

Методические указания
по выполнению практических работ
МДК 01.02. ПОДДЕРЖКА И ТЕСТИРОВАНИЕ ПРОГРАММНЫХ
МОДУЛЕЙ

для специальности 09.02.07 Информационные системы и программирование

СОДЕРЖАНИЕ

| | |
|------------------------------------------------------------------------------------------------------------------------------------------|-----|
| Пояснительная записка | 4 |
| Порядок выполнения практической и лабораторной работы..... | 5 |
| Рекомендации по оформлению практической и лабораторной работы..... | 5 |
| Критерии оценки практической и лабораторной работы | 5 |
| Перечень практических работ | 6 |
| Практическая работа № 2.1. Выявление ошибок и причин их появления..... | 7 |
| Практическая работа № 2.2. Тестирование «белым ящиком»..... | 8 |
| Практическая работа № 2.3. Тестирование «белым ящиком»..... | 8 |
| Практическая работа № 2.4. Тестирование «черным ящиком»..... | 10 |
| Практическая работа № 2.5. Тестирование «черным ящиком»..... | 10 |
| Практическая работа № 2.6. Модульное тестирование | 12 |
| Практическая работа № 2.7. Интеграционное тестирование..... | 18 |
| Практическая работа № 2.8. Разработка алгоритма поставленной задачи и реализация его средствами автоматизированного проектирования | 18 |
| Практическая работа № 2.9. Разработка алгоритма поставленной задачи и реализация его средствами автоматизированного проектирования | 18 |
| Практическая работа № 2.10. Использование инструментальных средств на этапе отладки программного модуля | 25 |
| Практическая работа № 2.11. Использование инструментальных средств на этапе отладки программного модуля | 25 |
| Практическая работа № 2.12. Тестирование на этапе сопровождения программного продукта | 30 |
| Практическая работа № 2.13. Место верификации среди процессов разработки программного обеспечения | 34 |
| Практическая работа № 2.14. Тестовые примеры. Классы эквивалентности. Ручное тестирование в MVSTE..... | 42 |
| Практическая работа № 2.15. Тестовое окружение | 51 |
| Практическая работа № 2.16. Модульное тестирование. Тестирование классов | 58 |
| Практическая работа № 2.17. Автоматизация модульного тестирования | 64 |
| Практическая работа № 2.18. Формальные инспекции | 70 |
| Практическая работа № 2.19. Покрытие программного кода | 73 |
| Практическая работа № 2.20. Повторяемость тестирования, зависимости тестовых примеров | 84 |
| Практическая работа № 2.21. Интеграционное тестирование в MVSTE..... | 96 |
| Практическая работа № 2.22. Тестирование в Microsoft Solutions Framework | 103 |
| Практическая работа № 2.23. Оформление документации на программные средства с использованием инструментальных средств | 113 |
| Практическая работа № 2.24. Оформление документации на программные средства с использованием инструментальных средств | 113 |
| Список литературы | 116 |

Пояснительная записка

Методические указания по выполнению лабораторных и практических работ по МДК 01.02. Поддержка и тестирование программных модулей разработаны в соответствии с рабочей программой профессионального модуля и предназначены для приобретения необходимых практических навыков и закрепления теоретических знаний, полученных обучающимися при изучении профессионального модуля, обобщения и систематизации знаний перед экзаменом.

Методические указания предназначены для обучающихся специальности 09.02.07 Информационные системы и программирование.

МДК 01.02. Поддержка и тестирование программных модулей относится к профессиональному циклу, изучается на 4 курсе и при его изучении отводится значительное место выполнению практических работ.

Освоение содержания МДК 01.02. Поддержка и тестирование программных модулей во время выполнения практических работ обеспечивает достижение обучающимися следующих **результатов:**

| Код | Наименование общих компетенций |
|---------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ОК 1. | Выбирать способы решения задач профессиональной деятельности, применительно к различным контекстам |
| ОК 2. | Осуществлять поиск, анализ и интерпретацию информации, необходимой для выполнения задач профессиональной деятельности. |
| ОК 3 | Планировать и реализовывать собственное профессиональное и личностное развитие. |
| ОК 4 | Планировать и реализовывать собственное профессиональное и личностное развитие. |
| ОК 5 | Планировать и реализовывать собственное профессиональное и личностное развитие. |
| ОК 6 | Проявлять гражданско-патриотическую позицию, демонстрировать осознанное поведение на основе традиционных общечеловеческих ценностей, применять стандарты антикоррупционного поведения |
| ОК 7 | Содействовать сохранению окружающей среды, ресурсосбережению, эффективно действовать в чрезвычайных ситуациях. |
| ОК 8 | Использовать средства физической культуры для сохранения и укрепления здоровья в процессе профессиональной деятельности и поддержания необходимого уровня физической подготовленности |
| ОК 9 | Использовать информационные технологии в профессиональной деятельности. |
| ОК 10 | Пользоваться профессиональной документацией на государственном и иностранном языках |
| ОК 11 | Использовать знания по финансовой грамотности, планировать предпринимательскую деятельность в профессиональной сфере |
| ПК 1.1. | Формировать алгоритмы разработки программных модулей в соответствии с техническим заданием |
| ПК 1.2. | Разрабатывать программные модули в соответствии с техническим заданием |
| ПК 1.3. | Выполнять отладку программных модулей с использованием специализированных программных средств |
| ПК 1.4. | Выполнять тестирование программных модулей |
| ПК 1.5. | Осуществлять рефакторинг и оптимизацию программного кода |

В результате освоения профессионального модуля обучающийся должен:

| | |
|-------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Иметь практический опыт | В разработке кода программного продукта на основе готовой спецификации на уровне модуля; использовании инструментальных средств на этапе отладки программного продукта; проведении тестирования программного модуля по определенному сценарию; |
|-------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

| | |
|-------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| | использовании инструментальных средств на этапе отладки программного продукта; разработке мобильных приложений |
| Уметь | осуществлять разработку кода программного модуля на языках низкого и высокого уровней; создавать программу по разработанному алгоритму как отдельный модуль; выполнять отладку и тестирование программы на уровне модуля; осуществлять разработку кода программного модуля на современных языках программирования; уметь выполнять оптимизацию и рефакторинг программного кода; оформлять документацию на программные средства |
| Знать | основные этапы разработки программного обеспечения; основные принципы технологии структурного и объектно-ориентированного программирования; способы оптимизации и приемы рефакторинга; основные принципы отладки и тестирования программных продуктов |

В соответствии с рабочей программой по ПМ 01. Разработка модулей программного обеспечения для компьютерных систем, практические работы по МДК 01.02. Поддержка и тестирование программных модулей проводятся в седьмом семестре. Целесообразность данной группировки обусловлена необходимостью обобщения и систематизации знаний перед экзаменом.

Рабочая программа профессионального модуля предусматривает проведение практических работ МДК 01.02. Поддержка и тестирование программных модулей в объеме 48 часов.

Порядок выполнения практической и лабораторной работы

- записать название работы, ее цель в тетрадь;
- выполнить основные задания в соответствии с ходом работы;
- выполнить индивидуальные задания.

Рекомендации по оформлению практической и лабораторной работы

Задания выполняются обучающимися по шагам. Необходимо строго придерживаться порядка действий, описанного в практической работе

Результаты выполнения практических работ необходимо сохранять в своей папке на компьютере или USB – накопителе.

В случае пропуска занятий обучающийся осваивает материал самостоятельно в свободное от занятий время и сдает практическую работу с пояснениями о выполнении.

Критерии оценки практической и лабораторной работы

- наличие Цель выполняемой работы, выполнение более половины основных заданий (удовлетворительно);
- наличие Цель выполняемой работы, выполнение всех основных и более половины дополнительных заданий (хорошо);
- наличие Цель выполняемой работы, выполнение всех основных и индивидуальных заданий (отлично).

Перечень практических работ

| № | Наименование разделов и тем профессионального модуля (ПМ) | Наименование лабораторных работ и практических занятий | Объем часов |
|-------|----------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------|-------------|
| | Раздел 2. Поддержка и тестирование программных модулей | | 2 |
| | МДК 01.02. Поддержка и тестирование программных модулей | | 2 |
| 1. | Тема 2.1 . Отладка и тестирование программного обеспечения | Практическая работа № 2.1. Выявление ошибок и причин их появления | 2 |
| 2. | | Практическая работа № 2.2. Тестирование «белым ящиком» | 2 |
| 3. | | Практическая работа № 2.3. Тестирование «белым ящиком» | 2 |
| 4. | | Практическая работа № 2.4. Тестирование «черным ящиком» | 2 |
| 5. | | Практическая работа № 2.5. Тестирование «черным ящиком» | 2 |
| 6. | | Практическая работа № 2.6. Модульное тестирование | 2 |
| 7. | | Практическая работа № 2.7. Интеграционное тестирование | 2 |
| 8. | Тема 2.2. Основные принципы отладки и тестирования программных продуктов номер и наименование темы | Практическая работа № 2.8. Разработка алгоритма поставленной задачи и реализация его средствами автоматизированного проектирования | 2 |
| 9. | | Практическая работа № 2.9. Разработка алгоритма поставленной задачи и реализация его средствами автоматизированного проектирования | 2 |
| 10. | | Практическая работа № 2.10. Использование инструментальных средств на этапе отладки программного модуля | 2 |
| 11. | | Практическая работа № 2.11. Использование инструментальных средств на этапе отладки программного модуля | 2 |
| 12. | Тема 2.3 Виды тестирования программных продуктов | Практическая работа № 2.12. Тестирование на этапе сопровождения программного продукта | 2 |
| 13. | | Практическая работа № 2.13. Введение Место верификации среди процессов разработки программного обеспечения | 2 |
| 14. | | Практическая работа № 2.14. Тестовые примеры. Классы эквивалентности. Ручное тестирование в MVSTE | 2 |
| 15. | | Практическая работа № 2.15. Тестовое окружение | 2 |
| 16. | | Практическая работа № 2.16. Модульное тестирование. Тестирование классов | 2 |
| 17. | | Практическая работа № 2.17. Автоматизация модульного тестирования | 2 |
| 18. | | Практическая работа № 2.18. Формальные инспекции | 2 |
| 19. | | Практическая работа № 2.19. Покрытие программного кода | 2 |
| 20. | | Практическая работа № 2.20. Повторяемость тестирования, зависимости тестовых примеров | 2 |
| 21. | | Практическая работа № 2.21. Интеграционное тестирование в MVSTE | 2 |
| 22. | | Практическая работа № 2.22. Тестирование в Microsoft Solutions Framework | 2 |
| 23. | Тема 2.4. Документирование | Практическая работа № 2.23. Оформление документации на программные средства с использованием инструментальных средств | 2 |
| 24. | | Практическая работа № 2.24. Оформление документации на программные средства с использованием инструментальных средств | 2 |
| Итого | | | 48 |

Практическая работа № 2.1. Выявление ошибок и причин их появления

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 № 2.2. Тестирование «белым ящиком»

Практическая работа № 2.3. Тестирование «белым ящиком»

Цель работы: изучить метод тестирования «Белым ящиком»

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 каждым годом. И у пользователей появился выбор что покупать и чем пользоваться.

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

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).

Метод белого ящика (white box testing, open box testing, clear box testing, glass box testing) – у тестирующего есть доступ к внутренней структуре и коду приложения, а также есть достаточно знаний для понимания увиденного.

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

Техники, основанные на структуре, или методе белого ящика

- тестирование операторов;
- тестирование альтернатив.

Альтернатива (decision): Точка программы, в которой управление имеет два или более альтернативных путей. Узел с двумя или более связями для разделения ветвей.

Тестирование условий альтернатив (decision condition testing): Разработка тестов методом белого ящика, при котором тестовые сценарии проектируются для исходов условий и результатов альтернатив.

Покрытие (coverage): Уровень, выражаемый в процентах, на который определенный элемент покрытия был проверен набором тестов.

Покрытие альтернатив (decision coverage): Процент результатов альтернативы, который был проверен набором тестов. Стопроцентное покрытие решений подразумевает стопроцентное покрытие ветвей и стопроцентное покрытие операторов.

Покрытие кода (code coverage): Метод анализа, определяющий, какие части программного обеспечения были проверены (покрыты) набором тестов, а какие нет, например, покрытие операторов, покрытие альтернатив или покрытие условий. Еще выделяют серый ящик.

Задание 1. Разработать программу на Python.

Даны длины сторон треугольника, определить вид треугольника и его площадь. Выполнить контроль вводимых чисел.

1. Разносторонний треугольник
2. Равнобедренный треугольник
3. Равносторонний треугольник

Ограничения:

- три числа не могут быть определены как стороны треугольника;
- если хотя бы одно из них меньше или равно 0;
- сумма двух из них меньше третьего.

Задание 2. Подготовить набор тестовых вариантов для обнаружения ошибок в программе.

Результат оформить в следующем виде:

Таблица 1

| А | В | С | Ожидаемый результат | Объект проверки |
|---|---|---|---------------------|-----------------|
|---|---|---|---------------------|-----------------|

| Значение | Значение | Значение | Что должно получиться | Значения вводимых данных, либо ожидаемый результат |
|----------|----------|----------|-----------------------|----------------------------------------------------|
| ... | ... | ... | ... | ... |

Задание 3. Разработать программу на Python.

Даны длины сторон треугольника, определить вид треугольника и его площадь.

Выполнить контроль вводимых чисел.

1. Остроугольный треугольник
2. Тупоугольный треугольник
3. Прямоугольный треугольник

Ограничения:

- три числа не могут быть определены как стороны треугольника;
- если хотя бы одно из них меньше или равно 0;
- сумма двух из них меньше третьего.

Подготовить набор тестовых вариантов для обнаружения ошибок в программе и оформить результат.

Задание 4. На основании проведенных тестов составьте рекомендации по исправлению ошибок, выявленных в ходе тестирования в виде отчета.

Пример:

1 тест. В ходе проведения первого теста было обнаружено, что при в ведении не корректных данных площадь все равно высчитывается.

Рекомендуется: в случае, если пользователь введет не корректные данные, следует выводить сообщение с просьбой исправить введенные значения. Добавить в программу проверку введенных значений на соответствие ограничения.

Практическая работа № 2.4. Тестирование «черным ящиком»

Практическая работа № 2.5. Тестирование «черным ящиком»

Цель работы: изучить метод тестирования «Черным ящиком»

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 каждым годом. 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).

Метод чёрного ящика (black box testing, closed box testing) – у тестировщика либо нет доступа к внутренней структуре и коду приложения, либо недостаточно знаний для их понимания, либо он сознательно не обращается к ним в процессе тестирования.

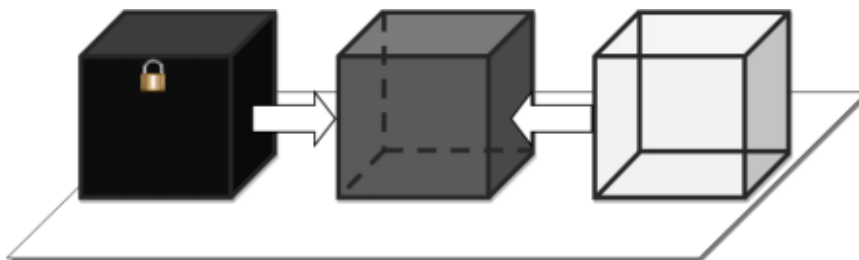


Рисунок 1

Разработка тестов методом черного ящика (black box test design technique)

Процедура создания и/или выбора тестовых сценариев, основанная на анализе функциональной или нефункциональной спецификации компонента или системы без знания внутренней структуры.

Техники разработки тестов на основе спецификаций, или методе черного ящика:

- эквивалентное разбиение;
- анализ граничных значений;
- тестирование таблицы решений;

Эквивалентное разбиение (equivalence partitioning)

Разработка тестов методом черного ящика, в которой тестовые сценарии создаются для проверки элементов эквивалентной области. Как правило, тестовые сценарии разрабатываются для покрытия каждой области как минимум один раз.

Входные данные для программного обеспечения или системы разбиваются на группы, от которых ожидается сходное поведение, то есть они должны обрабатываться аналогичным образом. Эквивалентные области (или классы) могут быть определены как для валидных, так и для невалидных данных, то есть тех значений, которые должны отвергаться.

Эквивалентное разбиение применимо на всех уровнях тестирования. Эквивалентное разбиение может быть использовано с целью покрытия входных и выходных данных. Оно может применяться при ручном вводе данных, при передаче данных через интерфейсы в систему, или при проверке параметров интерфейсов в интеграционном тестировании.

Анализ граничных значений (boundary value analysis): Разработка тестов методом черного ящика, при котором тестовые сценарии проектируются на основании граничных значений.

Граничное значение (boundary value): Входное значение или выходные данные, которое находится на грани эквивалентной области или на наименьшем расстоянии от обеих сторон грани, например, минимальное или максимальное значение области. Анализ граничных значений может применяться на всех уровнях тестирования

Таблица решений (decision table): Таблица, отражающая комбинации входных данных и/или причин с соответствующими выходными данными и/или действиям (следствиям), которая может быть использована для проектирования тестовых сценариев.

Таблицы решений – хороший метод для сбора системных требований, содержащих логические условия и документирования внутреннего дизайна системы. Они могут использоваться для записи сложных бизнес-правил, которые должна реализовывать система. Анализируются спецификации и определяются условия и действия системы. Входные условия и действия чаще всего формулируются таким образом, чтобы они могли принимать логические значения «истина» или «ложь».

Сильной стороной тестирования таблицы решений является то, что она создает комбинации условий, которые могли бы быть не проверены в ходе тестирования иным способом. Этот метод может быть применен ко всем ситуациям, в которых действие программного продукта зависит от нескольких логических альтернатив.

Задание 1. Написать калькулятор с небольшими багами.

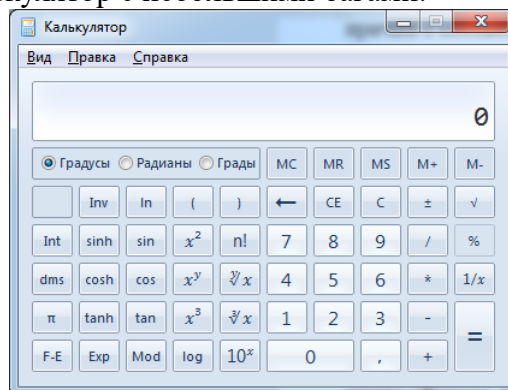


Рисунок 2

Задание 2. Обменяться программой с другими студентами. Провести тестирование и написать отчет в тетради.

Таблица 2

| Название теста | Описание сценария | Входные данные | Выходные данные | Удачное/неудачное тестирование | Предложения по исправлению найденных ошибок. | Пожелания пользователей |
|----------------|-----------------------------------------------------------|--------------------------------------------|-----------------|--------------------------------|----------------------------------------------|----------------------------------------------|
| Функция суммы | Сложение двух положительных чисел; Проверка результата | Первая переменная=3 Вторая переменная=8 | Результат=11 | Неудачное | - | Поле для ввода значений и вывода, объединить |

Практическая работа № 2.6. Модульное тестирование

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 каждым годом. 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 SWEBOOK.

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, SWEBOOK, 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.

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace MathTaskClassLibrary
8 {
9     public class Geometry
10     {
11         public int RectangleArea(int a, int b)
12         {
13             return a * b;
14         }
15     }
16 }
```

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...”.

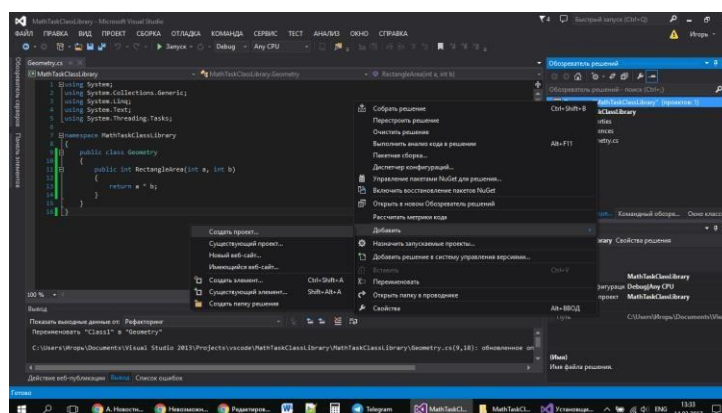


Рисунок 3

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

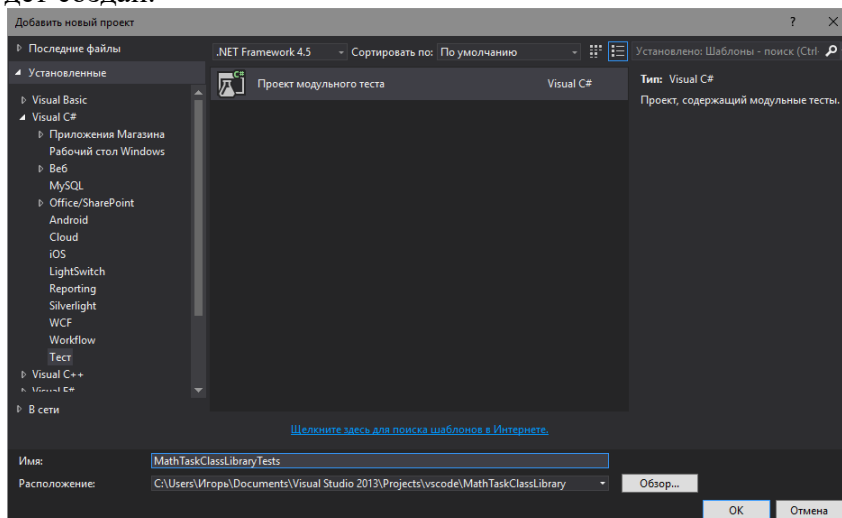


Рисунок 4

Перед Вами появится следующий код:

```
1 using System;
2 using Microsoft.VisualStudio.TestTools.UnitTesting;
3
4 namespace MathTaskClassLibraryTests
5 {
6     [TestClass]
7     public class UnitTest1
8     {
9         [TestMethod]
10        public void TestMethod1()
11        {
12        }
13    }
14 }
```

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”.

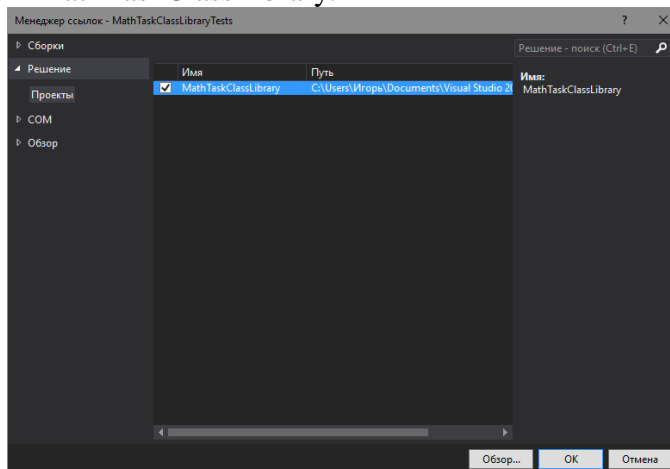


Рисунок 6

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

Займёмся написанием 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:

```

1 using System;
2 using Microsoft.VisualStudio.TestTools.UnitTesting;
3 using MathTaskClassLibrary;
4
5 namespace MathTaskClassLibraryTests
6 {
7     [TestClass]
8     public class GeometryTests
9     {
10         [TestMethod]
11         public void RectangleArea_3and5_15returned()
12         {
13             // исходные данные
14             int a = 3;
15             int b = 5;
16             int expected = 15;
17
18             // получение значения с помощью тестируемого метода
19             Geometry g = new Geometry();
20             int actual = g.RectangleArea(a, b);
21
22             // сравнение ожидаемого результата с полученным
23             Assert.AreEqual(expected, actual);
24         }
25     }
26 }

```

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”.

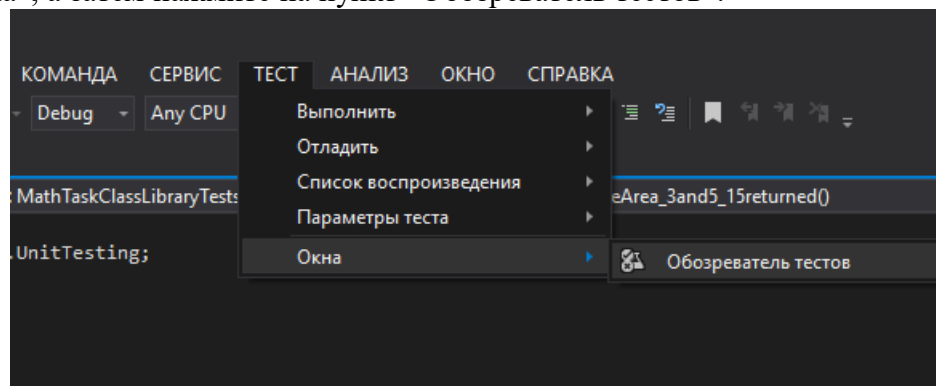


Рисунок 8

В студии появится следующее окно:

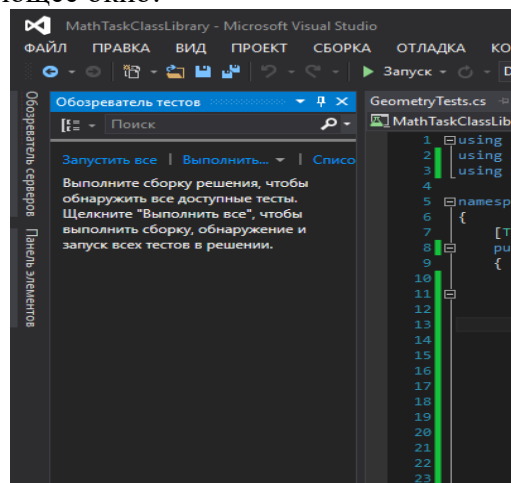


Рисунок 9

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.

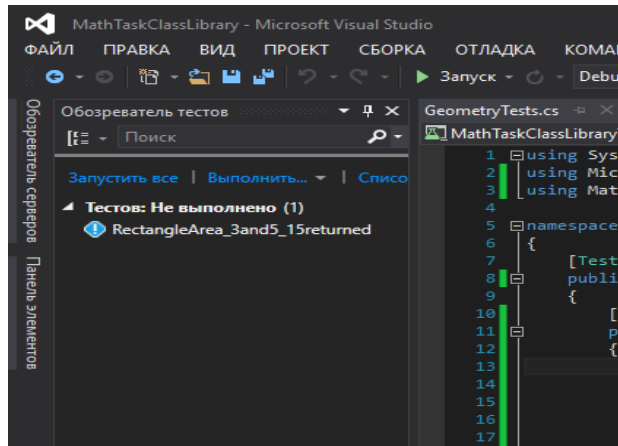


Рисунок 10

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”.

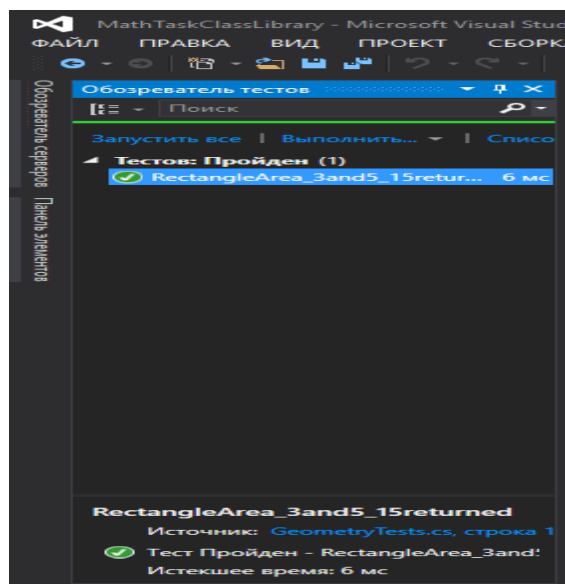


Рисунок 11

Зелёный кружок с галочкой означает, что модульный тест успешно пройден: ожидаемый и полученный результаты равны.

Изменим код метода `RectangleArea`, вычисляющего площадь прямоугольника, чтобы симитировать провал теста и посмотреть, как поведёт себя Visual Studio. Прибавим к возвращаемому значению 10.

Запустим unit-тест.

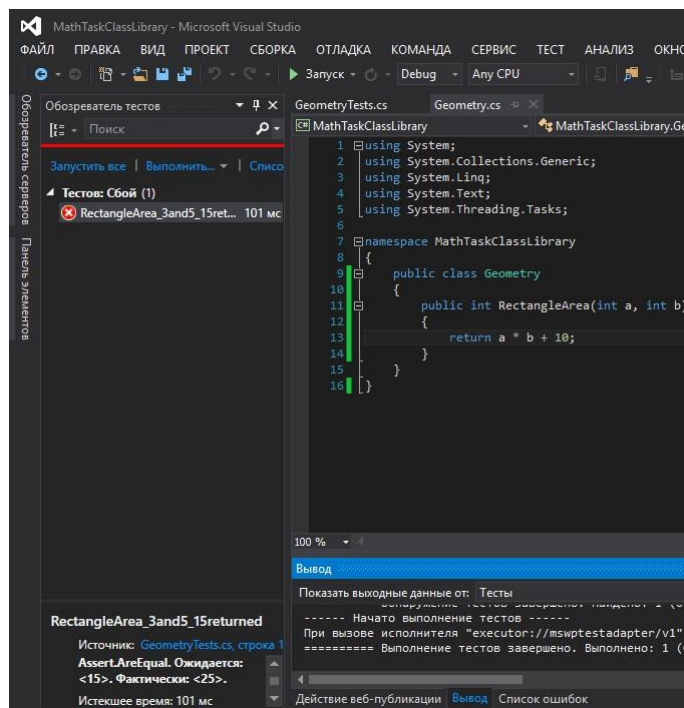


Рисунок 12

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 № 2.7. Интеграционное тестирование

Цель работы: Овладение навыками интеграционного тестирования.

Общие сведения

Интеграционное тестирование называют еще тестированием архитектуры системы. С одной стороны, это название обусловлено тем, что интеграционные тесты включают в себя проверки всех возможных видов взаимодействий между программными модулями и элементами, которые определяются в архитектуре системы – таким образом, интеграционные тесты проверяют полноту взаимодействий в тестируемой реализации системы. С другой стороны, результаты выполнения интеграционных тестов - один из основных источников информации для процесса улучшения и уточнения архитектуры системы, межмодульных и межкомпонентных интерфейсов. Т.е., с этой точки зрения, интеграционные тесты проверяют корректность взаимодействия компонент системы. В результате проведения интеграционного тестирования и устранения всех выявленных дефектов получается согласованная и целостная архитектура программной системы, т.е. можно считать, что интеграционное тестирование - это тестирование архитектуры и низкоуровневых функциональных требований. Интеграционное тестирование, как правило, представляет собой итеративный процесс, при котором проверяется совокупность модулей, возрастающая от итерации к итерации. В интеграционном тестировании выделяют три метода выполнения: восходящее тестирование; монолитное тестирование; нисходящее тестирование.

Задание: Согласно варианту провести один из методов интеграционного тестирования.

Практическая работа № 2.8. Разработка алгоритма поставленной задачи и реализация его средствами автоматизированного проектирования

Практическая работа № 2.9. Разработка алгоритма поставленной задачи и реализация его средствами автоматизированного проектирования

Цель работы: разработка алгоритма для создания программного продукта и анализ предметной области

Анализ поставленной задачи

Необходимо написать программу, которая будет выполнять действия на матрицах: умножения, сложения, вычитания, транспонирования. Программа должна решать введенные вручную матрицу в форму. Для удобства пользователя программа должна иметь интуитивно понятный интерфейс.

Выбор методов и разработка основных алгоритмов решения

В программе используется следующий алгоритм работы: в программе есть формы, в которые вводятся элементы матриц, элементы переводятся из String типа в Integer. Затем нужно нажать кнопку соответствующего действия. Выполняется алгоритм решения матриц и результат выводится в элемент DataGridView.

Для построения блок-схем использовалась программа Microsoft Office Visio 2013. С её помощью можно составлять различные диаграммы и схемы, в том числе, блок-схемы.

Разработка кода программного продукта на основе готовой спецификации на уровне модуля

Калькулятор матриц реализован на языке программирования C# в среде программирования Microsoft Visual Studio Ultimate 2013. Выбор языка C# обусловлен тем, что он современный и популярный объектно-ориентированный язык программирования, а среда Microsoft Visual Studio Ultimate 2013 является мощным средством, позволяющим быстро создать программу, обладающую графическим оконным интерфейсом.

На форме располагается 3 элемента DataGridView, в них будут размещаться матрицы. Так же 4 Button для выполнения действий над матрицами.

Использование инструментальных средств на этапе отладки программного модуля

При отладке программного продукта необходимо воспользоваться командой меню Отладка. В меню отладка существуют ряд команд, назначение которых представлено ниже.

Окна – открывает в интегрированной среде окно Точки останова, которое дает доступ ко всем точкам останова данного решения. Показывает в интегрированной среде окно Вывод.

Окно Вывод – это бегущий журнал множества сообщений, выдаваемых интегрированной средой, компилятором и отладчиком. Поэтому эта информация относится не только к сеансу отладки, а также открывает в интегрированной среде окно Интерпретация, которое позволяет выполнять команды:

- начать отладку – запускает приложение в режиме отладки;
- присоединиться к процессу – позволяет прикрепить отладчик к выполняющемуся процессу (исполняемому файлу). например, если запущено приложение без отладки, то можете потом прицепиться к этому выполняющемуся процессу и начать отладку;
- исключения – открывает диалоговое окно Исключения, которое позволяет выбрать способ останова отладчика для каждого исключительного состояния;
- шаг с заходом – запускает приложение в режиме отладки. для большинства проектов выбор команды шаг с заходом означает вызов отладчика на первой выполняемой строке приложения. таким образом, можно войти в приложение с первой строки;
- шаг с обходом – когда вы не находитесь в сеансе отладки, то команда шаг с обходом просто запускает приложение точно так же, как это сделала бы кнопка run;
- точка останова – включает или выключает точку останова на текущей (активной) строке кода текстового редактора. эта опция неактивна, если в интегрированной среде нет активного кодового окна;
- создавать точку останова – активирует диалоговое окно создавать точку останова позволяющее указать имя функции, для которой необходимо создать точку останова;
- удалить все точки останова – удаляет все точки останова из текущего решения;
- очистить все подсказки по данным – деактивирует (без удаления) все точки останова текущего решения;
- параметры и настройки – Прерывать выполнение, когда исключения пересекают границу домена приложения или границу между управляемым и машинным кодом.

Проведение тестирования программного модуля по определенному сценарию
Оценочное тестирование, которое также называют «тестированием системы в целом» целью которого является тестирование программы на соответствие основным требованиям. Эта стадия тестирования особенно важна для программных продуктов. 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 цветное 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 математических моделей;
- тестирование восстановления – проверка восстановления программного обеспечения, например, системы, включающей базу данных, после сбоя 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.

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.

Оформление документации на программное средство

Созданный программный продукт предназначен для выполнения арифметических действий над матрицами.

Чтобы запустить программу нужно запустить приложение.

Для того чтобы создать матрицы, необходимо ввести размерности матрицы и нажать кнопки «Построить». Затем ввести данные в матрицу и выбрать желаемое действие.

Программа имеет удобный интерфейс и предоставляет возможность с легкостью решать матрицы произвольных размерностей.

Моделирование бизнес- процессов предметной области

Основные элементы и понятия IDEF0. Графический язык IDEF0 удивительно прост и гармоничен. В основе методологии лежат четыре основных понятия:

Первым из них является понятие функционального блока (Activity Box). Функциональный блок графически изображается в виде прямоугольника и олицетворяет собой некоторую конкретную функцию в рамках рассматриваемой системы. По требованиям стандарта название каждого функционального блока должно быть сформулировано в глагольном наклонении (например, “производить услуги”, а не “производство услуг”).

Каждая из четырех сторон функционального блока имеет своё определенное значение (роль), при этом:

- Верхняя сторона имеет значение "Управление" (Control);
- Левая сторона имеет значение "Вход" (Input);
- Правая сторона имеет значение "Выход" (Output);
- Нижняя сторона имеет значение "Механизм" (Mechanism).

Каждый функциональный блок в рамках единой рассматриваемой системы должен иметь свой уникальный идентификационный номер.

Вторым элементом методологии IDEF0 является понятие интерфейсной дуги (Arrow). Также интерфейсные дуги часто называют потоками или стрелками. Интерфейсная дуга отображает элемент системы, который обрабатывается функциональным блоком или оказывает иное влияние на функцию, отображенную данным функциональным блоком.

Графическим отображением интерфейсной дуги является однонаправленная стрелка. Каждая интерфейсная дуга должна иметь свое уникальное наименование (Arrow Label). По требованию стандарта, наименование должно быть оборотом существительного.

С помощью интерфейсных дуг отображают различные объекты, в той или иной степени определяющие процессы, происходящие в системе. Такими объектами могут быть элементы реального мира (детали, вагоны, сотрудники и т.д.) или потоки данных и информации (документы, данные, инструкции и т.д.).

В зависимости от того, к какой из сторон подходит данная интерфейсная дуга, она носит название “входящей”, “исходящей” или “управляющей”. Кроме того, “источником” (началом) и “приемником” (концом) каждой функциональной дуги могут быть только функциональные блоки, при этом “источником” может быть

только выходная сторона блока, а “приемником” любая из трех оставшихся.

Необходимо отметить, что любой функциональный блок по требованиям стандарта должен иметь по крайней мере одну управляющую интерфейсную дугу и одну исходящую. Это и понятно – каждый процесс должен происходить по каким-то правилам (отображаемым управляющей дугой) и должен выдавать некоторый результат (выходящая дуга), иначе его рассмотрение не имеет никакого смысла.

При построении IDEF0 – диаграмм важно правильно отделять входящие интерфейсные дуги от управляющих, что часто бывает непросто.

В реальном процессе рабочему, производящему обработку, выдают заготовку и технологические указания по обработке (или правила техники безопасности при работе со станком). Ошибочно может показаться, что и заготовка и документ с технологическими указаниями являются входящими объектами, однако это не так. На самом деле в этом процессе заготовка обрабатывается по правилам отраженным в технологических указаниях, которые должны соответственно изображаться управляющей интерфейсной дугой.

Например, в случае рассмотрения предприятий и организаций существуют пять основных видов объектов: материальные потоки (детали, товары, сырье и т.д.), финансовые потоки (наличные и безналичные, инвестиции и т.д.), потоки документов (коммерческие, финансовые и организационные документы), потоки информации (информация, данные о

намерениях, устные распоряжения и т.д.) и ресурсы (сотрудники, станки, машины и т.д.). При этом в различных случаях входящими и исходящими интерфейсными дугами могут отображаться все виды объектов, управляющими только относящиеся к потокам документов и информации, а дугами-механизмами только ресурсы.

Обязательное наличие управляющих интерфейсных дуг является одним из главных отличий стандарта IDEF0 от других методологий классов DFD (Data Flow Diagram) и WFD (Work Flow Diagram).

Третьим основным понятием стандарта IDEF0 является декомпозиция (Decomposition). Принцип декомпозиции применяется при разбиении сложного процесса на составляющие его функции.

Декомпозиция позволяет постепенно и структурированно представлять модель системы в виде иерархической структуры отдельных диаграмм, что делает ее менее перегруженной и легко усваиваемой.

Модель IDEF0 всегда начинается с представления системы как единого целого – одного функционального блока с интерфейсными дугами, простирающимися за пределы рассматриваемой области. Такая диаграмма с одним функциональным блоком называется контекстной диаграммой, и обозначается идентификатором “А-0”.

В пояснительном тексте к контекстной диаграмме должна быть указана цель (Purpose) построения диаграммы в виде краткого описания и зафиксирована точка зрения (Viewpoint).

Определение и формализация цели разработки IDEF0 – модели является крайне важным моментом. Фактически цель определяет соответствующие области в исследуемой системе, на которых необходимо фокусироваться в первую очередь. Например, если мы моделируем деятельность предприятия с целью построения в дальнейшем на базе этой модели информационной системы, то эта модель будет существенно отличаться от той, которую бы мы разрабатывали для того же самого предприятия, но уже с целью оптимизации логистических цепочек.

Точка зрения определяет основное направление развития модели и уровень необходимой детализации. Четкое фиксирование точки зрения позволяет разгрузить модель, отказавшись от детализации и исследования отдельных элементов, не являющихся необходимыми, исходя из выбранной точки зрения на систему. Например, функциональные модели одного и того же предприятия с точек зрения главного технолога и финансового директора будут существенно различаться по направленности их детализации. Это связано с тем, что в конечном итоге, финансового директора не интересуют аспекты обработки сырья на производственных станках, а главному технологу ни к чему прорисованные схемы финансовых потоков. Правильный выбор точки зрения существенно сокращает временные затраты на построение конечной модели.

В процессе декомпозиции, функциональный блок, который в контекстной диаграмме отображает систему как единое целое, подвергается детализации на другой диаграмме. Получившаяся диаграмма второго уровня содержит функциональные блоки,

отображающие главные подфункции функционального блока контекстной диаграммы и называется дочерней (Child diagram) по отношению к нему (каждый из функциональных блоков, принадлежащих дочерней диаграмме соответственно называется дочерним блоком – Child Box). В свою очередь, функциональный блок - предок называется родительским блоком по отношению к дочерней диаграмме (Parent Box), а диаграмма, к которой он принадлежит – родительской диаграммой (Parent Diagram). Каждая из подфункций дочерней диаграммы может быть далее детализирована путем аналогичной декомпозиции соответствующего ей функционального блока. Важно отметить, что в каждом случае декомпозиции функционального блока все интерфейсные дуги, входящие в данный блок, или исходящие из него фиксируются на дочерней диаграмме. Этим достигается структурная целостность IDEF0 – модели. Наглядно принцип декомпозиции представлен на рисунке 2.4. Следует обратить внимание на взаимосвязь нумерации функциональных блоков и диаграмм - каждый блок имеет свой уникальный порядковый номер на диаграмме (цифра в правом нижнем углу прямоугольника), а обозначение под правым углом указывает на номер дочерней для этого

блока диаграммы. Отсутствие этого обозначения говорит о том, что декомпозиции для данного блока не существует.

Часто бывают случаи, когда отдельные интерфейсные дуги не имеет смысла продолжать рассматривать в дочерних диаграммах ниже какого-то определенного уровня в иерархии, или наоборот - отдельные дуги не имеют практического смысла выше какого-то уровня. Например, интерфейсную дугу, изображающую "деталь" на входе в функциональный блок "Обработать на токарном станке", не имеет смысла отражать на диаграммах более высоких уровней – это будет только перегружать диаграммы и делать их сложными для восприятия. С другой стороны, случается необходимость избавиться от отдельных "концептуальных" интерфейсных дуг и не детализировать их глубже некоторого уровня. Для решения подобных задач в стандарте IDEF0 предусмотрено понятие туннелирования. Обозначение "туннеля" (Arrow Tunnel) в виде двух круглых скобок вокруг начала интерфейсной дуги обозначает, что эта дуга не была

унаследована от функционального родительского блока и появилась (из "туннеля") только на этой диаграмме. В свою очередь, такое же обозначение вокруг конца (стрелки) интерфейсной дуги в непосредственной близости от блока – приёмника означает тот факт, что в дочерней по отношению к этому блоку диаграмме эта дуга отображаться и рассматриваться не будет. Чаще всего бывает, что отдельные объекты и соответствующие им интерфейсные дуги не рассматриваются на некоторых промежуточных уровнях иерархии – в таком случае, они сначала "погружаются в туннель", а затем, при необходимости "возвращаются из туннеля".

Последним из понятий IDEF0 является глоссарий (Glossary). Для каждого из элементов IDEF0: диаграмм, функциональных блоков, интерфейсных дуг существующий стандарт подразумевает создание и поддержание набора соответствующих определений, ключевых слов, повествовательных изложений и т.д., которые характеризуют объект, отображенный данным элементом. Этот набор называется глоссарием и является описанием сущности данного элемента. Например, для управляющей интерфейсной дуги "распоряжение об оплате" глоссарий может содержать перечень полей соответствующего дуге документа, необходимый набор виз и т.д. Глоссарий гармонично дополняет наглядный графический язык, снабжая диаграммы необходимой дополнительной информацией.

Принципы ограничения сложности IDEF0-диаграмм. Обычно IDEF0-модели несут в себе сложную и концентрированную информацию, и для того, чтобы ограничить их перегруженность и сделать удобочитаемыми, в соответствующем стандарте приняты соответствующие ограничения сложности:

- Ограничение количества функциональных блоков на диаграмме тремя-шестью. Верхний предел (шесть) заставляет разработчика использовать иерархии при описании сложных предметов, а нижний предел (три) гарантирует, что на соответствующей диаграмме достаточно деталей, чтобы оправдать ее создание;
- Ограничение количества подходящих к одному функциональному блоку (выходящих из одного функционального блока) интерфейсных дуг четырьмя.

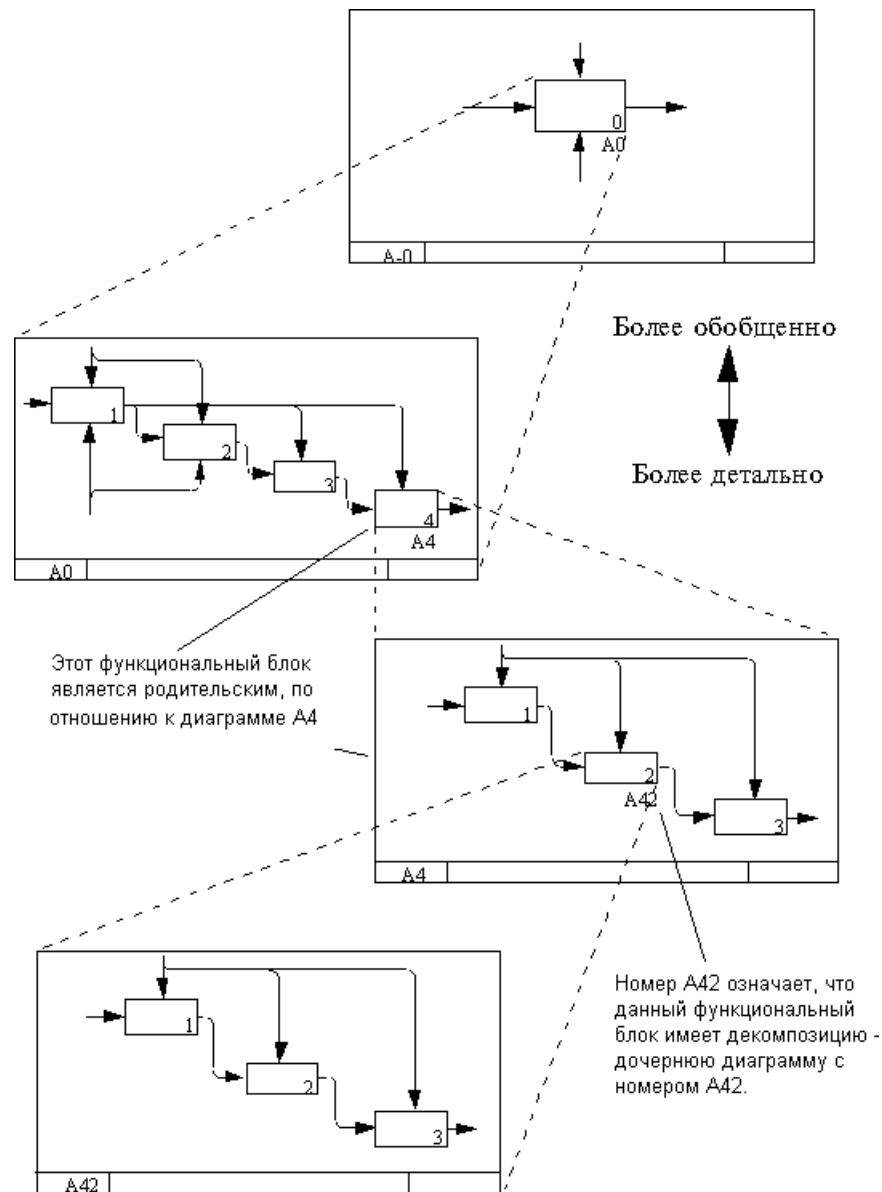


Рисунок 13

Разумеется, строго следовать этим ограничениям вовсе необязательно, однако, как показывает опыт, они являются весьма практичными в реальной работе.

Порядок выполнения практической работы

Для заданного преподавателем описания конкретного бизнес- процесса создать контекстную диаграмму A-0. Выделить основные его функции и создать диаграмму A0. Разбить каждую функцию на подфункции и диаграммы нижних уровней.

Спроектируйте логическую модель БД (прямое моделирование) в соответствии с Вашим вариантом. Задайте атрибуты для каждой определенной сущности. Введите связи между сущностями. Присвойте связям уникальные имена.

Контрольные вопросы

1. Каковы стадии жизненного цикла информационных систем, их основное содержание?
2. Как представляется функциональная модель деятельности в методологии IDEF0?
3. Каковы основные объекты диаграмм функциональной модели по методологии IDEF0?
4. Что обозначают работы в диаграммах функциональной модели, как они отображаются по методологии IDEF0?
5. Для чего предназначены стрелки в диаграммах функциональной модели, каковы их типы и виды?
6. Для чего предназначен словарь стрелок?
7. Каковы типы связей работ по методологии IDEF0?

8. Что такое туннелирование стрелок, для чего оно нужно, каковы виды туннелирования?

Практическая работа № 2.10. Использование инструментальных средств на этапе отладки программного модуля

Практическая работа № 2.11. Использование инструментальных средств на этапе отладки программного модуля

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

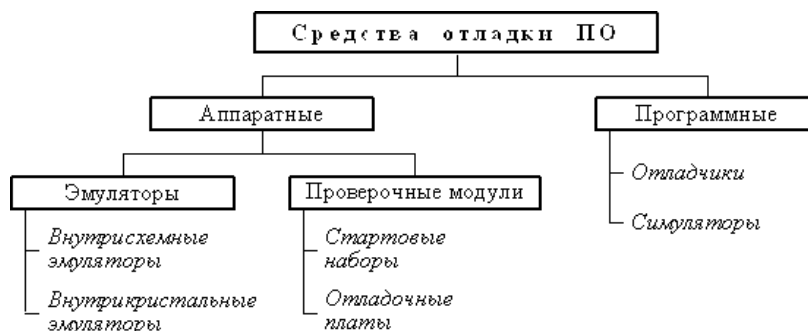


Рисунок 14

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 соединяется 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

Внутрикристалльные эмуляторы (On-Chip Emulator) позволяют проводить отладку программ без извлечения микропроцессора из системы. При этом осуществляется непосредственный контроль за выполнением программы, так как средства внутрикристалльной отладки обеспечивают прямой доступ к регистрам, памяти и периферии микропроцессора. Наиболее распространённым средством внутрикристалльной 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.

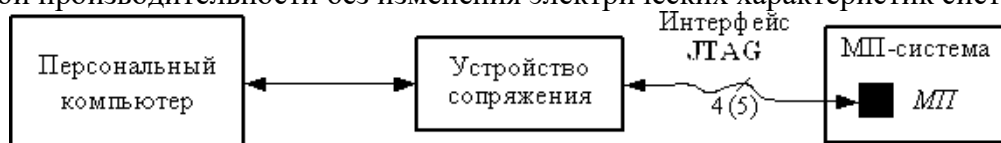


Рисунок 16

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 Studio в режиме симулятора.

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

Переход в режим отладчика в среде AVR 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 достижении 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 окно Watches, 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 окон Memory, 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. Обратит 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 В. 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 отладчика AVR 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 № 2.12. Тестирование на этапе сопровождения программного продукта

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 цветное 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 конвейеры, 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 сбоя 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 инсталлятор. Содержит встроенный скриптовый 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 – инструмент 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.

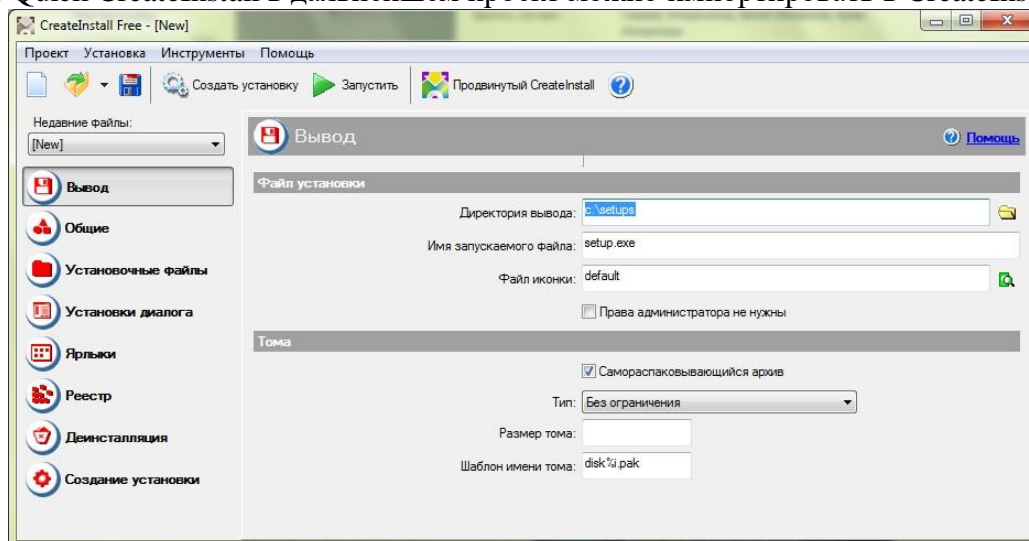


Рисунок 17

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 Installer, 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.

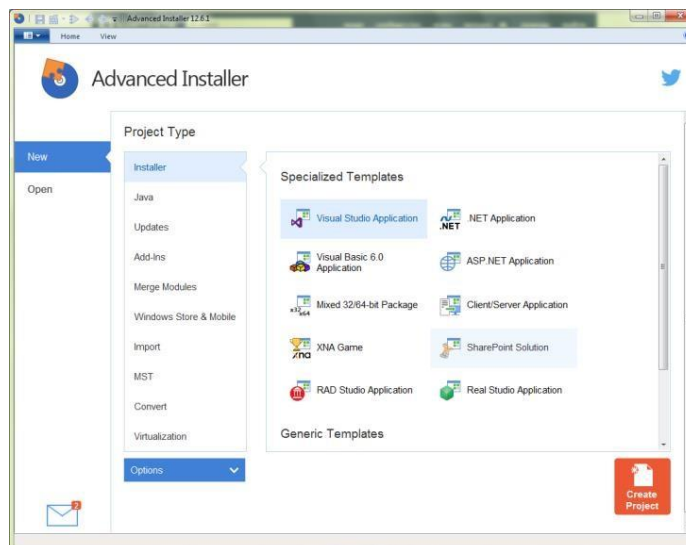


Рисунок 18

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.

Практическая работа № 2.13. Место верификации среди процессов разработки программного обеспечения

Цель работы: Изучение основ верификации процессов разработки программного обеспечения.

Наши занятия будут направлены на приобретение практических навыков в области тестирования и верификации программного обеспечения (ПО). На протяжении всего семестра мы будем изучать данный курс на одном сквозном примере – на программном продукте "Калькулятор".

Предположим, что мы являемся частью коллектива разработчиков, которому поступил заказ на разработку программной системы "Калькулятор" (в дальнейшем просто "Система"). Предположим также, что другая часть коллектива уже сформировала функциональные требования, архитектуру и написала программный код системы. Таким образом, на нас ложится участок жизненного цикла системы по тестированию и проверке требований.

Система "Калькулятор"

Основная цель Системы – вычисление математических выражений с корректной структурой. Формально данное предложение раскрыто в приложении к семинару, а здесь дадим некоторые комментарии. В самом простом случае, будем считать корректными следующие выражения:

1
1+1
(1+1)
(1+1)*2

и т.д., то есть, выражения, корректные в математическом смысле.

Однако, множество вычисляемых калькулятором выражений все же несколько "меньше", чем просто корректные математические выражения. Это связано с некоторыми математическими операциями и дробными числами, корректность обработки которых сложно протестировать: не будем забывать, что Калькулятор — прежде всего учебный пример.

Требования к системе

Система должна выполнять свою основную цель двумя способами: с помощью графического интерфейса и с помощью командной строки.

Архитектура

В архитектуре системы выделено 3 модуля. Каждый из модулей занимается определенной задачей. Соответственно, Система – это взаимодействие этих 3-х модулей. Разбиение Системы на модули вытекает из различной функциональности этих модулей. Рассмотрим их.

1. Модуль математических функций – так как Система будет иметь дело с математикой, нам потребуется подобный модуль. В него включены такие функции, как сложение, умножение и др.

2. Модуль анализа и вычисления выражений – это модуль, который занимается главной задачей Системы. Разбор и компиляция выражений – вот основные функции этого

модуля. Непосредственные вычисления этот модуль не проводит, а лишь вызывает функции из математического модуля.

3. Модуль графического интерфейса – обеспечивает управление системы в графической форме. Основные функции этого модуля – ввод и вывод данных.

Взаимодействие модулей показано на рис 19

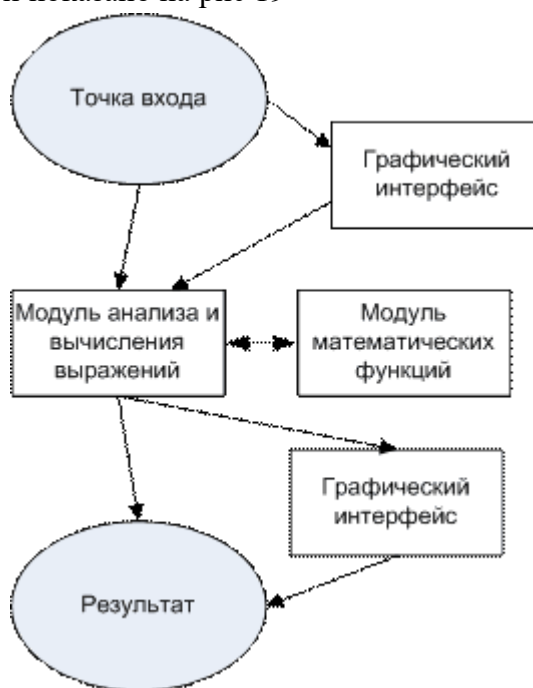


Рисунок 19

Как видно из рисунка, передать данные в Систему можно двумя способами: либо через графический модуль, либо через командную строку (последнее неявно прослеживается по рисунку). В любом случае, после передачи выражения Системе начинает работу модуль анализа и вычислений, который по мере необходимости использует модуль математики для вычисления арифметических функций. После окончания работы модуля анализа и вычислений на выход передается результат.

Программный код

Весь программный код Системы разбит на модули соответственно архитектуре. Это позволит нам тестировать каждый модуль отдельно, о чем мы и будем говорить далее.

Тестирование системы

Общее описание

Как было сказано, наша основная задача – протестировать Систему.

Если в общем случае рассматривать жизненный цикл системы (например, V-образный), то наша задача лежит где-то справа.

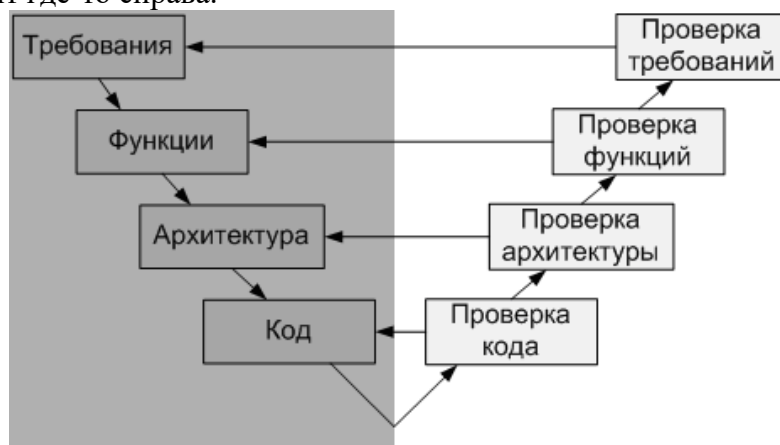


Рисунок 20

Применительно к нашей системе, жизненный цикл можно представить согласно рис 21. Сначала необходимо проверить код, затем архитектуру и требования.

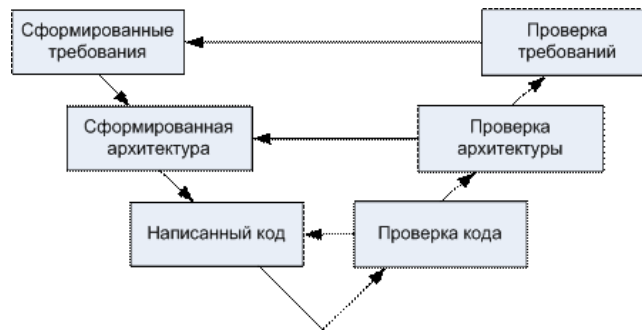


Рисунок 21

Проверка программного кода

На этом этапе необходимо проверить корректность работы написанного кода. Для этого предлагается проводить тестирование каждого модуля отдельно. То есть, мы будем тестировать модули по отдельности, подменяя используемые методы других модулей "заглушками".

Например, при тестировании модуля анализа и вычислений выражений модуль, отвечающий за вычисления простых математических функций, можно заменить на модуль, содержащий стандартные методы области Math. Так мы будем точно знать, что все ошибки, выявленные при тестировании, не имеют отношения к нашей заглушке. Таким образом, заменив все модули, кроме тестируемого, заглушками, мы сможем утверждать, что все ошибки, обнаруженные при тестировании, будут относиться к "настоящему" (тестируемому) модулю.

Более того, заглушки дают нам дополнительное преимущество в тестировании. Мы можем написать заглушки, возвращающие пользователю дополнительную информацию во время тестирования. Например, нам необходимо узнать значение определенной переменной во время выполнения программы. Для этого мы можем написать заглушку, которая будет записывать значение этой переменной в лог-файл или на консоль.

Немного о тестировании конкретных модулей.

GUI. На примере этого модуля можно было бы узнать, какие подходы существуют для тестирования современного графического интерфейса. В рамках данного курса этот вид тестирования рассматриваться не будет.

Математические функции. Этот модуль мы будем исследовать как "черный ящик" и выяснять, действительно ли реализованные в нем математические функции работают корректно.

Вычисление выражений. При тестировании этого модуля нам предстоит проверить корректность алгоритмов разбора и компиляции математических выражений.

При тестировании будет использоваться следующая последовательность действий.

Сначала мы познакомимся с методами ручного тестирования в среде разработки при ручном тестировании модуля анализа и вычисления выражений. Затем мы перейдем к модульному тестированию. В завершении тестирования компонент мы проведем формальные инспекции кода. После этого мы узнаем, что такое покрытия и как они используются в процессе тестирования.

Проверка архитектуры

После проверки каждого модуля по отдельности мы проведем интеграционное тестирование. На этом этапе проверяется, как модули взаимодействуют друг с другом. При условии, что все модули протестированы и ошибок в них не выявлено, все ошибки на этом этапе будут относиться именно к взаимодействию модулей между собой.

Проверка требований

После прохождения всех этапов тестирования необходимо провести проверку требований Системы в целом, то есть провести системное тестирование. Но в рамках данного курса этот вид тестирования рассматриваться не будет.

Приложение. Спецификация на программу "Калькулятор. Базовая версия" (с комментариями для преподавателя)

Данная спецификация требований далеко не полна, в частности, не полна спецификация пользовательского интерфейса, функциональных требований. Студентам предполагается дополнить спецификацию самостоятельно.

Общее описание

Калькулятор состоит из трех модулей – "Графический интерфейс", "Модуль, анализирующий и вычисляющий введенное выражение" (AnalazerClass.dll) и "Модуль, реализующий математические функции" (CalcClass.dll). После того, как пользователь введет вычисляемое выражение одним из двух вышеописанных способов, управление передается анализирующему модулю, который форматирует выражение, выделяя числа и операторы, проверяет корректность скобочной структуры, а также выявляет неверные с точки зрения математики конструкции (например, $3+*+3$), переводит выражение в обратную польскую запись, после чего вычисляет выражения, используя математические функции из модуля CalcClass.

Описание интерфейса.

Входные данные

Параметры вызова (формат командной строки)

calc.exe [expression]

expression – математическое выражение, удовлетворяющее требованию

Состояние информационного окружения.

В папке с программой также находятся файлы CalcClass.dll, AnalazerClass.dll

Выходные данные.

Коды возврата программы.

Число и 0 на новой строке – результат вычислений выражения.

Error: <сообщение об ошибке> и код ошибки на новой строке — сообщение об ошибке в случае несоответствия входного выражения требованиям

Состояние информационного окружения после завершения программы.

В папке с программой также находятся файлы CalcClass.dll, AnalazerClass.dll

Сообщения об ошибках, выдаваемые программой (коды ошибок).

Error 01 at <i> — Неправильная скобочная структура, ошибка на <i> символе

Error 02 at <i> — Неизвестный оператор на <i> символе

Error 03 — Неверная синтаксическая конструкция входного выражения

Error 04 at <i> — Два подряд оператора на <i> символе

Error 05 — Незаконченное выражение

Error 06 — Слишком малое или слишком большое значение числа для int. Числа должны быть в пределах от -2147483648 до 2147483647

Error 07 — Слишком длинное выражение. Максимальная длина — 65536 символов.

Error 08 — Суммарное количество чисел и операторов превышает 30

Error 09 – Ошибка деления на 0

Описание файлов, входящих в пакет калькулятора.

CalcClass.dll – библиотека, в которой реализованы все необходимые математические функции.

AnalazerClass.dll – модуль, в котором реализован синтаксический разбор выражения, а также его вычисление.

calc.exe – графическая оболочка, главный модуль.

Интерфейс пользователя.

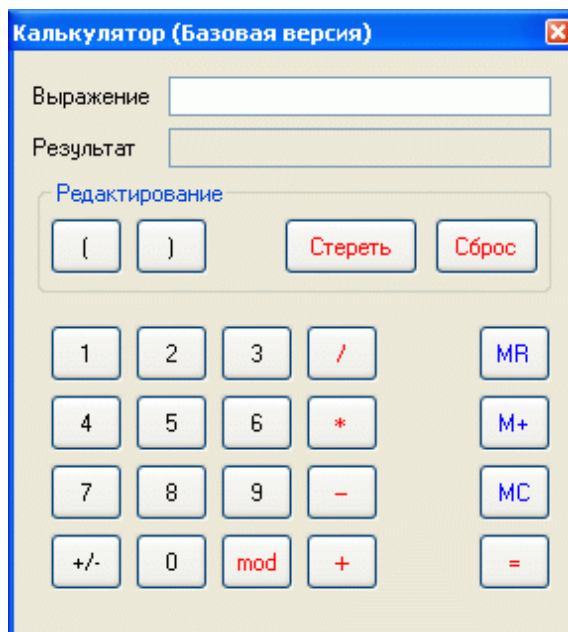


Рисунок 22

Клавиши "1" "2" "3" "4" "5" "6" "7" "8" "9" "0" "/" "*" "-" "+" "mod" "(" ")" – вводят соответствующий символ в поле выражение. Клавиша "Сброс" очищает поле "Выражение", клавиша "Стереть" удаляет последний введенный символ. Клавиша "=" начинает выполнение вычислений. "MR", "M+" и "MC" управляют памятью калькулятора, "+/-" — триггер унарного плюса унарного минуса.

Описание архитектуры

Как уже отмечалось выше, в архитектуре системы выделено 3 модуля. Каждый из модулей занимается определенной задачей. Соответственно, Система – это взаимодействие этих 3-х модулей. Рассмотрим их подробнее.

Модуль математических операций (CalcClass.dll)

Модуль содержит все математические функции, используемые в программе.

```

/// <summary>
    /// Функция сложения числа a и b
    /// </summary>
    /// <param name="a">слагаемое</param>
    /// <param name="b">слагаемое</param>
    /// <returns>сумма</returns>
    public static int Add(long a, long b)
    /// <summary>
    /// функция вычитания чисел a и b
    /// </summary>
    /// <param name="a">уменьшаемое</param>
    /// <param name="b">вычитаемое</param>
    /// <returns>разность</returns>
    public static int Sub(long a, long b)
    /// <summary>
    /// функция умножения чисел a и b
    /// </summary>
    /// <param name="a">множитель</param>
    /// <param name="b">множитель</param>
    /// <returns>произведение</returns>
    public static int Mult(long a, long b)
    /// <summary>
    /// функция нахождения частного
    /// </summary>

```

```

/// <param name="a">делимое</param>
/// <param name="b">делитель</param>
/// <returns>частное</returns>
public static int Div(long a, long b)
/// <summary>
/// функция деления по модулю
/// </summary>
/// <param name="a">делимое</param>
/// <param name="b">делитель</param>
/// <returns>остаток</returns>
public static int Mod(long a, long b)
/// <summary>
/// унарный плюс
/// </summary>
/// <param name="a"></param>
/// <returns></returns>
public static int ABS(long a)
/// <summary>
/// унарный минус
/// </summary>
/// <param name="a"></param>
/// <returns></returns>
public static int IABS(long a)
Используется также глобальная переменная:
/// <summary>
/// Последнее сообщение об ошибке
/// Поле и свойство для него
/// </summary>
private static string _lastError = "";
public static string lastError

```

Модуль математических операций

Модуль анализа и вычисления выражений

Состоит из следующих методов и свойств:

```

/// <summary>
/// позиция выражения, на которой отловлена синтаксическая ошибка (в
/// случае ловли на уровне выполнения - не определяется)
/// </summary>
private static int erposition = 0;
/// <summary>
/// Входное выражение
/// </summary>
public static string expression = "";
/// <summary>
/// Показывает, есть ли необходимость в выводе сообщений об ошибках.
/// В случае консольного запуска программы это значение - false.
/// </summary>
public static bool ShowMessage = true;
/// <summary>
/// Проверка корректности скобочной структуры входного выражения
/// </summary>
/// <returns>true - если все нормально, false - если есть
/// ошибка</returns>
/// метод бежит по входному выражению, символ за символом анализируя

```

```

его и считая количество скобочек. В случае возникновения
/// ошибки возвращает false, а в erposition записывает позицию, на
которой возникла ошибка.
public static bool CheckCurrency()
/// <summary>
/// Форматирует входное выражение, выставляя между операторами
пробелы и удаляя лишние, а также отлавливает неопознанные операторы, следит
за концом строки
/// также отлавливает ошибки на конце строки
/// </summary>
/// <returns>конечную строку или сообщение об ошибке, начинающиеся со
спец. символа &</returns>
public static string Format()
/// <summary>
/// Создает массив, в котором располагаются операторы и символы,
представленные в обратной польской записи (безскобочной)
/// На этом же этапе отлавливаются почти все остальные ошибки (см код). По сути
- это компиляция.
/// </summary>
/// <returns>массив обратной польской записи</returns>
public static System.Collections.ArrayList CreateStack()
/// <summary>
/// Вычисление обратной польской записи
/// </summary>
/// <returns>результат вычислений или сообщение об ошибке</returns>
public static string RunEstimate()
/// <summary>
/// Метод, организующий вычисления. По очереди запускает
CheckCorrncy, Format, CreateStack и RunEstimate
/// </summary>
/// <returns></returns>
public static string Estimate()

```

Модуль анализа и вычисления выражений

Модуль графического интерфейса – обеспечивает управление системы в графической форме. Основные функции этого модуля – ввод и вывод данных.

Взаимодействие модулей показано на рисунке:

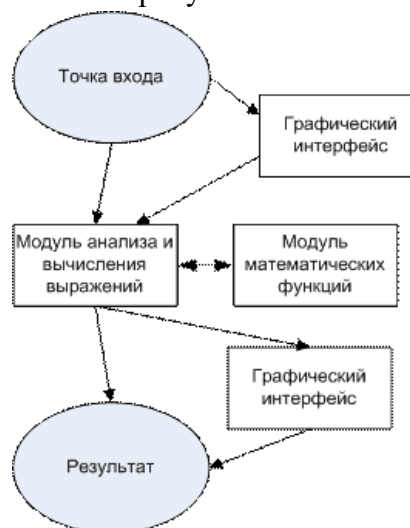


Рисунок 23

Функциональные требования

Требования к программе

Калькулятор должен выполнять следующие арифметические операции: сложение, вычитание, умножение, нахождение частного, нахождение остатка.

Калькулятор должен поддерживать работу с целыми числами в пределах от - 2147483648 до 2147483647 (в дальнейшем MININT и MAXINT). В случае выхода за эти пределы должно выдаваться сообщение об ошибке Error 06.

Калькулятор должен иметь память на одно целое число, а также возможность выводить это число на экран, сбрасывать его значение на 0 и прибавлять к нему любое другое число, введенное в поле ввода.

При нажатии на клавишу M+ к числу, записанному в память, прибавляется число, записанное в поле "Результат". При этом на сложение накладываются ограничения.

Если в поле "Результат" записан код ошибки, то при нажатии на клавишу M+ должно выдаваться сообщение "Невозможно преобразовать к числу".

При нажатии на кнопку MC число в памяти обнуляется.

При нажатии на кнопку MR число из памяти приписывается в конец выражения в строке "Выражение".

Калькулятор должен предоставлять возможность пользователю работать с операциями унарного плюса и унарного минуса.

Если между нажатиями на кнопку <+/-> проходит менее 3 секунд, то введенный оператор меняется на противоположный.

Если между нажатиями на кнопку <+/-> проходит более 3 секунд, то к выражению дописывается знак "-".

Калькулятор должен иметь графический интерфейс, содержащий кнопки с цифрами и арифметическими операциями, кнопкой равенства, кнопками работы с памятью, кнопками редактирования скобочек и кнопками сброса, переключателем унарного минуса/унарного плюса, текстовыми полями для ввода выражения и вывода результата.

При нажатии на клавишу <Enter> калькулятор должен проводить вычисления выражения.

При нажатии на клавишу <ESC> программа должна прекращать свою работу.

В случае неверно построенного вычисляемого выражения или несоответствия его требованиям в текстовое окно результата должно выводиться соответствующее сообщение.

Арифметические операции

Сложение.

Для чисел, каждое из которых меньше либо равно MAXINT и больше либо равно MININT, функция суммирования должна возвращать правильную сумму с точки зрения математики

Для чисел, сумма которых больше чем MAXINT и меньше чем MININT, а также в случае, если любое из слагаемых больше чем MAXINT или меньше чем MININT, программа должна выдавать ошибку Error 06

Вычитание.

Для чисел, каждое из которых меньше либо равно MAXINT и больше либо равно MININT, функция вычитания должна возвращать правильную разность с точки зрения математики

Для чисел, разность которых больше чем MAXINT и меньше чем MININT, а также в случае, если любое из чисел больше чем MAXINT или меньше чем MININT, программа должна выдавать ошибку Error 06

Умножение

Для чисел, произведение которых меньше либо равно MAXINT и больше либо равно MININT, функция умножения должна возвращать правильное произведение с точки зрения математики.

Для чисел, произведение которых больше чем MAXINT и меньше чем MININT, а также, в случае если любой из множителей больше чем MAXINT или меньше чем MININT, программа должна выдавать ошибку Error 06

Нахождение частного

Для чисел, меньших либо равных MAXINT и больших либо равных MININT, частное которых меньше либо равно MAXINT и больше либо равно MININT и делитель не равен 0, функция деления должна возвращать правильное частное с точки зрения математики.

Для чисел, частное которых больше чем MAXINT и меньше чем MININT, а также в случае, если любое из чисел больше чем MAXINT или меньше чем MININT, и для делителя, не равного 0, программа должна выдавать ошибку Error 06

Если делитель равен 0, программа должна выдавать ошибку Error 09

Деление с остатком

Для чисел, меньших либо равных MAXINT и больших либо равных MININT, остаток которых меньше либо равен MAXINT и больше либо равен MININT и делитель не равен 0, функция деления должна возвращать правильный остаток с точки зрения математики.

Для чисел, остаток которых больше чем MAXINT и меньше чем MININT, а также в случае, если любое из чисел больше чем MAXINT или меньше чем MININT, и для делителя, не равного 0, программа должна выдавать ошибку Error 06

Если делитель равен 0, программа должна выдавать ошибку Error 09

Унарный плюс \ минус.

Для чисел, меньших либо равных MAXINT и больших либо равных MININT, операция унарного плюса / минуса должна возвращать число соответствующего знака.

Для чисел больших MAXINT или меньших MININT функция должна выдавать ошибку Error 06

Дополнительные требования к входному выражению

Максимальное суммарное число операторов и чисел – 30.

Максимальная глубина вложенности скобочной структуры – 3.

В качестве унарного минуса используется символ " m ", в качестве унарного плюса — " p ".

Для операции нахождения частного – " / ", для нахождения остатка — " mod ".

Между операторами скобками и числами может быть любое количество пробелов.

Разрешается использовать лишь скобки вида " (" и ") "

Максимальная длина выражения – 65535 символов.

Практическая работа № 2.14. Тестовые примеры. Классы эквивалентности. Ручное тестирование в MVSTE

Цель работы: Изучение классов и тестирование в MVSTE.

Разработка тестовых примеров

Непосредственно для тестирования программного обеспечения необходимо определить проверочные задания, выполняемые системой или ее отдельной частью. Такие задачи называются тестовыми примерами.

Можно выделить два подхода к созданию тестовых примеров – исходя из функциональных требований (или из любой другой документации описывающей систему) и исходя из кода. Построение тестовых примеров, исходя из кода, и построение покрытия кода мы будем изучать позднее, а сейчас рассмотрим подробнее первый случай. При тестировании функциональности программы применяется подход "черного ящика", то есть для каждого требования к системе формируются тест-требования, которые, как правило, детализируют функциональные требования так, что на одно функциональное требование может приходиться несколько тест-требований. Сами тест-требования определяют, что должно быть протестировано, но не определяют, как. Конкретные значения задаются в тестовых примерах. Таким образом, одному тест-требованию может соответствовать сразу несколько тестовых примеров.

Каждый тестовый пример состоит из набора входных значений и набора ожидаемых выходных значений. Рассмотрим спецификацию из "Место верификации среди процессов разработки программного обеспечения".

Начнем мы с тестирования отдельных составляющих программы (в данном случае – модуль математика) на допустимые данные и, в частности, на допустимые граничные данные.

Рассмотрим пример.

Требование: Для чисел, меньших либо равных MAXINT и больших либо равных MININT, частное которых меньше либо равно MAXINT и больше либо равно MININT и делитель не равен 0, функция деления должна возвращать правильное частное с точки зрения математики.

Функция, которую будем тестировать:

```
/// <summary>
/// частное
/// </summary>
/// <param name="a">делимое</param>
/// <param name="b">делитель</param>
/// <returns>частное</returns>
public static long Div(long a, long b)
```

В принципе, тестирование такой функции легко автоматизируется при помощи Unit Testing, так как у нее уровень доступа public, т.е. к ней можно обратиться из любого класса. К тому же она является статической, что позволит вызывать ее, не создавая экземпляр класса CalcClass. Однако на этом семинаре мы рассмотрим ручное тестирование.

Прежде всего по этому функциональному требованию составим тест-требования. На первый взгляд, очевидно, что вопрос для проверки звучит так: "Проверить, что для чисел, меньших либо равных MAXINT и больших либо равных MININT, частное которых меньше либо равно MAXINT и больше либо равно MININT и делитель не равен 0, функция деления возвращает правильное частное с точки зрения математики". Однако, это не совсем так. Фраза "меньших либо равных" сразу же наводит на мысль о проверке двух случаев – 1) хотя бы одно из чисел строго равно MAXINT и 2) все числа меньше, чем MAXINT.

Замечание. Стоит заметить, что сейчас мы пишем очень подробные тест-требования, которые практически сразу можно отобразить в тестовые примеры. Такая ситуация наблюдается, например, в проектах, в которых тест-требования отсутствуют, а тестовые примеры пишутся сразу на основании функциональных требований.

Тест-требования

Проверить, что для чисел, меньших MAXINT и больших 0, функция деления возвращает правильное частное с точки зрения математики.

Проверить, что для делимого, меньшего MAXINT и большего 0, и делителя, меньшего 0 и большего MININT, функция деления возвращает правильное частное с точки зрения математики.

Проверить, что для делимого, меньшего 0 и большего чем MININT, и делителя, большего 0 и меньшего MAXINT, функция деления возвращает правильное частное с точки зрения математики.

Проверить, что для чисел, меньших 0 и больших MININT, функция деления возвращает правильное частное с точки зрения математики.

Проверить, что для делимого, равного 0, и делителя, меньшего MAXINT и большего 0, функция деления возвращает правильное частное с точки зрения математики.

Проверить, что для делимого, равного 0, и делителя, большего MININT и меньшего 0, функция деления возвращает правильное частное с точки зрения математики.

Проверить, что для делимого, равного MAXINT, и делителя, меньшего MAXINT и большего MININT, функция деления возвращает правильное частное с точки зрения математики.

Проверить, что для делимого, равного MININT, и делителя, меньшего MAXINT и большего MININT, функция деления возвращает правильное частное с точки зрения математики.

Проверить, что для делителя, равного MAXINT, и делимого, меньшего MAXINT и большего MININT, функция деления возвращает правильное частное с точки зрения математики.

Проверить, что для делителя, равного MININT, и делимого, меньшего MAXINT и большего MININT, функция деления возвращает правильное частное с точки зрения математики.

Проверить, что для делителя, равного MAXINT, и делимого, равного MININT, функция деления возвращает правильное частное с точки зрения математики.

Проверить, что для делителя, равного MAXINT, и делимого, равного MAXINT, функция деления возвращает правильное частное с точки зрения математики.

Проверить, что для делителя, равного MININT, и делимого, равного MININT, функция деления возвращает правильное частное с точки зрения математики.

Проверить, что для делителя, равного MININT, и делимого, равного MAXINT, функция деления возвращает правильное частное с точки зрения математики.

Замечание: на самом деле, еще можно проверить, что функция корректно работает с однозначными и многозначными числами. К тому же надо убедиться, что она правильно обрабатывает все цифры в числах, а, например, не генерирует исключение, если одно из чисел содержит цифру 9, или результат ее работы не зависит от того, делится ли одно число на другое нацело или нет.

Составим тестовые примеры и запишем их в виде таблицы.

Таблица 4

| № | Входные значения: делимое, делитель | Ожидаемый результат | Номер тест- требования | Примечания |
|-----|----------------------------------------|------------------------|---------------------------|--------------------------------------------------------------------|
| 1) | 43 / 21 | 2 | 1) | Самый частый случай – корректные входные данные |
| 2) | 87/-56 | -1 | 2) | -//- |
| 3) | -9/2 | -4 | 3) | -//- |
| 4) | -4321/-50 | 86 | 4) | -//- |
| 5) | 0/1234567890 | 0 | 5) | Часто ошибки проявляются при нулевых значениях переменных |
| 6) | 0/-1098765432 | 0 | 6) | -//- |
| 7) | 2147483647/95 | 22605091 | 7) и 1) | Проверка граничных условий |
| 8) | -2147483648/9 | -238609294 | 8) и 2) | -//- |
| 9) | 99/2147483647 | 0 | 9) и 1) | -//- |
| 10) | -87/-2147483648 | 0 | 10) и 4) | -//- |
| 11) | -2147483648/2147483647 | -1 | 11) | А здесь - ошибка |
| 12) | 2147483647/2147483647 | 1 | 12) | -//- |
| 13) | -2147483648/-2147483648 | 1 | 13) | Ошибка |
| 14) | 2147483647/-2147483648 | 0 | 14) | Ошибка |

Примечание. Здесь стоит так или иначе симитировать работу функции и исследовать ее функциональность вместе со студентами.

Заметим, что, мы не можем точно сказать, где произошла ошибка, — это хорошо видно из рис. 23

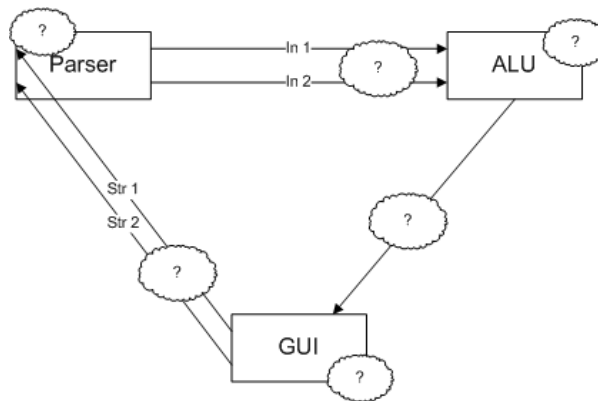


Рисунок 24

Части рисунка, помеченные вопросом – те части программы, в которых могла быть ошибка. Это сами вычисления, преобразование типов, ошибка GUI или передачи параметров (возможно, с некорректным преобразованием типов). О выявленных ошибках составляется отчет, который отдается разработчикам. При этом исправление ошибки в обязанности тестировщика не входит, так как он занимается именно тестированием, а не отладкой приложения.

В первом тесте мы ввели два числа и получили верный результат. Все остальные тесты точно такие же. Но перебрать их все не получится, так как всевозможных комбинаций – $429496736 * 429496735 = 184467445805156960$. Очевидно, что большинство входных значений приведут к одному и тому же результату, и нет смысла проверять их все. Если программа пройдет первый тест, то она, вероятнее всего, пройдет и остальные.

Если от двух тестовых примеров ожидается получить один и тот же результат, значит, они принадлежат одному классу. Такие множества примеров называются классами эквивалентности. Классы эквивалентности — это, в первую очередь, способ уменьшения необходимого числа тестовых примеров. При тестировании достаточно выполнить только один тестовый пример для каждого класса эквивалентности. Разбиение на классы эквивалентности особенно полезно, когда на вход системы может быть подано большое количество различных значений; тестирование каждого возможного значения привело бы к слишком большому объему тестирования.

Классы эквивалентности

Рассмотрим другой пример.

Требование: Для чисел, каждое из которых меньше либо равно MAXINT и больше либо равно MININT, функция суммирования должна возвращать правильную сумму с точки зрения математики.

Функция, которую будем тестировать:

```

/// <summary>
/// Сложение
/// </summary>
/// <param name="a">слагаемое</param>
/// <param name="b">слагаемое</param>
/// <returns>сумма</returns>
public static long Add(long a, long b)

```

По сравнению с предыдущим требованием у этого явно есть недостатки. Здесь ничего не говорится об ограничениях на сумму. Можно легко подобрать два таких числа, которые будут удовлетворять заявленному требованию, а их сумма будет выходить за границы int. Скорее всего, это ошибка проектирования программы. Тогда необходимо исправить спецификацию и сообщить об этом остальным участникам разработки, прежде всего ее составителю.

После исправлений функциональное требование будет выглядеть так:

Требование: Для чисел, меньших либо равных MAXINT и больших либо равных MININT, сумма которых меньше либо равна MAXINT и больше либо равна MININT, функция суммирования должна возвращать правильную сумму с точки зрения математики.

Тестирование на допустимые данные ничем не будет отличаться от тестирования функции деления. Составим классы эквивалентности.

В простейшем случае любое из слагаемых делится на 3 класса эквивалентности: MININT, MAXINT и промежуточное значение. Если подходить более серьезно, то можно выделить 7 допустимых классов эквивалентности: MININT, MININT+1, отрицательное не граничное число, 0, положительное не граничное число, MAXINT-1, MAXINT.

Учитывая то, что у нас два идентичных входных параметра, для полного рассмотрения всех классов эквивалентности необходимо составить и проверить $7*7=49$ тестовых примеров, что все равно гораздо меньше, чем полный перебор.

При этом, как показал тест с делением, ошибка может проявиться лишь в нескольких из этих примеров, которые не сильно отличаются от остальных граничных классов эквивалентности.

Некоторые классы эквивалентности не удовлетворяют требованию так как выводят сумму за допустимые пределы. Поведение метода на таких входных данных описано в требовании

На рис. 24 показано возможное выделение классов эквивалентности (цветами изображены области корректных и некорректных значений, а кружками — сами классы эквивалентности):



Рисунок 25

Иногда удобнее составить классы эквивалентности по выходному параметру (в данном случае их будет 7) и уже по ним подбирать входные данные и составлять тестовые примеры.

Основной способ поиска дефектов – передача системе данных, не предусмотренных требованиями: слишком длинных или слишком коротких строк, неверных символов, чисел за пределами вычислимого диапазона и т.п. Неверные данные, как и допустимые, также можно разделять на различные классы эквивалентности. В качестве простого примера снова рассмотрим функцию сложения.

Замечание. Как уже отмечалось выше, тест-требования составлены очень подробно и, фактически, соответствуют тестовым примерам. Поэтому поведение метода на некорректных данных описано в спецификации, хотя подобная ситуация в жизни редко встречается.

В требовании для чисел, сумма которых больше чем MAXINT и меньше чем MININT, а также в случае, если любое из слагаемых больше чем MAXINT или меньше чем MININT, программа должна выдавать ошибку Error 06

Интерфейс метода Add не позволяет нам узнать об ошибке, произошедшей во время выполнения метода. Среда .NET предоставляет мощное средство для отлавливания и обработки ошибок (и не только) во время выполнения программы – Exception (исключение). Именно с использованием исключений и будут работать методы класса CalcClass, но так как знакомство с ними произойдет позднее, то сейчас воспользуемся другим методом – создадим в классе математических функций глобальную статическую переменную типа string lastError. В нее будем записывать коды ошибок, произошедшие во время работы программы, а в сами методы вставим код, выводящий на экран сообщение об ошибке.

Замечание. Описание класса Exception и его применение можно прочитать по адресу <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfssystemexceptionclasstopic.asp>

О перехвате и обработке исключений — по адресу <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconexceptionhandlingfundamentals.asp>

Составим тест-требования.

1. Если одно из слагаемых больше, чем MAXINT, то функция должна выдать сообщение "Слишком малое или слишком большое значение числа для int. Числа должны быть в пределах от -2147483648 до 2147483647 " и записать в переменную lastError "Error 06".

2. Если одно из слагаемых меньше, чем MININT, то функция должна выдать сообщение "Слишком малое или слишком большое значение числа для int. Числа должны быть в пределах от -2147483648 до 2147483647 " и записать в переменную lastError "Error 06".

3. Если сумма слагаемых больше, чем MAXINT, то функция должна выдать сообщение "Слишком малое или слишком большое значение числа для int. Числа должны быть в пределах от -2147483648 до 2147483647" и записать в переменную lastError "Error 06".

4. Если сумма слагаемых меньше, чем MININT, то функция должна выдать сообщение "Слишком малое или слишком большое значение числа для int. Числа должны быть в пределах от -2147483648 до 2147483647 " и записать в переменную lastError "Error 06".

При составлении тестовых примеров этим тест-требованиям будет соответствовать более четырех примеров, так как необходимо проверить случаи, когда одно число больше MAXINT, а другое удовлетворяет требованиям или больше MAXINT и так далее. В то же время некоторые тестовые примеры могут покрывать сразу несколько тест-требований. Так, пример "Если первое слагаемое больше MAXINT, а второе слагаемое меньше MININT, при этом сумма чисел больше MAXINT, то функция возвращает сообщение "Слишком малое или слишком большое значение числа для int. Числа должны быть в пределах от -2147483648 до 2147483647 " и записывает в переменную lastError "Error 06" проверяет сразу первое, второе и третье тест-требование.

Для недопустимых данных также можно составить классы эквивалентности, причем как по входным, так и по выходным параметрам, и по ним подобрать тестовые примеры. В нашем случае на каждую переменную можно выделить 4 класса: много меньше MININT, MININT-1, MAXINT+1, много больше MAXINT. Таким образом, надо проверить 16 тестовых примеров.

На самом деле, взглянув на любую из рассмотренных функций, в общем случае выделяют 4 основных класса эквивалентности (рис. 26).

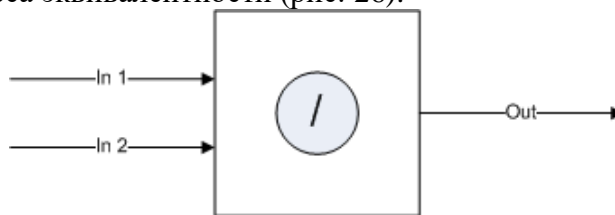


Рисунок 26

1. Оба входных параметра принадлежат допустимой области, и выходное значение принадлежит допустимой области.

2. Первый входной параметр принадлежит допустимой области, второй не принадлежит допустимой области

3. Первый входной параметр не принадлежит допустимой области, второй принадлежит допустимой области

4. Оба входных параметра принадлежат допустимой области, а значение функции не принадлежит допустимой области.

Это тот минимум, на котором и надо протестировать метод. Однако интуиция и опыт тестировщика подсказывают, что эти классы можно разбить на более мелкие подклассы, в которых часто возникают ошибки. Так, первый класс для функции нахождения частного мы разбили на 14 подклассов, в результате чего и обнаружили ошибку.

Помимо рассмотренных классов тестовых примеров, направленных на выявление различных дефектов в работе программной системы, выделяют также тестовые примеры инициализации системы, повторного ввода данных, устойчивости системы и другие.


Возможности MVSTE по ручному тестированию и описанию тестовых примеров (Manual Testing)

Замечание. Подробнее о ручном тестировании можно почитать по адресу [http://msdn2.microsoft.com/en-us/library/ms182615\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/ms182615(VS.80).aspx)

Некоторые тестовые примеры не могут быть выполнены в автоматическом режиме, слишком сложны для автоматизации их выполнения или их автоматическое выполнение потребует слишком много времени, и поэтому они требуют ручной работы тестировщика. MVSTE имеет инструмент для работы с ручными тестами.

Ручное тестирование в MVSTE представляет собой сценарий выполнения теста.

Рассмотрим процесс создания ручного теста.

Сначала создадим новый тестовый проект. Для этого зайдём в File->New->Project... (можно также нажать Ctrl+Shift+N или нажать на иконке  на панели Standart)

В появившемся диалоговом окне New Project выберем тип проекта Visual C#->Test->Test Project (если язык C# не выбран по умолчанию в MVSTE, то выберите Test Project->Test Documents-> Test Project). В поле Name зададим имя нашего проекта(например, ManualTestProject). Нажмем ОК. (рис. 26)

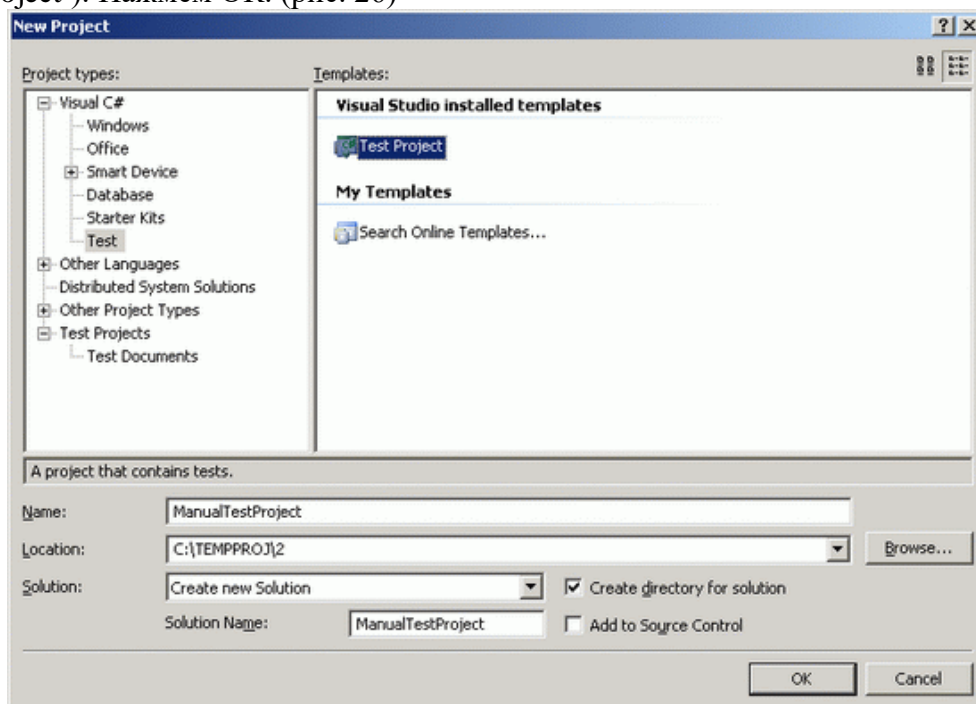


Рисунок 27

Новый тестовый проект создан.

Теперь посмотрим на окно Solution Explorer (рис. 28). Созданный тестовый проект содержит три файла, связанных с тестированием:

Таблица 5

| | |
|-------------------|----------------------------------------------------------------------------------------------------|
| AuthoringTest.txt | Примечания о создании тестов, включающие инструкции по добавлению дополнительных тестов к проекту. |
| UnitTest1.cs | Пустая структура unit test класса, куда помещаются дополнительные тесты. |
| ManualTest1.mht | Шаблон в формате Word, который заполняется инструкциями при ручном тестировании. |

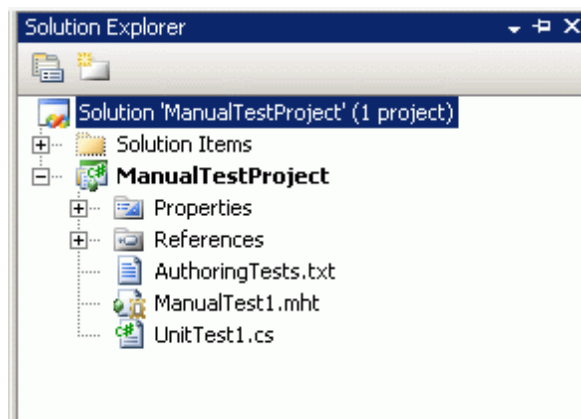


Рисунок 28

Замечание. Файл UnitTest1.cs нам не понадобится для ручного тестирования, поэтому его можно удалить из проекта. Для этого в Solution Explorer щелкнем по нему правой кнопкой мыши и нажмем в появившемся контекстном меню Delete. В появившемся окошке подтверждаем удаление, нажав ОК.

Замечание. Если на вашем компьютере не установлен Microsoft Office, то файл ManualTest1.mht не будет создан. Для добавления в проект шаблона для ручного тестирования нужно в меню Test выбрать New Test. В появившемся диалоговом окне Add New Test выбрать Manual Test(text format). В поле Test Name нужно ввести название теста, например ManualTest1.mtx. Ни в коем случае нельзя менять разрешение этого файла. В поле Add to Test Project выберем созданный нами ранее ManualTestProject. Нажмем ОК. (рис. 28) В наш тестовый проект будет добавлен файл с ручным тестом ManualTest1.mtx.

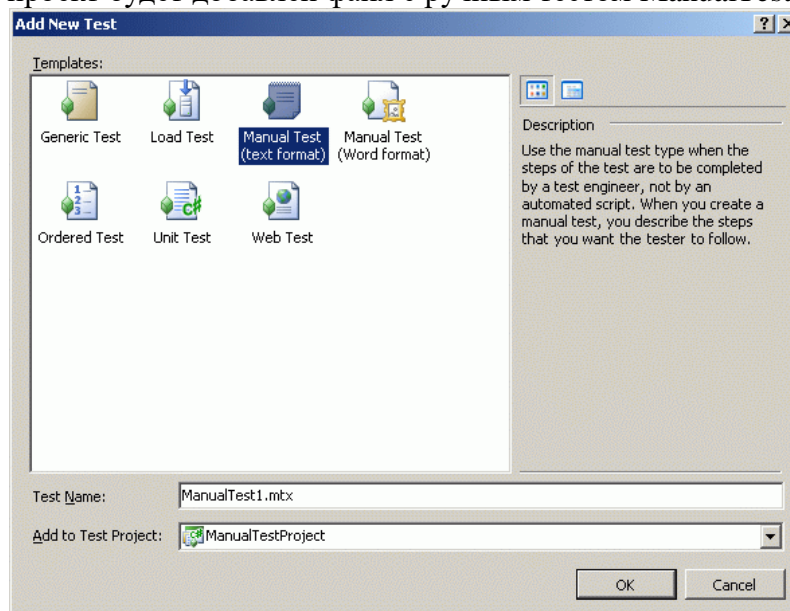


Рисунок 29

Теперь убедимся, что ручной тест добавлен и готов к выполнению. В меню Test нажмем на пункт Windows и в открывшемся подменю выберем Test View. Откроется окно Test View, в котором виден тест MyManualTest (рис. 30).

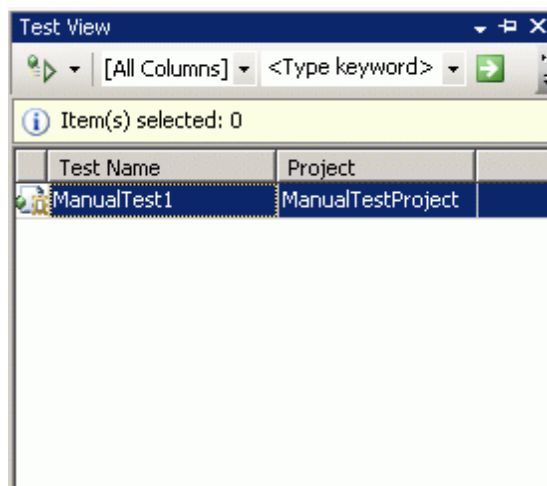



Рисунок 30


Новый ручной тест добавлен, и теперь все готово к его редактированию. Необходимо открыть шаблон теста (например, сделав двойной клик мышкой по ManualTest1.mht(ManualTest1.mtx) в Solution Explorer). Далее, следуя инструкции, вводим информацию о тесте в шаблон. Шаблон включает в себя название теста (Test Title), описание теста (Test Details), функциональность, которую надо проверить (Test Target), действия, которые необходимо совершить для проведения теста (Test Steps), и описание истории изменений теста (Revision History). После завершения редактирования необходимо сохранить шаблон.

Следующий этап – выполнение теста тестировщиком.

В окне Test View нажмем правой кнопкой мыши по созданному нами ручному тесту (ManualTest1) и выберем Run Selection (или нажмем в окне Test View на кнопку ).

Появится диалоговое окно, которое предупреждает о том, что тест будет выполнен, когда все ручные тесты будут пройдены. Нажимаем ОК. Через некоторое время появится диалоговое окно, сообщающее нам о том, что все ручные тесты готовы к выполнению. Опять нажимаем ОК.

Откроется окно Test Results, в котором наш тест будет помечен как Pending (выполняется), и окно MyManualTest[Running], начинающее выполнение теста. Следуя сценарию теста и оставляя свои комментарии в верхней части окна, тестировщик выполняет тест, после чего указывает, пройден тест или нет (Pass/Fail), и нажимает Apply в верхней части экрана. В окне Test Results отобразятся изменения, то есть вместо Pending будет Passed или Failed (в зависимости от того, что вы указали в окне MyManualTest[Running] после выполнения ручного теста).

Замечание. Результаты выполнения тестов можно экспортировать в отдельный файл. Для этого в окне Test Result надо нажать на кнопку Export Test Run Results , указать имя и местоположение файла.

Раздаточный материал

Программа

Будут выданы .exe и .dll файлы, которые нужно протестировать (тестирование черного ящика).

Шаблон отчета о проблеме

Задание

Составить тест-требования и провести ручное тестирование следующих методов:

Нахождение остатка

/// <summary>

/// Деление по модулю

/// </summary>

/// <param name="a">делимое</param>

/// <param name="b">делитель</param>

/// <returns>остаток</returns>

```
public static int Mod(long a, long b)
```

Унарный плюс

```
/// <summary>
```

```
/// унарный плюс
```

```
/// </summary>
```

```
/// <param name="a"></param>
```

```
/// <returns></returns>
```

```
public static int ABS(long a)
```

Унарный минус

```
/// <summary>
```

```
/// унарный минус
```

```
/// </summary>
```

```
/// <param name="a"></param>
```

```
/// <returns></returns>
```

```
public static int IABS(long a)
```

Вычитание

```
/// <summary>
```

```
/// вычитание
```

```
/// </summary>
```

```
/// <param name="a">уменьшаемое</param>
```

```
/// <param name="b">вычитаемое</param>
```

```
/// <returns>разность</returns>
```

```
public static int Sub(long a, long b)
```

По результатам ручного тестирования заполнить отчет о проблеме.

Практическая работа № 2.15. Тестовое окружение

Цель работы: выполнение тестового окружения

Тест

В каждом тестовом задании может быть несколько вариантов ответа. После проведения теста студенты могут попробовать обосновать свои неверные ответы.

1. В основные обязанности тестировщика входят:

1. Выявление ошибки
2. Исправление ошибки
3. Составление отчета об ошибке
4. Объяснение причины ошибки
5. Написание тестов

Ответ: 1, 3, 5

2. Одному тест-требованию может соответствовать:

1. только один тестовый пример
2. несколько тестовых примеров
3. не более двух тестовых примеров

Ответ: 2

3. Два тестовых примера проверяют один и тот же класс эквивалентности:

1. если от них получен один и тот же результат
2. если от них получена одинаковая реакция системы
3. если они построены по одному тест-требованию
4. если от них ожидается получить одинаковую реакцию системы

Ответ: 2, 4

4. Сколько классов эквивалентности в общем случае выделяют для функции с двумя целочисленными входными параметрами и одним целочисленным выходным значением?

1. 9
2. 14

3. 4
4. 7
5. 11

Ответ: 3

5. Ручное тестирование целесообразно применять:

1. если тестовый пример не может быть выполнен в автоматическом режиме
2. если тестовый пример построен по одному тест-требованию
3. если автоматизация выполнения тестового примера очень сложна
4. если автоматическое выполнение тестового примера требует много времени

Ответ: 1, 3, 4

Тестовое окружение

Основной объем тестирования практически любой сложной системы обычно выполняется в автоматическом режиме. Кроме того, тестируемая система обычно разбивается на отдельные модули, каждый из которых тестируется вначале отдельно от других, затем в комплексе.

Это означает, что для выполнения тестирования необходимо создать некоторую среду, которая обеспечит запуск и выполнение тестируемого модуля, передаст ему входные данные, соберет реальные выходные данные, полученные в результате работы системы на заданных входных данных (рис. 31). После этого среда должна сравнить реальные выходные данные с ожидаемыми и на основании данного сравнения сделать вывод о соответствии поведения модуля заданному.

Тестовое окружение также может использоваться для отчуждения отдельных модулей системы от всей системы. Разделение модулей системы на ранних этапах тестирования позволяет более точно локализовать проблемы, возникающие в их программном коде. Для поддержки работы модуля в отрыве от системы тестовое окружение должно моделировать поведение всех модулей, к функциям или данным которых обращается тестируемый модуль.

Поскольку тестовое окружение само является программой (причем, часто реализованной не на том языке программирования, на котором написана система), оно тоже должно быть протестировано. Целью тестирования тестового окружения является доказательство того, что тестовое окружение никак не искажает выполнение тестируемого модуля и адекватно моделирует поведение системы.

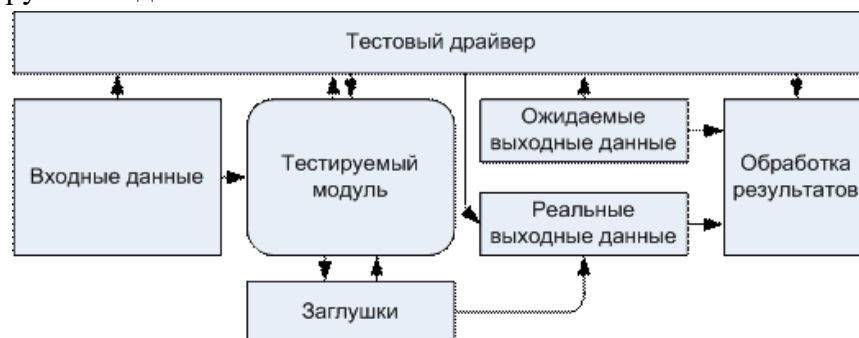


Рисунок 31

Тестовое окружение для программного кода на структурных языках программирования состоит из двух компонентов – драйвера, который обеспечивает запуск и выполнение тестируемого модуля, и заглушек, которые моделируют функции, вызываемые из данного модуля. Разработка тестового драйвера представляет собой отдельную задачу тестирования, сам драйвер должен быть протестирован, дабы исключить неверное тестирование. Драйвер и заглушки могут иметь различные уровни сложности, требуемый уровень сложности выбирается в зависимости от сложности тестируемого модуля и уровня тестирования. Так, драйвер может выполнять следующие функции:

1. Вызов тестируемого модуля
2. 1 + передача в тестируемый модуль входных значений и прием результатов
3. 2 + вывод выходных значений
4. 3 + протоколирование процесса тестирования и ключевых точек программы

Заглушки могут выполнять следующие функции:

1. Не производить никаких действий (такие заглушки нужны для корректной сборки тестируемого модуля и
2. Выводить сообщения о том, что заглушка была вызвана
3. 1 + выводиться сообщения со значениями параметров, переданных в функцию
4. 2 + возвращать значение, заранее заданное во входных параметрах теста
5. 3 + выводиться значение, заранее заданное во входных параметрах теста
6. 3 + принимать от тестируемого ПО значения и передавать их в драйвер

Тестовое окружение для объектно-ориентированного ПО выполняет те же самые функции, что и для структурных программ (на процедурных языках). Однако, оно имеет некоторые особенности, связанные с применением наследования и инкапсуляции.

Если при тестировании структурных программ минимальным тестируемым объектом является функция, то в объектно-ориентированном ПО минимальным объектом является класс. При применении принципа инкапсуляции все внутренние данные класса и некоторая часть его методов недоступна извне. В этом случае тестировщик лишен возможности обращаться в своих тестах к данным класса и произвольным образом вызывать методы; единственное, что ему доступно – вызывать методы внешнего интерфейса класса.

Существует несколько подходов к тестированию классов, каждый из них накладывает свои ограничения на структуру драйвера и заглушек.

1. Драйвер создает один или больше объектов тестируемого класса, все обращения к объектам происходят только с использованием их внешнего интерфейса. Текст драйвера в этом случае представляет собой т.н. тестирующий класс, который содержит по одному методу для каждого тестового примера. Процесс тестирования заключается в последовательном вызове этих методов. Вместо заглушек в состав тестового окружения входит программный код реальной системы, соответственно, отсутствует изоляция тестируемого класса. Именно такой подход к тестированию принят сейчас в большинстве методологий и сред разработки. Его классическое название – unit testing (тестирование модулей).

2. Аналогично предыдущему подходу, но для всех классов, которые использует тестируемый класс, создаются заглушки

3. Программный код тестируемого класса модифицируется таким образом, чтобы открыть доступ ко всем его свойствам и методам. Строение тестового окружения в этом случае полностью аналогично окружению для тестирования структурных программ.

4. Используются специальные средства доступа к закрытым данным и методам класса на уровне объектного или исполняемого кода – скрипты отладчика или accessors в Visual Studio.

Основное достоинство первых двух методов – при их использовании класс работает точно таким же образом, как в реальной системе. Однако в этом случае нельзя гарантировать того, что в процессе тестирования будет выполнен весь программный код класса и не останется протестированных методов.

Основной недостаток 3-го метода – после изменения исходных текстов тестируемого модуля нельзя дать гарантии того, что класс будет вести себя таким же образом, как и исходный. В частности, это связано с тем, что изменение защиты данных класса влияет на наследование данных и методов другими классами.

Тестирование наследования – отдельная сложная задача в объектно-ориентированных системах. После того, как протестирован базовый класс, необходимо тестировать классы-потомки. Однако, для базового класса нельзя создавать заглушки, т.к. в этом случае можно пропустить возможные проблемы полиморфизма. Если класс-потомок использует методы базового класса для обработки собственных данных, необходимо убедиться в том, что эти методы работают.

Таким образом, иерархия классов может тестироваться сверху вниз, начиная от базового класса. Тестовое окружение при этом может меняться для каждой тестируемой конфигурации классов.

На примере "Калькулятора"

Тесты, сделанные нами на прошлой неделе, как правило, выполняются не вручную. Для целей тестирования пишется специальная программа — тестовый драйвер, который и выполняет тестирование. Более того, такие программы часто пишутся на другом языке, нежели тестируемая программа, или создаются автоматически, с помощью специальных утилит.

На этом семинаре мы сами напишем простой тестовый драйвер на C# для тестирования функций "Калькулятора", используя спецификацию второго семинара.

Замечание. Код программы слегка изменен для упрощения компиляции отдельных модулей. Так, исключена работа с переменной Program.res, а класс CalcClass объявлен как public.

Для начала рассмотрим функцию деления. Тест-требования к ней мы уже составили. Для простоты будем пользоваться лишь четырьмя общими тест-требованиями.

1. Оба входных параметра принадлежат допустимой области, и выходное значение принадлежит допустимой области.
2. Первый входной параметр принадлежит допустимой области, второй не принадлежит допустимой области
3. Первый входной параметр не принадлежит допустимой области, второй принадлежит допустимой области
4. Оба входных параметров принадлежат допустимой области, а значение функции не принадлежит допустимой области.

Составим программу:

```
private void buttonStartDel_Click(object sender, EventArgs e)
{
    try
    {
        richTextBox1.Text = "";
        richTextBox1.Text += "Test Case 1\n";
        richTextBox1.Text += "Входные данные: a= 78508, b = -304\n";
        richTextBox1.Text += "Ожидаемый результат: res = 78204 &&"
            error = "\"\""+ "\n";
        int res = CalcClass.Add(78508, -304);
        string error = CalcClass.lastError;
        richTextBox1.Text += "Код ошибки: " + error + "\n";
        richTextBox1.Text += "Получившийся результат: " + "res = " +
            res.ToString() + " error = " + error.ToString() + "\n";
        if (res == 78204 && error == "")
        {
            richTextBox1.Text += "Тест пройден\n\n";
        }
        else
        {
            richTextBox1.Text += "Тест не пройден\n\n";
        }
    }
    catch (Exception ex)
    {
        richTextBox1.Text += "Перехвачено исключение: " +
            ex.ToString() + "\nТест не пройден.\n";
    }

    try
    {
        richTextBox1.Text += "Test Case 2\n";
    }
}
```

```

richTextBox1.Text += "Входные данные: a= -2850800078, b =
                        30000000000\n";
richTextBox1.Text += "Ожидаемый результат: res = 0 && error =
                        \nError 06\n";
int res = CalcClass.Add(-2850800078, 30000000000);
string error = CalcClass.lastError;
richTextBox1.Text += "Код ошибки: " + error + "\n";
richTextBox1.Text += "Получившийся результат: " + "res = " +
                        res.ToString() + " error = " + error.ToString() + "\n";
if (res == 0 && error == "Error 06")
{
    richTextBox1.Text += "Тест пройден\n\n";
}
else
{
    richTextBox1.Text += "Тест не пройден\n\n";
}
}
catch (Exception ex)
{
    richTextBox1.Text += "Перехвачено исключение: " +
                        ex.ToString() + "\nТест не пройден.\n";
}

try
{
    richTextBox1.Text += "Test Case 3\n";
richTextBox1.Text += "Входные данные: a= 30000000000, b = -
                        2850800078\n";
richTextBox1.Text += "Ожидаемый результат: res = 0 && error =
                        \nError 06\n";
int res = CalcClass.Add(30000000000, -2850800078);
string error = CalcClass.lastError;
richTextBox1.Text += "Код ошибки: " + error + "\n";
richTextBox1.Text += "Получившийся результат: " + "res = " +
                        res.ToString() + " error = " + error.ToString() + "\n";
if (res == 0 && error == "Error 06")
{
    richTextBox1.Text += "Тест пройден\n\n";
}
else
{
    richTextBox1.Text += "Тест не пройден\n\n";
}
}
catch (Exception ex)
{
    richTextBox1.Text += "Перехвачено исключение: " +
                        ex.ToString() + "\nТест не пройден.\n";
}

try
{

```

```

richTextBox1.Text += "Test Case 4\n";
richTextBox1.Text += "Входные данные: a= 2000000000, b =
                        2000000000\n";
richTextBox1.Text += "Ожидаемый результат: res = 0 && error =
                        \"Error 06\"\n";
int res = CalcClass.Add(2000000000, 2000000000);
string error = CalcClass.lastError;
richTextBox1.Text += "Код ошибки: " + error + "\n";
richTextBox1.Text += "Получившийся результат: " + "res = " +
                        res.ToString() + " error = " + error.ToString() + "\n";
if (res == 0 && error == "Error 06")
{
    richTextBox1.Text += "Тест пройден\n\n";
}
else
{
    richTextBox1.Text += "Тест не пройден\n\n";
}
}
catch (Exception ex)
{
    richTextBox1.Text += "Перехвачено исключение: " +
                        ex.ToString() + "\nТест не пройден.\n";
}
}

```

Листинг. Текст программы

Каждый тестовый пример находится внутри блока try-catch для того, чтобы перехватить любое сгенерированное исключение внутри методов Add().

При этом файл CalcClass.dll, в котором и реализованы все математические методы, необходимо добавить в References проекта.

Проведем тестирование и получим следующий результат:

Test Case 1

Входные данные: a= 78508, b = -304

Ожидаемый результат: res = 78204 && error = ""

Код ошибки:

Получившийся результат: res = 78204 error =

Тест пройден

Test Case 2

Входные данные: a= -2850800078, b = 3000000000

Ожидаемый результат: res = 0 && error = "Error 06"

Код ошибки: Error 06

Получившийся результат: res = 0 error = Error 06

Тест пройден

Test Case 3

Входные данные: a= 3000000000, b = -2850800078

Ожидаемый результат: res = 0 && error = "Error 06"

Код ошибки: Error 06

Получившийся результат: res = 0 error = Error 06

Тест пройден

Test Case 4

Входные данные: a= 2000000000, b = 2000000000

Ожидаемый результат: res = 0 && error = "Error 06"

Код ошибки: Error 06

Получившийся результат: res = 0 error = Error 06

Тест пройден

Точно такой же результат мы бы получили и при ручном тестировании, если бы выявленные ошибки были исправлены. Заметим, что при таком подходе к тестированию нам удастся локализовать ошибки. Если что-то работает не так, как надо, то можно с уверенностью утверждать, что ошибка содержится именно в функции деления, в то время, как на прошлом семинаре мы не могли сказать, где именно она произошла.

Замечание. Мы считаем, что тестовый драйвер сам не содержит ошибок. Тестирование тестового драйвера выходит за пределы изучаемой темы.

Раздаточный материал

Программа

Будут выданы .dll файлы, которые нужно протестировать методом "черного ящика", и пример тестового драйвера.

Составить тест-план и провести модульное тестирование следующих методов:

Нахождение остатка.

```
/// <summary>
    /// Деление по модулю
    /// </summary>
    /// <param name="a">делимое</param>
    /// <param name="b">делитель</param>
    /// <returns>остаток</returns>
    public static int Mod(long a, long b)
```

Унарный плюс.

```
/// <summary>
    /// унарный плюс
    /// </summary>
    /// <param name="a"></param>
    /// <returns></returns>
    public static int ABS(long a)
```

Унарный минус.

```
/// <summary>
    /// унарный минус
    /// </summary>
    /// <param name="a"></param>
    /// <returns></returns>
    public static int IABS(long a)
```

Вычитание.

```
/// <summary>
    /// вычитание
    /// </summary>
    /// <param name="a">уменьшаемое</param>
    /// <param name="b">вычитаемое</param>
    /// <returns>разность</returns>
    public static int Sub(long a, long b)
```

Умножение.

```
/// <summary>
    /// умножение
    /// </summary>
    /// <param name="a">множитель</param>
    /// <param name="b">множитель</param>
    /// <returns>произведение</returns>
    public static int Mult(long a, long b)
```



```

Деление.
/// <summary>
    /// частное
    /// </summary>
    /// <param name="a">делимое</param>
    /// <param name="b">делитель</param>
    /// <returns>частное</returns>
    public static int Div(long a, long b)

```

Практическая работа № 2.16. Модульное тестирование. Тестирование классов

Цель работы: Изучение основ модульного тестирования, тестирование классов

Модульное тестирование

Задачи и цели модульного тестирования

Каждая сложная программная система состоит из отдельных частей – модулей, выполняющих ту или иную функцию в составе системы. Для того, чтобы удостовериться в корректной работе системы в целом, необходимо вначале протестировать каждый модуль системы в отдельности. В случае возникновения проблем это позволит проще выявить модули, вызвавшие проблему, и устранить соответствующие дефекты в них. Такое тестирование модулей по отдельности получило название модульного тестирования (unit testing).

Для каждого модуля, подвергаемого тестированию, разрабатывается тестовое окружение, включающее в себя драйвер и заглушки, готовятся тест-требования и тест-планы, описывающие конкретные тестовые примеры.

Основная цель модульного тестирования – удостовериться в соответствии требованиям каждого отдельного модуля системы перед тем, как будет произведена его интеграция в состав системы.

При этом в ходе модульного тестирования решаются четыре основные задачи.

1. Поиск и документирование несоответствий требованиям – это классическая задача тестирования, включающая в себя не только разработку тестового окружения и тестовых примеров, но и выполнение тестов, протоколирование результатов выполнения, составление отчетов о проблемах.

2. Поддержка разработки и рефакторинга низкоуровневой архитектуры системы и межмодульного взаимодействия – эта задача больше свойственна "легким" методологиям типа XP, где применяется принцип тестирования перед разработкой (Test-driven development), при котором основным источником требований для программного модуля является тест, написанный до самого модуля. Однако, даже при классической схеме тестирования модульные тесты могут выявить проблемы в дизайне системы и нелогичные или запутанные механизмы работы с модулем.

3. Поддержка рефакторинга модулей – эта задача связана с поддержкой процесса изменения системы. Достаточно часто в ходе разработки требуется проводить рефакторинг модулей или их групп – оптимизацию или полную переделку программного кода с целью повышения его сопровождаемости, скорости работы или надежности. Модульные тесты при этом являются мощным инструментом для проверки того, что новый вариант программного кода работает в точности так же, как и старый.

4. Поддержка устранения дефектов и отладки — эта задача сопряжена с обратной связью, которую получают разработчики от тестирующих в виде отчетов о проблемах. Подробные отчеты о проблемах, составленные на этапе модульного тестирования, позволяют локализовать и устранить многие дефекты в программной системе на ранних стадиях ее разработки или разработки ее новой функциональности.

В силу того, что модули, подвергаемые тестированию, обычно невелики по размеру, модульное тестирование считается наиболее простым (хотя и достаточно трудоемким) этапом тестирования системы. Однако, несмотря на внешнюю простоту, с модульным тестированием связано две проблемы.

1. Не существует единых принципов определения того, что в точности является отдельным модулем.

2. Различия в трактовке самого понятия модульного тестирования – понимается ли под ним обособленное тестирование модуля, работа которого поддерживается только тестовым окружением, или речь идет о проверке корректности работы модуля в составе уже разработанной системы. В последнее время термин "модульное тестирование" чаще используется во втором смысле, хотя в этом случае речь скорее идет об интеграционном тестировании.

Эти две проблемы рассмотрены в двух следующих разделах.

Понятие модуля и его границ. Тестирование классов

Традиционное определение модуля с точки зрения его тестирования: "модуль – это компонент минимального размера, который может быть независимо протестирован в ходе верификации программной системы". В реальности часто возникают проблемы с тем, что считать модулем. Существует несколько подходов к данному вопросу:

- модуль – это часть программного кода, выполняющая одну функцию с точки зрения функциональных требований;
- модуль – это программный модуль, т.е. минимальный компилируемый элемент программной системы;
- модуль – это задача в списке задач проекта (с точки зрения его менеджера);
- модуль – это участок кода, который может уместиться на одном экране или одном листе бумаги;
- модуль – это один класс или их множество с единым интерфейсом.

Обычно за тестируемый модуль принимается либо программный модуль (единица компиляции) в случае, если система разрабатывается на процедурном языке программирования, либо класс, если система разрабатывается на объектно-ориентированном языке.

В случае систем, написанных на процедурных языках, процесс тестирования модуля происходит так, как это было рассмотрено в темах 2-4 – для каждого модуля разрабатывается тестовый драйвер, вызывающий функции модуля и собирающий результаты их работы, и набор заглушек, которые имитируют поведение функций, содержащихся в других модулях и не попадающих под тестирование данного модуля. При тестировании объектно-ориентированных систем существует ряд особенностей, прежде всего вызванных инкапсуляцией данных и методов в классах.

В случае объектно-ориентированных систем более мелкое деление классов и использование отдельных методов в качестве тестируемых модулей нецелесообразно в связи с тем, что для тестирования каждого метода потребуется разработка тестового окружения, сравнимого по сложности с уже написанным программным кодом класса. Кроме того, декомпозиция класса нарушает принцип инкапсуляции, согласно которому объекты каждого класса должны вести себя как единое целое с точки зрения других объектов.

Процесс тестирования классов как модулей иногда называют компонентным тестированием. В ходе такого тестирования проверяется взаимодействие методов внутри класса и правильность доступа методов к внутренним данным класса. При таком тестировании возможно обнаружение не только стандартных дефектов, связанных с выходами за границы диапазона или неверно реализованными требованиями, а также обнаружение специфических дефектов объектно-ориентированного программного обеспечения:

- дефектов инкапсуляции, в результате которых, например, скрытые данные класса оказываются недоступными при помощи соответствующих публичных методов;
- дефектов наследования, при наличии которых схема наследования блокирует важные данные или методы от классов-потомков;
- дефектов полиморфизма, при которых полиморфное поведение класса оказывается распространенным не на все возможные классы;

- дефектов инстанцирования, при которых во вновь создаваемых объектах класса не устанавливаются корректные значения по умолчанию параметров и внутренних данных класса.

Однако, выбор класса в качестве тестируемого модуля имеет и ряд сопряженных проблем.

Определение степени полноты тестирования класса. В том случае, если в качестве тестируемого модуля выбран класс, не совсем ясно, как определять степень полноты его тестирования. С одной стороны, можно использовать классический критерий полноты покрытия программного кода тестами: если полностью выполнены все структурные элементы всех методов, как публичных, так и скрытых, — то тесты можно считать полными.

Однако существует альтернативный подход к тестированию класса, согласно которому все публичные методы должны предоставлять пользователю данного класса согласованную схему работы и достаточно проверить типичные корректные и некорректные сценарии работы с данным классом. Т.е., например, в классе, объекты которого представляют записи в телефонной книжке, одним из типичных сценариев работы будет "Создать запись, искать запись и найти ее, удалить запись, искать запись вторично и получить сообщение об ошибке".

Различия в этих двух методах напоминают различия между тестированием "черного" и "белого" ящиков, но на самом деле второй подход отличается от "черного ящика" тем, что функциональные требования к системе могут быть составлены на уровне более высоком, чем отдельные классы, и установление адекватности тестовых сценариев требованиям остается на откуп тестирующему.

Протоколирование состояний объектов и их изменений. Некоторые методы класса предназначены не для выдачи информации пользователю, а для изменения внутренних данных объекта класса. Значение внутренних данных объекта определяет его состояние в каждый отдельный момент времени, а вызов методов, изменяющих данные, изменяет и состояние объекта. При тестировании классов необходимо проверять, что класс адекватно реагирует на внешние вызовы в любом из состояний. Однако, зачастую из-за инкапсуляции данных невозможно определить внутреннее состояние класса программными способами внутри драйвера.

В этом случае может помочь составление схемы поведения объекта как конечного автомата с определенным набором состояний (подобно тому, как это было описано в теме 2 в разделе "Генераторы сигналов. Событийно-управляемый код"). Такая схема может входить в низкоуровневую проектную документацию (например, в составе описания архитектуры системы), а может составляться тестирующим или разработчиком на основе функциональных требований к системе. В последнем случае для определения всех возможных состояний может потребоваться ручной анализ программного кода и определение его соответствия требованиям. Автоматизированное тестирование в этом случае может лишь определить, по всем ли выявленным состояниям осуществлялись переходы и все ли возможные реакции проверялись.

Тестирование изменений. Как уже упоминалось выше, модульные тесты – мощный инструмент проверки корректности изменений, внесенных в исходный код при рефакторинге. Однако, в результате рефакторинга только одного класса, как правило, не меняется его внешний интерфейс с другими классами (интерфейсы меняются при рефакторинге сразу нескольких классов). В результате обычных эволюционных изменений системы у класса может меняться внешний интерфейс, причем как по формальным (изменяются имена и состав методов, их параметры), так и по функциональным признакам (при сохранении внешнего интерфейса меняется логика работы методов). Для проведения модульного тестирования класса после таких изменений потребуется изменение драйвера и, возможно, заглушек. Но только модульного тестирования в данном случае недостаточно, необходимо также проводить и интеграционное тестирование данного класса вместе со всеми классами, которые связаны с ним по данным или по управлению.

Вне зависимости от того, на какие модули, подвергаемые тестированию, разбивается система, рекомендуется изложить принципы выделения тестируемых модулей в плане и стратегии тестирования, а также составить на базе структурной схемы архитектуры системы новую структурную схему, на которой отметить все тестируемые модули. Это позволит спрогнозировать состав и сложность драйверов и заглушек, требуемых для модульного тестирования системы. Такая схема также может использоваться позже на этапе модульного тестирования для выделения укрупненных групп модулей, подвергаемых интеграции.

Подходы к проектированию тестового окружения

Вне зависимости от того, какая минимальная единица исходных кодов системы выбирается за минимальный тестируемый модуль, существует еще одно различие в подходах к модульному тестированию.

Первый подход к модульному тестированию основывается на предположении, что функциональность каждого вновь разработанного модуля должна проверяться в автономном режиме без его интеграции с системой. Здесь для каждого вновь разрабатываемого модуля создается тестовый драйвер и заглушки, при помощи которых выполняется набор тестов. Только после устранения всех дефектов в автономном режиме производится интеграция модуля в систему и проводится тестирование на следующем уровне. Достоинством данного подхода является более простая локализация ошибок в модуле, поскольку при автономном тестировании исключается влияние остальных частей системы, которое может вызывать маскировку дефектов (эффект четного числа ошибок). Основным недостатком данного метода – повышенная трудоемкость написания драйверов и заглушек, поскольку заглушки должны адекватно моделировать поведение системы в различных ситуациях, а драйвер должен не только создавать тестовое окружение, но и имитировать внутреннее состояние системы, в составе которой должен функционировать модуль.

Второй подход построен на предположении, что модуль все равно работает в составе системы и если модули интегрировать в систему по одному, то можно протестировать поведение модуля в составе всей системы. Этот подход свойственен большинству современных "облегченных" методологий разработки, в том числе и XP.

В результате применения такого подхода резко сокращаются трудозатраты на разработку заглушек и драйверов – в роли заглушек выступает уже оттестированная часть системы, а драйвер выполняет только функции передачи и приема данных, не моделируя внутреннее состояние системы.

Тем не менее, при использовании данного метода возрастает сложность написания тестовых примеров – для приведения в нужное состояние системы заглушек, как правило, требуется только установить значения тестовых переменных, а для приведения в нужное состояние части реальной системы необходимо выполнить целый сценарий. Каждый тестовый пример в этом случае должен содержать такой сценарий.

Кроме того, при этом подходе не всегда удастся локализовать ошибки, скрытые внутри модуля, которые могут проявиться при интеграции следующих модулей.

Замечание. О том, что такое Reflection, можно прочесть на [http://msdn2.microsoft.com/en-us/library/cxz4wk15\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/cxz4wk15(VS.80).aspx)

На примере "Калькулятора"

Рассмотренный на предыдущем семинаре пример прост прежде всего за счет того, что нам не приходится создавать тестового окружения. Чтобы увидеть весь описанный механизм в действии, протестируем метод RunEstimate класса AnalizerClass. Этот метод использует методы из класса CalcClass, в надежности которых мы не уверены. Заменим эти методы заглушкой, состоящей исключительно из функций стандартного класса Math. Для этого воспользуемся файлом My.dll и добавим его в проект.

На семинаре мы не будем составлять тест-требования (этим студенты займутся в домашней работе). Продемонстрируем, как создать тестовое окружение и запустить метод. Проверяем операцию сложения на примере 2+2, т.е. в стеке до начала выполнения самой операции (т.е. после компиляции) находятся следующие элементы: " 2 ", " 2 ", " + ".

```
private void buttonStart_Click(object sender, EventArgs e)
```

```

    {
        // создаем провайдер для генерирования и компиляции кода на C#
        System.CodeDom.Compiler.CodeDomProvider prov =
System.CodeDom.Compiler.CodeDomProvider.CreateProvider("CSharp");
        // создаем параметры компилирования
        System.CodeDom.Compiler.CompilerParameters cmpparam = new
System.CodeDom.Compiler.CompilerParameters();
        // результат компиляции - библиотека
        cmpparam.GenerateExecutable = false;
        // не включаем информацию отладчика
        cmpparam.IncludeDebugInformation = false;
        // подключаем 2-е стандартные библиотеки и библиотеку
CalcClass.dll
        cmpparam.ReferencedAssemblies.Add(Application.StartupPath +
"\\CalcClass.dll");
        cmpparam.ReferencedAssemblies.Add("System.dll");
        cmpparam.ReferencedAssemblies.Add("System.Windows.Forms.dll");
        // имя выходной сборки - My.dll
        cmpparam.OutputAssembly = "My.dll";
        // компилируем класс AnalazerClass с заданными параметрами
        System.CodeDom.Compiler.CompilerResults res =
        prov.CompileAssemblyFromFile(cmpparam, Application.StartupPath +
"\\AnalazerClass.cs");
        // Выводим результат компилирования на экран
        if (res.Errors.Count != 0)
        {
            richTextBox1.Text += res.Errors[0].ToString();
        }
        else
        {
            // загружаем только что скомпилированную сборку(здесь тонкий
момент - если мы прото загрузим сборку из файла, то он будет заблокирован,
            // acces denied, поэтому вначале читаем его в поток и лишь
потом подключаем)
            System.IO.BinaryReader reader = new
System.IO.BinaryReader(new System.IO.FileStream(Application.StartupPath + "\\My.dll",
System.IO.FileMode.Open, System.IO.FileAccess.Read));
            Byte[] asmBytes = new Byte[reader.BaseStream.Length];
            reader.Read(asmBytes, 0, (Int32) reader.BaseStream.Length);
            reader.Close();
            reader = null;
            System.Reflection.Assembly assm =
System.Reflection.Assembly.Load(asmBytes);
            Type[] types = assm.GetTypes();
            Type analazer = types[0];
            // находим метод CheckCurrency - к счастью, он единственный
            System.Reflection.MethodInfo addinfo =
analazer.GetMethod("RunEstimate");
            System.Reflection.FieldInfo fieldopz =
analazer.GetField("opz");
            System.Collections.ArrayList ar = new
System.Collections.ArrayList();
            ar.Add("2");

```

```

        ar.Add("2");
        ar.Add("+");
        fieldopz.SetValue(null, ar);
        richTextBox1.Text += addinfo.Invoke(null, null).ToString();
        asmBytes = null;
    }
    prov.Dispose();
}

```

Замечание. На самом деле данный подход позволяет выявить множество недостатков программы, которые другими тестами не выявляются. Можно попробовать поэкспериментировать с "Калькулятором" и убедиться, что он работает корректно. Однако, если в тестируемый метод подать на вход не " 2 ", " 2 ", " + ", а " 2 ", " 2 ", " + ", " + ", то программа закончит работу с исключением. Это говорит о том, что метод RunEstimate написан не корректно. Можно, например, было бы скрыть, т. е. сделать доступ private, всем методам AnalaizerClass, кроме Estimate (это было бы более правильно, но для простоты тестирования они сделаны public. Стоит отметить, что Visual Studio 2005 имеет также механизмы для тестирования подобных методов.). Тем самым мы не позволим другим выполнять "потенциально опасные" методы и передавать им некорректные значения. Однако это не является достаточным механизмом защиты программы. Необходимо провести более качественную валидацию используемых методами параметров.

Замечание. К проблеме создания тестового окружения можно подойти с двух сторон – либо откомпилировать код, с заранее подключенными dll файлами к проекту, либо воспользоваться областью CodeDom и компилировать в процессе выполнения. Это особенно удобно, если нужно менять тестовое окружение в процессе работы.

Программа

Будут выданы .dll файлы, которые нужно протестировать методом "черного ящика" и пример тестового драйвера.

Составить тест-план и провести модульное тестирование следующих методов:

```

/// <summary>
/// Проверка корректности скобочной структуры входного выражения
/// </summary>

```

```

/// <returns>true - если все нормально,
false - если есть ошибка</returns>

```

```

/// метод бежит по входному выражению, символ за
символом анализируя его и считая количество скобок.

```

В случае возникновения

```

/// ошибки возвращает false, а в erposition записывает позицию,
на которой возникла ошибка.

```

```

public static bool CheckCurrency()

```

```

/// <summary>

```

```

/// Форматирует входное выражение, выставляя между
операторами пробелы и удаляя лишние, а также отлавливает
неопознанные операторы, следит за концом строки

```

```

/// а также отлавливает ошибки на конце строки

```

```

/// </summary>

```

```

/// <returns>конечную строку или сообщение об ошибке,
начинающиеся со спец. символа &</returns>

```

```

public static string Format()

```

```

/// <summary>

```

```

/// Создает массив, в котором располагаются операторы и
символы, представленные в обратной польской записи (безскобочной)

```

```

/// На этом же этапе отлавливаются почти все остальные
ошибки (см код). По сути - это компиляция.

```

```

/// </summary>
/// <returns>массив обратной польской записи</returns>
public static System.Collections.ArrayList CreateStack()
/// <summary>
/// Вычисление обратной польской записи
/// </summary>
/// <returns>результат вычислений или сообщение об ошибке</returns>
public static string RunEstimate()

```

Практическая работа № 2.17. Автоматизация модульного тестирования

Цель работы: выполнение автоматизации модульного тестирования

Тест

В каждом тестовом задании может быть несколько вариантов ответа. После проведения теста, студенты могут попробовать обосновать свои неверные ответы.

1. Тестовое окружение может использоваться для:
 1. запуска и выполнения тестируемого модуля
 2. передачи входных данных
 3. сбора ожидаемых выходных данных
 4. сравнения реальных выходных данных с ожидаемыми
 5. поддержки отчуждения отдельных модулей системы от всей системы

Ответ: 1, 2, 4, 5

2. Тестовое окружение для программного кода на структурных языках программирования состоит из:

1. драйвера
2. тестов
3. заглушек
4. исходного кода

Ответ: 1, 3

3. Модульное тестирование проводится для того, чтобы:
 1. удостовериться в корректной работе системы в целом
 2. удостовериться в корректной работе набора модулей
 3. удостовериться в корректной работе отдельного модуля

Ответ: 3

4. Модуль – это (с точки зрения наших семинарских занятий):

1. часть программного кода, выполняющая одну функцию с точки зрения функциональных требований
2. программный модуль, т.е. минимальный компилируемый элемент программной системы
3. задача в списке задач проекта
4. участок кода, который может уместиться на одном экране или одном листе бумаги
5. один класс или их множество с единым интерфейсом.

Ответ: 2

5. Какие основные задачи решаются в ходе модульного тестирования?

1. Поиск и документирование несоответствий требованиям
2. Поддержка разработки и рефакторинга низкоуровневой архитектуры системы и межмодульного взаимодействия
3. Рефакторинг модулей
4. Поддержка рефакторинга модулей
5. Отладка
6. Поддержка устранения дефектов и отладки

Ответ: 1, 2, 4, 6

Возможности MVSTE по автоматизации модульного тестирования

Замечание. Подробнее о модульном тестировании можно почитать по адресу [http://msdn2.microsoft.com/en-us/library/ms182515\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/ms182515(VS.80).aspx)

До сих пор мы выполняли часть работы вручную. Но при написании тестов тестировщик также может ошибиться, из-за чего в программе могут остаться различные ошибки. В случае, если программисты ведут разработку по методике экстремального программирования (XP), следуя практике написания тестов перед кодом (test driven development, TDD), количество тестов, которые нужно написать, становится по объему даже большим, чем сам код системы. Однако очевидно, что большую часть работы по разработке тестов отдельных методов (модульное тестирование, unit testing) можно автоматизировать. В MVSTE разработаны специальные средства для автоматизации модульного тестирования. Именно о них и пойдет речь дальше.

Начало работы

К моменту написания тестов мы уже имеем полностью готовый код. Можем приступить к созданию тестов.

Создание тестов

Для создания теста нажимаем правой кнопкой мыши на методе Add() и выбирая пункт меню Create Unit Tests.... Появится диалоговое окно, позволяющее создать тесты в другом проекте. По умолчанию, создаваемый проект — новый проект на Visual Basic, но также доступны тестовые проекты на C# и C++. Выбираем Visual C# и нажимаем кнопку ОК, перед тем введя имя проекта BaseCalculator.Test.

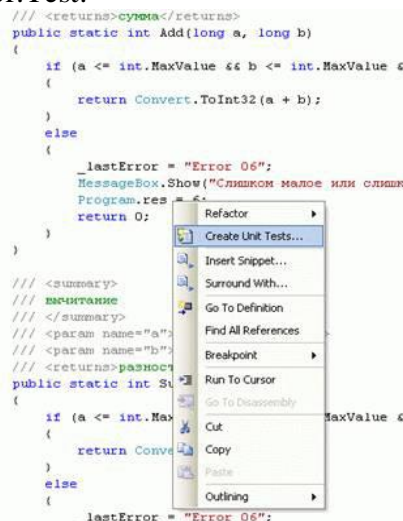


Рисунок 32

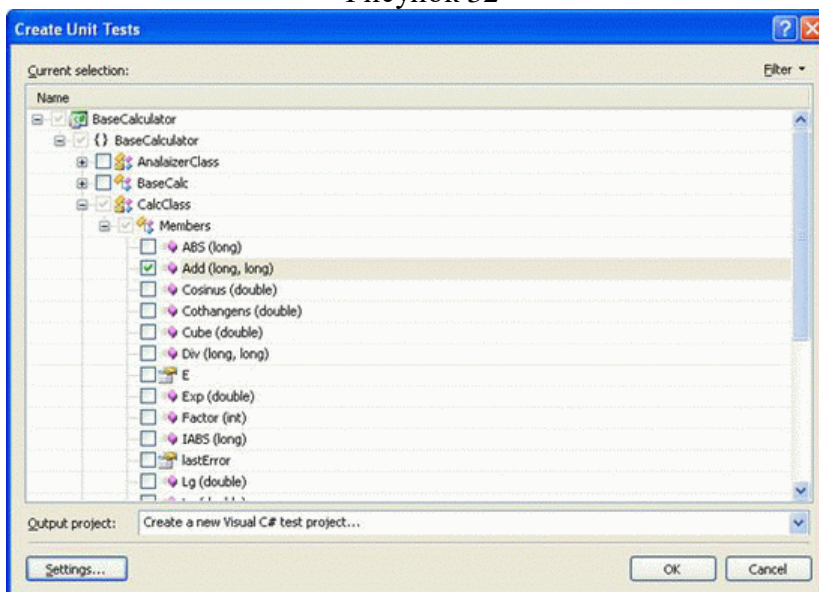


Рисунок 33

Созданный тестовый проект содержит четыре файла, связанных с тестированием.

Таблица 6

| Имя файла | Примечание |
|-------------------|----------------------------------------------------------------------------------------------------------------------------|
| AuthoringTest.txt | Примечания о создании тестов, включающие инструкции по добавлению дополнительных тестов к проекту |
| CalcClassTest.cs | Включает в себя сгенерированный тест для тестирования метода Add () наряду с методами для тестовой инициализации и очистки |
| ManualTest1.mht | Шаблон, который заполняется инструкциями при ручном тестировании |
| UnitTest1.cs | Пустая структура unit test класса, куда помещаются дополнительные тесты |

Так как ручное тестирование мы уже провели, и файл для тестов у нас уже есть, то мы удалим ManualTest1.mht и UnitTest1.cs.

В раздел References при генерации тестового проекта добавляется ссылка на Microsoft.VisualStudio.QualityTools.UnitTestFramework и проект BaseCalculator, который и будет тестироваться. Первое – сборка, которую использует "движок" модульного тестирования при выполнении тестов. Второе — это ссылка на ту сборку, которую мы тестируем.

По умолчанию, сгенерированный тест-метод – это шаблон со следующей реализацией:

```
/// <summary>
/// A test for Add (long, long)
/// </summary>
[DeploymentItem("BaseCalculator.exe")]
[TestMethod()]
public void AddTest()
{
    long a = 0; // TODO: Initialize to an appropriate value

    long b = 0; // TODO: Initialize to an appropriate value

    int expected = 0;
    int actual;

    actual = BaseCalculator.Test.
        BaseCalculator_CalcClassAccessor.Add(a, b);

    Assert.AreEqual(expected, actual,
        "BaseCalculator.CalcClass.Add did not return
        the expected value.");
    Assert.Inconclusive("Verify the correctness of this test method.");
}
```

Замечание. Сгенерированный код теста будет сильно зависеть от типа и сигнатуры того метода, который планируется тестировать. Например, мастер сгенерирует код, основанный на технологии reflection ("отражение"), для тестирования private функций. В нашем конкретном случае это не потребовалось, так как метод Add() объявлен как public().

Прежде всего, отметим, что сгенерированный код помечен атрибутом TestMethod типа TestMethodAttribute, а сам класс помечен атрибутом TestClassAttribute, которые объявлены в Microsoft.VisualStudio.QualityTools.UnitTesting.Framework. При помощи технологии Reflection движок модульного тестирования находит все тестовые классы в проекте, помеченные соответствующим атрибутом, а внутри все необходимые для тестирования методы.

Замечание. Об атрибутах можно почитать подробнее по адресу [http://msdn2.microsoft.com/en-us/library/system.attribute\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/system.attribute(VS.80).aspx)

В начале теста объявляется значение всех необходимых переменных, а также ожидаемое выходное значение. Затем происходит вызов нужного метода, которому передаются необходимые параметры. В нашем случае это

```
actual = BaseCalculator.Test.BaseCalculator_CalcClassAccessor.Add(a, b);
```

Затем идет вызов двух методов класса Assert. Прежде всего рассмотрим второй метод. Assert.Inconclusive("Verify the correctness of this test method.");

Наличие этого метода в тесте говорит о том, что реализация теста еще не закончена.

Сделаем реализацию нашего метода:

```
/// <summary>
/// A test for Add (long, long)
/// </summary>
[DeploymentItem("BaseCalculator.exe")]
[TestMethod()]
public void AddTest()
{
    long a = 150;

    long b = 350;

    int expected = 500;
    int actual;

    actual = BaseCalculator.
        Test.BaseCalculator_CalcClassAccessor.Add(a, b);

    Assert.AreEqual(expected, actual,
        "BaseCalculator.CalcClass.
        Add did not return the expected value.");
}
```

Создание тестов

Чтобы запустить все тесты в рамках проекта, необходимо просто запустить тестовый проект. Один из возможных способов сделать это — кликнуть правой кнопкой мыши на проекте BaseCalculator.Test в Solution explorer и выбрать Set as StartUp Project. Затем используем пункты меню Debug->Start (F5) или Debug->Start Without Debugging (Ctrl+F5), чтобы начать запуск тестов.

В окне Test Results будет показан список со всеми тестами проекта. В момент начала выполнения теста в нашем проекте содержалось два теста: один полностью реализованный тест AddTest, второй – неоконченный AddTest1. В момент запуска оба теста будут в состоянии "неоконченный" (Pending), но как только тесты будут выполнены, появятся результаты выполнения Passed и Inconcluseve, которые мы и ожидали.

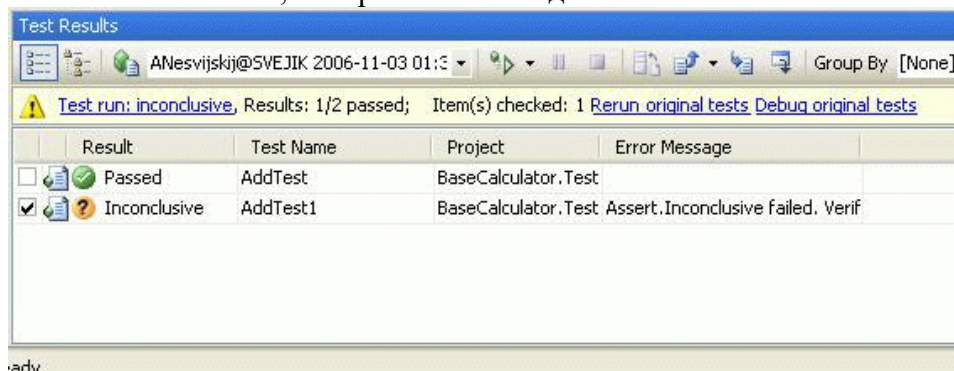


Рисунок 34

Замечание. Рис. 32 показывает окно Test Results. На этом скриншоте в дополнение к колонкам по умолчанию изображена колонка Error Message. Колонки могут быть добавлены

или удалены правым щелчком мыши по меню на заголовках колонки и выборе пункта меню Add/Remove Columns... .

Чтобы посмотреть дополнительные детали о тесте, мы можем дважды щелкнуть на нем в окне Test Results и открыть окно AddTest[Result]. В нем можно узнать информацию о скорости выполнения теста, его результате, возникшей ошибке и прочее.

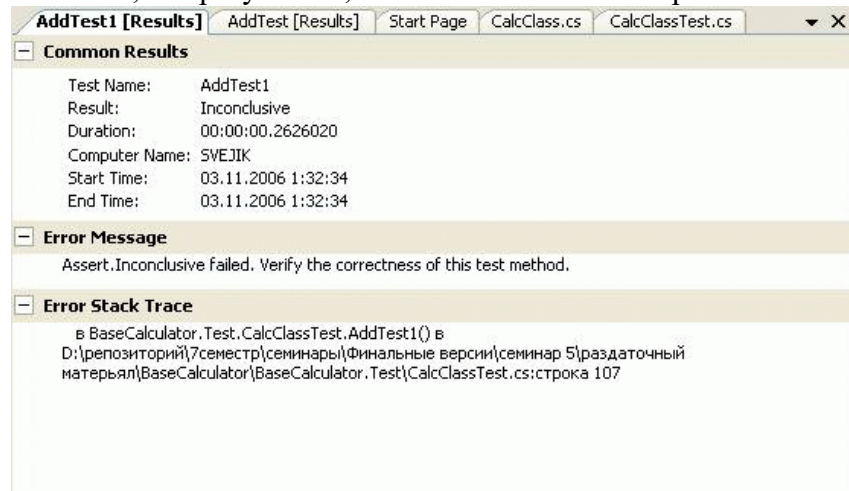


Рисунок 35

Кроме того, мы можем кликнуть правой кнопкой мыши на отдельных тестах и выбрать пункт меню Open Test, чтобы переместиться на код теста.

Обработка исключений

На прошлом семинаре мы обнаружили, что метод RunEstimate() класса AnalizerClass не достаточно хорошо проверяет объекты, с которыми он работает. Если инициализировать список opz значением {2,2,+,+}, то выполнение метода RunEstimate() приводит к генерации исключения. Действительно, реализуем тест:

```
/// <summary>
/// A test for RunEstimate ()
/// </summary>
[DeploymentItem("BaseCalculator.exe")]
[TestMethod()]
public void RunEstimateTest()
{
    string expected = null;
    string actual;
    // Подготовка тестового окружения
    BaseCalculator.Test.BaseCalculator_CalcClassAccessor._lastError = "";
    BaseCalculator.Test.BaseCalculator_AnalizerClassAccessor.opz =
        new System.Collections.ArrayList();
    BaseCalculator.Test.BaseCalculator_AnalizerClassAccessor.opz.Add("2");
    BaseCalculator.Test.BaseCalculator_AnalizerClassAccessor.opz.Add("2");
    BaseCalculator.Test.BaseCalculator_AnalizerClassAccessor.opz.Add("+");
    BaseCalculator.Test.BaseCalculator_AnalizerClassAccessor.opz.Add("+");

    actual = BaseCalculator.Test.BaseCalculator_AnalizerClassAccessor.RunEstimate();

    Assert.AreEqual(expected, actual,
        "BaseCalculator.AnalizerClass.RunEstimate did not return the expected value.");
}
```

Замечание. Для работы этого теста необходимо создать начальное тестовое окружение, при этом значение _lastError необходимо очистить, так как оно будет "испорчено" тестом AddTest1(). Подробнее о зависимости тестов от порядка выполнения и тестового окружения мы поговорим на девятом семинаре.

Несмотря на то, что явных блоков try-catch не стоит, сгенерированное исключение не приведет к прекращению работы тестов, а будет корректно обработано. В этом можно убедиться, заглянув в окно на RunEstimateTest[Result].



Рисунок 36

Предположим теперь, что при неверных входных параметрах метод RunEstimate() действительно должен генерировать исключение, которое будет перехватываться в другом месте. Создадим еще один тест:

```
/// <summary>
/// A test for RunEstimate ()
/// </summary>
[DeploymentItem("BaseCalculator.exe")]
[TestMethod()]
[ExpectedException(typeof(ArgumentOutOfRangeException),
    "Была обработана неверная синтаксическая конструкция")]
public void RunEstimateTest1()
{
    BaseCalculator.Test.BaseCalculator_CalcClassAccessor._lastError = "";
    BaseCalculator.Test.BaseCalculator_AnalizaizerClassAccessor.opz =
        new System.Collections.ArrayList();
    BaseCalculator.Test.BaseCalculator_AnalizaizerClassAccessor.opz.Add("2");
    BaseCalculator.Test.BaseCalculator_AnalizaizerClassAccessor.opz.Add("2");
    BaseCalculator.Test.BaseCalculator_AnalizaizerClassAccessor.opz.Add("+");
    BaseCalculator.Test.BaseCalculator_AnalizaizerClassAccessor.opz.Add("+");

    BaseCalculator.Test.BaseCalculator_AnalizaizerClassAccessor.RunEstimate();
}
```

Отметим, что, опять же, нет блока try-catch с явным тестом на ArgumentOutOfRangeException. Вместо этого тест включает дополнительный атрибут, ExpectedException, который принимает тип параметра, и произвольное сообщение об ошибке, которое будет показано, если исключение не было брошено. Когда тесты выполняются, среда будет явно следить за тем, чтобы исключение ArgumentException было сгенерировано, и если метод не будет генерировать такое исключение, то тест будет провален.

Программа

Будут выданы исходные файлы модулей для тестирования методом "белого ящика" средствами MVSTE, пример тестового драйвера.

Составить тест-план и провести модульное тестирование (средствами MVSTE) следующих методов:

1. `public static int Mod(long a, long b)`
2. `public static bool CheckCurrency()`
3. `public static int ABS(long a)`
4. `public static string Format()`
5. `public static int IABS(long a)`
6. `public static string Format()`
7. `public static int Sub(long a, long b)`
8. `public static System.Collections.ArrayList CreateStack()`
9. `public static int Mult(long a, long b)`
10. `public static System.Collections.ArrayList CreateStack()`
11. `public static int Div(long a, long b)`
12. `public static bool CheckCurrency()`

Практическая работа № 2.18. Формальные инспекции

Цель работы: Изучение основ формальной инспекции

Не во всех случаях возможна разработка автоматических или хотя бы четко формализованных ручных тестов для проверки функциональности программной системы. В некоторых случаях выполнение программного кода, подвергаемого тестированию, невозможно в условиях, создаваемых тестовым окружением (например, во встроенных системах, если программный код предназначен для обработки исключительных ситуаций, создаваемых только при установке системы на реальное оборудование). В других случаях верифицируется не программный код, а проектная документация на систему, которую нельзя "выполнить" или создать для нее отдельные тестовые примеры. И в тех и в других случаях обычно прибегают к методу экспертных исследований программного кода или документации на корректность или непротиворечивость.

Такие экспертные исследования обычно называют инспекциями или просмотрами. Существует два типа инспекций – неформальные и формальные.

Формальная инспекция является четко управляемым процессом, структура которого обычно четко определяется соответствующим стандартом проекта. Таким образом, все формальные инспекции имеют одинаковую структуру и одинаковые выходные документы, которые затем используются при разработке.

Факт начала формальной инспекции четко фиксируется в общей базе данных проекта. Также фиксируются документы, подвергаемые инспекции, списки замечаний, отслеживаются внесенные по замечаниям изменения. Этим формальная инспекция похожа на автоматизированное тестирование – списки замечаний имеют много общего с отчетами о выполнении тестовых примеров.

В ходе формальной инспекции группой специалистов осуществляется независимая проверка соответствия инспектируемых документов исходным документам. Независимость проверки обеспечивается тем, что она осуществляется инспекторами, не участвовавшими в разработке инспектируемого документа. Входами процесса формальной инспекции являются инспектируемые документы и исходные документы, а выходами – материалы инспекции, включающие список обнаруженных несоответствий и решение об изменении статуса инспектируемых документов. рис. 17.1 иллюстрирует место формальной инспекции в процессе разработки программных систем.



Рисунок 37

Этапы формальной инспекции и роли ее участников

Процесс формальной инспекции состоит из пяти фаз:

- инициализация;
- планирование;
- подготовка (экспертиза);
- обсуждение;
- завершение.

Инициализация

Руководитель проекта (преподаватель) выбирает объект инспекции (код, написанный на КР). Затем он назначает участников формальной инспекции: авторов (студенты, написавшие код: 2 или 3 человека), ведущего (сам преподаватель) и нескольких инспекторов (студенты, не участвовавших в написании инспектируемого кода: 4 или 5 человек). Ведущий также выполняет роль инспектора; остальные участники выполняют только одну роль.

Эта фаза формальной инспекции проводится на семинаре с использованием раздаточного материала.

Планирование

Считаем, что инспектируемые документы размещены в базе данных проекта, а их статус соответствует готовности к формальной инспекции. Считаем, что статус этих документов изменен так, чтобы отметить начало формальной инспекции, и менять их нельзя.

После этого ведущий заносит в бланк инспекции идентификаторы инспектируемых и исходных документов и номера их версий, список участников с указанием их ролей и дату фактического начала процесса инспекции, т.е. того момента, когда инспектируемые документы были переведены в состояние Review.

Подготовив бланк инспекции и определив время и место собрания, ведущий должен известить участников инспекции о времени и месте и разослать им подготовленный бланк инспекции.

Эта фаза формальной инспекции проводится на "Тестирование программного кода (покрытия)" (раздается раздаточный материал).

Подготовка

Получив назначение с прикрепленным к нему бланком инспекции, исходные и инспектируемые документы, инспекторы детально изучают инспектируемые документы, руководствуясь списком контрольных вопросов.

Перед началом просмотра исходного кода рекомендуется отметить пункты требований, на соответствие которым проверяется исходный код, а также записать обоснования того, почему эти требования не могут быть проверены в автоматическом режиме. После этого можно переходить к просмотру собственно исходного кода. Все пометки, которые придется вносить в ходе инспектирования в исходный код, необходимо делать не в файле, который будет выдан инспекторам, а в его копии, которая потом будет подшита к материалам

инспекции. Копия может быть в том же формате, что и исходный файл, либо распечатана на бумаге или выведена в формат DOC, PDF или аналогичный, допускающий комментирование.

Рекомендуется делать пометки, поясняющие, почему именно данный участок кода реализует требования. Такие пометки помогут на этапе собрания.

Рекомендуется также проверять наличие участков, гарантирующих робастность, даже если требования прямо не определяют необходимости обработки недопустимых значений. В случае, если потенциально возможна некорректная работа программы из-за отсутствия обработчиков неверных значений, рекомендуется отметить это в списке замечаний.

Все обнаруженные несоответствия должны быть точно локализованы, сформулированы и записаны.

Эта фаза формальной инспекции проходит в виде домашнего задания семинара 6 студентам, которые назначены инспекторами.

Обсуждение

Обсуждение проводится в форме одного собрания.

В ходе обсуждения ведущий синхронизирует работу участников, перечисляя номера или идентификаторы требований (в нашем случае в ходе инспекции проверяется соответствие исходного кода требованиям). По мере продвижения по документу инспекторы прерывают ведущего в тех местах, к которым у них имеются замечания. В случае отсутствия разногласий ведущий фиксирует несоответствие и продолжает продвижение по документу.

Если мнения участников по высказанному замечанию расходятся, то ведущий управляет дискуссией, последовательно предоставляя слово всем желающим высказаться, причем автор пользуется правом внеочередного предоставления слова. Если в результате дискуссии изменилась формулировка замечания, то ведущий записывает эту новую формулировку, затем зачитывает ее и, если все участники с ней согласны, продолжает продвижение по документу.

Результатом дискуссии может также быть признание отсутствия проблемы. В этом случае ведущий убеждается в том, что все с этим согласны, и продолжает продвижение по документу.

Участники должны стремиться обозначить проблемы, но не искать их решения. Достижение консенсуса по спорным вопросам также не является целью дискуссии. Если имеется расхождение во мнениях, то должны быть зафиксированы все альтернативные мнения. Ведущий должен прервать дискуссию, если оценивает ее как непродуктивную.

Все участники обязаны уважительно относиться к оппонентам, не перебивать говорящего и высказываться тогда, когда ведущий предоставит им слово. Не допускаются параллельные обсуждения узким составом – каждый участник обязан адресовать свои высказывания всему собранию, а не соседу.

Необходимо также избегать критики и оценки квалификации коллег. Целью инспекции является повышение качества inspectируемых документов, а не оценка квалификации автора или других участников инспекции.

В ходе обсуждения необходимо в бланке инспекции проставить ответы на контрольные вопросы и зафиксировать замечания. Для этого ведущий последовательно зачитывает контрольные вопросы. При отсутствии у всех инспекторов замечаний, нарушающих сформулированное в вопросе свойство, против вопроса ставится отметка (галочка) в графе "Да"; в противном случае отметка ставится в графе "Нет", а в графе "Ссылка на несоответствие" перечисляются номера соответствующих несоответствий, записанных в таблице для несоответствий, которая помещена в конце бланка инспекции. Отметка в графе "Неприменимо" ставится только в том случае, когда сформулированное в соответствующем вопросе свойство не может быть оценено для данного объекта инспекции; в этом случае в графе "Ссылка на несоответствие" записывается обоснование невозможности оценить данное свойство.

Для облегчения труда автора inspectируемого документа по исправлению замечаний каждое замечание, признанное на собрании существенным, рекомендуется точно трассировать на строки исходного кода и требований.

В конце обсуждения участники принимают решение о возможности принятия объекта инспекции в имеющейся версии либо о необходимости внесения исправлений и проведения повторной инспекции в полной или сокращенной форме. Объект инспекции может быть принят в имеющейся версии только при отсутствии несоответствий. Решение о проведении повторной инспекции в сокращенной форме принимается только в том случае, если все участники с этим согласны. Если хотя бы один из участников настаивает на полной форме повторной инспекции, то повторная инспекция должна проводиться в полной форме. Мнение ведущего учитывается наравне с мнениями других участников. Принятое решение фиксируется ведущим на бланке инспекции и заверяется подписями всех участников.

Теоретически возможна ситуация, когда автор не согласен ни с одним из зафиксированных замечаний, а инспекторы настаивают, что несоответствия существуют. В таком случае невозможно принять решение об изменении статуса инспектируемых документов, поэтому инспекция должна быть отложена, а решение проблемы вынесено за рамки процесса формальной инспекции.

На бланке инспекции также фиксируется продолжительность собрания и время, затраченное каждым из участников на подготовку.

Эта фаза формальной инспекции проходит на текущем семинаре.

Завершение

В конце собрания, по окончании обсуждения, инспекторы сдают ведущему свои рабочие материалы, которые включают в себя распечатки инспектируемых документов с пометками и бланки инспекции. Ведущий складывает эти материалы в прозрачную папку вместе с экземпляром бланка инспекции, заполненным в ходе обсуждения, причем титульный лист бланка инспекции должен лежать сверху, чтобы можно было по нему идентифицировать папки.

После собрания ведущий изменяет статус инспектируемых документов в соответствии с принятым решением – либо им присваивается статус "Принят", либо "Переработать".

В последнем случае необходима повторная инспекция, вид которой уточняется кратким комментарием (не проводится).

Эта фаза формальной инспекции проходит на текущем семинаре.

Практическая работа № 2.19. Покрытие программного кода

Цель работы: Изучение основ покрытия программного кода

Покрытие программного кода

На этом семинаре познакомимся с одной из оценок качества системы тестов — с ее полнотой, т.е. величиной той части функциональности системы, которая проверяется тестовыми примерами. Полная система тестов позволяет утверждать, что система реализует всю функциональность, указанную в требованиях, и, что еще более важно – не реализует никакой другой функциональности. Степень покрытия программного кода тестами – важный количественный показатель, позволяющий оценить качество системы тестов, а в некоторых случаях – и качество тестируемой программной системы.

Одним из наиболее часто используемых методов определения полноты системы тестов является определение отношения количества тест-требований, для которых существуют тестовые примеры, к общему количеству тест-требований, — т.е. в данном случае речь идет о покрытии тестовыми примерами тест-требований. В качестве единицы измерения степени покрытия здесь выступает процент тест-требований, для которых существуют тестовые примеры, называемый процентом покрытых тест-требований. Покрытие требований позволяет оценить степень полноты системы тестов по отношению к функциональности системы, но не позволяет оценить полноту по отношению к ее программной реализации. Одна и та же функция может быть реализована при помощи совершенно различных алгоритмов, требующих разного подхода к организации тестирования.

Для более детальной оценки полноты системы тестов при тестировании стеклянного ящика анализируется покрытие программного кода, называемое также структурным покрытием.

Во время работы каждого тестового примера выполняется некоторый участок программного кода системы, при выполнении всей системы тестов выполняются все участки программного кода, которые задействует эта система тестов. В случае, если существуют участки программного кода, не выполненные при выполнении системы тестов, система тестов потенциально неполна (т.е. не проверяет всю функциональность системы), либо система содержит участки защитного кода или неиспользуемый код (например, "закладки" или задел на будущее использование системы). Таким образом, отсутствие покрытия каких-либо участков кода является сигналом к переработке тестов или кода (а иногда – и требований).

К анализу покрытия программного кода можно приступать только после полного покрытия требований. Полное покрытие программного кода не гарантирует того, что тесты проверяют все требования к системе. Одна из типичных ошибок начинающего тестировщика – начинать с покрытия кода, забывая про покрытие требований.

Замечание. Необходимо помнить, что разработка тестовых примеров, обеспечивающих полное покрытие тестируемого программного кода, относится к структурному тестированию кода. Перед началом структурного тестирования должно быть полностью закончено функциональное тестирование кода как черного ящика (чем мы и занимались на предыдущих семинарах). Только после этого можно переходить к улучшению покрытия. В идеальном случае при полном покрытии функциональных требований должно получаться 100% покрытие кода. Однако на практике такое происходит только в случае очень простого кода. Причина недопокрытия кода при полном покрытии требований – либо неполнота требований, либо недостаточно полный анализ требований тестировщиком. В первом случае обычно требуется доработка требований, во втором – тест-требований и тест-плана.

Уровни покрытия

По строкам программного кода (Statement Coverage)

Для обеспечения полного покрытия программного кода на данном уровне необходимо, чтобы в результате выполнения тестов каждый оператор был выполнен хотя бы один раз. Перед началом тестирования необходимо выделить переменные, от которых зависит выполнение различных ветвей условий и циклов в коде – управляющие входные переменные. Изменение значений этих переменных будет влиять на то, какие строки кода будут выполняться в различных тестовых примерах.

Пример. В следующем фрагменте кода входными переменными являются `prev` и `ShowMessage`.

```
if (prev == "оператор" || prev == "унарный оператор")
{
    if (ShowMessage)
    {
        MessageBox.Show("Два подряд оператора на " +
            i.ToString() + " символе.");
    }
    else
    {
        Log.Write("Два подряд оператора на " +
            i.ToString() + " символе.")
    }
    Program.res = 4;
    return "&Error 04 at " +
        i.ToString();
}
```

Для того, чтобы полностью покрыть данный код по строкам, т.е. выполнить все строки кода, достаточно двух тестовых примеров:

Таблица 7

| | | |
|------|----------|----------|
| | 1 | 2 |
| prev | оператор | оператор |

| | | |
|-------------|------|-------|
| ShowMessage | true | false |
|-------------|------|-------|

В первом тестовом примере осуществляется вход в первый условный оператор if, затем в первую ветвь второго оператора if. Второй тестовый пример осуществляет аналогичные действия, но для второй ветви второго оператора if. В результате все строки кода оказываются выполненными. Легко увидеть, что, несмотря на полное покрытие по строкам, этот набор тестовых примеров не проверяет всей функциональности (даже не видя требований, логично предположить, что в них должно описываться поведение системы и для значения переменной prev = "оператор", и для значения prev = "унарный оператор").

Также проблемы этого метода покрытия можно увидеть и на примерах других управляющих структур. Например, проблемы возникают при проверке циклов do ... while – при данном уровне покрытия достаточно выполнение цикла только один раз, при этом метод совершенно нечувствителен к логическим операторам || и &&.

Другой особенностью данного метода является зависимость уровня покрытия от структуры программного кода. На практике часто не требуется 100% покрытия программного кода, вместо этого устанавливается допустимый уровень покрытия, например 75%. Проблемы могут возникнуть при покрытии следующего фрагмента программного кода:

```
if (condition)
    MethodA();
else
    MethodB();
```

Если MethodA() содержит 99 операторов, а MethodB() — один оператор, то единственного тестового примера, устанавливающего condition в true, будет достаточно для достижения необходимого уровня покрытия. При этом аналогичный тестовый пример, устанавливающий значение condition в false, даст слишком низкий уровень покрытия.

По веткам условных операторов (Decision Coverage)

Для обеспечения полного покрытия по данному методу каждая точка входа и выхода в программе и во всех ее функциях должна быть выполнена по крайней мере один раз и все логические выражения в программе должны принять каждое из возможных значений хотя бы один раз; таким образом, для покрытия по веткам требуется как минимум два тестовых примера.

Также данный метод называют: branch coverage, all-edges coverage, basis path coverage, DC, C2, decision-decision-path.

В отличие от предыдущего уровня покрытия данный метод учитывает покрытие условных операторов с пустыми ветками.

Пример. Для покрытия предыдущего примера кода по ветвям потребуется уже три тестовых примера. Это связано с тем, что первый условный оператор if имеет неявную ветвь – пустую ветвь else. Для обеспечения покрытия по ветвям необходимо покрывать и пустые ветви.

Таблица 8

| | 1 | 2 | 3 |
|-------------|----------|----------|---------|
| prev | оператор | оператор | операнд |
| ShowMessage | true | false | true |

Первые два тестовых примера аналогичны предыдущему случаю, третий предназначен для покрытия неявной ветви. При этом надо заметить, что значение переменной ShowMessage не играет никакой роли для покрытия – участок кода, использующий эту переменную, просто не выполняется.

Особенность данного уровня покрытия заключается в том, что на нем не учитываются логические выражения, значения компонент которых получаются вызовом методов.

Например, на следующем фрагменте программного кода

```
if (condition1 && (condition2 || Method()))
    statement1;
else
    statement2;
```

полное покрытие по веткам может быть достигнуто при помощи двух тестовых примеров:

Таблица 9

| | 1 | 2 |
|------------|------|------------|
| condition1 | true | false |
| condition2 | true | true/false |

В обоих случаях не происходит вызова метода Method(), хотя покрытие данного участка кода будет полным. Для проверки вызова метода Method() необходимо добавить еще один тестовый пример (который, однако, не улучшает степени покрытия по веткам):

Таблица 10

| | 1 | 2 | 3 |
|------------|------|------------|-------|
| condition1 | true | false | true |
| condition2 | true | true/false | false |

По компонентам логических условий

Для более полного анализа компонент условий в логических операторах существует несколько методов, учитывающих структуру компонент условий и значения, которые они принимают при выполнении тестовых примеров.

Покрытие по условиям (Condition Coverage)

Для обеспечения полного покрытия по данному методу каждая компонента логического условия в результате выполнения тестовых примеров должна принимать все возможные значения, но при этом не требуется, чтобы само логическое условие принимало все возможные значения. Так, например, при тестировании следующего фрагмента

```
if (condition1 || condition2)
```

```
    MethodA();
```

```
else
```

```
    MethodB();
```

для покрытия по условиям потребуется два тестовых примера:

Таблица 11

| | 1 | 2 |
|------------|-------|-------|
| condition1 | true | False |
| condition2 | false | True |

При этом значение логического условия будет принимать значение только true; таким образом, при полном покрытии по условиям не будет достигаться покрытие по веткам.

Покрытие по веткам/условиям (Condition/Decision Coverage)

Данный метод сочетает требования предыдущих двух методов – для обеспечения полного покрытия необходимо, чтобы как логическое условие, так и каждая его компонента приняла все возможные значения.

Для покрытия рассмотренного выше фрагмента с условием

```
(condition1 || condition2)
```

потребуется 2 тестовых примера:

Таблица 12

| | 1 | 2 |
|------------|------|-------|
| condition1 | true | false |
| condition2 | true | false |

Однако, эти два тестовых примера не позволят протестировать правильность логической функции – вместо OR в программном коде могла быть ошибочно записана операция AND.

Покрытие по всем условиям (Multiple Condition Coverage)

Для выявления неверно заданных логических функций был предложен метод покрытия по всем условиям. При данном методе покрытия должны быть проверены все возможные наборы значений компонент логических условий. Т.е. в случае n компонент потребуется 2^n тестовых примеров, каждый из которых проверяет один набор значений. Тесты, необходимые для полного покрытия по данному методу, дают полную таблицу истинности для логического выражения.

Несмотря на очевидную полноту системы тестов, обеспечивающей этот уровень покрытия, данный метод редко применяется на практике в связи с его сложностью и избыточностью.

Еще одним недостатком метода является зависимость количества тестовых примеров от структуры логического выражения. Так, для условий, содержащих одинаковое количество компонент и логических операций:

$a \ \&\& \ b \ \&\& \ (c \ || \ (d \ \&\& \ e))$

$((a \ || \ b) \ \&\& \ (c \ || \ d)) \ \&\& \ e$

потребуется разное количество тестовых примеров. Для первого случая для полного покрытия нужно 6 тестов, для второго – 11.

Метод MC/DC для уменьшения количества тестовых примеров при 3-м уровне покрытия кода

Для уменьшения количества тестовых примеров при тестировании логических условий фирмой Boeing был разработан модифицированный метод покрытия по веткам/условиям (Modified Condition/Decision Coverage или MC/DC). Данный метод широко используется при верификации бортового авиационного программного обеспечения согласно процессам стандарта DO-178B.

Для обеспечения полного покрытия по этому методу необходимо выполнение следующих условий:

- каждое логическое условие должно принимать все возможные значения;
- каждая компонента логического условия должна хотя бы один раз принимать все возможные значения;
- должно быть показано независимое влияние каждой из компонент на значение логического условия, т.е. влияние при фиксированных значениях остальных компонент.

Покрытие по этой метрике требует достаточно большого количества тестов для того, чтобы проверить каждое условие, которое может повлиять на результат выражения, однако это количество значительно меньше, чем требуемое для метода покрытия по всем условиям.

Пример 1. Рассмотрим фрагмент кода, который мы использовали как пример для покрытия по строкам и по веткам. Для покрытия данного участка кода по методу MC/DC введем условные обозначения. Обозначим проверку $prev == \text{"оператор"}$ как А, проверку $prev == \text{"унарный оператор"}$ - как В, а переменную ShowMessage — как С. Первые два обозначения сделаны для того, чтобы элементарными переменными для метода MC/DC были булевы переменные, а третье обозначение — для единообразия.

С учетом сделанных обозначений фрагмент кода может быть записан так:

```
if (A || B)
{
    if (C)
    {
        ...
    }
    else
    {
        ...
    }
}
```

Для тестирования первого условия по MC/DC надо показать независимость результата (т.е. функции $A \ || \ B$) от каждого аргумента. Соответственно, для этого используются три тестовых примера:

1. $A = 0, B = 0, A \ || \ B = 0$ (начальное значение)
2. $A = 1, B = 0, A \ || \ B = 1$ (показано влияние аргумента А)
3. $A = 0, B = 1, A \ || \ B = 1$ (показано влияние аргумента В)

Для тестирования ветвей (входящего в MC/DC) в зависимости от условия С необходимо, чтобы в тестовых примерах С принимало значение как true, так и false.

Итоговая таблица тестовых примеров для покрытия по MC/DC будет выглядеть следующим образом:

Таблица 13

| | 1 | 2 | 3 | 4 |
|-------------|-------------------|-------------------|-----------------------------|----------|
| prev | операнд(A=0, B=0) | оператор(A=1,B=0) | унарный оператор (A=0, B=1) | оператор |
| ShowMessage | false | false | false | true |

Количество тестовых примеров можно сократить до 3, если совместить примеры 3 и 4. Такое совмещение не повлияет на покрытие.

Таблица 14

| | 1 | 2 | 3 |
|-------------|-------------------|-------------------|----------------------------|
| prev | операнд(A=0, B=0) | оператор(A=1,B=0) | унарный оператор(A=0, B=1) |
| ShowMessage | false | false | true |

Пример 2. Для покрытия по MC/DC более сложных выражений рассмотрим следующий участок кода:

```
if ((operators.Count != 0 && operators.Peek().ToString() ==
    == "m") || operators.Peek().ToString() == "p")
{
    strarr.Add(operators.Pop());
}
```

Исходное условное выражение в операторе if можно записать как (A & B) || C. Данное выражение зависит от 3 переменных, т.е. может быть рассмотрено как булева функция с тремя аргументами. Согласно методу MC/DC для тестирования функции с тремя входами достаточно 4 тестовых примеров – один базовый и три показывающих независимое влияние каждого входа на выход.

Начнем построение набора тестов с самой внешней операции, т.е. с ||. Одним из аргументов этой операции является выражение (A & B). Будем пока рассматривать это выражение как единое целое. Для тестирования операции || по MC/DC требуется три тестовых примера:

1. A&B = 0, C = 0 (базовый пример)
2. A&B = 0, C = 1 (независимое влияние C на выход)
3. A&B = 1, C = 0 (независимое влияние A&B на выход)

В третьем тестовом примере значения A и B могут быть получены сразу, т.е. имеем

- 1) A = 1, B = 1, C = 0

Значения A = 1 и B = 1 являются базовыми для тестирования по MC/DC операции &&. Соответственно, необходимо рассмотреть еще два случая, при которых A = 0, B = 1 и A = 1, B = 0 для демонстрации независимого влияния аргументов A и B на значение функции. При этом необходимо, чтобы аргумент C был равен 0, дабы исключить его влияние на выход. Исходя из этих соображений, первый тестовый пример может быть записан как

- 1) A = 1, B = 0, C = 0

т.е. при этом проверяется влияние переменной B на значение функции. Во втором тестовом примере значение C = 1, поэтому он не может быть использован для проверки независимости аргументов A и B. Значения A и B в этом примере могут быть любыми, при условии, что A&B=0. Запишем второй тестовый пример как

- 2) A = 1, B = 0, C = 1

Для тестирования независимого влияния аргумента A необходимо добавить еще один тестовый пример, в котором A = 0, B = 1:

- 4) A = 0, B = 1, C = 0

Таким образом, мы построили 4 тестовых примера для проверки данного участка кода.

Таблица 15

| | 1 | 2 | 3 | 3 |
|--|---|---|---|---|
|--|---|---|---|---|

| | | | | |
|---|-----------|-----------|-----------|-----------|
| A | 1 (true) | 1 (true) | 1 (true) | 0 (false) |
| B | 0 (false) | 0 (false) | 1 (true) | 1 (true) |
| C | 0 (false) | 1 (true) | 0 (false) | 0 (false) |

При переходе от обозначений A, B, C к исходным получим следующие тестовые примеры:

Таблица 16

| | 1 | 2 | 3 | 3 |
|-----------------------------|-------------------|--------------------|------------------|------------------|
| operators.Count | 10 (не 0) | 10 (не 0) | 10 (не 0) | 0 (равен 0) |
| operators.Peek().ToString() | "k" (не m и не p) | "p" (не m, но "p") | "m" (m, но не p) | "m" (m, но не p) |

Анализ покрытия

Целью анализа полноты покрытия кода является выявление участков кода, которые не выполняются при выполнении тестовых примеров. Тестовые примеры, основанные на требованиях, могут не обеспечивать полного выполнения всей структуры кода. Поэтому для улучшения покрытия проводится анализ полноты покрытия кода тестами и, при необходимости, дополнительные проверки, направленные на выяснение причины недостаточного покрытия, а также определение необходимых действий по его устранению. Обычно анализ покрытия выполняется с учетом следующих соглашений.

1. Анализ должен подтвердить, что полнота покрытия тестами структуры кода соответствует требуемому виду покрытия и заданному минимально допустимому проценту покрытия.

2. Анализ полноты покрытия тестами структуры кода может быть выполнен с использованием исходного текста, если программное обеспечение не относится к уровню А. Для уровня А необходимо проверить объектный код, сгенерированный компилятором, чтобы установить, трассируется ли он в Исходный текст или нет. Если Объектный код не трассируется в Исходный текст, должны быть проведены проверки объектного кода на предмет правильности генерации последовательности команд. Примером объектного кода, который напрямую не трассируется в Исходный текст, но генерируется компилятором, может быть проверка выхода за заданные границы массива.

3. Анализ должен подтвердить правильность передачи данных и управления между компонентами кода.

Анализ полноты покрытия тестами может выявить часть исходного кода, которая не исполнялась в ходе тестирования. Для разрешения этого обстоятельства могут потребоваться дополнительные действия в процессе проверки программного обеспечения. Эта неисполняемая часть кода может быть результатом:

1. недостатков в формировании тестовых примеров или тестовых процедур, основанных на требованиях. В этом случае должны быть дополнен набор тестовых примеров или изменены тестовые процедуры для обеспечения покрытия упущенной части кода. При этом может потребоваться пересмотр метода (методов), используемого для проведения анализа полноты тестов на основе требований;

2. неадекватности в требованиях на программное обеспечение. В этом случае должны быть модифицированы требования на программное обеспечение, разработаны и выполнены дополнительные тестовые примеры и тестовые процедуры;

3. "мертвого" кода. Этот код должен быть удален, и проведен анализ для оценки эффекта удаления и необходимости перепроверки;

4. деактивируемого кода. Для деактивируемого кода, который не предполагается к выполнению в каждой конфигурации, сочетание анализа и тестов должно продемонстрировать возможности средств, которыми непреднамеренное исполнение такого кода предотвращается, изолируется или устраняется. Для деактивируемого кода, который выполняется только при определенных конфигурациях, должна быть установлена нормальная эксплуатационная конфигурация для исполнения этого кода, и для нее должны быть разработаны дополнительные тестовые примеры и тестовые процедуры, удовлетворяющие целям полноты покрытия тестами структуры кода;

5. избыточности условия. Логика работы такого условия должна быть пересмотрена. Например, в условии `if(A && B || !B)` принципиально невозможно проверить, что часть условия `A && B` будет равна `False` в случае, когда `A=True` и `B=False`, так как вторая часть условия `(!B)` будет равна `True` и общий результат логического выражения будет `True` ;

6. защитного кода. Эта часть кода используется для предотвращения исключительных ситуаций, которые могут возникнуть в процессе работы программы. Как пример, это может быть ветка `default` в операторе выбора `switch`, причем входное условие оператора `switch` может принимать определенные значения, которые он описывает, и как следствие, ветка `default` никогда не будет выполнена.

Отчеты о покрытии программного кода

Отчеты о покрытии и их связь с другими типами проектной документации

Данные о степени покрытия помещаются в отчеты о покрытии, генерируемые при выполнении тестов инструментальными средствами, поддерживающими процесс тестирования, т.е. по сути генерируются средой тестирования. Формат отчетов о покрытии обычно единый внутри проекта или нескольких проектов и часто зависит от особенностей инструментальных средств тестирования.

В отчете о покрытии в стандартизированной форме указываются участки программного кода тестируемой системы (или ее части), которые не были выполнены во время выполнения тестовых примеров, т.е. не были покрыты тестами. Причины непокрытия анализируются тестировщиками, по результатам анализа составляются отчеты о проблемах и запросы на изменение – документы, где описывается объекты разработки, которые необходимо изменить, и причины этих изменений.

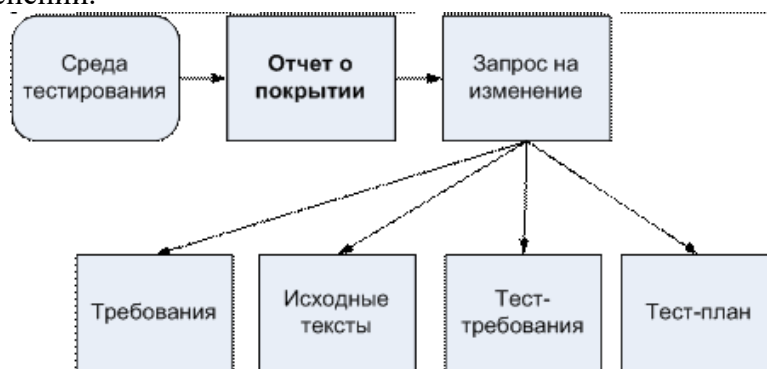


Рисунок 38

Недостаточное покрытие может свидетельствовать о неполноте системы тестов или тест-требований, в этом случае в запросе об изменении указывается на необходимость расширения системы тестов или тест-требований. Другой причиной недостаточного покрытия могут быть участки защитного кода, которые никогда не выполняются даже в случае нештатной работы системы. В этом случае в запросе на изменение указывается на необходимость модификации исходных текстов либо отмечается, что для этого участка программной системы не требуется покрытие. В качестве третьей причины недостаточного покрытия может выступать рассогласование требований и программного кода системы, в результате которого в коде могут остаться неиспользуемые более участки либо, наоборот, появиться участки, рассчитанные на будущее (и реализующие функциональность, не описанную в требованиях). В этом случае в запросе на изменение указывается на необходимость модификации требований и/или кода системы для приведения их в согласованное состояние.

Возможные формы отчетов о покрытии

Типичный отчет о покрытии представляет собой список структурных элементов покрываемого программного кода (функций или методов), содержащий для каждого структурного элемента следующую информацию:

1. название функции или метода;
2. тип покрытия (по строкам, по ветвям, MC/DC или иной);
3. количество покрываемых элементов в функции или методе (строк, ветвей, логических условий);

4. степень покрытия функции или метода (в процентах или в абсолютном выражении);
5. список непокрытых элементов (в виде участков непокрытого программного кода с номерами строк).

Кроме того, отчет о покрытии содержит заголовочную информацию, позволяющую идентифицировать отчет, и общий итог – общую степень покрытия всех функций, для которых собирается информация о покрытии.

Отчет о покрытии может создаваться либо для всех функций или методов программного модуля или всего проекта, либо выборочно для определенных функций или методов.

В случае, если размер функций, для которых генерируется выборочный отчет, невелик, может применяться другая форма отчета о покрытии, в котором покрытый и непокрытый программный код выделяются различными цветами. Такая форма неприменима для покрытия ветвей и логических условий, но может использоваться для покрытия по строкам.

Покрытие на уровне исходных текстов и на уровне машинных кодов

В некоторых случаях инструментальные средства сбора покрытия анализируют покрытие программного кода тестами не на уровне исходных текстов системы, а на уровне машинных инструкций. В этом случае степень покрытия зависит и от того, какой исполняемый код генерируется компилятором.

Поскольку степень покрытия может меняться в зависимости от оптимизации при генерации кода, в некоторых случаях даже при полном выполнении всех операторов языка высокого уровня, на котором написана программная система, не удастся достичь полного покрытия на уровне исполняемого кода.

Сбор информации о покрытии на уровне исполняемого кода наиболее часто применяется в высококритичных программных системах, где не допускается наличия "мертвого" исполняемого кода, который потенциально может привести к сбою или отказу во время работы системы. К таким системам, в первую очередь, можно отнести авиационные бортовые системы, медицинские системы и системы обеспечения безопасности информации.

Возможности MVSTE по построению покрытия кода

Замечание. Подробнее о покрытии кода можно прочитать по адресу: [http://msdn2.microsoft.com/en-us/library/ms182496\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/ms182496(VS.80).aspx)

Чтобы увидеть, какая часть кода вашего проекта фактически тестируется, используйте такой инструмент для тестировщиков в Visual Studio Team System, как Покрытие кода. Этот инструмент показывает процент кода, который был выполнен, и "раскрашивает" его, показывая, какие линии кода были выполнены, а какие нет.

На прошлых семинарах мы познакомились с возможностями MVSTE по автоматизации модульного тестирования на примерах тестирования метода Add класса CalcClass и метода RunEstimate класса AnalizerClass. Покажем на этих же примерах, какую часть кода покрыли созданные нами модульные тесты.

Для начала откроем BaseCalculator, с которым мы работали на "Тестовые примеры. Классы эквивалентности. Ручное тестирование в MVSTE" .

Далее в меню Test выбираем Edit Test Run Configuration. В подменю выбираем Local Test Run (localtestrun.testrunconfig), чтобы запустить файл конфигурации. (аналогично запустить файл конфигурации можно, щелкнув в Solution Explorer под Solution Items на localtestrun.testrunconfig). Появится диалоговое окно localtestrun.testrunconfig

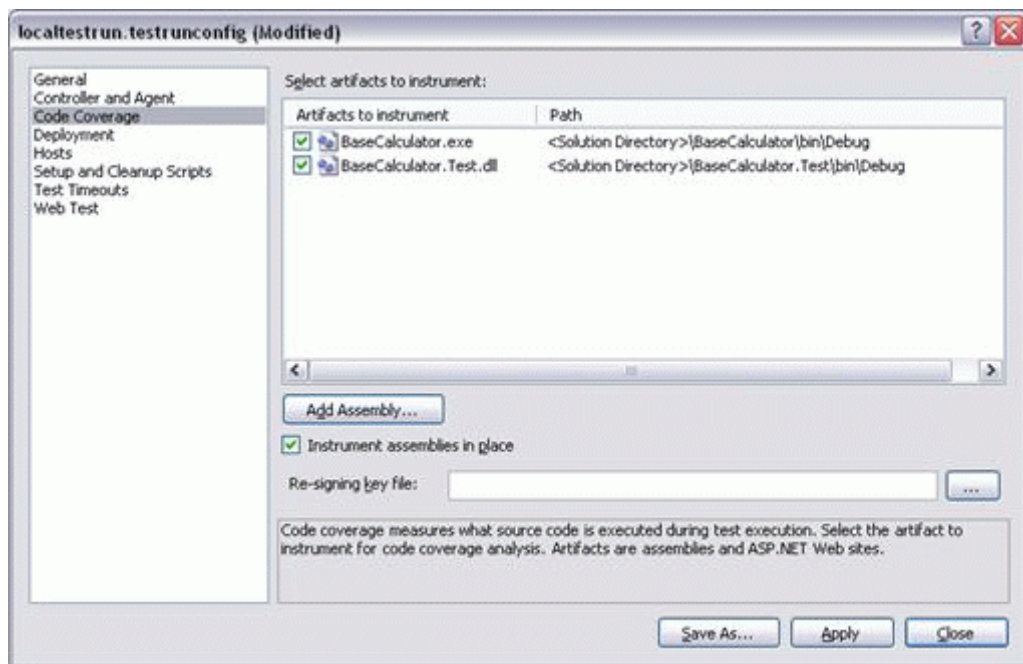


Рисунок 39

Выбираем Code Coverage.

В поле Select artifacts to instrument отмечаем пункты BaseCalculator.exe и BaseCalculator.Test.dll

Замечание. Если вы выбрали BaseCalculator.Test.dll, то MVSTE генерирует информацию о покрытии кода для методов в вашем тестовом проекте.

Нажмите Apply, затем закройте диалоговое окно. Мы настроили файл конфигурации.

Далее в меню Test выберите Select Active Test Run Configuration. Подменю показывает все конфигурации запуска теста вашего решения. Поместим метку на конфигурацию запуска, которую мы только что редактировали, localtestrun.testrunconfig ; что сделает её активной конфигурацией запуска теста.

В окне Test View выделяем все тесты и нажимаем кнопку Run Selection. Запустятся созданные нами на "Тестовые примеры. Классы эквивалентности. Ручное тестирование в MVSTE" тесты.

После выполнения всех тестов в окне Test Results в меню Test выберем Windows и в раскрывшемся подменю нажмем на Code Coverage Results. Откроется окно Code Coverage Results.

| Hierarchy | Not Covered (Blocks) | Not Covered (% Blocks) | Covered (Blocks) | Covered (% Blocks) |
|--------------------------------|----------------------|------------------------|------------------|--------------------|
| Belf@BLEFF 2006-11-08 01:34:12 | 1835 | 94.30 % | 111 | 5.70 % |
| BaseCalculator.Test.dll | 162 | 76.78 % | 49 | 23.22 % |
| BaseCalculator.exe | 1673 | 96.43 % | 62 | 3.57 % |

Рисунок 40

Замечание. Колонка Hierarchy изначально показывает единственный узел (в данном случае Belf@BLEFF 2006-11-08 01:34:12), который содержит данные обо всем покрытии кода, достигнутом в последнем запущенном и выполненном тесте. Он назван, используя следующий формат: <user name>@<computer name> <date> <time>.

Если вы раскроете этот узел, то увидите выбранные нами в файле конфигурации проекты BaseCalculator.exe и BaseCalculator.Test.dll

Раскроем узел для сборки BaseCalculator.exe, для пространства имен BaseCalculator и для класса CalcClass, как показано на рис.39

| Hierarchy | Not Covered (Blocks) | Not Covered (% Blocks) | Covered (Blocks) | Covered (% Blocks) |
|--------------------------------|----------------------|------------------------|------------------|--------------------|
| Belf@BLEFF 2006-11-08 01:34:12 | 1835 | 94.30 % | 111 | 5.70 % |
| BaseCalculator.Test.dll | 162 | 76.78 % | 49 | 23.22 % |
| BaseCalculator.exe | 1673 | 96.43 % | 62 | 3.57 % |
| BaseCalculator | 1655 | 96.39 % | 62 | 3.61 % |
| AnalyzerClass | 870 | 95.08 % | 45 | 4.92 % |
| BaseCalc | 630 | 100.00 % | 0 | 0.00 % |
| CalcClass | 137 | 89.54 % | 16 | 10.46 % |
| ctor() | 0 | 0.00 % | 1 | 100.00 % |
| ABS(int64) | 14 | 100.00 % | 0 | 0.00 % |
| Add(int64,int64) | 0 | 0.00 % | 13 | 100.00 % |
| Cosinus(float64) | 3 | 100.00 % | 0 | 0.00 % |
| Cothangens(float64) | 3 | 100.00 % | 0 | 0.00 % |
| Cube(float64) | 3 | 100.00 % | 0 | 0.00 % |
| Div(int64,int64) | 21 | 100.00 % | 0 | 0.00 % |

Рисунок 41

Строки в классе CalcClass представляют его методы. Колонки в окне Code Coverage Results показывают статистику покрытия для отдельных методов, для классов, и для всего пространства имен.

Замечание. В этой таблице можно выбирать, в каком порядке и какие колонки будут отображаться. Для этого нужно нажать правой кнопкой мыши в окне Code Coverage Results и выбрать в контекстном меню пункт Add/Remove Columns.

Замечание. Статистика покрытия кода показывает покрытие блоков и линий кода.

Чтобы посмотреть, какая именно часть кода была покрыта, щелчком два раза на строку с методом Add.

Откроется файл исходного текста CalcClass.cs на методе Add. В этом файле мы видим "окрашенный" код. Линии, окрашенные в голубой цвет, были выполнены в процессе выполнения тестов, а линии, окрашенные в красный, не были выполнены.

Замечание. Цвета окрашивания линий покрытия кода можно изменить. Для этого нужно зайти в Tools->Options. В открывшемся диалоговом окне нужно выбрать Environment->Fonts and Colors. Далее в раскрывающемся списке Show settings for выбираем Text Editor. Далее в Display items нужно выбрать область покрытия кода, цвет которой вы хотите изменить: это Coverage Not Touched Area, или Coverage Partially Touched Area, или Coverage Touched Area. Для этих областей покрытия кода можно изменить шрифт, его размер, жирность, цвет текста и его "окрашивание". По завершении, чтобы сохранить изменения и выйти из диалогового окна, нажмите OK.

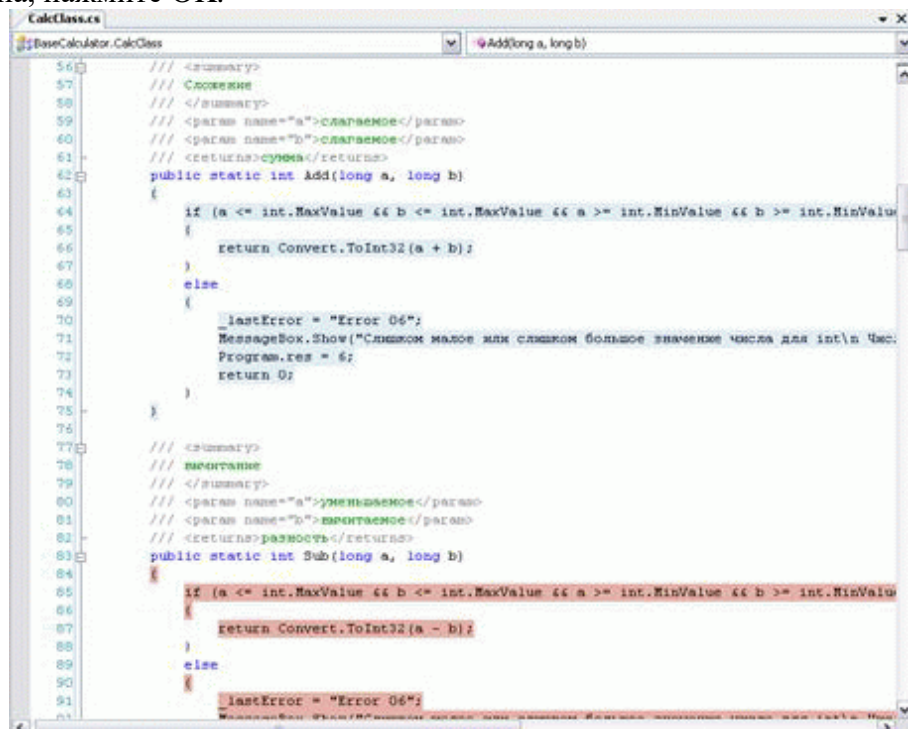



Рисунок 42

Прокручивая файл, вы можете увидеть в нем покрытие для других методов.

Замечание. Точно так же можно просмотреть покрытие кода наших модульных тестов, то есть, можно увидеть, какие из тестовых методов были осуществлены (раскрыв в окне Code Coverage Results сборку BaseCalculator.Test.dll). Применяется та же самая схема окрашивания: голубой показывает выполненный в процессе выполнения теста код; красный — невыполненный.

Замечание. Результаты покрытия кода можно экспортировать в отдельный XML-файл.

Для этого в окне Code Coverage Results нужно нажать на кнопку Export Results , указать имя и местоположение файла.

Программа

Модульные тесты, которые написали студенты, выполнив домашнее задание "Тестовые примеры. Классы эквивалентности. Ручное тестирование в MVSTE". Доработать модульные тесты, добившись максимального покрытия кода по MC\DC.

Практическая работа № 2.20. Повторяемость тестирования, зависимости тестовых примеров

Цель работы: анализ тестирования системы

Тест

В каждом тестовом задании может быть несколько вариантов ответа. После проведения теста студенты могут попробовать обосновать свои неверные ответы.

1. Полная система тестов позволяет утверждать, что:
1. система реализует всю функциональность, указанную в требованиях
2. система работает корректно
3. система не реализует функциональность, которая не указана в требованиях
4. система работает правильно
5. система реализует функциональность, которая не указана в требованиях
6. система не реализует функциональность, которая указана в требованиях

Ответ: 1, 3

2. Выберите верные утверждения:

1. Полное покрытие по веткам дает полное покрытие по строкам.
2. Полное покрытие по веткам не дает полного покрытия по строкам.
3. Полное покрытие по строкам без ветвления дает полное покрытие кода по веткам.
4. Полное покрытие по MC\DC не дает полного покрытия по строкам.

Ответ: 1, 3

3. Какие условия должны быть выполнены для обеспечения полного покрытия по методу MC\DC?

1. должно быть показано зависимое влияние каждой из компонент на значение логического условия
2. каждое логическое условие должно принимать все возможные значения
3. каждая компонента логического условия должна хотя бы один раз принимать все возможные значения
4. любая часть логического условия должна принимать хотя бы раз все возможные значения
5. должно быть показано независимое влияние каждой из компонент на значение логического условия

Ответ: 2, 3, 5

4. Согласно методу MC\DC для тестирования логической функции с тремя входами и одним выходом достаточно:

1. 3-х тестовых примеров
2. 4-х тестовых примеров
3. 5-х тестовых примеров

4. 6-х тестовых примеров

Ответ: 2

5. Одной из основных задач анализа полноты покрытия кода является:

1. выявление участков кода, которые выполняются при выполнении тестовых примеров
2. выявление участков кода, которые содержат ошибки
3. выявление участков кода, которые не выполняются при выполнении тестовых примеров
4. выявление участков кода, которые не содержат ошибок

Ответ: 3

Повторяемость тестирования

Задачи и цели обеспечения повторяемости тестирования при промышленной разработке программного обеспечения

Как уже было сказано в предыдущих темах, тестирование программной системы – не разовое мероприятие, а постоянный процесс, активный в течение всего жизненного цикла разработки системы. В течение этого процесса система неизбежно изменяется – либо в результате исправления ошибок, либо в результате расширения ее функциональности. Задача тестировщика в такой ситуации – подтвердить, что новая или исправленная функциональность не вызвала новые ошибки, а если ошибки все-таки возникли – определить причины их возникновения.

Самый простой, но в то же время действенный способ такого подтверждения – полное выполнение всех тестовых примеров после каждого существенного изменения системы и сравнение результатов выполнения тестов до и после изменения.

Если результаты выполнения тестов до внесения изменений были положительными (все тесты проходили успешно), то появление неуспешно пройденных тестов может означать, что в системе появились новые дефекты, вызванные исправлением старых.

В общем случае повторное выполнение тестов может завершиться одним из трех способов.

1. Все тесты пройдены успешно. В этом случае изменения не затрагивают уже протестированные функции, но может потребоваться разработка новых тестовых примеров для новых функций системы.

2. Часть тестов, ранее выполнявшихся успешно, завершается с отрицательным результатом. Причины этого могут быть следующие:

- корректное изменение функциональности тестируемой системы, в результате которого тестовый пример перестал соответствовать требованиям;
- некорректное изменение функциональности системы, в результате которого тестовый пример выявил расхождение с требованиями;
- влияние остаточных данных от предыдущих тестовых примеров, ранее остававшееся незамеченным.

Первые две причины различимы только при помощи анализа изменений в функциональных требованиях и тест-требованиях, а также текущего состояния тест-планов и тестового окружения. По результатам этого анализа в первом случае тестировщик вносит изменения в тестовый пример (и, возможно, разрабатываются новые тестовые примеры), во втором случае тестировщик уведомляет разработчиков о наличии дефекта.

3. Выполнение тестов аварийно завершается в самом начале или при выполнении определенного тестового примера.

Данная проблема чаще всего связана с изменением внешнего окружения тестируемой части системы, которое моделирует тестовое окружение. Из-за таких изменений могут меняться внешние интерфейсы, а также состав и формат входных и выходных данных. В результате тестовое окружение перестает обеспечивать необходимую для выполнения тестов инфраструктуру и возникает сбой процесса тестирования. Например, такой сбой может возникнуть в тестовом окружении при попытке обработать данные, выдаваемые системой в новом формате.

Если для выполнения тестов требуется сборка программных модулей тестового окружения и тестируемой системы в единый исполняемый код, то при изменении интерфейсов системы может возникнуть ситуация, когда невозможно не только выполнение тестов, а даже сборка окружения и системы. В этом случае также необходимо провести анализ изменений, внесенных в систему, и модифицировать в соответствии с ними тестовое окружение.

Иногда повторное выполнение всех тестов невозможно. Это может быть связано с большим временем выполнения всех тестов и ограниченным временем, отведенным на процесс тестирования. В этом случае часто применяется практика выборочного тестирования отдельных частей системы, затронутых изменениями. Полное тестирование при таком подходе проводится только после накопления достаточно большого количества изменений или на ключевых стадиях проекта.

Процесс, включающий в себя повторное выполнение всех тестов, называют регрессионным тестированием. Регрессионное тестирование включает в себя следующие стадии:

1. Анализ изменений в системе
2. Выбор тестовых примеров для проверки системы
3. Выполнение тестовых примеров
4. Анализ результатов выполнения
5. Модификация тестового окружения, тестовых примеров или уведомление разработчиков о дефекте системы.

Таким образом можно определить следующие основные задачи повторяемости тестирования при внесении изменений.

- Обеспечение возможности полного выполнения всех тестов, проверяющих функциональность системы или проведение анализа, позволяющего выявить тесты, которые должны быть повторно выполнены для тестирования изменившейся функциональности.
- Разработка тестовых примеров и тестового окружения с использованием методик, облегчающих модификацию при изменениях в тестируемой системе.
- Разработка тестовых примеров, структура которых полностью исключает их взаимное влияние по остаточным данным.

Целью повторяемости тестирования является постоянное обеспечение тестировщиков и разработчиков актуальной информацией о текущем состоянии системы и корректности изменений, внесенных в ходе разработки системы.

Предусловия для выполнения теста, настройка тестового окружения, оптимизация последовательностей тестовых примеров

Как уже было сказано ранее, входные данные в каждом тестовом примере явно задают начальное состояние тестируемой системы и режимы ее работы при выполнении тестового сценария.

Однако неявное влияние на выполнение теста оказывает и состояние тестового окружения. Под состоянием здесь понимается набор параметров, изменение любого из которых может повлиять либо на результат выполнения тестового примера, либо на возможность его корректной работы и завершения.

Например, для выполнения тестового примера тестируемой системе может потребоваться значительный объем дисковой или оперативной памяти. Если перед выполнением теста тестовое окружение выделит эту память под свои нужды, выполнение теста окажется невозможным. Та же самая ситуация может возникнуть и в случае, если окружение не освободит память после выполнения предыдущего тестового примера.

Эта информация обычно отсутствует в тест-планах, однако требуемое для выполнения тестов состояние тестового окружения необходимо учитывать при разработке тестовых примеров.

Хорошей практикой является оформление проверок на допустимость состояния тестового окружения в виде предусловий для выполнения теста. Это позволяет

диагностировать ситуации, возникающие при выборочном тестировании и приводящие к отказам тестового окружения.

На практике часто возникает ситуация в которой друг за другом следует несколько десятков тестовых примеров, а при регрессионном тестировании требуется выполнить, например, тестовые примеры с номерами от 25 по 40. Первый тестовый пример при этом инициализирует систему, а остальные работают с уже стартовавшей системой. Если просто выполнять тестовые примеры 25-40, то их выполнение окажется невозможным – они не инициализируют систему. Разумным выходом из этой ситуации является выполнение тестовых примеров 1, 25-40.

Зависимость между тестовыми примерами, настройки по умолчанию для тестовых примеров и их групп

Для облегчения проведения регрессионного тестирования (и тестирования вообще) тестовые примеры часто разбивают на группы. Каждая группа содержит набор тестовых примеров, проверяющих отдельную замкнутую часть функциональности тестируемой системы. При отборе тестовых примеров для частичного регрессионного тестирования их можно отбирать сразу группами.

Разбиение тестовых примеров на группы удобно и с точки зрения установки начального состояния тестового окружения для выполнения тестов – так, перед выполнением группы тестов можно инициализировать значения переменных или состояние системы, необходимое для выполнения всей группы. Например, если система работает в двух режимах – нормальном и сервисном, то перед выполнением группы тестов для нормального режима работы системы нужно устанавливать нормальный режим, а перед выполнением тестов для сервисного режима – сервисный. Такие установки называются настройками группы тестов по умолчанию (group defaults, test group defaults).

Перед выполнением каждого тестового примера может потребоваться установка одних и тех же переменных в одни и те же значения. Для того, чтобы не дублировать эти установки в описании каждого тестового примера, в тест-плане можно определить настройки по умолчанию для каждого теста (test case defaults).

Как видно из предыдущего раздела, для облегчения проведения выборочного регрессионного тестирования каждый тестовый пример должен быть полностью автономным – ход его выполнения и тем более, результат не должны зависеть от предыдущих тестовых примеров. Тем самым, при выборочном тестировании результат тестирования не зависит от выбранного набора тестовых примеров (тестового набора). Однако, на практике создание автономных тестов зачастую невозможно по различным причинам (как правило – из-за длительного времени выполнения таких тестов).

В случае, когда в наборе тестовых примеров тесты не являются автономными, говорят о тестовой зависимости. Тестовая зависимость бывает двух видов – предусмотренная структурой тестовых примеров и паразитная.

Пример предусмотренной тестовой зависимости был рассмотрен в предыдущем разделе – корректность выполнения тестов определялась порядком их выполнения. Такая тестовая зависимость требует документирования и сопровождения, как и сами описания тестовых примеров. Существует два вида документирования тестовых зависимостей:

- явное определение допустимого порядка выполнения тестовых примеров. Такой способ удобен при сравнительно небольшом общем количестве тестовых примеров, либо, при разбиении на группы – при небольшом размере групп тестовых примеров;
- определение допустимого порядка выполнения тестовых примеров при помощи предусловий. При таком способе корректность порядка выполнения тестовых примеров определяется при помощи проверки того, что либо тестируемая система, либо тестовое окружение находятся в необходимом состоянии для выполнения тестового примера.

Паразитные тестовые зависимости обычно вызваны некорректным составлением тест-плана. Проявляются они, как и предусмотренные зависимости, в том, что один (или более) тестовых примеров корректно работает только в том случае, если до него были выполнены

другие тестовые примеры. Причем такая зависимость не является предусмотренной тестировщиком. Природа паразитной тестовой зависимости схожа с природой ошибок использования неинициализированных или остаточных данных в динамической памяти при программировании.

На примере "Калькулятора"

Рассмотрим повторяемость тестирования на примере нашего "Калькулятора".

Рассмотрим свойство CalcClass.lastError:

```
/// <summary>
/// Последнее сообщение об ошибке.
/// </summary>
private static string _lastError = "";

public static string lastError
{
    get
    {
    }
}
```

Оно хранит последнее сообщение об ошибке. При этом "Калькулятор", вычисляя выражение после каждой арифметической операции, проверяет значение переменной и, если оно не равно пустой строке, выдает сообщение об ошибке и прерывает работу. Однако у свойства lastError нет аксессуара set, и значит, никакой внешний модуль не может поменять его значения. Напрашивается вопрос – а как же сбрасывается это значение? Проведем три теста подряд на методе сложения:

```
try
{
    richTextBox1.Text = "";
    richTextBox1.Text += "Test Case 1\n";
    richTextBox1.Text += "Входные данные: a= 78508, b = -304\n";
    richTextBox1.Text += "Ожидаемый результат: res = 78204 &&";
    error = "\"\" + "\n";
    int res = CalcClass.Add(78508, -304);
    string error = CalcClass.lastError;
    richTextBox1.Text += "Код ошибки: " + error + "\n";
    richTextBox1.Text += "Получившийся результат: " + "res = " +
        res.ToString() + " error = " + error.ToString() + "\n";
    if (res == 78204 && error == "")
    {
        richTextBox1.Text += "Тест пройден\n\n";
    }
    else
    {
        richTextBox1.Text += "Тест не пройден\n\n";
    }
}
catch (Exception ex)
{
    richTextBox1.Text += "Перехвачено исключение: " +
        ex.ToString() + "\nТест не пройден.\n";
}
try
```



```

{
    richTextBox1.Text += "Test Case 2\n";
    richTextBox1.Text += "Входные данные: a= -2850800078, b = 3000000000\n";
    richTextBox1.Text += "Ожидаемый результат: res = 0 && error = \nError 06\n";
    int res = CalcClass.Add(-2850800078, 3000000000);
    string error = CalcClass.lastError;
    richTextBox1.Text += "Код ошибки: " + error + "\n";
    richTextBox1.Text += "Получившийся результат: " + "res = " + res.ToString() + " error = " + error.ToString() + "\n";
    if (res == 0 && error == "Error 06")
    {
        richTextBox1.Text += "Тест пройден\n\n";
    }
    else
    {
        richTextBox1.Text += "Тест не пройден\n\n";
    }
}
catch (Exception ex)
{
    richTextBox1.Text += "Перехвачено исключение: " + ex.ToString() + "\nТест не пройден.\n";
}

try
{
    richTextBox1.Text += "Test Case 3\n(повторный тест)";
    richTextBox1.Text += "Входные данные: a= 78508, b = -304\n";
    richTextBox1.Text += "Ожидаемый результат: res = 78204 && error = \"\" + "\n";
    int res = CalcClass.Add(78508, -304);
    string error = CalcClass.lastError;
    richTextBox1.Text += "Код ошибки: " + error + "\n";
    richTextBox1.Text += "Получившийся результат: " + "res = " + res.ToString() + " error = " + error.ToString() + "\n";
    if (res == 78204 && error == "")
    {
        richTextBox1.Text += "Тест пройден\n\n";
    }
    else
    {
        richTextBox1.Text += "Тест не пройден\n\n";
    }
}
catch (Exception ex)
{
    richTextBox1.Text += "Перехвачено исключение: " + ex.ToString() + "\nТест не пройден.\n";
}

Результат:
Test Case 1
Входные данные: a= 78508, b = -304

```


Ожидаемый результат: res = 78204 && error = ""

Код ошибки:

Получившийся результат: res = 78204 error =

Тест пройден

Test Case 2

Входные данные: a= -2850800078, b = 30000000000

Ожидаемый результат: res = 0 && error = "Error 06"

Код ошибки: Error 06

Получившийся результат: res = 0 error = Error 06

Тест пройден

Test Case 3

(повторный тест)Входные данные: a= 78508, b = -304

Ожидаемый результат: res = 78204 && error = ""

Код ошибки: Error 06

Получившийся результат: res = 78204 error = Error 06

Тест не пройден

Как видно, несмотря на то, что третий тест операции сложения должен быть выполнен, он не проходит, хотя по первому тесту видно, что сложение работает правильно, а значение lastError точно такое же, что и во втором тесте. Это может свидетельствовать, например, о том, что при вызове метода Add в начале своей работы не очищается поле _lastError. Проведем тестирование всех функций:

```
try
{
    richTextBox1.Text += "Test Case 2\n";
    richTextBox1.Text += "Входные данные: a= -2850800078, b = 30000000000\n";
    richTextBox1.Text += "Ожидаемый результат: res = 0 && error = \nError 06\n";
    int res = CalcClass.Add(-2850800078, 30000000000);
    string error = CalcClass.lastError;
    richTextBox1.Text += "Код ошибки: " + error + "\n";
    richTextBox1.Text += "Получившийся результат: " + "res = " +
        res.ToString() + " error = " + error.ToString() + "\n";
    if (res == 0 && error == "Error 06")
    {
        richTextBox1.Text += "Тест пройден\n\n";
    }
    else
    {
        richTextBox1.Text += "Тест не пройден\n\n";
    }
}
catch (Exception ex)
{
    richTextBox1.Text += "Перехвачено исключение: " +
        ex.ToString() + "\nТест не пройден.\n";
}

try
{
    richTextBox1.Text += "Test Case 3\n(повторный тест)";
    richTextBox1.Text += "Входные данные: a= 78508, b = -304\n";
    richTextBox1.Text += "Ожидаемый результат: res = 78204 &&
        error = \n\"\" + "\n";
```

```

int res = CalcClass.Add(78508, -304);
string error = CalcClass.lastError;
richTextBox1.Text += "Код ошибки: " + error + "\n";
richTextBox1.Text += "Получившийся результат: " + "res = " +
    res.ToString() + " error = " + error.ToString() + "\n";
if (res == 78204 && error == "")
{
    richTextBox1.Text += "Тест пройден\n\n";
}
else
{
    richTextBox1.Text += "Тест не пройден\n\n";
}
}
catch (Exception ex)
{
    richTextBox1.Text += "Перехвачено исключение: " +
        ex.ToString() + "\nТест не пройден.\n";
}

try
{
    richTextBox1.Text += "Test Case 4 - проверяем вычитание на корректных данных";
    richTextBox1.Text += "Входные данные: a= 78508, b = -304\n";
    richTextBox1.Text += "Ожидаемый результат: res = 78812 &&
        error = \"\" + "\n";
    int res = CalcClass.Sub(78508, -304);
    string error = CalcClass.lastError;
    richTextBox1.Text += "Код ошибки: " + error + "\n";
    richTextBox1.Text += "Получившийся результат: " + "res = " +
        res.ToString() + " error = " + error.ToString() + "\n";
    if (res == 78508 && error == "")
    {
        richTextBox1.Text += "Тест пройден\n\n";
    }
    else
    {
        richTextBox1.Text += "Тест не пройден\n\n";
    }
}
catch (Exception ex)
{
    richTextBox1.Text += "Перехвачено исключение: " +
        ex.ToString() + "\nТест не пройден.\n";
}

try
{
    richTextBox1.Text += "Test Case 5 - проверяем произведение на корректных данных";
    richTextBox1.Text += "Входные данные: a= 78508, b = -304\n";
    richTextBox1.Text += "Ожидаемый результат: res = 23866432 &&
        error = \"\" + "\n";
    int res = CalcClass.Mult(78508, -304);

```

```

string error = CalcClass.lastError;
richTextBox1.Text += "Код ошибки: " + error + "\n";
richTextBox1.Text += "Получившийся результат: " + "res = " +
    res.ToString() + " error = " + error.ToString() + "\n";
if (res == 23866432 && error == "")
{
    richTextBox1.Text += "Тест пройден\n\n";
}
else
{
    richTextBox1.Text += "Тест не пройден\n\n";
}
}
catch (Exception ex)
{
    richTextBox1.Text += "Перехвачено исключение: " +
        ex.ToString() + "\nТест не пройден.\n";
}

```

Результат

Test Case 2

Входные данные: a= -2850800078, b = 3000000000

Ожидаемый результат: res = 0 && error = "Error 06"

Код ошибки: Error 06

Получившийся результат: res = 0 error = Error 06

Тест пройден

Test Case 3

(повторный тест) Входные данные: a= 78508, b = -304

Ожидаемый результат: res = 78204 && error = ""

Код ошибки: Error 06

Получившийся результат: res = 78204 error = Error 06

Тест не пройден

Test Case 4 — проверяем вычитание на корректных данных

Входные данные: a= 78508, b = -304

Ожидаемый результат: res = 78812 && error = ""

Код ошибки: Error 06

Получившийся результат: res = 78812 error = Error 06

Тест не пройден

Test Case 5 — проверяем произведение на корректных данных

Входные данные: a= 78508, b = -304

Ожидаемый результат: res = 23866432 && error = ""

Код ошибки: Error 06

Получившийся результат: res = -23866432 error = Error 06

Тест не пройден

Мы видим, что ни один из методов не очищает поле `_lastError`. Это может быть либо ошибкой проектирования, либо неправильной реализацией свойства `lastError`. Можно либо отправить на доработку все методы и функциональные требования, указав в них, что методы должны перед началом работы очищать свойство `lastError`, либо доработать свойство следующим образом:

```

public static string lastError
{
    get
    {
        string temp = _lastError;
    }
}

```

```

        _lastError = "";
        return temp;
    }
}

```

Таким образом, после любого чтения этой переменной, её значение снова будет равняться пустой строке.

Замечание. Несмотря на кажущуюся "притяннутость" этого примера, он очень характерен. Можно вернуться к "Тестовые примеры. Классы эквивалентности. Ручное тестирование в MVSTE" и удалить строки в тестовом модуле, очищающие значение `_lastError`. При запуске тестов третий тест вместо перехвата исключения сообщит, что метод закончил работу с кодом ошибки 3. Причина как раз в том, что после выполнения теста 2 значение `_lastError` не было очищено. Это еще раз свидетельствует о том, что тестам надо создавать корректное тестовое окружение. Дальше будет приведен еще один пример неправильного построения тестового окружения.

Рассмотренный пример является довольно простым, и ошибка будет легко выявлена при тестировании. В четвертом семинаре, при написании тестового драйвера для метода `RunEstimate()`, мы подключали сборку `My.dll`:

```

System.IO.BinaryReader reader = new System.IO.BinaryReader
(new System.IO.FileStream(Application.StartupPath +
+ "\\My.dll", System.IO.FileMode.Open,
System.IO.FileAccess.Read));
Byte[] asmBytes = new Byte[reader.BaseStream.Length];
reader.Read(asmBytes, 0, (Int32) reader.BaseStream.Length);
reader.Close();
reader = null;

```

```

System.Reflection.Assembly assm =
System.Reflection.Assembly.Load(asmBytes);

```

Может показаться, что мы выполняем лишние действия, и вместо всех этих строк кода легче применить метод `System.Reflection.Assembly.LoadFile`, который сразу подключит необходимую сборку по пути библиотеки.

Замечание. В Framework 2.0 метод `LoadFile()` объявлен как устаревший.

Дальше мы можем проводить сколько угодно тестов методов класса `AnaIaizerClass`.

Но если нам понадобится поменять заглушку класса `CalcClass` (например, на ту, которая выводит на экран какие-то дополнительные сведения), то мы получим ошибку `access denied`, т.к. сборка уже загружена и используется процессом, и перекомпилировать её не получится.

Это другая проблема регрессионного тестирования. Здесь уже ошибка не в тестируемой программе, а в самом построении тестов и тестового окружения. Как было сказано во вступлении, нужно либо для каждого теста заново проводить инициализацию тестового окружения, либо объединять тесты в группы с общей инициализацией, но следя за тем, чтобы тесты не запускались по отдельности.

Упорядоченные тесты (ordered tests) в MVSTE

Замечание. Подробнее об упорядоченных тестах можно почитать по адресу <http://msdn2.microsoft.com/en-us/library/ms182629.aspx>

Упорядоченный тест содержит в себе другие тесты и предназначается для того, чтобы запускать их в указанном порядке. Упорядоченный тест появляется как отдельный тест в окне `Test View`, а результаты его выполнения появляются одной строкой в окне `Test Results`.

Замечание. Упорядоченный тест может содержать тесты любого типа кроме нагрузочных тестов. Ещё одной особенностью упорядоченного теста является то, что, если мы запустим его удаленно или из командной строки, будет показано предупреждение о том, что, чтобы запустить тест, все ручные тесты, которые содержатся в нем, будут временно из него удалены.

Замечание. Прежде чем Вы начнете создавать упорядоченные тесты, у Вас должны быть в наличии другие тесты, которые можно будет включить в упорядоченный тест.

Рассмотрим сначала создание упорядоченного теста.

Для начала откроем BaseCalculator, с которым мы работали на "Тестовые примеры. Классы эквивалентности. Ручное тестирование в MVSTE" и "Тестирование программного кода (покрытия)". Он уже содержит тесты.

Добавим в тестовый проект упорядоченный тест. Для этого в меню Test выберем New Test. В появившемся диалоговом окне Add New Test выбираем Ordered Test. В поле Test Name введем название теста, например, OrderedTest.orderedtest, а в пункте Add to Test Project выберем наш тестовый проект BaseCalculator.Test. Для добавления теста в проект нажимаем ОК.

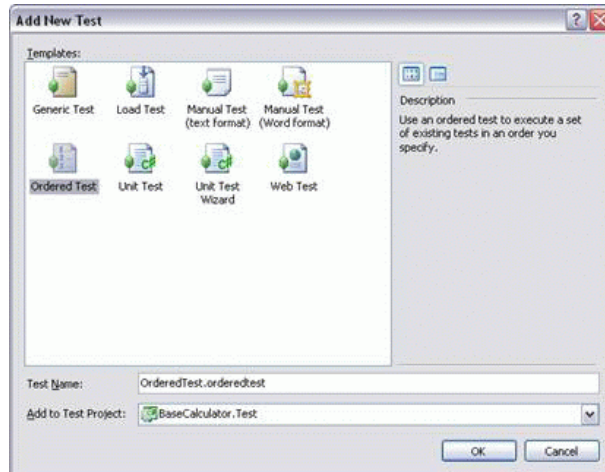


Рисунок 43

В Solution Explorer добавится файл OrderedTest.orderedtest и откроется окно редактирования упорядоченного теста OrderedTest.orderedtest. Мы будем использовать это окно, чтобы выбирать и включать тесты в наш упорядоченный тест.

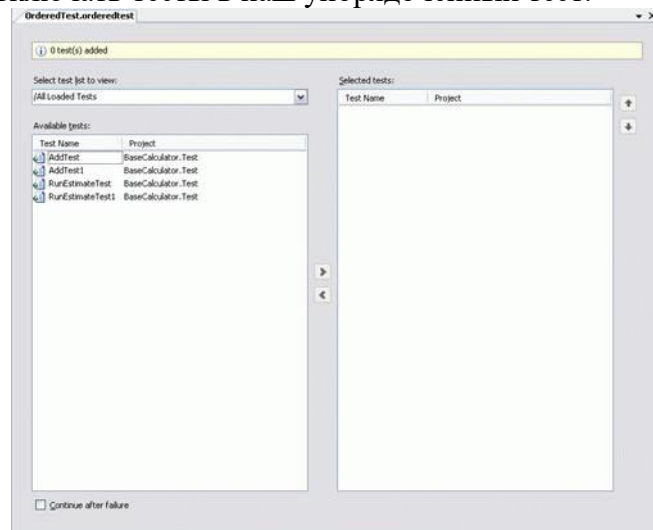




Рисунок 44



Рассмотрим подробнее работу с окном редактирования нашего упорядоченного теста OrderedTest.orderedtest.

В выпадающем меню Select Test List to View можно выбрать список тестов: Lists of Tests, Tests Not in a List, All Loaded Tests или определенный тестовый список. Выберем All Loaded Tests. В Available tests отобразятся все созданные нами ранее тесты.

Чтобы добавить тесты в упорядоченный тест, нужно в Available tests выделить те тесты, которые вы хотите добавить (например, используя SHIFT+click и CTRL+click), и нажать на стрелку вправо . Тесты добавлены в упорядоченный тест.

Замечание. Можно добавлять одни и те же тесты многократно к одному и тому же упорядоченному тесту.

Чтобы удалить тест из упорядоченного теста, нужно в Selected tests выделить те тесты, которые вы хотите удалить (например, используя SHIFT+click и CTRL+click), и нажать на стрелку влево . Тесты удалены из упорядоченного теста.

Чтобы изменить порядок тестов в упорядоченном тесте, нужно в Selected tests выделить те тесты, порядок которых вы хотите изменить (например, используя SHIFT+click и CTRL+click), и нажать на стрелку вверх  или вниз . Порядок тестов в упорядоченном тесте будет изменен.

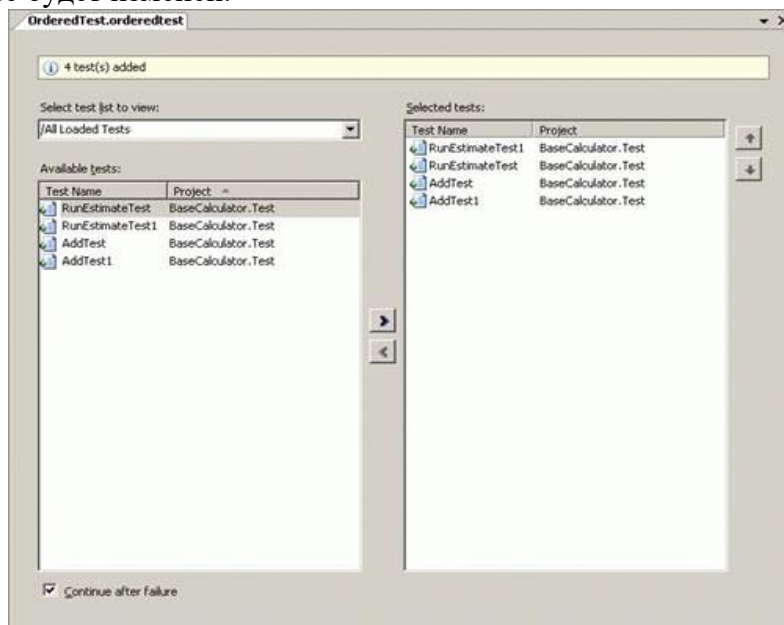


Рисунок 45

Замечание. Выставленный флаг Continue after failure указывает на то, что упорядоченный тест продолжит работать независимо от того, закончатся ли один или несколько входящих в его состав тестов неудачей. Если флаг Continue after failure не будет выставлен, то упорядоченный тест прекратит свою работу при первом же возникновении неудачного теста.

Упорядоченный тест готов для запуска. Следующий этап – запуск теста тестирующим.

В окне Test View нажмем правой кнопкой мыши по созданному нами упорядоченному тесту (OrderedTest) и выберем Run Selection (или нажмем в окне Test View на кнопку).

Откроется окно Test Results, в котором после выполнения упорядоченного теста отобразятся результаты его выполнения Passed или Failed

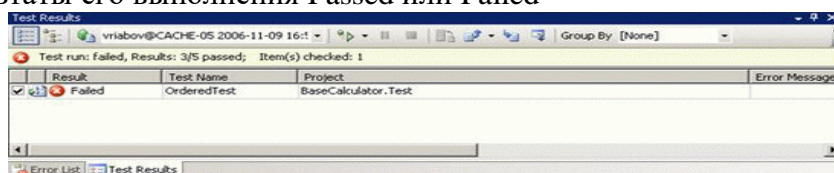


Рисунок 46

Чтобы увидеть результаты выполнения каждого из тестов, входящих в упорядоченный тест, в окне Test Results нужно щелкнуть два раза на строке для упорядоченного теста. Эти результаты появятся в окне OrderedTest [Results].

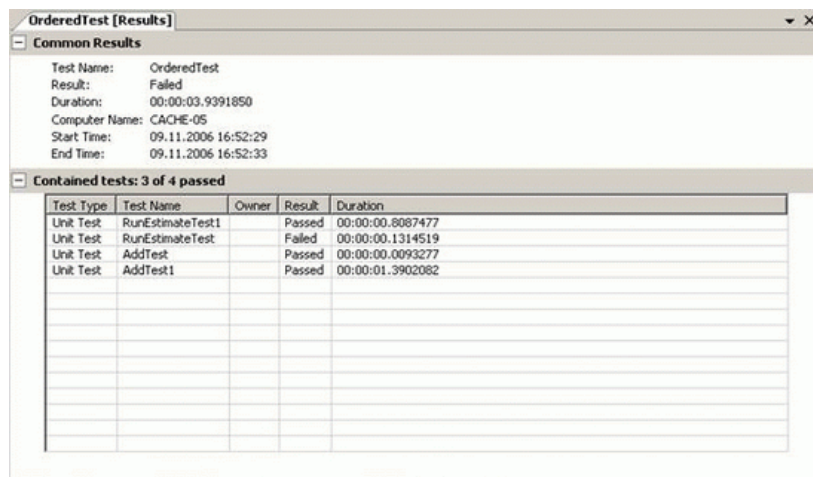


Рисунок 47

Чтобы увидеть детальные результаты отдельных тестов, нужно щелкнуть два раза на них в окне OrderedTest [Results].

Замечание. Если бы перед запуском теста не был выставлен флаг Continue after failure, то упорядоченный тест прекратил бы свою работу при первом же возникновении неудачного теста.

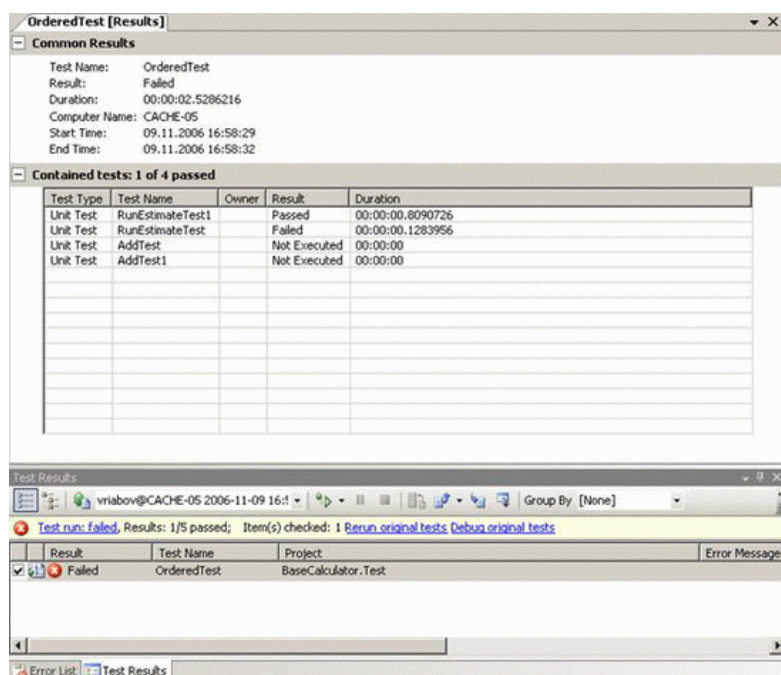


Рисунок 48

Замечание. Результаты выполнения тестов можно экспортировать в отдельный файл. Для этого в окне Test Result надо выбрать Export Test Run Results и указать имя и местоположение файла.

Программа

Будут выданы исходные тексты программы BaseCalculatorNew для тестирования методом "белого ящика" средствами MVSTE и пример тестового драйвера.

Написать тесты для методов работы с памятью калькулятора, используя упорядоченные тесты (ordered Tests) и расположив тесты в таком порядке, чтобы как можно реже проводить подготовку тестового окружения. Обосновать свой выбор.

Практическая работа № 2.21. Интеграционное тестирование в MVSTE

Цель работы: провести интеграционное тестирование в MVSTE

Интеграционное тестирование

Задачи и цели интеграционного тестирования

Результатом тестирования и верификации отдельных модулей, составляющих программную систему, является заключение о том, что эти модули являются внутренне непротиворечивыми и соответствуют требованиям. Однако отдельные модули редко функционируют сами по себе, поэтому следующая задача после тестирования отдельных модулей – тестирование корректности взаимодействия нескольких модулей, объединенных в единое целое. Такое тестирование называют интеграционным. Его цель – удостовериться в корректности совместной работы компонент системы.

Интеграционное тестирование называют еще тестированием архитектуры системы. С одной стороны, это название объясняется тем, что, интеграционные тесты включают в себя проверки всех возможных видов взаимодействий между программными модулями и элементами, которые определяются в архитектуре системы, – таким образом, интеграционные тесты проверяют полноту взаимодействий в тестируемой реализации системы. С другой стороны, результаты выполнения интеграционных тестов – один из основных источников информации для процесса улучшения и уточнения архитектуры системы, межмодульных и межкомпонентных интерфейсов. Т.е. с этой точки зрения интеграционные тесты проверяют корректность взаимодействия компонент системы.

В результате проведения интеграционного тестирования и устранения всех выявленных дефектов получается согласованная и целостная архитектура программной системы, т.е. можно считать, что интеграционное тестирование – это тестирование архитектуры и низкоуровневых функциональных требований.

Интеграционное тестирование, как правило, представляет собой итеративный процесс, при котором проверяется функциональность все более и более увеличивающейся в размерах совокупности модулей.

Задачи и цели интеграционного тестирования

Структурная классификация методов интеграционного тестирования

Как правило, интеграционное тестирование проводится уже по завершении модульного тестирования для всех интегрируемых модулей. Однако это далеко не всегда так. Существует несколько методов проведения интеграционного тестирования:

- восходящее тестирование;
- монолитное тестирование;
- нисходящее тестирование;

Восходящее тестирование. При использовании этого метода подразумевается, что сначала тестируются все программные модули, входящие в состав системы, и только затем они объединяются для интеграционного тестирования. При таком подходе значительно упрощается локализация ошибок: если модули протестированы по отдельности, то ошибка при их совместной работе есть проблема их интерфейса. Тогда область поиска проблем у тестирующего становится достаточно узкой, а поэтому гораздо выше вероятность правильно идентифицировать дефект.

Однако у восходящего метода тестирования есть существенный недостаток – необходимость в разработке драйвера и заглушек для модульного тестирования перед проведением интеграционного тестирования и необходимость в разработке драйвера и заглушек при интеграционном тестировании части модулей системы.

С одной стороны, драйверы и заглушки – мощный инструмент тестирования, с другой – их разработка требует значительных ресурсов, особенно при изменении состава интегрируемых модулей. Т.е. может потребоваться один набор драйверов для модульного тестирования каждого модуля, отдельный драйвер и заглушки для тестирования интеграции двух модулей из набора, отдельный – для тестирования интеграции трех модулей и т.п. В первую очередь, причина в том, что при интеграции модулей отпадает необходимость в некоторых заглушках, а также требуется изменение драйвера, которое будет поддерживать новые тесты, затрагивающие несколько модулей.

Монолитное тестирование предполагает, что отдельные компоненты системы серьезного тестирования не проходили. Основное преимущество данного метода – отсутствие необходимости в разработке тестового окружения, драйверов и заглушек. После разработки всех модулей выполняется их интеграция, затем система проверяется вся в целом, как она есть. Этот подход не следует путать с системным тестированием. Несмотря на то, что при монолитном тестировании проверяется работа всей системы в целом, основная задача этого тестирования – определить проблемы взаимодействия отдельных модулей системы. Задачей же системного тестирования является оценка качественных и количественных характеристик системы с точки зрения их приемлемости для конечного пользователя.

Тем не менее, монолитное тестирование имеет ряд серьезных недостатков:

- очень трудно выявить источник ошибки (идентифицировать ошибочный фрагмент кода);
- трудно организовать исправление ошибок;
- процесс тестирования плохо автоматизируется.

Нисходящее тестирование предполагает, что процесс интеграционного тестирования движется следом за разработкой. Сначала при нисходящем подходе тестируют только самый верхний управляющий уровень системы, без модулей более низкого уровня. Затем постепенно с более высокоуровневыми модулями интегрируются более низкоуровневые. В результате применения такого метода отпадает необходимость в драйверах (роль драйвера выполняет более высокоуровневый модуль системы), однако сохраняется нужда в заглушках.

Временная классификация методов интеграционного тестирования

На практике чаще всего в различных частях проекта применяются все рассмотренные в предыдущем разделе методы в совокупности. Каждый модуль тестируют по мере готовности отдельно, а потом включают в уже готовую композицию. Для одних частей тестирование получается нисходящим, для других – восходящим. В связи с этим представляется полезным рассмотреть еще один тип классификации типов интеграционного тестирования – классификацию по частоте интеграции:

- тестирование с поздней интеграцией;
- тестирование с постоянной интеграцией;
- тестирование с регулярной или послойной интеграцией.

Тестирование с поздней интеграцией – практически полный аналог монолитного тестирования. Интеграционное тестирование при такой схеме откладывается на как можно более поздние сроки проекта. Этот подход оправдывает себя в том случае, если система представляет собой конгломерат слабо связанных между собой модулей, которые взаимодействуют по какому-либо стандартному интерфейсу, определенному вне проекта (например, в случае, если система состоит из отдельных Web-сервисов).

Схематично тестирование с поздней интеграцией может быть изображено в виде цепочки R-C-V-R-C-V-R-C-V-I-R-C-V-R-C-V-I, где R – разработка требований на отдельный модуль, C – разработка программного кода, V – тестирование модуля, I – интеграционное тестирование всего, что было сделано раньше.

Тестирование с постоянной интеграцией подразумевает, что как только разрабатывается новый модуль системы, он сразу же интегрируется со всей остальной системой. Тесты для этого модуля проверяют как сугубо его внутреннюю функциональность, так и его взаимодействие с остальными модулями системы. Таким образом, этот подход совмещает в себе модульное тестирование и интеграционное. Разработки заглушек при таком подходе не требуется, но может потребоваться разработка драйверов. В настоящее время именно этот подход называют *unit testing*, несмотря на то, что в отличие от классического модульного тестирования здесь не проверяется функциональность изолированного модуля. Локализация ошибок межмодульных интерфейсов при таком подходе несколько затруднена, но все же значительно ниже, чем при монолитном тестировании. Большая часть таких ошибок выявляется достаточно рано именно за счет частоты интеграции и за счет того, что за одну итерацию тестирования проверяется сравнительно небольшое число межмодульных интерфейсов.

Схематично тестирование с постоянной интеграцией может быть изображено в виде цепочки R-C-I-R-C-I-R-C-I, в которой фаза тестирования модуля намеренно опущена и заменена на тестирование интеграции.

При тестировании с регулярной или послойной интеграцией интеграционному тестированию подлежат сильно связанные между собой группы модулей (слои), которые затем также интегрируются между собой. Такой вид интеграционного тестирования называют также иерархическим интеграционным тестированием, поскольку укрупнение интегрированных частей системы, как правило, происходит по иерархическому принципу. Однако, в отличие от нисходящего или восходящего тестирования, направление прохода по иерархии в этом подходе не задано.

Таблица 18

| | Восходяще | Нисходяще | Монолитное | Поздняя интеграция | Постоянная интеграция | Регулярная интеграция |
|--------------------|-------------------------------------|----------------------------------|----------------------------------------|----------------------------------------|----------------------------------|----------------------------------|
| Время интеграции | поздно (после тестирования модулей) | рано (параллельно с разработкой) | поздно (после разработки всех модулей) | поздно (после разработки всех модулей) | рано (параллельно с разработкой) | рано (параллельно с разработкой) |
| Частота интеграции | Редко | часто | редко | редко | часто | часто |
| Нужны ли драйверы | Да | нет | нет | нет | да | да |
| Нужны ли заглушки | Да | да | нет | нет | нет | да |

Таблица 18 представляет основные характеристики рассмотренных выше видов интеграционного тестирования. Время интеграции характеризует момент времени, когда проводится первое интеграционное тестирование и все последующие, частота интеграции – насколько часто при разработке выполняется интеграция. Необходимость в драйверах и заглушках определена в последних двух строках таблицы.

На примере "Калькулятора"

Как уже отмечалось, в MVSTE под unit-testing подразумевается именно интеграционное тестирование, а конкретно — тестирование с постоянной интеграцией. В "Автоматизация модульного тестирования" мы уже протестировали метод RunEstimate(), при интеграции классов AnalizerClass и CalcClass. Аналогично, составив требования к этой подсистеме из двух классов (а это будут требования ко всем методам AnalizerClass), можно провести следующий этап тестирования. В качестве примера протестируем все методы такой подсистемы, сделав по одному тестовому примеру на каждый метод:

```

/// <summary>
/// A test for RunEstimate ()
/// Проверяем, что, если в стеке находится корректное выражение, представленное
обратной польской записью, то
/// метод RunEstimate правильно посчитает это выражение
/// </summary>
[DeploymentItem("BaseCalculator.exe")]
[TestMethod()]
public void RunEstimateTest()
{
    string expected = "3";
    string actual;

```

```

        TestProjectCalculator.BaseCalculator_AnalaizerClassAccessor.opz =
            new ArrayList();
        ArrayList _opz =
            TestProjectCalculator.BaseCalculator_AnalaizerClassAccessor.opz;
        _opz.Add("7");
        _opz.Add("8");
        _opz.Add("+");
        _opz.Add("5");
        _opz.Add("/");

        actual
TestProjectCalculator.BaseCalculator_AnalaizerClassAccessor.RunEstimate();

        Assert.AreEqual(expected, actual,
            "BaseCalculator.AnalaizerClass.RunEstimate did not return the expected
value.");
    }

    /// <summary>
    /// A test for CheckCurrency ()
    /// Проверяет, что, если в выражении нарушена скобочная структура, то метод
возвращает false
    /// </summary>
    [DeploymentItem("BaseCalculator.exe")]
    [TestMethod()]
    public void CheckCurrencyTest()
    {
        bool expected = false;
        bool actual;

        TestProjectCalculator.BaseCalculator_AnalaizerClassAccessor.expression =
            "((5+3)*2-(3*10))-2)+((5+3)";
        actual
TestProjectCalculator.BaseCalculator_AnalaizerClassAccessor.CheckCurrency();

        Assert.AreEqual(expected, actual,
            "BaseCalculator.AnalaizerClass.CheckCurrency did not return the expected
value.");

        Assert.AreEqual(18,
            TestProjectCalculator.BaseCalculator_AnalaizerClassAccessor.erposition,
            "BaseCalculator.AnalaizerClass.CheckCurrency did not return the expected
value.");
    }

    /// <summary>
    /// A test for CreateStack ()
    /// Проверяет, что если отформатированное выражение равно
    /// "( ( 5 + 3 ) * 2 - ( 3 * 10 ) ) - 2 + ( 5 + 3 ) ",
    /// то стек содержит следующие элементы "5 3 + 2 * 3 10 * - 2 - 5 3 + + "
    /// </summary>
    [DeploymentItem("BaseCalculator.exe")]
    [TestMethod()]
    public void CreateStackTest()

```

```

{
    string expected = "5 3 + 2 * 3 10 * - 2 - 5 3 + + ";
    ArrayList actual;

    TestProjectCalculator.BaseCalculator_AnalaizerClassAccessor.expression =
        "( ( 5 + 3 ) * 2 - ( 3 * 10 ) ) - 2 + ( 5 + 3 ) ";

    actual = TestProjectCalculator.BaseCalculator_AnalaizerClassAccessor.Create
    Stack();

    string actualstr = "";
    foreach (object obj in actual)
    {
        actualstr += obj.ToString() + " ";
    }
    Assert.AreEqual(expected, actualstr,
        "BaseCalculator.AnalaizerClass.CreateStack did not return the expected
value.");
}

/// <summary>
/// A test for Estimate ()
/// Проверяет, что, если выражение, которое необходимо проверить, равно
/// ((5+3)*2-(3*10))-2+(5+3), то метод вернет его значение,
/// равное -8
/// </summary>
[DeploymentItem("BaseCalculator.exe")]
[TestMethod()]
public void EstimateTest()
{
    string expected = "-8";
    string actual;

    TestProjectCalculator.BaseCalculator_AnalaizerClassAccessor.expression =
        "((5+3)*2-(3*10))-2+(5+3)";
    actual = TestProjectCalculator.BaseCalculator_AnalaizerClassAccessor.Estimate();

    Assert.AreEqual(expected, actual,
        "BaseCalculator.AnalaizerClass.Estimate did not return the expected value.");
}

/// <summary>
/// A test for Format ()
/// Проверяет, что если выражение равно "
/// ((5+3)*2-(3 *10))-2+ (5+3)", то метод отформатирует его к виду:
/// "( ( 5 + 3 ) * 2 - ( 3 * 10 ) ) - 2 + ( 5 + 3 ) "
/// </summary>
[DeploymentItem("BaseCalculator.exe")]
[TestMethod()]
public void FormatTest()
{

```

```

string expected = "( ( 5 + 3 ) * 2 - ( 3 * 10 ) ) - 2 + ( 5 + 3 ) ";
string actual;
TestProjectCalculator.BaseCalculator_AnalaizerClassAccessor.expression =
    "((5+3)*2-(3 *10))-2+ (5+3)";
actual = TestProjectCalculator.BaseCalculator_AnalaizerClassAccessor.Format();

Assert.AreEqual(expected, actual,
    "BaseCalculator.AnalaizerClass.Format did not return the expected value.");
}

```

Запустив тесты, можно убедиться в корректной (для данных тестовых примеров) работе методов.

Замечание. Тестовых требований, как и функциональных, к методам нет, однако их можно составить по системным требованиям и описанию методов.

Как уже отмечалось в "Модульное тестирование", эти методы обладают многими недостатками, и некоторые из них генерируют исключения на некорректных входных данных. Это, в принципе, логично, так как запуск метода RunEstimate() подразумевает, что входное выражение уже обработано в методах CheckCurrency(), Format(), CreateStack(), а отдельно от них этот метод вызываться не будет. Для этого необходимо сделать уровень доступа к ним private и протестировать их только на корректных данных. Основное же внимание стоит уделить методу Estimate(), который поочередно запускает все вышеперечисленные методы и имеет уровень доступа public. Один из тестов для него приведен выше.

После того, как такая система из двух модулей протестирована, переходим к тестированию всей системы, присоединив последние модули.

Замечание. Модуль BaseCalc, который отвечает за пользовательский интерфейс, мы не тестируем, так как это выходит за рамки курса.

Теперь можно использовать системные требования, для тестирования программы в целом. Проведем тесты для метода Main(string[] args), так как это единственный метод, не считая уже протестированного Estimate(), который не относится к графическому интерфейсу и с которым работают пользователи:

```

/// <summary>
    ///A test for Main (string[])
    /// Для чисел, каждое из которых меньше либо равно MAXINT и больше либо равно
MININT,
    /// функция суммирования должна возвращать правильную сумму с точки зрения
математики
    ///</summary>
    [DeploymentItem("BaseCalculator.exe")]
    [TestMethod]
    public void MainTest()
    {
        string[] args = new string[1]; // TODO: Initialize to an appropriate value
        args[0] = "2+2";

        int expected = 0;
        int actual;

        actual = TestProjectCalculator.BaseCalculator_ProgramAccessor.Main(args);
        Assert.AreEqual(expected, actual,
            "BaseCalculator.Program.Main did not return the expected value.");
    }

    /// <summary>

```

```

    ///A test for Main (string[])
    /// Для чисел, сумма которых больше чем MAXINT и меньше чем MININT,
    /// а также в случае, если любое из слагаемых больше чем MAXINT или меньше чем
MININT,
    /// программа должна выдавать ошибку Error 06
    ///</summary>
    [DeploymentItem("BaseCalculator.exe")]
    [TestMethod()]
    public void MainTest1()
    {
        string[] args = new string[1]; // TODO: Initialize to an appropriate value
        args[0] = "2711477380+1000000";

        int expected = 6;
        int actual;

        actual = TestProjectCalculator.BaseCalculator_ProgramAccessor.Main(args);

        Assert.AreEqual(expected, actual,
            "BaseCalculator.Program.Main did not return the expected value.");
    }

```

И так далее. Таким образом, проверив методы Main() и Estimate(), а также некоторые методы визуального интерфейса, можно убедиться в соответствии системы требованиям или, наоборот, обнаружить какие-то ошибки в межмодульном взаимодействии.

Программа

Будут выданы исходные тексты программы BaseCalculatorNew для тестирования методом "белого ящика" средствами MVSTE и пример тестового драйвера.

Составить тест-план и провести интеграционное тестирование (средствами MVSTE) методов Main() и Estimate().

Практическая работа № 2.22. Тестирование в Microsoft Solutions Framework

Цель работы: провести тестирование в Microsoft Solutions Framework

Тест

В каждом тестовом задании может быть несколько вариантов ответа. После проведения теста студенты могут попробовать обосновать свои неверные ответы.

1. При использовании какого метода интеграционного тестирования сначала все программные модули, входящие в состав системы, тестируются и только затем объединяются для интеграционного тестирования?

1. восходящего
2. монолитного
3. нисходящего
4. с поздней интеграцией
5. с постоянной интеграцией
6. с регулярной интеграцией

Ответ: 1

2. При использовании какого метода интеграционного тестирования подразумевается, что, как только разрабатывается новый модуль системы, он сразу же интегрируется со всей остальной системой?

1. восходящего
2. монолитного
3. нисходящего
4. с поздней интеграцией
5. с постоянной интеграцией

6. с регулярной интеграцией

Ответ: 5

3. Для каких видов интеграционного тестирования нужен драйвер?

1. восходящего

2. монолитного

3. нисходящего

4. с поздней интеграцией

5. с постоянной интеграцией

6. с регулярной интеграцией

Ответ: 1, 5, 6

4. Для каких видов интеграционного тестирования нужны заглушки?

1. восходящего

2. монолитного

3. нисходящего

4. с поздней интеграцией

5. с постоянной интеграцией

6. с регулярной интеграцией

Ответ: 1, 3, 6

5. Для каких видов интеграционного тестирования при разработке часто выполняется интеграция?

1. восходящего

2. монолитного

3. нисходящего

4. с поздней интеграцией

5. с постоянной интеграцией

6. с регулярной интеграцией

Ответ: 3, 5, 6

Роль тестировщика в команде разработчиков ПО

В течение всего курса обсуждались и демонстрировались на примере программной системы "Калькулятор" подходы к тестированию и верификация программного обеспечения.

В конце курса мы поговорим о роли, обязанностях и задачах тестировщика в команде разработчиков программного обеспечения на примере методологии ведения проектов и разработки программного обеспечения Microsoft Solutions Framework (MSF) for Agile Software Development.

Microsoft Solutions Framework

Замечание. Подробнее о подходе MSF можно почитать по адресу <http://www.microsoft.com/rus/msdn/msf/> или по адресу <http://msdn.microsoft.com/vstudio/teamsystem/msf>

Microsoft Solutions Framework (MSF) – хорошо настраиваемый, масштабируемый, полностью интегрируемый набор процессов разработки программного обеспечения, принципов и проверенных практик, предназначенных для того, чтобы предоставить команде разработчиков программного обеспечения именно тот вид управления проектами, который им больше подходит.

MSF — это методология ведения проектов и разработки решений, которая базируется на принципах работы над продуктами как самой фирмы Microsoft, так и других компаний, работающих в области IT-индустрии.

MSF является схемой для принятия решений по планированию и реализации новых технологий в организациях. MSF включает обучение, информацию, рекомендации и инструменты для идентификации и структуризации информационных потоков бизнес-процессов и всей информационной инфраструктуры новых технологий.

Microsoft Solutions Framework представляет собой хорошо сбалансированный и гибкий набор методик организации процесса разработки, который может быть адаптирован под потребности практически любого коллектива разработчиков и проекта, вне зависимости от его

размера и сложности. MSF поддерживает самые различные подходы к организации процесса разработки, что позволяет команде разработчиков выбирать самый подходящий для них путь. Философия MSF утверждает, что не существует единой методологии разработки, которая оптимально будет соответствовать требованиям любых проектов. Но, тем не менее, любому проекту необходимо управление. MSF направлена на помощь в обеспечении этого управления. При этом MSF не налагает предписаний, а позволяет команде разработчиков настраивать предоставленные средства. Средства MSF могут быть применены по отдельности или все вместе. Главное — они позволят добиться успеха для многих типов проектов.

Главными принципами MSF можно назвать производительность, интегрируемость и расширяемость:

- **производительность.** Один из ключевых принципов MSF направлен на то, чтобы сделать команду разработчиков более производительной. Производительность в MSF поддерживается хорошо налаженным управлением процесса разработки;
- **интегрируемость.** Решения и управление представлены инструментальными средствами, посредством плавной интеграции любых наборов инструментальных средств, справки и содержания MSF. Все эти элементы легко обновляются через MSDN;
- **расширяемость.** Процесс управления и справка полностью настраиваемы в пределах MSF. Разработчики могут выбрать быстрый или более структурированный подход, каждый из которых включает в себя наборы предложенных сценариев, или определить свой собственный подход, используя эти сценарии.

MSF содержит не только рекомендации общего характера, но и предлагает адаптируемую модель коллектива разработчиков, определяющую взаимоотношения внутри коллектива, гибкую модель проектного планирования, основанного на управлении проектными группами, а также набор методик для оценки рисков.

Жизненный цикл процессов в MSF сочетает водопадную и спиральную модели разработки: проект реализуется поэтапно, с наличием соответствующих контрольных точек, а сама последовательность этапов может повторяться по спирали.



Рисунок 49

При таком подходе от водопадной модели берется простота планирования, от классической спиральной – легкость модификаций. Благодаря промежуточным контрольным точкам и обратной спирали верификации облегчается взаимодействие с заказчиком.

При управлении проектом четко ставится цель, которую необходимо достичь в результате, и учитываются ограничения, накладываемые на проект. Все виды ограничений могут быть отнесены к одному из трех видов: ограничения ресурсов, ограничения времени и ограничения возможностей. Эти три вида ограничений и приоритетность задач по их преодолению образуют треугольник приоритетов в MSF.

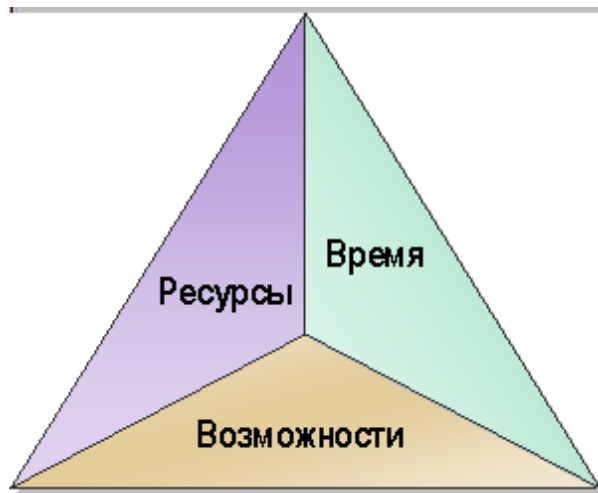


Рисунок 50

Треугольник приоритетов является основой для матрицы компромиссов – заранее утвержденных представлений о том, какие аспекты процесса разработки будут четко заданы, а какие будут согласовываться или приниматься как есть.

Microsoft выпустила среду разработки, в полной мере поддерживающей основные идеи MSF – Microsoft Visual Studio 2005 Team System. Это первый программный комплекс, представляющий собой не среду разработки для индивидуальных членов коллектива, а комплексное средство поддержки коллективной работы.



Рисунок 51

Замечание. В состав Visual Studio Team Edition входит специальная редакция для тестировщиков – Team Edition for Software Testers, с которой мы и работали на протяжении всего курса.

Также Visual Studio 2005 Team System включает в себя два шаблона методологий MSF, которые можно применять "как есть", настраивать под соответствие своим собственным потребностям или использовать как основу для создания своего собственного подхода для организации и управления процессом разработки программного обеспечения:

- MSF for Agile Software Development
- MSF for CMMI® Process Improvement

Тестировщик в MSF for Agile Software Development

MSF for Agile Software Development

Microsoft Solutions Framework (MSF) for Agile Software Development — управляемый сценариями, основанный на окружении, быстрый процесс разработки программного обеспечения для создания .NET и других объектно-ориентированных приложений. MSF for Agile Software Development включает в себя подходы по управлению требованиями к качеству, такими, как требования к оборудованию или требования к безопасности. На

основании окружения MSF помогает решить, как управлять проектом. Этот подход помогает создавать адаптивный процесс, который, преодолевая трудности большинства быстрых процессов разработки программного обеспечения, достигает целей, поставленных при создании проекта перед командой разработчиков.

О ролях

В MSF for Agile Software Development собрана вместе команда равных разработчиков, которая обеспечивает полный набор необходимых составляющих, связанных с созданием, использованием и обслуживанием создаваемого продукта. Каждый член команды, или роль, отвечает за удовлетворения нужд своей клиентуры, причем ни один клиент не является важнее другого. MSF for Agile Software Development содержит все необходимые методики и подходы для уверенности в том, что команда разработчиков принимает правильные решения.



Рисунок 52

Подробную информацию об MSF for Agile Software Development и ролях в этом подходе можно найти на сайте Microsoft в соответствующем разделе.

Наша цель заключается в том, чтобы рассказать о роли тестировщика в команде разработчиков, работающих по этому подходу.

О роли тестировщика

Главная задача тестировщика — обнаружить и сообщить о проблемах программного продукта, которые могут сказаться на качестве. Ключевая задача тестировщика — найти и сообщить о существенных ошибках в продукте, протестировав его. Также в обязанности тестировщика входит точное сообщение о проявлениях ошибки и описание какого-либо обходного пути для уменьшения этих проявлений. Тестировщик описывает ошибки, а также простые в понимании и выполнении шаги для устранения этих ошибок. Он участвует в группе по разработке стандартов качества продукта. Цель тестирования состоит в том, чтобы доказать, что известные функции работают правильно, и обнаружить новые проблемы продукта.

Далее будет описана последовательность работы тестировщика по подходу MSF for Agile Software Development.

Проверка сценария

Сценарий — тип рабочего элемента, описывающий действия пользователя в системе для достижения им определенной цели. Таким образом, в сценарии записаны шаги, которые необходимо сделать пользователю для достижения определенной цели. При этом не обязательно, чтобы сценарий описывал успешные пути достижения цели — вполне могут быть сценарии, описывающие шаги, которые не приведут к желаемому результату. С другой стороны, достичь одну и ту же цель в системе можно несколькими путями.

Сценарий разделен на задачи тестирования. Эти задачи назначены тестировщикам для разработки и выполнения тестовых ситуаций. Назначение сценария для тестирования означает, что необходимые функциональные возможности были включены в текущую сборку и готовы к тестированию. Проверка того, что сборка содержит функциональность, описанную в сценарии, требует понимания сценария и его граничных условий. Валидационные тесты разрабатываются для того, чтобы покрыть все функциональные возможности и граничные условия сценария. Все ошибки, возникающие в процессе тестирования, сохраняются в виде соответствующего рабочего элемента.

Для проверки сценария, в первую очередь, нужно определить подход к тестированию.

Подход к тестированию – это стратегия, определяющая разработку и выполнение тестов. Он также определяет качественные рекомендации по нагрузке на программный продукт. Подход к тестированию является отправной точкой для тест-плана на ранней стадии проекта, но развивается и изменяется он вместе с проектом. Подход к тестированию – объединение методик, в частности, методик по ручному и автоматизированному тестированию. Перед каждой итерацией документ, описывающий подход к тестированию, должен быть обновлен, чтобы отразить цели тестирования и тестовых данных, которые используются на новой итерации.

Для определения подхода для тестирования необходимо, чтобы сценарий был написан и утвержден.

Если это сделано, то выполняются следующие действия.

1. Определение окружения проекта, т.е. выявление уникальных проектных рисков и пользователей, на которых это может повлиять; выявление специальных ситуаций, которые могут повлиять на тестирование; учет влияния рисков; определение того, что произойдет, если возникнет ошибка.

2. Определение тестовой миссии, т.е. определение проектных целей, которые необходимо удовлетворить в течение тестирования; консультация с проектировщиком и бизнес-аналитиком по техническим неясностям и пользовательским рискам; сотрудничество с бизнес-аналитиком и проектировщиком для создания списка приоритетных технических неясностей и пользовательских рисков.

3. Оценивание возможных технологий, т.е. оценивание доступных инструментальных средств для тестирования; оценивание навыков команды тестировщиков; определение тестовых технологий, возможных и соответствующих проекту, основанных на доступных инструментах и навыках.

4. Определение показателей тестирования, т.е. работа с разработчиками для определения реалистичных порогов показателей покрытия кода для тестируемых модулей; использование тестового окружения, цели тестирования и технологии тестирования для определения тестовых показателей; определение того, что тестовые показатели будут часто включать в себя пороги для различных типов текстов (Web, нагрузочные, стрессовые) или процент автоматизированных тестов; работа с бизнес-аналитиком для определения реалистичных показателей порогов нагрузки продукта; добавление тестовых показателей к соответствующим отчетам и в документ, описывающий подход к тестированию; опубликование подхода к тестированию.

В результате выработки подхода к тестированию показатели тестов определены и будут являться обязательными при модульном тестировании.

Далее необходимо написать валидационные тесты.

Валидационные тесты проверяют, что система выполняет функции, описанные в сценарии. Они разрабатываются как тесты "чёрного ящика" приложения и проверяют области, наиболее важные для конечных пользователей. При разработке этих тестов не учитывается структура исследуемого кода. В качестве тестов используют тестовые ситуации, которые повторяют действия, выполняемые реальными пользователями.

Для создания валидационных тестов необходимо, чтобы сценарий был написан и утвержден, были назначены задачи тестирования сценария и определена область, проверяемая сценарием.

Если это сделано, то выполняются следующие действия.

1. Определение тестируемой области и среды, т.е. изолирование тестируемой области (тесты могут быть запущены как часть итерационных тестов, если они автоматизированы).

2. Определение деталей выполнения тестовых примеров, т.е. определение тестовых данных, необходимых для тестового примера; проверка документа "Подход к тестированию"; добавление тестовых данных к соответствующей итерационной части документа; определение всех ограничений для тестовых примеров, вызываемых в тестовых заданиях; определение всех граничных условий для тестовых примеров, вызываемых в тестовых заданиях; определение всех граничных условий для тестового примера; определение, могут ли тестовые примеры быть автоматическими; определение шагов для выполнения сценария.

3. Реализация тестовых примеров, т.е. создание папки для тестирования сценария, если она ещё не создана; написание документации для ручных тестовых примеров; написание автоматизированных тестовых примеров для итерационных тестов; размещение тестовых примеров в папку тестирования сценария; добавление всех тестов, определенных как итерационные, в папку итерационных тестов; сохранение тестов в версионном контроле и в документе, описывающим подход к тестированию.

В результате написания валидационных тестов все граничные условия и вся функциональность тестового задания будут покрыты.

Далее необходимо произвести выбор и запуск тестового примера.

Выбор и запуск тестового примера. Наиболее важная часть тестирования — выполнение тестов для текущей сборки. Как только тест-кейс выполнен, важно записать результаты по отношению к сценарию или требования к качеству.

Для выбора и запуска тестовой ситуации необходимо, чтобы тестовые примеры для тестового задания были написаны и была доступна сборка с необходимой функциональностью.

Если это сделано, то выполняются следующие действия.

1. Определение проводимых тестов, т.е., определение и уделение первостепенного внимания запускаемым тестам в соответствии с кратким обзором тестового плана в документе подхода к тестированию; проверка заметок сборки, чтобы убедиться, что тестируемая функциональность доступна.

2. Определение тестовых конфигураций, т.е. обнаружение и установление тестовых конфигураций, необходимых при тестировании (включает в себя аппаратные средства, программное обеспечение и тестовые данные).

3. Получение сборки, т.е. извлечение сборки с сервера, обеспечивающего версионный контроль.

4. Запуск тестирования, т.е. выбор тестовой папки для сценария или требования к качеству; выбор тестов для запуска; выполнение каждого шага теста, если это ручной тест; запуск теста, если он автоматизированный; использование тестовых данных из документа о подходе к тестированию текущей итерации.

5. Анализ тестовых результатов, т.е. присоединение сценария или требования к качеству к результатам тестирования; сравнение результатов теста с ожидаемыми результатами; выявление ошибок, если результаты не отвечают ожиданиям: если все тесты сценария или требования к качеству пройдены, закрытие рабочего элемента.

6. Закрытие рабочих элементов, т.е., если все тесты из тестового задания выполнены, закрытие рабочего элемента; уведомление владельца сценария, что нет заблокированных элементов.

В результате выбора и запуска тестовой ситуации тестовое задание будет закрыто, потому что все тесты пройдены, и будут занесены ошибки при непредвиденном поведении системы.

Далее необходимо произвести выявление ошибки.

Выявление ошибки. Всегда нужно проверять, не обнаружена ли уже проблема, перед тем как создавать новую ошибку. Необходимо выполнить шаги для воспроизведения ошибки, таким образом, чтобы она могла быть исследована. В отчет всегда нужно включать как можно больше деталей для помощи команде в определении лучшего способа разрешения ошибки. У каждой выявленной ошибки должен быть ответственный за ее исправление.

Этот этап производится, если тестирование выявило неожиданное поведение системы.

Если это так, то выполняются следующие действия.

1. Определение, не обнаружена ли аналогичная ошибка, т.е. прежде чем создать новую ошибку, проверить БД системы отслеживания ошибок, и убедиться, что проблема еще не идентифицирована; если проблема уже была идентифицирована и ошибка обнаружена, обновить форму отслеживания ошибки соответствующим образом; если проблема уже опубликована и ошибка исправлена, изменить состояние рабочего элемента ошибки с "Исправлено" на "Найдено заново"; добавить новую информацию и документировать новые шаги для воспроизведения ошибки; документировать номер сборки, где было идентифицировано некорректное поведение.

2. Регистрация новой ошибки, т.е., если ошибка не опубликована и может быть воспроизведена, ввести шаги для воспроизведения ошибки и детали результатов выполнения этих шагов в рабочем документе; включить столько деталей, сколько возможно, чтобы помочь команде, ответственной за приоритеты ошибок, понять проблему; там, где это применимо, прикрепить тестовые ситуации, результаты, конфигурации, сценарий или требования к качеству к отчету по ошибке.

3. Назначение ошибки, т.е. определение главной области проблемы; если более чем одна область может быть задействована, то выбрать одну из них; назначение ошибки на владельца затронутой области приложения; сохранение ошибки в базе данных отслеживания ошибок.

В результате этих действий ошибка будет должным образом введена в систему отслеживания ошибок и комментарии будут содержать информацию, необходимую для её исправления, или ошибка будет назначена соответствующим образом..

Далее необходимо произвести исследовательское тестирование.

Исследовательское тестирование – систематический способ проверить продукт без predetermined набора тестов. Существует множество эвристик, которые могут быть применены к проведению исследовательского тестирования. Эти эвристики включают в себя использование ролей, характеристики, переменный анализ, область поиска и тестирование различных состояний. Эвристика, обеспеченная этим руководством, описывает, как продукт протестирован с точки зрения определенной роли с целью создания новых требований. Для того, чтобы выполнить исследовательское тестирование таким образом, необходимо выбрать роль и работать через функциональные возможности системы, пытаясь достичь определённых целей. Если новые цели достигнуты или функциональные возможности не способны удовлетворить потребности роли, добавить новые или модифицировать существующие сценарии для удовлетворения этих потребностей. Лимит сессий исследовательского тестирования – не более двух часов.

Этот этап производится, если сборка готова к тестированию на новые требования или ошибки и роли опубликованы на портале проекта..

Если это выполнено, то выполняются следующие действия.

1. Установление длины сессии, т.е. установление периода времени для сессии исследовательского тестирования (этот период должен быть не меньше, чем полчаса, но не больше, чем два часа); постановка цели сессии; запуск и журналирование сессии.

2. Проверка ожидания для роли, т.е. выбор роли из списка опубликованных ролей; осуществление прохода по существующим функциональным возможностям, чтобы убедиться, что они соответствуют ожиданиям.

3. Предположение новых целей, т.е. исследование возможности роли; если реализация возможностей сложна для выполнения, определение новой ошибки или добавление нового сценария или требования к качеству в лист сценариев; изменение

существующих сценариев или требований к качеству; поиск новых целей или пропущенных функциональных возможностей, необходимых для данной версии продукта; добавление новых сценариев или требований к качеству входимостей в лист сценариев.

В результате этих действий новые сценарии и/или требования будут добавлены для отражения нового понимания системы.

После выполнения всех описанных выше действий закончится проверка сценария, в результате которой можно утверждать, что все валидационные тесты были запущены и все ошибки результатов были опубликованы.

Тестирование требований к качеству

Документы, описывающие требования к качеству системы, характеризуют такие качества системы, как максимальная нагрузка, доступность, надежность и сопровождаемость. Эти требования обычно принимают форму ограничений на то, как система должна работать.

Назначение требований к качеству для тестирования показывает, что сборка готова к тестированию. Во многих случаях сценарии присоединены к требованиям к качеству, чтобы показать области, которые будут проверяться. Тестирование требований к качеству требует, чтобы тесты на устройства, безопасность и нагрузку были завершены и ни один не был заблокирован. По результатам тестов, создаются отчеты для документирования обнаруженных проблем.

Для тестирования требований к качеству, в первую очередь, нужно определить подход к тестированию аналогично тому, как это делалось при тестировании сценариев.

Далее необходимо написать тесты на производительность.

Тесты на производительность измеряют время отклика приложения и гарантируют, что приложение соответствует установленным требованиям к качеству. Для написания тестов на производительность необходимо, чтобы было назначено тестовое задание, показывающее, что необходимо проверить работу требований к качеству.

Если это сделано, то выполняются следующие действия.

1. Понимание цели теста, т.е. цель тестирования должна быть ясно определена (например, не должно быть существенного различия в отклике продукта для пользователей с разной скоростью подключения к Интернету). Выход теста должен быть ясно задокументирован и представлен в виде диапазона значений.

2. Специфицирование тестовых конфигураций, т.е. определение тестовых конфигураций; документирование любых отклонений результатов теста, которые зависят от тестовых конфигураций; документирование внешних условий, которые могут отразиться на времени тестирования.

3. Планирование тестирования, т.е. планирование тестовых условий, включая предпосылки и установленный сценарий; пересмотр сценария, чтобы определить области, где производительность критична, а где – нет.

4. Документирование тестовых шагов, т.е. перечисление тестовых шагов так, чтобы все тестировщики, которые выполняют рабочие тесты, выполняли каждый шаг одним и тем же образом; отказ перечисления тестовых шагов может отразиться в неправильном измерении работы; использование автоматизации для построения тестовых ситуаций там, где это возможно; добавление тестовых ситуаций в папку для требований к качеству и итерационных тестов; регистрирование тестов и обновление рабочего листа подхода к тестированию с любыми тестовыми данными или другими взглядами на тест.

В результате написания тестов на производительность рабочие тесты будут завершены и зарегистрированы и будет проверено, что требования к качеству, относящиеся к функционированию системы, собраны.

Далее необходимо написать тесты безопасности.

Тесты безопасности, или "тесты проникновения", используют угрозы, найденные в процессе моделирования угроз, чтобы симитировать попытку противника достигнуть определенных целей в программе. Эта форма тестирования может быть разделена на три части: исследование, идентификация недостатка и эксплуатация. Тестирование проникновения может привести к открытию новых уязвимостей, которые становятся

требованиями безопасности или ошибками. В результате тестировщики должны знать об угрозах не меньше проектировщиков. Эта форма тестирования требует специальных навыков, таких, как умение думать и действовать как противник.

Для написания тестов безопасности необходимо, чтобы была назначена тестовая задача для требования безопасности, которое выполняется на данной итерации, а модель угрозы была актуальной и опубликованной.

Если это сделано, то выполняются следующие действия.

1. Исследование точки входа, т.е. идентифицирование точки входа системы и функциональности, отвечающей за защиту программы; сбор информации из модели угрозы, для определения ожидаемых направлений нападения; расположение по приоритетам точек входа и задание им уровней доверия.

2. Идентифицирование недостатков, т.е. написание тестовых примеров, использующих прямые и частично случайные тесты, чтобы попытаться обратиться к программе; применение направленных мер, нацеленных на обход определенных мер безопасности; полуслучайные нападения могут манипулировать форматом данных или протокола, чтобы проверить граничные условия или выявить ошибки приложения.

3. Слабости к эксплоитам, т.е. добавление тестовых примеров, эксплуатирующих любые обнаруженные уязвимости, чтобы попытаться обратиться к программе; принятие во внимание количество времени, необходимого для того, чтобы воспользоваться уязвимостью; сохранение этих ручных тестов в соответствующей папке; регистрация их; добавление всех необходимых тестовых данных в документ, описывающий подход к тестированию.

В результате написания тестов безопасности тестовые примеры будут покрывать все части требований к безопасности и все необходимые тестовые данные будут добавлены в документ, описывающий подход к тестированию.

Далее необходимо выбрать и запустить тестовые примеры аналогично тому, как это делалось при тестировании сценариев.

Далее необходимо провести выявление ошибки аналогично тому, как это делалось при тестировании сценариев.

Далее необходимо произвести исследовательское тестирование аналогично тому, как это делалось при тестировании сценариев.

После выполнения всех описанных выше действий закончится тестирование требований к качеству, в результате которой можно утверждать, что все тесты были запущены и все ошибки результатов были опубликованы.

Стрессовое тестирование

Стрессовое тестирование имеет много общего с тестированием производительности, однако его основная задача – не определить производительность системы, а оценить производительность и устойчивость системы в случае, когда для своей работы она выделяет максимально доступное количество ресурсов либо когда она работает в условиях их критической нехватки. Основная цель стрессового тестирования – вывести систему из строя, определить те условия, при которых она не сможет далее нормально функционировать. Для проведения стрессового тестирования используются те же самые инструменты, что и для тестирования производительности.

Закрытие ошибок

Ошибка – рабочий элемент, который сообщает о том, что в системе существует или существовала потенциальная проблема. Цель открытия ошибки состоит в том, чтобы точно сообщить об ошибках, причем так, чтобы разработчик, ознакомившийся с ошибкой, смог понять все составляющие проблемы. Описания в сообщении об ошибке должны обеспечивать прослеживание шагов, сделанных при столкновении с ошибкой, таким образом позволяя легко её воспроизводить.

Ошибка может быть закрыта по нескольким причинам: она исправлена, относится к другому релизу, демонстрирует несоответствие, которое не удастся воспроизвести, или дублирует уже открытую ошибку. После того, как она закрыта, никакая работа по ошибке не

совершается в течение текущей итерации. Заккрытие ошибок обычно происходит после проверки исправлений.

Для закрытия ошибки необходимо, чтобы она находилась в открытом состоянии и была указана для текущей версии программы

Сначала необходимо провести исправление ошибки.

Исправление ошибки. Проверяя исправления, смотрим на то, что ошибка была корректно исправлена и её исправление не повредило другой функциональности системы. После того, как ошибка исправлена разработчиком, тестировщик должен убедиться, что тестовый пример теперь выполняется. Если это так, то ошибку можно закрывать. Иначе ошибка снова переназначается разработчику.

Для написания исправления ошибки необходимо, чтобы сборка, которая устраняет эту ошибку, и тестовый пример, который осуществляет функциональные возможности, связанные с ошибкой, были найдены.

Если это сделано, то выполняются следующие действия.

1. Попытка воспроизводить ошибку, т.е. следовать шагам выполнения, как первоначально сообщено в описании ошибки, чтобы попытаться воспроизвести ошибку.

2. Поиск неожиданного поведения системы, т.е. проводится изучение смежных функциональных возможностей и попытка найти неожиданное поведение системы, связанное с ошибкой (это особенно важно, если задача разработки не была осуществлена полностью).

3. Получение подробной информации об ошибке, т.е., если описание ошибки непонятное, запрашивать у разработчика дополнительную информацию.

4. Переназначение ошибки, т.е., если тестовый пример не выполняется, то ошибка переназначается назад разработчику.

Далее необходимо провести собственно закрытие ошибки.

Для закрытия ошибки необходимо, чтобы сборка с исправленной ошибкой была обнаружена и тест, который обнаружил ошибку, выполнен; было решено не исправлять ошибку из-за ограничений на времени или ошибка была сдублирована и не может быть воспроизведена или будет задержана к другому релизу.

Если это сделано, то выполняются следующие действия.

1. Обновление дубликата ошибки, т.е., если ошибка является дубликатом существующей, необходимо обновить описание.

2. Новая ошибка не будет исправлена, т.е., если ошибка не должна быть исправлена, обновите описание ошибки, указав причину.

3. Ошибка исправлена, т.е., если, после выполнения шагов по воспроизведению ошибки, она не повторилась, закройте ошибку; укажите в описании ошибки сборку, используемую для проверки исправления, чтобы проконтролировать описание ошибки.

В результате закрытия ошибки, она будет закрыта в системе отслеживания ошибки с описанием поведения после исправления или объяснения того, почему ошибка не была устранена.

В результате закрытия ошибка будет закрыта с соответствующим кодом причины.

Практическая работа № 2.23. Оформление документации на программные средства с использованием инструментальных средств

Практическая работа № 2.24. Оформление документации на программные средства с использованием инструментальных средств

Цель работы: Разработать комплект документации на программное средство

Программное документирование – это процесс записи информации, произведенной процессами жизненного цикла.

Процесс содержит набор действий, которые планируют, проектируют, разрабатывают, производят, редактируют, распространяют и сопровождают те документы, в которых нуждаются все заинтересованные лица, такие как менеджеры, инженеры и пользователи программного средства.

Общие требования к составу и содержанию документов, поддерживающих создание программных средств, представлены в ряде стандартов разного ранга. Состав документов широко варьируется в зависимости от класса и характеристик объекта разработки.

Существует несколько стандартов в области обеспечения документирования программных средств.

ISO 12207 - Информационные технологии. Процессы жизненного цикла программного обеспечения.

В этом стандарте документированию посвящен специальный раздел в группе вспомогательных процессов.

ISO 9000- 3 - Общее руководство качеством и стандарты по обеспечению качества.

Управлению качеством документации посвящен специальный раздел 6.2. Эти задачи отражены также в ряде разделов стандарта, непосредственно регламентирующих управление качеством сложных программных средств.

ISO 6592 – Обработка информации. Руководство по разработке документации для вычислительных систем

Главная цель этого стандарта состоит в установлении базисной структуры документации, на основе которой возможно для любого проекта обеспечить эффективное совершенствование и реализацию информационной системы, ПС или БД. В стандарте установлены руководящие принципы создания документов для информационных систем.

ISO 9294 - Информационные технологии. Руководящие положения по управлению документацией на программное обеспечение

Технический отчет этого стандарта представляет руководство по документированию ПС для руководителей, отвечающих за создание программной продукции. Руководство предназначено для помощи в управлении разработкой и эффективном документировании программных проектов.

IEEE 1063-1993 - Пользовательская документация на программное обеспечение.

+В нем представлены наиболее полно общие требования к пользовательской документации на программные средства широкого применения. Стандарт определяет минимальные требования к структуре и содержанию комплекта документов для пользователей программных продуктов.

ГОСТ 34.602—89 Информационная технология. Комплекс стандартов на автоматизированные системы. Техническое задание на создание автоматизированной системы

Настоящий стандарт распространяется на автоматизированные системы для автоматизации различных видов деятельности (управление, проектирование, исследование и т. п.), включая их сочетания, и устанавливает состав, содержание, правила оформления документа “Техническое задание на создание (развитие или модернизацию) системы”.

ГОСТ Р 51904-2002 — Программное обеспечение встроенных систем. Общие требования к разработке и документированию.

Стандарт распространяется на процессы разработки и документирования программного обеспечения встроенных систем реального времени. Стандарт распространяется на все действия, имеющие отношение к разработке программного обеспечения.

ГОСТ 19.101 – Виды программ и программных документов.

Настоящий стандарт устанавливает виды программ и программных документов для вычислительных машин, комплексов и систем независимо от их назначения и области применения. Виды программных документов и их содержание приведены в таблице 1:

Таблица 19

| Вид программного документа | Содержание программного документа |
|----------------------------|----------------------------------------|
| 1 | 2 |
| Спецификация | Состав программы и документации на нее |

| | |
|----------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Ведомость держателей подлинников | Перечень предприятий, на которых хранят подлинники программных документов |
| Текст программы | Запись программы с необходимыми комментариями |
| Описание программы | Сведения о логической структуре и функционировании программы |
| Программа и методика испытаний | Требования, подлежащие проверке при испытании программы, а также порядок и методы их контроля |
| Техническое задание | Назначение и область применения программы, технические, технико-экономические и специальные требования, предъявляемые к программе, необходимые стадии и сроки разработки, виды испытаний |
| Пояснительная записка | Схема алгоритма, общее описание алгоритма и (или) функционирования программы, а также обоснование принятых технических и технико-экономических решений |
| Эксплуатационные документы | Сведения для обеспечения функционирования и эксплуатации программы |

Порядок выполнения работы:

Оформить пояснительную записку (ПЗ) на программный продукт.

ПЗ на программное средство должна иметь следующую структуру:

- 1) Постановка задачи;
- 2) Входные и выходные данные;
- 3) Среда разработки и обоснование выбора языка программирования;
- 4) Описание алгоритма;
- 5) Описание используемых классов и методов;
- 6) Заключение.

Приложение А – Техническое задание;

Приложение Б – Технический проект;

Приложение В – Иерархия функциональных диаграмм. Диаграмма сущность-связь.

Диаграмма потоков данных

Приложение Г UML-диаграммы

Приложение Д Листинг программы

Содержание отчета: ПЗ на электронном и бумажном носителе.

