с бета-тестирование, альфа-тестирование длится намного дольше, так как требует определенной подготовки. Ведь в этом случае необходимо подготавливать тест-кейсы и чек-листы, разрабатывать множество всевозможных сценариев, план проверки и т. п.

Бета-тестирование

По окончании работы с альфа-версией выпускается бета-версия. Она представляет собой реально работающую версию программы с полным функционалом.

Бета-тестирование (beta testing) – интенсивное использование почти готовой версии продукта с целью выявления максимального числа ошибок в его работе для их последующего устранения перед окончательным выходом (релизом) продукта на рынок, к массовому потребителю.

Бета-тестирование представляет собой реально работающую версию программы с полным функционалом.

Задача бета-тестов – оценить возможности и стабильность работы программы с точки зрения ее будущих пользователей. Поэтому, в отличие от альфа-тестирования, бета-тестирование предполагает привлечение добровольцев из числа обычных будущих пользователей продукта.

Такими добровольцами (бета-тестерами) часто движет любопытство к новому продукту. Они вполне согласны мириться с возможностью столкнуться с ошибками. Кроме любопытства, мотивация может быть обусловлена желанием повлиять на процесс разработки и получить более подходящий им продукт.

Будет плюсом, если это люди, которые ***уже имеют опыт работы*** с программами такого типа, а еще лучше – с предыдущей версией этой же программы. Обычно у компаний уже есть определенный круг лиц, с которыми они постоянно сотрудничают.

Надо сказать, что разработчики не испытывают недостатка в желающих принять участие в такой работе. Многие люди хотят попасть в закрытую группу, первыми узнать о новых функциях продукта, пользоваться тем, чего нет в свободном доступе.

*Наглядный **пример набора бета-тестировщиков** — социальная сеть Яндекс.Аура. Создатели сделали ограниченный доступ по ссылкам. Таким образом, в первые дни они набрали достаточное количество пользователей для первых тестов и закрыли набор. То есть каждый пользователь чувствовал себя там уникальным и с радостью помогал тестировать новый продукт. Продукт, к которому нет доступа больше ни у кого.*

Кстати, этим пользователям еще присвоили особенные “метки”, которые говорят о том, что они первопроходцы. Это тоже отличный стимул быть тем самым первым пользователем.

Разные компании выполняют бета-тестирование по-разному. Есть 2 вида бета-тестирования:

- ***Открытое*** бета-тестирование, когда продукт доступен для всех желающих. Этот вариант позволяет протестировать приложение в большей группе и получить большой объем обратной связи. Любой пользователь сможет присоединиться к открытому бета-тестированию и отправить личный отзыв.

- ***Закрытое*** бета-тестирование, когда продукт тестируется ограниченным количеством пользователей. Обычно это небольшая группа, в которую попадают по приглашениям.

Преимущества бета-тестирования:

- **Снижает риск** выхода продукта из строя посредством валидации клиента.
- Бета-тестирование позволяет компании тестировать **инфраструктуру после запуска**.
- **Повышает качество** продукции благодаря обратной связи с клиентами. Пользователи не только пишут про найденные баги, а также вносят свои предложения и пожелания по улучшению функционала продукта.
- Выявляют **баги, которые были не выявлены** на предыдущих этапах тестирования. Дешевле исправить их на данном этапе, чем при релизе на всех конечных пользователей.
- Является **экономичным методом сбора данных** по сравнению с аналогичными методами. Однако не всегда удается получать обратную связь без вложений. Иногда пользователям предлагают какие-то бонусы за найденные баги. Тем самым мотивируя их на активные действия.
- Создает **доброжелательность** с клиентами и повышает удовлетворенность клиентов.

Недостатки бета-тестирования:

- Управление тестированием – проблема. По сравнению с другими типами тестирования, которые обычно выполняются внутри компании в контролируемой среде, бета-тестирование выполняется в реальном мире, где у компании **редко есть контроль**.
- **Качество тестирования не на высоте**. Бета-тестировщики не всегда обладают даже минимальными техническими навыками, позволяющими дать качественную оценку ПО. Конечно, о чёткой локализации и понятном описании бага при бета-тестировании можно даже не говорить. Обычно это ложиться на плечи штатных тестировщиков.
- Поиск правильных пользователей бета-версии и поддержание их участия может **вызвать трудности**.
- Есть риск, что бета-тестировщики **разместят информацию** о вашем продукте еще до его выхода. Это может значительно подпортить ожидаемый от релиза эффект.
- **Неполное тестовое покрытие**. Практика показывает, при тестировании непрофессионалами часть функционала всегда остается не охваченной. Все же бета- тестировщики не профессионалы.

Гамма-тестирование

Часть авторов (например, Святослав Куликов в своем учебнике “Тестирование программного обеспечения. Базовый курс”) еще выделяют третий вид тестирования - гамма-тестирование.

Гамма-тестирование (gamma testing) — финальная стадия тестирования перед выпуском продукта, направленная на исправление незначительных дефектов, обнаруженных в бета-тестировании.

Как правило, также выполняется с максимальным привлечением конечных пользователей/заказчиков. Суть этого вида вкратце: продукт уже почти готов, и сейчас обратная связь от реальных пользователей используется для устранения последних недоработок.

Стоит отметить, что, обычно, тестирование все же делят на альфа и бета без упоминания о гамма-тестировании.

Альфа и бета. В чем отличия?

Таблица сравнения

	Альфа-тестирование	Бета-тестирование
Кто	Тестировщики и разработчики.	Клиенты/заказчики или конечные пользователи.
Когда	Ближе к концу процесса разработки, продукт в почти полностью работоспособном состоянии.	Ближе к релизу.
Сколько	Может потребоваться долгий цикл. Альфа-тестирование нередко в 3-5 раз длительнее бета-тестирования.	Несколько недель (иногда месяцев) с небольшим количеством основных итераций.
Цель	Повысить качество продукта и обеспечить готовность к бета-тестированию.	Повысить качество продукта и проверить готовность продукта к использованию реальными пользователями.
Требования к среде	Требует лабораторной и тестовой среды.	Не требует лабораторной и тестовой среды, проводится в режиме реального времени.
Методы тестирования	Используются все известные и актуальные для данного продукта методы тестирования.	Тестируют с применением реальности и воображения. Исследуют каждый элемент продукта в своей родной среде.
Фиксы (устранение багов)	Большинство известных критических проблем исправлены, некоторые функции могут быть изменены или добавлены в результате ранней обратной связи.	Большая часть собранной обратной связи рассматривается и/или применяется в будущих версиях продукта. Выполняются только важные/критические изменения.
Результат	Имеем представление о качестве продукта и соответствует ли он документации.	Узнаем, что пользователи думают о продукте, в чем видят его улучшения.

Как видно, альфа и бета тестирование имеют разные цели и задачи. У каждого из них есть свои плюсы и минусы. Конечно, будет неплохо использовать оба этих метода, так как они приносят свои «плоды». Но это не всегда возможно и финансово обосновано.

Очень часто бета-тестирование практикуется в игровой сфере. Все же один из основных плюсов бета-тестирования — понимание того, нужен ли аудитории наш продукт и стоит ли его выпускать в релиз.

ЛЕКЦИЯ 13

ТЕМА 10: Стрессовое тестирование. Тестирование производительности. Искусство отладки. Автоматизация тестирования. Инструментарии

Стрессовое тестирование

На предыдущих шагах тестирования способы «белого» и «черного ящиков» обеспечивали полную оценку нормальных программных функций и качества функционирования. Стрессовые тесты проектируются для навязывания программам ненормальных ситуаций. В сущности, проектировщик стрессового теста спрашивает, как сильно можно расшатать систему, прежде чем она откажет?

Стрессовое тестирование производится при ненормальных запросах на ресурсы системы (по количеству, частоте, размеру-объему).

Примеры:

a) генерируется 10 прерываний в секунду (при средней частоте 1, 2 прерывания в секунду);

b) скорость ввода данных увеличивается прямо пропорционально их важности (чтобы определить реакцию входных функций);

c) формируются варианты, требующие максимума памяти и других ресурсов;

d) генерируются варианты, вызывающие переполнение виртуальной памяти;

e) проектируются варианты, вызывающие чрезмерный поиск данных на диске.

По существу, испытатель пытается разрушить систему. Разновидность стрессового тестирования называется ***тестированием чувствительности***. В некоторых ситуациях (обычно в математических алгоритмах) очень малый диапазон данных, содержащийся в границах правильных данных системы, может вызвать ошибочную обработку или резкое понижение производительности. Тестирование чувствительности обнаруживает комбинации данных, которые могут вызвать нестабильность или неправильность обработки.

Тестирование производительности

В системах реального времени и встроенных системах недопустимо ПО, которое реализует требуемые функции, но не соответствует требованиям производительности.

Тестирование производительности проверяет скорость работы ПО в компьютерной системе. Производительность тестируется на всех шагах процесса тестирования. Даже на уровне элемента при проведении тестов «белого ящика» может оцениваться производительность индивидуального модуля. Тем не менее, пока все системные элементы не объединятся полностью, не может быть установлена истинная производительность системы. Иногда тестирование производительности сочетают со стрессовым тестированием. При этом нередко требуется специальный аппаратный и программный инструментарий. Например, часто требуется точное измерение используемого ресурса (процессорного цикла и т. д.). Внешний инструментарий регулярно отслеживает интервалы выполнения, регистрирует события (например, прерывания) и машинные состояния. С помощью инструментария испытатель может обнаружить состояния, которые приводят к деградации и возможным отказам системы.

Искусство отладки

Отладка — это локализация и устранение ошибок. Отладка является следствием успешного тестирования. Это значит, что если тестовый вариант обнаруживает ошибку, то процесс отладки уничтожает ее.

Итак, процессу отладки предшествует выполнение тестового варианта. Его результаты оцениваются, регистрируется несоответствие между ожидаемым и реальным результатами. Несоответствие является симптомом скрытой причины. Процесс отладки пытается сопоставить симптом с причиной, вследствие чего приводит к исправлению ошибки. Возможны **два исхода процесса отладки**:

a) причина найдена, исправлена, уничтожена;

b) причина не найдена.

Во втором случае отладчик может предполагать причину. Для проверки этой причины он просит разработать дополнительный тестовый вариант, который поможет проверить предположение. Таким образом, запускается итерационный процесс коррекции ошибки. Возможные разные способы проявления ошибок:

a) программа завершается нормально, но выдает неверные результаты;

b) программа зависает;

c) программа завершается по прерыванию;

d) программа завершается, выдает ожидаемые результаты, но хранимые данные испорчены (это самый неприятный вариант).

Характер проявления ошибок также может меняться.

Симптом ошибки может быть:

a) постоянным;

b) мерцающим;

c) пороговым (проявляется при превышении некоторого порога в обработке — 200 самолетов на экране отслеживаются, а 201-й — нет);

d) отложенным (проявляется только после исправления маскирующих ошибок).

В ходе отладки мы встречаем ошибки в широком диапазоне: от мелких неприятностей до катастроф. Следствием увеличения ошибок является усиление давления на отладчика — «найди ошибки быстрее!!!». Часто из-за этого давления разработчик устраняет одну ошибку и вносит две новые ошибки.

Английский термин *debugging* (отладка) дословно переводится как «ловля блох», который отражает специфику процесса — погоню за объектами отладки, «блохами». Рассмотрим, как может быть организован этот процесс «ловли блох».

Различают две группы **методов отладки**:

a) аналитические;

b) экспериментальные.

Аналитические методы базируются на анализе выходных данных для тестовых прогонов. Экспериментальные методы базируются на использовании вспомогательных средств отладки (отладочные печати, трассировки), позволяющих уточнить характер поведения программы при тех или иных исходных данных.

Общая стратегия отладки — обратное прохождение от замеченного симптома ошибки к исходной аномалии (месту в программе, где ошибка совершена).

В простейшем случае место проявления симптома и ошибочный фрагмент совпадают. Но чаще всего они далеко отстоят друг от друга.

Цель отладки — найти оператор программы, при исполнении которого правильные аргументы приводят к неправильным результатам. Если место проявления

симптома ошибки не является искомой аномалией, то один из аргументов оператора должен быть неверным. Поэтому надо перейти к исследованию предыдущего оператора, выработавшего этот неверный аргумент. В итоге пошаговое обратное прослеживание приводит к искомому ошибочному месту.

В разных методах прослеживание организуется по-разному. В **аналитических методах** — на основе логических заключений о поведении программы. **Цель** — шаг за шагом уменьшать область программы, подозреваемую в наличии ошибки. Здесь определяется корреляция между значениями выходных данных и особенностями поведения.

Основное ***преимущество аналитических методов*** *отладки* состоит в том, что исходная программа остается без изменений.

В экспериментальных методах для прослеживания выполняется:

a) Выдача значений переменных в указанных точках.
b) Трассировка переменных (выдача их значений при каждом изменении).
c) Трассировка потоков управления (имен вызываемых процедур, меток, на которые передается управление, номеров операторов перехода).

Преимущество экспериментальных методов отладки состоит в том, что основная рутинная работа по анализу процесса вычислений перекладывается на компьютер. Многие трансляторы имеют встроенные средства отладки для получения информации о ходе выполнения программы.

Недостаток экспериментальных методов отладки — в программу вносятся изменения, при исключении которых могут появиться ошибки. Впрочем, некоторые системы программирования создают специальный отладочный экземпляр программы, а в основной экземпляр не вмешиваются.

ЛЕКЦИИ 14, 15

ТЕМА 10: Стрессовое тестирование. Тестирование производительности. Искусство отладки. Автоматизация тестирования. Инструментарии

Отладка программного обеспечения

Целью лекции является дать представление о процессе и методах отладки программного обеспечения.

План лекции

1. Классификация ошибок

2. Методы отладки программного обеспечения

3. Методы и средства получения дополнительной информации об ошибке

4. Методика отладки программного обеспечения

Список литературы

Основная литература

1. Иванова Г.С. Технология программирования М.: Изд-во МГТУ им. Н.Э. Баумана, 2002. - 320 с.

2. Жоголев Е.А. Технология программирования М.: Научный мир, 2004. - 216 с.

3. Гагарина Л.Г., Кокорева Е.В., Виснадул Б.Д. Технология разработки программного обеспечения. М.: ИД "ФОРУМ" - ИНФРА-М, 2008. - 400с .

Дополнительная литература

1. Канер С., Фолк Д., Нгуен Е. Тестирование программного обеспечения. М.: ДиаСофт, 2001. - 544с.

2. Брауде Э. Технология разработки программного обеспечения. СПб.: Питер, 2004. - 655 с.

3. Баранов С.Н., Домарацкий А.Н., Ласточкин Н.К., Морозов В.П. Процесс разработки программных изделий. М.: ФИЗМАТЛИТ, Наука, 2000. - 176с.

Internet-ресурсы

1. www.intuit.ru- Интернет-университет информационных технологий.

2. http://citforum.ru/- Центр информационных технологий.

3. http://www.tstu.ru/r.php?r=education- Электронная библиотека ТГТУ.

4. http://www.edu.ru/- Библиотека Федерального портала «Российское образование»

Отладка программы — один их самых сложных этапов разработки программного обеспечения, требующий глубокого знания:

- специфики управления используемыми техническими средствами;
- операционной системы;
- среды и языка программирования;
- реализуемых процессов;
- природы и специфики различных ошибок;
- методик отладки и соответствующих программных средств.

Обсуждению последних двух вопросов и посвящается данная лекция.

Программа, свободная от ошибок, есть абстрактное теоретическое понятие.

Отладка программы (program debugging) - этап разработки программы, состоящий в локализации, выявлении и устранении программных ошибок, факт существования которых уже установлен.

Локализацией называют процесс определения оператора программы, выполнение которого вызвало нарушение нормального вычислительного процесса. Для исправления ошибки необходимо определить ее причину, т. е. определить оператор или фрагмент, содержащие ошибку. Причины ошибок могут быть как очевидны, так и очень глубоко скрыты.

Отладка имеет место тогда, когда очевидно, что программа либо не компилируется, либо работает неправильно.

Отладка программы предполагает обязательное наличие той или иной ошибки, в противном случае речь идет о тестировании.

Сложность отладки

Причины:

- требует от программиста глубоких знаний специфики управления используемыми техническими средствами, операционной системы, среды и языка программирования, реализуемых процессов, природы и специфики различных ошибок, методик отладки и соответствующих программных средств;
- психологически дискомфортна, так как необходимо искать собственные ошибки и, как правило, в условиях ограниченного времени;
- возможно взаимовлияние ошибок в разных частях программы, например, за счет затирания области памяти одного модуля другим из-за ошибок адресации;
- отсутствуют четко сформулированные методики отладки.

Классификация ошибок

В соответствии с этапом обработки, на котором появляются ошибки, различают (рисунок 10.1):

- **синтаксические ошибки** - ошибки, фиксируемые компилятором (транслятором, интерпретатором) при выполнении синтаксического и частично семантического анализа программы;
- **ошибки компоновки** - ошибки, обнаруженные компоновщиком (редактором связей) при объединении модулей программы;
- **ошибки выполнения** - ошибки, обнаруженные операционной системой, аппаратными средствами или пользователем при выполнении программы.

Рисунок 10.1 - Классификация ошибок по этапу обработки программы

Синтаксические ошибки

Синтаксические ошибки относят к группе самых простых, так как синтаксис языка, как правило, строго формализован, и ошибки сопровождаются развернутым комментарием с указанием ее местоположения. Определение причин таких ошибок, как правило, труда не составляет, и даже при нечетком знании правил языка за несколько прогонов удается удалить все ошибки данного типа.

Следует иметь в виду, что чем лучше формализованы правила синтаксиса языка, тем больше ошибок из общего количества может обнаружить компилятор и, соответственно, меньше ошибок будет обнаруживаться на следующих этапах. В связи с этим говорят о языках программирования с защищенным синтаксисом и с незащищенным синтаксисом. К первым, безусловно, можно отнести Pascal, имеющий очень простой и четко определенный синтаксис, хорошо проверяемый при компиляции программы, ко вторым – C/C++ со всеми их модификациями. Чего стоит хотя бы возможность выполнения присваивания в условном операторе в C/C+, например:

```
if(c=n) x=0;
```

В данном случае не проверятся равенство c и n, а выполняется присваивание с значения n, после чего результат операции сравнивается с нулем, если программист хотел выполнить не присваивание, а сравнение, то эта ошибка будет обнаружена только на этапе выполнения при получении результатов, отличающихся от ожидаемых.

Ошибки компоновки

Ошибки компоновки, как следует из названия, связаны с проблемами, обнаруженными при разрешении внешних ссылок. Например, предусмотрено обращение к подпрограмме другого модуля, а при объединении модулей данная подпрограмма не найдена или не стыкуются списки параметров. В большинстве случаев ошибки такого рода также удается быстро локализовать и устранить.

Ошибки выполнения

К самой непредсказуемой группе относятся ошибки выполнения. Прежде всего они могут иметь разную природу, и соответственно по-разному проявляться. Часть ошибок обнаруживается и документируется операционной системой. Выделяют четыре способа проявления таких ошибок:

- появление сообщения об ошибке, зафиксированной схемами контроля выполнения машинных команд, например, переполнении разрядной сетки, ситуации “деление на ноль”, нарушении адресации и т. п.;
- появление сообщения об ошибке, обнаруженной операционной системой, например, нарушении защиты памяти, попытке записи на устройства, защищенные от записи, отсутствии файла с заданным именем и т. п.;
- «зависание» компьютера, как простое, когда удается завершить программу без перезагрузки операционной системы, так и «тяжелое», когда для продолжения работы необходима перезагрузка;
- несовпадение полученных результатов с ожидаемыми.

Отметим, что, если **ошибки этапа выполнения** обнаруживает пользователь, то в двух первых случаях, получив соответствующее сообщение, пользователь в зависимости от своего характера, степени необходимости и опыта работы за компьютером:

либо попробует понять, что произошло, ища свою вину,

либо обратится за помощью,

либо постарается никогда больше не иметь дела с этим продуктом.

При «зависании» компьютера пользователь может даже не сразу понять, что происходит что-то не то, хотя его печальный опыт и заставляет волноваться каждый раз, когда компьютер не выдает быстрой реакции на введенную команду, что также целесообразно иметь в виду. Также опасны могут быть ситуации, при которых пользователь получает неправильные результаты и использует их в своей работе.

Ошибки выполнения

ВИД ОШИБОК	ПРИМЕР
1. Неправильная постановка задачи	Правильное решение неверно сформулированной задачи
2. Неверный алгоритм	Выбор алгоритма, приводящего к неточному или неэффективному решению задачи
3. Ошибки анализа	Неправильное программирование алгоритма
4. Семантические ошибки	Непонимание порядка выполнения команды
5. Синтаксические ошибки	Нарушение правил, определяемых языком программирования
6. Ошибки при выполнении операций	Отсутствие, указаний на ограничивающие условия вычислений (деление на нуль и т.д.)
7. Ошибки в данных	Неудачное определение возможного диапазона изменения данных
8. Ошибки в документации	Документация пользователя не соответствует действующему варианту программы

Причины ошибок выполнения очень разнообразны, а потому и локализация может оказаться крайне сложной.

Все возможные причины ошибок можно разделить на следующие группы:

- неверное определение исходных данных,
- логические ошибки,
- накопление погрешностей результатов вычислений (рисунок 10.2).

Рисунок 10.2 – Классификация ошибок этапа выполнения по возможным причинам

Неверное определение исходных данных происходит, если возникают любые ошибки при выполнении операций ввода-вывода:

- ошибки передачи,
- ошибки преобразования,
- ошибки перезаписи;
- ошибки данных.

Причем использование специальных технических средств и программирования с защитой от ошибок позволяет обнаружить и предотвратить только часть этих ошибок.

Логические ошибки имеют разную природу.

Ошибки, допущенных при **проектировании**:

- при выборе методов,
- при разработке алгоритмов;
- при определении структуры классов.

Ошибки непосредственно внесены при **кодировании** модуля:

– ошибки некорректного использования переменных, например, неудачный выбор типов данных, использование переменных до их инициализации, использование индексов, выходящих за границы определения массивов, нарушения соответствия типов данных при использовании явного или неявного переопределения типа данных, расположенных в памяти при использовании нетипизированных переменных, открытых массивов, объединений, динамической памяти, адресной арифметики и т. п.;
– ошибки вычислении, например, некорректные вычисления над неарифметическими переменными, некорректное использование целочисленной арифметики, некорректное преобразование типов данных в процессе вычислений, ошибки, связанные с незнанием приоритетов выполнения операций для арифметических и логических выражений, и т. п.;
– ошибки межмодульного интерфейса, например, игнорирование системных соглашений, нарушение типов и последовательности при передаче параметров, несоблюдение единства единиц измерения формальных и фактических параметров, нарушение области действия локальных и глобальных переменных;
– другие ошибки кодирования, например, неправильная реализация логики программы при кодировании, игнорирование особенностей или ограничений конкретного языка программирования.

Ошибки накопления погрешностей

Причины

Накопление погрешностей результатов числовых вычислений возникает:

- при некорректном отбрасывании дробных цифр чисел;
- некорректном использовании приближенных методов вычислений;
- игнорировании ограничения разрядной сетки представления вещественных чисел в ЭВМ и т. п.

Все указанные выше причины возникновения ошибок следует иметь в виду в процессе отладки.

Кроме того, **сложность отладки** увеличивается также вследствие влияния следующих факторов:

– опосредованного проявления ошибок;

– возможности взаимовлияния ошибок;

– возможности получения внешне одинаковых проявлений разных ошибок;

– отсутствия повторяемости проявлений некоторых ошибок от запуска к запуску (стохастические ошибки);

– возможности устранения внешних проявлений ошибок в исследуемой ситуации при внесении некоторых изменений в программу, например, при включении в программу диагностических фрагментов может аннулироваться или измениться внешнее проявление ошибок;

- написания отдельных частей программы разными программистами.

ОШИБКИ ОБЩЕГО ХАРАКТЕРА

– логические ошибки;

– ошибки в циклах;

– ошибки при работе с данными;

– ошибки в описании переменных;

– ошибки при работе с массивами;

– ошибки арифметических операций;

– ошибки в подпрограммах;

– ошибки ввода-вывода;

– ошибки логических операций.

Методы отладки программного обеспечения

Отладка программы в любом случае предполагает обдумывание и логическое осмысление всей имеющейся информации об ошибке. Большинство ошибок можно обнаружить по косвенным признакам посредством тщательного анализа текстов программ и результатов тестирования без получения дополнительной информации. При этом используют различные методы:

- ручного тестирования;
- индукции;
- дедукции;
- обратного прослеживания.

Метод ручного тестирования

Это - самый простой и естественный способ данной группы. При обнаружении ошибки необходимо выполнить тестируемую программу вручную, используя тестовый набор, при работе с которым была обнаружена ошибка.

Метод очень эффективен.

Но не применим для:

- больших программ;
- программ со сложными вычислениями;
- в тех случаях, когда ошибка связана с неверным представлением программиста о выполнении некоторых операций.

Данный метод часто используют как составную часть других методов отладки.

Метод индукции

Метод основан на тщательном анализе симптомов ошибки, которые могут проявляться как:

– неверные результаты вычислений;
– сообщение об ошибке.

Если компьютер просто «зависает», то фрагмент проявления ошибки вычисляют, исходя из последних полученных результатов и действий пользователя. Полученную таким образом информацию организуют и тщательно изучают, просматривая соответствующий фрагмент программы. В результате этих действий выдвигают гипотезы об ошибках, каждую из которых проверяют. Если гипотеза верна, то детализируют информацию об ошибке, иначе - выдвигают другую гипотезу. Последовательность выполнения отладки методом индукции показана на рисунке 10. 3 в виде схемы алгоритма.

Рисунок 10.3 - Схема процесса отладки методом индукции

Самый ответственный этап - **выявление симптомов ошибки**.

Организуя данные об ошибке, целесообразно записать все, что известно о ее проявлениях, причем фиксируют, как ситуации, в которых фрагмент с ошибкой выполняется нормально, так и ситуации, в которых ошибка проявляется. Если в результате изучения данных никаких гипотез не появляется, то необходима дополнительная информация об ошибке. Дополнительную информацию можно получить, например, в результате выполнения схожих тестов.

В процессе доказательства пытаются выяснить:

все ли проявления ошибки объясняет данная гипотеза, если не все, то либо гипотеза не верна, либо ошибок несколько.

Метод дедукции

По методу дедукции вначале формируют множество причин, которые могли бы вызвать данное проявление ошибки. Затем анализируя причины, исключают те, которые противоречат имеющимся данным. Если все причины исключены, то следует выполнить дополнительное тестирование исследуемого фрагмента. В противном случае наиболее вероятную гипотезу пытаются доказать. Если гипотеза объясняет полученные признаки ошибки, то ошибка найдена, иначе - проверяют следующую причину (рисунок 10.4).

Рисунок 10.4 - Схема процесса отладки методом дедукции

Метод обратного прослеживания

Для небольших программ эффективно применение метода обратного прослеживания.

Начинают с точки вывода неправильного результата. Для этой точки строится гипотеза о значениях основных переменных, которые могли бы привести к получению имеющегося результата. Далее, исходя из этой гипотезы, делают предложения о значениях переменных в предыдущей точке. Процесс продолжают, пока не обнаружат причину ошибки.

Методы и средства получения дополнительной информации

Для получения дополнительной информации об ошибке можно выполнить добавочные тесты или использовать специальные методы и средства:

- отладочный вывод;

- интегрированные средства отладки;

- независимые отладчики.

Средства отладки

Типы отладочных средств, применяемых при программировании:

1. Распечатывание содержимого памяти.
2. Отслеживание хода выполнения алгоритма.
3. Отслеживание обращений к переменным.
4. Отслеживание обращений к подпрограммам.
5. Проверка индексов.
6. Воспроизведение значений переменных.

МЕТОДЫ ОТЛАДКИ

1. Запуск программы из-под отладчика с пошаговой отладкой, просмотром состояний (переменных, стека, памяти, регистров и т. п.) в требуемых точках исполнения программы.
2. Логирования кода – вывод в файл (или консоль и т. п.) входных, выходных аргументов функций, промежуточных состояний (переменных, стека, памяти, передаваемых или получаемых каким-либо образом данных и т. п.) в процессе исполнения программы. При сложностях с воспроизведением сценария дефекта, логирование становится основной методикой отладки.
3. Анализ кода без исполнения программы – поиск причин возникновения дефекта с помощью анализа исходного кода программы, проблемного контента, конфигурации, состояния базы данных и т. п.
4. Анализ поведения системы или её части – изолирование проблемы, путём упрощения сценария (используя ручное или автоматическое тестирование). Аксиома звучит так: чем проще сценарий, тем проще отладить проблему. Если найти более простой сценарий, то отладка может упроститься.
5. Unit тестирование – выполнение автоматических unit test-ов в основном изолировано (т. е. в более простых сценариях) для функций (модулей, компонентов и т. п.), и таким образом автоматическое выявление проблемных участков кода. Unit тестирование в каком-то смысле одна из разновидностей отладки путём «анализа поведения системы».

6. Прототипирование – проверка функций (модулей, библиотек, и т. п.) в изоляции с помощью небольших примеров кода (прототипов). Прототип легче отлаживать, чем целевую систему. Если проблема воспроизводиться с помощью прототипа, отладка упрощается.

7. Отладка с помощью memory-dump-ов – разновидность логирования кода, только здесь логируется не просто некая структура памяти, а целиком вся память процесса и состояния регистров, когда возникает exception. По такому дампу памяти можно «раскрутить» состояние программы (стеков, очередей, переменных и т. п.), в котором она находилась во время паники. Достаточно много существует инструментальных средств для выполнения этой операции.

8. Отладка с помощью перехватов – в основном используется в случаях утечки ресурсов, разновидность логирования кода. Основная идея: перехват и логирование вызова функций выделения и освобождения ресурса, а также анализ состояния ресурсов (например, памяти) в требуемый момент времени или в нужной точке исполнения программы.

9. Профилирование кода (если необходима оптимизация производительности) – разновидность логирования кода, хотя часто выполняется с использованием специализированных инструментальных средств (профилировщиков). Этот метод отладки позволяет получить профиль исполнения программы – сколько и какая функция, строчка кода, модуль, и т. п. отнимают процессорного времени, и таким образом найти узкие места.

10. Выполнения программы (или её части) в другой среде (операционной системе, эмуляторе, симуляторе) – основная идея в том, что если нет инструментальных средств на целевой платформе, то можно спортировать код на другую платформу, где они есть. Также можно изначально писать кроссплатформенный код системы или

какой-то её части, и таким образом, при необходимости практически без портирования отлаживать код на другой платформе.

11. Отладка методом RPC (remote procedure call) – применимо в основном для встроенного программирования. Суть метода в возможности вызвать любую функцию (модуль и т. п.) передавая аргументы и получая результаты исполнения удалённо с одного хоста на другом вместо того, чтобы тратить время на компиляцию или обновление софта на удалённом хосте.
12. Отладка путём анализа документации, дизайна, требований или ограничений модулей (программных или аппаратных) – применимо в основном для сложных и крупных проектов. Основная идея понять по имеющейся документации допустимо ли поведение, происходящее в дефекте.
13. Отладка трансляцией кода – сложный алгоритм пишется или прототипируется на одном языке программирования) с наличием всех доступных инструментальных средств), а потом исходный код отлаженного алгоритма транслируется вручную или автоматически в другой язык программирования (целевой системы), для которого отсутствуют необходимые инструментальный средства.
14. Возможны и другие варианты, например, дисассемблерование с целью более низкоуровневого понимания, что происходит при выполнении программы. Т. е. анализируется некий промежуточный вариант кода, который в некоторых ситуациях легче отладить или понять.
15. Отладка разработкой интерпретатора - метод отладки, который используется, когда модуль требует частых изменений, а время построения приложения очень большое.

16. Для ускорения процесса и гибкости пишется небольшой интерпретатор кода с наличием управляющих конструкций. При наличии такого интерпретатора разработчик сравнительно не сложно создаёт скрипты, которые можно быстрее исправить и отладить.

Предотвращение ошибок

– Не применяйте непроверенных способов программирования.

– Старайтесь не использовать принцип умолчания.

– Никогда не допускайте зависимости работы программы от достоверности данных.

– Добивайтесь полноты логических решений.

– Стремитесь минимизировать число обращений к оператору ЭВМ.

Советы программисту

– Применяйте отладочный компилятор.

– Первым делом проверяйте программу за столом.

– Выполняйте эхо-проверку вводимых данных.

– Вводите средства отладки как можно раньше.

– Контролируйте правдоподобность вводимых данных.

– Используйте доступные для вас средства отладки.

– Делайте программу правильной с самого начала.

https://shareslide.ru/uncategorized/prezentatsiya-po-teme-otladka-programma-po

Отладочный вывод

Метод требует включения в программу дополнительного отладочного вывода в узловых точках. Узловыми считают точки алгоритма, в которых основные переменные программы меняют свои значения. Например, отладочный вывод следует предусмотреть до и после завершения цикла изменения некоторого массива значений. (Если отладочный вывод предусмотреть в цикле, то будет выведено слишком много значений, в которых, как правило, сложно разбираться.) При этом предполагается, что, выполнив анализ выведенных значений, программист уточнит момент, когда были получены неправильные значения, и сможет сделать вывод о причине ошибки.

Данный метод не очень эффективен и в настоящее время практически не используется, так как в сложных случаях в процессе отладки может потребоваться вывод большого количества - «трассы» значений многих переменных, которые выводятся при каждом изменении. Кроме того, внесение в программы дополнительных операторов может привести к изменению проявления ошибки, что нежелательно, хотя и позволяет сделать определенный вывод о ее природе.

Ошибки, исчезающие при включении в программу или удалению из нее каких-либо «безобидных» операторов, как правило, связаны с «затиранием» памяти. В результате добавления или удаления операторов область затирания может сместиться в другое место, и ошибка либо перестанет проявляться, либо будет проявляться по-другому.

Интегрированные средства отладки

Большинство современных сред программирования включают средства отладки, которые обеспечивают максимально эффективную отладку. Они позволяют:

- выполнять программу по шагам, причем как с заходом в подпрограммы, так и выполняя их целиком;
- предусматривать точки останова;
- выполнять программу до оператора, указанного курсором;
- отображать содержимое любых переменных при пошаговом выполнении;
- отслеживать поток сообщений и т. п.

Применять интегрированные средства в рамках среды достаточно просто. Используют разные приемы в зависимости от проявлений ошибки. Если получено сообщение об ошибке, то сначала уточняют, при выполнении какого оператора программы оно получено. Для этого устанавливают точку останова в начало фрагмента, в котором проявляется ошибка, и выполняют операторы в пошаговом режиме до проявления ошибки. Аналогично поступают при «зависании» компьютера.

Если получены неправильные результаты, то локализовать ошибку обычно существенно сложнее. В этом случае сначала определяют фрагмент, при выполнении которого получаются неправильные результаты. Для этого последовательно проверяют интересующие значения в узловых точках. Обнаружив значения, отличающиеся от ожидаемых, по шагам трассируют соответствующий фрагмент до выявления оператора, выполнение которого дает неверный результат.

Для уточнения природы ошибки возможен анализ машинных кодов, флагов и представления программы и значений памяти в 16-ричном виде. Причину ошибки определяют, используя один из рассмотренных методов. При этом для проверки гипотез также можно использовать интегрированные средства отладки.

Отладка с использованием независимых отладчиков. При отладке программ иногда используют специальные программы - отладчики, которые позволяют выполнить любой фрагмент программы в пошаговом режиме и проверить содержимое интересующих программиста переменных. Как правило такие отладчики позволяют отлаживать программу только в машинных командах, представленных в 16-ричном коде.

Общая методика отладки программного обеспечения

Суммируя все сказанное выше, можно предложить следующую методику отладки программного обеспечения, написанного на универсальных языках программирования для выполнения в операционных системах IVKDOSи Win32:

1 этап - изучение проявления ошибки - если выдано какое-либо сообщение или выданы неправильные или неполные результаты, то необходимо их изучить и попытаться понять, какая ошибка могла так проявиться. При этом используют индуктивные и дедуктивные методы отладки. В результате выдвигают версии о характере ошибки, которые необходимо проверить. Для этого можно применить методы и средства получения дополнительной информации об ошибке.

Если ошибка не найдена или система просто «зависла», переходят ко второму этапу.

2 этап - локализация ошибки - определение конкретного фрагмента, при выполнении которого произошло отклонение от предполагаемого вычислительного процесса. Локализация может выполняться:

путем отсечения частей программы, причем, если при отсечении некоторой части программы ошибка пропадает, то это может означать как то, что ошибка связана с этой частью, так и то, что внесенное изменение изменило проявление ошибки;

с использованием отладочных средств, позволяющих выполнить интересующих нас фрагмент программы в пошаговом режиме и получить дополнительную информацию о месте проявления и характере ошибки, например, уточнить содержимое указанных переменных.

При этом если были получены неправильные результаты, то в пошаговом режиме проверяют ключевые точки процесса формирования данного результата.

Как подчеркивалось выше, ошибка не обязательно допущена в том месте, где она проявилась. Если в конкретном случае это так, то переходят к следующему этапу.

3 этап - определение причины ошибки - изучение результатов второго этапа и формирование версий возможных причин ошибки. Эти версии необходимо проверить, возможно, используя отладочные средства для просмотра последовательности операторов или значений переменных.

4 этап - исправление ошибки - внесение соответствующих изменений во все операторы, совместное выполнение которых привело к ошибке.

5 этап - повторное тестирование - повторение всех тестов с начала, так как при исправлении обнаруженных ошибок часто вносят в программу новые.

Следует иметь в виду, что процесс отладки можно существенно упростить, если следовать основным рекомендациям структурного подхода к программированию:

- программу наращивать «сверху-вниз», от интерфейса к обрабатывающим подпрограммам, тестируя ее по ходу добавления подпрограмм;
- выводить пользователю вводимые им данные для контроля и проверять их на допустимость сразу после ввода;
- предусматривать вывод основных данных во всех узловых точках алгоритма (ветвлениях, вызовах подпрограмм).

Кроме того, следует более тщательно проверять фрагменты программного обеспечения, где уже были обнаружены ошибки, так как вероятность ошибок в этих местах по статистике выше. Это вызвано следующими причинами. Во-первых, ошибки чаще допускают в сложных местах или в тех случаях, если спецификации на реализуемые операции недостаточно проработаны. Во-вторых, ошибки могут быть результатом того, что программист устал, отвлекся или плохо себя чувствует. В-третьих, как уже упоминалось выше, ошибки часто появляются в результате исправления уже найденных ошибок.

Возвращаясь к рисунку 2, можно отметить, что проще всего обычно искать ошибки определения данных и ошибки накопления погрешностей: их причины локализованы в месте проявления. Логические ошибки искать существенно сложнее. Причем обнаружение ошибок проектирования требует возврата на предыдущие этапы и внесения соответствующих изменений в проект. Ошибки кодирования бывают как простые, например, использование неинициализированной переменной, так и очень сложные, например, ошибки, связанные с затиранием памяти. Затиранием памяти называют ошибки, приводящие к тому, что в результате записи некоторой информации не на свое место в оперативной памяти, затираются фрагменты данных или даже команд программы. Ошибки подобного рода обычно вызывают появление сообщения об ошибке. Поэтому определить фрагмент, при выполнении которого ошибка проявляется, несложно. А вот определение фрагмента программы, который затирает память - сложная задача, причем, чем длиннее программа, тем сложнее искать ошибки такого рода. Именно в этом случае часто прибегают к удалению из программы частей, хотя это и не обеспечивает однозначного ответа на вопрос, в какой из частей программы находится ошибка. Эффективнее попытаться определить операторы, которые записывают данные в память не по имени, а по адресу, и последовательно их проверить. Особое внимание при этом следует обращать на корректное распределение памяти под данные.

Контрольные вопросы

1. Какой процесс называют отладкой? В чем его сложность?

2. Назовите основные типы ошибок. Как они проявляются при выполнении программы?

3. Перечислите основные методы отладки. В чем заключается различие между ними? Возьмите любую программу, содержащую ошибки, и попробуйте найти ошибку, используя каждый из перечисленных методов. Какой метод для вас проще и естественней, и почему?

4. Какие средства получения дополнительной информации об ошибках вы знаете? Вспомните, какие ошибки вы искали дольше всего и почему. В каких случаях дополнительная информация позволяет найти ошибку?

Виды тестирования по степени автоматизации

Зачем тестировать руками, когда есть волшебная автоматизация? Бери да автоматизируй все подряд.

Но почему же автоматизированное тестирование до сих пор не вытеснило ручное? В чем плюсы и минусы каждого из них? Давайте разбираться.

Ручное тестирование

Ручное (мануальное) тестирование — это тестирование без помощи каких-либо программ, автоматизирующих работу.

Например, чтобы протестировать работу формы авторизации, мы сами заходим на сайт и вручную заполняем поля «Имя» и «Пароль». То есть мы выступаем в роли обычного пользователя продукта.

Напомню, что ручное тестирование строится на методах тестирования. Сюда относятся и техники тест-дизайна, и техники, основанные на опыте. Просто еще раз хочу обратить внимание, что тестирование — это заранее продуманная деятельность по сравнению фактического результата с ожидаемым, а не просто поиск ошибок «методом тыка».

Плюсы ручного тестирования:

- Пользовательский фидбек. Весь отчёт тестировщика может быть рассмотрен как обратная связь от потенциального пользователя.
- UI-фидбек. В наше время пользовательский интерфейс играет огромную роль и поэтому полностью протестировать его можно только вручную.
- Дешевизна. В краткосрочной перспективе ручное тестирование дешевле, чем инструменты автоматизированной проверки.
- Тестирование в реальном времени. Незначительные изменения могут быть исследованы сразу, без написания кода и его исполнения.
- Возможность исследовательского тестирования. Его целью является проверка разнообразных возможностей приложения. Важно, что используются не заранее составленные тест-кейсы, а придуманные на лету сценарии.

Минусы ручного тестирования:

- Человеческий фактор. Хотя UI и может быть протестирован только вручную, люди часто склонны к неэффективности. Некоторые ошибки могут остаться незамеченными.
- Трудоемкость повторного использование. Провести серию стандартных автоматических тестов проще, чем протестировать проект вручную после внесения даже небольших изменений.
- Невозможность проведения некоторых видов тестов, например, нагрузочного тестирования. Невозможно смоделировать большое количество пользователей вручную.

Автоматизированное тестирование

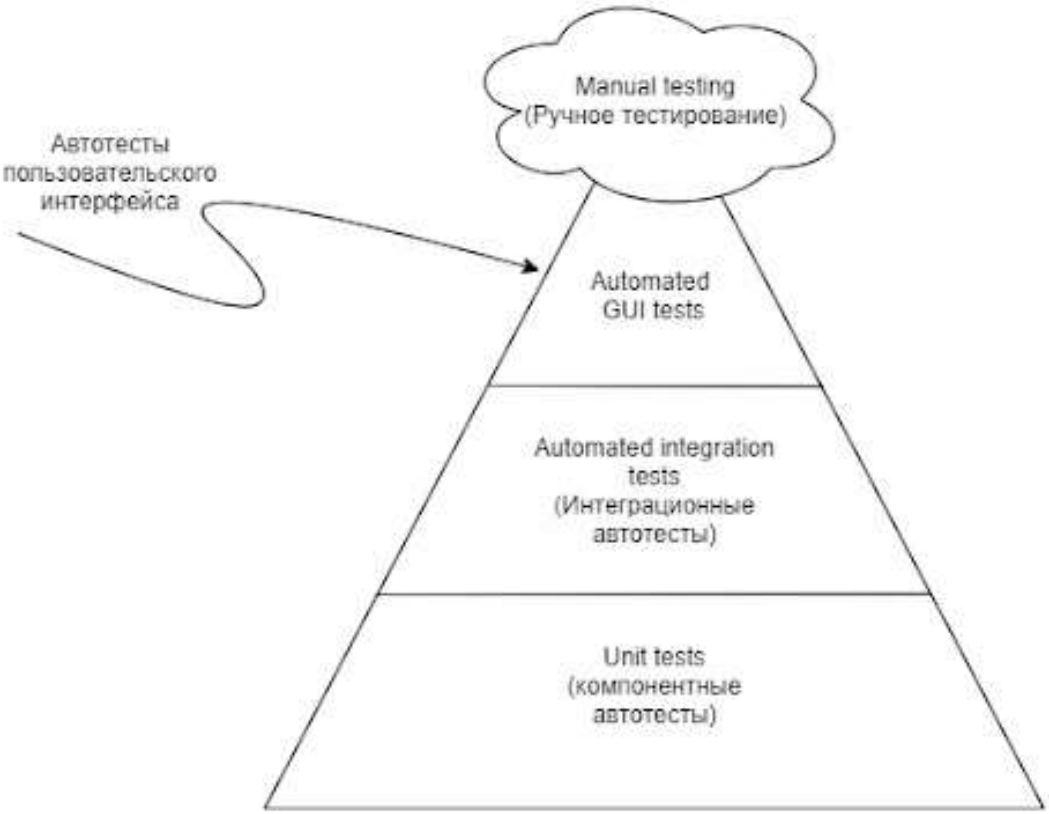
Автоматизированное тестирование предполагает использование специального программного обеспечения (помимо тестируемого) для контроля выполнения тестов и сравнения ожидаемого результата работы программы с фактическим.

Этот тип тестирования ***помогает автоматизировать часто повторяющиеся***, но необходимые для максимизации тестового покрытия, задачи.

Это отдельная дисциплина искусства тестирования. Значительная часть эффективности работы отдела тестирования зависит от того, какие задачи отданы для автоматизации и как эта автоматизация была осуществлена.

Подходы к автоматизации

Ниже представлена пирамида автоматизации Майка Кона, которая иллюстрирует эффективный подход к автоматизации тестирования.



Пирамида автоматизации

Ширина каждого уровня пирамиды показывает, сколько тестов должно быть на каждом уровне по сравнению с другими.

На нижнем уровне пирамиды находятся **компонентные** автотесты. Это тесты, которые проверяют на корректность отдельные модули исходного кода программы, наборы из одного или более программных модулей вместе с соответствующими управляющими данными, процедурами использования и обработки. Например, можно сделать тест на добавление одного товара в корзину. Или проверку одного вычисления на калькуляторе.

Средний уровень занимают **интеграционные** автотесты, которые верифицируют бизнес-поведение (но не через GUI). Такие тесты иногда называют API тестами. API — это интерфейс, который позволяет общаться напрямую с программой, минуя пользовательский интерфейс.

Например, вместо того, чтобы зайти на сайт, выбрать нужный товар и положить его в корзину, автотесты могут напрямую сказать сайту, отправив запрос “положи товар в корзину”.

Верхний уровень пирамиды — автотесты **пользовательского интерфейса**, которые непосредственно затрагивают пользовательский интерфейс. Например, проверяем отображение или изменение информации о сумме покупок в корзине через пользовательский интерфейс.

Что можно автоматизировать?

Давайте в качестве примера рассмотрим несколько программ автоматизации.

1. Программы для **помощи в черноящичном** тестировании. Например:

   — автоматическое создание аккаунтов пользователя;

   — запросы к базе данных и генерация файлов формата, утвержденного системой VISA, используя извлеченные данные;

   — генерация транзакции покупки в нашем магазине и т. д.

2. Программы для **регрессионного** тестирования. Это специальное ПО, созданное для буквального воспроизведения действий тестировщика.

   Например, согласно тест-кейсу, мы должны:

   - войти в систему,
   - выбрать товар,
   - положить его в корзину,
   - заплатить,
   - удостовериться, что баланс на кредитной карте уменьшается на сумму покупки.

Чтобы исполнить этот тест-кейс, мы должны запустить браузер, ввести имя пользователя и пароль, нажать на кнопку "Вход"... и, в конце концов, сравнить фактический и ожидаемый результаты. Теперь представьте себе, что некая программа делает те же самые действия за вас.

Такое ПО, как правило, поддерживает режим "Запись / Воспроизведение", т. е. когда мы нажимаем на кнопку "Запись" и начинаем кликать мышками и клацать клавишами клавиатуры, ПО записывает наши действия и, когда мы закончили, генерирует код. Этот код мы можем запустить с этим же ПО, и оно воспроизведет все наши клики и клацы, т. е. буквально будет водить курсором мышки, набирать текст и т. д.

3. Программы для тестирования **скорости и надежности**. Это программы, которые помогают проводить стрессовое и нагрузочное тестирование.

Плюсы и минуса автоматизации

Преимущества:

- Исключен «человеческий фактор» во время выполнения: тест-скрипт не допустит ошибки по неосторожности.
- Скорость выполнения выше возможностей человека.
- Автоматически формируемые и сохраняемые отчеты о результатах тестирования.
- Выполнение в фоне – во время выполнения тестов можно заниматься другими задачами или выполнять тест-скрипты в нерабочее время.
- Способность средств автоматизации выполнить тест-кейсы, в принципе непосильные для человека в силу своей сложности, скорости или иных факторов. Например, нагрузочное тестирование.

Недостатки:

- Необходим высококвалифицированный персонал в силу того факта, что автоматизация — это «проект внутри проекта» (со своими требованиями, планами, кодом и т. д.).
- Высокие затраты на сложные средства автоматизации, разработку и сопровождение кода тест-кейсов.
- Автоматизация требует более тщательного планирования и управления рисками, т. к. в противном случае проекту может быть нанесён серьёзный ущерб.
- Средств автоматизации крайне много, что усложняет проблему выбора того или иного средства и может повлечь за собой финансовые затраты (и риски), необходимость обучения персонала (или поиска специалистов).
- В случае ощутимого изменения требований, смены технологического домена, переработки интерфейсов (как пользовательских, так и программных) многие тест-кейсы становятся безнадёжно устаревшими и требуют создания заново.

Смешанное/полуавтоматизированное тестирование

Здесь ручной подход сочетается с автоматизированным. Например, с помощью программы создается новый аккаунт, а потом вручную генерируются транзакции покупки.

Почему ручное тестирование не вымерло?

Да, автоматизация значительно сокращает время на тестирование определенных фич. Плюс есть вещи, которые просто невозможно проверить вручную. Но не стоит забывать и о ее стоимости. Не все продукты финансово могут потянуть автоматизацию. Да и не всегда требуется что-то автоматизировать.

Также вручную можно протестировать практически любое приложение, в то время как автоматизировать стоит только стабильные системы. То есть мы не можем, по крайней мере на сегодняшний день, автоматизировать все проверки.

Автоматизированное тестирование используется главным образом для регрессии. Кроме того, некоторые виды тестирования, например, исследовательское тестирование, могут быть выполнены только вручную.

Автоматизация начинается с ручного тестирования. То есть для того, чтобы начать процесс автоматизации тестирования, нужно точно знать, что и как вы собираетесь делать. Идеальный автотест базируется на ручном тест-кейсе с должным уровнем детализации.

Ручное тестирование существует, так как невозможно автоматизировать все проверки и автоматизация не всегда финансово выгодна.

На реальных проектах часто используется комбинация ручного и автоматизированного тестирования. Уровень автоматизации зависит от многих факторов: тип проекта, особенности постановки производственных процессов в компании, финансовых возможностей и т. п.

Оба вида тестирования имеют как преимущества, так и недостатки. Комбинация обоих — хороший способ получить от тестирования максимальный результат.