ВВЕДЕНИЕ

Цель данного курса заключается в изучении принципов и способах тестирования и верификации программных продуктов.

По окончанию курса студенты обретут: знания о принципах структурного и функционального тестирования, по вопросам организации процесса тестирования; способность составлять тестовые варианты для обнаружения ошибок и их документировать, анализировать и исправлять, и количественно оценивать комплексную сложность программных продуктов.

Основные понятия

**Требования к программному обеспечению** — совокупность утверждений относительно атрибутов, свойств или качества программной системы, подлежащей реализации.

**Спецификация требований программного обеспечения** (Software Requirements Specification) — законченное описание поведения системы, которую требуется разработать.

**Дефект** («баг») - ошибка в [программе](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0) или системе, которая выдает неожиданный или неправильный результат относительно указанных в спецификации требований ПО.

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

**Тестирование по стратегии чёрного** ящика – стратегия тестирования, при котором программа рассматривается как чёрный ящик (нет сведений о ее внутренней структуре, нет доступа к коду программы). Целью тестирования ставится выяснение обстоятельств, в которых поведение программы не соответствует [спецификации](http://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B5%D1%86%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F).

**Тестировщик ПО** - специалист, который проводит тестирование программного обеспечения, с целью обеспечения и контроля качества программного продукта или информационной системы.

Полный перечень пунктов, которые должны содержаться в спецификации требований программного обеспечения согласно стандартам

**Введение:**

1. цели;
2. соглашения о терминах;
3. предполагаемая аудитория;
4. масштаб проекта;
5. ссылки на источники.

Общее описание:

1. видение продукта;
2. функциональность продукта;
3. классы и характеристики пользователей;
4. среда функционирования продукта (операционная среда);
5. рамки, ограничения, правила и стандарты;
6. документация для пользователей;
7. допущения и зависимости.

Функциональность системы:

1. функциональный блок X (таких блоков может быть несколько);
2. описание и приоритет;
3. причинно-следственные связи, алгоритмы (движение процессов, workflows);
4. функциональные требования.

Требования к внешним интерфейсам:

1. интерфейсы пользователя (UI);
2. программные интерфейсы;
3. интерфейсы оборудования;
4. интерфейсы связи и коммуникации.

Нефункциональные требования:

1. требования к производительности;
2. требования к сохранности (данных);
3. критерии качества программного обеспечения;
4. требования к безопасности системы.

Прочие требования:

1. Приложение А: Глоссарий;
2. Приложение Б: Модели процессов и предметной области и другие диаграммы;
3. Приложение В: Список ключевых задач.

В практическом задании следует описать видение продукта/программы, его описание и назначение, а также функциональные требования. По возможности могут быть описаны остальные пункты. Спецификация требований должна быть описана в одном документе.

Пример спецификации требований для программы, реализующей сортировку вставкой, представлен в ПРИЛОЖЕНИИ В.

ЛАБОРАТОРНАЯ РАБОТА № 1

**Тема:** Основы тестирования и классификация ошибок. Составление спецификации требований. Интуитивное тестирование

**Цель работы:** *Тестирование программы, классификация ошибок и составление отчётов по ошибкам*.

Задание к лабораторной работе:

1. Написание программы и спецификации требований к ней в соответствии с вариантом задания к лабораторной работе. Каждое функциональное требование должно быть описано.
2. Написание тестовых сценариев по спецификации требований; тестирование функциональных требований разработанной программы: провести тестирование программы в соответствии с шагами тестирования, представленными в теоретической части и описание найденных дефектов/недочётов/ошибок.

Проект может быть написан на любом языке программирования, должен являться прикладной программой (реализованной с помощью интерфейса, input/output файлов, или связанной с базой данных), также это может быть сайтом и т. д.

Тестирование включает:

1. Написание тестовых сценариев для проверки функциональности.
2. Описание найденных ошибок в отчетах.
3. Исправление ошибок в программе и тестирование её вновь.

Содержание отчета:

1. Постановка задачи.
2. Составление спецификации требований.
3. Шаги тестирования.
4. Результаты работы программы.
5. Приведение классификации найденных ошибок. Составление отчёта на одну ошибку в соответствии с приложениями А и Б.
6. Приложение. Листинг программы.

Теоретическая часть

*Пример серии тестов*

Процесс тестирования программного обеспечения можно отчасти назвать интуитивным, но в то же время в основе его лежит вполне систематизированный подход. Хорошо протестировать программу означает нечто гораздо более серьезное, чем просто "погонять" ее несколько минут, чтобы убедиться, что она работает. Эффективное тестирование требует тщательного анализа и строгого системного подхода.

Эта лабораторная работа является своеобразным введением, цель которой на простом примере продемонстрировать подход к тестированию программ, применяемых опытными специалистами этого дела. Для примера была выбрана маленькая незамысловатая программка с несколькими ошибками.

Вместе с программой, которую следует протестировать, необходимо иметь на неё спецификацию (описание).

Пример спецификации программы

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

Значения числа и степени должны быть целыми.

Значения числа, возводимого в степень, должны лежать в диапазоне – [0..999]. Значения степени должны лежать в диапазоне – [1..100].

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

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

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

Первый цикл тестирования

**Шаг 1**. Начало тестирования с простого и наиболее очевидного теста.

Для начала с программой нужно познакомиться и посмотреть, достаточно ли она стабильна, чтобы её можно было тестировать. В программах, предоставленных для первого формального тестирования, часто сразу же происходит сбой. В первом тексте складываются числа 2 и 3. Последовательность действий и результаты приведены в таблице 1.1.

Таблица 1.1 – Порядок действия и результаты сложения чисел 2 и 3

|  |  |  |
| --- | --- | --- |
| **№ действия** | **Ввод** | **Вывод** |
| 1 | Вводите ADDER и  нажимаете клавишу **<***Enter***>** | Экран мигает. Вверху экрана вы появляется знак  вопроса. |
| 2 | Нажимаете 2 | За знаком вопроса появляется цифра 2. |
| 3 | Нажимаете <*Enter*> | В следующей строке появляется знак вопроса. |
| 4 | Нажимаете 3 | За вторым знаком вопроса появляется цифра 3. |
| 5 | Нажимаете <*Enter*> | В третьей строке появляется цифра 5. На несколько строк ниже появляется еще один знак вопроса. |

На рисунке 1.1 видно, как выглядит экран по окончанию теста. Курсор указывает, где будет отображаться следующее вводимое число.

? 2

? 3

5

? \_

Рисунок 1.1 – Вид экрана по окончанию теста

Отчёт о проблемах, выявленных первым тестом:

“*Программа работает – она приняла числа 2 и 3 и вернула 5.*” Но проблемы, всё же, есть. Для их описания составляется отчёт, форма которого представлена в приложении A. Типы ошибок:

1. *Ошибка проектирования.* Нет никаких указаний на то, с какой программой работаете.
2. *Ошибка проектирования.* На экране нет никаких инструкций. Откуда знать, что нужно делать? Что, если вводите недопустимые числа? Отобразить инструкцию на экране не трудно, и она всегда будет перед глазами, в то время как печатная документация может потеряться.
3. *Ошибка проектирования.* Как остановить программу? Эта инструкция тоже должна быть на экране.
4. *Ошибка кодирования.* Сумма (число 5) выведена в стороне от слагаемых.

*Примечание*

Обязательно представляется отдельный “*Отчёт о проблеме*” (приложение A) по каждой ошибке. Описание всех четырех ошибок можно было бы поместить в один отчёт, но лучше этого не делать. Ошибки могут исправляться в разное время и сведения о тех из них, которые остались неисправленными, могут просто потеряться. Если программист захочет их сгруппировать, он сам рассортирует отчеты. Чтобы привлечь внимание к взаимосвязанным проблемам, просто поместите в отчеты соответствующие ссылки.

**Шаг 2**. Составление заметок о том, что ещё должно быть протестировано.

Выполнив первые, и самые очевидные тесты, следует подумать о том, что ещё следует протестировать. Свои соображения нужно записать: одни из записей примут форму заметок, другие же могут представлять собой достаточно строго формализованные описания серий тестов. Такие документированные группы тестов в дальнейшем могут послужить для проверки следующих версий программы. Примером может быть серия тестов, представленная в таблице 1.2, в которой для тестирования программы предлагается девять примеров. Тесты подобранны так, чтобы каждая цифра встречалась в них хотя бы один раз, чтобы было по одной комбинации чисел на каждую из вероятных проблем. А чтобы определить, на каких значениях вероятнее возникнут проблемы, эффективнее всего проверить граничные условия.

Вычисление количества возможных тестов не всегда является простой задачей. В допустимом диапазоне от -99 до 99 всего 199 чисел. Любое из них может стоять на первом месте и любое на втором. Всего получается 1992 = 39601 пар чисел. Это без учёта более сложных действий пользователя. Если же допустить использование клавиши редактирования, количество возможных тестов вырастет многократно. Задача определения количества возможных тестов относится к области математики (комбинаторный анализ). Из огромного количества возможных тестов нужно выбрать только наиболее важные тесты. Как только удаётся выделить группу (класс) однотипных тестов, можно провести несколько из них и проигнорировать остальные. Для отбора проводимых тестов есть важное правило: для выполнения всегда выбирайте из группы те тесты, на которых вероятнее всего ожидается сбой программы. Лучше всего подходят для тестирования примеры, лежащие на границе представленного классом диапазона значений. Именно на граничных значениях программы сбоят чаще всего.

Таблица 1.2 – Серия тестов для программы сложения двух чисел

|  |  |  |
| --- | --- | --- |
| **Входные**  **данные** | **Ожидаемый**  **результат** | **Замечания** |
| 99 + 99 | 198 | Пара наибольших чисел, которые может складывать программа. |
| -99 + -99 | -198 | В документации не сказано, что нельзя складывать отрицательные числа. |
| 99 + -14 | 85 | Большое первое число может повлиять на интерпретацию программой  второго числа. |
| -38 + 99 | 61 | Сложение отрицательного числа с положительным числом. |
| 5 + 99 | 104 | Проверка на то, влияет ли слишком большое второе число на  интерпретацию первого. |
| 9 + 9 | 18 | 9 является наибольшим числом из одной цифры. |
| 0 + 0 | 0 | Программы часто дают сбои при вводе нулей. |
| 0 + 23 | 23 | Программа может особым образом обрабатывать 0, поэтому его нужно  проверить в виде первого слагаемого. |
| -78 + 0 | 78 | Программа может особым образом обрабатывать 0, поэтому его нужно  проверить в виде второго слагаемого. |

*Классом* можно назвать группу значений, которые программа обрабатывает одним и тем же способом. Ограниченными значениями класса являются те входные данные, на которых программа меняет своё поведение. *Работа программиста* – проверять те практические точки, которые можно определить по листингу. *Задача тестировщика* – проанализировать программу с другой точки зрения, чтобы выявить те критические точки, которые программист пропустил. Поэтому классы возможных тестов следует выделять, исходя, прежде всего, из внешнего поведения программы. В результате набор тестов будет отличаться от того, который можно составить по листингу программы (именно в этом суть задачи тестировщика).

Важным моментом здесь является то, что границу значений обязательно нужно протестировать с двух сторон.

**Шаг 3**. Проверка допустимых значений и наблюдение за реакцией программы.

В таблице 2 приведены только допустимые значения входных данных программы. На следующем этапе тестирования можно создать такую же серию тестов для недопустимых значений. Ещё одна серия тестов может быть предназначена для проверки редактирования чисел: вводите значение, затем изменяете его и только после этого нажимаете <*Enter*>.

**Шаг 4**. Немного тестирования в режиме «свободного полёта».

Всегда записывайте, что делаете, и что происходит во время исследовательских тестов.

От формальных тестов следует перейти к неформальным тестам. В таблице 1.3 видно, что при малейшей провокации программа работает неверно (она “зависает”). Это приводит к затрате большего времени на перезапуск компьютера, нежели чем на тестирование.

Таблица 1.3 – Неформальные тесты для программы сложения двух чисел

|  |  |  |
| --- | --- | --- |
| **Тест** | **Особенности теста** | **Замечания** |
| 100 + 100 | Граничное условие – числа больше максимального допустимого значения (99). | Программа приняла 10. Когда ввели второй 0, чтобы получилось 100, программа повела себя так, как будто вы нажали <*Enter* >. Так же было и со вторым числом 100. В результате по окончании теста на экране было следующее:  ? 10  ? 10  20 |
| <*Enter*> +  <*Enter*> | Проверка реакции программы на отсутствие числовых данных. | Когда нажали <*Enter*>, программа напечатала 10 – последнее введенное вами число. То же было и после второго нажатия <*Enter*>, и в качестве суммы программа напечатала 20. |
| 123456 + 0 | Ввод более длинных чисел. | Программа приняла первые две цифры и проигнорировала остальные так же, как было с числом 100. В будущей версии программа будет принимать большие числа. |
| 1,2 + 5 | Числа с десятичной частью. | Реакция на десятичный разделитель такая же  как и на <*Enter*>. |
| A + b | Недопустимые символы. | Когда нажали <*Enter*> после <*А*> программа “зависла”. Для продолжения тестирования приходится перезапускать компьютер. |
| <*Ctrl* + *A*>, <*Ctrl*  + *B*>, <*Ctrl* + *C*>,  <*Ctrl* + *D*>,  <*F1*>, <*Esc*> | Управляющие символы и функциональные клавиши часто являются источниками проблем. | Для всех комбинаций клавиш, кроме <*Ctrl* + *C*>, программа отобразила графические символы, затем, после нажатия <Enter>, “зависла”. Нажатие  <*Ctrl* + *C*> привело к завершению программы и выходу в операционную систему. |

Столкнувшись с очередной проблемой, составляете о ней отчет. О сданных отчётах лучше написать для себя итоговые заметки. На этом тестирование первой версии программы можно считать завершенным.

**Шаг 5**. Подведение итогов о программе и выясненных недостатках.

Эта работа не всегда необходима, но часто оказывается очень полезной. До этого времени все время были сконцентрированы на конкретных деталях – анализировали допустимые данные, продумывали граничные условия и составляли тестовые примеры. В будущем больше времени будет тратиться на выполнение уже подготовленных тестов, чем на придумывание новых. Сейчас же самое время мысленно отступить немного назад и окинуть взглядом программу в целом, увидеть ее недостатки и продумать стратегию будущего тестирования.

Итоги первого цикла тестирования

Начали с простейшего из возможных тестов. Поскольку программа его прошла, разработали серию формальных тестов, чтобы проверить, как она работает с допустимыми данными. Эти тесты будут использованы и дальше. Поскольку часть проверок программа не прошла, на планирование дальнейших серий тестов времени можно не тратить. Вместо этого проведём несколько неформальных экспериментов и выясним, что программа вообще очень нестабильна. Записали несколько замечаний, к которым обратимся при тестировании следующей версии программы.

Если бы программа успешно прошла первую серию тестов, разрабатывается вторая, более обстоятельная. Если бы программа снова показала себя надежной, продолжается ее тестирование, пока не исчерпались бы идеи или отведенное время. Напоследок провели бы несколько беглых тестов, не входивших в ранее разработанные серии, и записали замечания на будущее. Завершив тестирование и всю бумажную работу, стоит потратить еще немного времени, чтобы обдумать результаты. Выполнили самое очевидное, но это еще только начало. Пока нет конкретного плана. Просто делали то, что первым приходило в голову. По ходу работы были определены две линии атаки. Теперь же *нужно выделить время* на обдумывание. Это очень важно, даже если сроки сильно поджимают.

Второй цикл тестирования

Поговорили с программистом, и он сказал, что скорость работы программы исключительно важна, а вот объём кода не имеет никакого значения. Резолюции программиста на отчётах тестировщика:

1. Ошибка проектирования: на экране нет названия программы. Резолюция: не будет исправлена.
2. Ошибка проектирования: на экране нет инструкций. Резолюция: не будет исправлена.

Примечание: замечание верное, но вывод инструкций замедлит работу программы.

1. Ошибка проектирования: невозможность остановить программу.

Резолюция: исправлена; на экране отображается подсказка – “*Для выхода нажмите*

*<Ctrl + C>*”.

1. Ошибка кодирования: сумма (5) выводится в стороне от слагаемых. Резолюция: исправлена.
2. Ошибка кодирования: программа “зависает” на отрицательных числах. Резолюция: исправлена; программа будет складывать и отрицательные числа.
3. Ошибка кодирования: программа интерпретирует третий введенный символ как нажатие

<*Enter*>.

Резолюция: в работе (ещё не исправлена).

1. Ошибка кодирования: сбой при вводе нечисловых данных. Резолюция: не проблема.

Примечание: не выполняйте этой операции.

1. Ошибка кодирования: сбой при вводе управляющих символов. Резолюция: не проблема.

Примечание: не используйте управляющие символы.

1. Ошибка кодирования: сбой при нажатии функциональных клавиш. Резолюция: не проблема.

Примечание: не используйте функциональные клавиши.

**Шаг 1.** Ознакомление с резолюцией программиста.

Так ознакомившись с резолюцией программиста, узнаете, что нужно делать, а чего не нужно. Как видите, хорошо, что не разрабатывали тестов для проверки кода обработки ошибок: этого кода нет и не будет. Более того, хотя программа и будет теперь обрабатывать отрицательные числа, она будет обрабатывать их не все, а только до -9. Числа от -10 до 199 длиной в три символа она по- прежнему не обрабатывает, интерпретируя третий символ как нажатие клавиши <*Enter*>. Обратившись к разработанной ранее серии тестов из таблицы 6, видно, что тесты с числами -99, -78 и -14 запускать не придётся. Вместо -78 и -14 необходимо взять пару однозначных отрицательных чисел.

Очень часто программист просит протестировать остальную часть программы, в то время как он занимается исправлением уже найденных ошибок – и это разумно. Некоторые из запланированных тестов нет смысла проводить до исправления ошибки, с другими же вполне можно поработать. Если ждать, пока можно будет провести “лучшие” из тестов, можно оставить без внимания целые области программы, на которые потом уже не хватит времени. В нашем примере можно протестировать числа между -1 и -9 – так хоть и не полностью, но будет проверено, как работает сложение отрицательных чисел, вместо того чтобы опустить эту проверку вообще.

Из резолюций на отчетах видно, какие тесты больше проводить не нужно, а какие нужно заменить новыми.

**Шаг 2.** Анализ комментариев к ошибкам, которые не будут исправлены.

Возможно, следует провести дополнительное тестирование. В нашей программе хуже всего обстоит дело с обработкой ошибок. Программист не собирается исправлять ситуацию. Как же быть? Чтобы добиться исправления ошибки, нужно продемонстрировать ситуацию, в которой её появление абсолютно недопустимо.

Чтобы придумать самые показательные примеры недопустимого поведения программы, необходимо постараться выявить суть ситуации. В нашем случае программа “зависает”, когда нажимаются определенные клавиши. Так она ведет себя с буквами, управляющими и функциональными клавишами. Фактически программа “зависает” при вводе любого недопустимого (нецифрового) символа. Программист говорит, что таких символов вводить не должны. С точки зрения тестировщика, программа должна вести себя вежливо и не заставлять перезапускать компьютер каждый раз, когда будет сделано что-то не так.

Программа также некорректно ведет себя в ответ на нажатие некоторых клавиш. Программист считает, что это не страшно, поскольку никто и не ждет, что программа примет эти клавиши. А что, если программа “зависнет” в ответ на ввод символов, которые, по мнению пользователя, она должна принять?! Если найти достаточно таких символов, программисту придется написать код для их обработки, а заодно уж он может обработать и остальные символы. Подумаем, какие же клавиши люди могут нажимать при работе с арифметической программой. Здесь нужно применять методику *мозгового штурма****.*** Запишите все клавиши, которые могут прийти человеку. Запишите, почему эти клавиши могут показаться уместными, при этом беспокойтесь, согласится ли программист с вашими предположениями – позднее можно ещё пересмотреть список и отобрать из него самое существенное. Вот вероятный список таких клавиш:

1. цифры;
2. знак минус (‘-’);
3. знак плюс (‘+’);
4. пробел до числа;
5. пробел после числа;
6. операции умножения и деления без остатка (‘\*’, ‘/’);
7. знак доллара (‘$’);
8. знак процента (‘%’);
9. скобки;
10. клавиша <*Backspace*>;
11. клавиша <*Detete*>;
12. клавиша <*lnsert*>;
13. клавиши управления курсором.

**Шаг 3**. Просмотр записей, которые были сделаны в прошлый раз.

Необходимо добавить к предыдущим замечаниям новые и приступить к тестированию. Сначала повторяются старые тесты и убедившись, что программа по-прежнему сможет сложить 2 и 2 и не получить при этом 5, т. е. обязательно тестируется логика программы.

По характеру дела видно, что программа отображает подсказку “*Для выхода нажмите «Ctrl*

*+ C»*” по каждой операции сложения. Вот что отображается при этом на экране:

? 99

? 99

198 – Для выхода нажмите «Ctrl + C»

? -9

? -9

-18 – Для выхода нажмите «Ctrl + C»

? \_

Программист говорит, что программа должна работать быстро. В таком случае всё, на что тратится лишнее время, является ошибкой.

Составляете отчёты по следующим ошибкам:

1. *Ошибка проектирования*. Для вывода на экран подсказки тратится лишнее компьютерное время. Поскольку одной из задач разработки является создание очень быстрой программы, это ошибка. Почему бы просто не написать “*Для выхода нажмите «Ctrl + C»”* внизу экрана, сразу при запуске программы и никогда больше ничего не выводить в этой строчке? Если программист достаточно аккуратен, большая часть тестов, включая и те, которые потребовали наибольшей изобретательности, не выявит ошибок.
2. *Ошибка кодирования*. Проблема такова – в ответ на нажатие клавиш редактирования и других предположительно допустимых клавиш программа “зависает”. Предлагаемое исправление – проверять каждый вводимый символ. Недопустимые символы игнорировать или выводить сообщение об ошибке.

Хороший тестировщик – не тот, кто выявит больше всего ошибок, и не тот, кто заставит смутиться даже самого первоклассного программиста. Лучшим является тот, кто добьётся исправления наибольшего количества ошибок.

Варианты задания к лабораторной работе

1. Нахождение наибольшего общего делителя при условии, что начальные значения

𝑥1 и 𝑥2 положительны.

1. Вычисление факториала неотрицательного целого числа.
2. Вычисление чисел Фибоначчи.
3. Поиск заданного символа в строке.
4. Выполнение конкатенации (сцепления) строк.
5. Определить частное *q* и остаток *r* от деления *x* на *y*.
6. Линейный поиск в упорядоченном по возрастанию элементов массиве *A*[1..*n*]. Определить порядковый номер *m* некоторого значения *х*.
7. Поиск значения *х* в двумерном массиве.
8. Написать программу, которая по данному фиксированному массиву *B*[0..*n* – 1], где

*n* > 0, записывает в *d* число нечётных значений в *B*[0..*n* – 1].

1. Нахождение наибольшего целого числа по данному фиксированному целому числу m > 0, которое не больше *n*, в массиве целых чисел.
2. Суммирование элементов массива *B*[0..*n* – 1] , где *n* > 0.
3. Вставка пробелов (выравнивание строки текста по правому краю).
4. Выравнивание текста по центру.
5. Определение максимальной возрастающей подпоследовательности значений, идущих не обязательно подряд.
6. Перестановка двух неперекрывающихся сегментов массива равного размера.
7. Подсчёт числа листьев в дереве.
8. Удаление лишних пробелов в тексте.
9. Суммирование комплексных чисел.
10. Умножение комплексных чисел.
11. Деление комплексных чисел.
12. Выполнение арифметических операций над комплексными числами.
13. Вводится строка слов. Вывести слова в обратном порядке.
14. В одномерном массиве найти максимальный из отрицательных элементов и поменять его местами с последним элементом массива.
15. В зависимости от выбора пользователя программа производит либо шифрование, либо дешифровку вводимой строки методом кода (шифра) Цезаря - каждая буква заменяется на букву, отстоящую в алфавите от исходной на определенное значение.
16. [Шифр Виженера.](https://ru.wikipedia.org/wiki/%D0%A8%D0%B8%D1%84%D1%80_%D0%92%D0%B8%D0%B6%D0%B5%D0%BD%D0%B5%D1%80%D0%B0)
17. Генератор случайных чисел.
18. Составьте программу печати прямоугольного треугольника из звездочек

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

используя цикл for. Введите переменную, значением которой является размер катета треугольника.

1. Напишите программу с циклами, которая рисует треугольник:

\*

\*\*\*

\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*

1. Напишите программу, подсчитывающую число символов, поступающих со стандартного ввода.
2. Составьте программу перекодировки вводимых символов со стандартного ввода по следующему правилу:

a -> b b -> c c -> d

...

z -> a

другой символ -> \*

Коды строчных латинских букв расположены подряд по возрастанию.

1. Напишите программу, печатающую квадраты и кубы целых чисел.
2. Напишите программу, печатающую сумму квадратов первых n целых чисел.
3. Напишите программу, которая переводит секунды в дни, часы, минуты и секунды.
4. Напишите программу, переводящую скорость из километров в час в метры в секундах.
5. Напишите программу, шифрующую текст файла путем замены значения символа (например, значение символа **C** заменяется на **C+1** или на **~C**).

Контрольные вопросы

1. Что вы понимаете под тестированием и верификацией программных продуктов?
2. Какие типы ошибок существуют?
3. Что такое классы тестов и для чего они предназначены?
4. На какие шаги можно разбить первый цикл тестирования любой программы?
5. Для чего нужен второй цикл тестирования?

ПРИЛОЖЕНИЯ

**Приложение A**

Форма документа “Отчёт о проблеме”

НАЗВАНИЕ КОМПАНИИ PortnovInc ОТЧЁТ О ПРОБЛЕМЕ *№*

ПРОГРАММА сумматор комплексных чисел

ВЫПУСК 1

ВЕРСИЯ 2

ТИП ОТЧЕТА (1 – 6) 3\_\_ СТЕПЕНЬ ВАЖНОСТИ (1 – 3) 3 ПРИЛОЖЕНИЯ (Д / Н ) Н

1 - Ошибка кодирования 1 - Блокирующая Если да, какие: 2 - Ошибка проектирования 2 - Критическая

1. - Предложение 3 - Значительная
2. - Расхождение с документацией 4 - Незначительная 5 - Взаимодействие с аппаратурой 5 - Тривиальная

6 - Вопрос

ПРОБЛЕМА нет указания прерывания программы при нажатии CTRL+C

МОЖЕТЕ ЛИ ВЫ ВОСПРОИЗВЕСТИ ПРОБЛЕМНУЮ СИТУАЦИЮ? (Д / Н) Д

ПОДРОБНОЕ ОПИСАНИЕ ПРОБЛЕМЫ И КАК ЕЕ ВОСПРОИЗВЕСТИ запустить программу, нажать CTRL + C

ПРЕДЛАГАЕМОЕ ИСПРАВЛЕНИЕ (НЕОБЯЗАТЕЛЬНО) добавить обработку нажатия CTRL + C

ОТЧЕТ ПРЕДСТАВЛЕН СОТРУДНИКОМ Портнов Максим ДАТА 06/03/ 2024

Следующие графы предназначены только для разработчиков

ФУНКЦИОНАЛЬНАЯ ОБЛАСТЬ ОТВЕТСТВЕННОЕ ЛИЦО Портнов Максим,

КОММЕНТАРИИ Будет сделано

СОСТОЯНИЕ (1 – 2) 1 ПРИОРИТЕТ (1 – 5) 4

1. – Открыто 1 - Высокий 4 - Ниже среднего
2. – Закрыто 2 - Выше среднего 5 - Низкий 3 - Средний

РЕЗОЛЮЦИЯ (1 – 9) 1 ИСПРАВЛЕННАЯ ВЕРСИЯ

1. - Рассматривается 4 - Отложено 7 - Отозвано составителем
2. - Исправлено 5 - Соответствует проекту 8 **-** Нужна дополнительная информация 3 - Не воспроизводится 6 - Не может быть исправлено 9 - Не согласен с предложением

РАССМОТРЕНО Портнов Максим Александрович ДАТА 06/03/ 2024

ПРОКОНТРОЛИРОВАНО Портнов Максим Александрович ДАТА 06/03/ 2024

СЧИТАТЬ ОТЛОЖЕННЫМ (Д / Н) Н

Приложение Б

**Информация по оформлению *“Отчёта о проблеме”***

**Отчет об ошибке** — это технический документ, описывающий ошибку/дефект, найденную в процессе тестирования ПО**.** В связи с этим необходимо отметить, что язык описания проблемы должен быть техническим. Должна использоваться правильная терминология при использовании названий элементов пользовательского интерфейса, действий пользователя и полученных результатах.

Требования к обязательным полям отчёта

Обязательными полями являются: краткое описание (*Bug Summary*), степень важности (*Severity*), приоритет (Priority), шаги к воспроизведению проблемы (*Steps to reproduce*) и т. д. Также необходимо прикрепить файлы (скриншоты, тестовые файлы и др.), которые показывают дефект наглядно, а также использованные для тестирования файлы.

Тип отчёта

В графе “Тип отчёта” указывается тип обнаруженной проблемы.

* 1. **Ошибка кодирования**. Программа ведёт себя не так, как должна, по мнению тестировщика. Например, если программа утверждает, что 2+2=3, то это явная ошибка кодирования. Программист же в ответ на отчёт о такой ошибке вполне может написать. Соответствует проекту.
  2. **Ошибка проектирования**. Программа соответствует проектной документации, в определённом вопросе тестировщик с этой документацией не согласен. Так особенно часто случается с элементами пользовательского интерфейса. На отчёте данного типа программист не может написать **Соответствует проекту**, если он считает, что проект верен, тогда он пишет **Не согласен с предложением**.
  3. **Предложение**. Отчёт такого типа не означает, что в программе что-то не так. В нём описывается идея, реализация которой, по мнению тестировщика, может улучшить программу.
  4. **Расхождение с документацией.** Программа ведёт себя не так, как описано в руководстве или интерактивной справки. В этом случае в отчёте следует указать, в каком именно документе и на какой странице найдено несоответствие. При этом в отчёте вовсе не утверждается, что ошибка именно в документации, а не в самой программе. Отчёты о расхождении с документацией обязательно должны совместно рассматриваться программистом и автором документации. О функциях программы, которые вообще нигде не описаны, также следует составлять отчёты данного типа.
  5. **Взаимодействие с аппаратурой**. Проблемы этого рода связаны с неудачным взаимодействием программы и определённого вида аппаратного обеспечения. Если причина неудачи заключается в неисправности устройства, отчёт о ней составлять не нужно. Однако если

программа не может работать ни с одной платой или устройством конкретного типа — это уже проблема, которую следует документировать.

* 1. **Вопрос.** Программа делает что-то, чего тестировщик не ожидает или не понимает. Отчёт – вопрос стоит составить при любых сомнениях. Если они окажутся основанными на действительной ошибке, программист её исправит. Если же программист откажется исправить ошибку или его объяснения не покажется вам достаточно разумным, можно будет составить отчёт об ошибке проектирования.

**Степень важности** (**Severity**) — это атрибут, характеризующий влияние дефекта на работоспособность приложения.

В этой графе тестировщик указывает, насколько, по его мнению, серьёзна выявленная проблема.

К сожалению, для определения степени важности проблемы не существует строго критерия. Бейзер, например, предлагает шкалу от 1 (незначительная ошибка, например грамматическая) до 10 (фатальная, вызывающая сбой в других системах, войны, убийства и т. д.). Однако при этом Бейзер не считает значительными недостатки, которые вызывают у пользователя раздражение или заставляют его терять время.

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

Градация степени важности ошибки (Severity)

1. **Блокирующая (Blocker)** Блокирующая ошибка, приводящая приложение в нерабочее состояние, в результате которого дальнейшая работа с тестируемой системой или ее ключевыми функциями становится невозможна. Решение проблемы необходимо для дальнейшего функционирования системы.
2. **Критическая (Critical)** Критическая ошибка, неправильно работающая ключевая бизнес- логика, дыра в системе безопасности, проблема, приведшая к временному падению сервера или приводящая в нерабочее состояние некоторую часть системы, без возможности решения проблемы, используя другие входные точки. Решение проблемы необходимо для дальнейшей работы с ключевыми функциями тестируемой системой.
3. **Значительная (Major) Значительная** ошибка, часть основной бизнес логики работает некорректно. Ошибка не критична или есть возможность для работы с тестируемой функцией, используя другие входные точки.
4. **Незначительная (Minor) Незначительная** ошибка, не нарушающая бизнес логику тестируемой части приложения, очевидная проблема пользовательского интерфейса.
5. **Тривиальная (Trivial) Тривиальная** ошибка, не касающаяся бизнес-логики приложения, плохо воспроизводимая проблема, малозаметная по средствам пользовательского интерфейса, проблема сторонних библиотек или сервисов, проблема, не оказывающая никакого влияния на общее качество продукта.

Приложения

К отчету о найденной ошибке можно приложить дискету с тестовыми данными или программу, эмулирующую действия пользователя, при которых проявляется данная ошибка. Можно приложить распечатки, копии экрана или собственные дополнительные пояснения. Проще говоря, все, что поможет программисту разобраться в ситуации и понять вашу точку зрения, следует передать ему вместе с отчетом и перечислить в графе “**Приложения”** чтобы эти материалы случайно не затерялись.

Проблема

В этой графе суть проблемы формулируется очень коротко – в одной-двух строчках. Но при этом описание должно быть и достаточно информативным, чтобы прочитавший его сотрудник смог сразу составить себе четкое представление о проблеме. Именно по нему он будет искать нужный отчет, если захочет возвратиться к нему повторно. Кроме того, следует иметь в виду, что в сводных перечнях ошибок, как правило, будут присутствовать всего несколько полей: Номер отчета, Степень важности, возможно Тип отчета и Проблема.

Если по этому короткому описанию проблема покажется менее серьезной, чем есть на самом деле, существует риск, что руководитель проигнорирует отчет. Но и слишком сгущать краски тоже нельзя, иначе вы прослывете паникёром.

В графе “**Проблема**” не следует рассказывать, как воспроизвести ошибку. Примером хорошего описания может быть такая строчка: “Сбой программы при попытке сохранения файла под не допустимым именем”.

Можете ли вы воспроизвести проблемную ситуацию

Ответом может быть: **Да, Нет** или **Не всегда**. Если с повторением ситуации возникли сложности, лучше отложить составление отчета до тех пор, пока дело не прояснится: либо вы убедитесь, что не знаете, как ее воспроизвести (и напишете **Нет**), либо поймете, что она носит нерегулярный характер (и напишите **Не всегда**). В последнем случае описать способ воспроизведения ситуации нужно особенно тщательно, указав, при каких обстоятельствах ошибка проявляется, а при каких – нет. Следует помнить, что, если написать в отчете **Да** или **Не всегда**, программист может попросить продемонстрировать описанную ситуацию, и если вы не сможете этого сделать, то зря потратите его время и потеряете доверие. С другой стороны, отчет о проблеме, которую невозможно воспроизвести, программист с большой вероятностью просто отложит, пока не появятся дополнительные отчеты.

Подробное описание проблемы и как ее воспроизвести

Прежде всего, следует подробно написать, в чем состоит проблема, и если это не очевидно, то почему вы считаете, что что-то не в порядке. Опишите все шаги и симптомы, все подробности, включая сообщения об ошибке. В этом разделе отчета лучше предоставить программисту избыточную информацию, чем написать слишком мало.

Составляя описание, тестировщик часто обнаруживает, что не знает точно, при каких условиях проявляется ошибка. Лучше увидеть это сразу и протестировать программу еще немного, чем давать программисту повод усомниться в вашей аккуратности.

Если же воспроизвести ошибку не удается даже после многих попыток, но при этом вы абсолютно уверены, что видели ее, составьте о ней максимально подробный отчет. Хороший программист сможет ее найти по вашему описанию, проанализировав программный код. Опишите все сообщения об ошибках, расскажите, что пытались делать. Но никогда не игнорируйте проблему только потому, что она не производится.

Предлагаемое исправление

Эта графа отчета не является обязательной. Если решение проблемы очевидно или, наоборот, у вас нет конкретного предложения, оставьте ее пустой.

Но не стоит пренебрегать ею, если вы знаете, как исправить найденный недостаток программы. Особенно это касается пользовательского интерфейса: программист может не исправить его просто потому, что не сможет быстро придумать, как это сделать. В то же время неудачный текст на экране или неудобное расположение элементов формы будут исправлены очень быстро, если предложить программисту готовый вариант решения.

Отчёт представлен сотрудником

Обязательно укажите здесь свою фамилию. Если у программиста возникнут вопросы, он должен знать, к кому обратиться. А анонимные отчеты часто вообще игнорируются.

Дата

В этой графе следует указать дату обнаружения проблемы. Это не дата написания отчета и не дата ввода его в компьютер. Поскольку программисты не всегда меняют номер версии программы после внесения в нее очередных изменений (иногда просто забывая это делать), указанная в отчете дата поможет идентифицировать версию программы, к которой он относится.

Функциональная область

В этой графе указывается, к какой категории относится выявленная проблема. Их полный перечень должен быть единым, чтобы во всех отчетах названия функциональных областей были одинаковыми. Кроме того, их не должно быть слишком много, и они должны быть очень чётко определены. Десяток функциональных областей часто является оптимальным количеством.

Поручено

В этой графе должно быть указано, кто из сотрудников отвечает за решение описанной проблемы. Как правило, это решает руководитель проекта, который передаёт отчет конкретному программисту. Тестировщик или даже руководитель группы тестирования не должен решать, кто конкретно должен внести исправления.

Комментарии

Если отслеживание ошибок и их исправления не автоматизировано, а ведётся на бумаге, эта графа используется программистом. Он может коротко записать, почему отчет отложен или как решена проблема.

В многопользовательских системах отслеживания ошибок данное поле используется гораздо более эффективно. Прежде всего, оно может быть любой длины, и каждый, кто имеет доступ к отчету, может внести собственный комментарий. Для исправления сложных ошибок или решения спорных проблем иногда может потребоваться целая дискуссия с несколькими участниками. Свое мнение может высказать программист, один или несколько тестировщиков, члены группы технической поддержки, авторы документации, менеджер по маркетингу, руководитель проекта и др. Это быстрый и очень эффективный способ обмена информацией и мнениями, гораздо более эффективный, чем электронная почта. Некоторые опытные сотрудники групп тестирования считают это поле базы данных одним из самых важных.

Состояние

В поле **Состояние** только что написанного отчета записывается **Открыто.** Поле исправления ошибки или принятия решения, не требующего дальнейшей работы с отчетом, значение этого поля изменяется на **Закрыто**.

В некоторых компаниях используются три варианта состояния вопроса: **Открыто, Закрыто и Решено**. Программисты ищут в базе данных отчеты по состоянию **Открыто**, а тестировщики по состоянию **Решено**. В нашей системе программисты ищут отчеты по резолюции **Рассматривается**, а тестировщики - по состояния **Открыто** с любыми резолюциями, кроме **Рассматривается**. Обе эти системы логически эквивалентны, но у каждой из них есть убежденные сторонники.

Градация Приоритета дефекта (Priority)

1. **Высокий (High) Ошибка** должна быть исправлена как можно быстрее, т.к. ее наличие является критической для проекта.
2. **Средний (Medium) Ошибка** должна быть исправлена, ее наличие не является критичной, но требует обязательного решения.
3. **Низкий (Low) Ошибка** должна быть исправлена, ее наличие не является критичной, и не требует срочного решения.

Порядок исправления ошибок по их приоритетам:

**High -> Medium -> Low**

Приложение В

Пример спецификации требований

Введение

а) Цели

Реализованный программный продукт Program Sort представляет собой приложение для сортировки массива чисел. Программный продукт должен включать пользовательский интерфейс с полями ввода и вывода массива, кнопкой сортировки. Необходимо наличие текстовых файлов, содержащих исходный и отсортированный массивы.

1. Соглашения о терминах ПО – программный продукт

Program Sort – программа для сортировки массива чисел по алгоритму «Вставка»

Program Sort Interface – интерфейс для ввода исходного массива и вывода отсортированного массива.

1. Ссылки на источники:

«Ссылки на алгоритм сортировки вставкой»

Общее описание

а) Видение продукта:

Данное ПО является прикладной программой для преподавателей и студентов для сортировки массива чисел по методу сортировки вставкой. ПО состоит из пользовательского интерфейса, а также создаваемого текстового файла для хранения данных.

1. Функциональность продукта:

Данное ПО реализует сортировку вещественных чисел в диапазоне от -32678 до 32677. Время обработки массива зависит от размера исходного массива, а также от того на сколько он не отсортирован.

1. Классы и характеристики пользователей

Преподаватели ВУЗа и студенты специальности информационных технологий дневной и заочной форм обучения.

***Функциональность системы*** *(функциональные требования)*

Требование 1

Исполняющий файл имеет имя Program\_Sort и формат .exe

Требование 2

Тестовый файл имеет имя Sort\_file и формат .txt

Требование 3

Тестовой файл создается после первого успешного запуска программы.

Требование 4

Тестовой файл обновляется при каждом успешном запуске программы.

Требование 5

Текстовой файл программы размещается в той директории, где сохранен исполняющий файл программы.

Требование 6

Program Sort Interface содержит поле для ввода массива, кнопку «Sort», поле для вывода отсортированного массива.

Требование 7

Поле для ввода массива вмещает 50 символов.

Требование 8

Символами являются числа массива, а также запятые, разделяющие их.

Требование 9

Сообщение об ошибке появляется в следующих случаях:

* в поле для ввода введено более 50 символов и нажата кнопка «Sort»;
* в поле для ввода помимо требуемых символов введены буквы и другие знаки и нажата кнопка «Sort»;
* в поле для ввода введено число менее – 32678, и нажата кнопка «Sort»;
* в поле для ввода введено число более 32677 и нажата кнопка «Sort»;
* в поле для ввода не введено ни одного символа и нажата кнопка «Sort»;
* в поле для ввода введен пробел между двумя числами.

*Пометка:* при синхронизации данных случаев сообщение об ошибке также должно появляться.

Требование 10

После нажатия кнопки «Sort» в поле для вывода должен появиться отсортированный массив в формате «число 1», «число 2» и так далее.

Требование 11

При успешном завершении программы текстовой файл содержит:

* в первой строке неотсортированный массив;
* во второй строке – отсортированный.

Требование 12

Массив в поле для ввода в Program Sort Interface равен массиву в первой строке текстового файла после успешного завершения работы программы.

Требование 13

Массив в поле для вывода в Program Sort Interface равен массиву во второй строке текстового файла после успешного завершения работы программы.

Требование 14

После закрытия сообщения об ошибке поле для ввода массива очищается автоматически.

Написание тестовых сценариев (test-case)

**Тестовые сценарии** (тесты, test-cases)– последовательность действий тестера в ПО для проверки его качества или качества его функциональной части. Тестовые сценарии удобнее записывать в таблице. В ней должны быть указаны в колонках следующие пункты:

1. Название теста.
2. Описание теста.
3. Шаг (номер шага).
4. Описание шага (для каждого шага).
5. Ожидаемый результат (для каждого шага).

Также при необходимости можно указывать сложность теста, его приоритет, комментарии, а также предусловия и постусловия.

Приложение Г

Пример составления отчета об ошибке:

**Описание бага:** при закрытии сообщения об ошибке поле для ввода массива не очищается автоматически.

Шаги к воспроизведению:

Номер шага

Описание шага

Ожидаемый результат

Шаг 1

Запустить Program Sort

Программа запущена без ошибок

Шаг 2

Шаг 3

Шаг 4

Шаг 5

В поле для ввода ввести: 51 символ

Нажать кнопку "Sort"

Закрыть сообщение об ошибке

Проверить что поле для ввода очищено автоматически

51 символ должны быть введен

Появилось сообщение об ошибке

Сообщение об ошибке закрыто. Поле для ввода

массива очистилось автоматически.

Поле для ввода очищено

**Результат шага 5**: после закрытия сообщения об ошибке поле для ввода осталось с введенным до этого массовом (51 символ).

Серьезность: Незначительная (Minor) Приоритет: Низкий (Low)

**Файлы:** «скриншот до закрытия сообщения об ошибке»

«скриншот после закрытия сообщения об ошибке»