

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ
ПО ВЫПОЛНЕНИЮ ПРАКТИЧЕСКИХ РАБОТ
по дисциплине

Информационно-коммуникационные технологии
для бакалавров направления подготовки
09.03.01 «Информатика и вычислительная техника»
направленность (профиль) "Программное обеспечение средств вычислительной техники и
автоматизированных систем"

Ставрополь 2026 г.

СОДЕРЖАНИЕ

Практическая работа № 1. Выявление ошибок и причин их появления.....	4
Практическая работа № 2. Тестирование «белым ящиком».....	6
Практическая работа № 3. Тестирование «черным ящиком»	9
Практическая работа № 4. Модульное тестирование.....	12
Практическая работа № 5. Интеграционное тестирование.....	18
Практическая работа № 6. Разработка алгоритма поставленной задачи и реализация его средствами автоматизированного проектирования.....	19
Практическая работа № 7. Использование инструментальных средств на этапе отладки программного модуля	26
Практическая работа № 8. Тестирование на этапе сопровождения программного продукта	31
Практическая работа № 9. Место верификации среди процессов разработки программного обеспечения.....	36
Практическая работа № 10. Тестовые примеры. Классы эквивалентности. Ручное тестирование в MVSTE	45
Практическая работа № 11. Тестовое окружение.....	54
Практическая работа № 12. Оформление документации на программные средства с использованием инструментальных средств.....	61
Список литературы.....	63
Основные источники	63
Дополнительные источники	63

Практическая работа № 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. 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 статистике 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, 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).

Метод белого ящика (white box testing, open box testing, clear box testing, glass box testing) у тестировщика 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.

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

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

Покрывание 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 введении 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 № 3. 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 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).

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.

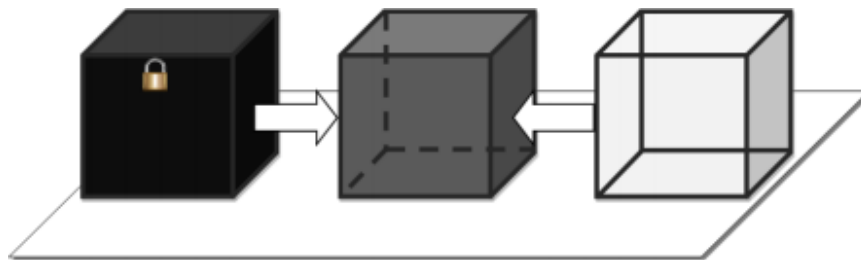


Рисунок 1

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.

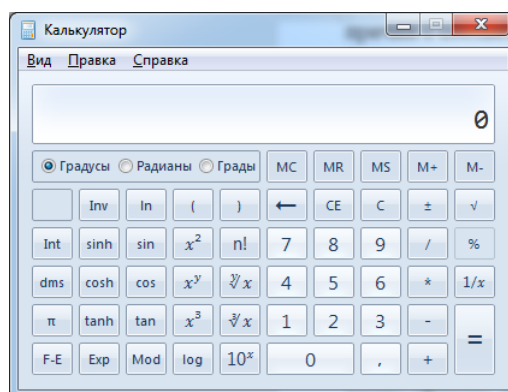


Рисунок 2

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 № 4. 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....

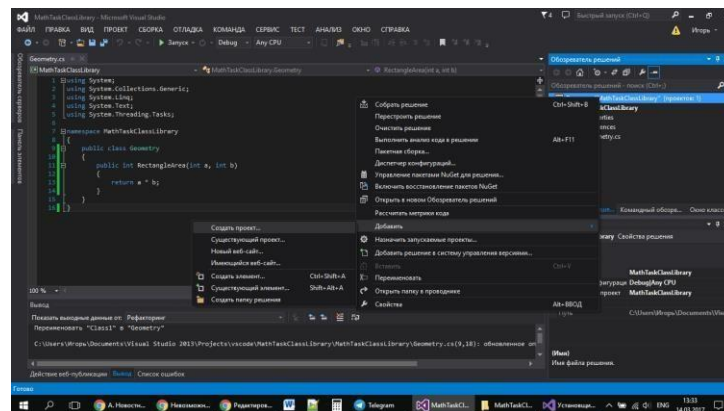


Рисунок 3

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

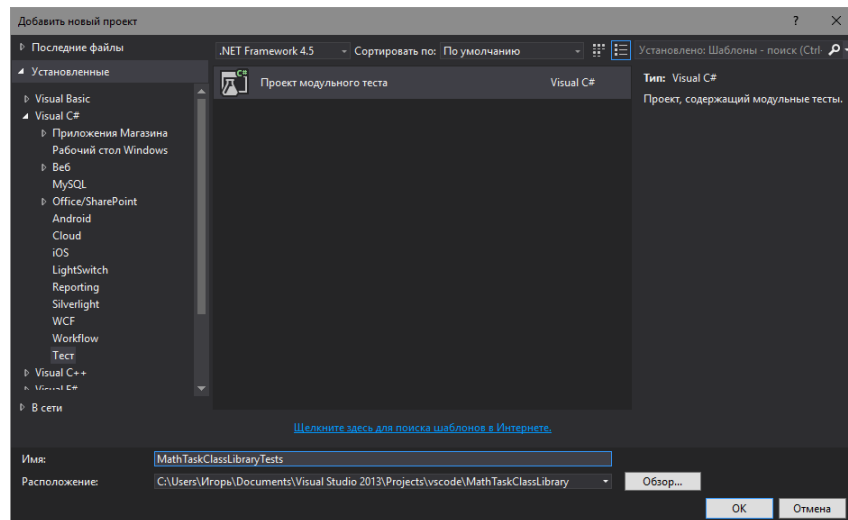


Рисунок 4

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 –OK».

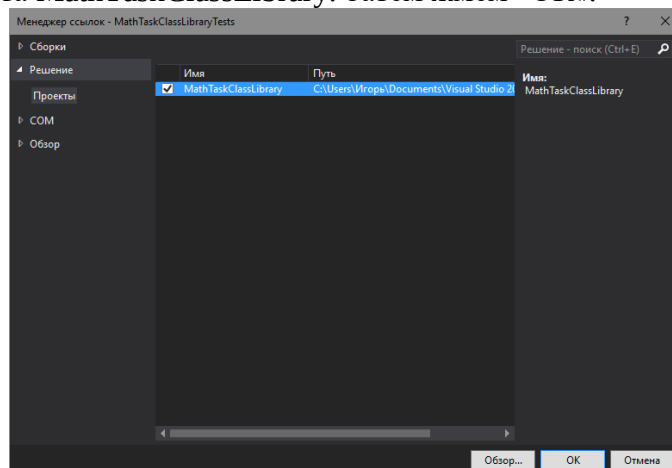


Рисунок 6

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

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

случае это число 15.

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

Тестирующий метод обычно содержит три необходимых компонента:

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

Соответственно тестирующий код будет таким:

```
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».

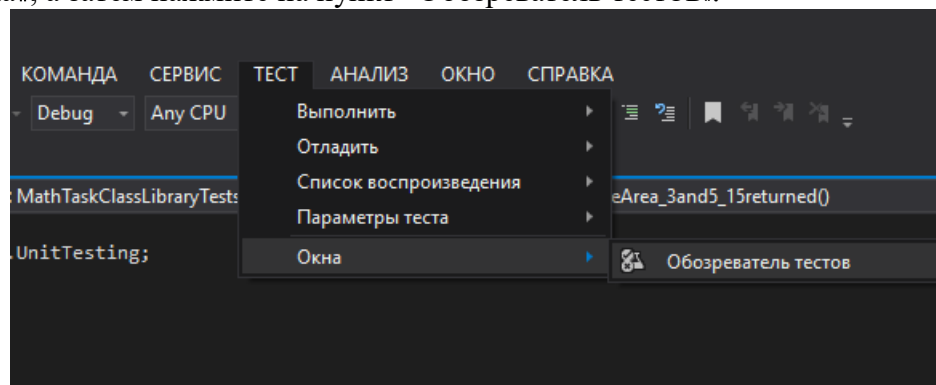


Рисунок 8

В студии появится следующее окно:

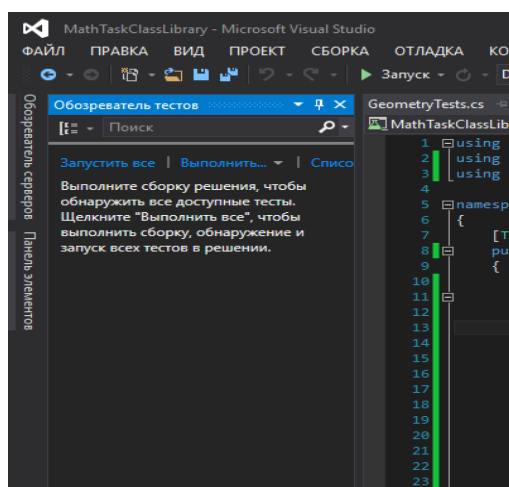


Рисунок 9

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.

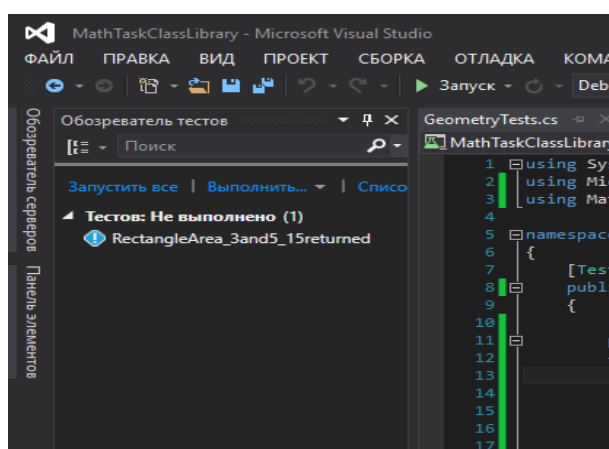


Рисунок 10

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».

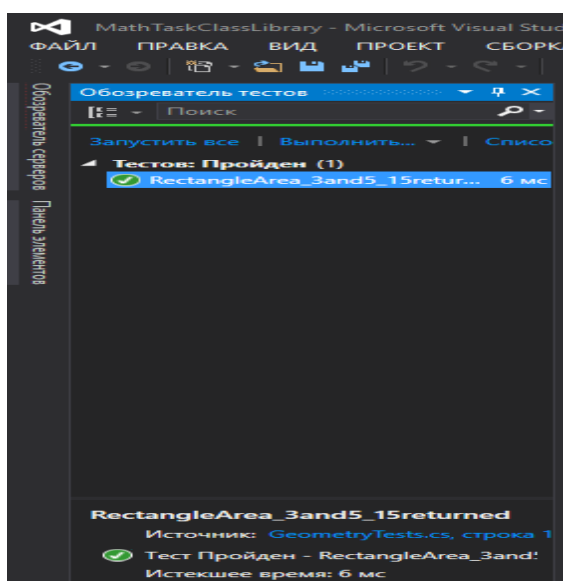


Рисунок 11

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

Изменим код метода `RectangleArea`, вычисляющего площадь прямоугольника, чтобы

симулировать провал теста и посмотреть, как поведёт себя Visual Studio. Прибавим к возвращаемому значению 10.

Запустим unit-тест.

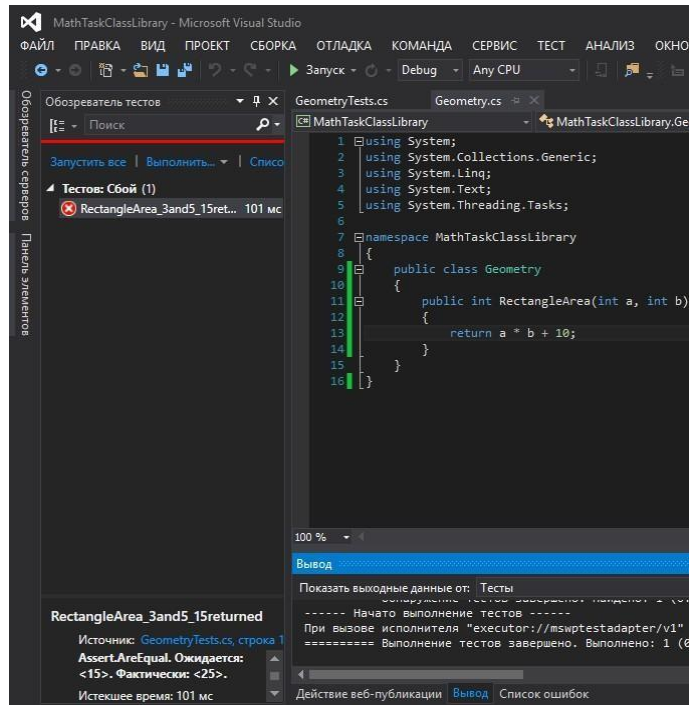


Рисунок 12

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 № 5. 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 № 6. 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 прикрепить 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 конверторы, 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 –A-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.

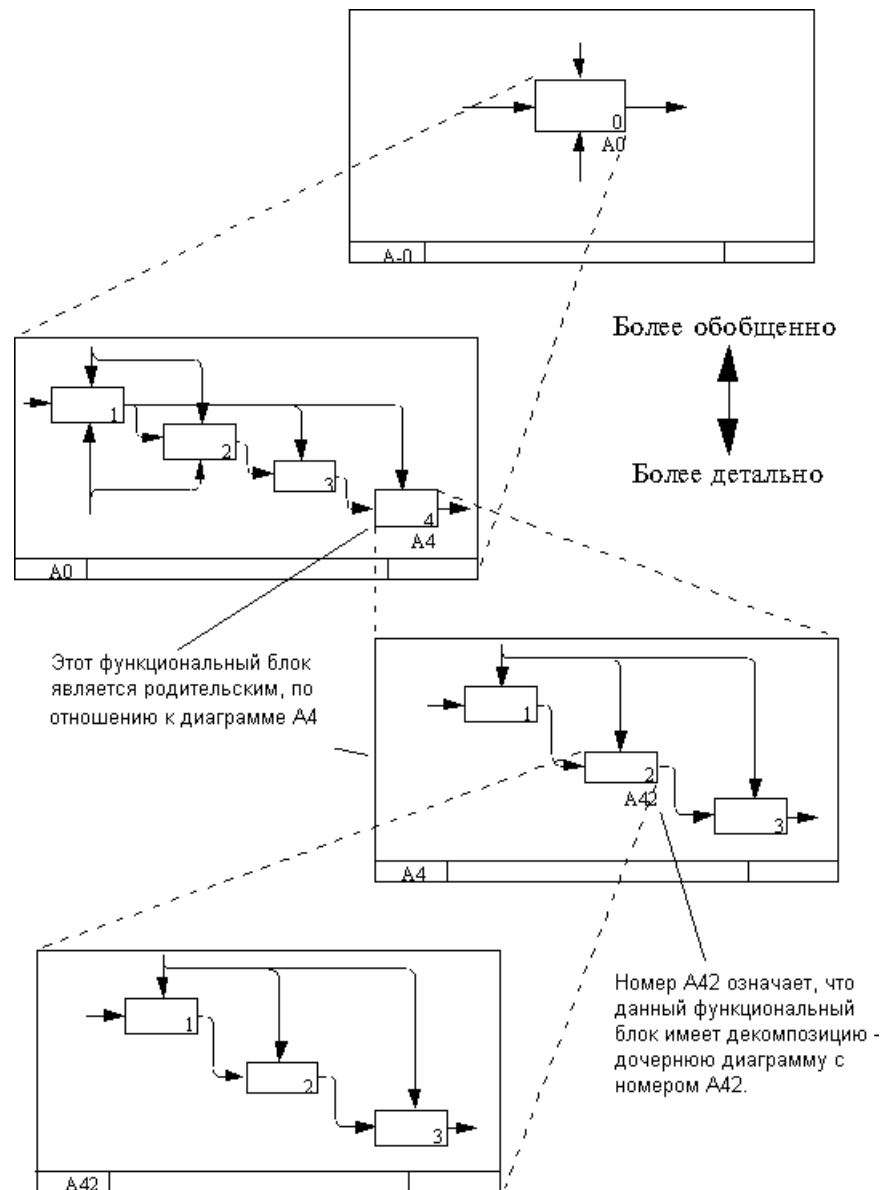


Рисунок 13

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 № 7. 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

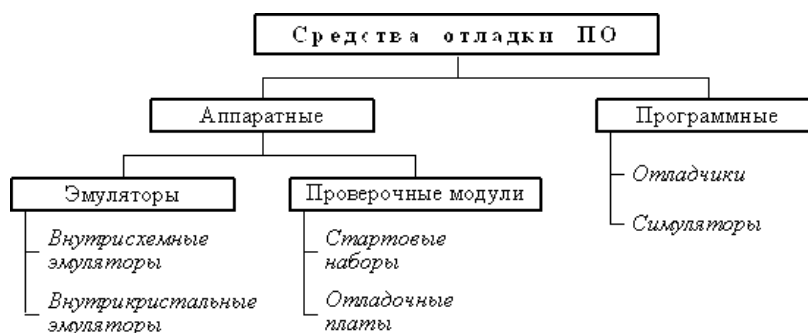


Рисунок 14

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.

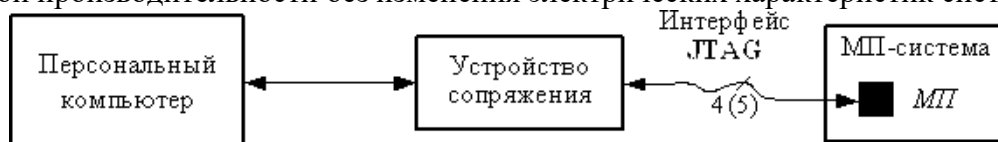


Рисунок 16

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 № 8. 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 сбоев оборудования и программы;

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

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

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

Естественно, целью всех этих проверок является поиск несоответствий техническому заданию. Считают, что только после выполнения всех видов тестирования программный продукт может быть представлен пользователю или к реализации. Однако на практике обычно выполняют не все виды оценочного тестирования, так как это очень дорого и трудоемко. Как правило, для каждого типа программного обеспечения выполняют те виды тестирования,

которые являются для него наиболее важными. Так базы данных обязательно тестируют на предельных объемах, а системы реального времени - на предельных нагрузках.

Системы для создания инсталляторов

Практика разработки коммерческого программного обеспечения показывает, что далеко не все пользователи умеют работать с архивами. Поэтому программы рекомендуется поставлять в виде исполняемых файлов, которые автоматически создают необходимые папки в файловой системе, копируют туда файлы программы, создают необходимые файлы настроек или ключи в реестре, а также 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.

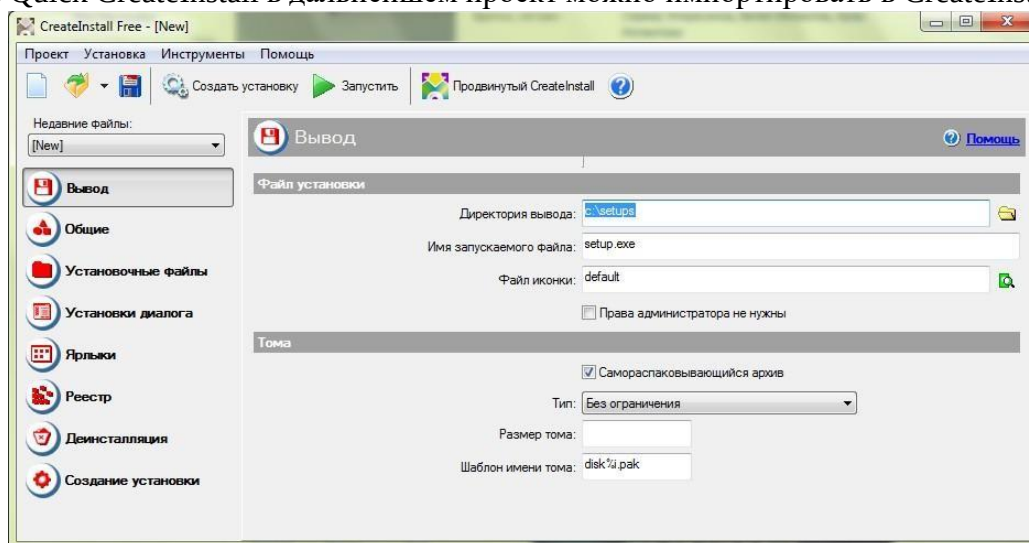


Рисунок 17

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.

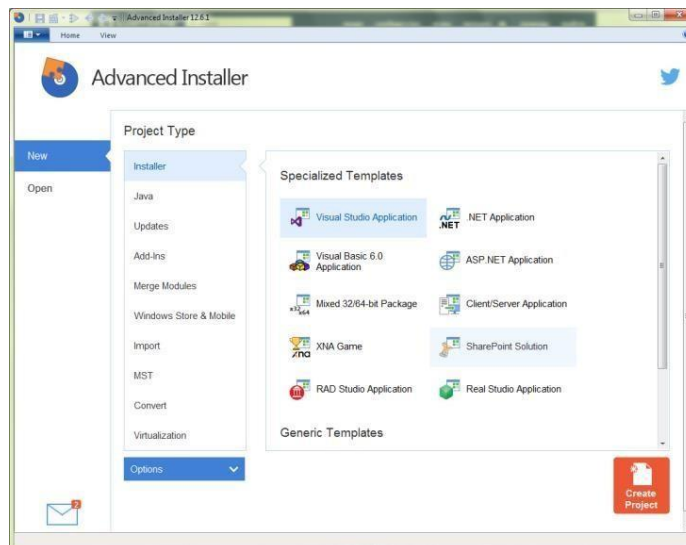


Рисунок 18

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 № 9. 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

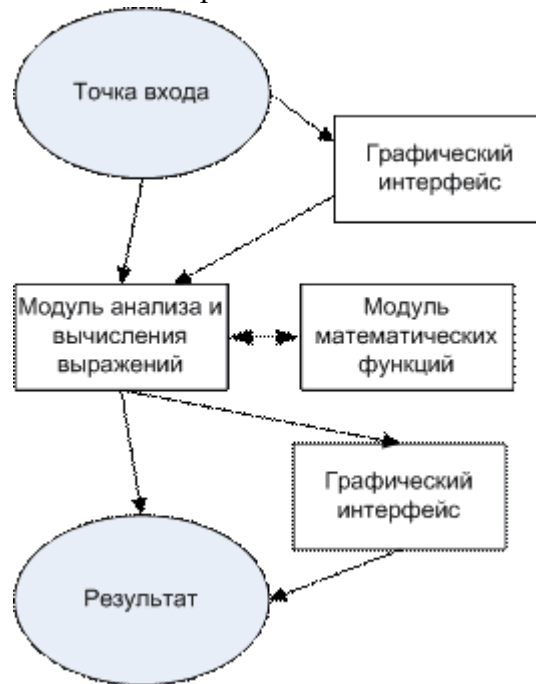


Рисунок 19

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.

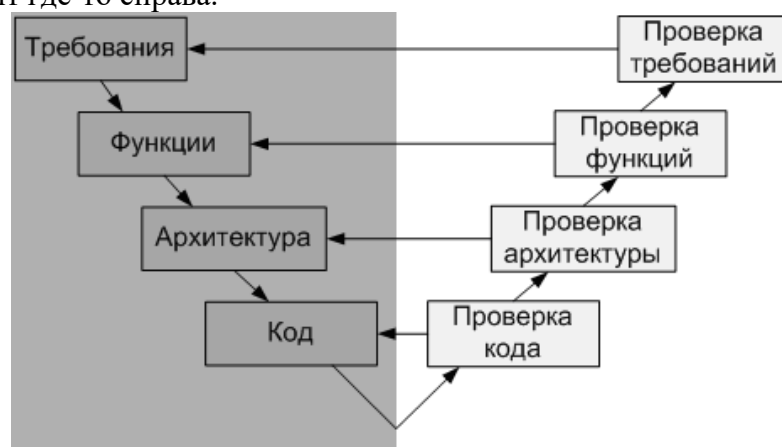


Рисунок 20

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

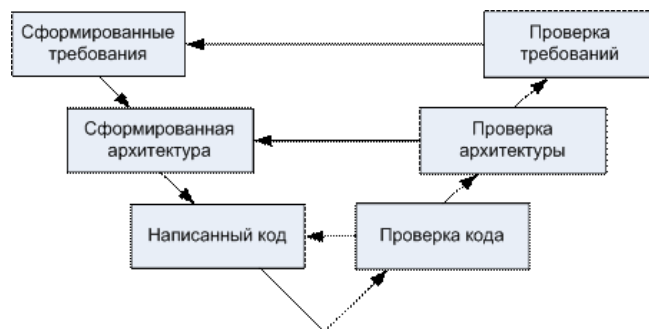


Рисунок 21

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.

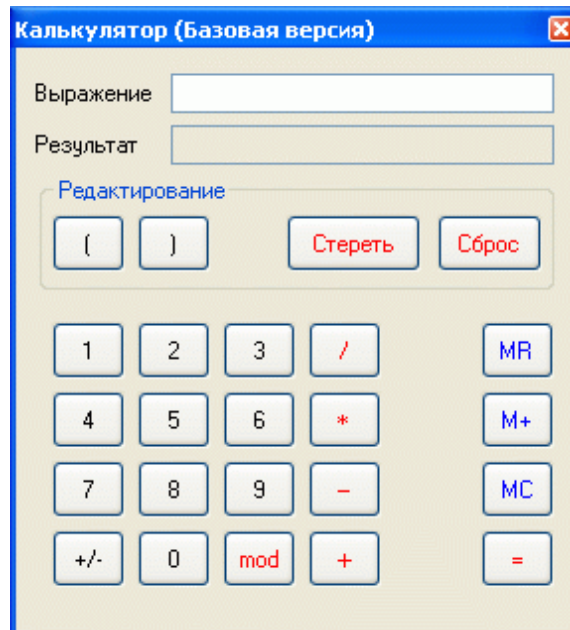


Рисунок 22

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:

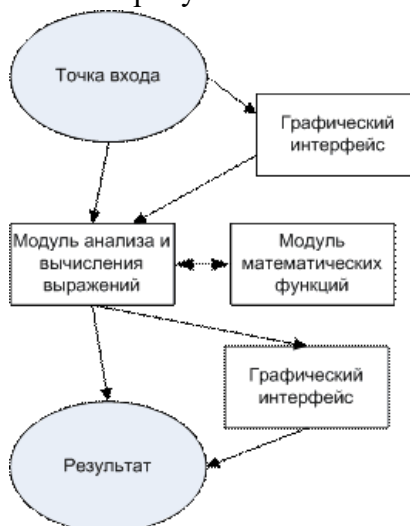


Рисунок 23

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 № 10. 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

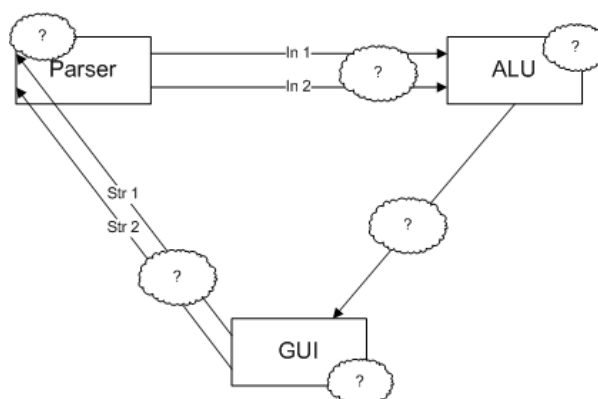


Рисунок 24

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/crcnexceptionhandlingfundamentals.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).

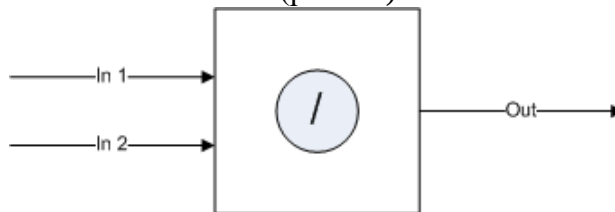


Рисунок 26

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 параметров 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 реинициализации 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 Standard)

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 OK. (рис. 26)

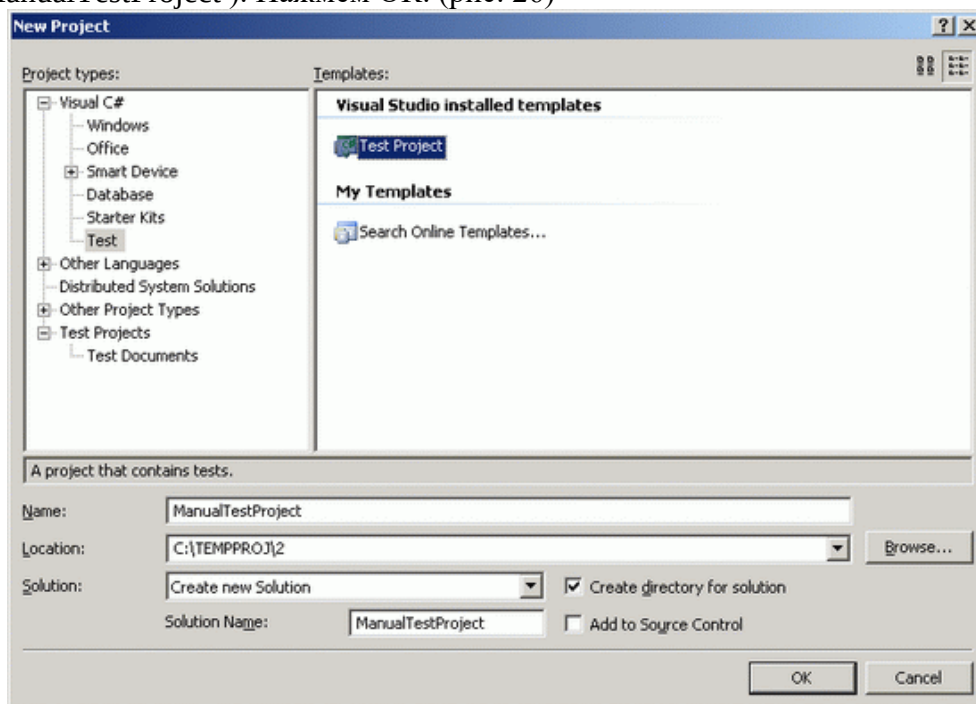


Рисунок 27

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.

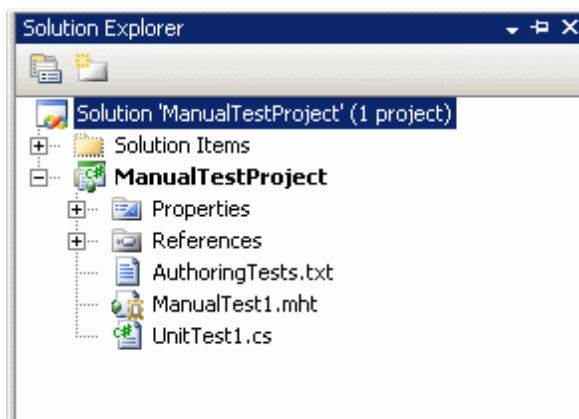


Рисунок 28

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.

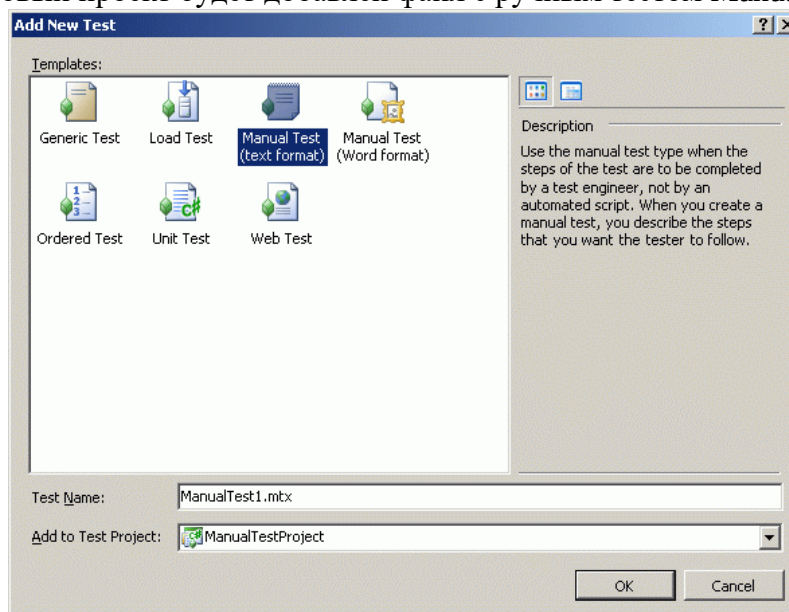


Рисунок 29

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).

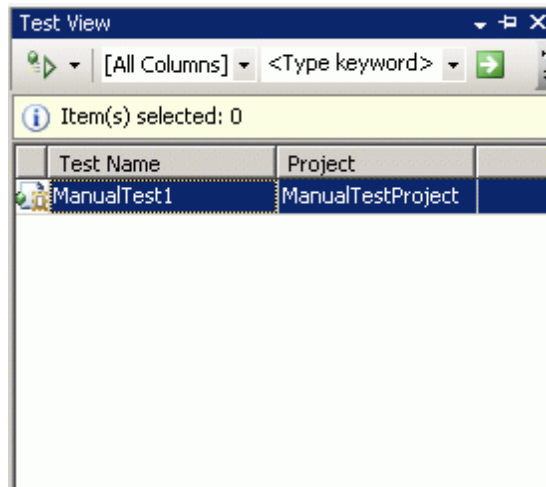


Рисунок 30

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 № 11. 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 никаким образом не искажает выполнение тестируемого модуля и адекватно моделирует поведение системы.

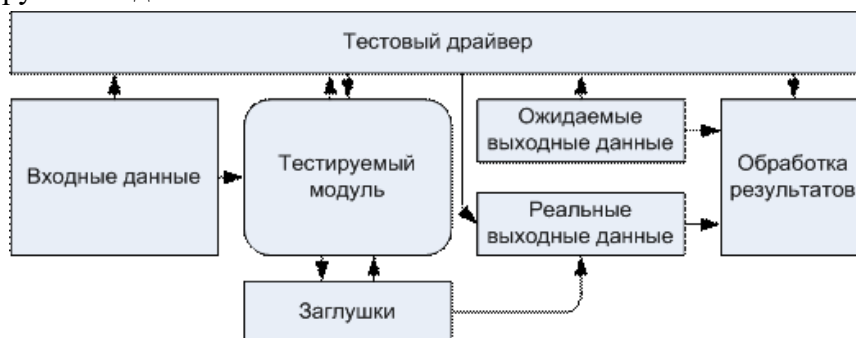


Рисунок 31

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 + выводить сообщения со значениями параметров, переданных в функцию
4. 2 + возвращать значение, заранее заданное во входных параметрах теста
5. 3 + выводить 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 непротестированных 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, проделанные нами на прошлом семинаре, как правило, проводятся не вручную. Для целей тестирования пишут специальную программу — тестовый драйвер, который и проводит 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 =
                        3000000000\n";
    richTextBox1.Text += "Ожидаемый результат: res = 0 && error =
                        \"Error 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";
richTextBox1.Text += "Входные данные: a= 3000000000, b = -
                        2850800078\n";
richTextBox1.Text += "Ожидаемый результат: res = 0 && error =
                        \"Error 06\"\n";
int res = CalcClass.Add(3000000000, -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= -28508000078, 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= 30000000000, b = -28508000078

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= 20000000000, b = 20000000000

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 № 12. 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.

Список литературы

Основные источники

1. Программная инженерия: учебное пособие / Министерство образования и науки РФ; Федеральное государственное автономное образовательное учреждение высшего образования «Северо-Кавказский федеральный университет»; сост. Т. В. Киселева, 1. - Ставрополь: СКФУ, 2017. - 137 с.: ил. - <http://biblioclub.ru/>. - Библиогр. в кн, экземпляров неограничено
2. Доррер, А. Г. Управление ИТ-проектами Электронный ресурс / Доррер А. Г., Доррер М. Г., Попов А. А. : учебное пособие для студентов бакалавриата по направлению подготовки 09.03.02 «информационные системы и технологии», профиль подготовки «информационные системы и технологии в промышленности», 09.03.04 «программная инженерия», профиль подготовки «разработка программно-информационных систем», всех форм обучения. - Красноярск: СибГУ им. академика М. Ф. Решетнёва, 2019. - 174 с.
3. Тюльпинова Н. В. Компьютерные и информационные технологии в науке и производстве: учебное пособие для магистров / Н. В. Тюльпинова. - Компьютерные и информационные технологии в науке и производстве, Весь срок охраны авторского права. - Электрон. дан. (1 файл). - Саратов: Вузовское образование, 2020. - 268 с. - электронный. - Книга находится в премиум-версии ЭБС IPR BOOKS. - ISBN 978-5-4487-0612-7, экземпляров неограничено
4. Каримов, А. М. Информатика и информационные технологии в профессиональной деятельности: практикум / А. М. Каримов, С. В. Смирнов, Г. Д. Марданов. - Информатика и информационные технологии в профессиональной деятельности, Весь срок охраны авторского права. - Электрон. дан. (1 файл). - Казань: Казанский юридический институт МВД России, 2020. - 120 с. - электронный. - Книга находится в премиум-версии ЭБС IPR BOOKS. - ISBN 2227-8397, экземпляров неограничено

Дополнительные источники

1. Мейер, Б.; Объектно-ориентированное программирование и программная инженерия / Б. Мейер. - 2-е изд., испр. - Москва: Национальный Открытый Университет «ИНТУИТ», 2016. - 286 с.: ил. - <http://biblioclub.ru/>, экземпляров неограничено
2. Барский А. Б.; Параллельные информационные технологии: учебное пособие / А. Б. Барский. - Параллельные информационные технологии, 2022-07-28. - Электрон. дан. (1 файл). - Москва: Интернет-Университет Информационных Технологий (ИНТУИТ), Ай Пи Ар Медиа, 2020. - 502 с. - электронный. - Книга находится в премиум-версии ЭБС IPR BOOKS. - ISBN 978-5-4497-0686-7, экземпляров неограничено
3. Мандра, А. Г.; Информатика и информационные технологии : лабораторный практикум / А. Г. Мандра, А. В. Попов, А. И. Дьяконов. - Информатика и информационные технологии, 2026-09-20. - Электрон. дан. (1 файл). - Самара: Самарский государственный технический университет, ЭБС АСВ, 2020. - 64 с. - электронный. - Книга находится в премиум-версии ЭБС IPR BOOKS. - ISBN 2227-8397, экземпляров неограничено

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
по организации и проведению самостоятельной работы
по дисциплине «Информационно-коммуникационные технологии»
для бакалавров направления подготовки
09.03.01 «Информатика и вычислительная техника»
направленность (профиль) "Программное обеспечение средств
вычислительной техники и автоматизированных систем"

Ставрополь
2026

ВВЕДЕНИЕ