Министерство образования и науки Российской Федерации
Волгоградский государственный архитектурно-строительный университет

Качество и надежность информационных систем

Методические указания к лабораторным занятиям

Составители Б. Х. Санжапов, Н. М. Рашевский

Волгоград
ВолгГАСУ
2016

УДК 004.05(076.5)
ББК 32.973.1я73
К309

К309 **Качество** и надежность информационных систем [Электронный ресурс]: методические указания к лабораторным занятиям / М-во образования и науки Рос. Федерации, Волгогр. гос. архит.-строит. ун-т; сост. Б. Х. Санжапов, Н. М. Рашевский. — Электронные текстовые и графические данные (0,6 Мбайт). — Волгоград : ВолгГАСУ, 2016. — Электронное издание сетевого распространения. — Систем. требования: PC 486 DX-33; Microsoft Windows XP; Internet Explorer 6.0; Adobe Reader 6.0. — Официальный сайт Волгоградского государственного архитектурно-строительного университета. Режим доступа: http://www.vgasu.ru/publishing/on-line/ — Загл. с титул. экрана.

Рассматриваются основные методы обеспечения качества и надежности информационных систем: разработка документации, использование систем контроля версий исходного кода, юнит-тестирование и использование метрик программного кода.

Для студентов, обучающихся по профилю «Информационные системы и технологии».

УДК 004.05(076.5)
ББК 32.973.1.я73

Лабораторная работа №1. Разработка технической документации

1 Цель работы

Освоение навыков выделения требований к программным продуктам и формирования технического задания по государственным стандартам.

2 Задание

1. Получите задание у преподавателя.
2. Для заданного варианта необходимо разработать техническое задание по одному из гостов.

При выполнении задания используйте теоретический материал приведенный ниже.

3 Теоретический материал

3.1 Общие сведения о стандартах

Техническое задание является исходным материалом для создания информационной системы или другого продукта. Поэтому техническое задание (сокращенно ТЗ) в первую очередь должно содержать основные технические требования к продукту и отвечать на вопрос, что данная система должна делать, как работать и при каких условиях.

Как правило, этапу составления технического задания предшествует проведение обследования предметной области, которое завершается созданием аналитического отчета. Именно аналитический отчет (или аналитическая записка) ложится в основу документа Техническое задание.
Если в отчете требования заказчика могут быть изложены в общем виде и проиллюстрированы UML-диаграммами, в техническом задании следует подробно описать все функциональные и пользовательские требования к

системе. Чем подробнее будет составлено техническое задание, тем меньше спорных ситуаций возникнет между заказчиком и разработчиком во время приемочных испытаний.

Таким образом, техническое задание является документом, который позволяет как разработчику, так и заказчику представить конечный продукт и впоследствии выполнить проверку на соответствие предъявленным требованиям.

Руководствующими стандартами при написании технического задания являются ГОСТ 34.602.89 «Техническое задание на создание автоматизированной системы» [1] и ГОСТ 19.201-78 «Техническое задание. Требования к содержанию и оформлению» [2]. Первый стандарт предназначен для разработчиков автоматизированных систем, второй для программных средств (разница между данными стандартами описана в статье «Что такое ГОСТ» [3]).

Список и описание разделов, которые должны содержаться в техническом задании согласно ГОСТу 19.201-78 Техническое задание. Требования к содержанию и оформлению:

1. Введение
2. Основания для разработки
3. Назначение разработки
4. Требования к программе или программному изделию
4.1. Требования к функциональным характеристикам
4.2. Требования к надежности
4.3. Условия эксплуатации
4.4. Требования к составу и параметрам технических средств
4.5. Требования к информационной и программной совместимости
4.6. Требования к маркировке и упаковке
4.7. Требования к транспортированию и хранению
4.8. Специальные требования
5. Требования к программной документации

6. Технико-экономические показатели

7. Стадии и этапы разработки

8. Порядок контроля и приемки

Список и описание разделов, которые должны содержаться в техническом задании согласно ГОСТу 34.602.89 Техническое задание на создание автоматизированной системы:

1. Общие сведения

2. Назначение и цели создания системы

3. Характеристика объекта автоматизации

4. Требования к системе

4.1. Требования к системе в целом

4.1.1. Требования к структуре и функционированию системы

4.1.2. Требования к функциям (задачам), выполняемым системой

4.1.3. Показатели назначения

4.1.4. Требования к надежности

4. 1.5. Требования к безопасности

4. 1.6. Требования к эргономике и технической эстетике

4. 1.7. Требования к транспортабельности для подвижных систем

4. 1.8. Требования к эксплуатации, техническому обслуживанию, ремонту и хранению компонентов системы

4. 1.9. Требования к защите информации от несанкционированного доступа

4. 1.10. Требования по сохранности информации при авариях

4. 1.11. Требования к защите от влияния внешних воздействий

4. 1.12. Требования к патентной чистоте

4. 1.13. Требования по стандартизации и унификации

4. 1.14. Дополнительные требования

4.2. Требования к численности и квалификации персонала системы и режиму его работы

4.3. Требования к видам обеспечения

5. Состав и содержание работ по созданию системы

6. Порядок контроля и приемки системы

7. Требования к составу и содержанию работ по подготовке объекта автоматизации к вводу системы в действие

8. Требования к документированию

9.Источники разработки

Таким образом, документ Техническое задание должен, отражать все требования к проектируемому продукту, выделенные на этапе аналитического исследования объекта автоматизации.

Основными разделами технического задания для обоих гостов являются:

- Общие сведения о системе (программе);
- Назначение, цели и задачи системы (программы);
- Требования к системе (функциональные требования, пользовательские требования, требования к системе в целом и тд);
- Требования к видам обеспечения;
- Требования к документированию;
- Стадии и этапы разработки;
- Порядок контроля и приемки системы (программы).

3.2 Общие сведения

Данный раздел должен содержать полное наименование системы и все варианты сокращений, которые будут использованы при разработке документации.

Пример:

«В данном документе создаваемая информационная система называется «Единое окно доступа к образовательным ресурсам», сокращенно ЕО. Систему Единое окно доступа к образовательным ресурсам далее в настоящем документе допускается именовать Единое окно или Система.»

Также сюда следует включить подразделы, сообщающие реквизиты организаций, участвующих в разработке (Заказчика и Исполнителя).

В подразделе «Основания для разработки» документа Техническое задание перечисляются основные документы, на основании которых выполняются данные работы. Например, для системы, выполняемой по заказу Правительства страны или другого Государственного органа, должны быть указаны законы, указы и постановления Правительства.

Далее следует указать сроки начала и окончания работ и сведения об источнике финансирования. Данная информация может быть указана и в конце технического задания в разделе с указанием стадий и этапов работ.

Неотъемлемой частью документа Техническое задание также должен быть список терминов и сокращений. Термины и сокращения лучше представить в виде таблицы с двумя столбцами «Термин» и «Полная форма».

Термины и сокращения располагаются в алфавитном порядке. В первую очередь принято давать расшифровку русскоязычным терминам и сокращениям, потом англоязычным.

3.3 Назначение и цели создания системы

Данный раздел должен содержать назначение и цели создания системы.

Пример:

«Информационная система «Единое окно доступа к образовательным ресурсам» предназначена для обеспечения пользователей полной, оперативной и удобной информацией, касающейся системы образования Российской федерации, организаций выполняющих функцию образовательных учреждений.

Основной целью Системы является формирование единой информационной среды и автоматизации бизнес-процессов Образовательных учреждений Российской Федерации.

Создание информационной системы «Единое окно» должно обеспечить:

- предоставление пользователям широкого спектра информационных ресурсов;

- повышение уровня информационной безопасности;

- повышение эффективности работы образовательных учреждений и ведомств за счет оптимизации ряда бизнес-процессов;

- повышение эффективности процесса взаимодействия информационных систем и сервисов внутри ведомства.

- Создание Системы позволит сократить эксплуатационные затраты в результате повышения эффективности работы ведомства.»

3.4 Требования к системе

Данный раздел предназначен для описания основных функциональных требований системы. Это самая важная часть технического задания, так как именно она станет основным аргументом исполнителя при спорах с Заказчиком в процессе сдачи системы в эксплуатацию. Поэтому к его написанию необходимо подойти наиболее тщательно.

В документе Техническое задание должны быть представлены все требования, выявленные на этапе проведения анализа объекта автоматизации. Лучше всего выделить основные бизнес-процессы, которые и должны быть раскрыты посредством описания функциональных требований.

Пример:

«4.1 Бизнес-процесс «Предоставление информации об образовательных учреждениях Российской Федерации. В данном бизнес-процессе выделяются следующие участники:

Модератор – работник ведомства, входящий в состав обслуживающего персонала Системы, ответственный за корректность предоставляемых данных

Автор – сотрудник образовательного учреждения, ответственный за размещение информации о своей организации.

Пользователь – гражданин, нуждающийся в получении информации о работе образовательных учреждений Российской Федерации.

4.1.1 Регистрация образовательного учреждения в Системе

Регистрация образовательного учреждения Российской Федерации осуществляется ответственным сотрудником учреждения («Постановление Правительства …»).

Процесс регистрации образовательного учреждения включает следующие шаги:

- Автор создает запись об организации;

- Автор заносит данные организации;

- Система проверяет наличие лицензии для данной организации

- Если лицензия существует в базе данных, Система отправляет Автору сообщение об успешной регистрации;

- Если лицензия не найдена в базе данных, Система отправляет сообщение Автору об отсутствии лицензии для данной организации.»

Информацию, приведенную в данном разделе, следует, более полно раскрыть в приложении к документу Техническое задание. В приложении к техническому заданию можно привести экранную форму и ниже описать все события, которые на ней присутствуют (создание, просмотр, редактирование, удаление и т.п.).

Требования к системе в целом включают раскрытие ее архитектуры с описанием всех подсистем. В данной части Технического задания следует описать требования к интеграции системы с другими продуктами (если таковые имеются). Далее в техническое задание должны быть включены:

- требования к режимам функционирования системы
- показатели назначения
- требования к надежности
- требования к безопасности
- требования к численности и квалификации персонала и режиму его работы

- требования к защите информации
- требования по сохранности информации при авариях
- требования к патентной чистоте
- требования по стандартизации и унификации
- и т.д.

3.5 Требованиям к видам обеспечения

В данном разделе должны быть представлены требования к математическому, информационному, лингвистическому, программному, техническому и др. видам обеспечения (если таковые имеются).

3.6 Требования к документированию

Раздел «Требования к документированию» включает перечень проектных и эксплуатационных документов, которые должны быть предоставлены заказчику.

Данный раздел технического задания также важен, как и описание функциональных требований, поэтому не следует ограничиваться фразой «Заказчику должна быть предоставлена вся документация согласно ГОСТ 34». Большинство документов из списка, указанного в ГОСТ 34.201-89 скорее всего не понадобятся ни исполнителю, ни заказчику, поэтому лучше сразу согласовать список на этапе разработки документа Техническое задание.

Минимальный пакет документов обычно включает:

- Техническое задание;
- Ведомость эскизного (технического) проекта;
- Пояснительная записка к Техническому проекту;
- Описание организации информационной базы;
- Руководство пользователя;

- Руководство администратора;
- Программа и методика испытаний;
- Протокол приемочных испытаний;
- Акт выполненных работ

Перечень документов в техническом задании лучше представить в виде таблицы, где указывается наименование документа и стандарт на основании, которого он должен быть разработан.

3.7 Стадии и этапы разработки

В данном разделе документа Техническое задание следует представить информацию обо всех этапах работ, которые должны быть проведены.
Описание этапа должно включать наименование, сроки, описание работ и конечный результат.

3.8 Порядок контроля и приемки системы

В данном разделе документа Техническое задание необходимо указать документ, на основании которого должны быть проведены приемо-сдаточные испытания.

При необходимости техническое задание может быть дополнено другими разделами, или сокращено путем удаления нецелесообразных пунктов.
При изменении структуры технического задания, во избежание конфликтных ситуаций, ее необходимо согласовать с заказчиком до разработки документа.

Лабораторная работа №2. Системы контроля версий

1 Цель работы

Получить навыки использования систем контроля версий при разработке программного обеспечения.

2 Задание

1 Создайте репозиторий на локальном компьютере для системы контроля версий Mercurial с использованием графической оболочки TortoiseHg.

2 Создайте тестовый проект на любом знакомом языке программирования.

3 Для тестового проекта выполните команды системы контроля версий Mercurial для:

- добавления файла в репозиторий;
- удаления файла из репозитория;
- создания коммитов;
- создания отдельных веток в репозитории.

3 Теоретический материал

3.1 Установка системы контроля версий

Последнюю стабильную версию графической оболочки TortoiseHg, можно скачать с репозитория проектов Source Forge по ссылке [4]. Процесс установки стандартный для MS Windows. Отдельно ядро системы контроля версий Mercurial устанавливать не нужно, посколько оно интегрировано в TortoiseHg.

3.2 Создание репозитория

Для того чтобы начать управлять историей изменений какого-либо файла, необходимо создать репозиторий, где эти изменения будут хранится. Для этого следует нажать правой кнопкой мыши на папке, вызвав контекстное меню, и выполнить команду TortoiseHg — «Create a repository here».

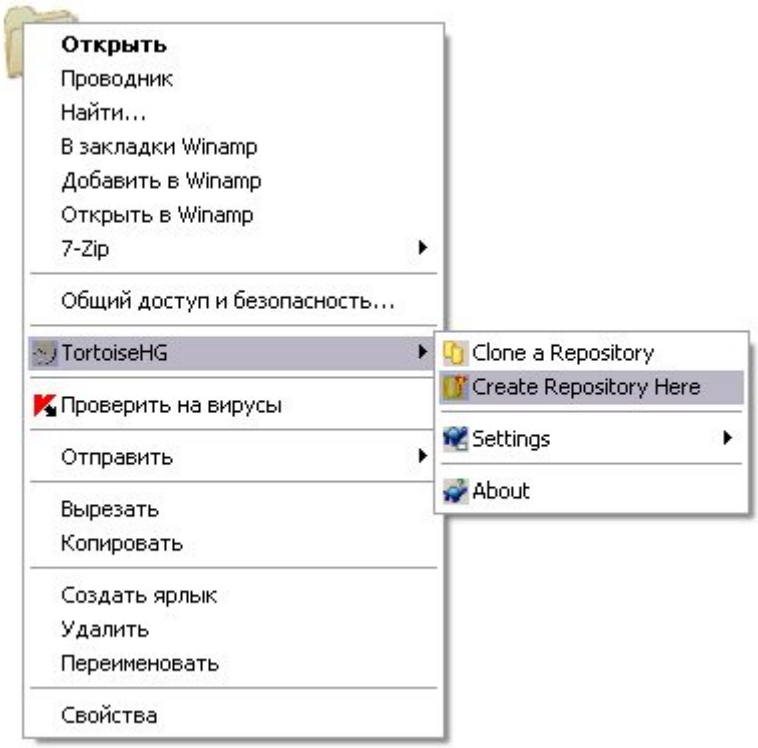

Рисунок 1 – контекстное меню TortoiseHg при создании репозитория

После выполнения этой команды, внутри выбранной папки, появится папка с именем .hg, в которой будут храниться все отмеченные версии файлов. При этом следует заметить, что репозиторий всегда будет создаваться внутри выбранной папки.

Далее необходимо указать, для каких именно файлов будет храниться история изменений, т.е. внести их в репозиторий.

Для того чтобы создать первую, начальную версию проекта, необходимо вызвать контекстное меню на папке с репозиторием и выполнить команду TortoiseHg — «Commit».

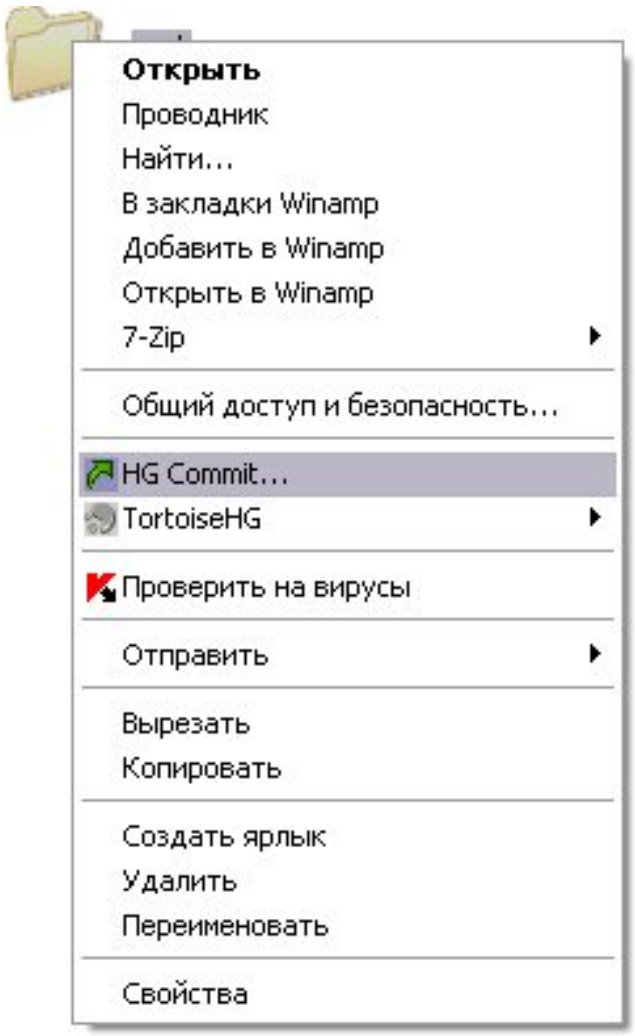

Рисунок 2 – Контекстное меню TortoiseHg для папки с репозиторием

Следует заметить, что команда Commit, становится доступной из контекстного меню, только после создания репозитория. Далее на экране должно появится окно, показанное на рисунке 3. Здесь необходимо отметить все или конкретные файлы, по которым будем отслеживать изменения. Также, нужно ввести имя для сохраняемой версии. Затем необходимо нажать кнопку Commit, и первая версия проекта будет сохранена.

Комманда Commit (одно из значений в английском — фиксировать), является основной при работе с Mercurial. То есть после того как кто-либо внес какие-либо серьезные изменения, или просто завершил очередной этап работы, необходимо зафиксировать изменения, в виде следующей ветки проекта.

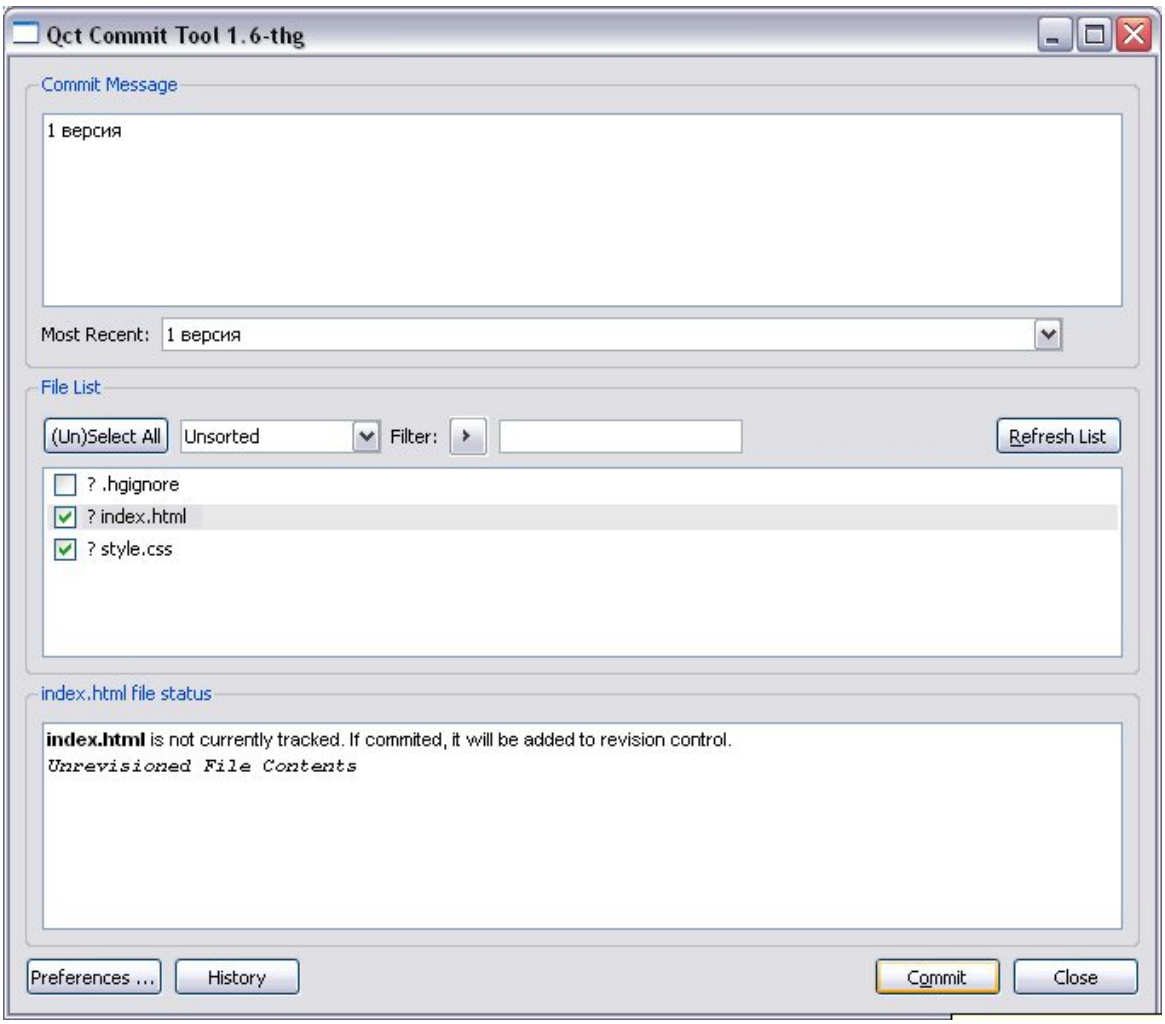

Рисунок 3 Окно фиксации изменений в TortoiseHg

3.3 Внесение изменений

Далее, внесем несколько незначительных изменений в один из файлов файлов. Прежде чем зафиксировать следующий этап работы над проектом, можно проследить, какие именно изменения были внесены после сохранения последней версии. Для этого, из контекстного меню, нужно выполнить команду: TortoiseHg — «VisualDiff».

Окно разделено на две области, левая область отображает прежний вариант файла, правая - измененный. Здесь можно внимательно проанализировать внесенные изменения, и если нужно отменить их, с помощью команды «Undo Changes», доступной все по тому же контекстному меню.

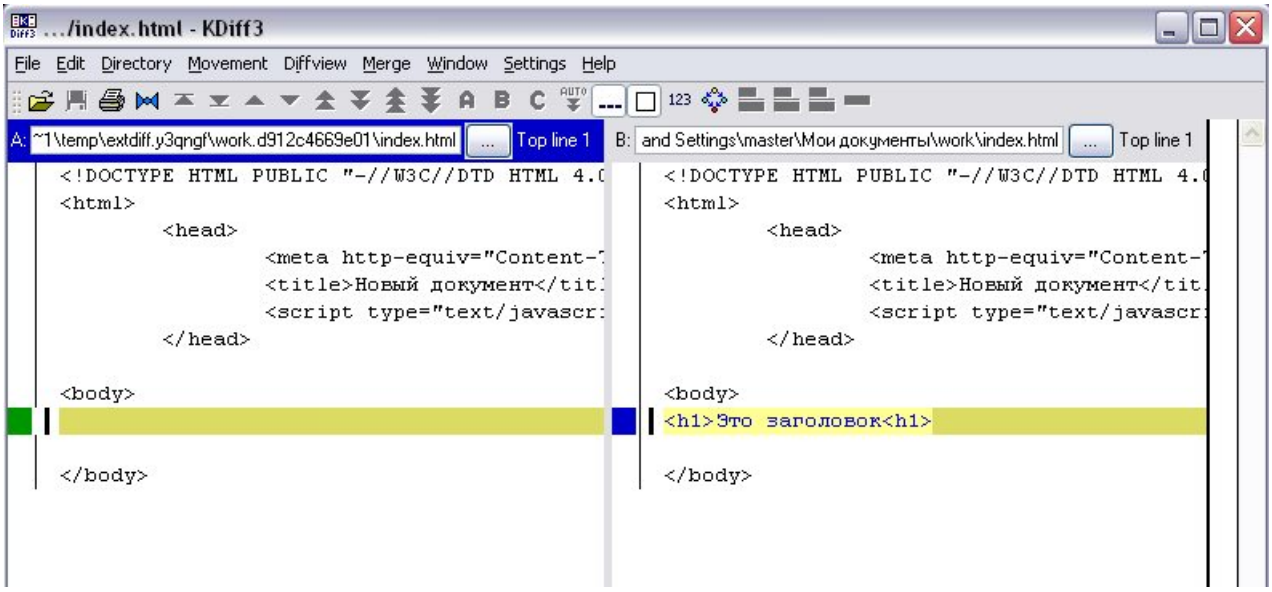

Рисунок 4 – Окно отображения разницы между файлами TortoiseHg

Если изменения внесены правильно, то можно выполнить команду «Commit», и сохранить вторую версию проекта. После этого можно посмотреть историю версий проекта, по команде TortoiseHg — «View Changelog».

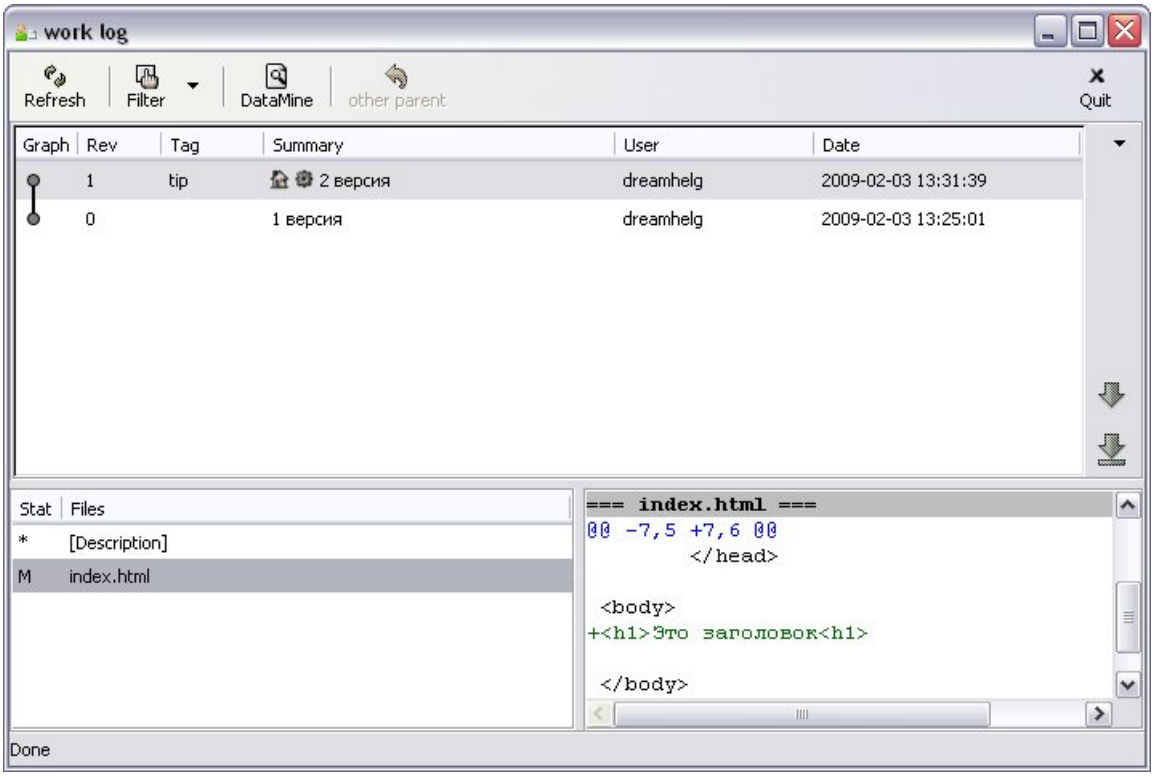

Рисунок 5 – Окно отображения истории версий TortoiseHG

Здесь можно увидеть все отмеченные этапы разработки, список измененных файлов и внесенных в них изменений на каждом этапе. Этапы нумеруются с нуля.

У tortoiseHg, есть множество других возможностей, таких как объединение нескольких веток разработки в одну, отслеживание состояния файлов, возможность создавать клон репозитория на внешнем носителе для работы на разных компьютерах, синхронизация репозиториев и др.

Лабораторная работа №3. Юнит тестирование.

1 Цель работы

Получение навыков разработки юнит-тестов и проведения тестирования программного обеспечения.

2 Задание

Создать набор юнит тестов для двух функций. Функции, для которых создаются юнит тесты, необходимо согласовать с преподавателем.

3 Теоретический материал

3.1 Теория юнит тестирования

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. Этот тип тестирования обычно выполняется программистами.

Модульное тестирование позже позволяет программистам проводить рефакторинг, будучи уверенными, что модуль по-прежнему работает корректно (регрессионное тестирование). Это поощряет программистов к изменениям кода, поскольку достаточно легко проверить, что код работает и после изменений.

Модульное тестирование помогает устранить сомнения по поводу отдельных модулей и может быть использовано для подхода к тестированию «снизу вверх»: сначала тестируя отдельные части программы, а затем программу в целом.

Модульные тесты можно рассматривать как «живой документ» для тестируемого класса. Клиенты, которые не знают, как использовать данный класс, могут использовать юнит-тест в качестве примера.

Поскольку некоторые классы могут использовать другие классы, тестирование отдельного класса часто распространяется на связанные с ним. Например, класс пользуется базой данных; в ходе написания теста программист обнаруживает, что тесту приходится взаимодействовать с базой. Это ошибка, поскольку тест не должен выходить за границу класса. В результате разработчик абстрагируется от соединения с базой данных и реализует этот интерфейс, используя свой собственный mock-объект. Это приводит к менее связанному коду, минимизируя зависимости в системе.

Тестирование программного обеспечения — комбинаторная задача. Например, каждое возможное значение булевской переменной потребует двух тестов: один на вариант TRUE, другой — на вариант FALSE. В результате на каждую строку исходного кода потребуется 3–5 строк тестового кода.

Как и любая технология тестирования, модульное тестирование не позволяет отловить все ошибки программы. В самом деле, это следует из практической невозможности трассировки всех возможных путей выполнения программы, за исключением простейших случаев.

При выполнении юнит-тестов происходит тестирование каждого из модулей по отдельности. Это означает, что ошибки интеграции, системного уровня, функций, исполняемых в нескольких модулях, не будут определены. Кроме того, данная технология бесполезна для проведения тестов на производительность. Таким образом, модульное тестирование более эффективно при использовании в сочетании с другими методиками тестирования.

Для получения выгоды от модульного тестирования требуется строго следовать технологии тестирования на всём протяжении процесса разработки программного обеспечения. Нужно хранить не только записи обо всех проведённых тестах, но и обо всех изменениях исходного кода во всех модулях. С этой целью следует использовать систему контроля версий ПО. Таким образом, если более поздняя версия ПО не проходит тест, который был успешно пройден ранее, будет несложным сверить варианты исходного кода и устранить ошибку. Также необходимо убедиться в неизменном отслеживании и анализе неудачных тестов. Игнорирование этого требования приведёт к лавинообразному увеличению неудачных тестовых результатов.

3.2 Среда Nunit.

NUnit – открытая среда юнит-тестирования приложений для .NET, которая позволяет создавать автоматические тесты. Nunit легко интегрируется с другими средствами разработки ПО, тесты NUnit можно запустить из консоли, графической оболочки, скрипта сборки, или любым другим программным способом.

3.2.1 Интеграция с MS Visual Studio

Сначала необходимо скачать NUnit с сайта [5]. Затем необходимо разархивировать и запустить установщик программы.

Существует по меньшей мере два способа использования NUnit. В данном пособии описано добавление NUnit как внешнего инструмента и запуск тестов NUnit при отладке роекта в Visual Studio.

Для того, чтобы интегрировать NUnit в Visual Studio, необходимо добавить его как внешний нструмент. Для этого в меню «Tools» необходимо выбрать пункт «External Tools». В появившемся окне необходимо указать:

- Title: название пункта меню, например, NUnit.

- Command: Путь до графического интерфейса NUnit, например, C:\Program Files (x86)\NUnit 2.5.3\bin\net-2.0\nunit.exe.

- Arguments: Аргументы передаваемые приложению, т.е. путь до сборки, например, $(ProjectDir)bin/Debug/$(TargetName)$(TargetExt).

- Initial directory - Рабочая папка - $(TargetDir).

После этого, если выбрать проект с тестами и выбрать в меню Tools пункт NUnit, стартует графическая оболочка программы.

Для того, чтобы тесты запускались при запуске отладки проекта, необходимо настроить запуск NUnit в свойствах проекта (вкладка Debug). Необходимо найти вкладку «Start external program»(запускать внешнюю программу) и задать путь до NUnit, например, C:\Program Files (x86)\NUnit 2.5.3\bin\net-2.0\nunit.exe.

Далее, во вкладке Command line arguments необходимо задать путь до файла настроек NUnit и категорию для автозапуска, например, файл настроек ..\..\NunitTests.nunit и категория /include:Debug /runselected (разделяются пробелом).

В папке проекта необходимо создать файл с названием NunitTests.nunit со следующим содержанием(название сборки необходимо заменить на свое):

<NUnitProject>

```
<Settings activeconfig="Debug" />
<Config name="Debug">
<assembly path="bin\Debug\[название сборки].dll" />
</Config>
<Config name="Release"></Config>
</NUnitProject>
```

В дальнейшем, тесты, которые необходимо запускать при отладке, отмечаются аттрибутом [Test, Category(«Debug»)].

3.2.2 Разработка тестов

Сначала необходимо создать проект библиотечного класса. Для этого, неободимо выбрать пункт меню «File -> New -> Project», в открывшемся окне выбрать Visual C # в установленных шаблонах и выбрать тип приложения «Class library». После этого необходимо задать название проекта и нажать «OK». В результате созданный проект появится в Solution Explorer (в правом верхнем углу Visual Studio).

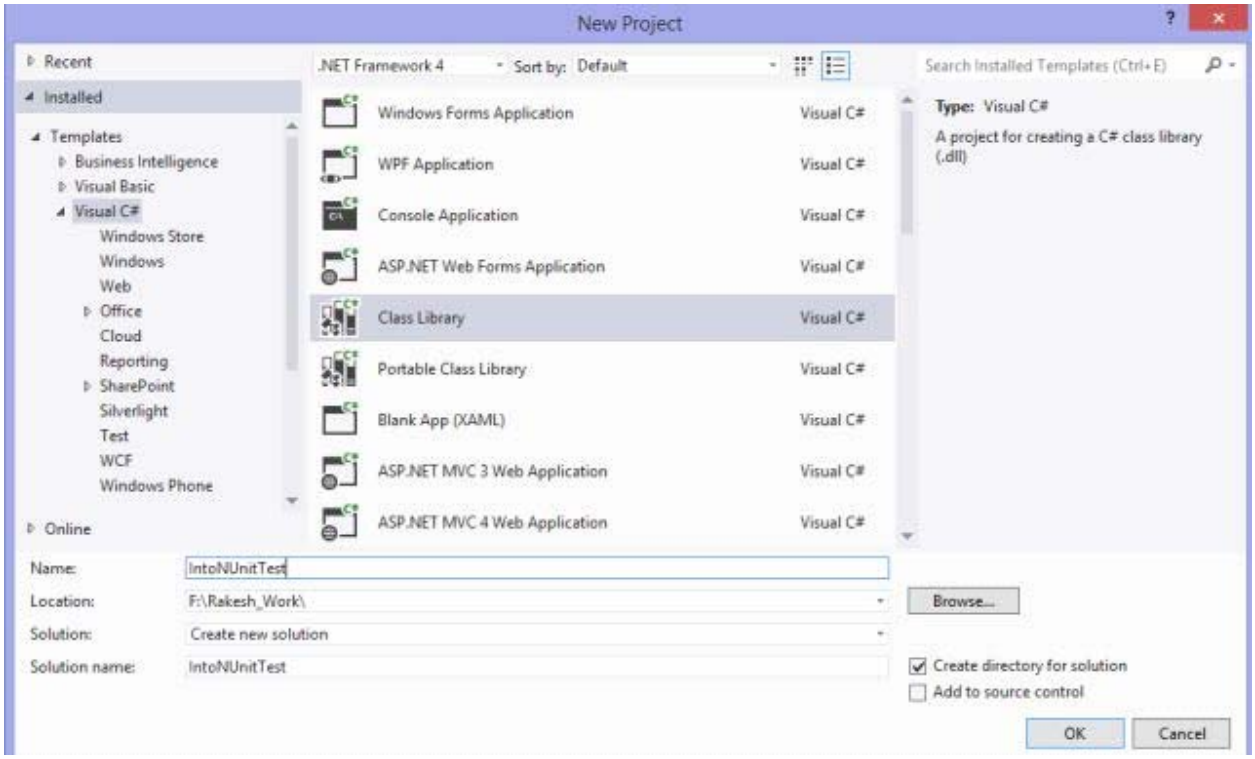

Рисунок 6 – Окно создания проекта

Далее необходимо добавить ссылку на NUnit в проект. Для этого необходимо кликнуть правой кнопкой мыши на ссылку, выбрать «Add reference -> Browse », затем nunit.framework.dll и «OK».

Далее необходимо создать тестовый класс. Для этого необходимо кликнуть правой кнопкой мыши на проекте, выбрать Add -> Class, ввести имя и нажать кнопку "Add".

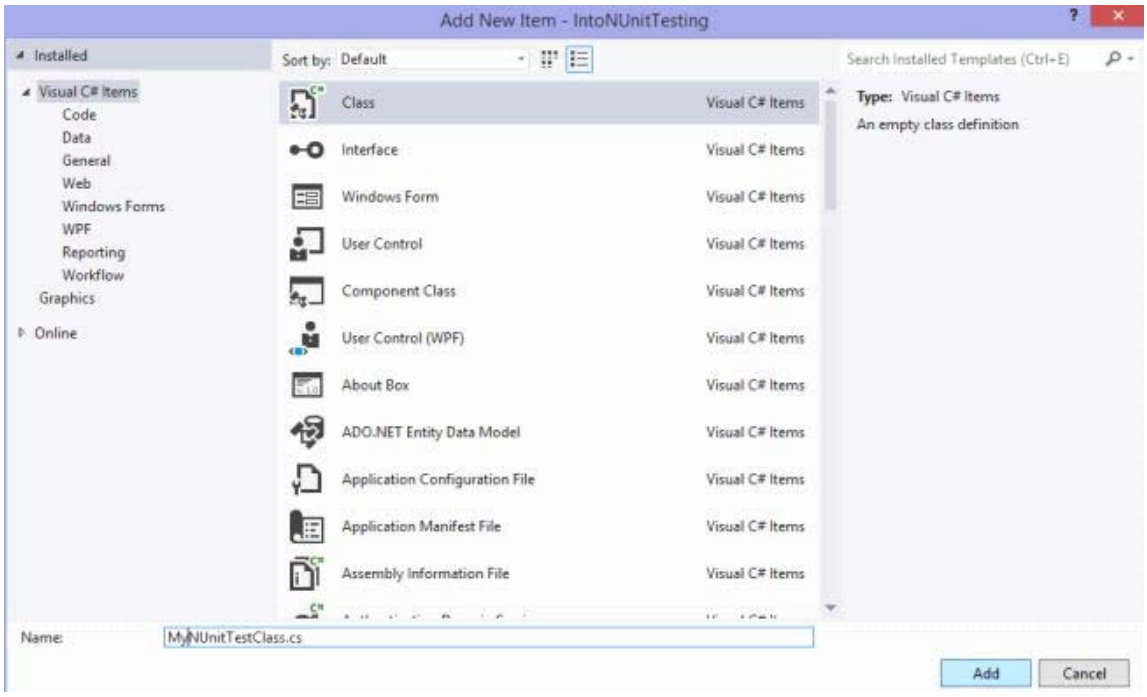

Рисунок 7 – Окно добавления класса

После этого отобразится созданный класс.

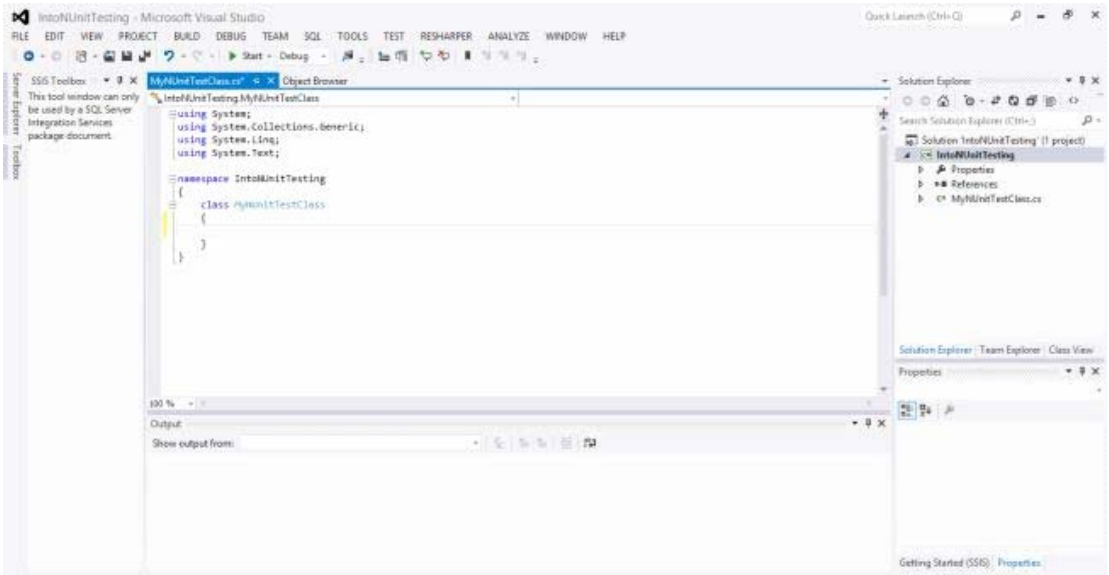

Рисунок 8 – Код только что созданного тестового класса

В код необходимо добавить сторку using NUnit.Framework; При написании кода тестового класса неободимо соблюдать некоторые условия:

- Каждый класс должен содержать атрибут [TestFixure] и должен быть общедоступен.

- В каждом методе должен присутствовать атрибут [Test].

- Оператор подтверждения об отсутствии ошибок: Булевские значения, описывающие, что должно быть ключевым словом, когда выполняется действие.

- Ожидаемое исключение: один из типов исключения, который мы ожидаем во время выполнения тест-метода.

- Установка: программа, которая запускается перед выполнением каждого тест-метода (например, регистрация в системе конкретного пользователя или инициализация одноэлементных классов).

- Демонтаж: программа, запускается после окончания каждого тест-метода (например, удаление строк из таблицы, которые были вставлены во время теста).

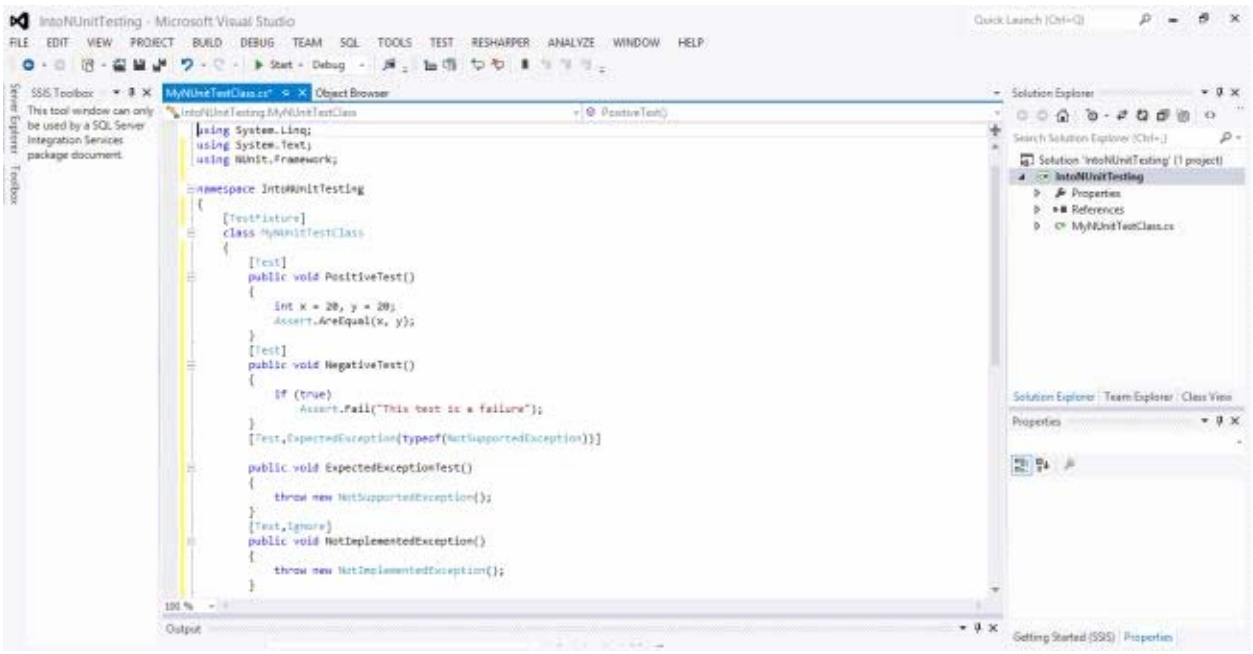

Рисунок 9 – Пример кода тестового класса

3.2.3 Запуск тестов

После написания всех тестов в тестовом классе, необходимо запустить тесты, чтобы проверить, проходят ли они успешно. Для запуска тестов нужно перейти в папку NUnit, выбрать NUnit Application (.exe) и дважды кликнуть по ней, затем выбрать команду File-> Open Project, выбрать проект, затем кликнуть Run.

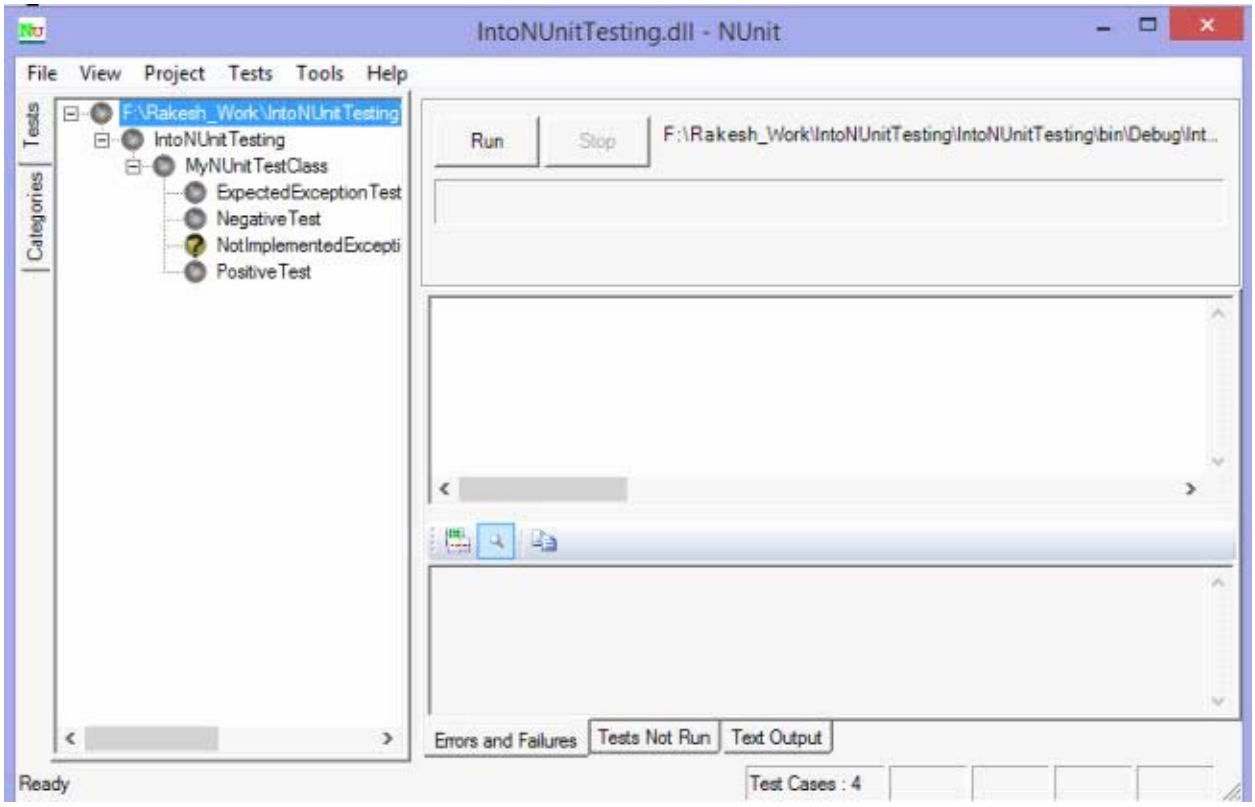

Рисунок 10 – Главное окно программы NUnit

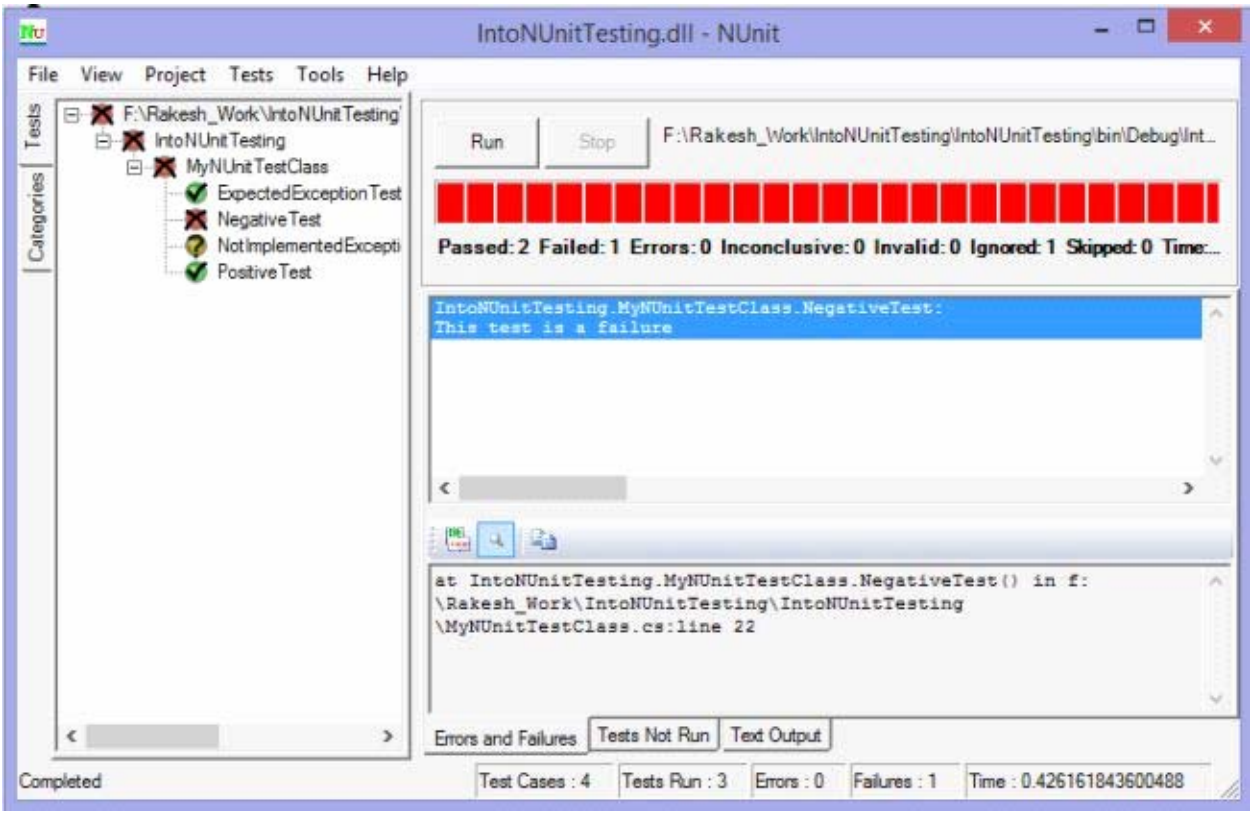

Рисунок 3.6 – Запуск тестов в NUnit.

После запуска тестов отобразится количество тестов, которые не сработали и количество невыполненных тестов.

Для запуска определенного теста, необходимо кликнуть по нему правой кнопкой мыши и выбрать Run test(s). Более подробно работа с NUnit описана в статье [6].

Лабораторная работа №4. Форматирование кода. Метрики кода.

1 Цель работы

Ознакомится с правилами форматирвоания программного кода и изучить метрики кода.

2 Задание

Оформить программный код согласно стандарту кодирования C#. Рассчитать значения используя метрики Холстеда. Для одной из функций посчитать цикломатическое число.

3 Теоретический материал

3.1 Стандарт форматирования кода C#.

В данном методическом пособии приведены фрагменты стандарта(соглашения) по оформлению и написанию кода на языке C#. Более подробно стандарт описан в статье [7].

Приведены основные правила оформления кода и приемы, используемые при написании программ. Соглашения по кодирвоанию предоставляют общие правила, позволяющие сохранить единый стиль написания кода, облегчив тем самым его понимание всеми участниками команды. Вводятся базовые правила написания программ, что позволит повысить предсказуемость выполнения программ, а также избежать ошибок при написании программ новыми участниками команды, не знакомыми с внутренними стандартами разработки.

3.1.1 Стили именования

Pascal case – первая буква каждого слова в имени идентификатора начинается с верхнего регистра.

Пример: TheCategory;

Camel case – первая буква первого слова в идентификаторе в нижнем регистре, все первые буквы последующих слов – в верхнем.

Пример: theCategory;

UpperCase – стиль используется только для сокращений, все буквы в имени идентификатора в верхнем регистре.

Пример: ID;

Hungarian notation – перед именем идентификатора пишется его тип в сокращенной форме.

Пример: strFirstName, iCurrentYear.

3.1.2 Общие правила именования идентификаторов

При именовании идентификаторов не используются аббревиатуры или сокращения, если только они не являются общепринятыми.

Пример: GetWindow(), а не GetWin();

Если имя идентификатора включает в себя сокращение – сокращение пишется в upper case. Исключение - когда имя идентификатора должно быть указано в camel case и сокращение стоит в начале имени идентификатора. В этом случае сокращение пишется в нижнем регистре.

Пример:

PPCAccount (PPC – сокращение от pay per click) для pascal case,

ppcAccount для camel case.

3.1.3 Использование верхнего и нижнего регистра в именах

Запрещается создавать два различных namespace’а, функции, типа или свойства с одинаковыми именами, отличающиеся только регистром. Запрещается создавать функции с именами параметров, отличающимися только регистром. Ниже приведены примеры НЕправильных названий.

Примеры:

KeywordManager и Keywordmanager;

KeywordManager.Keyword и KeywordManager.KEYWORD;

int id {get, set} и int ID {get, set};

findByID(int id) и FindByID(int id);

void MyFunction(string s, string S).

3.1.4 Правила именования классов

Следует избегать имен классов, совпадающих с именами классов .NET Framework. Для классов используется стиль именования pascal case. Для классов, унаследованных от CollectionBase используется суффикс Collection, перед которым указывается тип объектов, для которых используется коллекция.

Пример: UserCollection, CompanyCollection;

В качестве имен классов используются имена существительные. Имя класса не должно совпадать с именем namespace’а.

Пример: namespace Debugging, класс Debug;

Если класс представляет собой сущность, хранимую в базе данных – имя класса соответствует имени таблицы. В этом случае имя класса – это название сущности в единственном числе, имя таблицы – во множественном числе.

Пример: таблица Users, класс User;

При создании классов потомков их имена состоят из имени базового класса и суффикса класса потомка, если суффиксов несколько – они разделяются символом подчеркивания.

Пример:

базовый класс Figure,

потомок FigureCircle;

Имена файлов, в которых находятся классы, совпадают с именами классов. Для именования файлов используется стиль pascal case.

3.1.5 Правила именования интерфейсов

Имена интерфейсов начинаются с буквы I, после которой следует название интерфейса в pascal case.

Пример: IDisposable.

3.1.6 Правила именования generic'ов

Generic'и обозначаются буквой T, если generic'ов несколько их имена начинаются с буквы T.

Пример: GetItems<T>(int parentID).

3.1.7 Правила именования функций

Для именования функций используется стиль pascal case. Функции объявляются согласно следующему шаблону:

<Модификатор доступа> [Другие модификаторы] <Тип> <Название функции>();

Пример: protected abstract void HelloWorld();

Имена функций должны давать четкое представление о том, какое действие эта функция выполняет. Имя функции начинается с глагола, указывающего на то, какое действие она выполняет;

Большие функции, не умещающиеся на одном экране, делятся на несколько private функций меньшего размера, имена таких вспомогательных функций состоят из имени основной (большой) функции и существительного, глагола или фразы, которые уточняют действие вспомогательной функций, разделенные подчеркиванием. Основная и вспомогательная функции объединяются в регионы. Вспомогательные функции вызываются только из основной функции.

Пример:

основная функция – CheckProduct,

вспомогательные функции –CheckProduct_Price, CheckProduct_Url, CheckProduct_SearchTerm.

3.1.8 Правила именования параметров функций

Для именования параметров используется стиль camel case. Имена параметров должны давать четкое представление о том для чего используется параметр, и какое значение следует передать при вызове функции.

Пример:

public void EncodeString(string sourceString, ref string encodedString),

а не public void EncodeString(string string1, ref string string2).

В том случае, когда это не препятствует понимаю кода, в качестве имени параметра функции используется имя соответствующего параметру класса. Для коллекций и массивов используется имя объектов, содержащихся в коллекции или массиве.

Пример:

UserFactory.Create(Company company);

CheckUsers(UserCollection users);

Имена параметров не должны совпадать с именами членов класса, если этого не удается избежать, то для разрешения конфликтов используется ключевое слово this. В именах параметров не используется венгерская нотация.

Пример:

```
public void CreateUser(string firstName, string lastName)
{
    this.firstName  = firstName;
    this.lastName = lastName;
}
```

9 Правила именования свойств

Для именования свойств используется стиль pascal case. Свойства объявляются согласно следующему шаблону: <Модификатор доступа> [Другие модификаторы] <Тип> <Название свойства>;

Пример: public static User CurrentUser { get; }

В том случае, когда это не препятствует понимаю кода, в качестве имени свойства используется имя соответствующего свойству класса. Для коллекций и массивов используется имя объектов, содержащихся в коллекции или массиве.

Пример:

public User User { get; set; }

public UserCollection Users { get; set; }

Название свойства типа bool должно представлять из себя вопрос, требующий ответа да или нет.

Примеры названий: “CanDownload”, “HasKeywords”, “IsChecked”, “NeedsUpdate”.

3.1.10 Правила именования полей

Для именования полей, доступных вне класса, используется стиль pascal case, для private полей - camel case. Поля объявляются согласно следующему шаблону:

<Модификатор доступа> [Другие модификаторы] <Тип> <Название поля>;

Пример: private static User currentUser = null;

В том случае, когда это не препятствует понимаю кода, в качестве имени поля используется имя соответствующего полю класса. Для коллекций и массивов используется имя объектов, содержащихся в коллекции или массиве.

Пример:

public User User = new User();

public UserCollection Users = new UserCollection();

Название поля типа bool должно представлять из себя вопрос, требующий ответа да или нет.

Примеры названий: “CanDownload”, “HasKeywords”, “IsChecked”, “NeedsUpdate”.

3.1.11 Правила именования переменных

Для именования переменных используется стиль camel case. Переменные объявляются согласно следующему шаблону:

<Тип> <Название переменной>;

Пример: int userID = null;

В циклах foreach имя переменной назначается как имя массива в единственном числе.

Пример: foreach(Campaign newCampaign in NewCampaigns).

3.1.12 Правила именования констант

Для именования констант используется стиль pascal case.

3.1.13 Правила именования enum’ов

Для именования enum’ов и их значений используется стиль pascal case. Имена enum’ов указываются в единственном числе. Имена, как правило, состоят из имени сущности, к которой относится enum и названия содержимого enum’а (status, type, state).

Пример: KeywordStatus, ConnectionState, TaskType.

3.1.14 Правила именования исключений

Для exception’ов используется стиль pascal case. Имена классов для создаваемых custom exception’ов заканчиваются суффиком Exception. В качестве имени объекта исключения внтури catch, для исключений типа Exception, используется имя “ex”.

Пример: catch(Exception ex).

3.1.15 Форматирование кода

Используются стандартные настройки форматирования Visual Studio. В одном файле не объявляется больше одного namespace’а и одного класса (исключение – небольшие вспомогательные private классы). Фигурные скобки размещаются всегда на отдельной строке. В условии if-else всегда используются фигурные скобки. Размер tab’а – 4.

Использование строк длиннее 100 символов не желательно. При необходимости инструкция переносится на другую строку. При переносе части кода на другую строку вторая и последующая строки сдвигаются вправо на один символ табуляции.

Каждая переменная объявляется на отдельной строке;

Все подключения namespace’ов (using) размещаются в начале файла, системные namespace’ы объявляются над custom namespace’ами.

Если для свойства существует соответствующее поле (например, при загрузке по требованию), то поле объявляется над свойством.

Пример:

```
private User user;
public User User { get; set; }
```

Если set или get свойства состоит из одной операции – весь set или get размещается на одной строке.

Пример:

```
Public User
{
```

```
    get     {     return user;     }
}
```

Функции, поля и свойства группируются внутри класса по своему назначению. Такие группы объединяются в регионы.

3.1.16 Комментирование кода

Все комментарии должны быть на русском языке.

Для функций, классов, enum’ов, свойств и полей комментарии необходимо указывать в таком виде, чтобы по ним можно было автоматически сгенерировать документацию. Для этого используются стандартные tag’и такие как <summary>, <param> и <return>.

Для функций создающих exception’ы – возможные исключения необходимо указывать в tag’ах <exception>.

Для включения в документацию примеров использования необходимо применять tag’и <example>, <remarks> и <code>.

Для ссылок в документации необходимо использовать tag’и <see cref=””/> и <seeAlso cref=””/>.

При использовании в тексте комментариев символов, использующихся в xml как спецсимволы, необходимо использовать tag CDATA.

Комментарии к заголовкам функций, свойств, полей, интерфейсов и прочего необходимо указывать всегда.

Для функций, выполняющих сложные алгоритмы, не очевидные для восприятия, необходимо указывать подробные комментарии не только к заголовку функции, но и самому алгоритму с пояснением каждого шага выполнения алгоритма.

В случае внесения изменений в критические участки кода, ядро системы, либо когда сложно проследить последствия, которые может повлечь такое изменение, необходимо указывать подробный комментарий о том кто внес изменение, когда и по какой причине.

В том случае если необходимо временно добавить заплатку, без которой система не может работать, но заплатку в дальнейшем планируется убрать – необходимо добавлять ключевое слово “//TODO: ”, после которого указывается когда и что должно быть исправлено. Кроме этого необходимо указывать подробный комментарий о том, для чего предназначено временное исправление, кто его внес и когда.

При разработке кода, изменения которого могут повлечь за собой сбой в других частях системы, при этом связь между этими двумя частями программы неочевидна и ошибка не будет показана на этапе компиляции, либо если сам код неочевидным образом зависит от других частей системы – необходимо указывать подробное описания взаимосвязей.

3.1.17 Конфигурация

В конфигурационном файле ключи необходимо группировать по назначению. Перед началом каждой такой группы в комментариях необходимо указывать открывающий tag с названием группы, в конце – закрывающий tag.

Пример:

```
  <!--Connection strings -->
      <add key="MainDatabaseConnectionString" value="server=...;Integrated Security=SSPI;"/>
      <add key=" SupportDatabaseConnectionString" value="server=...;Integrated Security=SSPI;"/>
  <!-- /Connection strings -->
```

Для ключей, хранящих boolean значения, value может быть равно только true или false, а не 0/1 или yes/no.

3.1.18 Переменные и типы

Свойства необходимо использовать только тогда, когда это имеет смысл. Если при получении и сохранении значений никакая дополнительная логика не участвует – вместо свойства необходимо использовать поле (исключение – когда класс bind'ится на asp.net страницах, т.к. стандартный механизм bind'а имеет доступ только к public свойствам).

Необходимо использовать максимально простые типы данных. Так, например, необходимо использовать int, а не long, если известно, что для хранимых значений будет достаточно типа int.

Константы необходимо использовать только для простых типов данных.

Для сложных типов вместо констант необходимо использовать readonly поля.

Boxing и unboxing value типов необходимо использовать только когда это действительно необходимо.

При задании значений нецелых типов, значения должны содержать как минимум одну цифру до точки и одну после.

Необходимо использовать именования типов C#, а не .NET common type system (CTS).

Пример: int userID = -1; , а не Int32 userID=-1;

Модификаторы доступа необходимо указывать всегда. Не смотря на то, что по умолчанию назначается модификатор доступа private, поле модификатора не остается пустым – модификатор private необходимо указывать явным образом.

Пример: private User CreateUser(string firstName, string lastName);

Модификаторы доступа (private, protected, internal и public) необходимо указывать в зависимости от того, где требуется доступность соответствующего поля, свойства, функции, класса или конструктора. Модификатор public необходимо указывать только тогда, когда необходим доступ к полю из других проектов, internal – когда необходим доступ из

других классов внутри одного проекта, protected – для предоставления доступа классам – потомкам, во всех остальных случаях используется private, т.е. доступ ограничивается самим классом.

Поля и переменные инициализируются при их объявлении, когда это возможно.

Пример:

```
private int  userID = -1;
private string firstName = “”;
private string lastName = “”;
```

Когда для создания класса необходимо передать параметры, используемые при его инициализации, на конструктор по умолчанию (MyClass() { }) необходимо накладывать модификатор доступа private, чтобы избежать создания клиентами неинициализированного объекта.

Пример:

```
private User()
{       }
public User(int userID)
{       }
```

Вместо использования “magic numbers” для идентификаторов статусов, состояний и т.п. необходимо указывать константы или enum’ы. Идентификаторы состояний в виде чисел использовать нельзя.

Пример не правильного использования: public GetUserByStatus(int statusID);

Пример правильного использования: public GetUserByStatus(UserStatus userStatus);

Тип object необходимо использовать только когда это действительно необходимо, в большинстве случаев вместо него используются generic’и. Вместо Hashtable необходимо использовать Dictionary<>, вместо ArrayList используется List<>.

В том случае если в get’е или set’е какого-либо свойства выполняются сложные вычисления, если операция, выполняемая в get или set является преобразованием, имеет побочный эффект или долго выполняется – свойство должно быть заменено функциями.

Свойство не должно менять своего значения от вызова к вызову, если состояние объекта не изменяется. Если результат при новом вызове может быть другим при том же состоянии объекта, вместо свойства необходимо использовать функции.

Внутри get’а и set’а не должно быть обращений к коду, не связанному напрямую с получением или сохранением значения свойства, т.к. такие действия могут быть не очевидны для клиентов, использующих свойство.

Все настройки, влияющие на работу приложения, нельзя указывать жестко в коде, а необходимо выносить в config. Если есть возможность прописать значение по умолчанию – они должны быть прописаны, если значение по умолчанию не может быть задано и соответствующий ключ не прописан в config – необходимо создавать исключение.

3.1.19 Функции

Функции, возвращающие массив, всегда должны возвращать массив. Если нет данных – функции возвращают пустой массив, но не null. Это же касается коллекций;

В функциях никогда нельзя использовать больше 7-ми параметров. Если параметров больше – они объединяются в класс.

3.1.20 Управление выполнением программы

При использовании foreach по коллекции объектов – саму коллекцию никогда нельзя модифицировать (новые элементы не добавляются, существующие не удаляются).

Если задачу можно решить, используя рекурсию и используя циклы, предпочтение необходимо отдавать использованию циклов. Рекурсия необходимо применять только тогда, когда решение с использованием циклов сложнее, чем при использовании рекурсии.

Тернарные операции необходимо использовать только для простых проверок. В тех случаях, когда проверка сложная и включает в себя несколько условий – используются if/else.

В aspx файлах нельзя использовать код. Т.е. tag'и <%= "C# code" %> использовать нельзя. Это связано с тем, что при компиляции такой код не проверяется.

Сложные проверки, состоящие из множества условий, необходимо разбивать на несколько простых. Для сохранения промежуточных результатов необходимо использовать boolean переменные.

Классы, реализующие интерфейс IDisposable, создаются в директиве using.

Пример: using(SqlConnection sqlConnection = new SqlConnection) { }.

3.1.21 Исключения и их обработка

Блоки try-catch нельзя использовать для управления ходом работы программы, а только для обработки непредвиденных ошибок.

При прокидывании исключений выше по StackTrace'у, необходимо использовать оператор "throw;", а НЕ "throw ex;".

Custom exception'ы необходимо наследовать от класса Exception (а не ApplicationException или какого-то другого).

Исключения необходимо создавать всякий раз, когда функция не может быть выполнена – переданы неверные параметры при вызове функции, нет доступа к базе данных, неизвестные идентификаторы и т.п.

Все исключения должны быть записаны в log или показаны пользователю системы. Пустые секции catch использовать нельзя.

При записи информации об ошибке, как правило, пишется StackTrace.

3.2 Метрики Холстеда

В отличие от большинства отраслей материального производства, в вопросах проектов создания ПО недопустимы простые подходы, основанные на умножении трудоемкости на среднюю производительность труда. Это вызвано, прежде всего, тем, что экономические показатели проекта нелинейно зависят от объема работ, а при вычислении трудоемкости допускается большая погрешность.

Поэтому для решения этой задачи используются комплексные и достаточно сложные методики, которые требуют высокой ответственности в применении и определенного времени на адаптацию (настройку коэффициентов).

Современные комплексные системы оценки характеристик проектов создания ПО могут быть использованы для решения следующих задач:

- предварительная, постоянная и итоговая оценка экономических параметров проекта: трудоемкость, длительность, стоимость;

- оценка рисков по проекту: риск нарушения сроков и невыполнения проекта, риск увеличения трудоемкости на этапах отладки и сопровождения проекта и пр.;

- принятие оперативных управленческих решений – на основе отслеживания определенных метрик проекта можно своевременно предупредить возникновение нежелательных ситуаций и устранить последствия непродуманных проектных решений.

Метрики сложности программ принято разделять на три основные группы:

- метрики размера программ;

- метрики сложности потока управления программ;

- метрики сложности потока данных программ.

Метрики первой группы базируются на определении количественных характеристик, связанных с размером программы, и отличаются

относительной простотой. К наиболее известным метрикам данной группы относятся число операторов программы, количество строк исходного текста, набор метрик Холстеда. Метрики этой группы ориентированы на анализ исходного текста программ. Поэтому они могут использоваться для оценки сложности промежуточных продуктов разработки.

Метрики второй группы базируются на анализе управляющего графа программы. Представителем данной группы является метрика Маккейба.

Управляющий граф программы, который используют метрики данной группы, может быть построен на основе алгоритмов модулей. Поэтому метрики второй группы могут применяться для оценки сложности промежуточных продуктов разработки.

Метрики третьей группы базируются на оценке использования, конфигурации и размещения данных в программе. В первую очередь это касается глобальных переменных. К данной группе относятся метрики Чепина.

Метрика Холстеда относится к метрикам, вычисляемым на основании анализа числа строк и синтаксических элементов исходного кода программы.

Основу метрики Холстеда составляют четыре измеряемые характеристики программы:

- NUOprtr (Number of Unique Operators) — число уникальных операторов программы, включая символы-разделители, имена процедур и знаки операций (словарь операторов);

- NUOprnd (Number of Unique Operands) — число уникальных операндов программы (словарь операндов);

- Noprtr (Number of Operators) — общее число операторов в программе;

- Noprnd (Number of Operands) — общее число операндов в программе.

На основании этих характеристик рассчитываются оценки:

- Словарь программы (Halstead Program Vocabulary, HPVoc): HPVoc = NUOprtr + NUOprnd;

- Длина программы (Halstead Program Length, HPLen): HPLen = Noprtr + Noprnd;

- Объем программы (Halstead Program Volume, HPVol): HPVol = HPLen log2 HPVoc;

- Сложность программы (Halstead Difficulty, HDiff): HDiff = (NUOprtr/2) × (NOprnd / NUOprnd);

- На основе показателя HDiff предлагается оценивать усилия программиста при разработке, при помощи показателя HEff (Halstead Effort): HEff = HDiff × HPVol.

3.3 Расчет цикломатического числа.

Показатель цикломатической сложности является одним из наиболее распространенных показателей оценки сложности программных проектов. Данный показатель был разработан ученым Мак-Кейбом в 1976 г., относится к группе показателей оценки сложности потока управления программой и вычисляется на основе графа управляющей логики программы (control flow graph). Данный граф строится в виде ориентированного графа, в котором вычислительные операторы или выражения представляются в виде узлов, а передача управления между узлами – в виде дуг.

Показатель цикломатической сложности позволяет не только произвести оценку трудоемкости реализации отдельных элементов программного проекта и скорректировать общие показатели оценки длительности и стоимости проекта, но и оценить связанные риски и принять необходимые управленческие решения.

Упрощенная формула вычисления цикломатической сложности представляется следующим образом:

$$C = e – n + 2,$$

где e – число ребер, а n – число узлов на графе управляющей логики.

Как правило, при вычислении цикломатической сложности логические операторы не учитываются.

В процессе автоматизированного вычисления показателя цикломатической сложности, как правило, применяется упрощенный подход, в соответствии с которым построение графа не осуществляется, а вычисление показателя производится на основании подсчета числа операторов управляющей логики (if, switch и т.д.) и возможного количества путей исполнения программы.

Цикломатическое число Мак-Кейба показывает требуемое количество проходов для покрытия всех контуров сильносвязанного графа или количества тестовых прогонов программы, необходимых для исчерпывающего тестирования по принципу «работает каждая ветвь».

Показатель цикломатической сложности может быть рассчитан для модуля, метода и других структурных единиц программы.

Существует значительное количество модификаций показателя цикломатической сложности.

- «Модифицированная» цикломатическая сложность – рассматривает не каждое ветвление оператора множественного выбора (switch), а весь оператор как единое целое.

- «Строгая» цикломатическая сложность – включает логические операторы.

- «Упрощенное» вычисление цикломатической сложности – предусматривает вычисление не на основе графа, а на основе подсчета управляющих операторов.

Подробнее о метриках ПО можно узнать в статье [8]

Список используемых источников:

1. Гост 34.602-89 техническое задание на создание автоматизированной системы [Электронный ресурс] / http://it-gost.ru. – [2015]. - Режим доступа : http://it-gost.ru/content/view/21/39/.

2. Гост 19.201-78 техническое задание. Требования к содержанию и оформлению [Электронный ресурс] / http://it-gost.ru. – [2015]. - Режим доступа : http://it-gost.ru/content/view/20/41/.

3. Что такое ГОСТ [Электронный ресурс] / http://it-gost.ru. – [2015]. - Режим доступа : http://it-gost.ru/content/view/76/51/.

4. Страница загрузок TortoiseHg [Электронный ресурс] / http://sourceforge.net – [2015]. - Режим доступа : http://sourceforge.net/project/showfiles.php?group_id=199155&package_id=236242&release_id=654501.

5. Сайт проекта NUnit [Электронный ресурс] / http://www.nunit.org/. – [2015]. - Режим доступа : http://www.nunit.org/.

6. Вступление в Nunit [Электронный ресурс] / http://itvdn.com – [2015]. - Режим доступа : http://itvdn.com/ru/blog/article/entry-into-nunit.

7. Стандарты и правила оформления кода C# [Электронный ресурс] / http://michaelsmirnov.blogspot.ru/ – [2015]. - Режим доступа : http://michaelsmirnov.blogspot.ru/2011/01/c.html.

8. Метрики кода и их практическая реализация в Subversion и ClearCase. Часть 1 – метрики [Электронный ресурс] / http://cmcons.com – [2015]. - Режим доступа :http://cmcons.com/articles/CC_CQ/dev_metrics/mertics_part_1/.

План выпуска учеб.-метод. документ. 2016 г., поз. 17

Публикуется в авторской редакции

Подписано в свет 06.04.2016

Гарнитура «Таймс». Уч.-изд. л. 1,0. Объем данных 0,6 Мбайт.

PC 486 DX-33; Microsoft Windows XP; Internet Explorer 6.0; Adobe Reader 6.0

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Волгоградский государственный архитектурно-строительный университет»
400074, Волгоград, ул. Академическая, 1
http://www.vgasu.ru, info@vgasu.ru