«ҚОСТАНАЙ ЖОҒАРЫ ПОЛИТЕХНИКАЛЫҚ КОЛЛЕДЖІ» КМҚК

КГКП «КОСТАНАЙСКИЙ ПОЛИТЕХНИЧЕСКИЙ ВЫСШИЙ КОЛЛЕДЖ»

Учебно-методический комплекс

ПМ 4. ПРОВЕРКА РАБОТОСПОСОБНОСТИ РЕФАКТОРИНГА ПРОГРАММНОГО КОДА

Костанай, 2022

Лекция 1. Введение в тестирование.

1 Что такое тестирование?
2 Почему необходимо тестирование?
3 «7» принципов тестирования

Что такое тестирование?

Тестирование ПО - это способ оценить качество программного обеспечения и снизить риск сбоя программного обеспечения в работе. Процесс тестирования ПО включает в себя множество различных действий и выполнение тестов только одно из них.

Цели тестирования

- Оценить: требования, пользовательские истории, дизайн и код
- Проверить, были ли выполнены все указанные требования
- Проверить, завершен ли продукт и работает ли он так, как ожидают пользователи и другие заинтересованные стороны
- Повысить уверенность в уровне качества продукта
- Предотвратить дефекты
- Найти сбои и дефекты
- Предоставить заинтересованным сторонам достаточную информацию, позволяющую им принимать обоснованные решения, особенно в отношении уровня качества системы
- Снизить уровень риска неадекватного качества ПО (например, ранее необнаруженные сбои, возникающие в процессе работы)
- Соблюдать договорные, юридические или нормативные требования или стандарты и / или проверять соответствие объекта испытаний таким требованиям или стандартам

Цели тестирования могут различаться в зависимости от контекста (этапа) тестируемого компонента или системы, уровня тестирования и модели жизненного цикла разработки программного обеспечения.

Компонентное (модульное) тестирование

Нахождение max сбоев
Увеличение покрытия кода unit-тестами

Приемочное тестирование

Подтверждение, что система работает должным образом
Предоставление информации о рисках выпуска продукта

Тестирование и отладка

В 1978 году Glenford Myers разделил понятия debugging и testing.

• Во время тестирования мы выполняем тест, результат может показать сбои приложения, вызванные дефектами в ПО.

• Отладка - это деятельность в разработке, которая находит, анализирует и исправляет такие дефекты.

Почему необходимо тестирование?

Вклад тестирования. Тщательное тестирование может снизить риск отказов во время работы продукта.

Действие	Снижение риска
Проверка требований или пользовательской истории	Непроверяемого ф-ла, Неправильного ф-ла
«Близкое» общение с разработкой	Глобальных дефектов проектирования
«Близкое» общение с разработкой	Дефектов в коде

Проверка и валидация релиза	Дефектов в поставляемом продукте

QA, QC, тестирование

QA - обеспечивает правильность и предсказуемость процесса (техника управлением)
QC - предполагает контроль соблюдения требований, достижения должного уровня качества (метод проверки)

Тестирование - обеспечивает сбор статистических данных и внесение их в документы, созданные в рамках QC-процесса.

QA, QC, тестирование. Пример

QA - обеспечивает правильность и предсказуемость процесса
QC - предполагает контроль соблюдения требований, достижения должного уровня качества
Тестирование - обеспечивает сбор статистических данных и внесение их в документы, созданные в рамках QC-процесса.

QA, QC, тестирование. Пример

QA	QC
• Гарантирует, что вы делаете правильные вещи	• Гарантирует, что результаты того, что вы сделали, соответствуют вашим ожиданиям
• Определяет стандарты и методологии, которым необходимо следовать для удовлетворения требований заказчика	• Обеспечивает соблюдение стандартов при работе с продуктом
• Отвечает за полный жизненный цикл разработки программного обеспечения	• Отвечает за жизненный цикл тестирования программного обеспечения
• Не включает в себя выполнение программы	• Включает в себя выполнение программы

Error, defect, failures, bug

1843 – первое упоминание ошибки в аналитическом движке Чарльза Баббиджа.
1878 – Том Эдисон, первое слово «bug» в письме.

Error, defect, failures

Error (mistake) – логическая или другая ошибка, которая может привести к возникновению дефекта
Defect – различие между ожидаемым и фактическим результатом
Failure – сбой, к которому может привести дефект
Defect = (bug или issue или problem или incident или fault)

Дефекты, первопричины и следствия

Неправильные выплаты процентов из-за одной строки неправильного кода приводят к жалобам клиентов. Дефектный код был написан для пользовательской истории, которая была неоднозначной из-за неправильного понимания владельцем продукта, как рассчитывать проценты.
Первопричина - ?
Дефект - ?
Следствие - ?
Сбой (failure) - ?

7 принципов тестирования

1. Тестирование показывает наличие дефектов, а не их отсутствие
2. Исчерпывающее тестирование невозможно
3. Раннее тестирование экономит время и деньги
4. Кластеризация дефектов
5. Остерегайтесь парадокса пестицидов
6. Тестирование зависит от контекста
7. Отсутствие ошибок - заблуждение

Контрольные вопросы:

1. Что такое тестирование?
2. Цели тестирования
3. Дефекты, первопричины и следствия

Лекция 2. Методологии разработки программного обеспечения

Методология — это система принципов, а также совокупность идей, понятий, методов, способов и средств, определяющих стиль разработки программного обеспечения.

Адаптивные методологии

- нацелены на преодоление ожидаемой неполноты требований и их постоянного изменения;
- когда меняются требования, команда разработчиков тоже меняется;
- команда, участвующая в адаптивной разработке, с трудом может предсказать будущее проекта;
- существует точный план лишь на ближайшее время;
- более удаленные во времени планы существуют лишь как декларации о целях проекта, ожидаемых затратах и результатах.

Процесс – это последовательность операций, необходимых для достижения определенных целей.

Модели процессов создания и разработки ПО

Чтобы лучше разобраться в том, как тестирование соотносится с программированием и иными видами проектной деятельности, для начала рассмотрим самые основные — модели разработки ПО (как часть жизненного цикла ПО). При этом сразу подчеркнём, что разработка ПО является лишь частью жизненного цикла ПО, и здесь мы говорим именно о разработке. Моделей разработки ПО много, но, в общем случае, классическими можно считать **каскадную, v-образную, итерационную, инкрементальную, спиральную и гибкую.**

Знать и понимать модели разработки ПО необходимо затем, чтобы уже с первых дней работы понимать, что происходит вокруг, что, зачем и почему Вы делаете. Многие начинающие тестировщики отмечают, что ощущение бессмысленности происходящего посещает их, даже если текущие задания интересны. Чем полнее вы будете представлять картину происходящего на проекте, тем яснее Вам будет виден ваш собственный вклад в общее дело и смысл того, чем вы занимаетесь.

Ещё одна важная вещь, которую следует понимать, состоит в том, что никакая модель не является догмой или универсальным решением. Нет идеальной модели. Есть та, которая хуже или лучше подходит для конкретного проекта, конкретной команды, конкретных условий.

Каскадная (водопадная) модель сейчас представляет, скорее, исторический интерес, т.к. в современных проектах практически не применима. Она предполагает однократное выполнение каждой из фаз проекта, которые, в свою очередь, строго следуют друг за другом (Рис. 1.2). Очень упрощённо можно сказать, что, в рамках этой модели, в любой момент времени команде «видна» лишь предыдущая и следующая фаза. В реальной же разработке ПО приходится «видеть весь проект целиком» и возвращаться к предыдущим фазам, чтобы исправить недоработки или что-то уточнить.

Каскадная модель (waterfall)

Особенности каскадной модели:

— высокий уровень формализации процессов;

— большое количество документации;

— жесткая последовательность этапов жизненного цикла без возможности возврата на предыдущий этап.

Минусы:

• Waterfall-проект должен постоянно иметь актуальную документацию. Обязательная актуализация проектной документации. Избыточная документация.

• Очень не гибкая методология.

• Может создать ошибочное впечатление о работе над проектом (например, фраза «45% выполнено» не несёт за собой никакой полезной информации, а является всего лишь инструментов для менеджера проекта).

• У заказчика нет возможности ознакомиться с системой заранее и даже с «Пилотом» системы.

• У пользователя нет возможности привыкать к продукту постепенно.

• Все требования должны быть известны в начале жизненного цикла проекта.

• Возникает необходимость в жёстком управлении и регулярном контроле, иначе проект быстро выбьется из графиков.

• Отсутствует возможность учесть переделку, весь проект делается за один раз.

Плюсы:

• Высокая прозрачность разработки и фаз проекта.

• Чёткая последовательность.

• Стабильность требований.

• Строгий контроль менеджмента проекта.

• Облегчает работу по составлению плана проекта и сбора команды проекта.

• Хорошо определяет процедуру по контролю качества.

«Водоворот» или каскадная модель с промежуточным контролем

В этой модели предусмотрен промежуточный контроль за счет обратных связей. Но это достоинство порождает и недостатки. Затраты на реализацию проекта при таком подходе возрастают практически в 10 раз. Эта модель, как Вы уже поняли, является незначительной модификацией предыдущей и относится к первой группе.

При реальной работе, в соответствии с моделью, допускающей движение только в одну сторону, обычно возникают проблемы при обнаружении недоработок и ошибок, сделанных на ранних этапах. Но еще более тяжело иметь дело с изменениями окружения, в котором разрабатывается ПО (это могут быть изменения требований, смена подрядчиков, изменение политики разрабатывающей или эксплуатирующей организации, изменения отраслевых стандартов, появление конкурирующих продуктов и пр.).

Итеративная модель

Итеративные или инкрементальные модели (известно несколько таких моделей) предполагают разбиение создаваемой системы на набор кусков, которые разрабатываются с помощью нескольких последовательных проходов всех работ или их части.

Каскадная модель с возможностью возвращения на предшествующий шаг, при необходимости пересмотреть его результаты, становится итеративной. Итеративный процесс предполагает, что разные виды деятельности не привязаны намертво к определенным этапам разработки, а выполняются по мере необходимости, иногда повторяются, до тех пор, пока не будет получен нужный результат. Вместе с гибкостью и возможностью быстро реагировать на изменения, итеративные модели привносят дополнительные сложности в управление проектом и отслеживание его хода. При использовании итеративного подхода значительно сложнее становится адекватно оценить текущее состояние проекта и спланировать долгосрочное развитие событий, а также предсказать сроки и ресурсы, необходимые для обеспечения определенного качества результата.

Спиральная модель жизненного цикла программного обеспечения

Данная модель прекрасно сочетает в себе прототипирование и проектирование по стадиям. И из восходящей и нисходящей концепций в эту модель было взято все лучшее.

Преимущества модели:

1. Результат достигается в кратчайшие сроки.
2. Конкурентоспособность достаточно высокая.
3. При изменении требований не придется начинать все с «нуля».

Но у этой модели есть один существенный **недостаток: невозможность регламентирования стадий выполнения.**

V модель — разработка через тестирование

Данная модель имеет более приближенный к современным методам алгоритм, однако все еще имеет ряд недостатков. Является одной из основных практик экстремального программирования и предполагает регулярное тестирование продукта во время разработки.

V-модель обеспечивает поддержку в планировании и реализации проекта. В ходе проекта ставятся следующие задачи:

• **Минимизация рисков:** V-образная модель делает проект более прозрачным и повышает качество контроля проекта путём стандартизации промежуточных целей и описания соответствующих им результатов и ответственных лиц. Это позволяет выявлять отклонения и риски в проекте на ранних стадиях и улучшает качество управления проектов, уменьшая риски.

• **Повышение и гарантии качества:** V-Model —стандартизованная модель разработки, что позволяет добиться от проекта результатов желаемого качества. Промежуточные результаты могут быть проверены на ранних стадиях. Универсальное документирование облегчает читаемость, понятность и проверяемость.

• **Уменьшение общей стоимости проекта:** ресурсы на разработку, производство, управление и поддержку могут быть заранее просчитаны и проконтролированы. Получаемые результаты также универсальны и легко прогнозируются. Это уменьшает затраты на последующие стадии и проекты.

• **Повышение качества коммуникации между участниками проекта:** универсальное описание всех элементов и условий облегчает взаимопонимание всех участников проекта. Таким образом, уменьшаются неточности в понимании между пользователем, покупателем, поставщиком и разработчиком.

Модель на основе разработки прототипа

Данная модель основывается на разработке прототипов и прототипирования продукта и относится ко второй группе.

Прототипирование используется на ранних стадиях жизненного цикла программного обеспечения:

- Прояснить неясные требования (прототип UI).
- Выбрать одно из ряда концептуальных решений (реализация сценариев).
- Проанализировать осуществимость проекта.

Классификация прототипов:

- **Горизонтальные прототипы** — моделирует исключительно UI, не затрагивая логику обработки и базу данных.
- **Вертикальные прототипы** — проверка архитектурных решений.
- **Одноразовые прототипы** — для быстрой разработки.
- **Эволюционные прототипы** — первое приближение эволюционной системы.

Лекция 3. Место тестирования в процессе разработки программного обеспечения

Качество программного обеспечения (Software Quality) - это степень, в которой программное обеспечение обладает требуемой комбинацией свойств. [1061-1998 IEEE Standard for Software Quality Metrics Methodology].

Качество программного обеспечения (Software Quality) - это совокупность характеристик программного обеспечения, относящихся к его способности удовлетворять установленные и предполагаемые потребности. [ISO 8402:1994 Quality management and quality assurance].

Качество ПО

- Обеспечение качества (Quality Assurance - QA) - это совокупность мероприятий, охватывающих все технологические этапы разработки, выпуска и эксплуатации программного обеспечения (ПО)
- информационных систем, предпринимаемых на разных стадиях жизненного цикла ПО, для обеспечения требуемого уровня качества выпускаемого продукта.

- Контроль качества (Quality Control - QC) - это совокупность действий, проводимых над продуктом в процессе разработки, для получения информации о его актуальном состоянии в разрезах:
  - готовность продукта к выпуску,
  - соответствие зафиксированным требованиям,
  - соответствие заявленному уровню качества продукта.

- Тестирование программного обеспечения (Software Testing) - проверка соответствия между реальным и ожидаемым поведением программы, осуществляемая на конечном наборе тестов, выбранном определенным образом. [IEEE Guide to Software Engineering Body of Knowledge, SWEBOK, 2004]

- В более широком смысле, тестирование - это одна из техник контроля качества, включающая в себя активности:
  - по планированию работ (Test Management),
  - проектированию тестов (Test Design),
  - выполнению тестирования (Test Execution),
  - анализу полученных результатов (Test Analysis).

Классификация видов тестирования

По объекту тестирования	• Функциональное тестирование (functional testing)
	• Тестирование производительности (performance testing)
	  ◦ Нагрузочное тестирование (load testing)
	  ◦ Стресс-тестирование (stress testing)
	  ◦ Тестирование стабильности (stability / endurance / soak testing)
	• Юзабилити-тестирование (usability testing)
	• Тестирование интерфейса пользователя (UI testing)
	• Тестирование безопасности (security testing)
	• Тестирование локализации (localization testing)
	• Тестирование совместимости (compatibility testing)

По знанию системы	• Тестирование чёрного ящика (black box) • Тестирование белого ящика (white box) • Тестирование серого ящика (grey box)
По степени автоматизации	• Ручное тестирование (manual testing) • Автоматизированное тестирование (automated testing) • Полуавтоматизированное тестирование (semiautomated testing)

По степени изолированности компонентов	• Компонентное (модульное) тестирование (component/unit testing) • Интеграционное тестирование (integration testing) • Системное тестирование (system/end-to-end testing)

По времени проведения тестирования	• Альфа-тестирование (alpha testing)
	  • Тестирование при приёмке (smoke testing)
	  • Тестирование новой функциональности (new feature testing)
	  • Регрессионное тестирование (regression testing)
	  • Тестирование при сдаче (acceptance testing)
	• Бета-тестирование (beta testing)

По признаку позитивности сценариев	• Позитивное тестирование (positive testing) • Негативное тестирование (negative testing)
По степени подготовленности к тестированию	• Тестирование по документации (formal testing) • Тестирование ad hoc или интуитивное тестирование (ad hoc testing)

Верификация (Verification) - это процесс оценки системы или её компонентов с целью определения удовлетворяют ли результаты текущего этапа разработки условиям, сформированным в начале этого этапа [IEEE]. Т.е. выполняются ли наши цели, сроки, задачи по разработке проекта, определенные в начале текущей фазы.

Валидация (Validation) - это определение соответствия разрабатываемого ПО ожиданиям и потребностям пользователя, требованиям к системе [BS7925-1].

План Тестирования (Test Plan) - это документ, описывающий весь объем работ по тестированию:

- описание объекта,
- стратегии,
- расписания,
- критериев начала и окончания тестирования,
- необходимое в процессе работы оборудование,
- специальные знания,
- оценки рисков с вариантами их разрешения.

Мастер Тест План (Master Plan or Master Test Plan) - является более статичным в силу того, что содержит в себе высокоуровневую (High Level) информацию, которая не подвержена частому изменению в процессе тестирования и пересмотра требований.

Тест План (Test Plan) - детальный тест план, содержит более конкретную информацию по стратегии, видам тестировании, расписанию выполнения работ, является "живым" документом, который постоянно претерпевает изменения, отражающие реальное положение дел на проекте.

Лекция 4. Виды тестирования программного обеспечения

White/Black/Grey Box-тестирование

Для того, чтобы лучше понимать подходы к тестированию программного обеспечения, нужно, конечно же, знать, какие виды и типы тестирования в принципе бывают. Давайте начнем с рассмотрения основных типов тестирования, которые определяют высокоуровневую классификацию тестов.

Самым высоким уровнем в иерархии подходов к тестированию будет понятие типа, которое может охватывать сразу несколько смежных техник тестирования. То есть, одному типу тестирования может соответствовать несколько его видов. Рассмотрим, для начала, несколько типов тестирования, которые отличаются знанием внутреннего устройства объекта тестирования.

Black Box

Summary: Мы не знаем, как устроена тестируемая система.

Тестирование методом «черного ящика», также известное как тестирование, основанное на спецификации или тестирование поведения – техника тестирования, основанная на работе исключительно с внешними интерфейсами тестируемой системы.

Согласно ISTQB, тестирование черного ящика – это: тестирование, как функциональное, так и нефункциональное, не предполагающее знания внутреннего устройства компонента или системы; тест-дизайн, основанный на технике черного ящика – процедура написания или выбора тест-кейсов на основе анализа функциональной или нефункциональной спецификации компонента или системы без знания ее внутреннего устройства.

Почему именно «черный ящик»? Тестируемая программа для тестировщика – как черный непрозрачный ящик, содержания которого он не видит. Целью этой техники является поиск ошибок в таких категориях:

неправильно реализованные или недостающие функции;
ошибки интерфейса;
ошибки в структурах данных или организации доступа к внешним базам данных;
ошибки поведения или недостаточная производительности системы;

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.

Пример:

Тестировщик проводит тестирование веб-сайта, не зная особенностей его реализации, используя только предусмотренные разработчиком поля ввода и кнопки. Источник ожидаемого результата – спецификация.

Поскольку это тип тестирования, то он может включать и другие его виды. Тестирование черного ящика может быть как функциональным, так и нефункциональным. Функциональное тестирование предполагает проверку работы функций системы, а нефункциональное – общие характеристики нашей программы.

Техника черного ящика применима на всех уровнях тестирования (от модульного до приемочного), для которых существует спецификация. Например, при осуществлении системного или интеграционного тестирования, требования или функциональная спецификация будут основой для написания тест-кейсов.

Техники тест-дизайна, основанные на использовании черного ящика, включают:

- классы эквивалентности;
- анализ граничных значений;
- таблицы решений;
- диаграммы изменения состояния;
- тестирование всех пар.

Преимущества:

- тестирование производится с позиции конечного пользователя и может помочь обнаружить неточности и противоречия в спецификации;
- тестировщику нет необходимости знать языки программирования и углубляться в особенности реализации программы;
- тестирование может производиться специалистами, независимыми от отдела разработки, что помогает избежать предвзятого отношения;

- можно начинать писать тест-кейсы, как только готова спецификация.

Недостатки:

- тестируется только очень ограниченное количество путей выполнения программы;
- без четкой спецификации (а это скорее реальность на многих проектах) достаточно трудно составить эффективные тест-кейсы;
- некоторые тесты могут оказаться избыточными, если они уже были проведены разработчиком на уровне модульного тестирования.
- Противоположностью техники черного ящика является тестирование методом белого ящика, речь о котором пойдет ниже.

White Box

Summary: Нам известны все детали реализации тестируемой программы.

Тестирование методом белого ящика (также прозрачного, открытого, стеклянного ящика или же 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. Точно 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.

Согласно ISTQB: тестирование белого ящика – это:

- тестирование, основанное на анализе внутренней структуры компонента или системы;
- тест-дизайн, основанный на технике белого ящика – процедура написания или выбора тест-кейсов на основе анализа внутреннего устройства системы или компонента.
- Почему «белый ящик»? Тестируемая программа для тестировщика – прозрачный ящик, содержимое которого он прекрасно видит.

Пример:

Тестировщик, который, как правило, является программистом, изучает реализацию кода поля ввода на веб-странице, определяет все предусмотренные (как правильные, так и неправильные) и не предусмотренные пользовательские вводы и сравнивает фактический результат выполнения программы с ожидаемым. При этом ожидаемый результат определяется именно тем, как должен работать код программы.

Тестирование методом белого ящика похоже на работу механика, который изучает двигатель машины, чтобы понять, почему она не заводится.

Техника белого ящика применима на разных уровнях тестирования: от модульного до системного, но, главным образом, применяется именно для реализации модульного тестирования компонента его автором.

Преимущества:

- тестирование может производиться на ранних этапах: нет необходимости ждать создания пользовательского интерфейса;
- можно провести более тщательное тестирование с покрытием большого количества путей выполнения программы.

Недостатки:

- для выполнения тестирования белого ящика необходимо большое количество специальных знаний;
- при использовании автоматизации тестирования на этом уровне поддержка тестовых скриптов может оказаться достаточно накладной, если программа часто изменяется.

- Сравнение Black Box и White Box

Grey Box

Summary: Нам известны только некоторые особенности реализации тестируемой системы.

Тестирование методом серого ящика – метод тестирования программного обеспечения, который предполагает комбинацию White Box и Black Box подходов. То есть внутреннее устройство программы нам известно лишь частично. Предполагается, например, доступ ко 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.

Пример:

Тестировщик изучает код программы с тем, чтобы лучше понимать принципы ее работы и изучить возможные пути ее выполнения. Такое знание поможет написать тест-кейс, который наверняка будет проверять определенную функциональность.

Техника серого ящика применима на разных уровнях тестирования: от модульного до системного, но, главным образом, применяется на интеграционном уровне для проверки взаимодействия разных модулей программы.

Лекция 5. Причины и типы ошибок при тестировании программного обеспечения

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.

Отладка - это процесс локализации и исправления ошибок, обнаруженных при тестировании программного обеспечения. Локализацией называют процесс определения оператора программы, выполнение которого вызвало нарушение нормального вычислительного процесса. Доя 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.

Вцелом сложность отладки обусловлена следующими причинами:

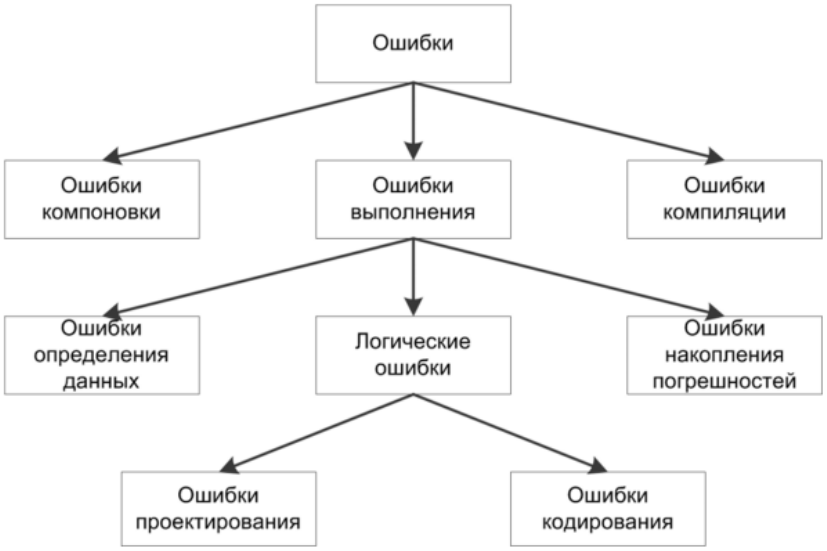
•требует от программиста глубоких знаний специфики управления используемыми техническими средствами, операционной системы, среды и языка программирования, реализуемых процессов, природы и специфики различных ошибок, методик отладки и соответствующих программных средств;

•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.

Всоответствии с этапом обработки, на котором проявляются ошибки, различают (рис. 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.

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 - Си со всеми его модификациями. Чего стоит хотя бы возможность выполнения присваивания в условном операторе в Си, 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.

Лекция 6. Методы и приемы формализации и алгоритмизации задач программного кода ПО

Общие методологические аспекты широкого класса компьютерных моделей позволяют исследовать механизм явления, протекающие в реальном объекте с большими или малыми скоростями, когда в натурных экспериментах с объектом трудно (или невозможно) проследить за изменениями, происходящими в течение короткого времени или когда получение достоверных результатов сопряжено с длительным экспериментом. При необходимости машинная модель “растягивает” или “сжимает” реальное время, так как машинное моделирование связано с понятием системного времени, отличного от реального. Кроме того, с помощью машинного моделирования можно обучать персонал АСОИУ принятию решений в управлении объектом.

Сущность машинного моделирования системы состоит в проведении численного эксперимента с моделью, которая представляет собой некоторый программный комплекс, описывающий формально и (или) алгоритмически поведение элементов системы *S* в процессе ее функционирования, т. е. в их взаимодействии друг с другом и внешней средой *E.*

Требованиями пользователя к модели *M* процесса функционирования системы *S* являются:

1. *Полнота модели* должна предоставлять пользователю возможность получения необходимого набора оценок характеристик системы с требуемой точностью и достоверностью.

2. *Гибкость модели* должна давать возможность воспроизведения различных ситуаций при варьировании структуры, алгоритмов и параметров системы.

3. Длительность разработки и реализации модели большой системы должна быть по возможности минимальной при учете ограничений на имеющиеся ресурсы.

4. Структура модели должна быть блочной, т. е. допускать возможность замены, добавления и исключения некоторых частей без переделки всей модели.

5. Информационное обеспечение должно предоставлять возможность эффективной работы модели с базой данных систем определенного класса.

6. Программные и технические средства должны обеспечивать эффективную (по быстродействию и памяти) машинную реализацию модели и удобное общение с ней пользователя.

7. Должно быть реализовано проведение целенаправленных (планируемых) машинных экспериментов с моделью системы с использованием аналитико-имитационного подхода при наличии ограниченных вычислительных ресурсов.

При машинном моделировании системы *S* характеристики процесса ее функционирования определяются на основе модели *M*, построенной исходя из имеющейся исходной информации об объекте моделирования.

При получении новой информации об объекте его модель пересматривается и уточняется с учетом новой информации, т.е. процесс моделирования, включая разработку и машинную реализацию модели, является *итерационным*. Этот итерационный процесс продолжается до тех пор, пока не будет получена модель *M*, которую можно считать адекватной в рамках решения поставленной задачи исследования и проектирования системы *S*.

Моделирование систем с помощью ЭВМ можно использовать в следующих случаях:

а) для исследования системы *S* до того, как она спроектирована, с целью определения чувствительности характеристики к изменениям структуры, алгоритмов и параметров объекта моделирования и внешней среды;

б) на этапе проектирования системы для анализа и синтеза различных вариантов системы и выбора такого варианта, который будет удовлетворять заданному критерию оценки эффективности системы при принятых ограничениях;

в) при эксплуатации системы, для получения информации, дополняющей результаты натурных испытаний (эксплуатации) реальной системы, и получения прогнозов развития системы во времени.

***Основные этапы моделирования больших систем*:**

1) построение концептуальной (описательной) модели системы и ее формализация;

2) алгоритмизация модели и ее компьютерная реализация;

3) получение и интерпретация результатов моделирования.

1. Построение концептуальной модели системы и ее формализация

На первом этапе машинного моделирования (построение концептуальной модели и ее формализация) формируется модель и строится ее формальная схема. *Основное назначение этого этапа* – переход от содержательного описания объекта к его логико-математической модели, другими словами процесс формализации. Моделирование на ЭВМ – наиболее эффективный метод оценки характеристик больших систем.

Модель должна быть адекватной, иначе нельзя получить достоверные результаты моделирования. Под ***адекватной моделью*** будем понимать модель, которая с определенной степенью приближения на уровне понимания моделируемой системы *S* разработчиком модели отражает процесс ее функционирования во внешней среде *E*.

Наиболее рационально строить модель функционирования системы по блочному принципу. Могут выделяться три автономные группы блоков такой модели:

1 группа: представляют собой имитатор воздействий внешней среды *E* на систему *S*;

2 группа: является собственно моделью процесса функционирования исследуемой системы ***S*;**

3 группа: служит для машинной реализации блоков двух первых групп, а также для фиксации и обработки результатов моделирования.

После перехода от описания моделируемой системы *S* к ее модели *M,* построенной по блочному принципу, строятся математические модели процессов, происходящих в различных блоках. Компьютерная модель представляет собой совокупность соотношений (например, уравнений, логических условий, операторов), определяющих характеристики процесса функционирования системы *S* в зависимости от структуры системы, алгоритмов поведения, параметров системы, воздействий внешней среды *E*, начальных условий и времени.

Формализации процесса функционирования любой системы *S* должно предшествовать изучение составляющих его явлений. Результатом является описание процесса, в котором изложены закономерности, характерные для исследуемого процесса, и постановку прикладной задачи.

Содержательное описание является исходным материалом для последующих этапов формализации. Для моделирования процесса функционирования системы на ЭВМ необходимо преобразовать математическую модель процесса в соответствующий моделирующий алгоритм и машинную программу.

Последовательность действий:

1. Постановка задачи машинного моделирования системы.
2. Анализ задачи моделирования системы.
3. Определение требований к исходной информации об объекте моделирования и организация ее сбора.
4. Выдвижение гипотез и принятие предположений.
5. Определение параметров и переменных модели.
6. Установление основного содержания модели.
7. Обоснование критериев оценки эффективности системы.
8. Определение процедур аппроксимации;
9. Описание концептуальной модели системы.
10. Проверка достоверности концептуальной модели.
11. Составление технической документации по первому этапу.

Алгоритмизация модели и ее компьютерная реализация

Вторым этапом моделирования является этап алгоритмизации модели и ее машинная реализация. Этот этап представляет собой этап, направленный на реализацию идей и математических схем в виде машинной модели *M* процесса функционирования систем *S*.

Принципы построения моделирующих алгоритмов

Процесс функционирования системы *S* можно рассматривать как последовательную смену ее состояний *z* в *k*-мерном пространстве. Задачей моделирования процесса функционирования исследуемой системы *S* является построение функций *z,* на основе которых можно провести вычисление интересующих характеристик процесса функционирования системы. Для этого необходимы соотношения, связывающие функции *z* с переменными, параметрами и временем, а также начальные условиями в момент времени $t = t_0$.

Рассмотрим функционирование некоторой *детерминированной системы SD*, в которой отсутствуют случайные факторы. Вектор состояний такой системы: *z*. Тогда состояние процесса в момент времени $t_0+j\Delta t$ может быть однозначно определено из соотношений математической модели по известным начальным условиям. Это позволяет строить моделирующий алгоритм процесса функционирования системы.

Для этого преобразуем соотношения модели *Z* к такому виду, чтобы сделать удобным вычисление по имеющимся значениям. Организуем счетчик системного времени, который в начальный момент времени показывает время t_0. В общем случае и начальные условия z_0 могут быть случайными, задаваемыми соответствующим распределением вероятностей. При этом структура моделирующего алгоритма для стохастических систем в основном остается прежней. Только вместо состояния *z* теперь необходимо вычислить распределение вероятностей для возможных состояний. Пусть счетчик системного времени показывает время t_0. В соответствии с заданным распределением вероятностей выбирается *z*. Далее, исходя из распределения, получается состояние z_i + и т.д., пока не будет построена одна из возможных реализаций случайного многомерного процесса) (t z_i в заданном интервале времени.

Рассмотренный принцип построения моделирующих алгоритмов называется ***принципом Δt.*** Это наиболее универсальный принцип, но с точки зрения затрат машинного времени он иногда оказывается неэкономичным.

При рассмотрении процессов функционирования некоторых систем можно обнаружить, что для них характерны два типа состояний:

1) ***особые состояния***, присущие процессу функционирования системы только в некоторые моменты времени;

2) ***регулярные состояния***, в которых процесс находится все остальное время.

Особые состояния характерны еще и тем обстоятельством, что функции состояний $Z(t)$ в эти моменты времени изменяются скачком, а между особыми состояниями изменение координат $Z(t)$ происходит плавно и непрерывно или не происходит совсем. Таким образом, следя при моделировании системы только за ее особыми состояниями в те моменты времени, когда эти состояния имеют место, можно получить информацию, необходимую для

построения функций $Z(t)$. Очевидно, для описанного типа систем могут быть построены моделирующие алгоритмы по "***принципу особых состояний***". Обозначим скачкообразное (релейное) изменение состояния z как δz, а "принцип особых состояний" – как ***принцип δz***.

Принцип δz отличается от принципа Δt тем, что шаг по времени в этом случае не постоянен, является случайной величиной и вычисляется в соответствии с информацией о предыдущем особом состоянии.

Принцип δz дает возможность для ряда систем существенно уменьшить затраты машинного времени на реализацию моделирующих алгоритмов.

2.2. Формы представления моделирующих алгоритмов

Удобной формой представления логической структуры моделей процессов функционирования систем и машинных программ является *схема*. На различных этапах моделирования составляются следующие схемы моделирующих алгоритмов и программ:

Обобщенная (укрупненная) схема моделирующего алгоритма задает общий порядок действий при моделировании системы без каких-либо уточняющих деталей.

Детальная схема моделирующего алгоритма содержит уточнения, отсутствующие в обобщенной схеме.

Логическая схема моделирующего алгоритма представляет собой логическую структуру модели процесса функционирования систем S.

Схема программы отображает порядок программной реализации моделирующего алгоритма с использованием конкретного математического обеспечения. Схема программы представляет собой интерпретацию логической схемы моделирующего алгоритма разработчиком программы на базе конкретного алгоритмического языка.

Последовательность действий:

1. Построение логической схемы модели.
2. Получение математических соотношении.
3. Проверка достоверности модели системы.
4. Выбор инструментальных средств для моделирования.
5. Составление плана выполнения работ по программированию.
6. Спецификация и построение схемы программы.
7. Верификация и проверка достоверности схемы программы.
8. Проведение программирования модели.
9. Проверка достоверности программы.
10. Составление технической документации по второму этапу.

Получение и интерпретация результатов моделирования

На третьем этапе компьютер используется для проведения рабочих расчетов по готовой программе. Результаты этих расчетов позволяют проанализировать и сделать выводы по характеристикам процесса функционирования исследуемой системы.

Последовательность действий:

1. Планирование машинного эксперимента с моделью системы.
2. Определение требований к вычислительным средствам.
3. Проведение рабочих расчетов.
4. Анализ результатов моделирования системы.
5. Представления результатов моделирования.
6. Интерпретация результатов моделирования.
7. Подведение итогов моделирования и выдача рекомендаций.
8. Составление технической документации.

Лекция 8. Воспроизведение ошибок тестирования приложений

Если ваш отчет об ошибках (баг-репорт) составлен правильно, то шансы на быстрое исправление этих багов - выше. Таким образом, исправление ошибки зависит от того, насколько качественно вы о ней сообщите. Составление отчетов об ошибках - не что иное, как навык, и сейчас мы рассмотрим, как его сформировать.

«Смысл написания отчета о проблемах (баг-репорта) состоит в том, чтобы исправить эти проблемы» - Cem Kaner. Если тестировщик не сообщает об ошибке правильно, программист, скорее всего, отклонит эту ошибку, заявив, что она невоспроизводима.

Это может повредить рабочему настрою тестировщиков, затронуть их профессиональную гордость, их эго.

Каковы качества хорошего баг-репорта в разработке программного обеспечения?

Любой может написать баг-репорт. Но не каждый может написать эффективный бар-репорт.

Вы должны уметь хорошо различать баг-репорт среднего качества и хороший баг-репорт. Как отличить хороший и плохой баг-репорты? Это очень просто, примените следующие характеристики и методы, чтобы качественно сообщить об ошибке.

Характеристики и методы включают в себя:

1) Наличие четко определенного номера ошибки:

Всегда присваивайте уникальный номер каждому сообщению об ошибке. Это, в свою очередь, поможет вам четко идентифицировать запись об ошибке. Если вы используете какой-либо инструмент автоматического формирования баг-репортов, то этот уникальный номер будет генерироваться автоматически каждый раз, когда вы делаете отчет. Запишите номер и краткое описание каждой ошибки, о которой вы сообщили.

2) Воспроизводимость:

Если найденная вами ошибка не воспроизводима, то она никогда не будет исправлена.

Вы должны четко указать шаги для воспроизведения ошибки. Не принимайте и не пропускайте ни одного шага воспроизведения. Ошибку, которая описана шаг за шагом, легко воспроизвести и исправить.

3) Будьте конкретны:

Не пишите очерк о проблеме.

Пишите конкретно и по существу. Попытайтесь описать обнаруженную проблему минимальным количеством слов и максимально эффективным способом. Не объединяйте описания нескольких багов в одном отчете, даже если они кажутся похожими. Напишите разные отчеты для каждой проблемы.

Эффективный баг-репортинг

Отчеты об ошибках являются важным аспектом тестирования программного обеспечения. Эффективный баг-репорт хорошо понимается командой разработчиков и позволяет избежать путаницы или недопонимания.

Хороший отчет об ошибке должен быть четким и кратким, без каких-либо пропущенных ключевых моментов. Любое отсутствие ясности ведет к недопониманию и замедляет процесс разработки. Описание дефектов и составление отчетов - одна из самых важных, но часто игнорируемых областей в жизненном цикле тестирования.

Правильно составленный текст отчета про найденный баг очень важен для регистрации ошибки. Один из важных моментов, которые должен иметь в виду тестер, — это не использовать командный тон в отчете. Такой тон нарушает моральное состояние коллектива и создает нездоровые рабочие отношения. Используйте нейтральный тон.

Не думайте, что если разработчик допустил ошибку, то вы можете использовать грубые слова. Прежде чем сообщать, не менее важно проверить, был ли уже баг-репорт по этой ошибке ранее или нет.

Дубликаты ошибок — это постоянная проблема в цикле тестирования. Проверяйте весь список обнаруженных багов. Иногда разработчики могут знать о наличествующей проблеме и игнорировать ее в будущем выпуске. Используйте специальные инструменты, такие как Bugzilla, который автоматически ищет дубликаты ошибок. Тем не менее, лучше всего дополнительно вручную искать дубликаты ошибок.

Четко указывайте информацию об ошибке: **«Как?»** и **«Где?»**. Отчет должен ясно показывать, как был выполнен тест и где именно произошел дефект. Читатель отчета должен легко воспроизвести ошибку и найти ее.

Имейте в виду, что **цель написания баг-репорта** - дать разработчику возможность визуализировать проблему. Он/она должен четко понимать суть дефекта, прочитав отчет об

ошибке. Не забудьте предоставить всю необходимую информацию, которую ищет разработчик.

Кроме того, имейте в виду, что отчет об ошибках будет сохранен для будущего использования и должен быть хорошо написан и содержать необходимую информацию. **Используйте содержательные предложения и простые слова**, чтобы описать найденные ошибки. Не используйте запутанные утверждения, которые тратят время читателя.

Сообщайте о каждой ошибке как о отдельной проблеме. В случае описания нескольких багов в одном отчете, вы не сможете закрыть его, пока все проблемы не будут решены.

Следовательно, **лучше всего разбить большие проблемы на отдельные баги**. Это гарантирует, что каждая ошибка может быть обработана отдельно. Хорошо написанный баг-репорт помогает разработчику воспроизвести ошибку на своем терминале. Это помогает им также правильно диагностировать проблему.

Как сообщить об ошибке?

Используйте следующий простой шаблон баг-репорта:

Это простая форма баг-репорта. Его содержание может варьироваться в зависимости от используемого вами инструмента отчетов об ошибках. Если вы пишете баг-репорт вручную, то необходимо упомянуть некоторые поля, например номер ошибки, который должен быть назначен вручную.

Составитель отчета: Ваше имя и адрес электронной почты.

Продукт: В каком продукте вы нашли эту ошибку.

Версия: Версия продукта с ошибкой, если таковая имеется.

Компонент: Основные подмодули продукта.

Платформа:

Укажите аппаратную платформу, на которой вы обнаружили эту ошибку. Различные платформы, такие как «ПК», «MAC», «HP», «Sun» и т. д.

Операционная система:

Укажите все операционные системы, в которых вы обнаружили ошибку. Операционные системы, такие как Windows, Linux, Unix, SunOS, Mac OS. Упомяните разные версии ОС, такие как Windows NT, Windows 2000, Windows XP и т. д., если это применимо.

Приоритет:

Когда следует исправлять ошибку? Приоритет обычно устанавливается от P1 до P5. P1 следует понимать, как «исправить ошибку с наивысшим приоритетом» и P5 - «исправить, если позволяет время».

Серьезность ошибки:

Описывает влияние ошибки.

Типы Серьезности ошибки:

- Блокировщик (Blocker): дальнейшая работа по тестированию невозможна.
- Критическая (Critical): сбой приложения, потеря данных.
- Major: серьезная потеря функциональности.
- Minor: незначительная потеря функциональности.
- Незначительная (Trivial): некоторые улучшения пользовательского интерфейса.
- Улучшение (Enhancement): запрос новой функции или некоторого улучшения существующей.

Статус ошибки:

Когда вы регистрируете ошибку в любой системе отслеживания ошибок, то по умолчанию статус ошибки будет «Новый».

Позднее ошибка проходит через различные этапы, такие как «Исправлено», «Проверено», «Повторно открыто», «Не исправлено» и т. д.

Назначить разработчику:

Если вы знаете, какой разработчик отвечает за тот конкретный модуль, в котором произошла ошибка, вы можете указать адрес электронной почты этого разработчика. В противном случае оставьте это поле пустым, так как это присвоит полю авторства ошибки значение владельца модуля, если менеджер не назначит ошибку разработчику.

Краткое резюме:

Добавьте краткое описание ошибки. Ориентируйтесь на 60 слов или меньше. Убедитесь, что составленное резюме отражает проблему и место, где она находится.

Описание:

Подробное описание ошибки.

Используйте следующие поля для поля описания:

- Воспроизводимые шаги: ясно упомяните шаги для воспроизведения ошибки.
- Ожидаемый результат: как приложение должно вести себя на вышеуказанных этапах.
- Фактический результат: каков фактический результат выполнения вышеупомянутых шагов, то есть поведение ошибки.

Это важные шаги в отчете об ошибках. Вы также можете добавить «Тип отчета» как еще одно поле, которое будет описывать тип ошибки.

Типы отчетов включают в себя:

1)Ошибка в коде
2)Ошибка проектирования
3)Новое предложение
4)Проблема с документацией
5) Аппаратная проблема

Важные фичи в вашем отчете об ошибках

Рассмотрим несколько составляющих отчета о найденном баге

Ниже приведены важные элементы баг-репорта:

1) Номер ошибки/идентификатор:

Номер ошибки или идентификационный номер (например, xyz007) значительно упрощает составление баг-репорта и поиск места ошибки. Разработчик может легко проверить, исправлена ли конкретная ошибка или нет. Это делает весь процесс тестирования и повторного тестирования более плавным и легким.

2) Наименование ошибки:

Заголовок ошибки читается чаще, чем любая другая часть баг-репорта. Стоит указать в нем всё о том, что входит в баг.

Название ошибки должно быть достаточно осмысленным, чтобы читатель мог его понять. Четкий заголовок ошибки облегчает понимание, и читатель легко сможет проверить, было ли сообщение об ошибке ранее и была ли она исправлена.

3) Приоритет:

В зависимости от серьезности ошибки, для нее может быть установлен приоритет. Ошибка может быть Blocker, Critical, Major, Minor, Trivial или предложением по улучшению функционала. Приоритет ошибки от P1 до P5 может быть задан так, чтобы важные из них просматривались первыми.

4) Платформа / Среда:

Указание конфигурации ОС и браузера необходимо для большей точности в баг-репорте. Это лучший способ сообщить о том, как можно воспроизвести ошибку. Без точной платформы или среды приложение может вести себя по-другому, и ошибка на стороне тестировщика может не повторяться на стороне разработчика. Поэтому лучше четко указать среду, в которой была обнаружена ошибка.

5) Описание:

Правильное описание ошибки помогает разработчику понять ошибку. Оно описывает возникшую проблему. Плохое описание создаст путаницу и потратит время разработчиков и тестеров.

Необходимо четко сообщить об эффекте в описании. Всегда полезно использовать полные предложения. Рекомендуется описывать каждую проблему отдельно, а не разрушать их полностью. Не используйте такие термины, как «я думаю» или «я считаю».

6) Шаги для воспроизведения ошибки:

Хороший отчет об ошибке должен четко указывать шаги для воспроизведения. Шаги должны включать действия, которые вызывают ошибку. Не делайте общих заявлений. Будьте конкретны в следующих шагах.

Хороший пример правильно написанной пошаговой процедуры приведен ниже:

Последовательность шагов:

- Выберите продукт wer05.
- Нажмите на «Добавить в корзину».
- Нажмите «Удалить», чтобы удалить продукт из корзины.

7) Ожидаемый и фактический результат:

Описание ошибки будет неполным без указания ожидаемых и фактических результатов. Необходимо обрисовать в общих чертах, каков результат теста и что ожидал бы пользователь в случае корректной работы программы. Читатель отчета должен знать, какой результат теста будет корректным. Ясно упомяните, что произошло во время теста и каков был результат.

8) Скриншот:

Одна картинка стоит тысячи слов. Сделайте скриншот с примером сбоя с соответствующими выделениями, чтобы указать дефект. Выделите неожиданные сообщения об ошибках светло-красным цветом. Это привлекает внимание к необходимой области.

Некоторые дополнительные советы, для написания хорошего баг-репорта

Ниже приведены некоторые дополнительные советы, чтобы написать хороший отчет об ошибке:

1) Немедленно сообщите о проблеме:

Если вы обнаружите какую-либо ошибку во время тестирования, не нужно ждать, чтобы написать подробный отчет об ошибке позже. Вместо этого напишите отчет об ошибке немедленно. Это обеспечит хорошее качество отчета и воспроизводимость шагов получения ошибках. Если вы решите написать отчет об ошибке позже, есть большие шансы пропустить важные детали в баг-репорте.

2) Воспроизведите ошибку три раза перед написанием баг-репорта:

Ваш баг должен быть воспроизводимым. Убедитесь, что ваши шаги достаточно четкие, чтобы воспроизвести ошибку без какой-либо двусмысленности. Если ваша ошибка не воспроизводима каждый раз, вы все равно можете подать ошибку, указав периодическую природу бага.

3) Протестируйте эту же ошибку на других похожих модулях:

Иногда разработчик использует один и тот же код для разных похожих модулей. Таким образом, вероятность того, что ошибка в одном модуле возникнет и в других подобных модулях, выше. Вы даже можете попытаться найти более серьезную версию найденной ошибки.

4) Составьте хорошее резюме ошибки:

Краткое описание ошибки поможет разработчикам быстро проанализировать природу ошибки. Низкое качество отчета излишне увеличит время разработки и тестирования. Правильно взаимодействуйте с вашим баг-репортом. Имейте в виду, что сводка об ошибках используется в качестве справочной информации для поиска ошибки в инвентаре ошибок.

5) Прочитайте несколько раз отчет об ошибке, прежде чем нажать кнопку «Отправить»:

Прочитайте все предложения, формулировки и шаги, которые используются в баг-репорте. Посмотрите, не создает ли какое-либо предложение двусмысленность, которая может привести к неправильной интерпретации. Следует избегать вводящих в заблуждение слов или предложений, чтобы составить четкое сообщение об ошибке.

Мы рассмотрели некоторые особенности составления отчета про найденный баг. Нет сомнений, что ваш баг-репорт должен быть качественным документом.

Сосредоточьтесь на написании хороших отчетов об ошибках и потратьте некоторое время на выполнение этой задачи, потому что именно качественный баг-репорт является основной точкой связи между тестером, разработчиком и менеджером. Менеджеры со своей

стороны должны объяснить своей команде, что составление хорошего отчета об ошибках является основной обязанностью любого тестировщика.

Ваши усилия по написанию хорошего отчета об ошибках не только сохранят ресурсы компании, но и создадут хорошие отношения между вами и разработчиками. Для лучшей производительности команды стремитесь написать лучший отчет об ошибках.

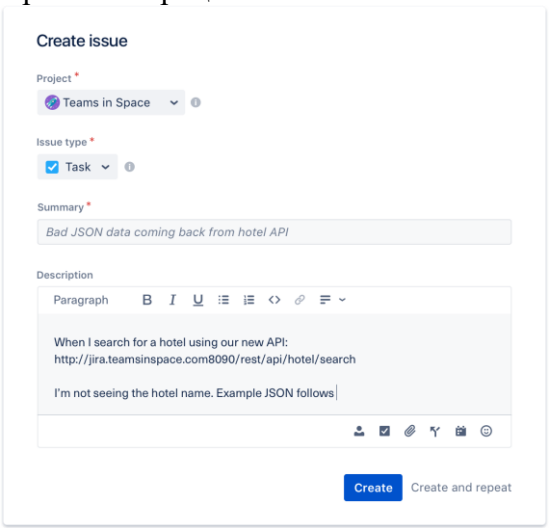
Лекция 9. Фиксировать ошибки в системе отслеживания ошибок

Что представляет собой инструмент отслеживания ошибок и задач?

Инструменты для отслеживания багов и задач помогают командам разработчиков выявлять, фиксировать и отслеживать баги в создаваемом ПО. Важно обеспечить каждому участнику команды возможность находить и фиксировать баги. Но еще важнее своевременно назначать задачи по их устранению подходящему сотруднику. Правильный инструмент для отслеживания багов и задач обеспечивает команде единое представление всех рабочих задач бэклога независимо от того, баг это или задание по разработке новой функции. Единый источник достоверной информации по всем типам задач помогает командам расставлять приоритеты с ориентиром на стратегические цели и непрерывно работать над поставкой ценности клиентам.

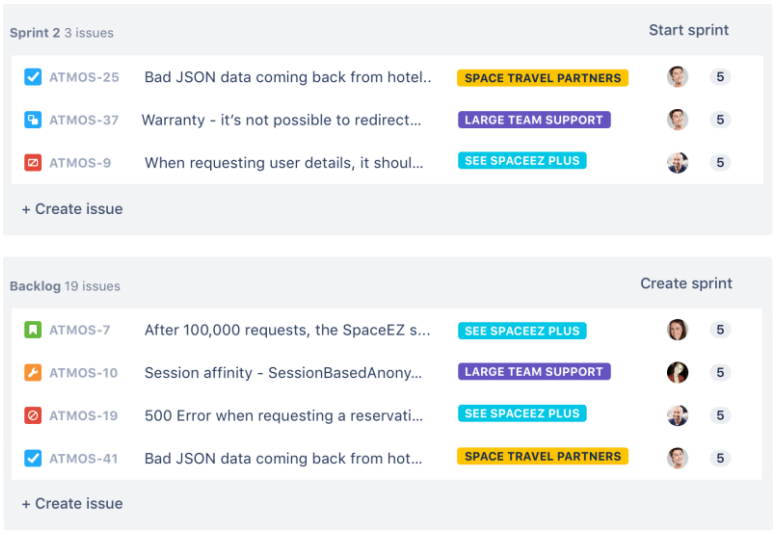
Выявляйте и отслеживайте баги ПО

Выявляйте баги в любой части проекта разработки ПО с помощью Jira Software. Как только баг найден, создайте задачу и добавьте к ней все необходимые подробности, например описания, уровень важности, снимки экрана, версию и т. д. Задачи могут касаться чего угодно: багов ПО, заданий по проекту, оставленных запросов и т. д. Для каждого типа задач можно настроить специальный рабочий процесс.



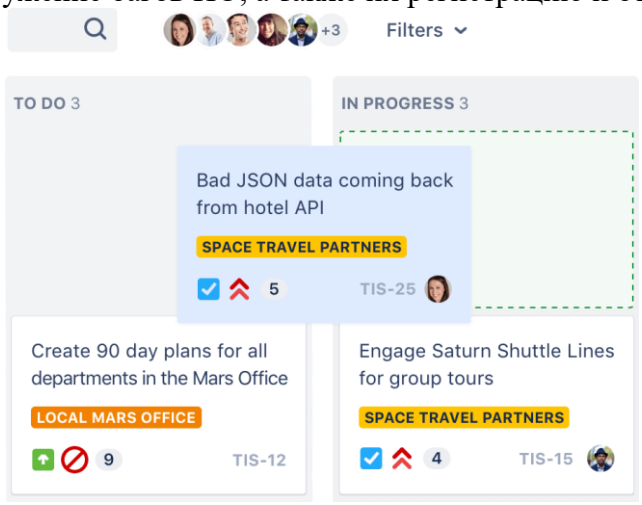
Легко назначайте задачи и расставляйте приоритеты

После выявления багов можно установить для каждой из них соответствующий приоритет, оценив важность бага, срочность его устранения, а также ресурсы команды. Назначить ответственного за устранение бага можно парой нажатий, а для расстановки приоритетов достаточно перетащить баги в бэклог команды или в столбец To do (К выполнению). С единым источником достоверной информации вы можете передавать нужные сведения всем участникам процесса и следить за тем, чтобы команда работала над поставленными задачами в порядке приоритета.



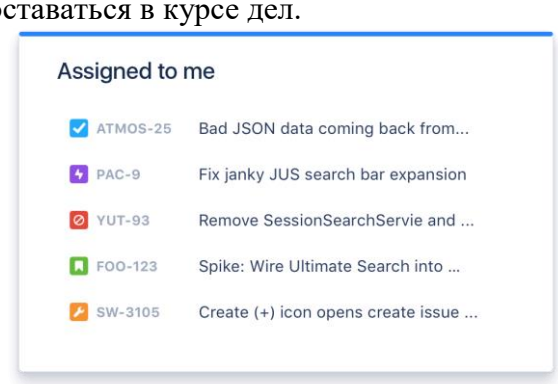
Отслеживайте баги от бэклога до устранения

Всегда оставайтесь в курсе событий, отслеживая баги и задачи на протяжении всего рабочего процесса команды. В Jira Software есть мощное ядро рабочего процесса, возможности планирования, расширенный поиск и система отчетности. Эти функции призваны упростить обнаружение багов ПО, а также их регистрацию и отслеживание.



Оставайтесь в курсе дел благодаря уведомлениям

Обеспечьте оповещение нужных людей в правильное время. Новые баги можно мгновенно направить нужному участнику команды. Уведомления в Jira доступны не только в форме @упоминаний. Можно также настроить автоматическое информирование сотрудников при изменении статуса задачи. Настройте проект так, чтобы при изменении ситуации решение Jira Software помогало всем оставаться в курсе дел.



Легкий путь от бэклога до релиза

Как только баги выявлены и в бэклоге расставлены приоритеты, разработчики ПО могут создать новые ветки в системе управления исходным кодом, такой как Bitbucket, и начать работу над устранением бага прямо из заявки Jira Software. По мере того как работа

над багом движется к завершению и развертыванию кода, Jira Software автоматически обновляет заявку, отражая выполненные запросы pull, слияния, сборки и другие операции. Так у всех участников команды появляется быстрый доступ к актуальной информации.

Рекомендации по отслеживанию багов в Jira Software

Вооружите команду актуальной информацией. Убедитесь, что каждый баг хорошо задокументирован, чтобы у разработчиков были нужные сведения для его воспроизведения и исправления. Создайте в Jira пользовательские поля, чтобы быстрее выявлять основные детали.

Быстро определяйте исполнителей и приоритеты. Назначайте баги специалистам в Jira с учетом приоритета и отправляйте соответствующие уведомления с помощью автоматизации.

Контролируйте своевременность устранения багов. Создавайте и настраивайте специальные рабочие процессы, чтобы команда могла сосредоточиться на эффективном управлении багами и быстром устранении.

Выведите процесс устранения багов на новый уровень. Сократите количество заданий, выполняемых вручную, благодаря функции оповещения наблюдателей об исправлениях, новых релизах и прочем через систему Jira Automation.

Выполните интеграцию Jira с инструментами разработки. Экономьте время, передавая данные непосредственно техническим специалистам, а также наблюдайте за конвейером разработки прямо из Jira.

Лекция 10. Тестовая документация. Чек-листы, тест-кейсы

Создание тестовой документации является вторым этапом жизненного цикла ПО.

Тестовая документация включает в себя:

- тест план;
- тестовая стратегия;
- чек-лист;
- тестовый сценарий;
- тестовый комплект;
- пользовательская история (User Story);
- отчет о дефекте.

Тест план (Test Plan) - это документ, описывающий весь объем работ по тестированию, начиная с описания объекта, стратегии, расписания, критериев начала и окончания тестирования, до необходимого в процессе работы оборудования, специальных знаний, а также оценки рисков с вариантами их разрешения.

Хороший тест план должен описывать следующее:

1. Что надо тестировать? Описание объекта тестирования: системы, приложения, оборудования.
2. Что будете тестировать? Список функций и описание тестируемой системы и её компонент в отдельности.
3. Как будете тестировать? Стратегия тестирования, а именно: виды тестирования и их применение по отношению к объекту тестирования.
4. Когда будете тестировать? Последовательность проведения работ: подготовка (Test Preparation), тестирование (Testing), анализ результатов (Test Result Analisys) в разрезе запланированных фаз разработки.

Критерии начала тестирования:

- готовность тестовой платформы (тестового стенда);
- законченность разработки требуемого функционала;
- наличие всей необходимой документации.

Критерии окончания тестирования - результаты тестирования удовлетворяют критериям качества продукта:

- требования к количеству открытых багов выполнены;
- выдержка определенного периода без изменения исходного кода приложения Code Freeze (CF);
- выдержка определенного периода без открытия новых багов Zero Bug Bounce (ZBB).

Преимущества тест плана:

1. Возможность приоритезации задач по тестированию.
2. Построение стратегии тестирования, согласованной со всей командой.
3. Возможность вести учет всех требуемых ресурсов (как технических, так и человеческих).
4. Планирование использования ресурсов на тестирование.
5. Просчет рисков, возможных при проведении тестирования.

Составляющей частью планирования тестирования (как отдельного документа или же процесса планирования в целом) является стратегия тестирования. Стратегия может быть:

1. Частью общего тест-плана.
2. Отдельным документом.

Тестовая стратегия определяет то, как мы тестируем продукт. Это набор мыслей и идей, которые направляют процесс тестирования.

В стратегии тестирования описывают:

1. Тестовую среду.
2. Анализ рисков проекта.
3. Инструменты, которые будут использовать, чтобы провести автоматизированное тестирование и для других целей.
4. План действий при непредвиденных обстоятельствах.

Стратегия может быть представлена как в виде традиционно расписанного документа, так и в более наглядном формате, например, используя таблицу:

Приоритет объектов системы и действий	Готовность тестов
Высокий	Готово
Средний	В процессе
Низкий	Не готово

Объект системы и действия	Стратегия	Функциональное тестирование	Нефункциональное тестирование				
			Юзабилити	Нагрузка	Скорость	Безопасность	Окружения
		Тесты	Тесты	Тесты	Тесты		Тесты
Объект 1	Т.к. это важный объект (один из ключевых объектов с высочайшим приоритетом), нам требуется детальное тестирование и подготовка тест-анализа на уровне S&T. После подготовки, эти тесты необходимо задокументировать, т.к. мы будем часто их проходить.	link	link	link		link	
Действие 1	Одно из ключевых действий. Требуется ТА по ДПЗ и согласование этого ТА со всей командой. Чаще всего здесь сразу вкратце выписывается, что именно планируем тестировать, какие параметры?	link			link	link	link
Действие 2	Одно из ключевых действий. Требуется ТА по ДПЗ и согласование этого ТА со всей командой.	link	link	link			
Действие 3	Важное действие, делаем ТА без согласования	link	link	link			
Действие 4	Сделаем полноценное ТА, если будет время, пока - исследовательское тестирование						
Объект 2	S&T						
Действие 1	Важное действие, делаем ТА без согласования	link	link	link	link		link
Действие 2	Сделаем полноценное ТА, если будет время, пока - исследовательское тестирование						

В общем, план тестирования устанавливает цели процесса тестирования, он определяет, что будет проверяться, а стратегия тестирования описывает, как достичь целей, поставленных в плане тестирования.

Чек-лист - набор идей по тестированию, разработке, планированию и управлению. А также, это перечень формализованных тестовых случаев в удобном для проведения проверок виде. Тестовые случаи в чек-листе не должны быть зависимыми друг от друга.

Обязательно должен содержать в себе следующую информацию:

- идея проверок;

- набор входных данных;
- ожидаемые результаты;
- булевая отметка о прохождении/непрохождении тестового случая;
- булевая отметка о совпадении/несовпадении фактического и ожидаемого результата по каждой проверке.

Может также содержать шаги для проведения проверки, данные об особенностях окружения и прочую информацию необходимую для проведения проверок.

Цель – обеспечить стабильность покрытия требований проверками необходимыми и достаточными для заключения о соответствии им продукта. Особенностью является то, что чек-листы компонуются теми тестовыми случаями, которые показательны для определенного требования.

Чек-лист, чаще всего, представляет собой обычный и привычный нам список, который может быть:

1. Списком, в котором последовательность пунктов не имеет значения (например, список значений некоего поля);
2. Списком, в котором последовательность пунктов важна (например, шаги в краткой инструкции).
3. Структурированным (многоуровневым) списком (вне зависимости от учёта последовательности пунктов), что позволяет отразить иерархию идей.

Чек-лист должен обладать рядом важных свойств:

- **Логичность.**Чек-лист пишется не «просто так», а на основе целей и для того, чтобы помочь в достижении этих целей. К сожалению, одной из самых частых и опасных ошибок при составлении чек-листа является превращение его в свалку мыслей, которые никак не связаны друг с другом.
- **Последовательность и структурированность.** Со структурированностью всё достаточно просто - она достигается за счёт оформления чек-листа в виде многоуровневого списка. Что касается последовательности, то, даже в том случае, когда пункты чек-листа не описывают цепочку действий, человеку всё равно удобнее воспринимать информацию в виде неких небольших групп идей, переход между которыми является понятным и очевидным (например, сначала можно прописать идеи простых позитивных тест-кейсов, потом идеи простых негативных тест-кейсов, потом постепенно повышать сложность тест-кейсов, но не стоит подавать эти идеи вперемешку).
- **Полнота и неизбыточность.** Чек-лист должен представлять собой аккуратную «сухую выжимку» идей, в которых нет дублирования (которые часто появляется из-за разных формулировок одной и той же идеи) и, в то же время ничто важное не упущено.

Правила составления чек-листов:

- Одна операция.
- Пункты чек-листа - это минимальные полные операции. Например, заказать изготовление визиток и доставить визитки в офис - это 2 разных операции. Поэтому в чек-листе они отображаются отдельными пунктами: визитки заказаны и визитки доставлены в офис.
- Пункты пишутся в утвердительной форме. Цель чек-листа – проверка готовности задачи, поэтому лучше составлять пункты в утвердительной форме - «заказаны, доставлены». Сравните формулировку: «заказать визитки» и «визитки заказаны».
- Оптимальное количество пунктов - до 20. Чек-листы не должны быть длинными. Если все же это требуется, то лучше разбить задачу на несколько этапов и составить к каждому этапу отдельный чек-лист.

Преимущества использования чек-листов:

- Структурирование информации у сотрудника. При записи необходимых действий у сотрудника чётко вырисовывается нужная последовательность задач.
- Повышение скорости обучения новых сотрудников. Не нужно повторять несколько раз последовательность операций. Достаточно провести короткий инструктаж и дать чек-лист для самостоятельной работы.

- Высокий результат, уменьшение количества ошибок. Как уже говорилось ранее, чек-листы помогают избежать проколов и ошибок по невнимательности.
- Взаимозаменяемость сотрудников.
- Экономия рабочего времени. Сотрудники будут значительно меньше времени тратить на переделывание задач.

Проверка	Результат	Комментарии
Операции с файлами	Passed	
Создание файла	Passed	
Открытие файла	Passed	
Сохранение документа	Passed	
Печать	Passed	
Редактирование файлов	Failed	http://bt.qatestlab.com/view.php?id=1084
Отмена	Passed	
Копирование	Passed	
Вырезание	Passed	
Вставка	Passed	
Удаление	Passed	
Поиск	Failed	http://bt.qatestlab.com/view.php?id=1085
Поиск с заменой	Failed	http://bt.qatestlab.com/view.php?id=1086
Вставка даты	Passed	
Форматирование	Passed	
Перенос строки	Passed	
Изменение шрифта	Passed	
Справка	Passed	

Рис. 4.1. Пример чек-листа

Тест кейсы

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

Высокоуровневый тест-кейс - тест-кейс без конкретных входных данных и ожидаемых результатов. Как правило, ограничивается общими идеями и операциями, схож по своей сути с подробно описанным пунктом чек-листа. Достаточно часто встречается в интеграционном тестировании и системном тестировании, а также на уровне дымового тестирования. Может служить отправной точкой для проведения исследовательского тестирования или для создания низкоуровневых тест-кейсов.

Низкоуровневый тест-кейс - тест-кейс с конкретными входными данными и ожидаемыми результатами. Представляет собой полностью готовый к выполнению тест-кейс и является наиболее классическим видом тест-кейсов. Начинающих тестировщиков чаще всего учат писать именно такие тесты, поскольку прописать все данные подробно намного проще, чем понять, какой информацией можно пренебречь, при этом не снизив ценность тест-кейса.

Спецификация тест-кейса - документ, описывающий набор тест-кейсов (включая их цели, входные данные, условия и шаги выполнения, ожидаемые результаты) для тестируемого элемента.

Спецификация теста - документ, состоящий из спецификации тест-дизайна, 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»).

Цель написания тест-кейсов:

Тестирование можно проводить и без тест-кейсов (не нужно, но можно; да, эффективность такого подхода варьируется в очень широком диапазоне, в зависимости от множества факторов). Наличие же тест-кейсов позволяет:

1. Структурировать и систематизировать подход к тестированию (без чего крупный проект почти гарантированно обречён на провал).

2. Вычислять метрики тестового покрытия (test coverage metrics) и принимать меры по его увеличению (тест-кейсы здесь являются главным источником информации, без которого существование подобных метрик теряет смысл).

3. Отслеживать соответствие текущей ситуации плану (сколько примерно понадобится тест-кейсов, сколько уже есть, сколько выполнено из запланированного на данном этапе количества и т.д.).

4. Уточнить взаимопонимание между заказчиком, разработчиками и тестировщиками (тест-кейсы зачастую намного более наглядно показывают поведение приложения, чем это отражено в требованиях)..

5. Хранить информацию для длительного использования и обмена опытом между сотрудниками и командами (или, как минимум, не пытаться удержать в голове сотни страниц текста).

6. Проводить регрессионное тестирование и повторное тестирование (которые без тест-кейсов было бы вообще невозможно выполнить).

7. Повышать качество требований (написание чек-листов и тест-кейсов - хорошая техника тестирования требований).

8. Быстро вводить в курс дела нового сотрудника, недавно подключившегося к проекту.

Жизненный цикл тест-кейса

В отличие от отчёта о дефекте, у которого есть полноценный развитый жизненный цикл, для тест-кейса речь идёт о наборе состояний, в которых он может находиться (См. рис.4.2. Жирным шрифтом отмечены наиболее важные состояния).

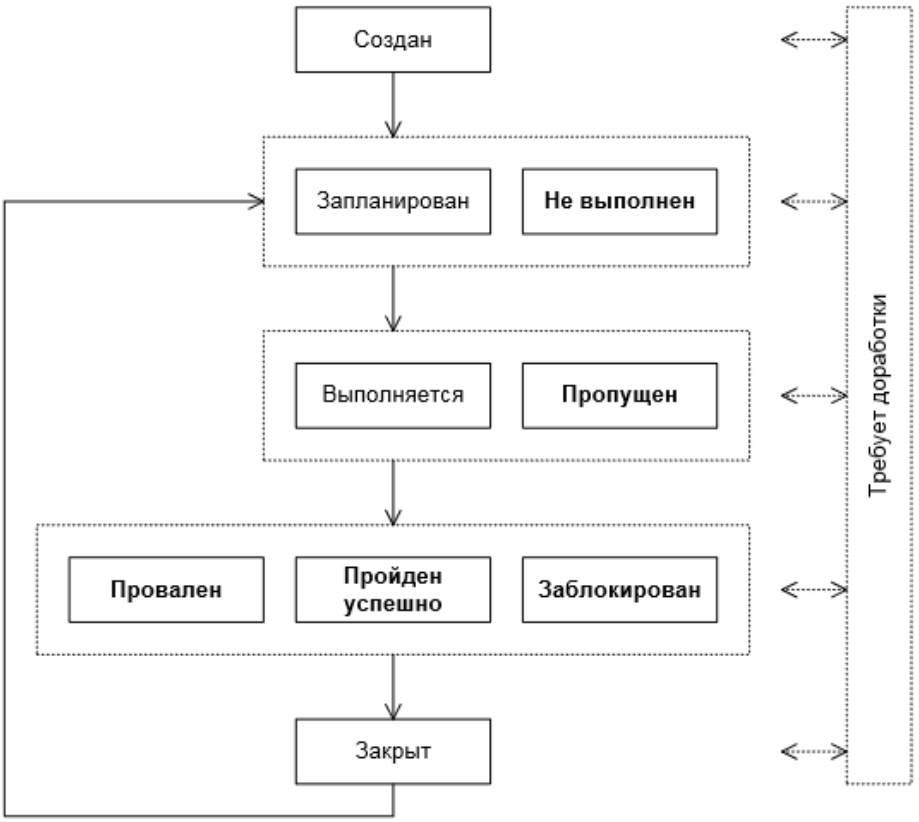


Рис. 4.2. Жизненный цикл тест-кейса

Создан (new) - типичное начальное состояние практически любого артефакта. Тест-кейс автоматически переходит в это состояние после создания.

Запланирован (planned, ready for testing) - в этом состоянии тест-кейс находится, когда он или явно включён в план ближайшей итерации тестирования, или, как минимум, готов для выполнения.

Не выполнен (not tested) - в некоторых системах управления тест-кейсами это состояние заменяет собой предыдущее («запланирован»). Нахождение тест-кейса в данном состоянии означает, что он готов к выполнению, но ещё не был выполнен.

Выполняется (work in progress) - если тест-кейс требует длительное время для выполнения, то он может быть переведён в это состояние для подчёркивания того факта, что работа идёт, и скоро можно ожидать её результатов. Если выполнение тест-кейса занимает мало времени, это состояние, как правило, пропускается, а тест-кейс сразу переводится в одно из трёх следующих состояний - «провален», «пройден успешно» или «заблокирован».

Пропущен (skipped) - бывают ситуации, когда выполнение тест-кейса отменяется по соображениям нехватки времени или изменения логики тестирования.

Провален (failed) - данное состояние означает, что в процессе выполнения тест-кейса был обнаружен дефект, заключающийся в том, что ожидаемый результат по как минимум одному шагу тест-кейса не совпадает с фактическим результатом. Если в процессе выполнения тест-кейса был «случайно» обнаружен дефект, никак не связанный с шагами тест-кейса и их ожидаемыми результатами, тест-кейс считается пройденным успешно (при этом, естественно, по обнаруженному дефекту создаётся отчёт о дефекте).

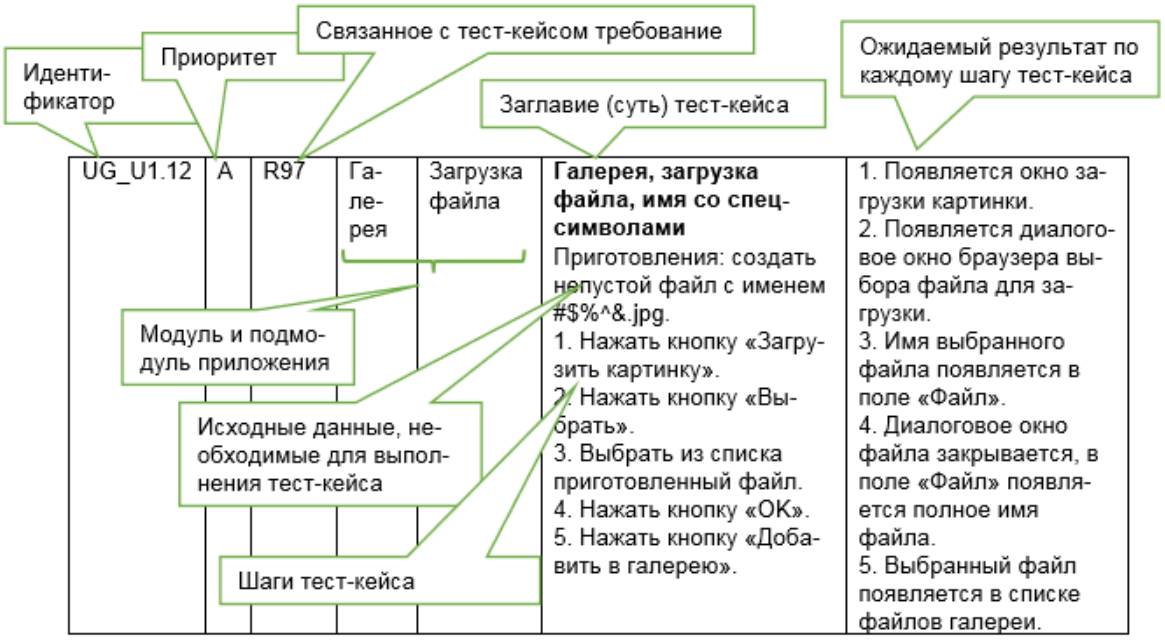
Пройден успешно (passed) - данное состояние означает, что в процессе выполнения тест-кейса не было обнаружено дефектов, связанных с расхождением ожидаемых и фактических результатов его шагов.

Заблокирован (blocked) - данное состояние означает, что по какой-то причине выполнение тест-кейса невозможно (как правило, такой причиной является наличие дефекта, не позволяющего реализовать некий пользовательский сценарий).

Закрыт (closed) - очень редкий случай, т.к. тест-кейс, как правило, оставляют в состояниях «провален / пройден успешно / заблокирован / пропущен». В некоторых системах управления тест-кейс переводят в данное состояние, чтобы подчеркнуть тот факт, что на данной итерации тестирования все действия с ним завершены.

Требует доработки (not ready) - как видно из схемы, в это состояние (или из него) тест-кейс может быть переведён в любой момент времени, если в нём будет обнаружена ошибка, если изменятся требования, по которым он был написан, или наступит иная ситуация, не позволяющая считать тест-кейс пригодным для выполнения и перевода в иные состояния.

Структура тест кейса



Идентификатор (identifier) представляет собой уникальное значение, позволяющее однозначно отличить один тест-кейс от другого и используемое во всевозможных ссылках. В общем случае идентификатор тест-кейса может представлять собой просто уникальный

номер, но (если позволяет инструментальное средство управления тест-кейсами) может быть и куда сложнее: включать префиксы, суффиксы и иные осмысленные компоненты, позволяющие быстро определить цель тест-кейса и часть приложения (или требований), к которой он относится (например, UR216_S12_DB_Neg).

Приоритет (priority) показывает важность тест-кейса. Он может быть выражен буквами (A, B, C, D, E), цифрами (1, 2, 3, 4, 5), словами («крайне высокий», «высокий», «средний», «низкий», «крайне низкий») или иным удобным способом. Количество градаций также не фиксировано, но, чаще всего, лежит в диапазоне от трёх до пяти.

Приоритет тест-кейса может коррелировать с:

- важностью требования, пользовательского сценария или функции, с которыми связан тест-кейс;
- потенциальной важностью дефекта, на поиск которого направлен тест-кейс;
- степенью риска, связанного с проверяемым тест-кейсом требованием, сценарием или функцией.

Основная задача этого атрибута - упрощение распределения внимания и усилий команды (более высокоприоритетные тест-кейсы получают их больше), а также упрощение планирования и принятия решения о том, чем можно пожертвовать в некоей форс-мажорной ситуации, не позволяющей выполнить все запланированные тест-кейсы.

Связанное с тест-кейсом требование (requirement) показывает то основное требование, проверке выполнения которого посвящён тест-кейс (основное, поскольку один тест-кейс может затрагивать несколько требований). Наличие этого поля улучшает такое свойство тест-кейса, как прослеживаемость. Заполнение этого поля является не обязательным.

Модуль и подмодуль приложения (module and submodule) указывают на части приложения, к которым относится тест-кейс, и позволяют лучше понять его цель. Идея деления приложения на модули и подмодули проистекает из того, что в сложных системах практически невозможно охватить взглядом весь проект целиком, и вопрос «как протестировать это приложение» становится недопустимо сложным. Тогда приложение логически разделяется на компоненты (модули), а те, в свою очередь, на более мелкие компоненты (подмодули). Как правило, иерархия модулей и подмодулей создаётся как единый набор для всей проектной команды, чтобы исключить путаницу из-за того, что разные люди будут использовать разные подходы к такому разделению или даже просто разные названия одних и тех же частей приложения. В реальности проще всего отталкиваться от архитектуры и дизайна приложения. Например, в уже знакомом нам приложении можно выделить такую иерархию модулей и подмодулей:

- *Механизм запуска*:
  - механизм анализа параметров;
  - механизм сборки приложения;
  - механизм обработки ошибочных ситуаций.
- *Механизм взаимодействия с файловой системой:*
  - механизм обхода дерева SOURCE_DIR;
  - механизм обработки ошибочных ситуаций.

Заглавие (суть) тест-кейса (title) призвано упростить и ускорить понимание основной идеи (цели) тест-кейса без обращения к его остальным атрибутам.

Исходные данные, необходимые для выполнения тест-кейса (precondition, preparation, initial data, setup), позволяют описать всё то, что должно быть подготовлено до начала выполнения тест-кейса, например:

- состояние базы данных;
- состояние файловой системы и её объектов;
- состояние серверов и сетевой инфраструктуры.

Шаги тест-кейса (steps) описывают последовательность действий, которые необходимо реализовать в процессе выполнения тест-кейса.

Общие рекомендации по написанию шагов таковы:

1. Начинайте с понятного и очевидного места, не пишите лишних начальных шагов (запуск приложения, очевидные операции с интерфейсом и т.п.).

2. Даже если в тест-кейсе всего один шаг, нумеруйте его (иначе возрастает вероятность в будущем случайно «приклеить» описание этого шага к новому тексту).

3. Если вы пишете на русском языке, то используйте безличную форму (например, «открыть», «ввести», «добавить» вместо «откройте», «введите», «добавьте»), в английском языке не надо использовать частицу «to» (т.е. «запустить приложение» будет «start application», не «to start application»).

4. Соотносите степень детализации шагов и их параметров с целью тест-кейса, его сложностью, уровнем и т.д. В зависимости от этих и многих других факторов степень детализации может варьироваться от общих идей до предельно чётко прописанных значений и указаний.

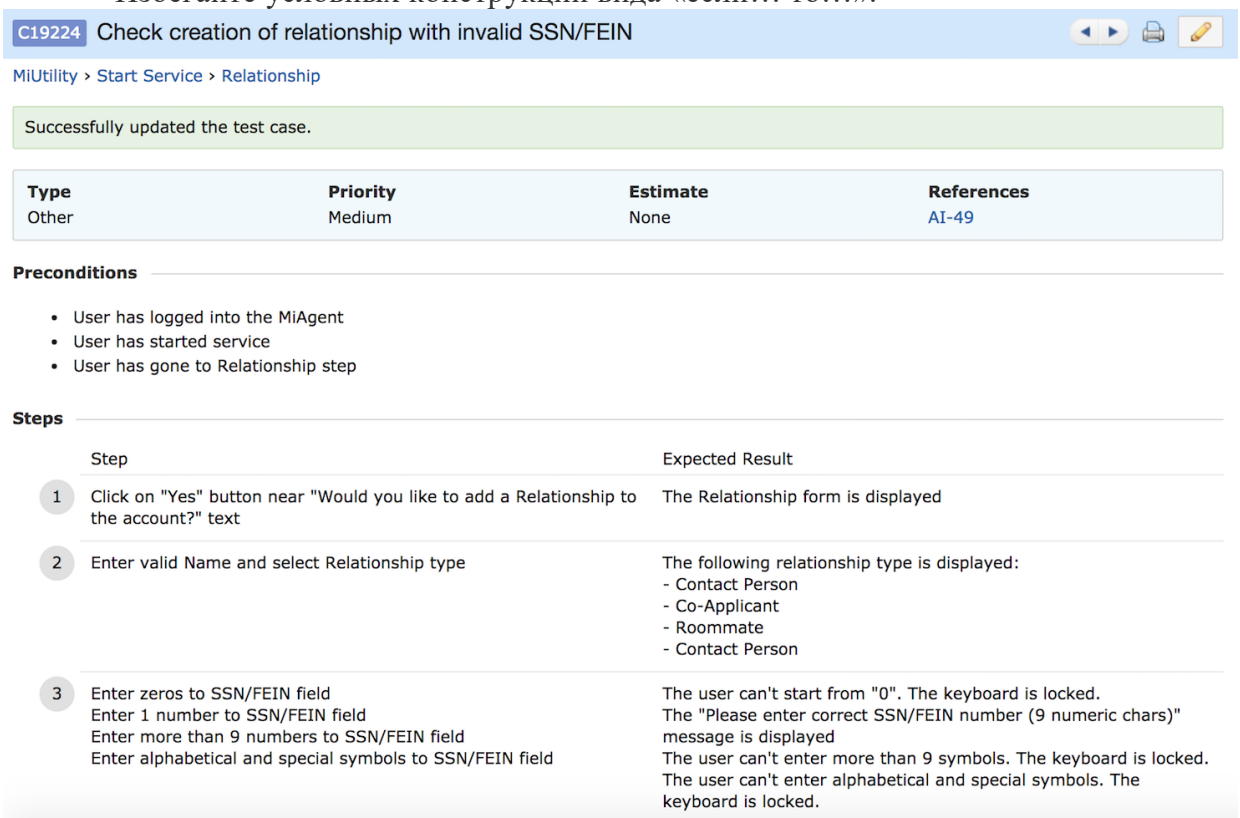
5. Ссылайтесь на предыдущие шаги и их диапазоны для сокращения объёма текста (например, «повторить шаги 3–5 со значением…»).

6. Пишите шаги последовательно, без условных конструкций вида «если… то…».

Ожидаемые результаты (expected results) по каждому шагу тест-кейса описывают реакцию приложения на действия, описанные в поле «шаги тест-кейса». Номер шага соответствует номеру результата.

По написанию ожидаемых результатов можно порекомендовать следующее:

- Описывайте поведение системы так, чтобы исключить субъективное толкование (например, «приложение работает верно» — плохо, «появляется окно с надписью…» — хорошо).
- Пишите ожидаемый результат по всем шагам без исключения, если у вас есть хоть малейшие сомнения в том, что результат некоего шага будет совершенно тривиальным и очевидным (если вы всё же пропускаете ожидаемый результат для какого-то тривиального действия, лучше оставить в списке ожидаемых результатов пустую строку — это облегчает восприятие).
- Пишите кратко, но не в ущерб информативности.
- Избегайте условных конструкций вида «если… то…».



C19224 Check creation of relationship with invalid SSN/FEIN

MiUtility › Start Service › Relationship

Successfully updated the test case.

Type	Priority	Estimate	References
Other	Medium	None	AI-49

Preconditions

- User has logged into the MiAgent
- User has started service
- User has gone to Relationship step

Steps

	Step	Expected Result
1	Click on "Yes" button near "Would you like to add a Relationship to the account?" text	The Relationship form is displayed
2	Enter valid Name and select Relationship type	The following relationship type is displayed: - Contact Person - Co-Applicant - Roommate - Contact Person
3	Enter zeros to SSN/FEIN field Enter 1 number to SSN/FEIN field Enter more than 9 numbers to SSN/FEIN field Enter alphabetical and special symbols to SSN/FEIN field	The user can't start from "0". The keyboard is locked. The "Please enter correct SSN/FEIN number (9 numeric chars)" message is displayed The user can't enter more than 9 symbols. The keyboard is locked. The user can't enter alphabetical and special symbols. The keyboard is locked.

Набор тест-кейсов (test case suite, test suite, test set) - совокупность тест-кейсов, выбранных с некоторой общей целью или по некоторому общему признаку.

Наборы тест-кейсов можно разделить на **свободные** (порядок выполнения тест-кейсов не важен) и **последовательные** (порядок выполнения тест-кейсов важен).

Преимущества свободных наборов:

- Тест-кейсы можно выполнять в любом удобном порядке, а также создавать «наборы внутри наборов».
- Если какой-то тест-кейс завершился ошибкой, это не повлияет на возможность выполнения других тест-кейсов.

Преимущества последовательных наборов:

- Каждый следующий в наборе тест-кейс, в качестве входного состояния приложения, получает результат работы предыдущего тест-кейса, что позволяет сильно сократить количество шагов в отдельных тест-кейсах.
- Длинные последовательности действий куда лучше имитируют работу реальных пользователей, чем отдельные «точечные» воздействия на приложение.

К отдельному подвиду последовательных наборов тест-кейсов (или даже неоформленных идей тест-кейсов, таких, как пункты чек-листа) можно отнести *пользовательские сценарии* (или сценарии использования), представляющие собой цепочки действий, выполняемых пользователем в определённой ситуации для достижения определённой цели.

Классификация наборов тест-кейсов

		По изолированности тест-кейсов друг от друга	
		Изолированные	**Обобщённые**
По образованию тест-кейсами строгой последовательности	**Свободные**	Изолированные свободные	Обобщённые свободные
	Последовательные	Изолированные последовательные	Обобщённые последовательные

- **Набор изолированных свободных тест-кейсов:** действия из раздела «приготовления» нужно повторять перед каждым тест-кейсом, а сами тест-кейсы можно выполнять в любом порядке.
- **Набор обобщённых свободных тест-кейсов:** действия из раздела «приготовления» нужно выполнить один раз (а потом просто выполнять тест-кейсы), а сами тест-кейсы можно выполнять в любом порядке.
- **Набор изолированных последовательных тест-кейсов:** действия из раздела «приготовления» нужно повторять перед каждым тест-кейсом, а сами тест-кейсы нужно выполнять в строго определённом порядке.
- **Набор обобщённых последовательных тест-кейсов:** действия из раздела «приготовления» нужно выполнить один раз (а потом просто выполнять тест-кейсы), а сами тест-кейсы нужно выполнять в строго определённом порядке.

Главное преимущество изолированности: каждый тест-кейс выполняется в «чистой среде», на него не влияют результаты работы предыдущих тест-кейсов.

Главное преимущество обобщённости: приготовления не нужно повторять (экономия времени).

Главное преимущество последовательности: ощутимое сокращение шагов в каждом тест-кейсе, т.к. результат выполнения предыдущего тест-кейса является начальной ситуацией для следующего.

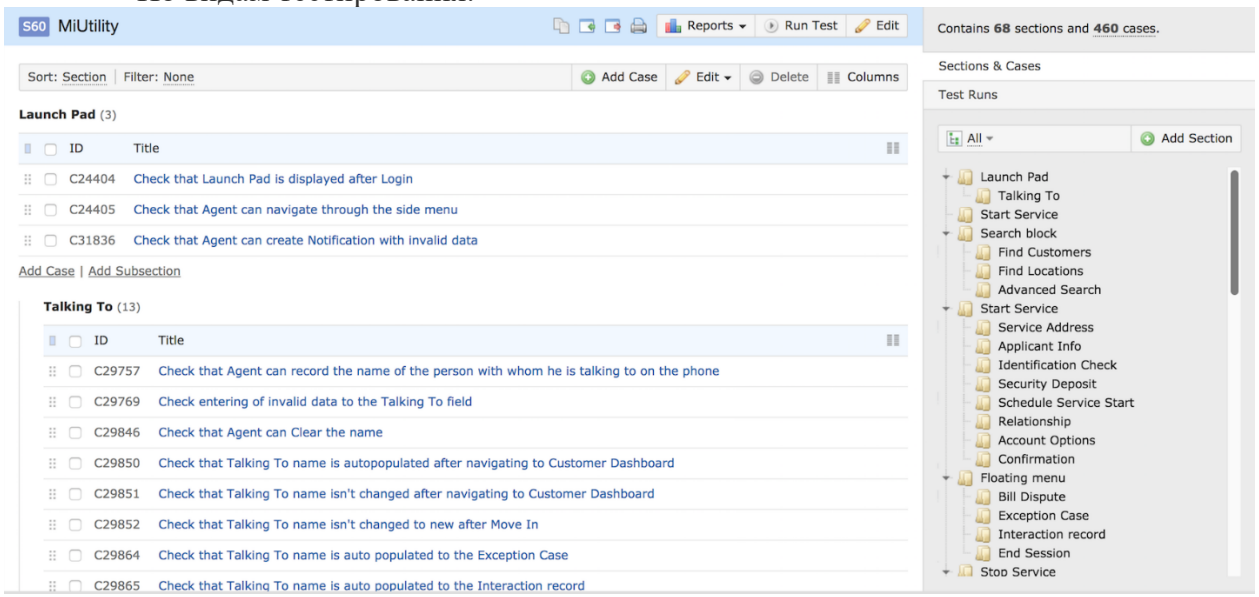
Главное преимущество свободы: возможность выполнять тест-кейсы в любом порядке, а также то, что при провале какого-то тест-кейса (приложение не пришло в ожидаемое состояние) остальные тест-кейсы по-прежнему можно выполнять.

Набор тест-кейсов всегда создаётся с какой-то целью, на основе какой-то логики, и по этим же принципам в набор включаются тесты, обладающие подходящими свойствами.

Подходы к составлению наборов тест-кейсов:

- На основе чек-листов.
- На основе разбиения приложения на модули и подмодули. Для каждого модуля (или его отдельных подмодулей) можно составить свой набор тест кейсов.

- По принципу проверки самых важных, менее важных и всех остальных функций приложения.
- По принципу группировки тест-кейсов для проверки некоего уровня требований или типа требований, группы требований или отдельного требования.
- По принципу частоты обнаружения тест-кейсами дефектов в приложении (например, мы видим, что некоторые тест-кейсы раз за разом завершаются неудачей, значит, мы можем объединить их в набор, условно названный «проблемные места в приложении»).
- По архитектурному принципу: наборы для проверки пользовательского интерфейса и всего уровня представления, для проверки уровня бизнес-логики, для проверки уровня данных.
- По области внутренней работы приложения, например, «тест-кейсы, затрагивающие работу с базой данных», «тест-кейсы, затрагивающие работу с файловой системой», «тест-кейсы, затрагивающие работу с сетью».
- По видам тестирования.



Лекция 11. Алгоритмы тестирования приложений

Цель: сформировать знания о координации проверки результатов тестирования приложений

QA-команды, внедрившие автоматизацию, все еще прогоняют часть своих автоматизированных кейсов вручную.

Но ЗАЧЕМ?

- QA-командам нравится прогонять некоторые тесты вручную, потому что они "очень важны".
- QA-команды не доверяют своему автотесту на сто процентов. При нехватке доверия ручные тесты играют роль обеспечения качества автотестов. Однако такой подход все равно не кажется хорошей идеей. Создать хорошую, устойчивую автоматизацию, конечно, нетривиальная задача – но если уж вы вкладываетесь в автоматизацию, займитесь ей серьезно и создайте скрипты, которым можно доверять.
- Это происходит, если нет четкого определения или понимания того, что именно автоматизировано. Поэтому люди тратят время на прогон ручных тестов, которые уже автоматизированы, предполагая, что не автоматизировано ничего и работая так, как будто автоматизации нет. Зачем им тогда вообще автоматизаторы?
- За автотесты отвечает другая команда. Две тест-команды, для ручного и для автоматизированного тестирования – это само по себе неплохо, и в ряде случаев позволяет наилучшим образом достичь эффективной и достойной доверия

автоматизации. Плохо то, что эти команды могут быть полностью разъединены и работать над одним проектом, не общаясь и не кооперируясь.

- Команды прогоняют тесты вручную, чтобы внести результаты автотестов в системы тест-менеджмента и получить полные и внятные отчеты для руководства. Стремление к видимости процессов – одна из ценностей разработки в PractiTest. Неважно, сколько интеграций или фич тест-менеджмента мы добавляем – мы всегда фокусируемся на кросс-командной коммуникации.
- Ряд автоматизированных и ручных тестов "связаны", а не "продублированы". К примеру, автотест убеждается в инсталляции на все ОС, а "связанный с ним" ручной тест проверяет влияние абнормального сетевого трафика в ходе процесса установки. По сути, эти тесты дополняют друг друга.
- Некоторые тесты автоматизированы "наполовину", а не полностью – к примеру, тесты, в ходе которых проводится ряд операций с постоянной съемкой скриншотов. Эти скриншоты затем быстро просматриваются вручную в поисках проблем и багов GUI.
- Дополнительные ручные тесты добавляются к уже имеющейся автоматизации. К примеру, автоматизация используется для проверки установки системы и валидации процедур по добавлению, удалению и изменению данных. Когда эти тесты готовы, можно взять созданную систему и прогонять на ней дополнительные ручные тесты на основании существующих данных и настроек.

Некоторые из этих причин имеют смысл, но во всех перечисленных случаях есть проблема правильной координации автоматизации и ручного тестирования.

Проблемы координации автоматизированного и ручного тестирования

Проблемы, возникающие при координации ручного и автоматизированного тестирования, не связанные напрямую с повторным прогоном тестов. Вот самые интересные моменты:
Определить, что нужно перевести из ручного в автоматизированное тестирование
Это первая проблема, и одно из наиболее важных решений, так как оно будет определять ценность вашей автоматизации.
Автоматизация должна учитывать три главных фактора:
Коэффициент окупаемости – выгодность автоматизации теста, то, что вы получите от автоматизации этого конкретного сценария. Обычно это функция временных затрат на автоматизацию и количества прогонов автотеста – и сбереженного в результате времени.
Сложность теста – тут речь о том, насколько сложно создать автотест, и насколько легко провести этот тест вручную. Существуют тесты, чересчур сложные для ручного прогона (к примеру, тесты API), а ряд тестов почти невозможно автоматизировать (к примеру, удобство использования). Нужно грамотно выбирать, какие тесты больше подходят для автоматизации, а какие –для ручного тестирования.
Стабильность приложения – возможно, главная проблема любого автотеста – это способность справляться с изменениями в приложении. Некоторые платформы автоматизации делают это лучше других, но ни одна не сможет импровизировать и вникать так, как это делают люди. Этот фактор также поможет вам сделать выбор, что именно автоматизировать, и не работать над частями приложения, которые в ближайшем будущем будут радикально переработаны.
Координация задач между командами ручного и автоматизированного тестирования
Серьезные проблемы также возникают из-за исходных различий между хорошим ручным тестировщиком и хорошим автоматизатором.
Кто за что отвечает?

- Должны ли автоматизаторы решать, что тестировать, или это решение за ручными тестировщиками?
- Кто отвечает за расписание прогонов – автоматизаторы или ручные тестировщики?
- Что будет, если автотест упадет? Должен ли "ручник" проверять баг, или это задача команды автоматизации?

Ответы на все эти вопросы нужно дать заранее, и тут тоже нет идеальных решений. Хороший автоматизатор ближе к разработчику, чем к тестировщику, и поэтому ответственность должна больше лежать на ручных тестировщиках. По сути, автоматизаторы предоставляют услугу, сотрудничая с ручными тестировщиками и поставляя им наилучший автоматизированный результат для их тест-сценариев.
Автоматизация – часть общих усилий по тестированию, и поэтому решение, что и когда прогонять, должно приниматься группой, отвечающей за тестирование в целом (обычно это также ложится на плечи ручных тестировщиков).

Снятие табу с тест-автоматизации

Комментарий Марко Вензелааера на LinkedIn: "Некоторые команды автоматизации превращают автоматизацию в "черную магию", секреты которой бдительно оберегаются от непосвященных". Марко озвучил то, что некоторые автоматизаторы думают, что если их работа воспринимается как нечто загадочное, тяжелое и крайне сложное, то это обезопасит их рабочие места. Они крайне успешно сеют эти мысли в головах ручных тестировщиков, у которых нет опыта программирования – они искренне верят всему, что им говорят коллеги-автоматизаторы.
Помимо ложного чувства гарантии занятости, эти автоматизаторы создают что-то вроде табу вокруг своей работы, и ручные тестировщики не пытаются ее понять или как-то повлиять на процесс. В результате это поведение вредит координации усилий и портит картину **общих достижений тест-команды**.
Как же координировать работу автоматизаторов и ручных тестировщиков в команде?
Ключ к поддержанию здоровых взаимоотношений ручного и автоматизированного тестирования – это **командная работа и коммуникация**.

Вам нужно убедиться, что обе команды стремятся к одной и той же цели, координируют свои задачи на основании единых приоритетов, и осознают, какой вклад вносит каждый из них в достижение целей организации. Иными словами, они должны понимать, как их совместная работа делает тестирование быстрее, шире и эффективнее.

Тривиальные вещи, которые надо учесть:

- Убедитесь, что тестировщики скоординированы, проводя регулярные планерки и убеждаясь в активном участии обеих команд в планировании тест-задач.
- Пусть обе команды работают в интегрированном окружении, в котором выполняются и ручные, и автоматизированные тесты. Это позволит тестировщикам и другим участникам команды видеть полную картину и понимать, что покрыто, что протестировано, и каков результат тестов, как на этапе планирования, так и на этапе выполнения. В конце концов, никого не волнует, как именно проведен тест – вручную или нет, важны его результаты.
- Наладьте процесс так, чтобы автоматизация была инструментом, помогающим вашей ручной команде. Правильный процесс – залог хороших взаимоотношений ручных и автоматизированных тестов. Обе команды должны понимать, что задача автоматизации – освободить время для ручного прогона более сложных тестов, которые сложно или дорого автоматизировать. Понимание, что команды дополняют друг друга, а не соревнуются, позволит им сконцентрироваться на общих проблемах, а не на разногласиях.

Вам нужно убедиться, что у ваших команд есть общая цель и инфраструктура, позволяющая координировать их работу.

Контрольные вопросы:

1. Основное назначение большого количество проектной документации при создании сложной программной системы?
2. Что такое тест-план?
3. Какая документация, сопровождает процесс верификации?

Лекция 13. Применение методов и средств проверки

работоспособности программного кода ПО

В фундаментальную концепцию проектирования ПС входят базовые положения, стратегии, методы, которые применяются на процессах ЖЦ и обеспечивают тестирование (верификацию) на множестве тестовых наборов данных. К методам проектирования ПС относятся структурные, объектно-ориентированные и др. Их основу составляют теоретические, инструментальные и прикладные средства, которые влияют на процесс тестирования.

Теоретические средства определяют процесс программирования и тестирования программного продукта. К ним относятся методы верификации и доказательства правильности *спецификации программ* (см. "Формальные спецификации, доказательство и верификация программ"), метрики измерения (Холстеда, цикломатичная сложность Маккейба и др.) в качестве отдельных характеристик как формализованных элементов теории определения правильности и гарантии свойств конечного *ПО*. Например, концепция " *чистая комната* " базируется на формализмах доказательства и изучения свойств процессов кодирования и тестирования программ. Что касается тестирования как процесса, то это проверка правильности работы программы *по* заданным описаниям тестов и покрытия данными соответствующих критериев программы

Инструментальные средства - это способы поддержки кодирования и тестирования (компиляторы, генераторы программ, отладчики и др.), а также организационные средства планирования и *отбора тестов* для программ, которые обеспечивают обработку текста на ЯП и подготовку для них соответствующих тестов.

При проверке правильности программ и систем рассматриваются процессы верификации, валидации и тестирования ПС, которые регламентированы в стандарте ISO/IEC 12207 жизненного *цикла ПО*. В некоторой зарубежной литературе процессы верификации и тестирования отождествляются. С теоретической точки зрения *верификация* была рассмотрена в "Формальные спецификации, доказательство и верификация программ" , здесь же будут определены задачи и действия, соответствующих процессов ЖЦ.

Тестирование - это процесс обнаружения ошибок в *ПО* путем исполнения выходного кода ПС на тестовых данных, сбора рабочих характеристик в динамике выполнения в конкретной операционной среде, выявления различных ошибок, дефектов, отказов и изъянов, вызванных нерегулярными и аномальными ситуациями или аварийным прекращением работы *ПО*.Важное *место* в проведении верификации и тестирования занимают организационные аспекты - *деятельность* группы специалистов, осуществляющих планирование этих процессов, подготовку тестовых данных и наблюдение за тестированием.

7.1. Процессы ЖЦ верификация и валидация программ

Верификация и *валидация*, как методы, обеспечивают соответственно проверку и *анализ* правильности выполнения заданных функций и соответствия *ПО* требованиям заказчика, а также заданным спецификациям. Они представлены в стандартах как самостоятельные процессы ЖЦ и используются, начиная от этапа анализа требований и кончая проверкой правильности функционирования программного кода на заключительном этапе, а именно, тестировании.

Для этих процессов определены цели, задачи и действия *по* проверке правильности создаваемого продукта (рабочие, промежуточные продукты) на этапах ЖЦ. Рассмотрим их трактовку в стандартном представлении.

Процесс верификации. Цель процесса - убедиться, что каждый *программный продукт* (и/или сервис) проекта отражает согласованные требования к их реализации. Этот процесс основывается:

- на стратегии и критериях верификации применительно ко всем рабочим программным продуктам;
- на выполнении действий стандарта по верификации;
- на *устранении недостатков*, обнаруженных в программных (рабочих и промежуточных) продуктах;
- на согласовании результатов верификации с заказчиком.

Процесс верификации может проводиться исполнителем программы или другим сотрудником той же организации, или сотрудником другой организации, например заказчиком. Этот процесс включает в себя действия *по* его внедрению и выполнению.

Внедрение процесса заключается в определении критических элементов (процессов и программных продуктов), которые должны подвергаться верификации, в выборе исполнителя верификации, инструментальных средств поддержки процесса верификации, в составлении плана верификации и его утверждении. В процессе верификации выполняются задачи проверки условий: контракта, процесса, требований, интеграции, проекта, кода и документации.При верификации согласно плану и требований заказчика проверяется правильность выполнения функций системы, интерфейсов и взаимосвязей компонентов, а также доступа к данным и к средствам защиты.

Процесс валидации. Цель процесса - убедиться, что специфические требования для программного продукта выполнены, и осуществляется это с помощью:

- разработанной стратегии и критериев валидации для всех *рабочих продуктов*;
- оговоренных действий по проведению валидации;
- демонстрации соответствия разработанных программных продуктов требованиям заказчика и правилам их использования;
- согласования с заказчиком полученных результатов валидации.

Процесс валидации может проводиться самим исполнителем или другим лицом, например, заказчиком, осуществляющим действия *по* внедрению и проведению этого процесса *по* плану, в котором отражены элементы и задачи проверки. При этом используются методы, инструментальные средства и процедуры выполнения задач процесса для установления соответствия тестовых требований и особенностей использования программных продуктов проекта.

На других процессах ЖЦ выполняются дополнительные действия:

- проверка и контроль проектных решений с помощью методик и процедур просмотра хода разработки;
- обращение к *CASE-системам,* которые содержат процедуры проверки требований к продукту;
- просмотры и инспекции промежуточных результатов на соответствие их требованиям для подтверждения того, что ПО имеет корректную реализацию требований и удовлетворяет условиям выполнения системы.

Таким образом, основные задачи процессов верификации и валидации состоят в том, чтобы *проверить и подтвердить*, что конечный *программный продукт* отвечает назначению и удовлетворяет требованиям заказчика. Эти процессы взаимосвязаны и определяются, как правило, одним общим термином "*верификация* и *валидация*" или "*Verification and Validation*" (V&V).

V&V основаны на планировании их как процессов, так и проверки для наиболее критичных элементов проекта: *компонент*, интерфейсов (программных, технических и информационных), взаимодействий объектов (протоколов и сообщений), передач данных между компонентами и их защиты, а также оставленных тестов и *тестовых процедур*.

После проверки отдельных компонентов системы проводятся их *интеграция* и повторная *верификация* и *валидация* интегрированной системы, создается комплект документации, отображающий правильность проверки формирования требований, результатов инспекций и тестирования.

7.2. Тестирование программ

Тестирование можно рассматривать, как процесс семантической отладки (проверки) программы, заключающийся в исполнении последовательности различных наборов контрольных тестов, для которых заранее известен результат. Т.е. тестирование предполагает выполнение программы и получение конкретных результатов выполнения тестов

Тесты подбираются так, чтобы они охватывали как можно больше типов ситуаций алгоритма программы. Менее жесткое требование - выполнение хотя бы один раз каждой ветви программы.

Исторически первым видом тестирования была *отладка*.

Отладка - это проверка описания программного объекта на ЯП с целью обнаружения в нем ошибок и последующее их устранение. Ошибки обнаруживаются компиляторами при их синтаксическом контроле. После этого проводится *верификация по* проверке правильности кода и *валидация по* проверке соответствия продукта заданным требованиям.

Целью тестирования - проверка работы реализованных функций в соответствии с их спецификацией. На основе внешних спецификаций функций и проектной информации на процессах ЖЦ создаются функциональные тесты, с помощью которых проводится тестирование с учетом требований, сформулированных на этапе анализа *предметной области*. Методы *функционального тестирования* подразделяются на статические и динамические.

7.2.1. Статические методы тестирования

Статические методы используются при проведении инспекций и рассмотрении спецификаций компонентов без их выполнения.Техника статического анализа заключается в методическом просмотре (или обзоре) и анализе структуры программ, а также в доказательстве их правильности. Статический анализ направлен на анализ документов, разработанных на всех этапах ЖЦ и заключается в инспекции исходного кода и сквозного контроля программы.

Инспекция ПО - это статическая проверка соответствия программы заданным спецификациями, проводится путем анализа различных представлений результатов проектирования (документации, требований, спецификаций, схем или исходного кода программ) на процессах ЖЦ. Просмотры и инспекции результатов проектирования и соответствия их требованиям заказчика обеспечивают более высокое качество создаваемых ПС.

При инспекции программ рассматриваются документы рабочего проектирования на этапах ЖЦ совместно с независимыми экспертами и участниками разработки ПС. На начальном этапе проектирования инспекция предполагает проверку полноты, целостности, однозначности, непротиворечивости и совместимости документов с исходными требованиями к программной системе. На этапе реализации системы под *инспекцией* понимается анализ текстов программ на соблюдение требований стандартов и принятых руководящих документов технологии программирования.

Эффективность такой проверки заключается в том, что привлекаемые эксперты пытаются взглянуть на проблему "со стороны" и подвергают ее всестороннему критическому анализу.

Эти приемы позволяют на более ранних этапах проектирования обнаружить ошибки или дефекты путем многократного просмотра исходных кодов. Символьное тестирование применяется для проверки отдельных участков программы на входных символьных значениях.

Кроме того, разрабатывается множество новых способов автоматизации символьного выполнения программ. Например, автоматизированное средство статического контроля для языков ориентированной разработки, инструменты автоматизации *доказательства корректности* и автоматизированный аппарат сетей Петри.

7.2.2. Динамические методы тестирования

Динамические методы тестирования используются в процессе выполнения программ. Они базируются на графе, связывающем причины ошибок с ожидаемыми реакциями на эти ошибки. В процессе тестирования накапливается информация об ошибках, которая используется при оценке надежности и качества ПС.

Динамическое тестирование ориентировано на проверку корректности ПС на множестве тестов, прогоняемых по ПС, в целях проверки и сбора данных на этапах ЖЦ и проведения измерения отдельных показателей (число отказов, сбоев) тестирования для оценки характеристик качества, указанных в требованиях, посредством выполнения системы на ЭВМ. Тестирование основывается на систематических, статистических, (вероятностных) и имитационных методах.

Дадим краткую их характеристику.

Систематические методы тестирования делятся на методы, в которых программы рассматриваются как "черный ящик" (используется информация о решаемой задаче), и методы, в которых программа рассматривается как "белый ящик" (используется структура программы). Этот вид называют тестированием с управлением по данным или управлением по входувыходу. Цель - выяснение обстоятельств, при которых поведение программы не соответствует ее спецификации. При этом количество обнаруженных ошибок в программе является критерием качества входного тестирования.

Цель динамического тестирования программ по принципу "черного ящика" - выявление одним тестом максимального числа ошибок с использованием небольшого подмножества возможных входных данных.

Методы "черного ящика" обеспечивают:

- эквивалентное разбиение;
- анализ граничных значений;
- применение функциональных диаграмм, которые в соединении с реверсивным анализом дают достаточно полную информацию о функционировании тестируемой программы.

Эквивалентное разбиение состоит в разбиении входной области данных программы на конечное число классов эквивалентности так, чтобы каждый тест, являющийся представителем некоторого класса, был эквивалентен любому другому тесту этого класса.

Классы эквивалентности выделяются путем перебора входных условий и разбиения их на две или более групп. При этом различают два типа классов эквивалентности: правильные, задающие входные данные для программы, и неправильные, основанные на задании ошибочных входных значений.Разработка тестов методом эквивалентного разбиения осуществляется в два этапа: выделение классов эквивалентности и построение тестов. При построении тестов, основанных на выборе входных данных, проводится символическое выполнение программы.

Итак, методы тестирования по принципу "черного ящика" используются для тестирования функций, реализованных в программе, путем проверки несоответствия между реальным поведением функций и ожидаемым поведением с учетом спецификаций требований. Во время подготовки к этому тестированию строятся таблицы условий, причинно-следственные графы и области разбивки. Кроме того, подготавливаются тестовые наборы, учитывающие параметры и условия среды, которые влияют на поведение функций. Для каждого условия определяется множество значений и ограничений предикатов, которые тестируются.

Метод "белого ящика" позволяет исследовать внутреннюю структуру программы, причем обнаружение всех ошибок в программе является критерием исчерпывающего тестирования маршрутов потоков (графа) передач управления, среди которых рассматриваются:

- (а) критерий покрытия операторов - набор тестов в совокупности должен обеспечить прохождение каждого оператора не менееодного раза;
- (б) критерий тестирования ветвей (известный как покрытие решений или покрытие переходов) - набор тестов в совокупности должен обеспечить прохождение каждой ветви и выхода, по крайней мере, один раз.

Критерий (б) соответствует простому структурному тесту и наиболее распространен на практике. Для удовлетворения этого критерия необходимо построить систему путей, содержащую все ветви программы. Нахождение такого оптимального покрытия в некоторых случаях осуществляется просто, а в других является более сложной задачей.

Тестирование по принципу "белого ящика" ориентировано на проверку прохождения всех путей программ посредством применения путевого и имитационного тестирования.

Путевое тестирование применяется на уровне модулей и графовой *модели программы* путем выбора тестовых ситуаций, подготовки данных и включает тестирование следующих элементов:

- операторов, которые должны быть выполнены хотя бы один раз, без учета ошибок, которые могут остаться в программе иззабольшого количества логических путей и необходимости прохождения подмножеств этих путей;
- путей по заданному графу потоков управления для выявления разных маршрутов передачи управления с помощью путевых предикатов, для вычисления которого создается набор тестовых данных, гарантирующих прохождение всех путей. Однако все пути протестировать бывает невозможно, поэтому остаются не выявленные ошибки, которые могут проявиться в процессе эксплуатации;
- блоков, разделяющих программы на отдельные частиблоки, которые выполняются один раз или многократно при нахождении путей в программе, включающих совокупность блоков реализации одной функции либо нахождения входного множества данных, которое будет использоваться для выполнения указанного пути.

"Белый ящик" базируется на структуре программы, в случае "черного ящика", о структуре программы ничего неизвестно. Для выполнения тестирования с помощью этих "ящиков" известными считаются выполняемые функции, входы (входные данные) и выходы (выходные данные), а также логика обработки, представленные в документации.

Практическая работа 1. Тестирование документации

1. Выбрать объект реального мира (например, карандаш, стол, чашка, клавиатура, сумка и др.) с целью последующей разработки тестовых проверок для него.

2. Разработать различные проверки в соответствии с классификацией видов тестирования для выбранного объекта реального мира. Результаты внести в таблицу 1.1.

Таблица 1.1 – Тестировые проверки для различных видов тестирования

Объект тестирования: указать		
Вид тестирования	**Краткое определение вида тестирования**	**Тестовые проверки**
Functional Testing		
Safety Testing		
Security Testing		
Compatibility Testing		
GUI Testing		
Usability Testing		
Accessibility Testing		
Internationalization Testing		
Performance Testing		
Stress Testing		
Negative Testing		
Black Box Testing		
Automated Testing		
Unit/Component Testing		
Integration Testing		

3. Разработать композицию тестов для первой поставки программного обеспечения (build 1), состоящей из трех модулей (модуль 1, модуль 2, модуль 3).

4. Разработать композицию тестов для второй поставки программного обеспечения (build 2): исправлены заведенные дефекты, доставлена новая функциональность – модуль 4.

5. Разработать композицию тестов для третьей поставки программного обеспечения (build 3): заказчик решил расширять рынки сбыта и просит осуществить поддержку программного обеспечения на английском языке.

6. Разработать композицию тестов для четвертой поставки программного обеспечения (build 4): заказчик хочет убедиться, что программное обеспечение выдержит нагрузку в 2000 пользователей.

7. Оформить отчет

Контрольные вопросы:

1. Что такое тестирование?
2. Что такое качество программного обеспечения?
3. Что такое дефект?
4. Назовите три условия обнаружения дефекта.
5. Какие существуют виды тестирования в зависимости от объекта тестирования? Дайте характеристику каждому.
6. Какие существуют виды функционального тестирования? Дайте характеристику каждому.
7. Какие существуют виды нефункционального тестирования? Дайте характеристику каждому.
8. Какие существуют виды тестирования в зависимости от глубины покрытия? Дайте характеристику каждому.
9. Какие существуют тестовые активности? Дайте характеристику каждому.
10. Какие существуют виды тестирования в зависимости от знания кода?

Практическая работа 2. Выполнение тестовых процедур на тестовых данных

Цель: выявлить и описать пользовательские требования в виде вариантов использования (Use Cases).

План занятия:

1. Изучить теоретические сведения.
2. Выполнить практическое задание по лабораторной работе.
3. Оформить отчёт и ответить на контрольные вопросы.

Теоретические сведения

Требование (Requirement) – описание того, какие функции и с соблюдением каких условий должен выполнять программный продукт в процессе решения полезной для пользователя задачи.

Значение требований:

- Позволяют понять, что и с соблюдением каких условий система должна делать.
- Предоставляют возможность оценить масштаб изменений и управлять изменениями.
- Являются основой для формирования плана проекта (в том числе плана тестирования).
- Помогают предотвращать или разрешать конфликтные ситуации.
- Упрощают расстановку приоритетов в наборе задач.
- Позволяют объективно оценить степень прогресса в разработке проекта.

Работа над требованиями включает следующие этапы: выявление требований, анализ требований (моделирование бизнес-процессов, прототипирование интерфейсов, приоритезация требований, результат этапа – визуализация требований), документирование требований (результат этапа – спецификация), тестирование (валидация) требований. Работу с требованиями на этапах выявления, анализа, документирования, как правило, выполняет бизнесаналитик. Тестирование требований выполняет тестировщик.

В иерархии требований существует три уровня: уровень бизнес-требований, уровень пользовательских требований, уровень продуктных требований (функциональные и нефункциональные требования).
Бизнес-требования выражают цель, ради которой разрабатывается продукт (зачем он нужен, какая от него ожидается польза).

Пользовательские требования описывают задачи, которые пользователь может выполнять с помощью разрабатываемой системы, и по своей сути представляют собой недетализированные функциональные требования. Поскольку здесь уже появляется описание поведения системы, требования этого уровня могут быть использованы для оценки объёма работ, стоимости проекта, времени разработки. Пользовательские требования оформляются в виде вариантов использования (Use Cases), пользовательских историй (User Stories), пользовательских сценариев (User Scenarios).
Функциональные требования описывают поведение системы, т.е. её действия (вычисления, преобразования, проверки, обработку и т.д.). Нефункциональные требования описывают свойства системы (удобство использования, безопасность, надёжность, расширяемость и т.д.), которыми она должна обладать при реализации своего поведения. Выявление и описание требований: Use Case.
Вариант использования (Use Case) продукта описывает последовательность взаимодействия системы и внешнего действующего лица. Действующим лицом может быть человек, другая система ПО или аппаратное устройство, взаимодействующее с системой для достижения некой цели.
Варианты использования меняют традиционный подход к сбору информации: пользователей не спрашивают, что с их точки зрения, должна делать система, а выясняют, какие задачи собирается с ее помощью решать пользователь. Цель такого подхода — описать все подобные задачи. До включения каждого варианта использования в утвержденную версию требований заинтересованные в проекте лица проверяют, не выходит ли он за границы проекта. Теоретически в конечный набор вариантов использования должна входить вся желаемая функциональность системы.

Описание варианта использования включает следующие категории:

- уникальный идентификатор;
- имя, кратко описывающее задачи пользователи в формате «глагол + объект», например «разместить заказ»;
- краткое текстовое описание на естественном языке;
- список предварительных условий, которые должны быть удовлетворены до начала разработки варианта использования;
- выходные условия, описывающие состояние системы после успешного завершения разработки варианта использования;
- пронумерованный список действий, иллюстрирующий последовательность этапов взаимодействия лица и системы от предварительных условий до выходных условий. Пример варианта использования приведен в таблице 2.1.

Таблица 2.1 – Пример варианта использования

Идентификатор и название:	UC-4 Запросить химикат
Основное действующее лицо:	Сотрудник, разместивший заказ на химикат

Описание:	Сотрудник, разместивший заказ на химикат, указывает в запросе необходимый химикат, вводя его название или идентификатор или импортируя его структуру из соответствующего графического средства. Система выполняет запрос, предлагая контейнер с химикатом со склада или позволяя создать запрос на заказ у поставщика.
Триггер:	Сотрудник указывает, что хочет заказать химикат.
Предварительные условия:	PRE-1. Личность пользователя аутентифицирована. PRE-2. Пользователь имеет право запрашивать химикаты. PRE-3. База данных по запасам химикатов в данный момент доступна.
Выходные условия:	POST-1. Запрос сохраняется в Chemical Tracking System. POST-2. Запрос отправлен на склад химикатов или поставщику.
Нормальное направление развития варианта использования:	4.0 Запросить химикат со склада 1. Сотрудник указывает требуемый химикат. 2. Система перечисляет контейнеры с необходимым химикатом, имеющиеся на складе. 3. Сотрудник может просмотреть историю любого контейнера. 4. Сотрудник выбирает определенный контейнер или просит отправить запрос поставщику (см. 4.1). 5. Сотрудник вводит остальную информацию, чтобы завершить запрос. 6. Система сохраняет запрос и отправляет его на склад химикатов.

Окончание таблицы 2.1

Альтернативное направление развития варианта использования:	4.1 Запросить химикат у поставщика 1. Сотрудник ищет химикат по каталогам поставщика (см. 4.1.E1). 2. Система отображает список поставщиков, где также указаны размеры, класс и цена контейнеров. 3. Сотрудник выбирает поставщика, размер, класс и количество контейнеров. 4. Сотрудник вводит остальную информацию, необходимую для запроса. 5. Система сохраняет запрос и перенаправляет его поставщику.

Исключения:	4.1.E1. Химиката нет в продаже 1. Система отображает сообщение «У поставщиков нет такого химиката». 2. Система предлагает сотруднику запросить другой химикат или выйти из программы. 3а. Сотрудник просит запросить другой химикат. 4а. Система заново начинает нормальное направление варианта использования. 3б. Сотрудник решает выйти из системы. 4б. Система завершает вариант использования.
Бизнес-правила:	BR-28, BR-31

Существует несколько сценариев варианта использования (см. таблицу 2.1).
Один сценарий считается нормальным направлением развития (normal course) варианта использования, его также называют основным направлением, главным успешным сценарием и благоприятным путем. Нормальное направление для варианта использования «Запрос химиката» — запрос химиката, который есть на складе.
Другие допустимые сценарии из варианта использования, называются альтернативными направлениями (alternative courses) или вторичными сценариями (secondary scenarios). Они также могут привести к успешному выполнению задания и удовлетворяют выходным условиям варианта использования. Однако они представляют вариации решения задачи или диалоговой последовательности, необходимой для выполнения задачи. В определенной точке принятия решений в диалоговой последовательности нормальное направление может перейти в альтернативное, а затем вернуться обратно в нормальное.
Условия, препятствующие успешному завершению задания, называются исключениями (exceptions). Если в процессе сбора информации не указано, как обрабатывать исключение, то возможны два пути: 1) разработчики предложат лучший по их мнению способ обработки исключений; 2) при генерации пользователем неверного условия произойдет сбой системы, так как никто не предусмотрел такой ситуации. Иногда исключения рассматриваются как тип альтернативного направления, однако эти понятия следует разделять. Не обязательно реализовывать каждое альтернативное направление, которое определяют для варианта использования; кроме того, можно отложить его реализацию до следующего выпуска. Однако необходимо реализовать исключения, из-за которых завершение сценариев может оказаться неуспешным.

Расширение (extend) и включение (include).

При составлении вариантов использования часто можно столкнуться с ситуацией, когда альтернативное направление варианта использования само по себе можно выделить в автономный вариант использования. В таком случае можно расширить (extention) нормальное направление, включив этот отдельный вариант использования в нормальный поток.
Пример: Вариант использования "Запросить химикат" может включать в себя поиск по каталогу поставщика. Но при этом запросить химикат можно и без поиска по каталогам, а поиск по каталогу может выполняться как отдельная бизнес-задача пользователей. Поэтому логично расширить вариант использования "Запросить химикат" отдельным вариантом использования "Поиск по каталогам поставщика". Такая связь будет означать, что при выполнении варианта использования "Запросить химикат" может выполняться, а может и не выполняться вариант использования "Поиск по каталогам поставщика.
Иногда же несколько вариантов использования имеют общие наборы этапов. Чтобы избежать повторения этих этапов в каждом варианте использования, можно определить отдельный вариант использования и указать, что он включен (include) в другие варианты использования как подвариант.

Пример: Если покупатель совершает покупку товара, то он обязательно должен его оплатить. Нет случая, когда покупатель просто за что-то платит (т.е. оплата товара не является отдельной независимой деятельностью покупателя), но при этом процесс оплаты сам по себе достаточно сложный, включающий различные шаги и альтернативные варианты (оплата различными способами). Поэтому логично его выделить в отдельный вариант использования, при этом включить в вариант использования "Покупка товара". Такая связь будет означать, что при выполнении варианта использования "Купить товар" обязательно будет задействован и вариант использования "Оплатить товар".

Определение вариантов использования.

Определить варианты использования можно несколькими способами:

- сначала определить действующие лица, а затем бизнес-процессы, в которых каждое лицо участвует;
- выразить бизнес-процессы в терминах определенных сценариев, обобщить сценарии в варианты использования и определить действующие лица для каждого варианта; определить внешние события, на которые система должна реагировать, а затем соотнести эти события с участвующими лицами и определенными вариантами использования;
- определить вероятные варианты использования на основе функциональных требований; если какие-либо требования невозможно проследить до какого-либо варианта использования, необходимо задуматься, нужны ли они.

Как правило, пользователи сначала определяют самые важные варианты использования, поэтому порядок предлагаемых тем позволит получить представление о приоритетах.

Преимущества применения вариантов использования состоят в том, что каждый вариант сосредоточен на поставленной задаче и пользователе. Тщательное изучение этапов взаимодействия лица и системы помогает еще на ранних стадиях разработки выявить неясности и неточности, а также позволяет составить варианты тестирования на основе вариантов использования. Способ с применением вариантов использования позволяет выявить функциональные требования, с помощью которых пользователи будут выполнять конкретные задачи. Кроме прочего варианты использования облегчают расстановку приоритетов требований. Высшим приоритетом обладают те функциональные требования, которые созданы на основе вариантов использования с высшим приоритетом. Высший приоритет назначается по следующим причинам:

- варианты использования описывают один из основных бизнес-процессов, активизируемых системой;
- многие пользователи часто обращаются к ним;
- их запросил привилегированный класс пользователей;
- они предоставляют возможности, необходимые для соответствия требованиям;
- функции других систем зависят от их наличия.

Существуют также и преимущества технического характера. С помощью варианта использования можно выявить некоторые важные объекты предметной области и их взаимоотношения. Разработчики, использующие объектно-ориентированные методы проектирования, могут преобразовать варианты использования в объектные модели, такие, как диаграммы классов и диаграммы последовательностей.

Практическое задание:

1. Получить у преподавателя задание, содержащее идею и бизнес-цели подлежащего разработке программного продукта.
2. Определить действующие лица и сформулировать наиболее вероятные варианты использования подлежащего разработке программного продукта.
3. Полностью описать три варианта использования подлежащего разработке программного продукта.

4. Для каждого варианта использования указать уникальный идентификатор; имя в формате «глагол + объект»; краткое текстовое описание; предварительные условия; выходные условия; пронумерованный список действий нормального направления развития.

5. Для каждого варианта использования при необходимости указать пронумерованный список действий альтернативного направления (направлений) развития.

6. Для каждого варианта использования при необходимости указать исключения.

7. Оформить отчет и защитить лабораторную работу.

Содержание отчета:

1. Цель работы.
2. Описание вариантов использования подлежащего разработке программного продукта.
3. Выводы по работе.

Контрольные вопросы:

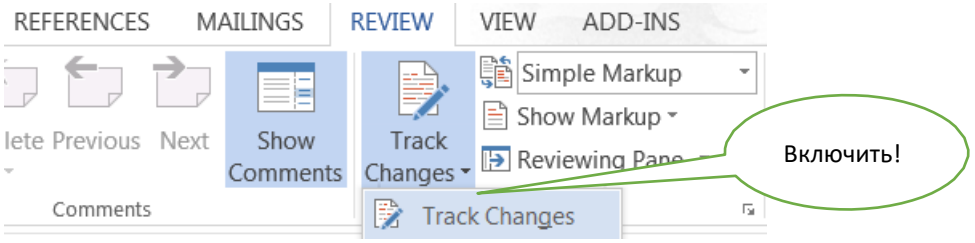
1. Что такое требование?
2. Какие значения имеют требования на проекте?
3. Какие существуют этапы работы над требованиями?
4. Кто выполняет работу с требованиями?
5. Какие существуют уровни требований?
6. Что такое вариант использования?
7. Для чего нужен вариант использования?
8. Какие элементы входят в состав описания варианта использования?
9. Что такое основной сценарий варианта использования?
10. Что такое альтернативный сценарий варианта использования?

Практическая работа 3. Анализ полученных результатов тестирования

Изменение формата файла и документа. По какой-то непонятной причине очень многие начинающие тестировщики стремятся полностью уничтожить исходный документ, заменив текст таблицами (или наоборот), перенеся данные из Word в Excel и т.д. Это можно сделать только в одном случае: если вы предварительно договорились о подобных изменениях с автором документа. В противном случае вы полностью уничтожаете чью-то работу, делая дальнейшее развитие документа крайне затруднительным.

Самое худшее, что можно сделать с документом, — это сохранить его в итоге в некоем формате, предназначенном скорее для чтения, чем для редактирования (PDF, набор картинок и тому подобное).

Если требования изначально создаются в некоей системе управления требованиями, этот вопрос неактуален, но высокоуровневые требования большинство заказчиков привыкли видеть в обычном DOCX-документе, а Word предоставляет



такие прекрасные возможности работы с документом, как отслеживание изменений (см. рисунок 2.2.i) и комментарии (см. рисунок 2.2.j).

Рисунок 2.2.i — Активация отслеживания изменений в Word

В итоге получается результат, представленный на рисунке 2.2.j: исходный формат сохраняется (а автор к нему уже привык), все изменения хорошо видны и могут быть приняты или отклонены в пару кликов мыши, а типичные часто повторяющиеся вопросы вы можете помимо указания в комментариях вынести в отдельный список и поместить его в том же документе.

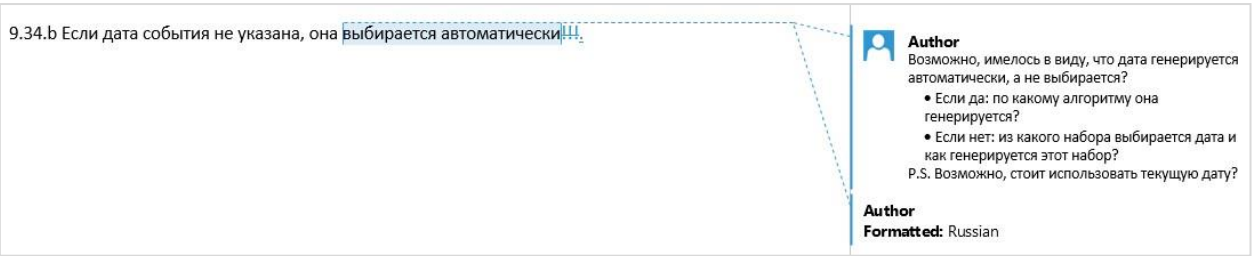


Рисунок 2.2.j — Правильно выглядящий документ с правками

И ещё два маленьких, но неприятных момента относительно таблиц:

- Выравнивание ВСЕГО текста в таблице по центру. Да, выравнивание по цен- тру хорошо смотрится в заголовках и ячейках с парой-тройкой слов, но если так выровнен весь текст, читать его становится сложно.
- Отключение границ ячеек. Такая таблица намного хуже читается.

Отметка того факта, что с требованием всё в порядке. Если у вас не возникло вопросов и/или замечаний к требованию — не надо об этом писать. Любые пометки в документе подсознательно воспринимаются как признак проблемы, и такое «одобрение требований» только раздражает и затрудняет работу с документом — сложнее становится заметить пометки, относящиеся к проблемам.

Описание одной и той же проблемы в нескольких местах. Помните, что ваши пометки, комментарии, замечания и вопросы тоже должны обладать свойствами хороших требований (настолько, насколько эти свойства к ним применимы). Если вы много раз в разных местах пишете одно и то же об одном и том же, вы нарушаете как минимум свойство модифицируемости. Постарайтесь в таком слу- чае вынести ваш текст в конец документа, укажите в нём же (в начале) перечень пунктов требований, к которым он относится, а в самих требованиях в коммента- риях просто ссылайтесь на этот текст.

Написание вопросов и комментариев без указания места требования, к которым они относятся. Если ваше инструментальное средство позволяет ука- зать часть требования, к которому вы пишете вопрос или комментарий, сделайте это (например, Word позволяет выделить для комментирования любую часть тек- ста — хоть один символ). Если это невозможно, цитируйте соответствующую часть текста. В противном случае вы порождаете неоднозначность или вовсе делаете вашу пометку бессмысленной, т.к. становится невозможно понять, о чём вообще идёт речь.

Задавание плохо сформулированных вопросов. Эта ошибка была подробно рассмотрена выше (см. раздел «Техники тестирования требований» и таблицу 2.2.a. Однако добавим, что есть ещё три вида плохих вопросов:

- Первый вид возникает из-за того, что автор вопроса не знает общепринятой терминологии или типичного поведения стандартных элементов интерфейса (например, «что такое чек-бокс?», «как в списке можно выбрать несколько пунктов?», «как подсказка может всплывать?»).
- Второй вид плохих вопросов похож на первый из-за формулировок: вместо того, чтобы написать «что вы имеете в виду под {чем-то}?», автор вопроса пишет «что такое {что-то}?» То есть вместо вполне логичного уточнения по- лучается ситуация, очень похожая на рассмотренную в предыдущем пункте.
- Третий вид сложно привязать к причине возникновения, но его суть в том,

что к некорректному и/или невыполнимому требованию задаётся вопрос наподо- бие «что будет, если мы это сделаем?». Ничего не будет, т.к. мы это точно не сделаем. И вопрос должен быть совершенно иным (каким именно — зави- сит от конкретной ситуации, но точно не таким).

И ещё раз напомним о точности формулировок: иногда одно-два слова могут на корню уничтожить отличную идею, превратив хороший вопрос в плохой. Срав- ните: «Что такое формат даты по умолчанию?» и «Каков формат даты по умолча- нию?». Первый вариант просто показывает некомпетентность автора вопроса, то- гда как второй — позволяет получить полезную информацию.

К этой же проблеме относится непонимание контекста. Часто можно увидеть вопросы в стиле «о каком приложении идёт речь?», «что такое система?» и им по- добные. Чаще всего автор таких вопросов просто вырвал требование из контекста, по которому было совершенно ясно, о чём идёт речь.

Написание очень длинных комментариев и/или вопросов. История знает случаи, когда одна страница исходных требований превращалась в 20–30 страниц текста анализа и вопросов. Это плохой подход. Все те же мысли можно выразить значительно более кратко, чем сэкономить как своё время, так и время автора ис- ходного документа. Тем более стоит учитывать, что на начальных стадиях работы с требованиями они весьма нестабильны, и может получиться так, что ваши 5–10 страниц комментариев относятся к требованию, которое просто удалят или изменят до неузнаваемости.

Критика текста или даже его автора. Помните, что ваша задача — сделать требования лучше, а не показать их недостатки (или недостатки автора). Потому комментарии вида «плохое требование», «неужели вы не понимаете, как глупо это звучит», «надо переформулировать» неуместны и недопустимы.

Категоричные заявления без обоснования. Как продолжение ошибки «критика текста или даже его автора» можно отметить и просто категоричные заяв- ления наподобие «это невозможно», «мы не будем этого делать», «это не нужно». Даже если вы понимаете, что требование бессмысленно или невыполнимо, эту мысль стоит сформулировать в корректной форме и дополнить вопросами, позво- ляющими автору документа самому принять окончательное решение. Например, «это не нужно» можно переформулировать так: «Мы сомневаемся в том, что дан- ная функция будет востребована пользователями. Какова важность этого требова- ния? Уверены ли вы в его необходимости?»

Указание проблемы с требованиями без пояснения её сути. Помните, что автор исходного документа может не быть специалистом по тестированию или биз- нес-анализу. Потому просто пометка в стиле «неполнота», «двусмысленность» и т.д. могут ничего ему не сказать. Поясняйте свою мысль.

Сюда же можно отнести небольшую, но досадную недоработку, относящуюся к противоречивости: если вы обнаружили некие противоречия, сделайте соответ- ствующие пометки во всех противоречащих друг другу местах, а не только в одном из них. Например, вы обнаружили, что требование 20 противоречит требованию 30. Тогда в требовании 20 отметьте, что оно противоречит требованию 30, и наоборот. И поясните суть противоречия.

Плохое оформление вопросов и комментариев. Старайтесь сделать ваши вопросы и комментарии максимально простыми для восприятия. Помните не только о краткости формулировок, но и об оформлении текста (см., например, как на рисунке 2.2.j вопросы структурированы в виде списка — такая структура воспри- нимается намного лучше, чем сплошной текст). Перечитайте свой текст, исправьте опечатки, грамматические и пунктуационные ошибки и т.д.

Описание проблемы не в том месте, к которому она относится. Класси- ческим примером может быть неточность в сноске, приложении или рисунке, кото- рая почему-то описана не там, где она находится, а в тексте, ссылающемся на со- ответствующий элемент. Исключением может считаться противоречивость, при ко-

торой описать проблему нужно в обоих местах.

Ошибочное восприятие требования как «требования к пользователю». Ранее (см. «Корректность» в «Свойства качественных требований») мы говорили, что требования в стиле «пользователь должен быть в состоянии отправить сооб- щение» являются некорректными. И это так. Но бывают ситуации, когда проблема намного менее опасна и состоит только в формулировке. Например, фразы в стиле

«пользователь может нажать на любую из кнопок», «пользователю должно быть видно главное меню» на самом деле означают «все отображаемые кнопки должны быть доступны для нажатия» и «главное меню должно отображаться». Да, эту недо- работку тоже стоит исправить, но не следует отмечать её как критическую проблему.

Скрытое редактирование требований. Эту ошибку можно смело отнести к разряду крайне опасных. Её суть состоит в том, что тестировщик произвольно вносит правки в требования, никак не отмечая этот факт. Соответственно, автор документа, скорее всего, не заметит такой правки, а потом будет очень удивлён, когда в продукте что-то будет реализовано совсем не так, как когда-то было описано в требованиях. Потому простая рекомендация: если вы что-то правите, обязательно отмечайте это (средствами вашего инструмента или просто явно в тексте). И ещё лучше отмечать правку как предложение по изменению, а не как свершившийся факт, т.к. автор исходного документа может иметь совершенно иной взгляд на си- туацию.

Анализ, не соответствующий уровню требований. При тестировании требований следует постоянно помнить, к какому уровню они относятся, т.к. в противном случае появляются следующие типичные ошибки:

- Добавление в бизнес-требования мелких технических подробностей.
- Дублирование на уровне пользовательских требований части бизнес-требо- ваний (если вы хотите увеличить прослеживаемость набора требований, имеет смысл просто использовать ссылки).
- Недостаточная детализация требований уровня продукта (общие фразы, до- пустимые, например, на уровне бизнес-требований, здесь уже должны быть предельно детализированы, структурированы и дополнены подробной тех- нической информацией).

Практическая работа 4. Проверка компонентов инструментария и тестируемого приложения на корректное начальное состояние для начала тестирования

Практическая работа 5. Составление отчета по выполнению рабочего задания

Цель: составить итоговый отчет о результатах тестирования web приложения.

План занятия:

1. Изучить теоретические сведения.
2. Выполнить практическое задание по лабораторной работе.
3. Оформить отчёт и ответить на контрольные вопросы.

Теоретические сведения

Итоговый отчет о качестве проверенного функционала является неотъемлемой частью работы, которую каждый тестировщик должен выполнить по завершению тестирования.

Итоговый отчет можно разделить на части с соответствующей информацией:

1. Общая информация.
2. Сведения о том, кто и когда тестировал программный продукт.
3. Тестовое окружение.
4. Общая оценка качества приложения.

5. Обоснование выставленного качества.
6. Графическое представление результатов тестирования.
7. Детализированный анализ качества по модулям.
8. ТОП-5 самых критичных дефектов.
9. Рекомендации.

Пример итогового отчета приведен на рисунке 6.1.

Далее рассмотрим подробно каждую часть итогового отчета.

Общая информация включает:

- название проекта,
- номер сборки,
- модули, которые подверглись тестированию (в случае, если тестировался не весь проект),
- виды тестов по глубине покрытия (Smoke Test, Minimal Acceptance Test, Acceptance Test), тестовые активности (New Feature Test, Regression Testing, Defect Validation),
- количество обнаруженных дефектов,
- вид рабочей тестовой документации (Acceptance Sheet, Test Survey, Test Cases).

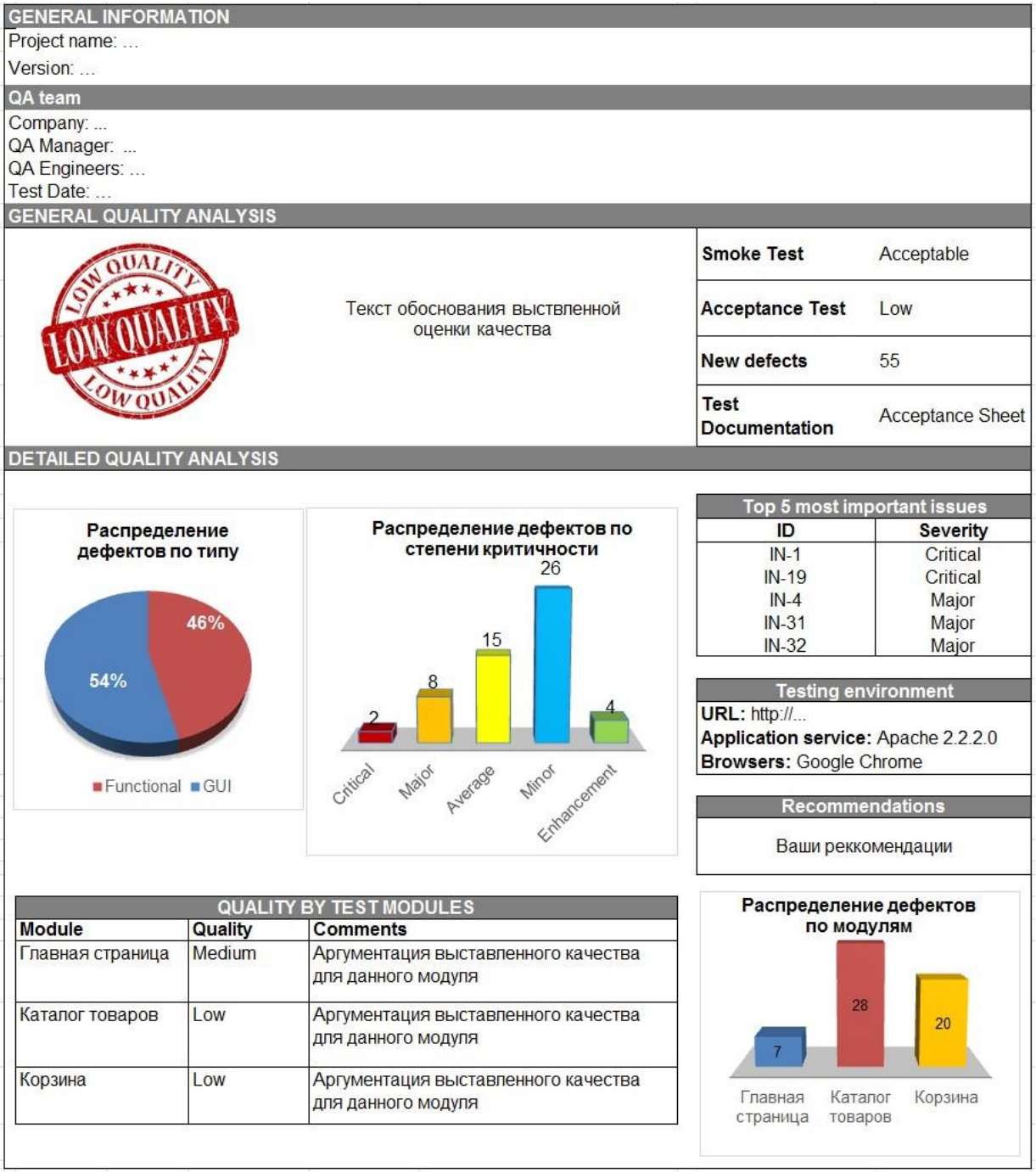


Рисунок 6.1 – Пример итогового отчета о результатах тестирования

Сведения о том, кто и когда тестировал программный продукт, включают информацию о команде тестирования с указанием контактных данных и временном интервале тестирования.

Тестовое окружение содержит: ссылку на проект, брарузер, операционную систему и другую информацию, конкретизирующую особенности конфигурации.

Общая оценка качества приложения выставляется на основании общего впечатления от работы с приложением и внесенных дефектов (количество, важность). Обязательно учитывается этап разработки проекта – то, что не критично в начале работы, становится важным при выпуске программного продукта. Уровни качества: Высокое (High), Среднее (Medium), Низкое (Low).

Обоснование выставленного качества является наиболее важной частью отчета, т.к. здесь отражается общее состояние сборки, а именно:

- качество сборки на текущий момент,

- факторы, повлиявшие на выставление именно такого качества сборки: указание функционала, который заблокирован для проверки, перечисление наиболее критичных дефектов и объяснение их важности для пользователя или бизнеса заказчика,
- анализ качества проверенного функционала: улучшилось оно или ухудшилось по сравнению с предыдущей версией,
- если качество сборки ухудшилось, то обязательно должны быть указаны регрессионные места,
- наиболее нестабильные части функционала с указнием причин, по которым они таковыми являются.

В данном разделе показывается аналитическая работа тестировщика, наиболее слабые места и наиболее критичные дефекты, динамика изменения качества проекта.
Графическое представление результатов тестирования способствует более полному и быстрому пониманию текстовой информации (см. рисунок 4.1).
Если необходимо продемонстрировать процентное соотношение, то целесообразно использовать круговые диаграммы (например, процентное соотношение функциональных дефектов и дефектов GUI).
Столбчатые диаграммы лучше подойдут там, где важно визуализировать количество дефектов в зависимости от степени их критичности или в зависимости от локализации (распределение дефектов по модулям).
Отразить в итоговом отчёте динамику качества по всем сборкам лучше всего удастся с помощью линейного графика.

Детализированный анализ качества по модулям.

В данной части отчета описывается более подробная информация о проверенных частях функционала, устанавливается качество каждой проверенной части функционала (модуля) в отдельности, дается аргументация выставленного уровня качества. Как правило, данный раздел отчета представляется в табличной форме. В зависимости от вида проводимых тестовых активностей, эта часть отчета будет отличаться.
При оценке качества функционала на уровне Smoke теста, оно может быть либо Приемлемым (Acceptable), либо Неприемлемым (Unacceptable). Если все наиболее важные функции работают корректно, то качество всего функционала на уровне Smoke может быть оценено, как Примлемое.
Если это релизная или предрелизная сборка, то для выставления Приемлемого качества на уровне Smoke не должно быть найдено функциональных дефектов.
В части о детализированной информации качества сборки следует более подробно описать проблемы, которые были найдены во время теста.
При оценке качества функционала на уровне Defect Validation указывается качество о проведении валидации дефектов, а именно:

- Общее количество всех дефектов, поступивших на проверку.
- Количество неисправленных дефектов и их процент от общего количества.
- Список дефектов, которые не были проверены и причины, по которым этого не было сделано.
- Наглядная таблица с неисправленными дефектами.

По вышеуказанным результатам выставляется качество теста. Если процент неисправленных дефектов < 10%, то качество Приемлемое (Acceptable), если > 10%, то качество Неприемлемое (Unacceptable).
При оценке качества функционала на уровне New Feature Test (полный тест нового функционала) качество отдельно проверенного функционала может быть: Высокое (High), Среднее (Medium), Низкое (Low).
Важно отдельно указывать информацию о качестве каждого модуля нового функционала с аргументацией выставленной оценки.
При оценке качества функционала на уровне Regression Testing нужно анализировать динамику изменения качества проверенной функциональности в сравнении с более ранними версиями сборки. Для этого приводится сравнительная характеристика каждой из частей функционала в сравнении с предыдущими версиями сборки, даются ясные

пояснения о выставлении соответствующего качества каждой функции в отдельности. Также как и у предыдущего вида тестов, качество этих может быть: Высокое (High), Среднее (Medium), Низкое (Low).
ТОП-5 самых критичных дефектов содержит список ссылок на наиболее критичные дефекты с указанием их названия и уровня критичности.
Рекомендации включают краткую информацию о всех проблемах, характерных сборке, с пояснениями, насколько оставшиеся проблемы являются критичными для конечного пользователя. Обязательно указывают функционал и дефекты, скорейшее исправление которых является наиболее приоритетным. Кроме того, если сборка является релизной или предрелизной, то любое ухудшение качества является критичным и важно это обозначить.

Практическое задание:

1. Составить итоговый отчет по результатам тестирования web приложения.
2. Указать общую информацию о тестируемом продукте (название, номер сборки, виды выполненных тестов, количество обнаруженных дефектов, вид рабочей тестовой документации).
3. Указать, кто и когда тестировал программный продукт.
4. Описать тестовое окружение (ссылку на web приложение, браузер).
5. Указать общую оценку качества протестированного приложения и подробно ее обосновать.
6. Графически (в виде круговой диаграммы) отразить процентное соотношение дефектов GUI и функциональных дефектов.
7. Графически (в виде столбчатой диаграммы) отразить распределение дефектов по различным степеням критичности.
8. Графически (в виде столбчатой диаграммы) отразить распределение дефектов по модулям.
9. Произвести детальный анализ качества всех модулей протестированного приложения с аргументацией выставленных уровней качества.
10. Привести список пяти наиболее критичных дефектов.
11. Сформулировать рекомендации по улучшению качества программного продукта.
12. Оформить отчет и защитить лабораторную работу.

Содержание отчета:

1. Цель работы.
2. Итоговый отчет о результатах тестирования web приложения.
3. Выводы по работе.
4. Ответы на контрольные вопросы

Контрольные вопросы:

1. Какая структура итогового отчета о результатах тестирования?
2. Что содержится в разделе Общая информация?
3. Что содержится в разделе Тестовое окружение?
4. Как выставляется общая оценка качества приложения?
5. Как обосновать выставленную оценку качества?
6. Для чего используется графическое представление результатов тестирования в итоговом отчете?
7. Что содержится в разделе Детализированный анализ качества?
8. Что содержится в разделе Рекомендации?

Практическая работа 6. Система управления тестами

Цель работы: изучить проблематику создания сложной программной системы в отношении к разрабатываемой ИС.

Краткие теоретические сведения

Особенности разработки сложных (больших) программных систем. Из года в год увеличиваются разнообразие и сложность систем, получивших в международной научно-технической практике название систем, интенсивно использующих программное обеспечение

– Software Intensive Systems (SIS). В системах такого рода функциональный потенциал определяется программным обеспечением (ПО) или зависит от ПО в существенной мере. В таких системах программные компоненты взаимодействуют друг с другом и компонентами и подсистемами другой природы, датчиками, приборами и людьми, вовлеченными в процессы использования SIS. К числу SIS, например, относятся разнородные автоматизированные системы управления, встроенные бортовые транспортные системы, телекоммуникационные и корпоративные системы, в том числе и на базе web-сервисов. Для разработок SIS типичны крупномасштабные проекты – десятки или сотни разработчиков, месяцы или годы разработки, сотни тысяч или десятки миллионов долларов, комплектование из многочисленных разнородных подсистем, большая часть из которых включает программные системы. Не все программные системы сложны. Существует множество программ, которые задумываются, разрабатываются, сопровождаются и используются одним и тем же человеком. Обычно это начинающий программист или профессионал, работающий изолированно. Нельзя сказать, что все такие системы плохо сделаны или тем более усомниться в квалификации их создателей. Но такие системы, как правило, имеют очень ограниченную область применения и короткое время жизни. Обычно их лучше заменить новыми, чем пытаться повторно использовать, переделывать или расширять. Разработка подобных программ скорее утомительна, чем сложна, так что изучение этого процесса нас не интересует. Какого-либо одного формального признака, отличающего обычную программу от сложной, не существует. В целом сложные программы выгодно отличаются разнообразием предоставляемого сервиса и количеством обрабатываемой информации. Возможно обозначить лишь некоторые качественные характеристики, свойственные сложной программе. Сложная программа характеризуется также более сложным алгоритмом обработки событий. В частности, такая программа предполагает некоторую реакцию на вмешательство пользователя в управляемый процесс или объект. Существенно, что сложные программы предназначены для многократного использования и применения разными пользователями. В связи с этим следует обратить внимание на ряд необходимых свойств программного обеспечения.

Обычно сложная программа обладает следующими свойствами:

– программа решает одну или несколько связанных прикладных задач, зачастую сначала не имеющих четкой постановки и настолько важных для каких-либо лиц или организаций, что те приобретают значимые выгоды от ее использования;

– программа не предназначена для решения каких-либо прикладных задач, но от нее зависит эффективное решение этих прикладных задач. Это системные программы, например операционные системы, системы управления базами данных, различные инструментальные системы и т. п.;

– существенно, чтобы программа была удобной в использовании. В частности, она должна включать достаточно полную и понятную пользователям документацию, возможно, специальную документацию для администраторов, а также набор документов для обучения работе с программой;

– программа должна обладать высокой производительностью, высокой реактивностью или удовлетворять другим требованиям, в противном случае ее использование по назначению (на реальных данных) может привести к значимым для пользователей потерям;

– программа должна обладать высокой надежностью. Неправильная работа программы может нанести ощутимый ущерб пользователям и другим организациям и лицам, даже если сбои происходят не слишком часто;

– для выполнения своих задач программа должна удовлетворять требованиям совместимости, переносимости и интеграции с другими программами и программно-аппаратными системами и обеспечивать работу на разных платформах;

– пользователи, работающие с программой, могут приобретать дополнительные выгоды от того, что программа развивается, в нее вносятся новые функции и устраняются ошибки. Поэтому необходимо наличие проектной документации, позволяющей развивать ее, возможно, вовсе не тем разработчикам, которые ее создавали, без больших затрат на обратную разработку (реинжиниринг);

– в разработку программы вовлечено значительное количество людей

(десятки и сотни человек). Большую программу практически невозможно написать с первой попытки, с небольшими усилиями и в одиночку;

– большая программа имеет намного большее количество ее возможных пользователей по сравнению с небольшими программами и еще больше тех лиц, деятельность которых будет так или иначе затронута ее работой и результатами. Более подробно теоретические сведения и методики изложены в.

Контрольные вопросы

1. Что такое управление процессом разработки?
2. Что такое гибкость программного обеспечения?
3. Как описывается поведение программных систем?
4. Что такое сложность ПО?

Практическая работа 10. Программное обеспечение для управления тест-кейсами

Как будем искать систему?

Совсем недавно передо мной встала очень на вид простая задача - выбрать для небольшой компании (28 человек) систему управления тест кейсами. Поручили мне эту задачу в силу того, что в компании я пока один единственный тестировщик, а если правильнее и точнее сказать, то QA-engineer.

Итак, приступим, первое с чего стоит начать - это скорее всего Ютуб, посмотрим как работают большие компании, кто и что говорит о той или иной системе, почитаем статьи на Хабре, что и вам советую сделать, несмотря на подобные статьи, как эта.

В чем сложность выбора?

Во-первых система должна нравиться и удовлетворять вашим предпочтениям. Например: легкое и простое создание кейсов, гибкая настройка (кастомизация), удобство использования, определите для себя, какая система должна быть?

Во-вторых: в системе, скорее всего вы будете работать не один, у вас есть руководитель (тим лид), может несколько тестировщиков (джунов), как вы будете строить процессы взаимодействия с командой? Нужна система, которая сможет помочь отследить кто, когда, и что делал или редактировал, и, желательно систему предоставления визуальных отчетов - в каком состоянии проверки находятся на текущий момент времени? А также не стоит забывать про планирование тестирования (тест раны).

Ну и втретьих - это цена. Обычно она складывается из суммы баксов за 1 человеко-месяца.

TestLink. Плюсы открытое ПО. Минусы: морально и физически устарела, при установке могут возникнуть сложности, сложная или невозможная интеграция с Jira.

TXT, DOC, XLSX. Плюсы: Бесплатно, можно сразу планировать работу, можно создавать черновики тест-кейсов, использовать как архив или заготовки. Минусы: никакой интеграции, нет контроля и учета времени.

Testia Tarantula. Плюсы: бесплатная. Минусы: плохая отчетность, нет интеграции, нет кастомизации.

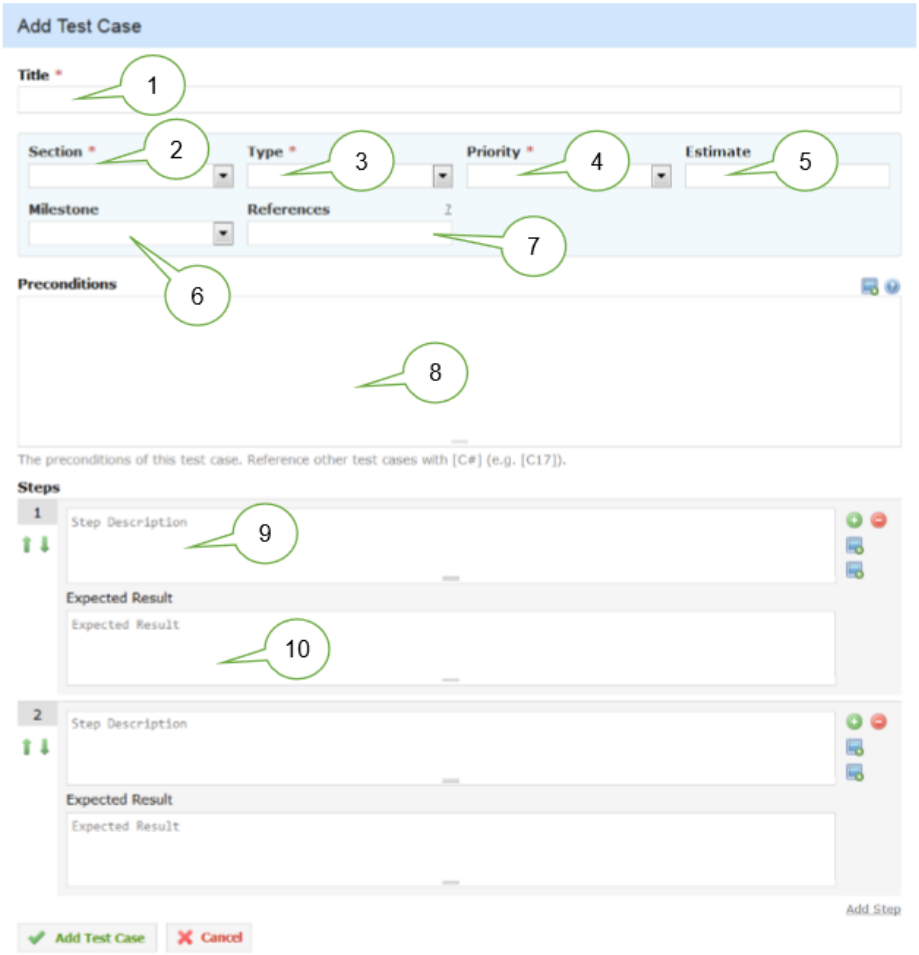
Qtest. Триал версия только под заказ. TestLodge. Будет отдельный абзац, см. ниже

Фавориты YouTube и статей: TestRail, Zephyr for Jira, QASE

После предварительного отбора выбор остановился на этих трех программах. Личное мнение: на первый взгляд, т.к. я изначально установил TestRail, мне он кажется «роднее» , но учитывая дальнейшее развитие и потребность в возможностях интеграции, более выгодным может оказаться Zephyr for Jira, также нельзя игнорировать такой бесплатный инструмент, как QASE, бесплатный (до 3 пользователей), интуитивно понятный интерфейс, детализированный отчет, вехи, и вообще ничем не уступающий вышеупомянутым системам. Давайте теперь каждую разберем по-подробнее.

TestRail

Инструментальным средством управления тест кейсами является TestRail.



Title (заглавие) здесь данное поле является обязательным для заполнения.

Section (секция) — очередная вариация на тему «Модуль» и «Подмодуль», позволяющая создавать иерархию секций, в которых можно размещать тест-кейсы.

Type (тип) здесь по умолчанию предлагает выбрать один из вариантов: automated (автоматизированный), functionality (проверка функциональности), performance (производительность), regression (регрессионный), usability (удобство использования), other (прочее).

Priority (приоритет) здесь представлен числами, по которым распределены следующие словесные описания: must test (обязательно выполнять), test if time (выполнять, если будет время), don’t test (не выполнять).

Estimate (оценка) содержит оценку времени, которое необходимо затратить на выполнение тест-кейса.

Milestone (ключевая точка) позволяет указать ключевую точку проекта, к которой данный тест-кейс должен устойчиво показывать положительный результат (выполняться успешно).

References (ссылки) позволяет хранить ссылки на такие артефакты, как требования, пользовательские истории, отчёты о дефектах и иные документы (требует дополнительной настройки).

Preconditions (приготовления) представляет собой классику описания предварительных условий и необходимых приготовлений к выполнению тест-кейса.

Step Description (описание шага) позволяет добавлять описание отдельного шага тест-кейса.

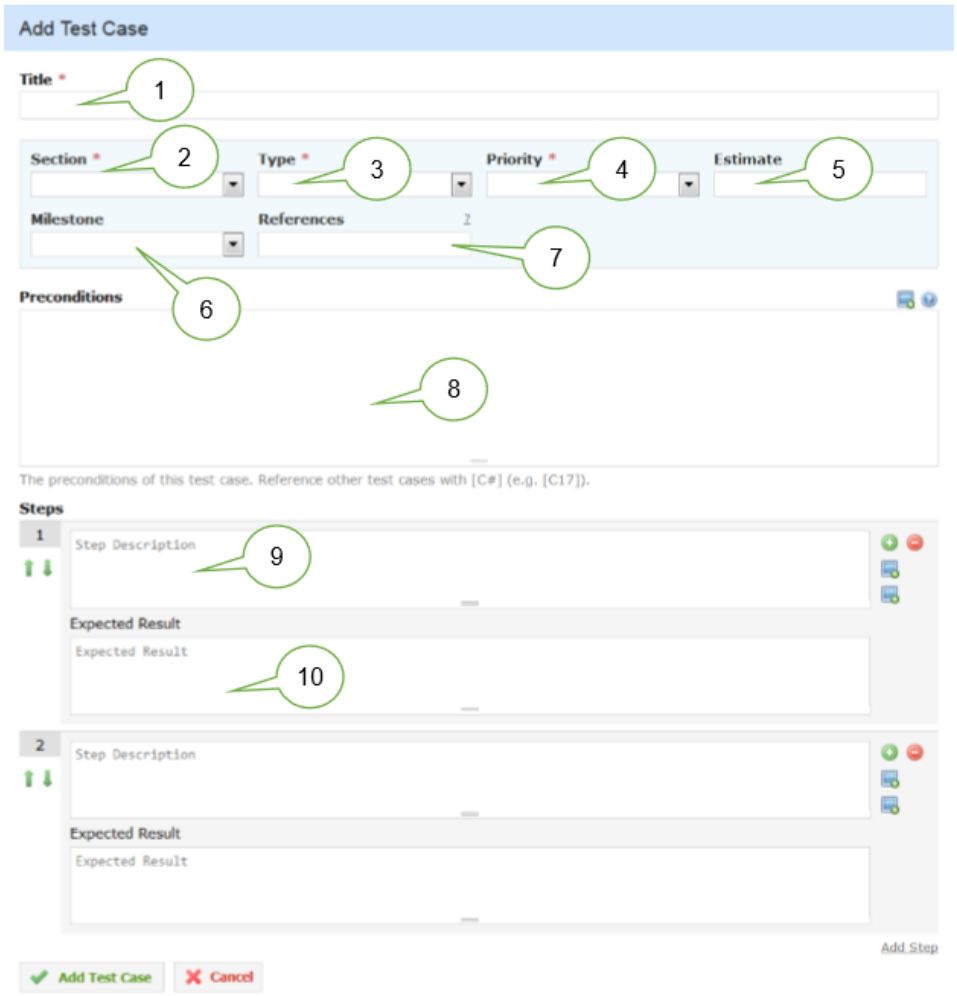
Expected Results (ожидаемые результаты) позволяет описать ожидаемый результат по каждому шагу.

Инструментом для управления тест кейсами является TestLink

Тестирование ПО

Программное обеспечение для управления тест-кейсами

Инструментальным средством управления тест кейсами является TestRail.



Title (заглавие) здесь данное поле является обязательным для заполнения.

Section (секция) — очередная вариация на тему «Модуль» и «Подмодуль», позволяющая создавать иерархию секций, в которых можно размещать тест-кейсы.

Type (тип) здесь по умолчанию предлагает выбрать один из вариантов: automated (автоматизированный), functionality (проверка функциональности), performance (производительность), regression (регрессионный), usability (удобство использования), other (прочее).

Priority (приоритет) здесь представлен числами, по которым распределены следующие словесные описания: must test (обязательно выполнять), test if time (выполнять, если будет время), don’t test (не выполнять).

Estimate (оценка) содержит оценку времени, которое необходимо затратить на выполнение тест-кейса.

Milestone (ключевая точка) позволяет указать ключевую точку проекта, к которой данный тест-кейс должен устойчиво показывать положительный результат (выполняться успешно).

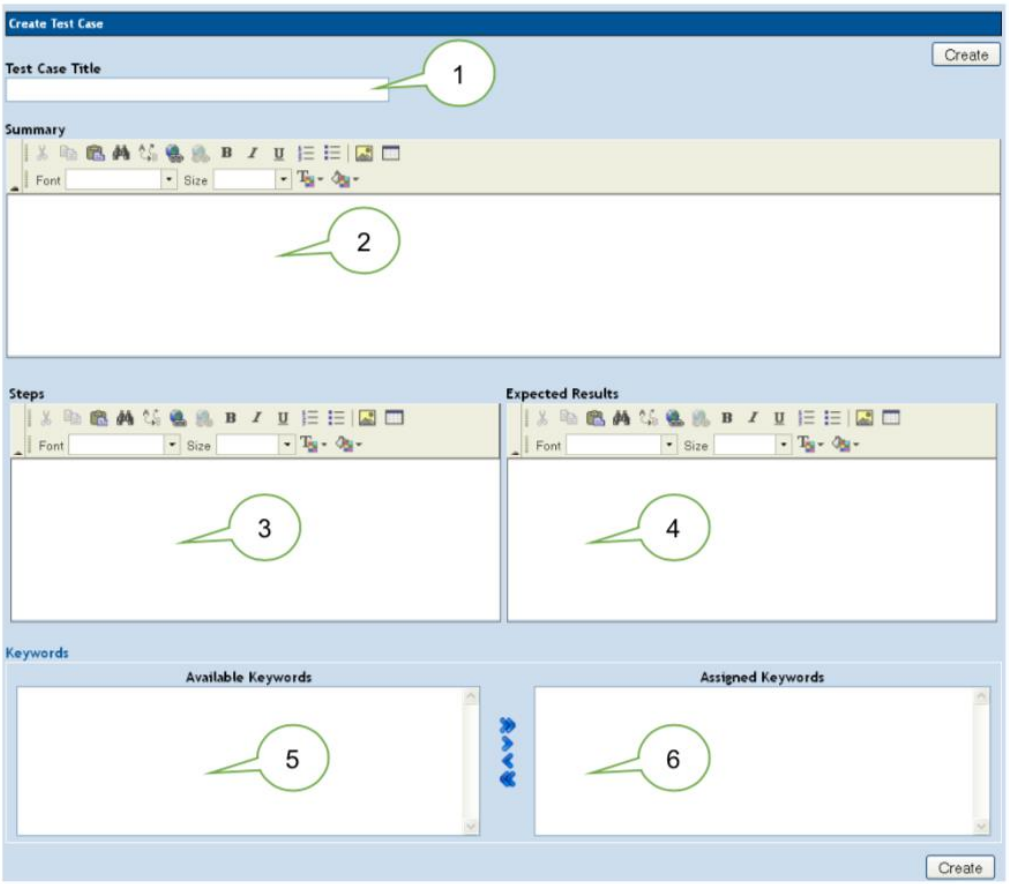
References (ссылки) позволяет хранить ссылки на такие артефакты, как требования, пользовательские истории, отчёты о дефектах и иные документы (требует дополнительной настройки).

Preconditions (приготовления) представляет собой классику описания предварительных условий и необходимых приготовлений к выполнению тест-кейса.

Step Description (описание шага) позволяет добавлять описание отдельного шага тест-кейса.

Expected Results (ожидаемые результаты) позволяет описать ожидаемый результат по каждому шагу.

Инструментом для управления тест кейсами является TestLink



Title (заглавие) здесь тоже является обязательным для заполнения.

Summary (описание) позволяет добавить любую полезную информацию о тест-кейсе (включая особенности выполнения, приготовления и т.д.).

Steps (шаги выполнения) позволяет описать шаги выполнения.

Expected Results (ожидаемые результаты) позволяет описать ожидаемые результаты, относящиеся к шагам выполнения.

Available Keywords (доступные ключевые слова) содержит список ключевых слов, которые можно проассоциировать с тест-кейсом для упрощения классификации и поиска тест-кейсов.Это ещё одна вариация идеи «Модулей» и «Подмодулей» (в некоторых системах реализованы оба механизма).

Assigned Keywords (назначенные ключевые слова) содержит список ключевых слов, проассоциированных с тест-кейсом.

JIRA + Zephyr

JIRA — это, главным образом, средство отслеживания ошибок, целью которого является контроль процесса разработки с задачами, ошибками и другими типами гибких карт.

Zephyr — один из многих плагинов JIRA, расширяющих возможности JIRA.

С помощью их комбинации Вы получите полный сервис, в соответствии с функциональностью инструментов управления тестированием:

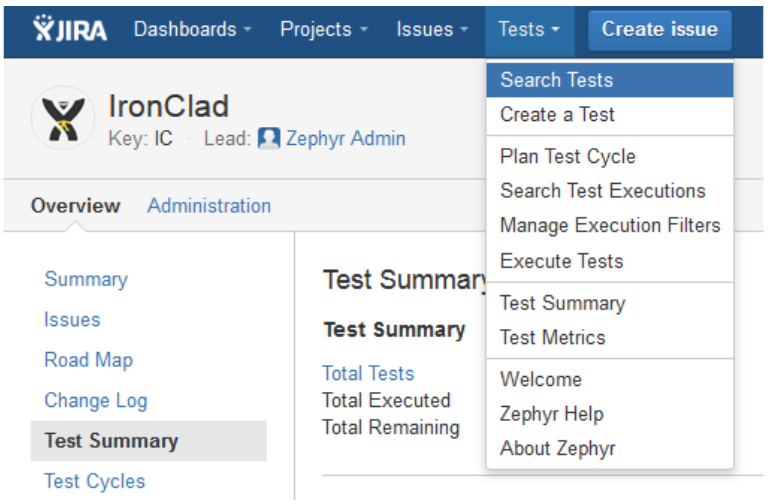
Создание тест-плана.

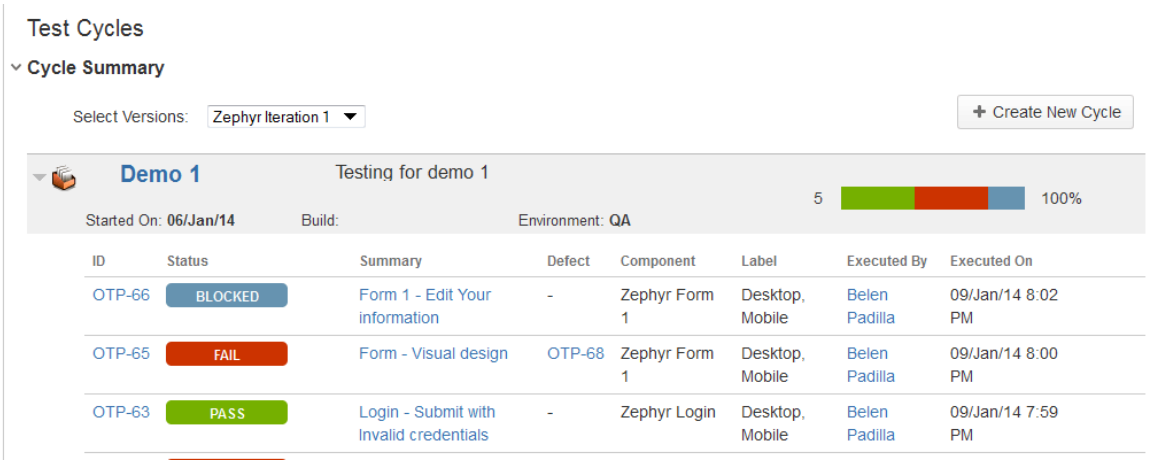
Описание тестовых случаев.

Выполнение тестирования.

Создание отчетов.

Если тестовый продукт ведет себя неправильно, вы можете немедленно создать отчет о ошибке.





Практическая работа 7. Система контроля дефектов

Цель. Изучить процессы тестирования и отладки программного обеспечения.

Оборудование. ПК

Ход работы

1. Ознакомиться с теоретической частью.
2. Выполнить практическое задание.
3. Ответить на контрольные вопросы.
4. Оформить отчет.

Теоретическая часть

Отладка – это процесс локализации и исправления ошибок, обнаруженных при тестировании программного обеспечения.

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. В целом сложность отладки обусловлена следующими причинами:

- 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:

- *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.

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 которыми была обнаружена ошибка. Метод очень эффективен, но не применим для больших программ, программ со сложными вычислениями и в тех случаях, когда ошибка связана с неверным представлением программиста о выполнении некоторых операций. Данный метод часто используют как составную часть других методов отладки.

Общая методика отладки программных продуктов, написанных для выполнения в операционных системах MS DOS и Win32:

1 этап – изучение проявления ошибки;

2 этап – определение локализации ошибки;

3 этап – определение причины ошибки;

4 этап – исправление ошибки;

5 этап – повторное тестирование.

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).

Спецификация программы, программная спецификация (program specification) - точная и полная формулировка определенной задачи или группы задач, содержащая сведения,

необходимые для построения алгоритма их решения. Содержит описание результата, который должен быть достигнут с помощью конкретной программы, а также действий, выполняемых программой для достижения конечного результата без упоминания того, как указанный результат достигается

Практическая часть

Задание 1. Запишите вариант в отчет.

Задание 2. Согласно поставленной задаче выполните ручную отладку:

- Опишите математическую модель задачи с указанием имен и назначения переменных;
- Опишите спецификацию программы;
- Запишите алгоритм программы;
- Выполните отладку логики программы методом «грубой силы» с помощью соседа;
- Составьте тестовые наборы для проверки функционала системы.

Задание 3. Результаты выполнения практического задания запишите в отчет.

Контрольные вопросы

1. Какие методы тестирования вы знаете?
2. В чем заключаются методы «черного» и «белого» ящика?
3. На каком этапе проводится ручная отладка?
4. Опишите методы отладки.

Варианты заданий

Создать Windows-приложение, реализующие линейный и разветвляющийся алгоритмы, которые размещены на разных вкладках окна формы. На вкладке линейного алгоритма предусмотреть поля ввода значений переменных и поле вывода результата вычисления. На вкладке разветвляющегося алгоритма предусмотреть поля для ввода значений переменных, поле вывода результатов расчета по одной из трех формул в зависимости от результата выполнения условия. В качестве *f(x)* использовать по выбору: *cos(x)* или x^2 или e^x. Пример рабочей формы представлен на рисунке 1.

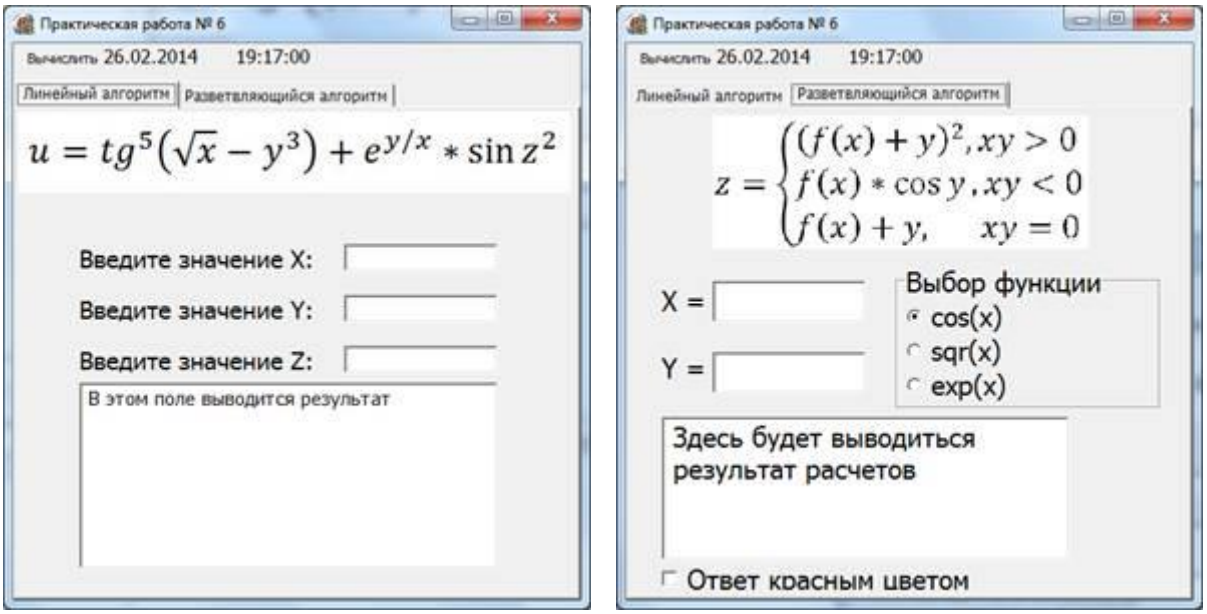


Рисунок 1 – Windows-приложение

Линейный алгоритм:

1. $t = \frac{2\cos\left(x-\frac{\pi}{6}\right)}{0.5+\sin^2 y}\left(1+\frac{z^2}{3-\frac{z^2}{5}}\right)$.

2. $u = \frac{\sqrt[3]{8+|x-y|^2+1}}{x^2+y^2+2} - e^{|x-y|} * (tg^2 z + 1)^x$.

3. $v = \frac{1+\sin^2(x+y)}{\left|x-\frac{2y}{1+x^2y^2}\right|} * x^{|y|} + \cos^2\left(arctg\frac{1}{z}\right)$.

4. $w = |\cos x - \cos y|^{1+2\sin^2 y} * \left(1 + z + \frac{z^2}{2} + \frac{z^3}{3} + \frac{z^4}{4}\right)$.

5. $\alpha = \ln\left(y^{-\sqrt{|x|}}\right) * \left(x - \frac{y}{2}\right) + \sin^2 arctg(z)$.

6. $\beta = \sqrt{10\left(\sqrt[3]{x} + x^{y+2}\right)} * (\arcsin^2 z - |x-y|)$.

7. $\gamma = 5arctg(x) - \frac{1}{4}\arccos(x) * \frac{x+3|x-y|+x^2}{|x-y|z+x^2}$.

Разветвляющийся алгоритм:

1. $a = \begin{cases} \ln(y+2) + f(x), & x/y > 0 \\ \ln|y| - \mathrm{tg}(f(x)), & x/y < 0 \\ f(x) * y^3, & \text{иначе} \end{cases}$

2. $b = \begin{cases} \ln|y + f(x)| +, & 3 < xy < 8 \\ \cos(f(x)) - y, & xy > 12 \\ \mathrm{sh}(f(x) + \mathrm{cs}(y)), & \text{иначе} \end{cases}$

3. $c = \begin{cases} f^3(x) + \mathrm{ctg}(y), & xy > 12 \\ \mathrm{sh}(f^3(x)) + y^2, & xy < 7 \\ \cos(x - f^3(x)), & \text{иначе} \end{cases}$

4. $d = \begin{cases} \mathrm{ctg}(y) + f(x), & x/y > 0 \\ \ln|y| + \mathrm{tg}(f(x)), & x/y < 0 \\ f(x) * y^3, & \text{иначе} \end{cases}$

5. $k = \begin{cases} (f(x)+y)^2, & 4 > xy > 1 \\ f(x) * \mathrm{tg}(y), & 8 < xy < 10 \\ f(x) + y, & \text{иначе} \end{cases}$

6. $l = \begin{cases} f^2(x) + \mathrm{arctg}(f(x)), & 1 \le x < 5 \\ (y - f(x))^2 + \mathrm{arctg}(f(x)), & y > x \\ (y + f(x))^3 + 0.5, & \text{иначе} \end{cases}$

7. $m = \begin{cases} f^3(x) + \sin(y), & xy < 5 \\ \mathrm{ch}(f^3(x)) + y^2, & xy > 7 \\ \cos(x + f^3(x)), & \text{иначе} \end{cases}$

8. $l = \begin{cases} e^{f(x)-|y|}, & 0.5 < xy < 5 \\ \sqrt{|f(x)+y|}, & 0.1 < xy < 0.5 \\ 2f^2(x), & \text{иначе} \end{cases}$

Содержание отчета

1. Тема. Цель.
2. Оборудование.
3. Результат выполнения практического задания.
4. Ответы на контрольные вопросы.
5. Вывод.

Практическая работа 12. Разработка тестовых наборов

Краткие теоретические сведенья

Качество *программного обеспечения* определяется в стандарте *ISO* 9126 как вся совокупность его характеристик, относящихся к возможности удовлетворять высказанные или подразумеваемые потребности всех заинтересованных лиц.

Тестирование – это проверка соответствия программы требованиям, осуществляемая путем наблюдения за ее работой в специально, искусственно созданных ситуациях, выбранных определенным образом.

7 принципов тестирования:

- Тестирование демонстрирует наличие дефектов
- Исчерпывающее тестирование не возможно
- Ранее тестирование
- Скопление дефектов
- Парадокс пестицида
- Тестирование зависит от контекста
- Заблуждение об отсутствии ошибок

Тест-кейс — это минимальный (атомарный) компонент теста, как правило, он нацелен только на один элемент объекта тестирования. Чем меньше у тест-кейса покрытие функциональности, тем четче область поиска причины в случае найденной ошибки.

Основные составные части тест кейса:

Заголовки

В этой части тест-кейса собраны необходимые его реквизиты, набор которых в каждом проекте различный. Как минимум, должны присутствовать: идентификатор тест-кейса, заголовок или краткое описание тестируемой функциональности и идентификатор покрываемого требования

Тестовые шаги и результаты

Эта часть уникальная для каждого тест-кейса, потому как и является его «телом». Тестовые шаги пишутся в виде списка, перечня шагов, которые проходит тестировщик. Каждому шагу, как правило, соответствует описание ожидаемого результата. Поэтому удобно эту часть тест-кейса оформлять в виде таблицы с тремя колонками: номер шага, действие и ожидаемый результат.

Задание:

Написать тест кейсы, позволяющие детально протестировать функционал (по возможности используя техники тест дизайна), соответствующие вашей теме.

Дополнительные вопросы:

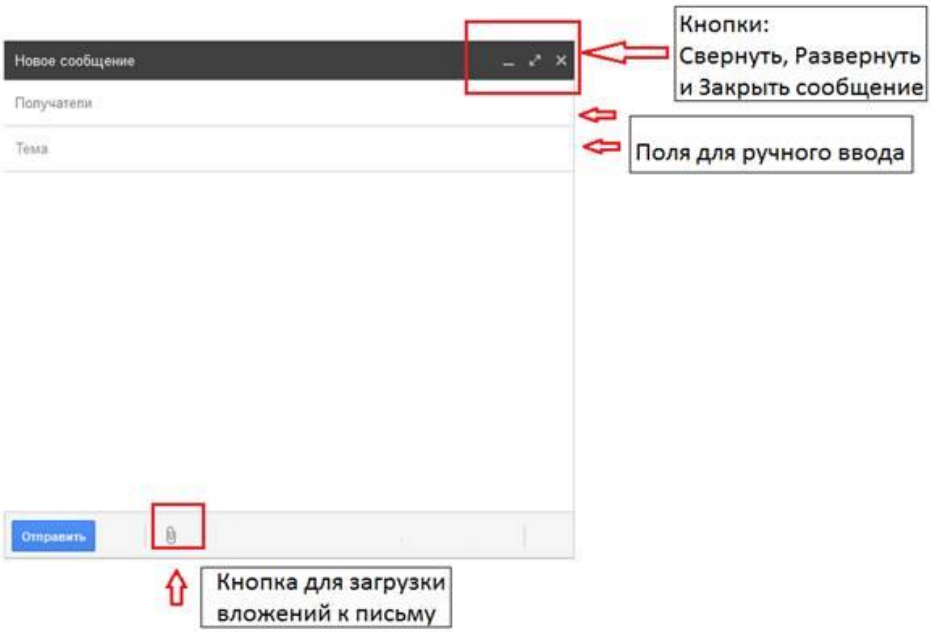
1. Что такое тестирование? Цели тестирования, стадии.
2. Что такое качество ПО? Стандарт ISO 9126.
3. Что такое тест кейс? Из чего состоит тест кейс? Принципы написания тест кейсов.
4. Техники тест-дизайна.

Темы:

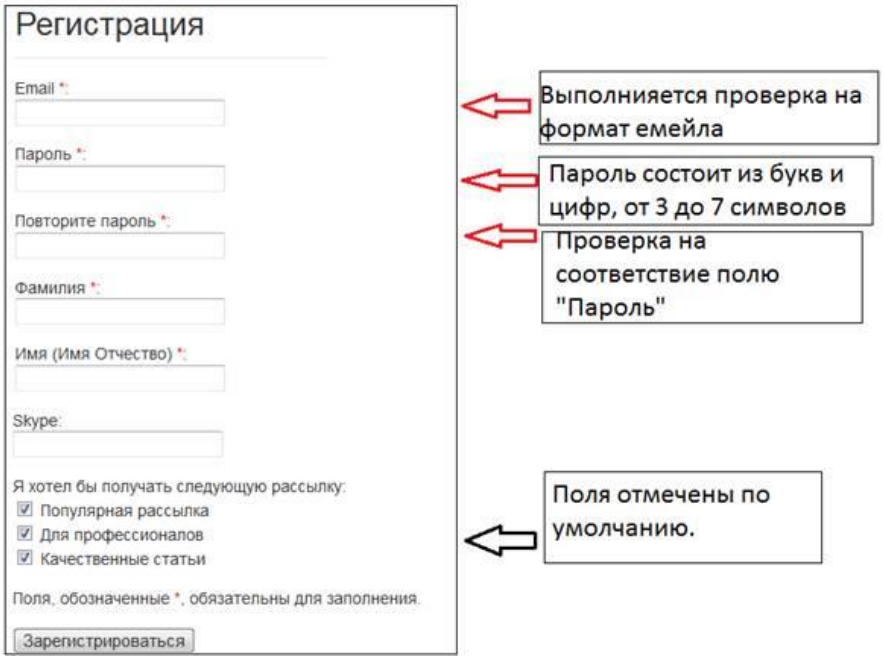
1. Ваш проект «Реализация онлайн почтового клиента». Нужно описать тест кейсы на функционал: Отправка сообщения. Путь к форме отправки

сообщения: нажатие кнопки «Новое сообщение» на панели инструментов. Дизайн с комментариями прилагается.

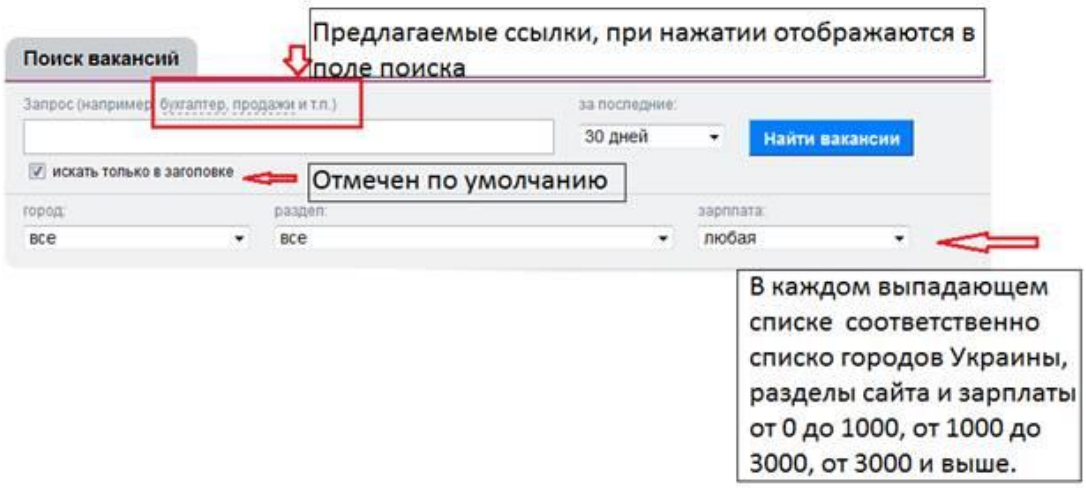
2.



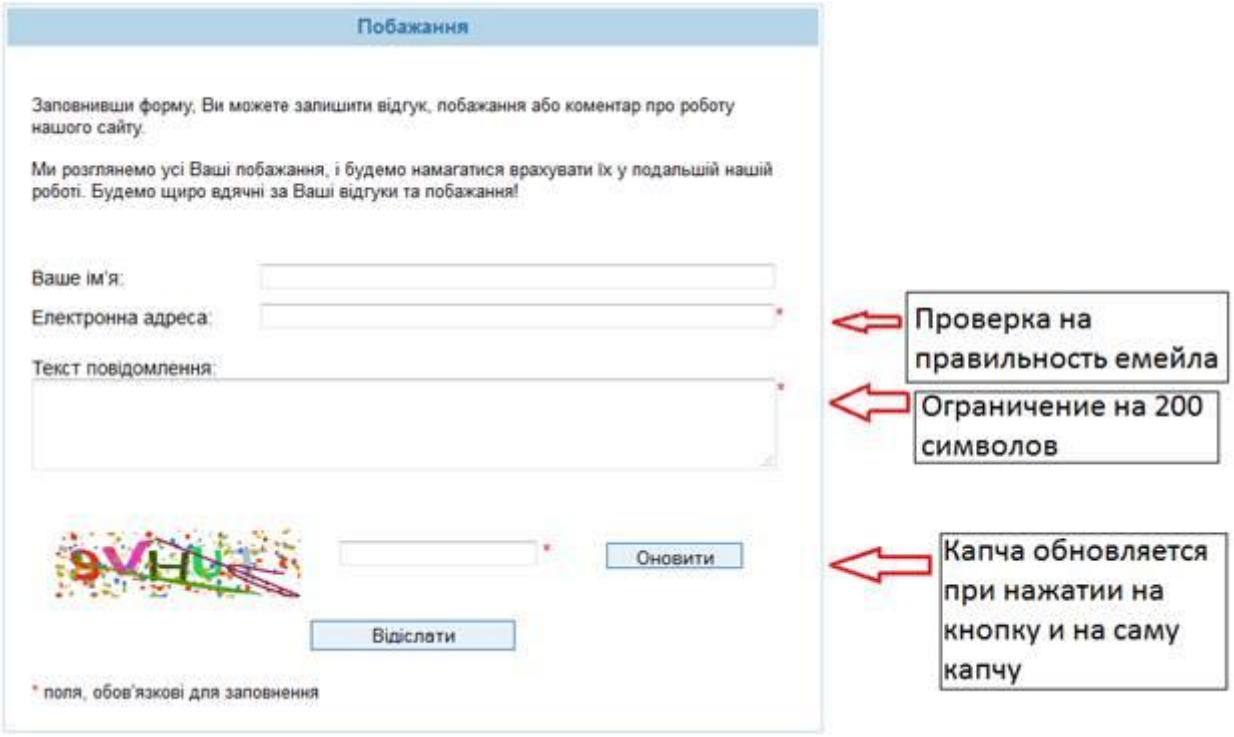
3. Ваш проект «Реализация системы библиотека». Нужно описать тест кейсы на функционал «Регистрация пользователя». Путь к форме регистрации: главная страничка сайта, кнопка «Регистрация». Дизайн с комментариями прилагается.
4.



3. Ваш проект «Реализация системы поиска работы и сотрудников». Необходимо написать тест кейсы на функционал «Поиск вакансии». Путь к форме поиска: главная страничка – кнопка поиск на панели инструментов. Дизайн с комментариями прилагается.



5. Ваш проект «Реализация сайта университета». Необходимо написать тест кейсы на форму отправки пожелания о работе сайта, которую можно найти пройдя по ссылке «Про нас» с главной странички сайта. Дизайн с комментариями прилагается.



Практическая работа 13. Разработка комплектов тестов

Цель работы: разработать рабочую тестовую документацию для тести- рования web-приложения.

Теоретические сведения

1.1 Рабочая тестовая документация

Создание тестовой документации значительно улучшает качество про- дукта за счет более тесного сотрудничества, уточнения деталей при разра- ботке плана тестирования и документации. После завершения тестирования наличие тестовой документации позволяет

проверить, насколько успешно были проведены все этапы тестирования.

Существует несколько разновидностей рабочей тестовой документации (таблица 1):

1. Check List
2. Acceptance Sheet
3. Test Survey
4. Test Cases

Check List **–** высокоуровневый список проверок, набор правил и крите- риев, по которым проводится тестирование приложения. Описывает основ- ные проверки для типовой функциональности.

Acceptance Sheet **-** документ, который содержит подробный перечень всех модулей и функций приложения, а также результаты всех тестов данных функций. Как правило, содержит статистику по наиболее важным показате- лям каждой сборки, определяющим ее качество.

Test Survey **-** документ, который содержит подробный перечень всех модулей и функций приложения, конкретные проверки для них, а также ре- зультаты всех тестов. В некоторых случаях для проверок может быть указан ожидаемый результат. Как правило, содержит статистику по наиболее важ- ным показателям каждой сборки, определяющим ее качество.

Test Cases **(набор тест-кейсов) -** набор входных значений, предусло- вий выполнения, ожидаемых результатов и постусловий выполнения, разра- ботанный для определенной цели или тестового условия, таких как выполне- ние определенного пути программы или же для проверки соответствия опре- деленному требованию.

Основной фактор выбора тестовой документации – сложность бизнес- логики проекта. Кроме того, определяющими факторами могут быть сроки проекта, размер команды и объем проекта.

На одном проекте могут комбинироваться несколько типов тестовой документации. Например, для всего проекта составлен Test Survey, но для наиболее сложных частей составлены тест-кейсы.

ТАБЛИЦА 1 – ВИДЫ РАБОЧЕЙ ТЕСТОВОЙ ДОКУМЕНТАЦИИ

Тип документации	Что описываем	Когда используем	Пример
Checklist	Основные проверки	Для типовой функциональности	Протестировать форму входа в почту
Acceptance Sheet	Части функциональности, подлежащие проверке.	Небольшие, простые по бизнес-логике про- екты Часто выпоняемые те- сты (Smoke test)	Форма входа на сайт
Test Survey	Конкретные проверки в рамках отдельных кусков функциональности. Может содержать ожидае- мый результат.	Средние или большие проекты, с понятной бизнес-логикой	Форма входа на сайт: - Корректные данные - Неверное имя пользователя - Неверный пароль...
Test Cases	Пошаговое описание, инструкции по тестированию. Всегда содержит ожидаемый результат.	Большие и долгосрочные проекты, требующие глубоких знаний в предметной области	Форма входа на сайт: 1. Откройте форму входа 2. Введите имя пользователя test1 3. Введите пароль test1 4. Нажмите кнопку «Войти» Ожидаемый результат: пользователь перехо- дит на домашнюю страницу

1.2 Тестирование web-приложения

Web-приложениями будем называть любые приложения, предоставля- ющие web-интерфейс. В настоящее время такие приложения получают все большее распространение: системы управления предприятиями и драйверы сетевых принтеров, интернет-магазины и коммутаторы связи – это только небольшая часть приложений, обладающих web-интерфейсом. В отличие от обычного графического пользовательского интерфейса web-интерфейс отоб- ражается не самим приложением, а стандартизированным посредником – web-браузером. web-браузер берет на себя все взаимодействие с пользовате- лем и обращается к web-приложению только в случае необходимости.

Web-приложения в первую очередь характеризуются тем, что их поль- зовательский интерфейс имеет стандартизированную архитектуру, в которой:

1) для взаимодействия с пользователем используется web-браузер;
2) взаимодействие с пользователем четко разделяется на этапы, в тече- ние которых браузер работает с одним описанием интерфейса;
3) эти этапы разделяются однозначно выделяемыми обращениями от браузера к приложению;
4) для описания интерфейса применяется стандартное представление (HTML);
5) коммуникации между браузером и приложением осуществляются по стандартному протоколу (HTTP).

Поэтому тестирование web-приложений обладает рядом особенностей, и при проведении самого

процесса тестирования необходимо обращать вни- мание на следующие аспекты:

1. единство дизайна;
2. навигация;
3. функциональность;
4. совместимость с браузером;
5. совместимость с операционной системой;
6. "дружественность";
7. "работоспособность";

Единство дизайна. Под единством дизайна понимается не только, а точнее не столько сочетаемость цвета элементов (так как это удел дизайне- ра), сколько соблюдение выбранной цветовой гаммы, придающей всем стра- ницам сайта "единство". Сюда входят цвета фона (или рисунок), ссылок (в т.ч. посещенной и активной), а также любых других элементов, расположен- ных на странице. Кроме того, на этом же этапе необходимо оценивать размер и вид используемого шрифта для различных уровней вложения текста (заго- ловки различных уровне, собственно текст, ссылки, примечания и т.п.) Здесь же имеет смысл оценивать совместимость с дизайном звуков, рисунков и анимации, а также проверять имеет ли место единство отображения при ис- пользовании других экранных расширений и глубин цвета.

Навигация. Навигация предполагает тестирование перемещения по сайту, что дает представление о возможности любого пользователя легко найти необходимый раздел, независимо от способа реализации меню (тек- стовые ссылки, картинки, единая картинка с картой ссылок и др.). На этом же этапе оценивается логичность перемещения между формами, кнопками и другими элементами страницы при помощи TAB, курсорных клавиш и т.п.

Функциональность. Общие подходы к тестированию функционально- сти веб-страниц аналогичны таковым при тестировании приложений. Ниже приведен примерный перечень основной функциональности веб-страниц:

- ссылки (работоспособность, открытие в том же или новом окне и т.п., полное отсутствие битых ссылок)
- формы (ввод текста, чисел, использование маски, работа с незапол- ненными полями, длина вводимых символов, корректная работа чек-боксов, комбо-боксов, radio buttons, логичность установок "по умолчанию" и т.д.).
- базы данных (поиск, добавление информации, редактирование, уда- ление, проверка на дублирование информации).
- доступ (различные роли и права)
- секретность (работа с паролями, передача данных, защита и т.д.)
- кеширование (проверка на установку кеширования и обновления файлов)
- проверка работы с браузером (refresh, forward/back, изменение раз- меров окна, выбор кодировки, скроллинг, отключение флеша, скрипта)
- фреймы (загрузка страниц, скроллинг и т.п.)
- анимация (наличие, изменение размеров, загрузка и т.д.)
- аудио и видео (наличие, размещение, качество и др.)
- activex
- печать (корректно ли печатаются страницы)
- загрузка (как на сайт, так и с сайта)
- экспорт/импорт данных (если есть такая ф-я)
- интеграция (например: возможность входа на сайт через социальные сети или оплата услуг через paypal account)
- рекомендации для достижения хорошего веб приложения (в некото- рых частных случаях могут не соблюдаться - например, не всегда валидаци- онные сообщения будут красного цвета):
- как внешний вид так и наполнение сайта должны быть надлежащего уровня (с сайтом должно быть приятно и удобно работать);

– все элементы должны быть выровнены, выравнивание одних и тех же элементов в различных частях одного и того же сайта должно быть уни- фицировано (например: выравнивание кнопок по левому краю);

– шрифт всех ярлыков должен быть одинакового цвета;

– не должно быть битых ссылок (ссылок не ведущих на необходимый контент, несуществующих ссылок);

– каждая под-страница (так называемые "child pages" или "subpages") должна открываться в новом окне;

– на каждой странице должны присутствовать удобные элементы навигации. такие как: хлебные крошки, вперед/назад, сохранить/отменить и т.д.;

– если в таблице содержится больше 10 элементов информации, то должна присутствовать "пагинация" (навигационные ссылки: следую- щая/предыдущая, первая/последняя и номера страниц);

– у каждой страницы наряду с заголовком должен быть подзаголовок выдержанные в одном стиле (шрифт, цвет и т.д.);

– каждый ярлык должен начинаться с заглавной буквы;

– желательно, чтобы валидационные сообщения были красного цвета;

– все обязательные для заполнения поля должны быть жирного шриф- та или помечены специальным символом: *.

Совместимость с браузером. Общеизвестно, что в силу конкуренции, тот или иной браузер имеет нередко даже существенные отличия в отобра- жении одной и той же страницы. Для того, чтобы убедится, что любой поль- зователь сможет получить всю необходимую информацию требуется прово- дить тестирование Web-страниц в различных браузерах. Кроме того имеются различия и в разных версиях одного и того же браузера. Это также необхо- димо учитывать при тестировании.

"Дружественность". Под "дружественностью" мы понимаем то, насколько прост, легок в обращении и интуитивно понятен интерфейс сайта: легка ли навигация, доступно ли меню, не используются ли раздражающие пользователя приемы, не много ли всплывающих окон, все ли ссылки явля- ются "рабочими", все ли необходимые данные доступны для пользователя и т.д. Например, если на сайте есть файл для скачивания, то желательно, чтобы пользователь имел возможность заранее знать его размер, мог оценить время закачки.

"Работоспособность". Проверка на "работоспособность" подразуме- вает оценку скорости загрузки как страниц сайта в целом, так и каждого эле- мента в отдельности. Сюда включается оценка размера используемых рисун- ков, html-файлов, аудио и видео файлов, адаптация их к различным типам со- единений (от обычного модемного dial-up соединения, начиная с 14400, до высокоскоростных технологий).

Порядок выполнения работы

1. Получить задание у преподавателя.
2. Разработать рабочую тестовую документацию для web-приложения.
3. Оформить отчет и защитить лабораторную работу.

Содержание отчета

1. Цель работы.
2. Краткие теоретические сведения.
3. Рабочая тестовая документация.
4. Выводы по работе.

Контрольные вопросы

1. Какие существуют разновидности рабочей тестовой документации?
2. Check List: что описывают и когда используют?
3. Acceptance Sheet: что описывают и когда используют?
4. Test Survey: что описывают и когда используют?

5. Test Cases: что описывают и когда используют?

6. Что такое web-приложение?

7. Какую архитектуру имеет web-приложение?

8. На какие особенности необходимо обращать внимание при тестиро- вании web-приложения, охарактеризуйте эти особенности.

9. Какие основные проверки необходимо выполнять при тестировании web-приложения?

Практическая работа 15. Воспроизведение ошибок тестирования приложений

Цель. Изучить процесс и организацию тестирования; критерии качества тестирования; методы, используемые при тестировании; классификацию и оценку ошибок.

Оборудование. ПК

Ход работы

1. Ознакомиться с теоретической частью.
2. Выполнить практическое задание.
3. Ответить на контрольные вопросы.
4. Оформить отчет.

Теоретическая часть

Тестирование – процесс выявления ошибок в программе, а *отладка* – процесс их устранения. Более многословное определение тестирования даёт стандарт IEEE 829-1983 «*Standard for Software Test Documentation*»: «тестирование – это процесс анализа ПО, направленный на выявление отличий между его реально существующими и требуемыми свойствами и на оценку свойств ПО».

Отличие между реально существующим и требуемым свойствами называется *дефектом*, или *ошибкой*. Среди программистов распространён термин «*bug*» (жучок) – баг. Авторство терминов «*bug*» и «*debugging*» приписывают то знаменитому изобретателю Томасу Эдисону, в фонограф которого заползали жучки, то американке Грейс Хоппер, которая в конце II мировой войны работала с компьютером *Mark II* и вынуждена была искать и устранять из него жучков, вызывавших замыкания и другие сбои в работе. Итак, ошибки есть отклонения объектов (программ, документов) от спецификаций, при условии, что последние существуют и являются правильными, а также отклонения от требований «здравого смысла», называемых также *общесистемными требованиями* (к ним, например, относятся законы математики и логики).

Объектами тестирования являются: исходные тексты программ; исполняемые модули (программы, библиотеки); документация. Тестирование документации должно выявить расхождения между содержанием документов и описанных в них программ.

Термином *тест* называют *эксперимент*, выполняемый над программой, для которого определены критерии успешности. *Тестовые данные* – это конкретные данные, подаваемые программе при эксперименте. Важно понимать, что входные данные – это не только числа, строки, даты, файлы и т. д., но и *действия пользователя*, выполняемые при работе с пользовательским интерфейсом. Те или иные манипуляции с элементами пользовательского интерфейса также рассматриваются как тестовые данные.

Тестовая процедура, или *тестовый сценарий*, – это спецификация проведения эксперимента, которая описывает порядок ввода тестовых данных и критерии успешности, то есть те признаки, по которым следует судить о правильности работы программы (успех) или о наличии ошибки (неуспех). По сути, конкретный тест, или *тестовый случай* (*test case*), есть выполнение тестового сценария для конкретного набора тестовых данных.

Три принципа тестирования:

- необходимо создавать тесты, которые с высокой вероятностью *находят* ошибки, а не демонстрируют правильность работы программы;
- необходимо привлекать для тестирования *сторонних* специалистов, поскольку программист психологически неспособен выполнить исчерпывающее тестирование своего собственного кода;
- тесты должны проводиться *регулярно*, в соответствии с *планом* на основании *регламента*.

Основные *проблемы* организации тестирования программы состоят в том, что:

- нередко отсутствует эталон, которому должны соответствовать результаты тестирования;

- невозможно создать набор тестов для исчерпывающей проверки сложной системы;
- отсутствуют практически пригодные формальные критерии качества тестирования.

Исправление ошибки, внесенной на стадии анализа требований и обнаруженной на стадии окончательной проверки, стоит в среднем в 100 раз дороже, чем исправление этой же ошибки на стадии анализа (по другим источникам – в 500 раз дороже). Для исправления проблем, выявленных при сопровождении продукта, программисты по статистике тратят до 60 % времени, *пытаясь понять документацию и логику программы*.

Тестирование является весьма затратной деятельностью. Поэтому в большинстве случаев разработчики заранее формулируют какой-либо критерий качества создаваемых программ (определяют так называемую *планку качества*), добиваются выполнения этого критерия и после этого выпускают продукт на рынок. Такая концепция получила название *Good Enough Quality* (достаточно хорошее качество) в противовес концепции *Best Possible Quality* (наилучшее качество).

Концепция *Good Enough Quality* вовсе не эквивалентна отказу от полноценного тестирования. Выпуск плохо протестированного продукта из-за недостатка времени – это всегда плохо. Практика показывает, что пользователи склонны со временем забывать даже значительные задержки с выпуском продукта, но плохое качество выпущенного продукта запоминается на всю жизнь.

Критерии качества тестирования

Критерии качества тестирования иногда называют *критериями покрытия* (*test coverage*), поскольку они показывают степень охвата, «покрытия» тех или иных аспектов продукта при тестировании. Критерии покрытия можно разделить на две группы: критерии покрытия структуры (*structural test coverage*) и критерии покрытия поведения (*behavioral test coverage*).

Критерий покрытия поведения указывает на полноту функционального тестирования (см. п. 6.3.6), то есть степень, в которой тестирование охватило все функциональные возможности системы. Покрытие поведения является полным, если при тестировании каждая функциональная возможность задействована хотя бы раз. Этот критерий является сравнительно легко достижимым по сравнению со структурными критериями (см. ниже). Следует также отметить, что полное покрытие поведения является абсолютно необходимым. Это тот минимум, который обязаны выполнить тестеры. Тем не менее полное покрытие поведения выявляет лишь самые очевидные ошибки и не даёт никаких гарантий качества.

Структурные критерии включают оценки полноты покрытия операторов, маршрутов и данных.

1. Полнота покрытия операторов. Критерий достигает 100 %, если при тестировании были выполнены все операторы программы (хотя бы один раз). Критерий указывает, не остались ли участки кода, в которые управление не попадало ни разу. Если есть множество не отработавшего кода, говорить о качественном тестировании не приходится. Существуют продукты, которые позволяют автоматически зафиксировать и визуально показать отработавшие операторы, например *Rational Pure Coverage*.

2. Полнота покрытия маршрутов. Можно достигнуть 100 % полноты покрытия операторов, но многие ошибки не будут выявлены, так как они возникают не на всех маршрутах. Маршрутом, или логическим путём, называют конкретную последовательность выполнения операторов. За счёт операторов ветвления и циклов в программе может существовать множество потенциальных маршрутов. Критерий достигает 100 %, если при тестировании отработали все возможные маршруты программы (хотя бы один раз). Полнота покрытия маршрутов – очень сильный критерий, достигнуть его максимума в большой программе нереально. По ряду оценок, при самом тщательном тестировании программы достигается примерно 60%-я полнота покрытия маршрутов, а с использованием специальных средств автоматизации тестирования можно достигнуть 90 %. Установить значение полноты покрытия маршрутов тяжело. Поэтому данный критерий слабо пригоден на практике.

3. Полнота покрытия данных. Даже если добиться 100%-й полноты покрытия маршрутов, – это недостаточно для выполнения полного тестирования. Исследования показали, что только 35 % ошибок остаются из-за упущенных маршрутов. Дело в том, что даже на одних и тех же маршрутах ошибки могут возникнуть или нет, в зависимости от разных входных данных. Критерий полноты покрытия данных достигает 100 %, если при тестировании были проверены все возможные входные данные во всех возможных сочетаниях. Это самый сильный критерий, но практически недостижимый. Даже для единственного 16-разрядного параметра придётся провести 65536 тестов. Два таких параметра потребуют уже 655362 тестов (все сочетания значений). Однако и 100%-я полнота покрытия данных не гарантирует выявления всех ошибок, поскольку некоторые ошибки могут зависеть от состояния среды, в которой работает программа, например от содержимого оперативной памяти.

Методы, используемые для тестирования

1. Инспекция кода

Анализ исходного текста называют также статическим тестированием, ручным методом и инспекцией (экспертизой) кода. Инспекция кода не требует запуска программ. При тестировании исходного текста выполняется сквозной просмотр текста программы непосредственно разработчиком или группой специалистов с целью поиска нарушений логики и типовых ошибок.

Все специалисты отмечают, что этот метод эффективнее и дешевле обычного тестирования, основанного на эксперименте. Результативность метода составляет, по разным оценкам, от 60 до 90 % от всех выявленных ошибок. Успех метода основан на том, что большинство ошибок являются достаточно шаблонными и поэтому легко выявляются экспертами при целенаправленном поиске. Кроме того, внимательное чтение кода позволяет выявить ошибки, которые должны проявиться лишь при особом сочетании условий и, скорее всего, не будут найдены при обычном тестировании.

Ручной метод требует утомительной, интенсивной работы интеллекта и не поддаётся формализации, из-за чего, видимо, и используется столь редко, несмотря на высокую эффективность. Весьма подробное описание инспекции кода приведено Майерсом.

2. Многократная разработка

Как мы помним, одной из проблем тестирования является то, что зачастую отсутствуют эталоны, с которыми можно сравнивать выдаваемые системой результаты. Поэтому при работе системы не всегда вообще возможно понять, что система ошиблась в результатах или в предписанном поведении.

Иногда к разработчикам приходит идея о независимой реализации того или иного модуля, алгоритма или даже всей программы несколькими группами разработчиков. Идея состоит в том, что можно сравнивать работу нескольких вариантов программы при одних и тех же условиях. Если результаты работы будут идентичными, можно считать, что ошибки нет ни в одной из реализаций, в противном случае в одной из реализаций есть ошибка.

Метод многократной разработки действительно применялся, но впоследствии от него практически отказались. Причины этого состоят в следующем. Метод чрезвычайно затратен. Даже двукратная разработка требует удвоения ресурсов, однако бесполезна для поиска ошибки, ведь при расхождении результатов невозможно понять, в какой версии ошибка. Если выполнить трёхкратную разработку, то можно ожидать, что в двух версиях результаты совпадут, тогда ошибка находится в третьей. Но что, если не совпадут все три результата?

Но это только полбеды. Ведь метод исходит из предположения о том, что совпадение результатов гарантирует отсутствие ошибки. Оказалось, что это не так. Длительный опыт продемонстрировал интересный психологический феномен: люди склонны делать одни и те же ошибки, *типовые ошибки*. Существуют виды ошибок, которые люди делают с довольно большой вероятностью (на этом феномене основан успех инспекции кода).

3. Классы эквивалентности и граничные условия

В тестировании программного обеспечения приходится принимать решение о том, следует ли использовать тот или иной тестовый набор или в этом нет необходимости. Поскольку хочется, с одной стороны, минимизировать набор тестов, чтобы сократить время тестирования, а с другой стороны, обеспечить достаточную полноту тестирования, задача отбора является крайне актуальной.

А. Баранцев пишет, что использование критериев отбора для формирования тестовых наборов наиболее ярко и наглядно можно показать на примере отбора музейных экспонатов:

«Классический критерий отбора музейного объекта […] во многом отличен от принципов составления архива и библиотеки. Если последние претендуют на максимальную полноту (чем более обширно собрание, тем оно ценнее), то музейные коллекции распределяются между двумя полюсами – вещь маргинальная и вещь идеальная, образцовая. То есть, с одной стороны, мы имеем предметы, которые, не обладая самостоятельной ценностью, дают представление о целом классе, с другой стороны – всё уникальное, из ряда вон выходящее. Да, набор тестов – это не библиотека и не архив, это – музей».

При формировании набора тестов обязательно следует включать в него тесты ровно двух указанных типов: тесты, которые, не обладая самостоятельной ценностью, дают представление о целом классе, и тесты «маргинальные», пограничные.

Для формирования образцовых тестов применяется *метод эквивалентных разбиений*, состоящий в разбиении значений входных параметров на *классы эквивалентности*. Затем из каждого класса

эквивалентности выбираются значения-представители. Если один тест класса эквивалентности выявляет ошибку, то с большой вероятностью и другие тесты этого класса будут выявлять эту ошибку.

Проще всего выделение классов эквивалентности можно показать на примере численных параметров. Если параметр должен принимать значения из интервала [0;100], для него естественным образом выделяются три класса эквивалентности: один класс допустимых значений [0;100] и два – недопустимых: (–∞;0) и (100;+∞). При тестировании следует проверить поведение системы при подаче значения из *каждого* класса.

Часто классы эквивалентности приходится формировать для целой совокупности параметров, если они тесно связаны. Например, подпрограмма должна выполнять поиск подстроки *s* в строке *S*. Очевидно, что классы должны быть сформированы с учётом наличия/отсутствия *s* в *S*. Минимально допустимый набор классов должен включать классы {*s* есть в *S*} и {*s* нет в *S*}.

Не менее важной является проверка граничных или близких к границе значений, так называемый *метод анализа граничных значений*. Практика показывает, что вероятность ошибки особенно высока на границах классов эквивалентности. Метод анализа граничных значений предписывает для каждой границы тестировать значения *точно на границе*, и значения, достаточно *близкие к граничному*.

В примере с числом в интервале [0;100] следует протестировать следующие значения: {–0,00001; 0; 0,00001; 99,99999; 100; 100,00001}. В примере с поиском подстроки *s* в строке *S* следует протестировать следующие пограничные условия:

- *длина*(*s*) = *длина*(*S*)–1;
- *длина*(*s*) = *длина*(*S*);
- *длина*(*s*) = *длина*(*S*)+1;
- *длина*(*s* и/или *S*) = 0;
- *длина*(*s* и/или *S*) = 1;
- *s* = *nil*;
- *S* = *nil*.

Некоторые виды тестирования уделяют основное внимание либо типичным, либо маргинальным объектам или явлениям. Так, например, при исследовании удобства пользовательского интерфейса логично предполагать поведение наиболее *типичного* пользователя, а при нагрузочном тестировании для системы целенаправленно создаются не типичные, а всё более невыносимые условия.

Организация тестирования

Наиболее плодотворно то тестирование, которое осуществляется целенаправленно и регулярно и не ограничивается простым автономным тестированием, выполняемым самими программистами. Поэтому задачей руководства проекта является организация этого процесса, что включает подбор команды тестирования, разработку регламента тестирования, организацию разработки тестов и контроль за исполнением регламента.

Регламент тестирования должен определять обязанности и порядок взаимодействия программистов, тестеров и менеджеров, а также состав инструментальных средств и правила их использования.

В команде тестирования специалистов делят на тест-инженеров и тестеров. Тест-инженер (*test engineer*) – наиболее квалифицированный специалист, который владеет навыками планирования тестирования, разработки и проведения тестов, а также понимает предметную область. Тестер (*test technician, test specialist*) – менее квалифицированный специалист, который главным образом занимается выполнением тестов.

Для систематического тестирования следует подготовить набор тестов, обеспечивающих желаемую степень покрытия. Разработка тестов фактически сводится к разработке тестовых сценариев и подготовке тестовых данных для этих сценариев.

Поскольку основным видом тестирования является функциональное тестирование, тестерам следует по меньшей мере обеспечить *покрытие поведения*.

Идеальным источником для создания полного набора тестовых сценариев являются варианты использования. Сценарии вариантов использования являются почти готовыми сценариями функционального тестирования. Обычно сценарий тестирования записывают в виде таблицы, в которой в столбце «Действие» последовательно описывают шаги сценария, выполняемые пользователем, в столбце «Ожидаемый результат» – правильную, ожидаемую реакцию системы. При проведении теста таблица дополняется

столбцом «Фактический результат», в котором регистрируется правильность или неправильность работы программы.

Тестовые данные могут быть вписаны прямо в сценарий либо прилагаться в отдельном файле или документе. Это особенно важно, если один и тот же сценарий следует использовать для нескольких наборов разных тестовых данных.

Тестовые данные не всегда возможно просто записать в документе. Если программе требуются входные файлы, то эти файлы следует заранее подготовить и расположить в оговорённых папках, а в сценарии можно сослаться на файлы по именам. Иногда ситуация ещё сложнее, если поток входных данных следует генерировать в большом объёме и/или с большой скоростью. В этом случае приходится создавать специальные программы, которые генерируют тестовые данные с нужными свойствами1

Из всего набора созданных тестов следует выделить те тесты, которые будут использоваться при дымовом тестировании.

Для того чтобы максимально сократить время подготовки тестовых данных, желательно принять следующие меры:

- воспользоваться генератором тестовых данных;
- подготовить тестовые данные силами проектной группы:
- привлечь пользователей к подготовке тестовых данных; взять данные, подготовленные какой-то другой автоматизированной системой;
- выделить тестовые данные из имеющихся файлов;
- выделить тестовые данные из внешних документов.

Классификация ошибок

Существует много классификаций ошибок. Наиболее простой является классификация ошибок по серьёзности:

- Ошибки с **высокой серьёзностью** препятствуют решению важных задач. Они проявляются всегда или с высокой степенью вероятности.
- Ошибки со **средней серьёзностью** препятствуют решению маловажных задач. Также к этому классу относят ошибки, которые проявляются редко, при маловероятном стечении обстоятельств.
- Ошибки с **низкой серьёзностью** не препятствуют решению задач и скорее раздражают пользователей, чем по-настоящему мешают им. Типичные примеры: орфографические ошибки в пользовательском интерфейсе; неполнота документации.

Очевидно, что серьёзность ошибок фактически определяет приоритетность их исправления, хотя и не безусловно.

Вторая классификация распределяет ошибки по видам деятельности, которые привели к их возникновению.

1. Ошибки анализа. Это самые тяжёлые по последствиям ошибки. Примеры: не предусмотрена нужная функциональность; дублирование функций; лишние функции; неверно оценены требования к техническим средствам; неверно оценены требования к производительности или переносимости.

2. Ошибки проектирования. В рамках предложенного проекта невозможно достичь требуемой в ТЗ функциональности и критериев качества. Самые тяжёлые ошибки проектирования – архитектурные.

3. Ошибки документации. Расхождения документов с программами, отсутствие нужных сведений и нужных документов.

4. Ошибки программной реализации. Это самый широкий класс ошибок. Рассмотрим важные виды ошибок программной реализации:

4.1. Ошибки пользовательского интерфейса. Интерфейс не позволяет использовать какие-либо существующие функциональные возможности (целиком или частично). Например, реализовано несколько методов, интерфейс не позволяет выбирать нужный. Метод имеет важные параметры, интерфейс не позволяет их настраивать и т. д.

4.2. Ошибки вычислений. Программа не учитывает возникновение или накопление ошибок вычислений (погрешности, округления, переполнения и пр.)

4.3. Ошибки использования методов. Каждый метод должен сопровождаться описанием или оценкой вычислительных погрешностей и других ограничений, снижающих его универсальность. Отсутствие таких оценок может привести к неверной работе программы, реализующей данный метод.

Программисту тяжело обнаруживать ошибки метода, поскольку для этого требуется высокая квалификация в предметной области метода. Однако при полной уверенности, что все другие ошибки

устранены и метод реализован правильно, сопровождаемой отказом программы в получении ожидаемых результатов, программист может предположить дефект в самом методе.

Решение данной проблемы необходимо искать совместно с автором метода, если это возможно, либо совместно с квалифицированным специалистом в данной области. Наиболее часто, разумеется, виноват программист, однако опыт свидетельствует о том, что ошибки в методах встречаются чаще, чем хотелось бы. Как правило, методы «не работают» на некоторых специфических наборах данных или при некоторых математических упрощениях, которые не следовало делать. В результате метод применим в 99 % случаев, однако в 1 % даёт сбой.

4.4. Ошибки взаимодействия с устройствами и программами. Программа не учитывает возможность отсутствия, неисправности или ошибок функционирования используемых устройств (дисков, принтеров и т. д.) или внешних программ, с которыми устанавливается связь по специальному или известному протоколу (DDE, OLE, FTP, HTTP и т. д.).

4.5. Ошибки синхронизации. Программа не учитывает возможность ошибок, вызванных неверным порядком выполнения действий (например, принтер ещё не выбран, а печать вызывается), отсутствием синхронизации процессов и т. п. Типичная ситуация – повторный вызов функции или программы во время её выполнения.

4.6. Ошибки неверного использования памяти. Ошибки этого подкласса нередко приводят к зависанию или общему сбою программы. Их труднее всего находить, поскольку проявления таких ошибок зачастую зависят от текущего содержимого памяти, которое может быть различным в разное время и на разных машинах. Типичные примеры ошибок неверного использования памяти таковы:

- ошибки распределения свободной памяти, в том числе выделение памяти неверного размера (слишком большого или меньше нужного) и освобождение невыделенной памяти (часто в результате повторного освобождения);
- ошибки доступа, чаше всего выход за границы массива (по указателю или индексу);
- использование неинициализированных переменных.

Оценки ошибок

Тестирование предназначено для поиска ошибок, а не для демонстрации того, что для некоторых данных программа работает правильно. Тем не менее чем больше ошибок найдено, тем меньше уверенности в надёжной работе модуля, так как если найдено много ошибок, то, во-первых, их было много и, во-вторых, возможно, многие ошибки остались необнаруженными. Поэтому необходимо разработать метод, позволяющий оценивать количество ошибок в модуле. Если приблизительное число ошибок известно, то чем больше их найдено и исправлено, тем больше уверенности в корректности модуля.

Фирмой IBM установлено, что в среднем одна ранее не обнаруженная ошибка появляется в каждых 100 операторах программы. Большинство ошибок содержится в сопряжениях модулей и операторах ввода-вывода. Такого рода ошибки затрагивают обычно сразу несколько модулей. После того как проведено тестирование отдельно каждого модуля, последние объединяются в систему. Экспериментально установлено, что в 90 % всех новых модулей и в 15 % старых при этом необходимо вносить изменения. Приблизительно в 15 % новых модулей и 6 % старых следует внести более 10 изменений.

Если осталось D старых модулей и сформировано M новых, число необходимых изменений N определяется эмпирически найденным выражением

$$N = 2(0{,}9M + 0{,}15D) + 23(0{,}1M + 0{,}06D)$$

Эксперименты показали, что число ошибок в неоттестированных программах пропорционально $E^{2/3}$, где E – мера Холстеда, характеризующая сложность программы. Коэффициент пропорциональности равен примерно 1/3200. В программах, прошедших стадии тестирования и отладки, это отношение сохраняется, но коэффициент пропорциональности уменьшается.

Различные формулы оценки количества ошибок не учитывают вероятность внесения k новых ошибок при исправлении n старых.

Вероятность того, что новый тестовый прогон позволит обнаруживать ошибки, будет зависеть от процента оставшихся ошибок. Последние ошибки обнаружить труднее. Формула

$$avg(T_n) = \frac{1}{\beta}\sum_{k=0}^{n-1}\frac{1}{N-k}$$

позволяет установить среднее количество проверок, необходимых для обнаружения *n* ошибок. Здесь β – неизвестное значение, которое может быть определено экспериментально. Исходя из предыдущей формулы, получаем формулу

$$P = \frac{avg(T_n)}{avg(T_N)}$$

которая показывает, какой процент *P* тестовых прогонов, необходимых для обнаружения всех *N* ошибок, был сделан, когда в процессе этих прогонов было найдено *n* ошибок.

Если *n*=10, по формуле получим, что половина всех ошибок может быть обнаружена в первых 22% тестовых прогонов, необходимых для обнаружения всех ошибок. Процент прогонов возрастает до 37, если требуется найти 7 из 10 ошибок, и до 66, если требуется обнаружить 9 ошибок из 10, то есть третья часть времени затрачивается на обнаружение одной последней ошибки. Если в программе имеется 100 ошибок, то 90 обнаруживается за первые 44 % тестовых прогонов.

Этот закон, устанавливающий, что эффект от тестирования уменьшается со временем, позволяет сделать вывод о необходимости прекращать тестирование в тот момент, когда оно становится экономически невыгодным. Кроме того, можно утверждать, что в отлаженном модуле почти всегда остаются ошибки.

Практическая часть

Задание 1. Перечислите критерии качества тестирования.

Задание 2. Перечислите методы инспекции кода.

Задание 3. Перечислите классификацию ошибок, обнаруживаемых при тестировании программного продукта.

Задание 4. Выполните оценку ошибок программы нахождения среднего арифметического *n* чисел.

Задание 5. Результаты выполнения практического задания запишите в отчет.

Контрольные вопросы

1. Что такое тестирование? Что является объектами тестирования?
2. Изобразите и опишите информационные потоки при тестировании.
3. Опишите виды тестирования.
4. Поясните понятия «тест», «тестовые данные», «тестовый эксперимент».

Практическая работа 17. Внесение информаций о дефекте в систему контроля дефектов

Цель :

- Изучить назначение , правила заполнения дефектной ведомости.
- Отработать навыки визуального определения дефектов
- Заполнение технической документации

Данный документ служит основанием для написания сметы и выделения средств на ремонтные работы

Дефектная ведомость относится к первичной документации и фиксирует изъяны, поломки, всевозможный брак оборудования, устройств, материалов, используемых в деятельности предприятия. Для того, чтобы провести их ремонт и восстановление по всем правилам нужно соблюсти определенную процедуру, частью которой является составление дефектной ведомости.

Документ носит сопроводительный характер при выявлении различного рода дефектов. В него вписываются:

Составление дефектной ведомости web приложения

Наименование организации _____

Ф.И. членов группы _____

_____

Наименование прибора_____

Место нахождения_____
Дата составления _____

Дефектная ведомость

№ п.п	**неисправность**	**Причина неисправности**	**Устранение неисправности**

Подпись _____

Дефектная ведомость

№ п.п	**неисправность**	**Причина неисправности**	**Устранение неисправности**
1	*Обрыв провода*	*Короткое замыкание*	*замена*
2	*стартер*	*Срок службы*	*замена*
3	*дроссель*		*ремонт*

Ответить на вопросы:

1. Что такое верификация?
2. Укажите основную цель верификации.
3. Для повышения эффективности использования человеческих ресурсов при разработке верификация должна быть тесно интегрирована с ...
4. Что такое отладка?
5. Что такое жизненный цикл разработки программной системы?
6. С чего начинается жизненный цикл?
7. Укажите основные понятия жизненного цикла.
8. Укажите основные модели жизненного цикла и их определения.
9. Укажите работы на каждом этапе каскадного жизненного цикла.
10. Особенность ... жизненного цикла состоит в том, что переход к следующему этапу происходит только тогда, когда ... завершены все работы ... этапа
11. В каскадном жизненном цикле сначала ... готовятся все требования к системе, затем по ним ... готовятся все требования к программному обеспечению, ... разрабатывается архитектура системы и так далее до тестирования.
12. Что является витком в спиральной модели жизненного цикла?
13. Укажите процессы любой модели жизненного цикла.
14. Укажите основную цель процесса управления конфигурациями.
15. Что такое целостность?
16. Что такое процесс гарантии качества?
17. Гарантирует ли процесс гарантии качества разработку качественной программной системы?

Практическая работа 20. Разработка и оформление контрольных примеров для проверки работоспособности программного кода ПО.

Цель работы: изучить возможность создания автоматических тестов, для модульного тестирования.

Сегодня тестирование – это обязательная часть процесса разработки программного обеспечения (далее – ПО). Это связано с жесткими правилами конкуренции для компаний, производящих программные продукты (ПП).

Раньше таких компаний на рынке было мало и пользователи программных продуктов были продвинутыми и заменяли тестеров. Если в программе обнаруживались баги, то пользователь звонил или отправлял письмо в компанию, где ошибку исправляли и по почте отправляли дискетку со свежим релизом. Но начиная с 1990 года согласно статистики продажи персональных компьютеров с каждым годом удваивались. И появилась армия пользователей, которая не готова была что-то тестировать. Если что-то не устроило было проще обменять на другой софт, т.к. число компаний производящих ПО тоже увеличивалось с каждых готом. И у пользователей появился выбор что покупать и чем пользоваться.

Таким образом, тестирование ушло внутрь компаний, и появилась профессия тестировщика.

Рассмотрим определение, которое записано в SWEBOK.

Тестирование ПО – это проверка соответствия между реальным поведением программы и ее ожидаемым поведением на конечном наборе тестов, выбранном определенным образом. [IEEE Guide to Software Engineering Body of Knowledge, SWEBOK, 2004].

Все виды тестирования можно условно разделить на две большие группы: Статическое тестирование (static testing).

Динамическое тестирование (dynamic testing).

Статическое тестирование – это процесс анализа самой разработки программного обеспечения, т. е. тестирование без запуска программы.

К данной группе можно отнести анализ кода. Данный вид тестирования осуществляется в основном программистами. Проводят тестирование артефактов разработки программного обеспечения, таких как требования, дизайн или программный код, проводимое без исполнения этих артефактов. Например, с помощью рецензирования или статического анализа.

Статический анализ кода (static code analysis) – это анализ исходного кода, производимый без его исполнения.

Динамическое тестирование – это тестовая деятельность, предусматривающая эксплуатацию (запуск) программного продукта.

Динамическое тестирование предполагает запуск программы, выполнение всех ее функциональных модулей и сравнение фактического ее поведения с ожидаемым.

Статическое тестирование позволяет обнаружить дефекты, которые являются результатом ошибки и привести к сбоям в программном обеспечении. Динамическое тестирование позволяет продемонстрировать непосредственно сбои в программном обеспечении.

Существует несколько признаков, по которым принято производить классификацию видов тестирования.

По знанию системы выделяют:

- тестирование «черного ящика» (black box testing);
- тестирование «белого ящика» (white box testing);
- тестирование «серого ящика» (grey box testing).

Модульное тестирование, или юнит-тестирование (англ. unit testing) — процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы.

Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

Цель модульного тестирования — изолировать отдельные части программы и показать, что по отдельности эти части работоспособны.

Задание 1. Создание проекта программы, модули которого будут тестироваться.

Разработаем проект содержащий класс, который вычисляет площадь прямоугольника по длине двух его сторон.

Создадим в Visual Studio новый проект Visual C# -> Библиотека классов. Назовём его MathTaskClassLibrary.

Class1 переименуем в Geometry.

В классе реализуем метод, вычисляющий площадь прямоугольника. Для демонстрации остановимся на работе с целыми числами. Код программы приведён ниже.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MathTaskClassLibrary
{
    public class Geometry
    {
        public int RectangleArea(int a, int b)
        {
            return a * b;
        }
    }
}
```

Рисунок 2

Создание проекта для модульного тестирования в Visual Studio.

Чтобы выполнить unit-тестирование, необходимо в рамках того же самого решения создать ещё один проект соответствующего типа.

Правой кнопкой щёлкните по решению, выберите “Добавить” и затем “Создать проект…”.

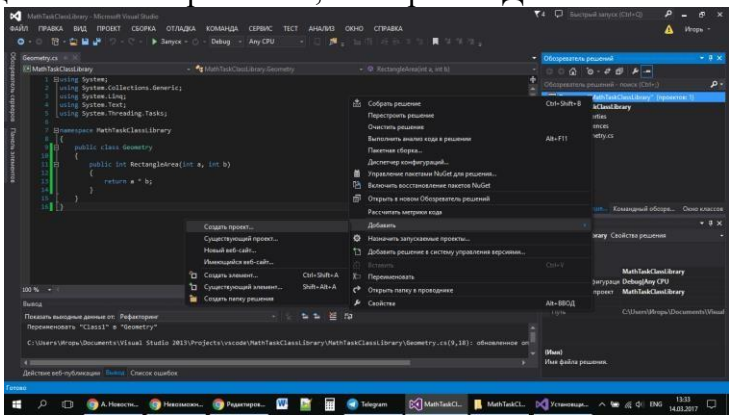


Рисунок 3

В открывшемся окне в группе Visual C# щёлкните “Тест”, а затем выберите “Проект модульного теста”. Введите имя проекта MathTaskClassLibraryTests и нажмите “ОК”. Таким образом проект будет создан.

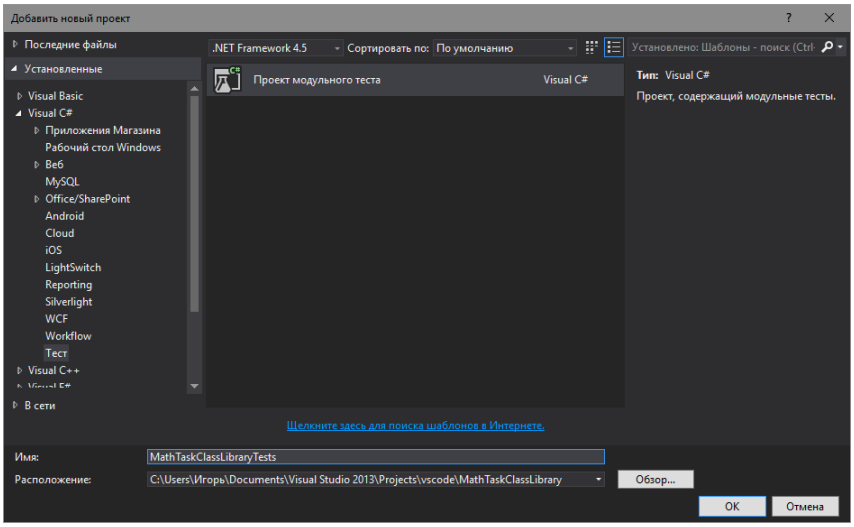


Рисунок 4

Перед Вами появится следующий код:

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace MathTaskClassLibraryTests
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

Рисунок 5

Директива [TestMethod] обозначает, что далее идёт метод, содержащий модульный (unit) тест. А [TestClass] в свою очередь говорит о том, что далее идёт класс, содержащий методы, в которых присутствуют unit-тесты.

В соответствии с принятыми соглашениями переименуем класс UnitTest1 в GeometryTests.

Затем в References проекта необходимо добавить ссылку на проект, код которого будем тестировать. Правой кнопкой щёлкаем на References, а затем выбираем “Добавить ссылку…”.

В появившемся окне раскрываем группу “Решение”, выбираем “Проекты” и ставим галочку напротив проекта MathTaskClassLibrary. Затем жмём “ОК”.

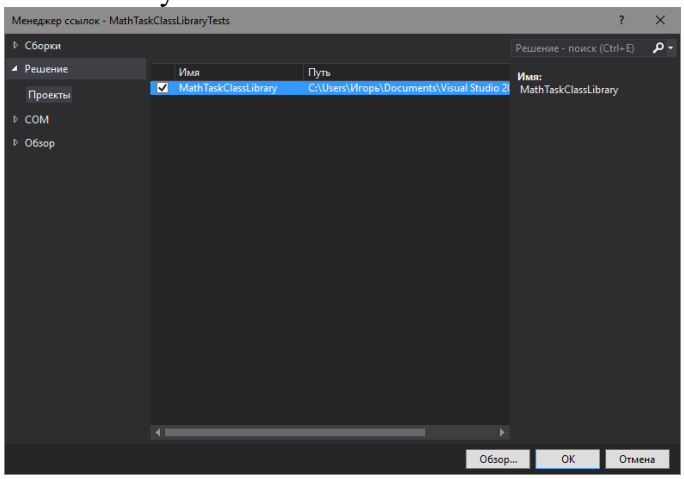


Рисунок 6

Также в коде необходимо подключить с помощью директивы using следующее пространство имён: using MathTaskClassLibrary;

Займёмся написание теста. Проверим правильно ли вычисляет программа площадь прямоугольника со сторонами 3 и 5. Ожидаемый результат (правильное решение) в данном случае это число 15.

Переименуем метод TestMethod1() в RectangleArea_3and5_15returned(). Новое название метода поясняет, что будет проверяться (RectangleArea – площадь прямоугольника) для каких значений (3 и 5) и что ожидается в качестве правильного результата (15 returned).

Тестирующий метод обычно содержит три необходимых компонента:

1. исходные данные: входные значения и ожидаемый результат;
2. код, вычисляющий значение с помощью тестируемого метода;
3. код, сравнивающий ожидаемый результат с полученным.

Соответственно тестирующий код будет таким:

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using MathTaskClassLibrary;

namespace MathTaskClassLibraryTests
{
    [TestClass]
    public class GeometryTests
    {
        [TestMethod]
        public void RectangleArea_3and5_15returned()
        {
            // исходные данные
            int a = 3;
            int b = 5;
            int expected = 15;

            // получение значения с помощью тестируемого метода
            Geometry g = new Geometry();
            int actual = g.RectangleArea(a, b);

            // сравнение ожидаемого результата с полученным
            Assert.AreEqual(expected, actual);
        }
    }
}
```

Рисунок 7

Для сравнения ожидаемого результата с полученным используется метод AreEqual класса Assert. Данный класс всегда используется при написании unit тестов в Visual Studio.

Теперь, чтобы просмотреть все тесты, доступные для выполнения, необходимо открыть окно “Обозреватель тестов”. Для этого в меню Visual Studio щёлкните на кнопку “ТЕСТ”, выберите “Окна”, а затем нажмите на пункт “Обозреватель тестов”.

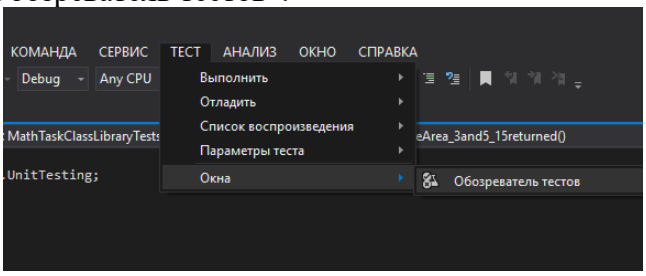


Рисунок 8

В студии появится следующее окно:

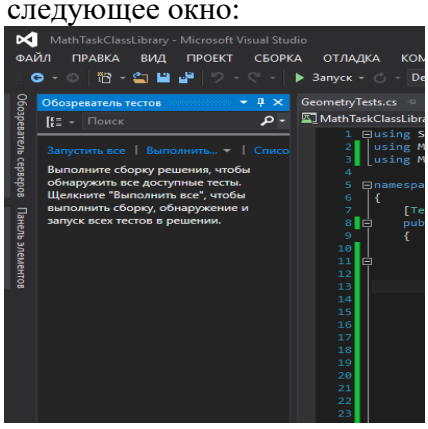


Рисунок 9

В данный момент список тестов пуст, поскольку решение ещё ни разу не было собрано. Выполним сборку нажатием клавиш Ctrl + Shift + B. После её завершения в “Обозревателе тестов” появится наш тест.

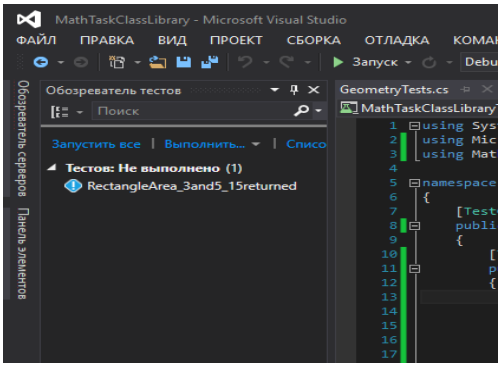


Рисунок 10

Синяя табличка с восклицательным знаком означает, что указанный тест никогда не выполнялся. Выполним его. Для этого нажмём правой кнопкой мыши на его имени и выберем “Выполнить выбранные тесты”.

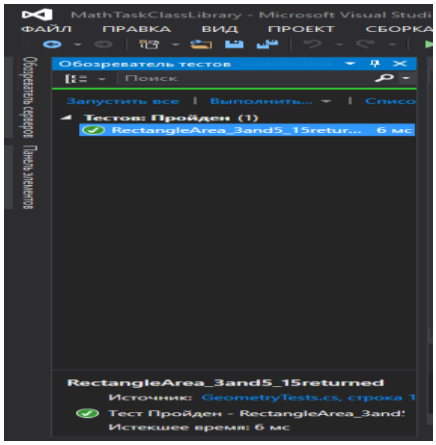


Рисунок 11

Зелёный кружок с галочкой означает, что модульный тест успешно пройден: ожидаемый и полученный результаты равны. Изменим код метода RectangleArea, вычисляющего площадь прямоугольника, чтобы сымитировать провал теста и посмотреть, как поведёт себя Visual Studio. Прибавим к возвращаемому значению 10.

Запустим unit-тест.

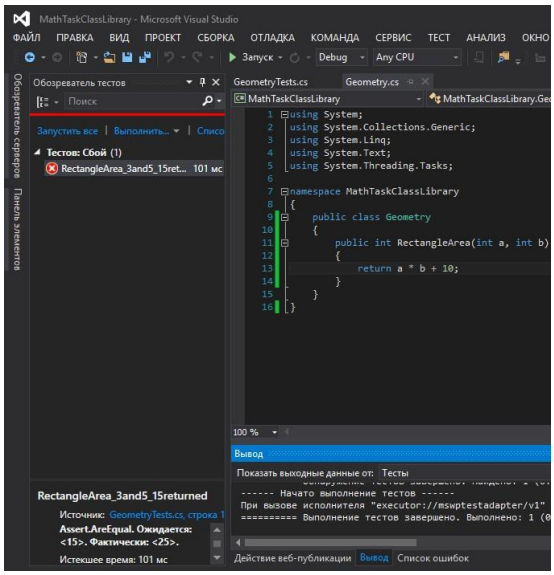


Рисунок 12

Как Вы видите, красный круг с крестиком показывает провал модульного теста, а ниже указано, что при проверке ожидалось значение 15, а по факту оно равно 25.

Задание 2. Разработать программу для подсчета объема цилиндра и создать модульный тест.

Практическая работа 21. Разработка процедур генерации тестовых наборов данных программного кода ПО.

Цель работы: ознакомление с аппаратными и программными средствами отладки ПО; изучение команд отладчика среды AVR Studio; приобретение навыков отладки программ под управлением отладчика.

Основные сведения

Особенность отладки ПО устройств на базе встраиваемых МП (в том числе однокристальных микроконтроллеров) состоит в отсутствии в их составе развитых средств для реализации пользовательского интерфейса и ограниченных возможностях системного ПО. В то же время именно для встраиваемых микропроцессорных систем этап отладки является чрезвычайно ответственным, так как для них характерна тесная взаимосвязь работы ПО и аппаратных средств.

Взаимодействие микропроцессора (микроконтроллера) с датчиками и исполнительными устройствами происходит путём передачи данных через регистры периферийных устройств (регистры ввода-вывода). Отдельные разряды таких регистров задают режимы работы периферийных устройств, имеют смысл готовности к обмену, завершения передачи данных и т. п. Состояние этих разрядов может устанавливаться как программно, так и аппаратно. При отладке ПО часто приходится переходить на уровень межрегистровых передач и проверять правильность установки отдельных разрядов. Кроме того, на этапе отладки может производиться оптимизация алгоритма, нахождение критических участков кода и проверка надёжности разработанного ПО.

Для решения указанных задач применяются аппаратные и программные средства отладки ПО

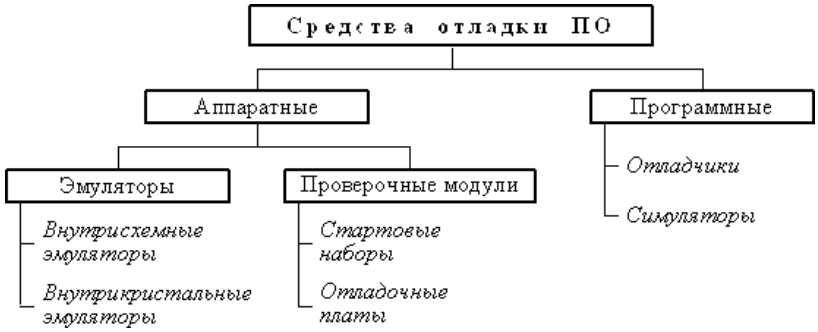


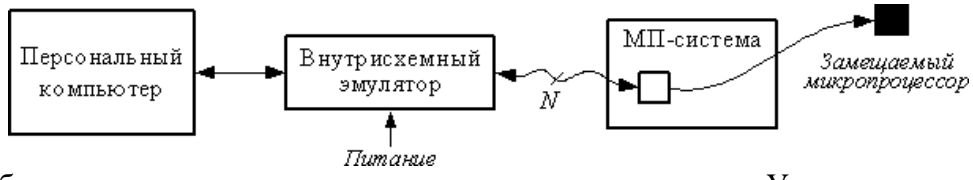
Рисунок 14

К аппаратным средствам отладки относятся аппаратные эмуляторы и проверочные модули.

Аппаратные эмуляторы предназначены для отладки программного и аппаратного обеспечения микропроцессорных систем в режиме реального времени. Они работают под управлением «ведущего» компьютера, оснащённого специальным ПО – программами- отладчиками (см. ниже). Основными видами аппаратных эмуляторов являются:

- внутрисхемные эмуляторы или эмуляторы-приставки, замещающие микропроцессор в отлаживаемой системе;
- внутрикристальные эмуляторы, представляющие собой одно из внутренних устройств микропроцессора.

Внутрисхемный эмулятор (In-Circuit Emulator, ICE) – это устройство, содержащее аппаратный имитатор процессора и схему управления имитатором. При отладке с помощью эмулятора микропроцессор извлекается из отлаживаемой системы, на его место подключается контактная колодка, количество и назначение контактов которой идентично выводам замещаемого микропроцессора. С



помощью гибкого кабеля контактная колодка соединятся с эмулятором. Управление процессом отладки осуществляется с персонального компьютера. Эмуляторам-приставкам присущи следующие недостатки: высокая стоимость, недостаточная надёжность, высокое энергопотребление, влияние на электрические характеристики цепей, к которым подключается эмулятор.

Рисунок 15

Внутрикристальные эмуляторы (On-Chip Emulator) позволяют проводить отладку программ без извлечения микропроцессора из системы. При этом осуществляется непосредственный контроль за выполнением программы, так как средства внутрикристальной отладки обеспечивают прямой доступ к регистрам, памяти и периферии микропроцессора. Наиболее распространённым средством внутрикристальной отладки является последовательный интерфейс IEEE 1149.1, известный как JTAG (Joint Test Action Group – Объединённая рабочая группа по автоматизации тестирования). Последовательный отладочный порт JTAG микропроцессора с помощью специального устройства сопряжения подключается к компьютеру, чем обеспечивается доступ к отладочным средствам процессора. Такой способ отладки также называют сканирующей эмуляцией. Достоинствами этого способа являются возможность выполнения различных действий на процессоре без его изъятия из системы, использование малого числа выводов процессора и поддержка его максимальной производительности без изменения электрических характеристик системы.

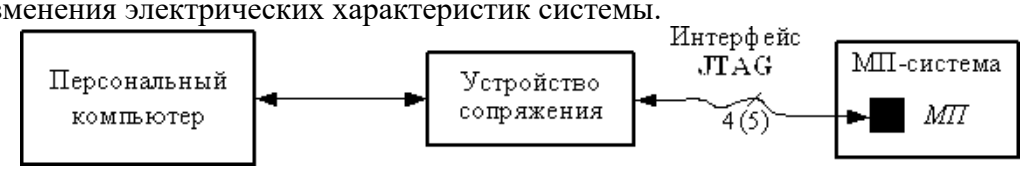


Рисунок 16

Проверочные модули предназначены для быстрой отладки программного обеспечения в реальном масштабе времени. Проверочные модули бывают двух видов: стартовые наборы и отладочные платы.

Стартовые наборы (Starter Kit) предназначены для обучения работе с конкретным микропроцессором. Стартовый набор позволяет изучить характеристики микропроцессора, отладить не слишком сложные программы, выполнить несложное макетирование, проверить возможность применения микропроцессора для решения конкретной задачи. В состав стартового набора входят плата, ПО и комплект документации. На плате устанавливаются микропроцессор, устройство загрузки программ, последовательные или параллельные порты, разъёмы для связи с внешними устройствами и другие элементы. Плата подключается к компьютеру через параллельный или последовательный порт. Стартовые наборы удобны на начальном этапе работы с микропроцессором.

Отладочные платы (Evaluation Board) предназначены для проверки разработанного алгоритма в реальных условиях. Они позволяют проводить отладку и оптимизацию алгоритма с использованием установленной на плате периферии, а также изготовить на базе платы законченное устройство. Обычно на плате размещаются микропроцессор, схемы синхронизации, интерфейсы расширения памяти и периферии, схема электропитания и др. Плата подключается к компьютеру через параллельный или последовательный порт или непосредственно устанавливается в слот PCI.

Основными программными средствами отладки являются симуляторы и отладчики.

Симуляторы (simulator) или симуляторы системы команд представляют собой программы, имитирующие работу того или иного процессора на уровне его команд. Симуляторы обычно используются для проверки программы или её отдельных частей перед испытанием на аппаратных средствах.

Отладчики (debugger) представляют собой программы, предназначенные для анализа работы созданного программного обеспечения. Можно указать следующие возможности отладчиков.

1. Пошаговое выполнение. Программа выполняется последовательно, команда за командой, с возвратом управления отладчику после каждого шага.
2. Прогон. Выполнение программы начинается с указанной команды и осуществляется без остановки до конца программы.
3. Прогон с контрольными точками. При выполнении программы происходит останов и передача управления отладчику после выполнения команд с адресами, указанными в списке контрольных точек.
4. Просмотр и изменение содержимого регистров и ячеек памяти. Пользователь имеет возможность выводить на экран и изменять (модифицировать) содержимое регистров и ячеек памяти.

Отладчики ПО встраиваемых микропроцессоров обычно используются совместно с внутрисхемными или внутрикристальными эмуляторами, а также могут работать в режиме симулятора. Некоторые отладчики позволяют также выполнять профилирование, т. е. определять действительное время выполнения некоторого участка программы. Иногда функцию профилирования

выполняет специальная программа – профилировщик (profiler).

Средства отладки ПО AVR-микроконтроллеров. Аппаратные средства отладки программного обеспечения AVR-микроконтроллеров представлены внутрисхемным эмулятором ICE50, внутрикристальным эмулятором JTAG ICE, а также стартовым набором STK500.

К программным средствам отладки ПО AVR-микроконтроллеров относятся отладчик и симулятор, входящие в состав среды AVR Studio. Отладчик среды AVR Studio позволяет проводить отладку программ как в исходных кодах (например, ассемблера), так и в кодах дизассемблера (оттранслированной или скомпилированной программы, записанной с помощью мнемоник ассемблера). Вызов окна с кодом дизассемблера производится командой Disassembler меню View или командой Goto Disassembly контекстного меню редактора исходного текста. Обратное переключение в окно исходного текста осуществляется командой Goto Source контекстного меню окна Disassembler.

Отладчик среды AVR Studio может использоваться с внутрисхемным эмулятором ICE50, внутрикристальным эмулятором JTAG ICE, отладочной платой STK500 или симулятором. Указание способа отладки производится при создании проекта. Симулятор среды AVR Studio предназначен для предварительной отладки программ без применения аппаратных средств. В дальнейшем в настоящем лабораторном практикуме для отладки создаваемых программ предполагается применение отладчика среды AVR Studioв режиме симулятора.

Отладка ПО в среде AVR Studio. Команды отладчика в программе AVR Studio находятся в меню Debug.

Переход в режим отладчика в средеAVR Studio осуществляется автоматически при использовании для трансляции программы команды Build and Run или командой Start Debugging меню Debug при использовании для трансляции команды Build. Выход из режима отладчика производится командой Stop Debugging меню Debug.

Пошаговое выполнение программы задаётся командами Step Into, Step Over меню Debug. Команда Step Into позволяет выполнить одну команду программы (в том числе команду вызова подпрограммы). Для завершения выполнения подпрограммы может использоваться команда Step Out. Команда Step Over также выполняет одну команду программы, но если это команда вызова подпрограммы, последняя полностью выполняется за один шаг. Следующая выполняемая команда (команда, адрес которой содержится в программном счётчике) обозначается символом в окне исходного текста программы. Сброс выполнения программы осуществляется с помощью команды Reset.

Прогон (запуск или продолжение выполнения) программы осуществляется командой Run. Для остановки выполнения программы служит команда Break.

Контрольные точки представляют собой специальные маркеры для программы- отладчика и могут быть трёх типов: точки останова, точки трассировки и точки наблюдения.

Точки останова задаются командой Toggle Breakpoint меню Debug или контекстного меню редактора исходного текста программы. Точка останова обозначается в редакторе исходного текста символом слева от помечаемой строки. Просмотреть заданные точки останова можно на закладке Breakpoints окна Output; там же точки останова могут быть запрещены (путём сброса флажка напротив точки останова) и разрешены (путём установки флажка). При достижении точки останова во время прогона программы её выполнение приостанавливается. Повторный вызов команды установки точки останова на той же строке программы приводит к удалению точки останова. Удалить все заданные точки останова позволяет команда Remove Breakpoints меню Debug или команда Remove all Breakpoints контекстного меню закладки Breakpoints окна Output. Параметры точки останова задаются в диалоговом окне Breakpoint Condition, вызов которого осуществляется командой Breakpoints Properties контекстного меню редактора исходного текста программы. Установка флажка Iterations позволяет задать количество итераций (повторных выполнений) команды до останова прогона программы. При установке флажка Watchpoint по достижению точки останова производится только обновление значений регистров и ячеек памяти в окнах просмотра. Флажки Iterations и Watchpoint не должны устанавливаться одновременно. Установка флажка Show message обеспечивает отображение сообщений о достижении точки останова на закладке Breakpoints окна Output. Вызов диалогового окна задания свойств и удаление точки останова могут быть произведены из контекстного меню закладки Breakpoints окна Output.

Точки трассировки предназначены для контроля выполнения программы в режиме реального времени. Трассировка позволяет отслеживать так называемую трассу программы – изменение

содержимого регистров и ячеек памяти при выполнении определённых команд (команд, по адресам которых заданы точки трассировки). В среде AVR Studio функция трассировки может использоваться только при отладке программы с применением внутрисхемного эмулятора; при работе в режиме симулятора функция трассировки недоступна.

Точки наблюдения задаются командой Add to Watch контекстного меню редактора исходного текста программы. Точки наблюдения представляют собой символические имена регистров или ячеек памяти, содержимое которых необходимо отслеживать. При выполнении команды Add Watch на экране появляется окноWatches, разделённое на четыре столбца: Name (символическое имя точки наблюдения), Value (значение), Type (тип), Location (местонахождение). Новая точка наблюдения может быть также задана в выделенной ячейке столбца Name окна Watches или командой Quickwatch в окне редактора исходного текста программы (при этом курсор должен находиться на имени регистра или ячейки памяти). Значения, отображаемые в столбце Value, обновляются при изменении содержимого соответствующего регистра или ячейки памяти. Удалить заданные точки наблюдения можно из окна Watches.

Отладчик среды AVR Studio также обеспечивает следующие функции: выполнение до курсора (команда Run to Cursor меню Debug) и последовательное выполнение команд с паузами между ними (команда Auto Step меню Debug).

Для удобства использования в процессе отладки ряд команд отладчика доступен с клавиатуры (табл. 3).

Таблица 3

Команда отладчика	Клавиша	Команда отладчика	Клавиша
Run	F5	Step Into	F11
Break	Ctrl+F5	Step Out	Shift+F11
Reset	Shift+F5	Step Over	F10
Run to Cursor	Ctrl+F10	Toggle Breakpoint	F9

Для просмотра и изменения содержимого регистров и ячеек памяти служат команды Registers, Memory, Memory 1, Memory 2, Memory 3 меню View.

По команде Registers на экране отображается окно Registers, в котором приводятся шестнадцатеричные представления содержимого РОН. Изменение (модификация) содержимого регистров производится путём двойного щелчка мышью. Наблюдение за содержимым РОН может быть также произведено с помощью дерева устройств микроконтроллера, находящегося на закладке I/O окна Workspace. Для этого необходимо раскрыть объекты Register 0-15 и

Register 16-31 щелчком мыши по знаку «+».

Команды Memory, Memory 1, Memory 2, Memory 3 обеспечивают вызов оконMemory, служащих для отображения содержимого ячеек оперативной и энергонезависимой памяти данных, памяти программ, регистров ввода-вывода и РОН. Выбор типа памяти, отображаемой в окне Memory, производится с помощью списка, расположенного в панели управления окна (Data – оперативная память данных, Eeprom – энергонезависимая память данных, I/O – регистры ввода-вывода, Program – память программ, Register – РОН).

Для наблюдения за состоянием процессора необходимо раскрыть объект Processor закладки I/O окна Workspace. При этом будет отображена следующая информация: содержимое программного счётчика (Program Counter); содержимое указателя стека (Stack Pointer), количество тактов, прошедших с начала выполнения (Cycle Counter); содержимое 16- разрядных регистров-указателей X, Y и Z; тактовая частота (Frequency); затраченное на выполнение время (Stop Watch).

Для контроля содержимого регистров ввода-вывода необходимо раскрыть объект I/O * закладки I/O окна Workspace, где * – тип микроконтроллера. Регистры ввода-вывода, входящие в объект I/O, сгруппированы по типам периферийных устройств.

Модифицированные значения содержимого регистров и ячеек памяти действуют только во время текущего сеанса отладки, в исходный текст программы изменения не заносятся.

ЗАДАНИЕ

Провести отладку созданной в программы с помощью симулятора-отладчика среды AVR Studio, проделав следующие операции.

1. Выполнить программу в пошаговом режиме, отслеживая изменение содержимого используемых в программе регистров. Обратить внимание на изменение содержимого программного

счётчика. Сравнить содержимое программного счётчика при выполнении команд с их адресами в памяти программ, приведёнными в листинге трансляции и окне памяти программ.

2. Выполнить прогон программы. Проверить правильность результата работы программы.

3. Задать точку останова на команде загрузки в РОН числа B. Включить режим отображения сообщений о достижении точки останова. Выполнить прогон программы с контрольными точками. Задать точку останова на команде умножения. Выполнить прогон программы с контрольными точками. Удалить заданные точки останова.

4. Задать точки наблюдения в используемых РОН. Выполнить программу в пошаговом режиме, отслеживая изменение их содержимого.

СОДЕРЖАНИЕ ОТЧЁТА

Отчёт должен содержать: титульный лист с указанием номера и названия лабораторной работы, номера группы и фамилий выполнивших работу; цель работы; краткие теоретические сведения (классификацию средств отладки ПО, перечень основных функций программ- отладчиков); список использованных команд отладчикаAVR Studio с указанием их назначения.

Контрольные вопросы

1. Классификация средств отладки прикладного ПО встраиваемых МП.
2. Виды и особенности аппаратных средств отладки ПО.
3. Основные функции программных средств отладки ПО.
4. Пошаговое выполнение программы и его возможности.
5. Особенности прогона программы с контрольными точками.
6. Контрольные точки: типы, назначение, использование.

Практическая работа 22. Интерпретация и диагностика данных программного кода ПО.

Цель работы: Научиться создавать инсталляционные файлы; выполнять оценочное тестирование программного продукта.

Теоретическая часть

После завершения комплексного тестирования приступают к оценочному тестированию, целью которого является тестирование программы на соответствие основным требованиям. Эта стадия тестирования особенно важна для программных продуктов, предназначенных для продажи на рынке.

Оценочное тестирование, которое также называют «тестированием системы в целом», включает следующие виды:

- тестирование удобства использования - последовательная проверка соответствия программного продукта и документации на него основным положениям технического задания;
- тестирование на предельных объемах - проверка работоспособности программы на максимально больших объемах данных, например, объемах текстов, таблиц, большом количестве файлов и т. п.;
- тестирование на предельных нагрузках - проверка выполнения программы на возможность обработки большого объема данных, поступивших в течение короткого времени;
- тестирование удобства эксплуатации - анализ психологических факторов, возникающих при работе с программным обеспечением; это тестирование позволяет определить, удобен ли интерфейс, не раздражает ли цветовое или звуковое сопровождение и т. п.;
- тестирование защиты - проверка защиты, например, от несанкционированного доступа к информации;
- тестирование производительности - определение пропускной способности при заданной конфигурации и нагрузке;
- тестирование требований к памяти - определение реальных потребностей в оперативной и внешней памяти;
- тестирование конфигурации оборудования - проверка работоспособности программного обеспечения на разном оборудовании;
- тестирование совместимости - проверка преемственности версий: в тех случаях, если очередная версия системы меняет форматы данных, она должна предусматривать специальные конвекторы,

обеспечивающие возможность работы с файлами, созданными предыдущей версией системы;

тестирование удобства установки - проверка удобства установки;

тестирование надежности - проверка надежности с использованием соответствующих математических моделей;

тестирование восстановления - проверка восстановления программного обеспечения, например системы, включающей базу данных, после сбоев оборудования и программы;

тестирование удобства обслуживания - проверка средств обслуживания, включенных в программное обеспечение;

тестирование документации - тщательная проверка документации, например, если документация содержит примеры, то их все необходимо попробовать;

тестирование процедуры - проверка ручных процессов, предполагаемых в системе и др.

Естественно, целью всех этих проверок является поиск несоответствий техническому заданию. Считают, что только после выполнения всех видов тестирования программный продукт может быть представлен пользователю или к реализации. Однако на практике обычно выполняют не все виды оценочного тестирования, так как это очень дорого и трудоемко. Как правило, для каждого типа программного обеспечения выполняют те виды тестирования,которые являются для него наиболее важными. Так базы данных обязательно тестируют на предельных объемах, а системы реального времени - на предельных нагрузках.

Системы для создания инсталляторов

Практика разработки коммерческого программного обеспечения показывает, что далеко не все пользователи умеют работать с архивами. Поэтому программы рекомендуется поставлять в виде исполняемых файлов, которые автоматически создают необходимые папки в файловой системе, копируют туда файлы программы, создают необходимые файлы настроек или ключи в реестре, а так же пункты меню запуска программы и ярлыки на рабочем столе. Для упрощения создания инсталляторов существует много специализированных программных продуктов.

Знакомство пользователя с программой чаще всего начинается с запуска инсталлятора. Внешний вид («упаковка») и функциональность продукта определяется разработчиком. Пользователю нужно иметь возможность проконтролировать процесс, выставив нужные параметры установки. Для разработчика же важно, чтобы, как минимум, его программа была установлена корректно, а инсталлятор был совместим с необходимыми платформами.

Решений для создания инсталляторов достаточно много. Чаще всего используется подсистема Windows Installer, которая уже входит в инструментарий операционной системы. Но существуют и альтернативные решения – как платные, так и бесплатные, различной функциональности. Зачастую с их помощью можно создавать пакеты с инсталлятором, не зависящим от Windows Installer.

Основные критерии выбора системы создания инсталлятора следующие:

- среда разработки, интерфейс, поддержка сценариев;
- работа с проектом, типы создаваемых пакетов, возможности импорта проектов из других сред разработки;
- пользовательские опции инсталлятора: поддержка языков, профилей и другие опции;
- поддержка расширений.

Свободные программы для создания инсталляторов:

- NSIS (Nullsoft Scriptable Install System) – один из самых популярных инсталляторов. Обладает богатыми возможностями, которые присутствуют в большинстве коммерческих продуктов. Позволяет устанавливать различные параметры сжатия при создании дистрибутива;
- IzPack – java инсталлятор. Это универсальный инсталлятор, способен создавать дистрибутивы для Unix, Linux, FreeBSD, Mac OS X и Windows 2000, XP. Позволяет создавать как обычные пакеты инсталляции, так и Web инсталляторы, которые подгружают необходимые файлы по мере необходимости. Данная возможность позволяет свести к минимуму количество загружаемых файлов в зависимости от требуемой конфигурации установки;
- Inno Setup –довольно популярный простой инталлятор. Содержит встроенный скриптовой язык;
- WiX (Windows Installer XML) – специализированный продукт от Microsoft для создания MSI и MSM инсталляционных пакетов.

Коммерческие программы для создания инсталляторов:

– InstallShield – один из самых известных продуктов в ряду инсталляторов;

– WISE – простой в освоении с богатыми возможностями генератор инсталляторов;

– VISE - профессиональный инсталлятор для Windows, MacOS X и Macintosh;

– CreateInstall это универсальный, гибкий и мощный инсталлятор как для профессиональных разработчиков, так и для начинающих. С помощью этой программы Вы можете создать полнофункциональные инсталляционные программы для Ваших приложений, а также самораспаковывающиеся архивы с высокой степенью сжатия и многое другое;

– Advanced Installer – позволяет создавать инсталляторы для java приложений. Создает дополнительный исполняемый файл.

CreateInstall

Домашняя страница: http://www.createinstall.ru/

CreateInstall – инструментарий для создания установщиков. В его основу заложено две особенности – контроль над процессом установки и неограниченная расширяемость. Обе возможности реализованы благодаря языку программирования Gentee, применяемому для написания сценариев.

Интерфейс CreateInstall разбит на 3 вкладки – «Проект», «Скрипт установки» и «Скрипт деинсталляции». Первый раздел позволяет задать общие настройки инсталлятора: информация о продукте, поддерживаемые языки, пути, внешний вид. Дополнительно, инсталлятор можно защитить цифровой подписью и установить пароль.

«Проект» – не равноценная замена двух последующих разделов, т. е. для создания дистрибутива нужно тщательно настроить скрипты установки и деинсталляции. Соответствующие параметры отображаются в виде групп, можно отобразить их единым списком.

Дополнением для CreateInstall служит утилита Quick CreateInstall (рисунок 16). Она значительно упрощает создание инсталлятора, предоставляя только базовые настройки проекта. Из Quick CreateInstall в дальнейшем проект можно импортировать в CreateInstall.

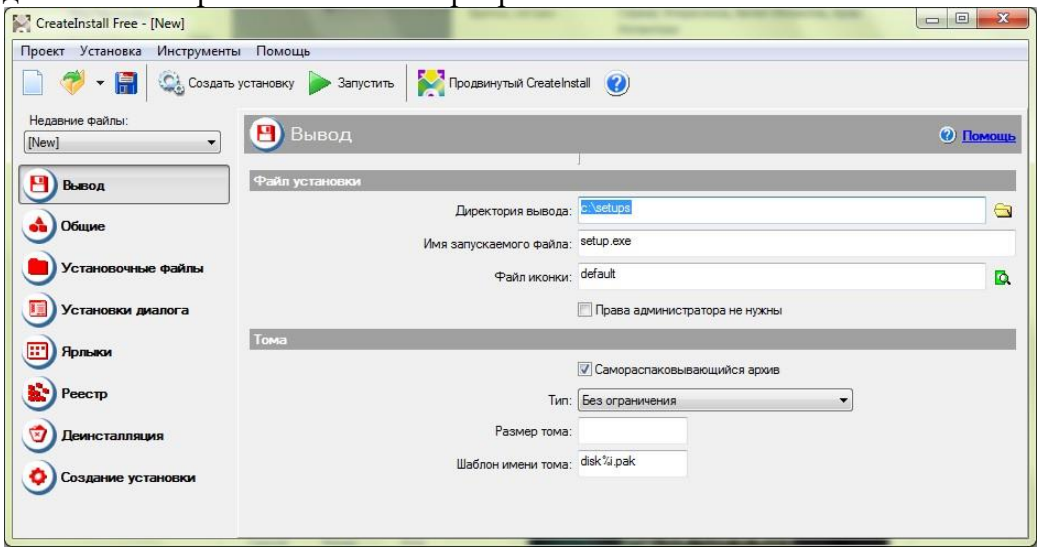


Рисунок 17

Код проекта не предназначен для самостоятельного редактирования, переноса в IDE- среду, экспорта. Хотя язык Gentee имеет отличный потенциал: как минимум, это переменные и функции, условные выражения и синтаксис, базирующийся на C, C++ и Java.

Существует 3 редакции программы – полная, light (простая) и бесплатная. Интерфейс и справка доступны на русском языке.

Advanced Installer

Advanced Installer основывается на технологии Windows Inslaller, позволяя создавать msi-, exe- и других видов дистрибутивов. Этому способствует продуманный интерфейс и работа с проектами. В Advanced Installer можно обнаружить немало возможностей, которых нет в других подобных комплексах.

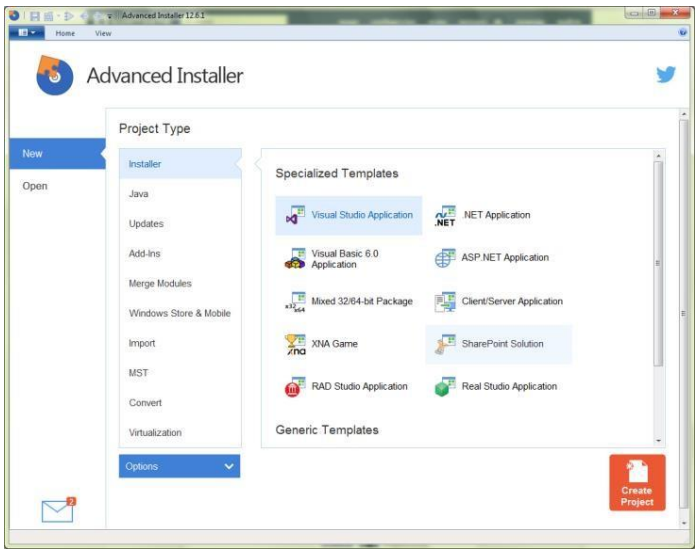


Рисунок 18

Примечательно, прежде всего, разнообразие проектов: сюда входят инсталляторы, Java- установщики, обновления, дополнения, модули слияния и другие. В разделе меню Installer собраны команды импорта проектов из Visual Studio, RAD Studio, Real Studio, Visual Basic. Здесь раскрывается потенциал Advanced Installer во взаимодействии с IDE-средами.

Для каждого из выбранных типов проекта предусмотрен детальный мастер настройки. Есть общие шаблоны – Simple, Enterprise, Architect или Professional. Большая часть проектов доступна только для определенных типов лицензии, общедоступные проекты обозначены как None в графе License Required.

Как уже сказано, при создании проекта можно воспользоваться пошаговым мастером, где, в частности, доступен выбор способа распространения пакета, языков локализации, настройка пользовательского интерфейса, ввод текста лицензии и другие опции. Advanced Installer позволяет выбрать вариант распространения программы – оставить данные без компрессии, разделить на CAB-архивы, сохранить в MSI и др., добавить цифровую подпись, потребовать ввод серийного номера и т. д.

Главное окно Advanced Installer (редактор проекта), в простом режиме отображения (Simple), содержит несколько секций:

– Product Information (Информация о продукте) – ввод сведений о продукте, параметры установки.

– Requirements (Требования) – указание аппаратных и системных требований, зависимостей ПО. Также имеется возможность создания пользовательских условий.

– Resources (Ресурсы) – редактор ресурсов (файлов и ключей реестра).

– Deployment (Развертывание) – выбор типа распространения продукта. Это может быть MSI, EXE или веб-инсталлятор. Для MSI, EXE ресурсы можно поместить отдельно от инсталлятора.

– System Changes – переменные среды.

При выборе ресурсов могут использоваться файлы, ключи реестра, переменные окружения, конфигурационные ini, драйверы, базы данных и переводы. С помощью модулей объединения можно добавить и другие ресурсы, такие как сервисы, разрешения, ассоциации и др.

Для выполнения более сложных задач позволяется использовать пользовательские действия, EXE, DLL или скрипты, написанные на C, C++, VBS или JS. Для создания сценариев предусмотрен удобный редактор.

Однако следует отметить, что в режиме Simple доступна лишь малая часть разделов. Работая с Advanced Installer в ознакомительном режиме, есть смысл зайти в настройки и переключиться в другой режим работы с проектом. После этих действий

становятся доступны новые подразделы редактора.

Задание

1. С помощью системы создания инсталляторов создайте из программы, созданной ранее, установочный файл.
2. Выполните тестирование удобства установки.
3. Выполните тестирование конфигурации оборудования.
4. Выполните тестирование восстановления.
5. Выполните тестирование удобства эксплуатации при помощи соседа.
6. Результаты выполнения практического задания

запишите в отчет. Контрольные вопросы

1. Что является целью тестирования программ?
2. Какие подходы к тестированию вы знаете? В чем они заключаются?
3. Обоснуйте необходимость создания инсталляторов программ.

Контрольно-измерительный материал

1. Программный продукт – это

1) программа для удовлетворения нужд разработчиков, предназначенная для продажи
2) комплекс взаимосвязанных программ для решения определенной проблемы массового спроса, подготовленный к реализации как любой вид промышленной продукции
3) программная реализация решения задачи на компьютере
4) результат разработки какого-либо технического задания

2. Отличительной особенностью программных продуктов является

1) системность
2) простота
3) универсальность
4) надежность

3. Сопровождение программного продукта – это

1) снабжение программного продукта необходимой документацией
2) обнаружение и исправление ошибок
3) поддержка работоспособности программного продукта, переход на его новые версии, внесение изменений, исправление обнаруженных ошибок и т.д.
4) проверка работоспособности каждой разработанной функции, процедуры, модуля

4. Мобильность программных продуктов – это

1) независимость от технического комплекса системы обработки данных, операционной среды, сетевой технологии обработки данных, специфики предметной области и
т.п.
2) точность выполнения предписанных функций обработки
3) способность к внесению изменений
4) обеспечение дружественного интерфейса для работы конечного пользователя, наличие контекстно-зависимой подсказки или обучающей системы в составе программного средства

5. В условиях существования рынка программных продуктов важными его характеристиками являются:

1) количество продаж, наличие программ-конкурентов, длительность продаж
2) стоимость, количество продаж, время нахождения на рынке, известность фирмы

разработчика и программы
3) внешний интерфейс программы, количество продаж, наличие программ конкурентов
4) модифицируемость, надежность, универсальность, известность фирмы – разработчика

6. Основными показателями качества программных продуктов является:
1) алгоритмическая сложность, полнота и системность функций обработки, объем файлов программы
2) стоимость, количество продаж, наличие программных продуктов аналогичного назначения
3) мобильность, надежность, эффективность, модифицируемость, коммуникативность, учет человеческого фактора
4) модифицируемость, надежность, наличие программных продуктов аналогичного назначения

7. При индивидуальной разработке фирма-разработчик создает программный продукт для…
1) конкретного заказчика
2) массового использования
3) внедрения в специальные организации
4) для удовлетворения собственных нужд

8. Модифицируемость программных продуктов означает…
1) независимость от технического комплекса системы обработки данных, операционной среды, сетевой технологии обработки данных, специфики предметной области и т.п.
2) точность выполнения предписанных функций обработки
3) способность к внесению изменений, например расширение функций обработки, переход на другую техническую базу обработки и т.п.
4) обеспечение дружественного интерфейса для работы конечного пользователя, наличие контекстно-зависимой подсказки или обучающей системы в составе программного средства

9. Жизненный цикл программы – это
1) временной интервал, начиная с момента замысла программы и кончая прекращением всех видов его пользований
2) временной интервал, начиная с момента введения программы в эксплуатацию
3) промежуток времени, который определяет наиболее эффективное использование создаваемой программы
4) временная характеристика разработки программного продукта

10. Программы малого Жизненного Цикла – это программы
1. когда время разработки программы значительно меньше времени эксплуатации программы
2. когда время разработки программы значительно больше времени использования программы
3. когда время разработки программы равно времени эксплуатации программы
4. нет правильного ответа

11. На этапе сбора и анализа требований заказчик должен
a. выяснить, прежде всего, необходимость обеспечения безопасности системы и данных
b. выяснить, прежде всего, функции, которые должен выполнять программный продукт
c. выяснить, прежде всего, сроки написания программы
d. собрать литературу по разрабатываемому программному продукту

12. Самая распространенная модель Жизненного цикла программного продукта это
1) итерационная
2) V - образная
2) спиральная
3) каскадная

13. Классическая модель ЖЦПО характеризуется следующими основными особенностями
1. последовательным выполнением входящих в ее состав этапов
2. наличием обратных связей между этапами
3. отсутствием временного перекрытия этапов
4. отсутствием (или определенным ограничением) возврата к предыдущим этапам
5. наличием результата после каждого этапа разработки

14. Выберите правильную последовательность этапов спиральной модели жизненного цикла программного продукта:
1) техническое проектирование, сопровождение ПП, сбор и анализ требований заказчика, кодирование, уточнение функциональных характеристик, тестирование и отладка
2) кодирование, техническое проектирование, уточнение функциональных характеристик, сопровождение ПП, тестирование и отладка
3) кодирование, техническое проектирование, уточнение функциональных характеристик, тестирование и отладка
4) определение требований, анализ, реализация и тестирование, внедрение

15. V – образная модель ЖЦ разработки ПО предполагает:
1. отсутствие временного перекрытия этапов
2. наличие обратной связи
3. возможность сокращения времени разработки ПО
4. возможность увеличения жизненного цикла программного продукта

16. На втором этапе каскадной модели ЖЦ разработки ПО (Требования ПО) осуществляется…
1. составление концептуальной структуры системы
2. определение функциональности программного компонента
3. составление детальной спецификации архитектуры системы
4. составление набора тестовых данных

17. Проверка корректности требований при использовании V – образной модели ЖЦ разработки ПО осуществляется…
1. после каждого этапа разработки
2. после разработки всей системы
3. после разработки черновой версии системы
4. после разработки набора тестовых данных

18. Выберите правильную последовательность этапов жизненного цикла программного продукта:
1) техническое проектирование, сопровождение ПП, сбор и анализ требований заказчика, кодирование, уточнение функциональных характеристик, тестирование и отладка
2) сбор и анализ требований, проектирование системы, кодирование, создание программной документации, сопровождение

3) кодирование, сбор и анализ требований заказчика, техническое проектирование, уточнение функциональных характеристик, сопровождение ПП, тестирование и отладка
4) сбор и анализ требований заказчика, уточнение функциональных характеристик, техническое проектирование, кодирование, тестирование и отладка, сопровождение ПП

19. Во вспомогательные процессы ЖЦ программного продукта входит:
1) документирование, верификация, аттестация, обеспечение качества, совместная оценка, разрешение проблем, аудит
2) управление, создание инфраструктуры, усовершенствование, обучение
3) разработка, приобретение, поставка, эксплуатация, сопровождение
4) кодирование, тестирование, сопровождение

20. Одним из достоинств классического жизненного цикла программного продукта является
1) дает план и временной график по всем этапам проекта
2) в конце всей работы заказчику будут доступны результаты проекта
3) системный анализ каждого элемента программы
4) отсутствие временного перекрытия этапов разработки программного продукта

21. Итерационная модель ЖЦПО характеризуется следующими основными особенностями:
1. последовательным выполнением входящих в ее состав этапов
2. наличием обратных связей между этапами
3. отсутствием временного перекрытия этапов
4. отсутствием (или определенным ограничением) возврата к предыдущим этапам
5. возможность проведение корректировки после каждого этапа

22. В конце каждого витка спирали спиральной модели ЖЦ разработки ПО получаем…
1. готовый программный продукт
2. одну версию программного продукта
3. версию программного продукта с набором тестовых данных
4. черновую модель программного продукта

23.Спиральная модель ЖЦ разработки ПО предполагает:
1. отсутствие временного перекрытия этапов
2. наличие обратной связи
3. возможность сокращения времени разработки ПО

24. На втором этапе каскадной модели ЖЦ разработки ПО (Требования к ПО) осуществляется…
1. определение функциональности программного компонента
2. составление детальной спецификации архитектуры системы
3. составление концептуальной структуры системы
4. написание программного кода

25. Происходит ли интеграция отдельных компонент системы при разработке ПП по экстремальной модели ЖЦ?
1. да
2. Нет

26. Какую модель жизненного цикла разработки ПО целесообразнее использовать, если нет четко определенных требований к будущей системе?

1) каскадную
2) спиральную
3) V – образную
4) итерационную

27. Программное средство - это
1) программа для удовлетворения нужд разработчиков, предназначенная для продажи
2) программа, предназначенная для многократного применения на различных объектах и разработанная любым способом
3) программная реализация решения задачи на компьютере
4) результат разработки какого-либо технического задания

28. Качество ПП - это
1) совокупность свойств этого продукта, которые удовлетворяют определенным потребностям пользователей в соответствии с его назначением;
2) те свойства данного продукта, благодаря которым программный продукт может функционировать в любой программной среде;
3) совокупность свойств программного продукта, которые удовлетворяют требованиям ЕСПД и базовым международным стандартам.

29. Изучаемость ПП включает в себя:
1) удобочитаемость, тестируемость, информативность;
2) внедряемость, понятность, удобочитаемость;
3) документированность, понятность, удобочитаемость

30. Функциональная пригодность программного продукта включает в себя:
1) точность, защищенность, надежность;
2) эффективность и внедряемость;
3) понятность, стабильность, надежность.

31. Понятность ПП заключается в …
1) наличии в составе программы информации необходимой и достаточной для понимания назначения программы, существующих ограничений, входных и выходных данных и результатов обработки;
2) степени, которой пользователь может изучить назначение ПП, результат ее работы и текст этой программы;
3) быстрой модификации с целью приспособления к изменяющимся условиям функционирования.

32. Программа является надежной, если…
1) выдаваемый результат работы имеет допустимые значения отклонений от аналогичных отклонений;
2) она продолжает свою работу при возникновении сбоев;
3) она при всех одинаково вводимых данных обеспечивает полную повторяемость результата.

33. Программа является эффективной, если…
1) она правильно работает при любых допустимых вариантах исходных данных;
2) объем требуемых ресурсов для ее выполнения не превышает допустимой границы;
3) она работает должным образом не только автономно, но и как часть информационной системы.

34. Программа является совместимой, если…

1) она работает должным образом не только автономно, но и как часть информационной системы;

2) ее качества могут быть продемонстрированы на практике;

3) она допускает быструю модификацию с целью приспособления к изменяющимся условиям функционирования.