МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Кузбасский государственный технический университет имени Т. Ф. Горбачева»

Кафедра информационных и автоматизированных производственных систем

Составитель
А. В. Матисов

ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Методические материалы

Рекомендовано цикловой методической комиссией специальности
СПО 09.02.07 Информационные системы и программирование
в качестве электронного издания
для использования в образовательном процессе

Кемерово 2018

Рецензенты

Ванеев О. Н. – кандидат технических наук, доцент кафедры информационных и автоматизированных производственных систем

Чичерин И. В. – кандидат технических наук, доцент кафедры информационных и автоматизированных производственных систем

Матисов Александр Вениаминович

Инструментальные средства разработки программного обеспечения: методические материалы [Электронный ресурс] для студентов специальности СПО 09.02.07 Информационные системы и программирование очной формы обучения / сост. А. В. Матисов; КузГТУ. – Электрон. издан. – Кемерово, 2018.

Приведено содержание практических и лабораторных занятий, материал, необходимый для успешного изучения дисциплины.

Назначение издания – помощь обучающимся в получении знаний по дисциплине «Инструментальные средства разработки программного обеспечения» и организация практических и лабораторных занятий.

СОДЕРЖАНИЕ

1 РАЗРАБОТКА СТРУКТУРЫ ПРОЕКТА

1.1 ЦЕЛЬ РАБОТЫ

Цель работы – освоить процедуру создания иерархической структуры проекта, освоить процедуры установки, удаления и изменения связей между задачами, ознакомиться с различными типами связей между задачами.

1.2 ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ

Когда цели проекта в основном ясны, можно переходить к разработке структуры проекта – комплекса работ (задач), которые необходимо выполнить для достижения поставленных целей.

Одним из методов первоначального проектирования комплекса задач проекта является метод «мозгового штурма». Данный метод применяется в том случае, если ставящиеся в проекте цели настолько новы, что Вы имеете только общее представление о задачах, которые могут встать в процессе реализации проекта. В этом случае менеджер проекта проводит опрос экспертов или просто собирает совещание, на котором компетентные в различных областях реализации проекта лица генерируют задачи, которые необходимо будет решать. Обычно сначала составляется список основных задач и ставятся задания по разбиению их на более детальные.

Последовательная детализация задач проекта требует сбора дополнительной информации и, возможно, дополнительных консультаций со специалистами. Данный метод еще называют планированием сверху-вниз. Областями наиболее частого применения этого метода являются создание новых объектов техники, новых технологий или новых организационных структур. И наоборот, если Вы планируете деятельность в хорошо известной области, и пути достижения целей ясны, то менеджеру достаточно сконструировать комплекс задач проекта из определенных работ, уже описанных ранее, оцененных по затратам времени и ресурсов. Данный метод применяется в строительстве, а также для планирования деятельности организации, реализующей однотип-

ные проекты (например, съемки рекламных роликов или фильмов).

Итак, разработка комплекса задач проекта является первым шагом в создании детального расписания. Исходными данными для процедуры разработки комплекса задач являются поставленные перед проектом цели. Цель нашего проекта – разработка нового программного обеспечения. Формально процесс разработки комплекса задач представляет собой детализацию деятельности по достижению поставленных целей до отдельных, поддающихся оценке и управлению шагов (операций, работ). В TimeLine такой шаг (операция, работа) называется задачей. Если предстоящий проект достаточно прост, то и разработка комплекса составляющих его задач не составляет труда.

Начальный вариант комплекса задач может быть разработан в результате совещания менеджеров, составляющих команду проекта. В этом случае на основе опыта участников проекта генерируется список работ, которые потребуется выполнить для достижения поставленных целей. Данный список работ может выглядеть, например, следующим образом:

- проектирование;
- программирование;
- тестирование;
- документация;
- внедрение.

Водим названия перечисленных выше задач в электронную таблицу диаграммы Гантта. Для этого необходимо установить курсор в столбце Задача, ввести название задачи (например, проектирование) и нажать клавишу Enter. В результате строка с названием задачи появилась в электронной таблице, а курсор переместился на следующую строку. Таким же образом вводятся названия остальных четырех задач, приведенные выше.

Введенные названия задач выглядят несколько расплывчато и неопределенно. Значит, их нужно детализировать на более конкретные задачи. Например, задачу проектирование можно детализировать следующим образом:

- Получение заказа.
- Заключение договора.
- Обсуждение заказа.

– Опрос пользователей.
– Черновик технического задания (ТЗ).
– Обсуждение ТЗ.
– Оформление ТЗ.
– Согласование ТЗ с заказчиком.
– Исправление ТЗ.
– Утверждение ТЗ.

Задача, имеющая входящие в нее задачи более низкого уровня, называется составной (родительской). Задача, не подразделяющаяся на задачи более низкого уровня, называется детальной (простой, дочерней).

Детализирующая задача может, в свою очередь, являться родительской задачей для задач более низкого уровня.

Комплекс задач проекта:

– Проектирование
- Получение заказа
- Заключение договора
- Обсуждение заказа
- Опрос пользователей
- Черновик технического задания (ТЗ)
- Обсуждение ТЗ
- Оформление ТЗ.
- Согласование ТЗ с заказчиком
- Исправление ТЗ
- Утверждение ТЗ
– Программирование
- Пользовательский интерфейс
- Программирование
- Отладка
- Передача на тестирование
- Устранение ошибок
- Блок расчета
- Программирование
- Отладка
- Передача на тестирование
- Устранение ошибок
- Драйверы
- Программирование

- Отладка
- Передача на тестирование
- Устранение ошибок
- Библиотеки
- Программирование
- Отладка
- Передача на тестирование
- Устранение ошибок
- Сборка системы
- Совместная отладка системы
- Передача на тестирование
- Устранение ошибок
- Передача на тестирование заказчику
- Доработки
- Передача на тестирование

– Тестирование

- Разработка плана тестирования
- Первый этап тестирования
- Пользовательский интерфейс
- Тестирование
- Возврат разработчику
- Блок расчета
- Тестирование
- Возврат разработчику
- Драйверы
- Тестирование
- Возврат разработчику
- Библиотеки
- Тестирование
- Возврат разработчику
- Второй этап тестирования
- Тестирование
- Возврат разработчикам
- Тестирование у заказчика
- Завершающий этап тестирования

– Документация

- Первый вариант
- Рецензирование

- Учет замечаний рецензентов
- Учет доработок
- Документация готова
– Внедрение
- Установка ПО
- Обучение пользователей
- Система принята

Для получения информации по проекту с определенным уровнем детализации удобно использовать такие возможности, как свертывание и развертывание составных задач. Свернуть составную задачу означает убрать с экрана простые задачи, являющиеся ее компонентами. Свернутые задачи помечаются знаком «+» в списке задач. Развернуть составную задачу означает снова отобразить на экране составляющие ее детальные задачи. Time-Line позволяет сворачивать и разворачивать как отдельные составные задачи проекта, так и все одновременно.

Для свертки одной задачи необходимо щелкнуть на ее названии дважды левой клавишей мыши. Таким же образом производится развертка свернутой задачи.

Для свертки всех задач проекта выберите в меню Иерархия команду Свернуть все или нажмите кнопку  , расположенную на панели инструментов. Развернуть все задачи проекта, можно выбрав в меню Иерархия команду Развернуть все или нажав кнопку  , расположенную на панели инструментов.

Можно отображать на экране информацию только о выделенной задаче, интересующей нас в данный момент. Этот режим активизируется с помощью команды Крупный план из меню Иерархия или с помощью кнопки  , расположенной на панели инструментов. Для отображения информации обо всех задачах используется команда Общий план из меню Иерархия или кнопка Общий план, расположенная справа в нижней части экрана.

Определим тип каждой задачи и добавим пояснения к отдельным задачам (таблица 1.1).

Связь по времени отображает в расписании логическую зависимость между работами. Наиболее частой причиной таких зависимостей являются технологические ограничения (начало одних работ зависит от результатов других), хотя возможны и огра-

ничения, диктуемые другими соображениями. Связи между работами образуют структуру сети. Комплекс взаимосвязей между работами также часто называют логической структурой проекта, поскольку он определяет последовательность выполнения работ. В соответствии с установленными связями работы делятся на предшествующие и последующие. Предшествующая работа является обеспечивающей для последующей работы, т.е. для начала выполнения последующей работы требуется выполнение всех предшествующих.

Таблица 1.1

Задача	Заметки
…	
Программирование	На Visual C++
…	
Тестирование	
…	
Первый этап тестирования	Поручить испытателю ПО
…	
Второй этап тестирования	Поручить испытателю ПО
…	
Завершающий этап тестирования	Участвует заказчик
…	
Внедрение	
…	
Система принята	Составить акт приемки в эксплуатацию

Разработка корректной структуры связей между задачами – достаточно непростая процедура, особенно для крупных проектов. Часто менеджер не может заранее определить правильный набор взаимосвязей между работами. Рекомендуется проводить совещания по сетевому планированию, чтобы определить взаимосвязи между работами и последовательность выполнения работ. Особенно, если к проекту привлекаются различные сторонние организации. Основное внимание уделяется определению последовательных и параллельных работ и ограничений, которые накладываются на параллельные работы.

Для планирования зависимостей между задачами могут использоваться четыре типа связей предшествования:

– Конец-Начало. Это наиболее часто встречающаяся зависимость, при которой последующая задача не может быть начата, пока не завершена предшествующая.

– Начало-Начало. Эта зависимость чаще применяется для моделирования работ, которые должны выполняться одновременно. В этом случае для начала задачи необходимо, чтобы предшествующая задача только началась и, может быть, проработала в течение определенного периода, называемого временем задержки.

– Конец-Конец. Этот тип взаимосвязи используется, если окончание последующей работы зависит от окончания некоторой работы-предшественницы, но начинаться работы могут независимо.

– Начало-Конец. Этот тип используется редко, но он может быть полезен, когда при планировании требуется задержать окончание работы на как можно более длительный срок, связав ее окончание с началом другой работы.

Временная связь может быть установлена между любыми двумя задачами, в том числе, принадлежащими к разным иерархическим уровням проекта. Нельзя установить связь только между иерархически зависимыми задачами, т.е. между составной (родительской) и ее дочерней задачами. Это логически очевидно.

Иногда, планируя крупные проекты, удобно сначала установить связи между задачами верхнего уровня, а затем – между детальными задачами внутри составных задач верхнего уровня. Очевидно, что задача Программирование должна начаться после окончания задачи Проектирование. Установим между этими задачами связь типа Конец-Начало. Для этого выполните следующие действия:

После того как связь установлена, ее можно отобразить на временной диаграмме Гантта. Для этого нажмите кнопку  , расположенную на панели инструментов. Повторное нажатие этой кнопки скрывает связи, отображаемые на диаграмме Гантта.

Описанная выше операция позволяет установить жесткую связь типа Конец-Начало. Аналогичным образом устанавливаются связи между задачами Тестирование и Внедрение, а также между задачами Документация и Внедрение.

Часто на практике оказывается, что между задачами должна быть установлена нежесткая связь. Под нежесткой связью понимается связь с перекрытием или задержкой. Допустим, последующая работа не может начаться раньше, чем через два дня после окончания предшествующей работы (связь с задержкой). Если последующая работа должна начаться за два дня до окончания предшествующей работы, то это связь с перекрытием. Нежесткая связь может быть установлена путем модификации существующей жесткой связи. Задать или модифицировать любой тип связи и определить задержку или перекрытие, если связь нежесткая, можно с помощью формы Временные характеристики.

Предположим, что после завершения проектирования программистам требуется пять дней, чтобы изучить техническое задание и распределить между собой работы по программированию. Только после этого каждый из них приступит к программированию своей подзадачи. Таким образом, между задачами Проектирование и Программирование необходимо установить нежесткую связь с задержкой в пять дней. Выполним установку такой связи путем модификации существующей жесткой связи.

Теперь установим связи между детальными задачами внутри составной задачи Проектирование. Все эти детальные задачи должны выполняться последовательно друг за другом, все связи между ними имеют один и тот же тип.

Следует отметить, что связываемые таким образом задачи необязательно должны располагаться в таблице последовательно друг за другом. В таком случае при выделении задач щелчками левой клавиши мыши необходимо удерживать нажатой клавишу Ctrl. Важно, чтобы при выделении группы задач соблюдался порядок их следования.

Особую роль в процессе автоматической установки и редактирования временных связей играет опция Инструктор. Инструктор анализирует предпринятые действия. При неоднозначности установки он выдает на экран диалоговое окно, запрашивающее уточняющую информацию по конкретной ситуации.

Инструктор является необходимым средством при составлении расписания начинающими пользователями, поскольку он отслеживает разного рода спорные ситуации и их последствия. Следует помнить, что при отключенной опции Инструктор сооб-

щение о существовании связей не выводится. Поэтому возможность удаления связей таким способом при отключенной опции Инструктор исключена.

Для отключения или включения опции Инструктор необходимо воспользоваться командой Глобальные параметры из меню Настройки. Если опция Инструктор включена, то соответствующая кнопка на панели инструментов имеет вид:  . Если же эта опция отключена, то кнопка выглядит так:  . Отключение или включение опции Инструктор можно выполнить, нажав на эту кнопку.

Обсуждая установку логических связей между задачами, нельзя не сказать о циклах. В результате ошибочной установки зависимостей между задачами в сети может возникнуть замкнутая последовательность задач (цикл). С точки зрения логики сетевого графика эта ситуация недопустима (получается, что задачи зависят друг от друга и, следовательно, каждая – от себя самой). Система не допустит возникновения подобной ситуации. Система сигнализирует пользователю о возникновении цикла (если установлен автоматический пересчет сети, то сразу, а если ручной, то после пересчета сети).

1.3 ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Данное занятие предполагает выполнение следующих этапов:

– изучить методические указания;
– выполнить выданное задание;
– ответить на контрольные вопросы.

1.4 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какая задача называется составной (родительской)?
2. Какая задача называется детальной (простой, дочерней)?
3. Как вставить новую задачу в расписание?
4. Сколько уровней иерархии может быть использовано?
5. Как отключить отображение иерархии задач в электронной таблице?
6. Как восстановить иерархию задач?

7. Какие типы связей предшествования Вы знаете? Что означает каждый из них?

8. Можно ли установить связь между задачами, принадлежащими к разным иерархическим уровням проекта?

9. Между какими задачами связь установить нельзя?

10. Какие типы нежесткой связи Вы знаете? Что они обозначают?

2 РАЗРАБОТКА МОДУЛЬНОЙ СТРУКТУРЫ ПРОЕКТА (ДИАГРАММЫ МОДУЛЕЙ)

2.1 ЦЕЛЬ РАБОТЫ

Цель работы – изучение процесса разработки модульной структуры программного обеспечения.

2.2 ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ

На этапе проектирования осуществляется построение модели реализации (или физической модели), демонстрирующей, как система будет удовлетворять предъявленным к ней требованиями (без технических подробностей). Одним из самых распространенных является метод структурного проектирования.

Техника структурных карт (схем) используется на фазе проектирования для того, чтобы продемонстрировать, каким образом системные требования будут отражаться комбинацией программных структур. При этом наиболее часто применяются две техники: структурные карты Константайна (Constantine), предназначенные для описания отношений между модулями, и структурные карты Джексона (Jackson), предназначенные для описания внутренней структуры модулей.

Структурные карты Константайна являются моделью отношений иерархии между программными модулями. Узлы структурных карт соответствуют модулям и областям данных, потоки изображают межмодульные вызовы. Базовым элементом структурной карты является модуль.

Возможно использовать различные типы модулей:

– модуль: используется для представления обрабатывающего фрагмента для его локализации на диаграмме (рисунок 2.1, а).

– подсистема: ранее определенный модуль, детализированный посредством декомпозиции ранее определенных диаграмм. Может повторно использоваться любое число раз на любых структурных картах (рисунок 2.1, б).

– библиотека: отличается от подсистемы тем, что определена вне проекта данной системы (рисунок 2.1, в).

Взаимодействие между модулями может осуществляться как непосредственно, так и с передачей данных и параметров управления. При графическом представлении карты передача данных изображается в форме стрелки с незакрашенным кружком на конце, а передача управления – в виде стрелки с закрашенным кружком на конце.

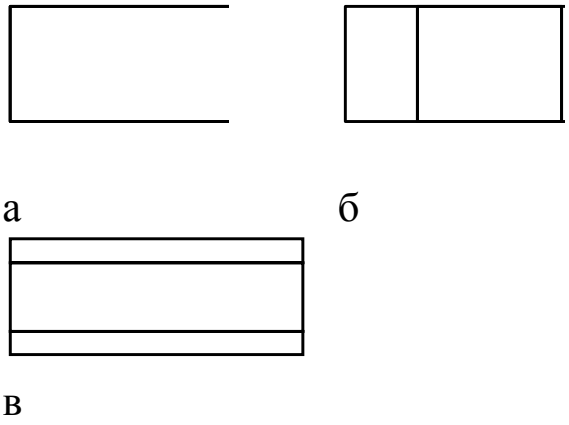

Рисунок 2.1 – Нотация структурных карт Константайна

В качестве примера рассмотрим структурную карту Константайна, описывающую процесс оформления некоторого заказа (рисунок 2.2). Базовым модулем является процедура оформления заказа, вызывающая, соответственно, процесс управления оформлением заказа. Из последнего модуля, в свою очередь, последовательно могут быть вызваны последующие модули, описанные как подсистемы. Практически все модули передают или принимают данные или параметры управления. Так, например, процесс «Проверка пароля» принимает на вход данные о введенном пользователем пароле, а в ответ на основании результатов проверки инициируют породивший его процесс управления оформлением заказа, переводя его на новый уровень.

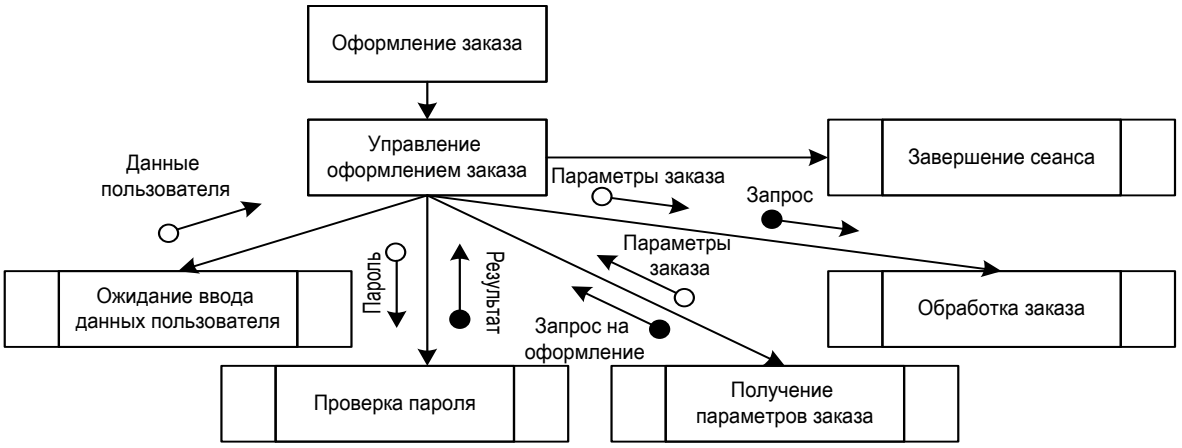

Рисунок 2.2 – Пример структурной карты Константайна

Техника структурных карт Джексона основана на методологии структурного программирования Джексона и заключается в продуцировании диаграмм (структурных карт) для графического иллюстрирования внутримодульных (а иногда и межмодульных) связей и документирования проекта архитектуры системы ПО. При этом техника позволяет осуществлять проектирование нижнего уровня структуры ПО и на этом этапе является близкой к традиционным блок-схемам.

По аналогии со структурными картами Константайна диаграмма Джексона может включать объекты следующих типов:

– структурный блок (базовая компонента методологии) представляет частную функцию или блок кодов с одним входом и одним выходом;

– процедурный блок является специальным видом структурного блока, представляющим вызов ранее определенной процедуры;

– библиотечный блок аналогичен процедурному и представляет вызов библиотечного модуля.

Для разделения блоков карт Константайна и Джексона графические блоки последней нотации целесообразно сопроводить горизонтальной линии, размещаемой под верхней гранью блока. Пример использования структурных карт Джексона приведен на рисунке 2.3. Он аналогичен рассмотренному выше примеру с картой Константайна.

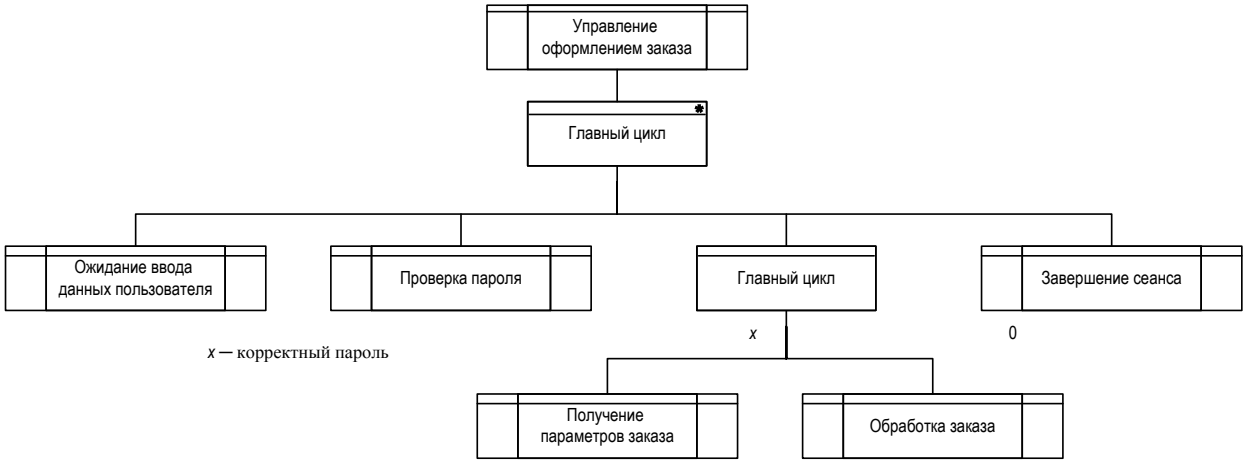

Рисунок 2.3 – Пример структурной карты Джексона

На основании технического проекта (а именно, перечня автоматизируемых функций и функциональной модели работы автоматизированной системы) необходимо выполнить рабочее проектирование автоматизированной системы.

Поскольку рассматривается гипотетическое предприятие, разработку рабочего проекта предлагается свести к проектированию архитектуры программного обеспечения. Для этого необходимо предусмотреть отдельные модули обработки каждой функции и подфункции, заявленной в техническом проекте. Предварительно нужно составить перечень шагов, которые должен пройти пользователь в процессе решения соответствующих задач посредством автоматизируемой системы.

Для описания модульной структуры программного обеспечения воспользоваться структурными картами Константайна или Джексона. Для демонстрации работоспособности предложенной модульной системы разработать экранные формы, отражающие последовательное развитие одного из модулей спроектированных структурных карт.

В соответствии с требованиями, предъявляемыми техническим заданием, и результатами внешнего проектирования разработаем модульную структуру подсистемы обслуживания клиента по его кредитной карте в банкомате.

В составе программного обеспечения можно выделить следующие программные модули: Головной модуль (Main module), Модуль управления устройством считывания кредитной кары (Credit cart control module), Модуль аутентификации

(Autentification module) и Модуль получения и обработки запроса на обслуживание (Reception and processing module). Кроме этого в состав ПО необходимо включить модуль данных кредитной карты (Credit cart data).

Основной функцией Головного модуля является организация общего управления поведением подсистемы и выполняет вызов всех остальных программных модулей.

Модуль управления устройством считывания кредитной карты выполняет функции связанные с обработкой кредитной карты: ввод, считывание хранящейся на ней информации, удаление.

Модуль аутентификации выдает сообщение клиенту на ввод ключевых данных, выполняет получение пароля и проверку его правильности.

Модуль получения и обработки запроса на обслуживание выполняет следующие функции: Получение запроса на обслуживание и проверка возможности его исполнения, Обработка запроса на обслуживание, включающая такие действия как:

– обработка внутренней банковской документации по клиенту;

– распечатка баланса клиента;

– выдача наличных денег и информирование компьютера банка об изъятых из банка деньгах;

– распечатка операции клиента.

На рисунке 2.4 приведена структурная карта, демонстрирующая отношения между указанными модулями системы.

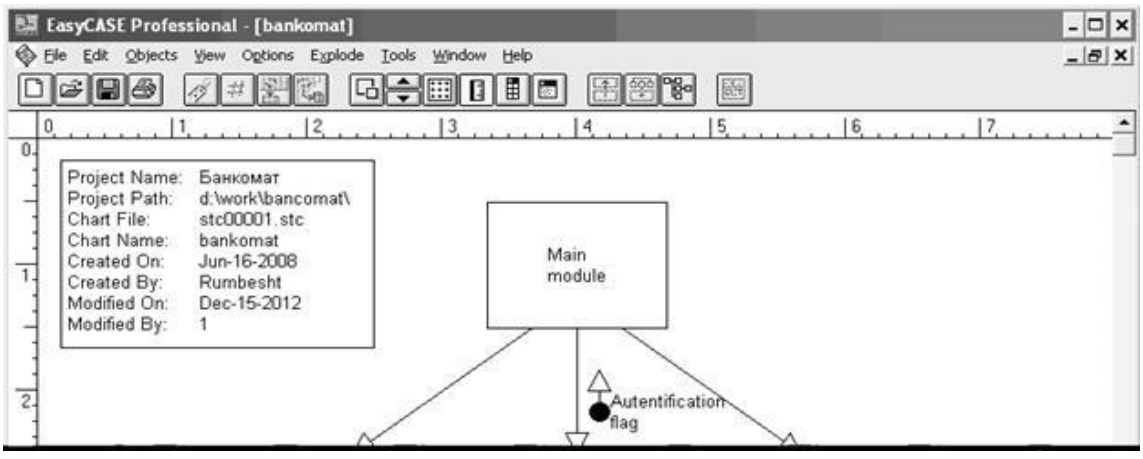

Рисунок 2.4 – Модульная структура программного обеспечения

Согласно этой диаграмме головной модуль обращается к модулям управления устройством считывания кредитной карты, аутентификации и получения и обработки запроса на обслуживание. Вызов указанных модулей осуществляется согласно внутренней логики головного модуля, реализующей следующий сценарий: При инициации действий со стороны клиента головной модуль вызывает модуль управления устройством считывания кредитной карты для ее ввода и считывания с нее информации. После считывания управление возвращается Головному модулю, который затем обращается к модулю аутентификации, проверяющему подлинность клиента и вместе с результатом проверки возвращающему управление Головному модулю. В зависимости от результатов аутентификации Головной модуль либо вызывает модуль управления устройством считывания для удаления кредитной карты, либо обращается к модулю получения и обработки запроса на обслуживание для предоставления требуемого сервиса. Если осуществляется вызов получения и обработки запроса на обслуживание, то после завершения его работы Головной модуль обращается к модулю управления устройством считывания для удаления кредитной карты.

Обмен данными между программными модулями осуществляется через общую область памяти, в которую модуль управления устройством считывания помещает данные о пароле (Parol), атрибуты клиента (Client Attributes) и лимит денег на счету (Limit of money). Модуль аутентификации получает из этой общей области памяти сведения о пароле и возвращает в головной модуль управляющий параметр Autentification flag, содержащий результат аутентификации. Модуль получения и обработки запроса на обслуживание для своей работы получает из общей области памяти атрибуты клиента и лимит денег на счету.

2.3 ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Данное занятие предполагает выполнение следующих этапов:

- изучить методические указания;
- выполнить выданное задание;
- ответить на контрольные вопросы.

2.4 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое рабочее проектирование?
2. На основании результатов каких этапов разработки выполняется рабочее проектирование?
3. Для чего используются структурные карты?
4. Какие методики проектирования структурных карт существуют?
5. Чем отличаются структурные карты Константайна от структурных карт Джексона?

3 РАЗРАБОТКА ПЕРЕЧНЯ АРТЕФАКТОВ И ПРОТОКОЛОВ ПРОЕКТА

3.1 ЦЕЛЬ РАБОТЫ

Цель работы – изучение процесса разработки необходимого перечня артефактов и протоколов создаваемого проекта.

3.2 ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ

3.2.1 Основные понятия

Артефакт – это любой созданный искусственно элемент программной системы.

К элементам программной системы, а, следовательно, и к артефактам, могут относиться исполняемые файлы, исходные тексты, веб-страницы, справочные файлы, сопроводительные документы, файлы с данными, модели и многое другое, являющееся физическим носителем информации. Другими словами, артефактами являются те информационные элементы, которые тем или иным способом используются при работе программной системы и входят в ее состав.

С понятием «компонент» часто ассоциируют компонентное или сборочное программирование, однако это не верно с точки зрения UML (англ. Unified Modeling Language – унифицированный язык моделирования: язык графического описания для объектного моделирования в области разработки программного обеспечения, моделирования бизнес-процессов, системного проектирования и отображения организационных структур).

UML является языком широкого профиля, это – открытый стандарт, использующий графические обозначения для создания абстрактной модели системы, называемой UML-моделью. UML был создан для определения, визуализации, проектирования и документирования, в основном, программных систем. UML не является языком программирования, но на основании UML-моделей возможна генерация кода.

В терминах UML компонентное или сборочное программирование манипулирует артефактами!

Компонент (в UML) – это часть модели, описывающая логическую сущность, которая существует только во время проектирования (design time), хотя в дальнейшем ее можно связать с физической реализацией (артефактом) времени исполнения (run time).

Унифицированный язык моделирования UML – это стандартный инструмент для разработки «чертежей» программного обеспечения. Его можно использовать для визуализации, спецификации, конструирования и документирования артефактов программных систем. UML подходит для моделирования любых систем – от информационных систем масштаба предприятия до распределенных Web-приложений и даже встроенных систем реального времени.

На практических занятиях язык UML используется для спецификирования (построения точных, недвусмысленных и полных моделей) системы и ее документирования.

Язык UML предоставляет стандартный способ написания проектной документации на системы, включая концептуальные аспекты, такие как бизнес-процессы и функции системы, а также конкретные аспекты, такие как выражения языков программирования, схемы баз данных и повторно используемые компоненты программного обеспечения.

Rational Unified Process (RUP) – методология разработки программного обеспечения (ПО), созданная компанией Rational Software.

В основе RUP лежат следующие принципы:

– ранняя идентификация и непрерывное (до окончания проекта) устранение основных рисков;

– концентрация на выполнении требований заказчиков к исполняемой программе (анализ и построение модели прецедентов (вариантов использования));

– ожидание изменений в требованиях, проектных решениях и реализации в процессе разработки;

– компонентная архитектура, реализуемая и тестируемая на ранних стадиях проекта;

– постоянное обеспечение качества на всех этапах разработки проекта (продукта);

– работа над проектом в сплочённой команде, ключевая роль в которой принадлежит архитекторам.

Артефакты – это некоторые продукты проекта, порождаемые или используемые в нем при работе над окончательным продуктом.

Весь процесс разработки программной системы рассматривается в RUP как процесс создания артефактов. Причем то, что попадает в руки конечного пользователя, будь то программный модуль или программная документация, – это один из подклассов всех артефактов проекта.

Каждый член проектной группы создает свои артефакты и несет за них ответственность. Программист разрабатывает программу, руководитель – проектный план, а аналитик – модели системы. RUP позволяет определить, когда, кому и какой артефакт необходимо создать.

Система гиперссылок построена таким образом, что можно легко переходить от работ к артефактам, создаваемым в процессе конкретной деятельности, а через них к ролям исполнителей и обратно. К артефактам можно добраться различными путями – например, через список процессов, примеры итераций, роли исполнителей, а можно просто найти нужный артефакт в дереве ссылок, что позволяет рассматривать один и тот же процесс разработки с различных точек зрения – руководителя и исполнителя, пользователя и программиста.

В ходе анализа RUP определены следующие артефакты:

– Архитектура программного средства Software Architecture Document;
– Документ по инсталляции Instaliation Artifacts;
– Документ-концепция Vision;
– Журнал тестирования Test Log;
– Интерфейс Interface;
– Инфраструктура разработки Development Infrastructure;
– История изменения требований Storyboard;
– Компоненты развертывания Deployment Unit;
– Конфигурация тестовой средыTest Enviroment Configuration;
– Модель данных Data Model;
– Модель предметной области Business Case;
– Модель прецедентов использования Use-Case Model;

– Модель развертывания Deployment Model;
– Модель разработки Design Model;
– Модель реализации Implementation Model;
– Набор тестов Test Suide;
– Отчет о тестировании Test Evalution Summary;
– Отчет рецензента Review Record;
– Оценка итераций Iteration Assessment;
– План итераций Iteration Plan;
– План приема/сдачи продукта Product Acceptance Plan;
– План развертывания Deployment Plan;
– План создания программного средства Software Development Plan;
– План тестирования Test Plan;
– Порядок работ Work Order;
– Прецеденты Use-Case;
– Продукт Product;
– Проект подсистем Design Subsystem;
– Прототип пользовательского интерфейса User-Interface Prototype;
– Реализация подсистем Implementation Subsystem;
– Результат тестирования Test results;
– Руководство пользователя End-User Support Material;
– Сборка Build;
– Сборка прецедентов Use-Case Package;
– Словарь Glossary;
– Спецификация требований к ПО Software Requirement Specification;
– Список акторов Actor;
– Список потребностей заинтересованных лиц Stakeholder Request;
– Список тестов Test-Ideals List;
– Список требований Change Request;
– Список устраненных замечаний Release Notes;
– Стратегия тестирования Test Strategy;
– Требования к ПО Software Requirement.

Весь процесс разработки с точки зрения RUP рассматривается в двух плоскостях. В динамике процесс выражается через циклы, фазы, итерации и вехи, а в статике – через виды деятель-

ности, технологические процессы, артефакты и роли исполнителей.

3.2.2 Разработка перечня артефактов в реальном проекте

Эта модель взята из реального проекта. Она была создана по той причине, что из-за множества процессов из RUP и процессов в рассматриваемой организации возникала путаница в следующих вопросах:

- какие именно результаты нужно получить на выходе;
- кто отвечает за артефакт;
- какая группа должна нести за него ответственность,
- какие средства нужно использовать;
- каковы отношения между артефактами и элементами модели;
- используется ли данный RUP-артефакт в этом проекте;
- на какой итерации данный артефакт создается;
- каков уровень детализации;
- как следует связать артефакты существующего процесса с RUP-артефактами;
- и так далее.

Эта информация относится к проекту, в котором методология RUP применялась к большинству процессов, но не ко всем, поскольку были дополнительные процессы, а процесс «Анализ и проектирование» был приспособлен к конкретному проекту. Это не окончательная модель RUP. Её можно рассматривать лишь как идею модели, которая может быть использована для создания вашего проекта.

Введение.

На первой итерации менеджер процессов вместе с руководителем проекта и другими сотрудниками потратил немало времени, чтобы определить, какие артефакты должны быть созданы и на каких итерациях.

Обсуждался уровень детализации для каждого артефакта и членов группы. Это было необходимо для составления начального описания процесса разработки проекта, а также для того, чтобы руководитель проекта мог разработать план итераций проекта и оценить сроки выполнения.

Проблема.

Первая проблема заключалась в том, что пока шло планирование, остальные сотрудники не могли приступить к работе над артефактами, но эта проблема не связана с рассматриваемым здесь решением, она характерна лишь для некоторых проектов. Это требует отдельного обсуждения.

После того, как было тщательно распланировано создание артефактов и доведено для сведения руководителю проекта, артефакты были опубликованы в описании процесса разработки. Сразу выяснилось, что многие неверно понимают такие термины как «Use Case» (прецедент). Некоторые думали, что это маленькая овальная схема в Rational Rose (популярное средство визуального моделирования объектно-ориентированных информационных систем), другие полагали, что это документ Word, который содержит только текст, третьи – и то, и другое, а некоторые считали, что это модель всех прецедентов. То же самое было со многими артефактами, типами артефактов и элементами модели.

Были и другие проблемы, например, некоторые не понимали набор инструментов, где должен находиться тот или иной артефакт, где его следует искать и т. д.

Было также непонятно и то, кто должен отвечать за артефакты, за их создание, за соблюдение сроков и т. д. К кому обращаться, если что-то понадобится исправить или улучшить и т. п.

Существовала также серьёзная проблема с жаргонными названиями артефактов, которые придумывались несмотря на то, что артефакты в RUP уже имеют названия.

Решение.

Решение исходит из того факта, что поскольку программное обеспечение должно содержать модели и соответствующие концепции, то же самое должен содержать и процесс разработки.

На начальной стадии многие менеджеры с административным уклоном, например, руководитель проекта, менеджер по программированию, представители пользователей и т. д. не могли читать UML и даже не хотели разбираться в модели. Однако после того, как им объяснили некоторые самые простые понятия моделирования классов, они поняли значения терминов, разобрались в модели и смогли отвечать за свои части модели.

В результате легко стали находиться забытые артефакты, решаться конфликты по поводу того, что за один артефакт отвечают разные группы. В конечном итоге проектировщики смогли итеративно обновлять описание процесса разработки, публиковать отношения между артефактами и распределение ответственности – кто за какой артефакт отвечает, и всё это с использованием описанных ниже пакетов, моделей классов и диаграмм операций.

Использование.

– С помощью цвета разработчики указывали, какие артефакты из RUP обязательно будут использоваться, какие, возможно, будут использоваться, а какие определённо не будут использоваться. Зелёный = используется, оранжевый = возможно, красный = нет, белый = решает другая группа;

– Цвет пакета указывал также на его происхождение – RUP или собственный;

– Чтобы обозначить <<артефакт>>, <<группу>>, <<операцию>> и т. д., использовалась система обозначений классов со стереотипами;

– Поля атрибутов класса использовались для указания инструмента, в котором должен находиться артефакт, например, в ClearCase, Rose, Word, ReqPro, ClearQuest, Excel и т. п.

– Поле атрибута класса также использовалось для обозначения ответственного за артефакт (владельца артефакта), ответственного за его создание и т. д.;

– Ключ указывал на обозначение цветом, способ чтения UML и т. п.;

– У разработчиков был ключ для обозначения типа артефакта – RUP-артефакт, не RUP-артефакт или смешанный;

– Разработанная модель была опубликована с помощью RationalRose на веб-сайте организации в качестве составной части описания процесса разработки.

– Выводы.

Сотрудники организации сочли эту модель очень полезной, поэтому можно усовершенствовать RUP, предложив эту модель как альтернативную форму документирования описания процесса разработки. Если для большинства других процессов разрабатываются свои модели, то можно и для управления проектом разработать отдельную модель.

Очевидно, при наличии стандартной модели RUP, конкретный проект можно гораздо быстрее подогнать под стандарт RUP. Для малых проектов модель, вероятно, вообще не нужна. Решение этого вопроса также зависит от того, насколько хорошо проектная группа знакома с RUP.

3.3 ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Данное занятие предполагает выполнение следующих этапов:

- изучить методические указания;
- выполнить выданное задание;
- ответить на контрольные вопросы.

3.4 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что понимается под артефактом при разработке проекта?
2. Какие элементы программной системы могут быть отнесены к артефактам?
3. Что представляет собой язык UML и для чего он используется?
4. Перечислите основные принципы методологии разработки программного обеспечения RUP.
5. На конкретном примере охарактеризуйте основные этапы разработки перечня артефактов при разработке проекта.

4 НАСТРОЙКА РАБОТЫ СИСТЕМЫ КОНТРОЛЯ ВЕРСИЙ (ТИПОВ ИМПОРТИРУЕМЫХ ФАЙЛОВ, ПУТЕЙ, ФИЛЬТРОВ И ДР. ПАРАМЕТРОВ ИМПОРТА В РЕПОЗИТОРИЙ)

4.1 ЦЕЛЬ РАБОТЫ

Цель работы – получение первоначальных навыков использования систем контроля версий исходного кода программ, получение первоначальных навыков организации коллективной разработки программного обеспечения. Создание в системе контроля версий репозитория для нового проекта и выполнение всех основных действий с исходным кодом программы, связанных с контролем версий.

4.2 ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ

Управление версиями – это искусство работы с изменениями информации. Долгое время оно было жизненно важным инструментом программистов, которым необходимо произвести небольшие изменения в программе, или же сделать откат изменений, возвращаясь к предыдущей версии. Однако полезность систем управления версиями выходит далеко за пределы мира разработчиков программного обеспечения. Управление версиями требуется повсюду, где можно встретить людей, использующих компьютер для работы с постоянно изменяющейся информацией.

Software Configuration Management или Конфигурационное управление подразумевает под собой комплекс методов, направленных на то, чтобы систематизировать изменения, вносимые разработчиками в программный продукт в процессе его разработки и сопровождения, сохранить целостность системы после изменений, предотвратить нежелательные и непредсказуемые эффекты, а также сделать процесс внесения изменений более формальным. Изначально управление конфигурацией применялось не в программировании, но в связи с высокой динамичностью сферы разработки ПО, в ней она особенно полезна. К процедурам можно отнести создание резервных копий, контроль исходного кода, требований проекта, документации и т. д. Степень формальности

выполнения данных процедур зависит от размеров проекта, и при правильной ее оценке данная концепция может быть очень полезна. Конфигурационное управление требует выполнения множества трудоемких рутинных операций. На практике, в большинстве случаев, для конфигурационного управления применяются специальные системы контроля версий исходного кода программ. В качестве примера такой системы рассмотрим самую распространенную на сегодняшний день – Subversion.

Subversion – это система управления версиями с открытым исходным кодом. Subversion позволяет управлять файлами и каталогами, а так же сделанными в них изменениями во времени. Это позволяет восстановить более ранние версии данных, даёт возможность изучить историю всех изменений. Благодаря этому многие считают систему управления версиями своего рода «машиной времени».

Subversion может работать через сеть, что позволяет использовать её на разных компьютерах. В какой то степени, возможность большого количества людей независимо от их местоположения совместно работать над единым комплектом данных поощряет сотрудничество. Когда нет того ответственного звена цепи, того контролирующего элемента, который утверждает все изменения, работа становится более эффективной. При этом не нужно опасаться, что отказ от контролирующего элемента повлияет на качество, ведь благодаря сохранению истории изменений, даже если при изменении данных будут допущены ошибки, всегда можно сделать откат изменений к прежнему состоянию.

Некоторые системы управления версиями выступают также в качестве систем управления конфигурацией программного обеспечения. Такие системы специально созданы для управления деревьями исходного кода и имеют множество особенностей, непосредственно относящихся к разработке программ: они понимают языки программирования и предоставляют инструменты для сборки программ. Subversion не является такой системой, она представляет собой систему общего назначения, которую можно использовать для управления любым набором файлов. Для обучающихся это будут исходники разрабатываемых программ, а для кого-то другого это может быть, например, список продуктов или сведённое цифровое видео.

TortoiseSVN.

TortoiseSVN – это бесплатный Windows-клиент с открытыми исходным кодом для системы управления версиями Apache™ Subversion®. То есть TortoiseSVN управляет файлами и директориями во времени. Файлы хранятся в центральном хранилище. Хранилище больше похоже на обычный файловый сервер, кроме того он запоминает каждое изменение когда-либо сделанное в ваших файлах и директориях. Это позволяет вам восстановить старые версии ваших файлов и проверить историю изменений – как, когда и кто изменял ваши данные. Вот почему многие думают о Subversion, и вообще о системах управления версиями, как о своего рода «машине времени».

Возможности TortoiseSVN.

Что делает TortoiseSVN хорошим клиентом Subversion? Вот краткий список возможностей:

Интеграция с оболочкой.

TortoiseSVN интегрируется непосредственно в оболочку Windows (т. е. в Проводник). Это значит, что вы можете работать с уже знакомыми инструментами, и вам не надо переключаться на другое приложение каждый раз, когда вам необходимы функции для управления версиями!

И вам даже не обязательно использовать именно Проводник. Контекстные меню TortoiseSVN работают во многих других файловых менеджерах, и в диалогах для открытия файлов, используемых в большинстве стандартных Windows-приложений. Однако вы должны учитывать, что TortoiseSVN изначально разработан как расширение для Проводника Windows, и, возможно, в других приложениях интеграция будет неполной, например, могут не отображаться пометки на значках.

Пометки на значках.

Статус каждого версированного файла и папки отображается при помощи маленькой пометки поверх основного значка. Таким образом, вы сразу можете видеть состояние вашей рабочей копии.

Графический интерфейс пользователя.

При просмотре списка изменений файла или папки вы можете кликнуть на ревизию, чтобы увидеть комментарии для этой фиксации. Также доступен список измененных файлов – всего

лишь сделайте двойной клик на файле, чтобы увидеть, какие конкретно изменения были внесены.

Диалог фиксации – это список, в котором перечислены все файлы и папки, которые будут включены в фиксацию. У каждого элемента списка имеется флажок, чтобы вы могли выбрать именно то, что вы хотите включить в фиксацию. Неверсированные файлы также могут быть представлены в этом списке, чтобы вы не забыли добавить в фиксацию новый файл или папку.

Простой доступ к командам Subversion.

Все команды Subversion доступны из контекстного меню Проводника. TortoiseSVN добавляет туда собственное подменю.

Поскольку TortoiseSVN является клиентом Subversion, мы хотели бы показать и некоторые из возможностей самой Subversion:

– Версирование папок.

CVS отслеживает только историю отдельных файлов, тогда как Subversion реализует «виртуальную» версионную файловую систему, которая отслеживает изменения в целых деревьях папок во времени. Файлы и папки являются версированными. В результате, есть команды переместить и копировать, реально выполняемые на стороне клиента и работающие непосредственно с файлами и папками.

– Атомарные фиксации.

Фиксация сохраняется в хранилище либо полностью, либо не сохраняется вообще. Это позволяет разработчикам фиксировать изменения, собранные в логически связанные части.

– Версированные метаданные.

Каждый файл и папка имеет прикреплённый невидимый набор «свойств». Вы можете создавать и сохранять произвольные пары ключ/значение для собственных нужд. Свойства тоже версируются во времени, как и содержимое файла.

– Возможность выбора сетевого уровня.

В Subversion есть абстрагируемое понятие доступа к хранилищу, которое упрощает реализацию новых сетевых механизмов. «Усовершенствованный» сетевой сервер Subversion является модулем для веб-сервера Apache, который использует для взаимодействия диалект HTTP под названием WebDAV/DeltaV. Это даёт Subversion большие преимущества в стабильности и совместимо-

сти, и предоставляет различные ключевые возможности без дополнительных затрат: проверка личности (аутентификация), проверка прав доступа (авторизация), сжатие потока данных при передаче, просмотр хранилища. Также доступна меньшая, автономная версия сервера Subversion, взаимодействующая по собственному протоколу, который легко может быть туннелирован через ssh.

– Единый способ обработки данных.

Subversion получает различия между файлами при помощи бинарного разностного алгоритма, который работает одинаково как с текстовыми (читаемыми человеком), так и с бинарными (не читаемыми человеком) файлами. Оба типа файлов содержатся в хранилище в сжатом виде, а различия передаются по сети в обоих направлениях.

– Эффективные ветки и метки.

Стоимость создания веток и меток не обязательно должна быть пропорциональна размеру проекта. Subversion создаёт ветки и метки, просто копируя проект с использованием механизма, похожего на жёсткие ссылки в файловых системах. Благодаря этому, операции по созданию веток и меток происходят за одинаковое, очень малое время и занимают очень мало места в хранилище.

Установка.

TortoiseSVN поставляется пользователям в виде простого в использовании установочного файла (на странице загрузки http://tortoisesvn.net/downloads.html необходимо выбрать дистрибутив, соответствующий разрядности системы). Сделайте на нем двойной клик и следуйте инструкциям – остальное он сделает за вас. Также на странице загрузки необходимо скачать языковой пакет и установить после установки основной программы. Не забудьте перезагрузить компьютер после установки.

Основная концепция.

Перед тем, как мы погрузимся в работу с настоящими файлами, важно получить представление о том, как работает Subversion и какие термины используются.

Хранилище.

Subversion использует центральную базу данных, которая содержит все ваши версированные файлы с их полной историей. Эта база данных называется хранилищем. Хранилище обычно

находится на файловом сервере, на котором установлен Subversion, по запросу поставляющий данные клиентам Subversion (например, TortoiseSVN). Если вы делаете резервное копирование, то копируйте ваше хранилище, так как это оригинал всех ваших данных.

Рабочая копия.

Это именно то место, где вы работаете. Каждый разработчик имеет собственную рабочую копию, иногда называемую «песочницей», на своем локальном компьютере. Вы можете получить из хранилища последнюю версию файлов, поработать над ней локально, никак не взаимодействуя с кем-либо еще, а когда вы будете уверены в изменениях вы можете зафиксировать эти файлы обратно в хранилище.

Рабочая копия не содержит историю проекта, но содержит копию всех файлов, которые были в хранилище до того, как вы начали делать изменения. Это обозначает, что можно легко узнать, какие конкретно изменения вы сделали.

Вам также нужно знать, где найти TortoiseSVN, потому что в меню «Пуск» его нет. Это потому, что TortoiseSVN – расширение проводника Windows, поэтому для начала нужно запустить проводник. Сделайте правый клик на папке в проводнике и вы увидите новые пункты в контекстном меню.

Создание хранилища.

Для настоящего проекта вам понадобится хранилище, созданное в безопасном месте, и сервер Subversion, чтобы управлять им. Для обучения мы будем использовать функцию локального хранилища Subversion, которая разрешает прямой доступ к хранилищу, созданного на вашем жестком диске, и не требует наличия сервера.

Сначала необходимо создать новую пустую директорию на вашем компьютере. Она может быть где угодно, но в данном примере мы собираемся назвать её C:\svn_repos. Теперь сделайте правый клик на новой папке и в контекстном меню выберите TortoiseSVN → Создать здесь хранилище.

Хранилище, созданное внутри папки, готово к использованию. Также мы создадим внутреннюю структуру папок, нажав кнопку Создать структуру каталогов.

Импорт проекта.

Сейчас у нас есть хранилище, но оно совершенно пустое в данный момент. Давайте предположим, что у нас есть набор файлов в C:\Projects\Widget1, который мы хотели бы добавить. Перейдите к папке Widget1 в Проводнике и сделайте правый клик на ней. Теперь выберите пункт TortoiseSVN → Импорт..., который вызовет диалог для импорта файлов.

К хранилищу Subversion обращаются по URL-адресу, который позволяет нам указать хранилище где угодно в Интернете. В данном случае нам нужно указать на наше локальное хранилище, которое имеет URL-адрес file:///c:/svn_repos/trunk, и к которому мы добавляем имя нашего проекта Widget1. Обратите внимание, что после file: есть 3 слэша и везде используются прямые слэши.

Другая важная функция данного диалога – это окно Сообщение импорта, в которое вы можете добавить сообщение о том, что вы делаете. Когда вам понадобится просмотреть историю проекта, эти сообщения будут ценным подспорьем для просмотра какие изменения и когда были сделаны.

В нашем случае мы напишем что-нибудь простое, например, так: «Импорт проекта Виджет1». Нажмите OK, чтобы добавить папку в ваше хранилище.

Извлечение рабочей копии.

Сейчас у нас есть проект в нашем хранилище, и нам надо создать рабочую копию для повседневной работы. Заметьте, что импортирование папки не превращает автоматически эту папку в рабочую копию. Для создания свежей рабочей копии в Subversion используется термин Извлечь. Мы собираемся извлечь папку Widget1 из нашего хранилища в папку для разработки называемуюC:\Projects\Widget1-Dev. Создайте эту папку, затем сделайте правый клик на ней и выберите пункт TortoiseSVN → Извлечь.... Введите URL-адрес для извлечения, в данном случае file:///c:/svn_repos/trunk/Widget1, и кликните на OK. Наша папка для разработки заполнится файлами из хранилища.

Вы заметите, что внешний вид этой папки отличается от обычной папки. У каждого файла появился зелёный флажок в левом углу. Это значки статуса TortoiseSVN, которые присутствуют только в рабочей копии. Зелёный статус означает, что файл не отличается от версии файла, находящегося в хранилище.

Внесение изменений.

Можно приступать к работе. В папке Виджет1-Дев мы начинаем изменять файлы – предположим, мы вносим изменения в файлы Виджет1.c и ПрочтиМеня.txt.

Обратите внимание, что значки на этих файлах теперь стали красными и показывают, что изменения были сделаны локально.

Но какие были изменения? Нажмите правой кнопкой на одном из изменённых файлов и выберите команду TortoiseSVN →Различия. Запустится инструмент TortoiseSVN для сравнения файлов и он покажет, какие точно строки в файлах были изменены.

Нас устраивают изменения, поэтому давайте обновим хранилище. Это действие называется Фиксировать изменения. Нажмите правой кнопкой на папке Виджет1-Дев и выберите команду TortoiseSVN → Фиксировать.

Появится диалог фиксации со списком изменённых файлов и напротив каждого будет галочка. Вы можете выбрать лишь несколько файлов из списка для фиксации, но в нашем случае мы будем фиксировать изменения в обоих файлах. Введите сообщение с описанием сделанных изменений и нажмите OK. Появится диалог с прогрессом процесса фиксации файлов в хранилище и мы закончили фиксацию.

Добавление новых файлов.

Во время работы над проектом, вам понадобится добавлять новые файлы – предположим вы добавили новые функции в файле Экстра.c и добавили справку в существующем файле Создать-файл. Нажмите правой кнопкой на папке и выберите команду TortoiseSVN → Добавить.

Диалог добавления показывает все неверсированные файлы и вы можете выбрать те файлы, которые вы хотите добавить. Другой способ добавления файлов – это нажать правой кнопкой на самом файле и выбрать команду TortoiseSVN → Добавить.

Теперь, если вы откроете папку для фиксации, новый файл будет отображаться как Добавлен и существующий файл как Изменён. Обратите внимание, что вы можете дважды нажать на изменённый файл, чтобы просмотреть какие именно изменения были сделаны.

Просмотр истории проекта.

Одной из самых полезных функций TortoiseSVN является диалоговое окно журнала. Оно показывает список всех фиксаций изменений в файле или папке, а также все те детальные сообщения, которые вы вводили при фиксации изменений.

Верхняя панель показывает список всех фиксированных ревизий вместе с началом сообщения фиксации. Если вы выберите одну из этих ревизий, то средняя панель отобразит полное сообщение журнала для той ревизии и нижняя панель покажет список измененных файлов и папок.

У каждой из этих панелей есть контекстное меню, которое предоставляет много других способов использования информации. В нижней панели вы можете дважды нажать на файл, чтобы просмотреть какие именно изменения были внесены в той ревизии. Прочтите «Диалоговое окно журнала ревизий», чтобы узнать больше.

Отмена изменений.

Одной общей функцией всех систем управления ревизиями является функция, которая позволяет вам отменить изменения, которые вы внесли ранее. Как вы и догадались, в TortoiseSVN это легко сделать.

Если вы хотите избавиться от изменений, которые вы еще не успели фиксировать и восстановить нужный файл в том виде, в котором он был перед началом изменений, то выберите команду TortoiseSVN → Убрать изменения. Это действие отменит ваши изменения (в Корзину) и вернет фиксированную версию файла, с которой вы начинали. Если же вы хотите убрать лишь некоторые изменения, то вы можете использовать инструмент TortoiseMerge для просмотра изменений и выборочного удаления измененных строк.

Если вы хотите отменить действия определённой ревизии, то начните с диалогового окна журнала и найдите проблемную ревизию. Выберите команду Контекстное меню → Отменить изменения из этой ревизии и те изменения будут отменены.

Работа с сетью.

С Subversion можно работать как посредством сети интернет, так и локально. Воспользуемся сервисом Assembla

(https://www.assembla.com/). Зарегистрировавшись там, вы получите 1 Gb места под репозиторий.

Создав его и настроив, вы получите ссылку вида https://subversion.assembla.com/svn/название_репозитория, которую можно использовать в любом SVN клиенте. К примеру, чтобы в Visual SVN добавить свой проект в репозиторий (Репозиторий, хранилище – место, где хранятся и поддерживаются какие-либо данные. Чаще всего данные в репозитории хранятся в виде файлов, доступных для дальнейшего распространения по сети), вам нужно нажать Add Solution to Subversion, после чего указать локальное хранилище вашего проекта, нажать Далее и ввести вашу ссылку.

Все, теперь сверху появится панель с основными SVN-функциями (Show Log, Update, Commit, Switch Branch, Branch и Merge) и можно приступать к полноценной работе.

В общем и целом сайт позволяет так же настроить репозиторий и получить на неделю или две пробный премиум (платные доп. функции, источник жизни сайта). Основные функции же полностью бесплатны. Чтобы товарищи по команде могли работать с общим для команды репозиторием, на assembla.com, необходима их регистрация на сайте. После регистрации, пользователи могут быть добавлены владельцу репозитория в список команды. На самом сайте можно посмотреть всю информацию о проекте и изменениях, и даже поставить свой баннер с ссылкой.

4.3 ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Данное занятие предполагает выполнение следующих этапов:

– изучить методические указания;

– выполнить задание:

1. Установить TortoiseSVN на компьютере.
2. Создать новый проект.
3. Создать локальный репозиторий для своего проекта.
4. Удалить созданный проект на своем компьютере и обновить проект из репозитория.

5. Внести изменения в файлах с исходными кодами и сохранить изменения в репозитории. Обновить файлы с исходными кодами из репозитория.

6. Внести изменения в файлах с исходными кодами таким образом, чтобы у двух участников проекта изменения были в одном и том же файле. Сохранить изменения в репозитории. Устранить обнаруженные конфликты версий. Повторно сохранить изменения в репозитории.

7. Создать отдельную ветку проекта. Внести изменения в файлы с исходными кодами. Сохранить изменения в репозитории.

8. Объединить созданную на предыдущем шаге ветку с основной веткой проекта.

9. Вывести на экран лог изменений файла, в котором было наибольшее количество изменений.

10. Отобразить на экране сравнение файла до и после внесения одного из изменений.

11. Создать репозиторий в сети Интернет. Повторить шаги 4–6.

– ответить на контрольные вопросы.

4.4 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что понимается под управлением версиями?
2. Что подразумевает под собой конфигурационное управление?
3. Дайте характеристику системе управления версиями Subversion.
4. Что представляет собой TortoiseSVN? Перечислите возможности этого программного средства.
5. Дайте представление о том, как работает Subversion и какие термины используются.
6. Что понимается под репозиторием?

5 РАЗРАБОТКА И ИНТЕГРАЦИЯ МОДУЛЕЙ ПРОЕКТА (КОМАНДНАЯ РАБОТА)

5.1 ЦЕЛЬ РАБОТЫ

Цель работы – изучить принципы разработки и интеграции модулей проекта и принципы формирования и управления командной работы.

5.2 ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ

Для упрощения управления проектом, организации и координации проектных работ все действия, направленные на достижение целей проекта, разбивают на отдельные составляющие - процессы управления проектом. Управление проектом по стандарту PMBoK (Project Management Body of Knowledge – Управление проектами) выполняется с помощью 44 процессов, которые объединены в пять групп, называемых «группы процессов управления проектом» http://www.intuit.ru/studies/courses/2196/267/literature - literature.9:

1. Группа процессов инициации.
2. Группа процессов планирования.
3. Группа процессов исполнения.
4. Группа процессов мониторинга и управления.
5. Группа завершающих процессов.

Все пять групп процессов имеют четкие зависимости, они выполняются в одной и той же последовательности в каждом проекте с определенным наложением. Степень наложения определяется условиями выполнения конкретного проекта.

Процессы, входящие в группу процессов, могут иметь взаимосвязи как в рамках данной группы процессов, так и с процессами других групп.

Для успешного достижения целей проекта необходимо не только управлять каждым процессом в отдельности, но и обеспечить комплексный подход к управлению с учетом взаимосвязей,

взаимозависимостей как отдельных процессов, так и групп процессов.

С целью структуризации управления проектом процессы управления проектом распределены по девяти областям знаний:

1. Управление интеграцией.
2. Управление содержанием.
3. Управление временем.
4. Управление стоимостью.
5. Управление персоналом.
6. Управление коммуникациями.
7. Управление качеством.
8. Управление рисками.
9. Управление снабжением.

Управление интеграцией.

Область знаний «Управление интеграцией» включает все пять групп процессов: Инициация, Планирование, Исполнение, Управление и контроль, Завершение.

Результаты процессов из группы Инициация являются входящей информацией для группы процессов Планирование. В свою очередь, результаты групп процессов Управление и контроль являются входящими для группы процессов Завершение.

Прежде чем перейти к рассмотрению процессов управления из области интеграции, определим, что же понимается под интеграцией процессов.

Понятие интеграции процессов управления.

Интеграция процессов управления проектом – это взаимосвязи групп процессов и входящих в них процессов, обеспечивающие непрерывный и комплексный подход к управлению проектной деятельностью.

Цель интеграции состоит в достижении эффективного взаимодействия процессов управления проектами, обеспечивающих достижение целей проекта.

Интеграция управления проектом требует, чтобы все процессы управления проектами были выстроены и связаны с другими процессами для облегчения их координации.

Необходимость в интеграции процессов управления проектами обусловлена взаимодействием процессов управления. Эти процессы взаимодействуют между собой сложным образом, по-

этому рассмотрим на отдельных примерах, как выстраивается интеграционное взаимодействие групп процессов управления проектной деятельностью.

Проектная деятельность начинается с процессов инициации – с момента подписания договора с Заказчиком (или согласования с Заказчиком условий договора). При инициации определяются цели, задачи, результаты, сроки проекта, формируется команда управления проектом, определяются необходимые ресурсы, подготавливаются при необходимости рабочие места, разрабатываются необходимые для управления проектом документы. На этом инициация проекта завершается. Команда управления проектом приступает к процессу планирования проекта, составляется расписание проекта. Как правило, вначале разрабатывается укрупненное расписание, которое должно соответствовать этапам договора, затем осуществляется его детализация. С точки зрения управления интеграцией, договор является точкой входа для процесса планирования. Именно договором определяются результат и сроки проекта. По завершению составления расписания проекта – когда определены задачи, их исполнители, сроки выполнения, – приступают к выполнению проектных работ. Процесс планирования при этом не заканчивается, он продолжается практически до момента завершения проекта. В ходе выполнения работ первоначальное укрупненное расписание проекта детализируется, уточняется. А это, в свою очередь, означает необходимость построения интеграционного взаимодействия процессов планирования с процессами исполнения работ.

Процессы группы «исполнение» выстраиваются в соответствии с применяемой на проекте методологией внедрения информационной системы.

С момента инициации проекта осуществляется непрерывный контроль над всей проектной деятельностью, включая и процессы планирования, и процессы исполнения работ, и процессы завершения, то есть процессы контроля интегрируются со всеми группами процессов управления проектами. Результатом процессов контроля могут быть решения, управляющие воздействия на планирование, изменение хода проектных работ, процедуры закрытия проекта.

Процессы завершения формализуют приемку разработанной информационной системы (ИС). При успешном завершении приемки ИС осуществляется закрытие проекта (включая финансовое и организационное закрытие проекта).

Не все процессы могут понадобиться в каждом конкретном выполняемом проекте или его фазе, и не все взаимодействия могут быть к ним применимы.

При организации работы над проектом необходимо решить две главные задачи:

- формирование команды проекта;
- организация эффективной работы команды.

В зависимости от специфики, размера и типа проекта в его реализации могут принимать участие от одной до нескольких десятков (иногда сотен) организаций и отдельных специалистов. У каждой из них свои функции, степень участия в проекте и мера ответственности за его реализацию. Специалистов и организации, в зависимости от выполняемых ими функций, принято объединять в совершенно конкретные группы (категории) участников проекта, в состав которых входят: заказчики, инвесторы, проектировщики, поставщики ресурсов, подрядчики, консультанты, лицензиары, финансовые институты – банки и, наконец, команда проекта, возглавляемая руководителем проекта – менеджером проекта (в терминологии, принятой на Западе – проект-менеджера), а также, в зависимости от специфики проекта, другие участники.

Следует отметить, что участники проекта – категория более широкая, чем команда проекта.

Команда проекта – одно из главных понятий управления проектами. Это группа сотрудников, непосредственно работающих над осуществлением проекта и подчиненных руководителю последнего; основной элемент его структуры, так как именно команда проекта обеспечивает реализацию его замысла. Эта группа создается на период реализации проекта и после его завершения распускается.

Очевидно, что количество людей в команде определяется объемом работ, предусмотренным проектом. Как правило, лидеры (менеджеры) функционально и(или) предметно ориентированных групп специалистов и составляют команду управления

проектом. Лидеры групп – это руководители, координаторы усилий всех членов группы; члены группы – непосредственные исполнители, которые имеют возможность сосредоточиваться на конкретной работе. При необходимости некоторые роли членов команды могут совмещаться.

Взаимоотношения участников проекта внутри команды проекта, создаваемой для управления последним, раскрывает ее организационная структура команды проекта. Существует два основных принципа формирования команды для управления проектом.

1. Ведущие участники проекта – заказчик и подрядчик (кроме них, могут быть и другие участники) создают собственные группы, которые возглавляют руководители проекта, соответственно, от заказчика и подрядчика. Эти руководители подчиняются единому руководителю проекта. В зависимости от организационной формы реализации проекта, руководитель от заказчика или от подрядчика может являться руководителем всего проекта. Руководитель проекта во всех случаях имеет собственный аппарат сотрудников, осуществляющих координацию деятельности всех участников проекта.

2. Для управления проектом создается единая команда во главе с руководителем проекта. В команду входят полномочные представители всех участников проекта для осуществления функций согласно принятому распределению зон ответственности.

Система управления командой проекта включает организационное планирование, кадровое обеспечение проекта, создание команды проекта, а также осуществляет функции контроля и мотивации трудовых ресурсов проекта для эффективного хода работ и завершения проекта. Система нацелена на руководство и координацию деятельности команды проекта, использует стили руководства, методы мотивации, административные методы, повышение квалификации кадров на всех фазах жизненного цикла проекта.

Сложность и комплексность задач по управлению проектом рождает потребность в высокой технической компетентности, владении большими объемами экономических, правовых, управленческих знаний, поэтому создание профессиональной проектной команды – необходимое условие эффективной работы над проектом.

Суть команды – в общем для всех ее членов обязательстве, определяемом наличием некоего назначения, в которое верят все члены команды: ее миссии, которая для проекта заключается в его эффективной его реализации.

Для команды проекта необходимо наличие у ее членов комбинации взаимодополняющих навыков, которые составляют три категории:

- технические и/или функциональные, то есть профессиональные, навыки;
- навыки по решению проблем и принятию решений;
- навыки межличностного общения (принятие риска, полезная критика, активное слушание и т. д.).

Она обладает такими существенными признаками, как:

- внутренняя организация, состоящая из органов управления, контроля и санкций;
- групповые ценности, на основе которых формируется чувство общности в команде и создается общественное мнение;
- собственный принцип обособления, отличающий ее от других команд;
- групповое давление, т. е. воздействие на поведение членов команды общими целями и задачами деятельности;
- стремление к устойчивости благодаря механизму отношений, возникающих между людьми в ходе решения общих задач;
- закрепление определенных традиций.

Команда – это самостоятельный субъект деятельности, который может быть рассмотрен с точки зрения свойств, процессов, параметров, характерных для социальной группы.

Microsoft Visual Studio – линейка продуктов компании Майкрософт, включающих интегрированную среду разработки программного обеспечения (IDE-среда) и ряд других инструментальных средств. Данные продукты позволяют разрабатывать как консольные приложения, так и приложения с графическим интерфейсом, в том числе с поддержкой технологии Windows Forms, а также веб-сайты, веб-приложения, веб-службы как в родном, так и в управляемом кодах для всех платформ, поддерживаемых Microsoft Windows, Windows Mobile, Windows CE, .NET Framework, Xbox, Windows Phone .NET Compact Framework и Microsoft Silverlight.

Для выполнения практической работы необходимо установить комплекс программ Visual Studio на вашем компьютере.

Командная строка разработчика для Visual Studio облегчает использование средств .NET Framework. Также эта командная строка автоматически задает определенные переменные среды.

Поиск командной строки на компьютере.

В зависимости от версии Visual Studio и дополнительно установленных пакетов SDK может иметься несколько вариантов командной строки.

SDK (от англ. Software Development Kit) – набор средств разработки, который позволяет специалистам по программному обеспечению создавать приложения для определённого пакета программ, программного обеспечения базовых средств разработки, аппаратной платформы, компьютерной системы, игровых консолей, операционных систем и прочих платформ.

Программист, как правило, получает SDK непосредственно от разработчика целевой технологии или системы. Часто SDK распространяется через Интернет. Многие SDK распространяются бесплатно для того, чтобы побудить разработчиков использовать данную технологию или платформу.

В 64-разрядных версиях Visual Studio представлены 32-разрядная и 64-разрядная версии командной строки.(32-разрядная и 64-разрядная версии большинства инструментов являются одинаковыми. Однако некоторые инструменты работают по-разному в 32-разрядных и 64-разрядных средах.) Если приведённые ниже действия не работают, можно попробовать найти файлы на компьютере вручную или запустить командную строку из Visual Studio.

В Windows 10:

1. В поле поиска на панели задач начните вводить имя средства, например `dev` или `developer command prompt` .Откроется список установленных приложений, которые соответствуют вашему шаблону поиска. Если вы ищете другую командную строку, попробуйте ввести другое условие поиска, например, `prompt`.

2. Выберите пункт Командная строка разработчика для Visual Studio (или нужную вам командную строку).

Windows 8.1.

1. Перейдите на начальный экран, например нажав клавишу с логотипом Windows на клавиатуре.

2. На начальном экране нажмите CTRL+TAB, чтобы открыть список приложений, а затем введите V. Появится список, включающий все установленные командные строки Visual Studio.

3. Выберите элемент Командная строка разработчика(или нужную командную строку).

Windows 8.

1. Перейдите на начальный экран, например нажав клавишу с логотипом Windows на клавиатуре.

2. На начальном экране нажмите на клавиатуре клавишу с логотипом Windows+ Z.

3. Выберите значок представления «Приложения»в нижней части экрана, а затем введите V. Появится список, включающий все установленные командные строки Visual Studio.

4. Выберите элемент Командная строка разработчика(или нужную командную строку).

Windows 7.

1. В меню Пуск разверните список Все программы, а затем папку Microsoft Visual Studio.

2. В зависимости от установленной версии Visual Studio выберите пункт Инструменты Visual Studio, Командная строка Visual Studio или нужную командную строку.

Если установлен пакет SDK, например Windows 10 SDK или предыдущие версии, в системе могут быть дополнительные командные строки для архитектур ARM, x86 или x64.Требуемая версия командной строки указана в документации по соответствующим инструментам.

Поиск файлов на компьютере вручную.

Обычно ярлыки для установленных командных строк помещаются в папку меню «Пуск» для Microsoft Visual Studio, например, в такой путь C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Visual Studio 2017\Visual Studio Tools.Но если по какой-то причине поиск командной строки не дает ожидаемых

результатов, попробуйте вручную найти ярлык на компьютере. Попробуйте ввести имя файла командной строки, например VsDevCmd.bat, или перейдите в папку средств, например в C:\Program Files (x86)\ Microsoft Visual Studio\2017\Enterprise\Common7\Tools (точный путь зависит от версии, выпуска и расположения установки Visual Studio).

Запуск командной строки из Visual Studio.

Для упрощения доступа можно включить командную строку разработчика для Visual Studio или другую командную строку в меню Инструменты в Visual Studio. Чтобы сделать средство доступным, добавьте его в список внешних инструментов. Ниже приведены инструкции.

1. Запустите Visual Studio.

2. Выберите меню Инструменты и затем выберите в нем пункт Внешние инструменты.

3. В диалоговом окне Внешние инструменты нажмите кнопку Добавить. Появится новый элемент.

4. Введите заголовок для нового пункта меню, например Command Prompt.

5. В поле Команда укажите файл, который должен запускаться, напримерC:\Windows\System32\cmd.exe.

6. В поле Аргументы укажите, где найти конкретную командную строку, которую вы хотите использовать, например:

/k «C:\Program Files (x86)\ Microsoft Visual Studio\2017\Enterprise\Common7\Tools\VsDevCmd.bat»(эта команда запускает командную строку разработчика, установленную с Visual Studio 2017 Enterprise).Измените это значение в соответствии с вашей версией, выпуском и расположением установки Visual Studio.

7. Выберите значение для поля Исходный каталог, например Каталог проекта.

8. Нажмите кнопку ОК.

Новый элемент добавляется в меню, и вы можете вызывать командную строку из меню Инструменты.

5.3 ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Данное занятие предполагает выполнение следующих этапов:

– изучить методические указания;
– выполнить выданное задание;
– ответить на контрольные вопросы.

5.4 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Как происходит управление проектом по стандарту PMBoK?
2. Как с целью структуризации управления проектом распределяются процессы управления проектом?
3. Понятие интеграции процессов управления.
4. Какие главные задачи решаются при организации работы над проектом?
5. Что понимается под командой проекта, как и на какой период она формируется?
6. Дайте характеристику двум основным принципам формирования команды для управления проектом.

6 ОТЛАДКА ОТДЕЛЬНЫХ МОДУЛЕЙ ПРОГРАММНОГО ПРОЕКТА

6.1 ЦЕЛЬ РАБОТЫ

Цель работы – изучить основные принципы и процесс отладки отдельных модулей программного проекта.

6.2 ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ

6.2.1 Основные понятия

Отладка программного средства (ПС) – это деятельность, направленная на обнаружение и исправление ошибок в ПС с использованием процессов выполнения его программ.

Тестирование ПС – это процесс выполнения его программ на некотором наборе данных, для которого заранее известен результат применения или известны правила поведения этих программ. Указанный набор данных называется тестовым или просто тестом. Таким образом, отладку можно представить в виде многократного повторения трех процессов: тестирования, в результате которого может быть констатировано наличие в ПС ошибки, поиска места ошибки в программах и документации ПС и редактирования программ и документации с целью устранения обнаруженной ошибки. Другими словами:

Отладка = Тестирование + Поиск ошибок + Редактирование.

В зарубежной литературе отладку часто понимают только как процесс поиска и исправления ошибок (без тестирования), факт наличия которых устанавливается при тестировании. Иногда тестирование и отладку считают синонимами. В нашей стране в понятие отладки обычно включают и тестирование, поэтому мы будем следовать сложившейся традиции. Следует, однако, отметить, что тестирование используется и как часть процесса аттестации ПС.

6.2.2 Принципы и виды отладки программного средства

Успех отладки ПС в значительной степени предопределяет рациональная организация тестирования. При отладке ПС отыскиваются и устраняются, в основном, те ошибки, наличие которых в ПС устанавливается при тестировании. Тестирование не может доказать правильность ПС, в лучшем случае оно может продемонстрировать наличие в нем ошибки. Другими словами, нельзя гарантировать, что тестированием ПС практически выполнимым набором тестов можно установить наличие каждой имеющейся в ПС ошибки. Поэтому возникают две задачи.

Первая задача: подготовить такой набор тестов и применить к ним ПС, чтобы обнаружить в нем по возможности большее число ошибок. Однако чем дольше продолжается процесс тестирования (и отладки в целом), тем большей становится стоимость ПС. Отсюда вторая задача: определить момент окончания отладки ПС (или отдельной его компоненты). Признаком возможности окончания отладки является полнота охвата пропущенными через ПС тестами (т.е. тестами, к которым применено ПС) множества различных ситуаций, возникающих при выполнении программ ПС, и относительно редкое проявление ошибок в ПС на последнем отрезке процесса тестирования. Последнее определяется в соответствии с требуемой степенью надежности ПС, указанной в спецификации его качества.

Для оптимизации набора тестов, то есть для подготовки такого набора тестов, который позволял бы при заданном их числе (или при заданном интервале времени, отведенном на тестирование) обнаруживать большее число ошибок в ПС, необходимо, во-первых, заранее планировать этот набор и, во-вторых, использовать рациональную стратегию планирования (проектирования) тестов. Проектирование тестов можно начинать сразу же после завершения этапа внешнего описания ПС.

Возможны разные подходы к выработке стратегии проектирования тестов, которые можно условно разместить между следующими двумя крайними подходами. Левый крайний подход заключается в том, что тесты проектируются только на основании изучения спецификаций ПС (внешнего описания, описания архитектуры и спецификации модулей). Строение модулей при

этом никак не учитывается, то есть они рассматриваются как черные ящики. Фактически такой подход требует полного перебора всех наборов входных данных, так как в противном случае некоторые участки программ ПС могут не работать при пропуске любого теста, а это значит, что содержащиеся в них ошибки не будут проявляться. Однако тестирование ПС полным множеством наборов входных данных практически неосуществимо. Правый крайний подход заключается в том, что тесты проектируются на основании изучения текстов программ с целью протестировать все пути выполнения каждой программ ПС. Если принять во внимание наличие в программах циклов с переменным числом повторений, то различных путей выполнения программ ПС может оказаться также чрезвычайно много, так что их тестирование также будет практически неосуществимо.

Оптимальная стратегия проектирования тестов расположена внутри интервала между этими крайними подходами, но ближе к левому краю. Она включает проектирование значительной части тестов по спецификациям, но она требует также проектирования некоторых тестов и по текстам программ. При этом в первом случае эта стратегия базируется на принципах:

- на каждую используемую функцию или возможность – хотя бы один тест,
- на каждую область и на каждую границу изменения какой-либо входной величины – хотя бы один тест,
- на каждую особую (исключительную) ситуацию, указанную в спецификациях, – хотя бы один тест.

Во втором случае эта стратегия базируется на принципе: каждая команда каждой программы ПС должна проработать хотя бы на одном тесте.

Оптимальную стратегию проектирования тестов можно конкретизировать на основании следующего принципа: для каждого программного документа (включая тексты программ), входящего в состав ПС, должны проектироваться свои тесты с целью выявления в нем ошибок. Во всяком случае, этот принцип необходимо соблюдать в соответствии с определением ПС и содержанием понятия технологии программирования как технологии разработки надежных ПС. В связи с этим выделяют разные виды тестирования в зависимости от вида программного документа, на

основании которого строятся тесты. В нашей стране различают два основных вида отладки (включая тестирование): автономную и комплексную отладку ПС.

Автономная отладка ПС означает последовательное раздельное тестирование различных частей программ, входящих в ПС, с поиском и исправлением в них фиксируемых при тестировании ошибок. Она фактически включает отладку каждого программного модуля и отладку сопряжения модулей.

Комплексная отладка означает тестирование ПС в целом с поиском и исправлением фиксируемых при тестировании ошибок во всех документах (включая тексты программ ПС), относящихся к ПС в целом. К таким документам относятся определение требований к ПС, спецификация качества ПС, функциональная спецификация ПС, описание архитектуры ПС и тексты программ ПС.

6.2.3 Заповеди отладки программного средства

Рассмотрим общие рекомендации по организации отладки ПС. Но сначала следует отметить некоторый феномен, который подтверждает важность предупреждения ошибок на предыдущих этапах разработки: по мере роста числа обнаруженных и исправленных ошибок в ПС растет также относительная вероятность существования в нем необнаруженных ошибок. Это объясняется тем, что при росте числа ошибок, обнаруженных в ПС, уточняется и наше представление об общем числе допущенных в нем ошибок, а значит, в какой-то мере, и о числе необнаруженных еще ошибок.

Ниже приводятся рекомендации по организации отладки в форме заповедей.

Заповедь 1. Считайте тестирование ключевой задачей разработки ПС, поручайте его самым квалифицированным и одаренным программистам; нежелательно тестировать свою собственную программу.

Заповедь 2. Хорош тот тест, для которого высока вероятность обнаружить ошибку, а не тот, который демонстрирует правильную работу программы.

Заповедь 3. Готовьте тесты как для правильных, так и для неправильных данных.

Заповедь 4. Документируйте пропуск тестов через компьютер; детально изучайте результаты каждого теста; избегайте тестов, пропуск которых нельзя повторить.

Заповедь 5. Каждый модуль подключайте к программе только один раз; никогда не изменяйте программу, чтобы облегчить ее тестирование.

Заповедь 6. Пропускайте заново все тесты, связанные с проверкой работы какой-либо программы ПС или ее взаимодействия с другими программами, если в нее были внесены изменения (например, в результате устранения ошибки).

6.2.4 Автономная отладка программного средства

При автономной отладке ПС каждый модуль на самом деле тестируется в некотором программном окружении, кроме случая, когда отлаживаемая программа состоит только из одного модуля. Это окружение состоит из других модулей, часть которых является модулями отлаживаемой программы, которые уже отлажены, а часть – модулями, управляющими отладкой (отладочными модулями, см. ниже). Таким образом, при автономной отладке тестируется всегда некоторая программа (тестируемая программа), построенная специально для тестирования отлаживаемого модуля. Эта программа лишь частично совпадает с отлаживаемой программой, кроме случая, когда отлаживается последний модуль отлаживаемой программы. В процессе автономной отладки ПС производится наращивание тестируемой программы отлаженными модулями: при переходе к отладке следующего модуля в его программное окружение добавляется последний отлаженный модуль. Такой процесс наращивания программного окружения отлаженными модулями называется интеграцией программы. Отладочные модули, входящие в окружение отлаживаемого модуля, зависят от порядка, в каком отлаживаются модули этой программы, от того, какой модуль отлаживается и, возможно, от того, какой тест будет пропускаться.

При восходящем тестировании это окружение будет содержать только один отладочный модуль (кроме случая, когда отлаживается последний модуль отлаживаемой программы), который будет головным в тестируемой программе. Такой отладочный модуль называют ведущим (или драйвером). Ведущий отладоч-

ный модуль подготавливает информационную среду для тестирования отлаживаемого модуля (т. е. формирует ее состояние, требуемое для тестирования этого модуля, в частности, путем ввода некоторых тестовых данных), осуществляет обращение к отлаживаемому модулю и после окончания его работы выдает необходимые сообщения. При отладке одного модуля для разных тестов могут составляться разные ведущие отладочные модули.

При нисходящем тестировании окружение отлаживаемого модуля в качестве отладочных модулей содержит отладочные имитаторы (заглушки) некоторых еще не отлаженных модулей. К таким модулям относятся, прежде всего, все модули, к которым может обращаться отлаживаемый модуль, а также еще не отлаженные модули, к которым могут обращаться уже отлаженные модули (включенные в это окружение). Некоторые из этих имитаторов при отладке одного модуля могут изменяться для разных тестов.

На практике в окружении отлаживаемого модуля могут содержаться отладочные модули обоих типов, если используется смешанная стратегия тестирования. Это связано с тем, что как восходящее, так и нисходящее тестирование имеет свои достоинства и свои недостатки.

К достоинствам восходящего тестирования относятся:

- простота подготовки тестов,
- возможность полной реализации плана тестирования модуля.

Это связано с тем, что тестовое состояние информационной среды готовится непосредственно перед обращением к отлаживаемому модулю (ведущим отладочным модулем).

Недостатками восходящего тестирования являются следующие его особенности:

- тестовые данные готовятся, как правило, не в той форме, которая рассчитана на пользователя (кроме случая, когда отлаживается последний, головной, модуль отлаживаемой программ);
- большой объем отладочного программирования (при отладке одного модуля приходится составлять много ведущих отладочных модулей, формирующих подходящее состояние информационной среды для разных тестов);

- необходимость специального тестирования сопряжения модулей.

К достоинствам нисходящего тестирования относятся следующие его особенности:

- большинство тестов готовится в форме, рассчитанной на пользователя;
- во многих случаях относительно небольшой объем отладочного программирования (имитаторы модулей, как правило, весьма просты и каждый пригоден для большого числа, нередко – для всех, тестов);
- отпадает необходимость тестирования сопряжения модулей.

Недостатком нисходящего тестирования является то, что тестовое состояние информационной среды перед обращением к отлаживаемому модулю готовится косвенно – оно является результатом применения уже отлаженных модулей к тестовым данным или данным, выдаваемым имитаторами. Это, во-первых, затрудняет подготовку тестов и требует высокой квалификации разработчика тестов, а во-вторых, делает затруднительным или даже невозможным реализацию полного плана тестирования отлаживаемого модуля. Указанный недостаток иногда вынуждает разработчиков применять восходящее тестирование даже в случае нисходящей разработки. Однако чаще применяют некоторые модификации нисходящего тестирования, либо некоторую комбинацию нисходящего и восходящего тестирования. Исходя из того, что нисходящее тестирование, в принципе, является предпочтительным, остановимся на приемах, позволяющих в какой-то мере преодолеть указанные трудности.

Прежде всего, необходимо организовать отладку программы таким образом, чтобы как можно раньше были отлажены модули, осуществляющие ввод данных, – тогда тестовые данные можно готовить в форме, рассчитанной на пользователя, что существенно упростит подготовку последующих тестов. Далеко не всегда этот ввод осуществляется в головном модуле, поэтому приходится в первую очередь отлаживать цепочки модулей, ведущие к модулям, осуществляющим указанный ввод. Пока модули, осуществляющие ввод данных, не отлажены, тестовые данные поставля-

ются некоторыми имитаторами: они либо включаются в имитатор как его часть, либо вводятся этим имитатором.

При нисходящем тестировании некоторые состояния информационной среды, при которых требуется тестировать отлаживаемый модуль, могут не возникать при выполнении отлаживаемой программы ни при каких входных данных. В этих случаях можно было бы вообще не тестировать отлаживаемый модуль, так как обнаруживаемые при этом ошибки не будут проявляться при выполнении отлаживаемой программы ни при каких входных данных. Однако так поступать не рекомендуется, так как при изменениях отлаживаемой программы (например, при сопровождении ПС) не использованные для тестирования отлаживаемого модуля состояния информационной среды могут уже возникать, что требует дополнительного тестирования этого модуля (а этого при рациональной организации отладки можно было бы не делать, если сам данный модуль не изменялся). Для осуществления тестирования отлаживаемого модуля в указанных ситуациях иногда используют подходящие имитаторы, чтобы создать требуемое состояние информационной среды. Чаще же пользуются модифицированным вариантом нисходящего тестирования, при котором отлаживаемые модули перед их интеграцией предварительно тестируются отдельно (в этом случае в окружении отлаживаемого модуля появляется ведущий отладочный модуль, наряду с имитаторами модулей, к которым может обращаться отлаживаемый модуль). Однако, представляется более целесообразной другая модификация нисходящего тестирования: после завершения нисходящего тестирования отлаживаемого модуля для достижимых тестовых состояний информационной среды следует его отдельно протестировать для остальных требуемых состояний информационной среды.

Часто применяют также комбинацию восходящего и нисходящего тестирования, которую называют методом сэндвича. Сущность этого метода заключается в одновременном осуществлении как восходящего, так и нисходящего тестирования, пока эти два процесса тестирования не встретятся на каком-либо модуле где-то в середине структуры отлаживаемой программы. Этот метод при разумном порядке тестирования позволяет воспользоваться достоинствами как восходящего, так и нисходящего

тестирования, а также в значительной степени нейтрализовать их недостатки.

Весьма важным при автономной отладке является тестирование сопряжения модулей. Дело в том, что спецификация каждого модуля программы, кроме головного, используется в этой программы в двух ситуациях: во-первых, при разработке текста (иногда говорят: тела) этого модуля и, во-вторых, при написании обращения к этому модулю в других модулях программы. И в том, и в другом случае в результате ошибки может быть нарушено требуемое соответствие заданной спецификации модуля. Такие ошибки требуется обнаруживать и устранять. Для этого и предназначено тестирование сопряжения модулей. При нисходящем тестировании тестирование сопряжения осуществляется попутно каждым пропускаемым тестом, что считают достоинством нисходящего тестирования. При восходящем тестировании обращение к отлаживаемому модулю производится не из модулей отлаживаемой программы, а из ведущего отладочного модуля. В связи с этим существует опасность, что последний модуль может приспособиться к некоторым «заблуждениям» отлаживаемого модуля. Поэтому, приступая (в процессе интеграции программы) к отладке нового модуля, приходится тестировать каждое обращение к ранее отлаженному модулю с целью обнаружения несогласованности этого обращения с телом соответствующего модуля (и не исключено, что виноват в этом ранее отлаженный модуль). Таким образом, приходится частично повторять в новых условиях тестирование ранее отлаженного модуля, при этом возникают те же трудности, что и при нисходящем тестировании.

Автономное тестирование модуля целесообразно осуществлять в четыре последовательно выполняемых шага.

Шаг 1. На основании спецификации отлаживаемого модуля подготовить тесты для каждой возможности и каждой ситуации, для каждой границы областей допустимых значений всех входных данных, для каждой области изменения данных, для каждой области недопустимых значений всех входных данных и каждого недопустимого условия.

Шаг 2. Проверить текст модуля, чтобы убедиться, что каждое направление любого разветвления будет пройдено хотя бы на одном тесте. Добавить недостающие тесты.

Шаг 3. Проверить текст модуля, чтобы убедиться, что для каждого цикла существуют тесты, обеспечивающие, по крайней мере, три следующие ситуации: тело цикла не выполняется ни разу, тело цикла выполняется один раз и тело цикла выполняется максимальное число раз. Добавить недостающие тесты.

Шаг 4. Проверить текст модуля, чтобы убедиться, что существуют тесты, проверяющие чувствительность к отдельным особым значениям входных данных. Добавить недостающие тесты.

6.2.5 Комплексная отладка программного средства

Как уже было сказано выше, при комплексной отладке тестируется ПС в целом, причем тесты готовятся по каждому из документов ПС. Тестирование этих документов производится, как правило, в порядке, обратном их разработке. Исключение составляет лишь тестирование документации по применению, которая разрабатывается по внешнему описанию параллельно с разработкой текстов программ – это тестирование лучше производить после завершения тестирования внешнего описания. Тестирование при комплексной отладке представляет собой применение ПС к конкретным данным, которые в принципе могут возникнуть у пользователя (в частности, все тесты готовятся в форме, рассчитанной на пользователя), но, возможно, в моделируемой (а не в реальной) среде. Например, некоторые недоступные при комплексной отладке устройства ввода и вывода могут быть заменены их программными имитаторами.

Тестирование архитектуры ПС. Целью тестирования является поиск несоответствия между описанием архитектуры и совокупностью программ ПС. К моменту начала тестирования архитектуры ПС должна быть уже закончена автономная отладка каждой подсистемы. Ошибки реализации архитектуры могут быть связаны, прежде всего, с взаимодействием этих подсистем, в частности, с реализацией архитектурных функций (если они есть). Поэтому хотелось бы проверить все пути взаимодействия между подсистемами ПС. При этом желательно хотя бы протестировать все цепочки выполнения подсистем без повторного вхождения последних. Если заданная архитектура представляет ПС в качестве малой системы из выделенных подсистем, то число таких цепочек будет вполне обозримо.

Тестирование внешних функций. Целью тестирования является поиск расхождений между функциональной спецификацией и совокупностью программ ПС. Несмотря на то, что все эти программы автономно уже отлажены, указанные расхождения могут быть, например, из-за несоответствия внутренних спецификаций программ и их модулей (на основании которых производилось автономное тестирование) функциональной спецификации ПС. Как правило, тестирование внешних функций производится так же, как и тестирование модулей на первом шаге, т.е. как черного ящика.

Тестирование качества ПС. Целью тестирования является поиск нарушений требований качества, сформулированных в спецификации качества ПС. Это наиболее трудный и наименее изученный вид тестирования. Ясно лишь, что далеко не каждый примитив качества ПС может быть испытан. Завершенность ПС проверяется уже при тестировании внешних функций. На данном этапе тестирование этого примитива качества может быть продолжено, если требуется получить какую-либо вероятностную оценку степени надежности ПС. Однако, методика такого тестирования еще требует своей разработки. Могут тестироваться такие примитивы качества, как точность, устойчивость, защищенность, временная эффективность, в какой-то мере – эффективность по памяти, эффективность по устройствам, расширяемость и, частично, независимость от устройств. Каждый из этих видов тестирования имеет свою специфику и заслуживает отдельного рассмотрения. Мы здесь ограничимся лишь их перечислением. Легкость применения ПС оценивается при тестировании документации по применению ПС.

Тестирование документации по применению ПС. Целью тестирования является поиск несогласованности документации по применению и совокупностью программ ПС, а также выявление неудобств, возникающих при применении ПС. Этот этап непосредственно предшествует подключению пользователя к завершению разработки ПС (тестированию определения требований к ПС и аттестации ПС), поэтому весьма важно разработчикам сначала самим воспользоваться ПС так, как это будет делать пользователь. Все тесты на этом этапе готовятся исключительно на основании только документации по применению ПС. Прежде

всего, должны тестироваться возможности ПС как это делалось при тестировании внешних функций, но только на основании документации по применению. Должны быть протестированы все неясные места в документации, а также все примеры, использованные в документации. Далее тестируются наиболее трудные случаи применения ПС с целью обнаружить нарушение требований относительности легкости применения ПС.

Тестирование определения требований к ПС. Целью тестирования является выяснение, в какой мере ПС не соответствует предъявленному определению требований к нему. Особенность этого вида тестирования заключается в том, что его осуществляет организация-покупатель или организация-пользователь ПС как один из путей преодоления барьера между разработчиком и пользователем. Обычно это тестирование производится с помощью контрольных задач – типовых задач, для которых известен результат решения. В тех случаях, когда разрабатываемое ПС должно придти на смену другой версии ПС, которая решает хотя бы часть задач разрабатываемого ПС, тестирование производится путем решения общих задач с помощью как старого, так и нового ПС (с последующим сопоставлением полученных результатов). Иногда в качестве формы такого тестирования используют опытную эксплуатацию ПС – ограниченное применение нового ПС с анализом использования результатов в практической деятельности. По существу, этот вид тестирования во многом перекликается с испытанием ПС при его аттестации, но выполняется до аттестации, а иногда и вместо аттестации.

6.3 ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Данное занятие предполагает выполнение следующих этапов:

- изучить методические указания;
- выполнить выданное задание;
- ответить на контрольные вопросы.

6.4 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что понимается под отладкой программного средства?
2. Что понимается под тестированием программного средства?
3. Что такое автономная отладка программного средства?
4. Что такое комплексная отладка программного средства?
5. Что такое ведущий отладочный модуль?
6. Что такое отладочный имитатор программного модуля?

7 ОРГАНИЗАЦИЯ ОБРАБОТКИ ИСКЛЮЧЕНИЙ

7.1 ЦЕЛЬ РАБОТЫ

Цель работы – изучение процесса возникновения, обнаружения неправильных действий программы и механизма обработки исключений.

7.2 ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ

7.2.1 Основные понятия

Примерами неправильных действий в программе могут служить попытки деления на нуль или, допустим, выход индекса за пределы объявленного диапазона. При обнаружении такого рода попытки система, в которой отсутствуют средства обработки исключений, реагирует сообщением о ней и прерыванием работы программы. Это очень нежелательное аварийное завершение программы. Программист должен отреагировать на него определенным образом – внести такие изменения в программу, которые позволят программе завершиться естественным образом, и снова запустить ее на выполнение. Такой подход утомителен и может привести к новым ошибкам.

Работа C#-подсистемы вносит существенные коррективы в этот порядок. Она организована таким образом, что в программе может быть определен блок кода, называемый «обработчиком исключений», который автоматически выполняется при выявлении недопустимых действий в программе, и ее работоспособность не теряется. Достигается такой эффект с помощью определенных операторов языка и грамотного их применения программистом. Таким образом, механизм обработки исключительных ситуаций позволяет программе, содержащей некоторые ошибки, продолжить свое выполнение. Достигается это за счет логического разделения вычислительного процесса на две части: обнаружение аварийной ситуации и ее обработки. Это важно не только для лучшей структуризации программы. Главное то, что обе эти части работают совместно, но разделяя действия между собой: одна часть обнаруживает ошибку, а другая реагирует на нее.

Еще одним преимуществом обработки исключительных ситуаций в C# является определение стандартных исключений для таких распространенных программных ошибок, как деление на нуль, или попадание вне диапазона определения индекса, или ряда других. Чтобы отреагировать на возникновение таких ошибок, программа должна отслеживать и обрабатывать эти исключения. Без знания возможностей C#-подсистемы обработки исключений успешное программирование на C# попросту невозможно.

7.2.2 Механизм обработки исключений

В C# исключения представляются классами. Все классы исключений должны быть выведены из встроенного класса исключений Exception, который является частью пространства имен System. Таким образом, все исключения – подклассы класса Exception. Класс Exception имеет два производных класса System Exception и Application Exception, поддерживающие две общие категории исключений:

- исключения, которые генерируются общей системой поддержки времени выполнения C#;
- исключения, которые генерируются прикладными программами C#.

В обработке исключений используются несколько ключевых слов языка: try, catch, finally, throw, использование которых взаимосвязано. Так например ключевые слова catch, finally нельзя использовать без ключевого слова try.

Программные инструкции, которые нужно проконтролировать на предмет исключений, помещаются в try-блок. Оператор try содержит три части:

- контролируемый блок, предваряемый ключевым словом try. В контролируемый блок включаются потенциально опасные операторы программы. Все функции, прямо или косвенно вызываемые из блока, также считаются ему принадлежащими;
- один или несколько обработчиков исключений – блоков catch, в которых описывается, как обрабатываются ошибки различных типов;
- блок завершения finally выполняется независимо от того, возникла ошибка в контролируемом блоке или нет.

Рассмотрим, каким образом реализуется обработка исключительных ситуаций.

- Обработка исключения начинается с появления ошибки. Функция или операция, в которой возникла ошибка, генерирует исключение. Часто оно генерируется в функциях, вложенных в блок try.
- Выброшенное исключение может быть перехвачено программным путем с помощью catch-блока и обработано соответствующим образом.
- Если обработчик не найден, вызывается стандартный обработчик исключения. Его действия зависят от конфигурации среды. Обычно он выводит на экран окно с информацией об исключении и завершает текущий процесс. Системные исключения автоматически генерируются C#-системой динамического управления.
- Чтобы сгенерировать исключение вручную, используется ключевое слово throw.
- Любой код, который должен быть обязательно выполнен при выходе из try-блока, помещается в блок finally.

Отсутствовать могут либо блоки catch, либо блок finally, но не оба одновременно.

Итак, ядром обработки исключений являются блоки try и catch. Эти ключевые слова работают «в одной связке»; формат записи try/catch-блоков обработки исключений имеет следующий вид:

```
try {
// Блок кода, подлежащий проверке на наличие ошибок.
}
catch (Тип искл1 объект искл) {
// Обработчик для исключения Тип искл1
}
catch (Тип искл21 объект искл) {
// Обработчик для исключения Тип искл2
}
```

Здесь Тип искл – это тип сгенерированного исключения. После «выброса» исключение перехватывается соответствующей инструкцией catch, которая его обрабатывает. Как видно из формата записи try/catch-блоков, с try-блоком может быть связана не

одна, а несколько catch-инструкций. Их последовательность должна располагаться непосредственно за блоком try. Блоки catch просматриваются в том порядке, в котором они записаны. Какой именно из них будет выполнен, определит тип исключения. Другими словами, будет выполнена та catch-инструкция, тип исключения которой совпадает с типом сгенерированного исключения (а все остальные будут проигнорированы). После перехвата исключения параметр объект искл, если он присутствует, примет значение Тип искл.

Таблица 1 – Наиболее часто используемые исключения

Исключение	Значение
ArrayTypeMismatchException	Тип сохраняемого значения несовместим с типом массива
DivideByZeroException	Попытка деления на нуль
IndexOutOfRangeException	Индекс массива оказался вне диапазона
InvalidCastException	Неверно выполнено динамическое приведение типов
OutofMemoryException	Недостаточный объем свободной памяти
OverflowException	Имеет место арифметическое переполнение
NullReferenceException	Попытка использовать нулевую ссылку
StackoverflowException	Переполнение стека

Если обработчику исключения не нужен доступ к объекту исключения, то параметр объект искл можно опустить. Если исключение не генерируется, то try-блок завершается нормально, и все его catch-инструкции игнорируются. В таблице 1 представлены наиболее часто используемые исключения, определенные в пространстве имен System.

Иногда требуется перехватывать все исключения, независимо от их типа. Для этого используется catch-инструкция без параметров. В этом случае создается обработчик, который используется, чтобы программа гарантированно обработала все исключения.

В любом случае, произошло исключение или нет, управление передается в блок завершения finally (если он существует), а затем – первому оператору, находящемуся непосредственно за оператором try. В завершающем блоке обычно записываются операторы, которые необходимо выполнить независимо от того, возникло исключение или нет, например, закрытие файлов, с которыми выполнялась работа в контролируемом блоке, или вывод информации.

7.3 ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Данное занятие предполагает выполнение следующих этапов:

– изучить методические указания;
– выполнить выданное задание;
– ответить на контрольные вопросы.

7.4 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Приведите примеры неправильных действий при работе программы, в результате которых происходит прерывание работы программы.
2. Как организована работа C#-подсистемы при обнаружении неправильных действий в программе?
3. В чем заключается механизм обработки заключений?
4. Как устроен try-блок?
5. Каким образом реализуется обработка исключительных ситуаций?

8 ПРИМЕНЕНИЕ ОТЛАДОЧНЫХ КЛАССОВ В ПРОЕКТЕ

8.1 ЦЕЛЬ РАБОТЫ

Цель работы – изучение свойств и методов встроенных классов отладки проектов и применения этих классов для отладки программ.

8.2 ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ

8.2.1 Классы Debug и Trace

Атрибут условной компиляции Conditional характеризует метод, но не отдельный оператор. Иногда хотелось бы иметь условный оператор печати, не создавая специального метода. Такую возможность и многие другие полезные свойства предоставляют классы Debug и Trace.

Классы Debug и Trace – это классы-двойники. Оба они находятся в пространстве имен Diagnostics, имеют идентичный набор статических свойств и методов с идентичной семантикой. В чем же разница? Методы класса Debug имеют атрибут условной компиляции с константой DEBUG, действуют только в Debug-конфигурации проекта и игнорируются в Release-конфигурации. Методы класса Trace включают два атрибута Conditionalc константами DEBUG и TRACE и действуют в обеих конфигурациях.

Одна из основных групп методов этих классов – методы печати данных: Write, WriteIf, WriteLine, WriteLineIf. Методы перегружены, в простейшем случае позволяют выводить некоторое сообщение. Методы со словом If могут сделать печать условной, задавая условие печати в качестве первого аргумента метода, что иногда крайне полезно. Методы со словом Line дают возможность дополнять сообщение символом перехода на новую строку.

По умолчанию методы обоих классов направляют вывод в окно Output. Однако это не всегда целесообразно, особенно для Release-конфигурации. Замечательным свойством методов классов Debug и Trace является то, что они могут иметь много «слушателей», направляя вывод каждому из них. Свойство Listeners этих классов возвращает разделяемую обоими классами коллек-

цию слушателей – TraceListenerCollection. Как и всякая коллекция, она имеет ряд методов для добавления новых слушателей: Add, AddRange, Insert – и возможность удаления слушателей: Clear, Remove, RemoveAt и другие методы. Объекты этой коллекции в качестве предка имеют абстрактный класс TraceListener. Библиотека FCL включает три неабстрактных потомка этого класса:

- DefaultTraceListener – слушатель этого класса, добавляется в коллекцию по умолчанию, направляет вывод, поступающий при вызове методов классов Debug и Trace, в окно Output;
- EventLogTraceListener – посылает сообщения в журнал событий Windows;
- TextWriterTraceListener – направляет сообщения объектам класса TextWriter или Stream; обычно один из объектов этого класса направляет вывод на консоль, другой – в файл.

Можно и самому создать потомка абстрактного класса, предложив, например, XML-слушателя, направляющего вывод в соответствующий XML-документ. Как видите, система управления выводом очень гибкая, позволяющая получать и сохранять информацию о ходе вычислений в самых разных местах.

Помимо свойства Listeners и методов печати, классы Debug и Trace имеют и другие важные методы и свойства:

- Assert и Fail, проверяющие корректность хода вычислений – о них мы поговорим особо;
- Flush – метод, отправляющий содержание буфера слушателю (в файл, на консоль и так далее). Следует помнить, что данные буферизуются, поэтому применение метода Flush зачастую необходимо, иначе метод может завершиться, а данные останутся в буфере;
- AutoFlush – булево свойство, указывающее, следует ли после каждой операции записи данные из буфера направлять в соответствующий канал. По умолчанию свойство выключено, и происходит только буферизация данных;
- Close – метод, опустошающий буфера и закрывающий всех слушателей, после чего им нельзя направлять сообщения.

У классов есть и другие свойства и методы, позволяющие, например, заниматься структурированием текста сообщений.

Рассмотрим пример работы, в котором отладочная информация направляется в разные каналы – окно вывода, консоль, файл:

```
public void Optima()
{
        double x, y=1;
        x= y - 2*Math.Sin(y);
        FileStream f = new FileStream(«Debuginfo.txt»,
          FileMode.Create, FileAccess.Write);
        TextWriterTraceListener writer1 =
          new TextWriterTraceListener(f);
        TextWriterTraceListener writer2 =
          new TextWriterTraceListener(System.Console.Out);
        Trace.Listeners.Add( writer1);
        Debug.Listeners.Add( writer2);
        Debug.WriteLine(«Число слушателей:» +
          Debug.Listeners.Count);
        Debug.WriteLine(«автоматический вывод из буфера:»+
          Trace.AutoFlush);
        Trace.WriteLineIf(x<0, «Trace: « + «x= « + x.ToString()
           + « y = « + y);
        Debug.WriteLine(«Debug: « + «x= « + x.ToString() +
          « y = « + y);
        Trace.Flush();
        f.Close();
}
```

В коллекцию слушателей вывода к слушателю по умолчанию добавляются еще два слушателя класса TextWriterTraceListener. Заметьте, что хотя они добавляются методами разных классов Debug и Trace, попадают они в одну коллекцию. Как и обещано, один из этих слушателей направляет вывод в файл, другой на консоль. На рисунке 8.1 на фоне окна кода показаны три канала вывода – окно Output, консоль, файл – содержащие одну и ту же информацию.

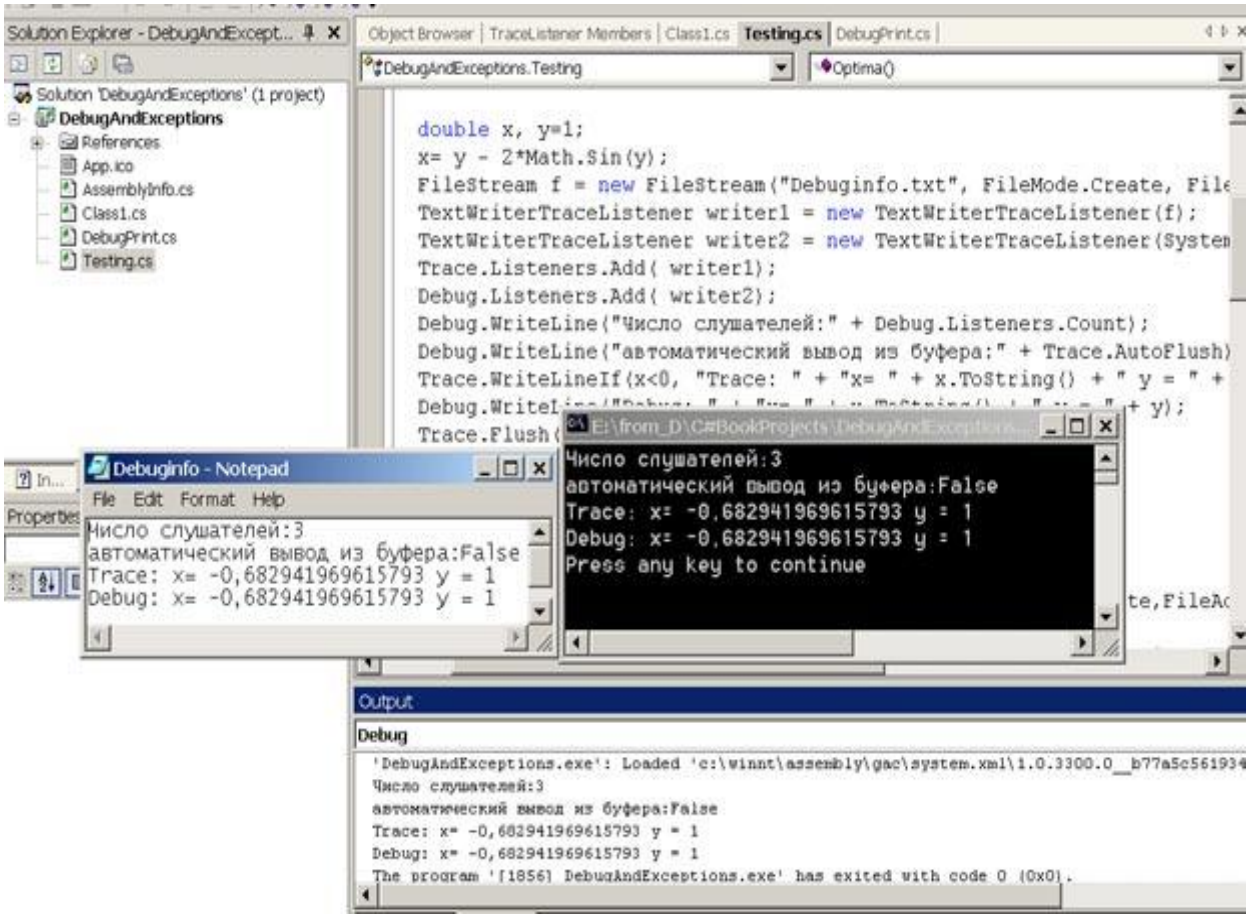

Рисунок 8.1 – Три канала вывода

8.2.2 Метод Флойда и утверждения Assert

Лет двадцать назад большие надежды возлагались на формальные методы доказательства правильности программ, позволяющие доказывать корректность программ аналогично доказательству теорем. Реальные успехи формальных доказательств невелики. Построение такого доказательства не проще написания корректной программы, а ошибки столь же возможны и часты, как и ошибки программирования. Тем не менее, эти методы оказали серьезное влияние на культуру проектирования корректных программ, появление в практике программирования понятий предусловия и постусловия, инвариантов и других важных понятий.

Одним из методов доказательства правильности программ был метод Флойда, при котором программа разбивалась на участки, окаймленные утверждениями – булевскими выражениями (предикатами). Истинность начального предиката должна была

следовать из входных данных программы. Затем для каждого участка доказывалось, что из истинности предиката, стоящего в начале участка, после завершения выполнения соответствующего участка программы гарантируется истинность следующего утверждения – предиката в конце участка. Конечный предикат описывал постусловие программы.

Схема Флойда используется на практике, по крайней мере, программистами, имеющими вкус к строгим методам доказательства. Утверждения становятся частью программного текста. Само доказательство может и не проводиться: чаще всего у программиста есть уверенность в справедливости расставленных утверждений и убежденность, что при желании он мог бы провести и строгое доказательство. В C# эта схема поддерживается тем, что классы Debug и Trace имеют метод Assert, аргументом которого является утверждение. Что происходит, когда вычисление достигает соответствующей точки и вызывается метод Assert? Если истинно булево выражение в Assert, то вычисления продолжаются, не оказывая никакого влияния на нормальный ход вычислений. Если оно ложно, то корректность вычислений под сомнением, их выполнение приостанавливается и появляется окно с уведомлением о произошедшем событии.

В этой ситуации у программиста есть несколько возможностей:

- прервать выполнение, нажав кнопку Abort;
- перейти в режим отладки (Retry);
- продолжить вычисления, проигнорировав уведомление.

В последнем случае сообщение о возникшей ошибке будет послано всем слушателям коллекции TraceListenerCollection.

Рассмотрим простой пример, демонстрирующий нарушение утверждения:

```
public void WriteToFile()
{
        Stream myFile = new
         File-
Stream(«TestFile.txt»,FileMode.Create,FileAccess.Write);
        TextWriterTraceListener myTextListener =
          new TextWriterTraceListener(myFile);
```

```
            int y = Debug.Listeners.Add(myTextListener);
            TextWriterTraceListener myWriter =
               new TextWriterTraceListener(System.Console.Out);
            Trace.Listeners.Add(myWriter);
            Trace.AutoFlush = true;
            Trace.WriteLine(«автоматический вывод из буфера:»
               + Trace.AutoFlush);
            int x = 22;
            Trace.Assert(x<=21, «Перебор»);
            myWriter.WriteLine(«Вывод только на консоль»);
            //Trace.Flush();
            //Вывод только в файл
            byte[] buf = {(byte)'B',(byte)'y'};
            myFile.Write(buf,0, 2);
            myFile.Close();
}
```

Как и в предыдущем примере, здесь создаются два слушателя, направляющие вывод отладочных сообщений на консоль и в файл. Когда произошло нарушение утверждения Assert, оно было проигнорировано, но сообщение о нем автоматически было направлено всем слушателям. Метод также демонстрирует возможность параллельной работы с консолью и файлом. На рисунке8.2 показаны результаты записи в файл:

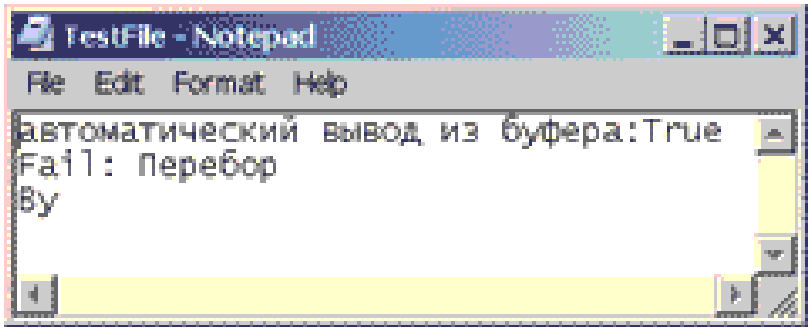

Рисунок 8.2 – Файл с записью сообщения
о нарушении утверждения Assert

Вариацией метода Assert является метод Fail, всегда приводящий к появлению окна с сообщением о нарушении утвержде-

ния, проверка которого осуществляется обычным программным путем.

8.2.3 Классы StackTrace и BooleanSwitch

В библиотеке FCL имеются и другие классы, полезные при отладке. Класс StackTrace позволяет получить программный доступ к стеку вызовов. Класс BooleanSwitch предоставляет механизм, аналогичный константам условной компиляции. Он разрешает определять константы, используемые позже в методе условной печати WriteIf классов Debug и Trace. Мощь этого механизма в том, что константы можно менять в файле конфигурации проекта, не изменяя код проекта и не требуя его перекомпиляции.

8.3 ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Данное занятие предполагает выполнение следующих этапов:

– изучить методические указания;
– выполнить выданное задание;
– ответить на контрольные вопросы.

8.4 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Дайте характеристику свойств и методов классов Debug и Trace.
2. В чем сходство и различие классов Debug и Trace?
3. Основные принципы метода доказательства правильности программ – метода Флойда.
4. Дайте характеристику свойств и методов классов StackTrace и BooleanSwitch.

9 ОТЛАДКА ПРОЕКТА

9.1 ЦЕЛЬ РАБОТЫ

Цель работы – изучить действия необходимые для отладки проекта.

9.2 ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ

Любой серьезный проект требует отладки. Программисту необходимо проверить правильность выполнения написанных им алгоритмов, а в случае некорректной работы программы найти и устранить возникшие ошибки.

Режим отладки проекта дает программисту возможность:

– указать точки останова в программе (точки, где выполнение программы будет остановлено);

– запустить программу до ближайшей точки останова;

– выполнить программу по шагам;

– просмотреть значения переменных в процессе выполнения программы.

Рассмотрим действия, которые необходимо выполнить для отладки проекта.

Установка точки останова. В файле с исходным кодом программы нужно установить курсор на строчку, где требуется остановка. После этого выполнить одно из действий:

– Debug → Toggle Breakpoint;

– нажать горячую клавишу F9;

– щелкнуть мышкой слева от текста программы напротив требуемой строки (рисунок 9.1).

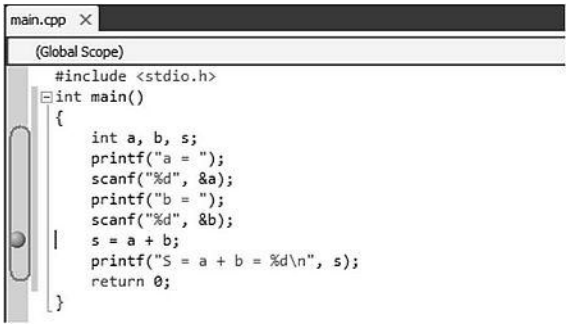

Рисунок 9.1 – Текст программы

Запуск программы до ближайшей точки останова. После указания точки останова можно запустить программу: команда Debug → Start Debugging или нажать горячую клавишу F5.

Программа начнет выполняться до тех пор, пока не будет достигнута точка останова. О достижении этой точки будет свидетельствовать стрелочка, отображаемая поверх точки останова (рисунок 9.2). В этой точке выполнение программы будет остановлено до указания дальнейших действий.

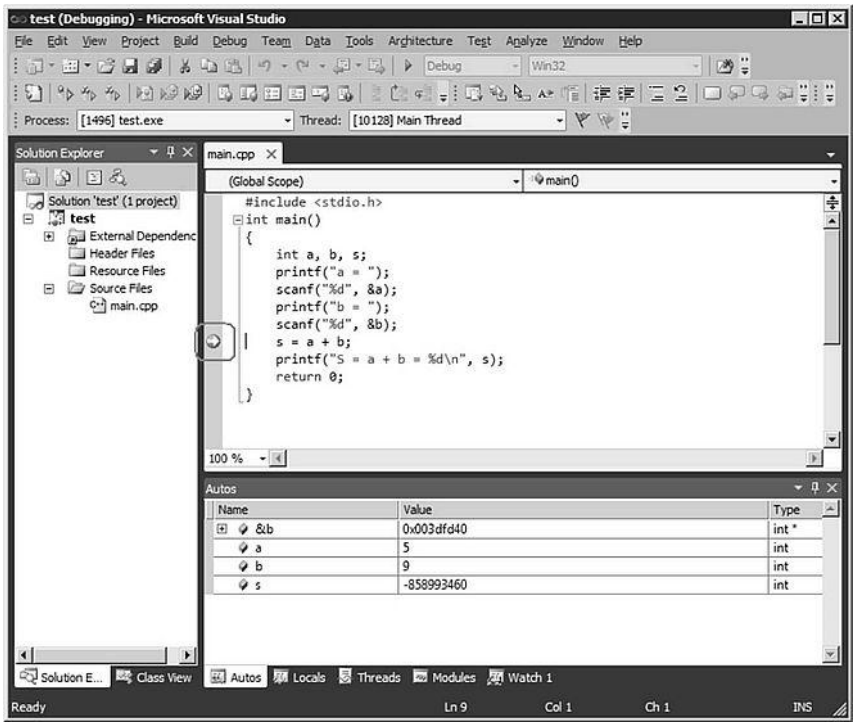

Рисунок 9.2 – Достижение точки останова в программе

Выполнение программы по шагам. Выполнение программы по шагам можно осуществить следующим образом. Команда Debug → Step Over выполняет одну строчку программы.

Если очередным шагом программы является вызов функции, то программист может проанализировать работу этой функции. Для этого в режиме отладки используется команда Debug → Step Into или горячая клавиша F11. После этой команды отладчик заходит «внутрь» функции и программист может продолжить пошаговое выполнение команд. Если отладка функции закончена и необходимо вернуться в вызывающую программу, то можно использовать команду Debug → Step Out или сочетание клавиш Shift+Fll.

Просмотр значения переменных в процессе выполнения программы. В любой момент отладки программист может просмотреть значения переменных, используемых в программе. Для этого используется нижняя часть окна среды Microsoft Visual Studio (рисунок 9.2). В этом подокне доступны несколько вкладок:

– Autos – список переменных, создаваемый отладчиком автоматически;
– Local – список локальных переменных;
– Threads – список потоков;
– Modules – список модулей, используемых для работы программы;
– Watch 1 – список переменных, указываемых программистом самостоятельно.

Для наблюдения значений переменных используются вкладки Autos и Watch 1.

Для добавления переменной на вкладку Watch 1 необходимо сделать ее активной и ввести с клавиатуры имя переменной. Если указанная переменная имеется в программе, то ее значение будет автоматически отражено в поле Value.

Таким образом, используя описанные приемы, программист может тщательно проанализировать выполнение написанной программы, проследить изменения значений используемых переменных, устранить некорректную работу реализованных алгоритмов и добиться правильного выполнения программы.

9.3 ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Данное занятие предполагает выполнение следующих этапов:

– изучить методические указания;
– выполнить выданное задание;
– ответить на контрольные вопросы.

9.4 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое ИСР (IDE)?
2. Какие составные части включает в себя среда Microsoft Visual Studio?

3. Опишите процесс создания исполняемого файла программы (*.exe).

4. Какими способами можно создать проект в среде Microsoft Visual Studio?

5. Какие команды служат для сохранения проекта? Объясните различие между доступными командами.

6. Как закрыть проект после окончания работы с ним?

7. Что включает в себя компиляция проекта?

8. Как найти ошибку в тексте программы, если она была обнаружена компилятором?

9. Какие возможности отладки проекта предоставляет среда Microsoft Visual Studio?

10. Каким образом можно просмотреть значения переменных при отладке проекта?

10 ИНСПЕКЦИЯ КОДА МОДУЛЕЙ ПРОЕКТА

10.1 ЦЕЛЬ РАБОТЫ

Цель работы – изучение формальных инспекций программного кода.

10.2 ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ

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

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

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

Факт начала формальной инспекции четко фиксируется в общей базе данных проекта. Также фиксируются документы, подвергаемые инспекции, списки замечаний, отслеживаются внесенные по замечаниям изменения. Этим формальная инспекция похожа на автоматизированное тестирование – списки замечаний

имеют много общего с отчетами о выполнении тестовых примеров.

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

Процесс формальной инспекции состоит из пяти фаз:

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

Руководитель проекта выбирает объект инспекции. Затем он назначает участников формальной инспекции: авторов, ведущего и нескольких инспекторов. Ведущий также выполняет роль инспектора; остальные участники выполняют только одну роль.

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

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

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

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

Перед началом просмотра исходного кода рекомендуется отметить пункты требований, на соответствие которым проверяется исходный код, а также записать обоснования того, почему эти требования не могут быть проверены в автоматическом режиме. После этого можно переходить к просмотру собственно исходного кода. Все пометки, которые придется вносить в ходе инспектирования в исходный код, необходимо делать не в файле, который будет выдан инспекторам, а в его копии, которая потом будет подшита к материалам инспекции. Копия может быть в том же формате, что и исходный файл, либо распечатана на бумаге или выведена в формат DOC, PDF или аналогичный, допускающий комментирование.

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

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

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

10.3 ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Данное занятие предполагает выполнение следующих этапов:

– изучить методические указания;
– выполнить выданное задание;
– ответить на контрольные вопросы.

10.4 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Краткая характеристика метода экспертных исследований программного кода или документации на корректность или непротиворечивость.

2. Дайте характеристику этапов формальной инспекции и ролей её участников.

3. Характеристика Инициализации как фазы формальной инспекции.

4. Характеристика Планирования как фазы формальной инспекции.

5. Характеристика Подготовки как фазы формальной инспекции.

6. В чём заключается сущность неформальной инспекции?

11 ТЕСТИРОВАНИЕ ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ СРЕДСТВАМИ ИНСТРУМЕНТАЛЬНОЙ СРЕДЫ РАЗРАБОТКИ

11.1 ЦЕЛЬ РАБОТЫ

Цель работы – изучения процесса тестирования интерфейса пользователя различными методами.

11.2 ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ

Графический интерфейс пользователя (Graphical user interface, GUI) – разновидность интерфейса обеспечивающее взаимодействие через графические элементы (меню, кнопки, значки, списки и т. п.).

Варианты реализации GUI:

– интерфейс настольного приложения;
– мобильный интерфейс;
– веб-интерфейс.

Цели тестирования:

– обеспечение качественного взаимодействия с пользователем;
– выявление ошибок функциональности;
– выявление необработанных исключений при взаимодействии с интерфейсом;
– выявление потери или искажения данных, передаваемых через элементы интерфейса;
– выявление ошибки в интерфейсе (несоответствие проектной документации, отсутствие элементов интерфейса).

Особенности тестирования:

– тест планы в виде сценариев работы пользователя;
– сценарии на естественном языке или в виде скриптов;
– выполнение тестов в ручном режиме или с помощью эмулятора;
– анализ экранных форм и видимых элементов, а не внутренних переменных.

Покрытие – участие интерфейсных элементов в тестах.

Найденный дефект: несоответствие реального поведения требованиям или проблемы в требованиях.

Проблема отчета об ошибке: расплывчатость формулировок.

Типы требований к UI

– требования к внешнему виду пользовательского интерфейса и формам взаимодействия с пользователем;

– требования к размещению элементов управления на экранных формах;

– требования к содержанию и оформлению выводимых сообщений;

– требования к форматам ввода;

– требования по доступу к внутренней функциональности системы при помощи пользовательского интерфейса;

– требования к реакции системы на ввод пользователя;

– требования к времени отклика на команды пользователя.

Функциональное тестирование UI:

– анализ требований к пользовательскому интерфейсу;

– разработка тест-требований и тест-планов для проверки пользовательского интерфейса;

– выполнение тестовых примеров и сбор информации о выполнении тестов;

– определение полноты покрытия пользовательского интерфейса требованиями;

– составление отчетов о проблемах в случае не совпадения поведения системы и требований либо в случае отсутствия требований на отдельные интерфейсные элементы.

Оценка покрытия UI. Функциональное покрытие – покрытие требований к пользовательскому интерфейсу. Структурное покрытие – для обеспечения полного структурного покрытия каждый интерфейсный элемент должен быть использован в тестовых примерах хотя бы один раз. Структурное покрытие с учетом состояния элементов интерфейса и внутреннего состояния системы – поведение некоторых интерфейсных элементов может изменяться в зависимости от внутреннего состояния системы. Каждое такое различимое поведение интерфейсного элемента должно быть проверено.

Ручное тестирование. Плюсы:

– контроль корректности проводится человеком;

– поиск «косметических» дефектов;

– анализ успешности прохождения теста будет выполняться не по формальным признакам, а согласно человеческому восприятию.

Минусы:

– требуются значительные человеческие и временные ресурсы;

– при проведении регрессионного тестирования и вообще любого повторного тестирования – на каждой итерации повторного тестирования пользовательского интерфейса требуется участие тестировщика-оператора.

Автоматическое тестирование. Плюсы:

– снижение стоимости тестирования;

– высокая скорость выполнения;

– больший объем покрытия;

– не требуется участие оператора-тестировщика при проведении регрессионного тестирования или любого другого повторного тестирования продукта.

Минусы:

– анализ успешности прохождения теста будет выполняться по формальным признакам;

– невозможность поиска «косметических» дефектов;

– высокая стоимость поддержки по сравнению с «обычными»функциональными тестами.

Автоматизация тестирования UI:

1) Координатный метод.

```
click(120,70)
type_keys(«Hello world»)
```

2)Распознавание образов.

```
find(«OK_button.png»).click()
```

3) Подход, использующий механизмы реализации специальных возможностей (accessibility) и особенности реализации некоторых GUI фреймворков.

```
dlg.ColorComboBox.select(«Green»).
```

3)Гибридный метод.

Координатный метод – каждый элемент графического интерфейса пользователя ищется по координатам, заданным относительно окна или экрана

Плюсы:

– быстрая разработка тестов;

– простота поиска элементов при неизменности условий.

Минусы:

– крайне дорогая поддержка тестов;

– зависимость тестов от настроек платформы (разрешение, шрифты, DPI, ...);

– невозможность отслеживания состояний объекта (активна кнопка или нет, установлен флажок у checkbox или неустановлен, и т. д.).

Координатный метод позволяет тестировать:

– кнопка;

– относительные координаты кнопки;

– размер кнопки;

– абсолютные координаты.

Пример тестирования по координатному методу:

```
def main():
waitForObject(«:_QWidget»)
sendEvent(«QResizeEvent», «:_QWidget», 22, 22, 769, 474)
waitForObject(«:_QGraphicsItem»)
mouseClick(«:_QGraphicsItem», 221, 193, 1, Qt.LeftButton)
waitForObject(«:_QLineEdit»)
dragItemBy(«:_QLineEdit», 153, -191, 26, 198, 1,
Qt.LeftButton)
waitForObject(«:_QLineEdit»)
sendEvent(«QMouseEvent», «:_QLineEdit», QE-
vent.MouseButtonRelease, 179, 7, Qt.LeftButton, 1)
waitForObject(«:MDC: Авторизация_QGraphicsView»)
type(«:MDC: Авторизация_QGraphicsView», «<Ctrl+A>«)
waitForObject(«:MDC: Авторизация_QGraphicsView»)
type(«:MDC: Авторизация_QGraphicsView», «
squish@mail.ru»)
waitForObject(«:MDC: Авторизация_QGraphicsView»)
type(«:MDC: Авторизация_QGraphicsView», «<Tab>«)
waitForObject(«:MDC: Авторизация_QGraphicsView»)
type(«:MDC: Авторизация_QGraphicsView», «1234»)
waitForObject(«:MDC: Авторизация_CStartUpWidget»)
```

sendEvent(«QMoveEvent», «:MDC: Авторизация_CStartUpWidget», 577, 70, 734, 62)

mouseClick(«:MDC: Авторизация_QWidget», 66, 372, 1, Qt.LeftButton)

waitForObject(«:MDC v1.0.3.1.nightly_CContactListWidget»)

Распознавание образов – метод поиска элементов UI с использованием распознавания образов и (или) сравнение с образцом.

Плюсы:

– быстрая разработка тестов;

– простота поиска элементов при неизменности условий

Минусы:

– крайне дорогая поддержка тестов;

– зависимость тестов от настроек платформы (разрешение, шрифты, DPI, ...);

– низкая «интеллектуальность» тестов.

Пример теста, построенного на системе распознавания образов:

– Запустить приложение.

– Найти главное окно приложения.

– Найти требуемый элемент, сравнив с шаблоном.

– Произвести действие:

- Шаблон поиска
- Click( )
- Действие Accessibility-метод

Метод управления UI через встроенные в ОС/SDK технологии воздействия на элементы UI. Технологии:

– Microsoft Active Accessibility (MSAA);

– Assistive Technologies Service Provider Interface (AT-SPI).

Плюсы:

– более тесное взаимодействие с элементами UI;

– независимость от параметров отображения;

– близкий к кодированию язык.

Минусы:

– не всегда есть поддержка;

– слабая кроссплатформенность.

Гибридный подход – использование комбинации методов для улучшения результатов. Пример: управление элементами

формы через accessibility или координатный методы, анализ диаграммы через распознавание образов

ИнструментарийWindows:

– Coded UI (Visual Studio);

– Ranorex Automation Tools;

– Winium.Cruciatus (2GIS);

– Microsoft UI Automation (MSAA);

– AutoIt.

ИнструментарийAndroid:

– Espresso (Google);

– Robotium;

– UI Automator;

– Monkey testing.

Инструментарий Web;

– Selenium.

Универсальный инструментарий:

– Robot Framework.

ПримерSelenium.

[info] Executing: |open| https://www.google.com.ar/?gws_rd=ssl | |

[info] Wait for the new page to be fully loaded

[info] Executing: | click | id=lst-ib | |

[info] Executing: | type | id=lst-ib | test |

[info] Wait for all ajax requests to be done

[info] Executing: | sendKeys | id=lst-ib | ${KEY_ENTER} |

[info] Wait for the new page to be fully loaded

[info] Executing: | click | link=test – Википедия | |

[info] Test case passed

Тестирование удобства использования (Usability testing) – исследование, выполняемое с целью определения, удобен ли некоторый искусственный объект для его предполагаемого применения.

Цели:

– выявление сильных и слабых мест в интерфейсе для дальнейшего улучшения его в ходе итерационного процесса разработки;

– оценка общего качества интерфейса – например, для выбора одного из двух возможных вариантов.

Этапы тестирования:

1) Исследовательское – проводится после формулирования требований к системе и разработки прототипа интерфейса.

2) Оценочное – проводится после разработки низкоуровневых требований и детализированного прототипа пользовательского интерфейса. Оценочное тестирование углубляет исследовательское и имеет ту же цель.

3) Валидационное – проводится ближе к этапу завершения разработки. На этом этапе проводится анализ соответствия интерфейса программной системы стандартам, регламентирующим вопросы удобства интерфейса, проводится общее тестирование всех компонентов пользовательского интерфейса с точки зрения конечного пользователя.

4) Сравнительное – данный вид тестирования может проводиться на любом этапе разработки интерфейса.

Механизмы оценки

– Производительность, эффективность (efficiency) – сколько времени и шагов понадобится пользователю для завершения основных задач приложения, например, размещение новости, регистрации, покупка и т. д.

– Правильность (accuracy) – сколько ошибок сделал пользователь во время работы с приложением.

– Активизация в памяти (recall) – как много пользователь помнит о работе приложения после приостановки работы с ним на длительный период времени (повторное выполнение операций после перерыва должно проходить быстрее, чем у нового пользователя).

– Эмоциональная реакция (emotional response) – как пользователь себя чувствует после завершения задачи –растерян, испытал стресс. Порекомендует ли пользователь систему своим друзьям.

Стадии теста:

1) Подготовка – подготовка и проверка оборудования и места тестирования, начального состояния системы, наличия всех необходимых материалов – задания, мануалов и т. п.

2) Введение – приветствие участника, пояснение цели теста и процесса, предупреждение о записи действий и слов участника и т. д.

3) Непосредственно тестирование – роль проводящего тестирование минимальна, говорить («думать вслух») и делать должен участник.

4) «Разбор полетов» – вопросы участнику на тему субъективного удовлетворения от работы, замечаний и предложений, пояснения каких-либо действий участника входе теста.

Тестирование специальных возможностей (Accessibility testing) – проверка возможности использования программного обеспечения людьми с ограниченными возможностями, такими как:

– нарушения зрения;
– нарушения подвижности;
– нарушения слуха;
– нарушения когнитивной функции.

Инструменты:

– экранный диктор;
– экранная лупа;
– распознавание речи;
– экранная клавиатура.

Примеры тестов:

– Проверка правильной работы пользовательского интерфейса на дисплее с высоким DPI.

– Проверка правильной работы пользовательского интерфейса в высококонтрастном режиме.

– Проверка правильной работы пользовательского интерфейса при использовании экранной лупы.

– Проверка доступности с клавиатуры элементов управления пользовательского интерфейса.

– Проверка пользовательского интерфейса с использованием специализированного ПО.

11.3 ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Данное занятие предполагает выполнение следующих этапов:

– изучить методические указания;
– выполнить выданное задание;
– ответить на контрольные вопросы.

11.4 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Назовите цели тестирования и особенности тестирования при тестировании графического интерфейса пользователя (GUI).
2. Назовите типы требований к интерфейсу пользователя.
3. Назовите этапы функционального тестирования пользовательского интерфейса.
4. Назовите виды покрытий интерфейса пользователя (UI).
5. Достоинства и недостатки ручного тестирования.
6. Достоинства и недостатки автоматического тестирования.
7. Метод поиска элементов UI с использованием распознавания образов и (или) сравнение с образцом.
8. В чём сущность координатного метода тестирования UI?
9. Дайте характеристику Accessibility-метода.

12 РАЗРАБОТКА ТЕСТОВЫХ МОДУЛЕЙ ПРОЕКТА ДЛЯ ТЕСТИРОВАНИЯ ОТДЕЛЬНЫХ МОДУЛЕЙ

12.1 ЦЕЛЬ РАБОТЫ

Цель работы – изучение вопросов, связанных с модульным тестированием.

12.2 ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ

12.2.1 Задачи и цели модульного тестирования

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

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

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

При этом в ходе модульного тестирования решаются четыре основные задачи.

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

2) Поддержка разработки и рефакторинга низкоуровневой архитектуры системы и межмодульного взаимодействия – эта за-

дача больше свойственна «легким» методологиям типа XP, где применяется принцип тестирования перед разработкой (Testdriven development), при котором основным источником требований для программного модуля является тест, написанный до самого модуля. Однако, даже при классической схеме тестирования модульные тесты могут выявить проблемы в дизайне системы и нелогичные или запутанные механизмы работы с модулем.

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

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

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

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

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

12.2.2 Понятие модуля и его границ. Тестирование классов

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

– модуль – это часть программного кода, выполняющая одну функцию с точки зрения функциональных требований;

– модуль – это программный модуль, т. е. минимальный компилируемый элемент программной системы;

– модуль – это задача в списке задач проекта (с точки зрения его менеджера);

– модуль – это участок кода, который может уместиться на одном экране или одном листе бумаги;

– модуль – это один класс или их множество с единым интерфейсом.

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

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

В случае объектно-ориентированных систем более мелкое деление классов и использование отдельных методов в качестве тестируемых модулей нецелесообразно в связи с тем, что для тестирования каждого метода потребуется разработка тестового окружения, сравнимого по сложности с уже написанным программным кодом класса. Кроме того, декомпозиция класса нарушает

принцип инкапсуляции, согласно которому объекты каждого класса должны вести себя как единое целое с точки зрения других объектов.

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

– дефектов инкапсуляции, в результате которых, например, сокрытые данные класса оказывается недоступными при помощи соответствующих публичных методов;

– дефектов наследования, при наличии которых схема наследования блокирует важные данные или методы от классов-потомков;

– дефектов полиморфизма, при которых полиморфное поведение класса оказывается распространенным не на все возможные классы;

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

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

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

Однако существует альтернативный подход к тестированию класса, согласно которому все публичные методы должны предоставлять пользователю данного класса согласованную схему работы и достаточно проверить типичные корректные и некорректные сценарии работы с данным классом. То есть, например, в

классе, объекты которого представляют записи в телефонной книжке, одним из типичных сценариев работы будет «Создать запись – Искать запись и найти ее – удалить запись – искать запись вторично и получить сообщение об ошибке».

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

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

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

Тестирование изменений. Как уже упоминалось выше, модульные тесты – мощный инструмент проверки корректности изменений, внесенных в исходный код при рефакторинге. Однако, в результате рефакторинга только одного класса, как правило, не

меняется его внешний интерфейс с другими классами (интерфейсы меняются при рефакторинге сразу нескольких классов). В результате обычных эволюционных изменений системы у класса может меняться внешний интерфейс, причем как по формальным (изменяются имена и состав методов, их параметры), так и по функциональным признакам (при сохранении внешнего интерфейса меняется логика работы методов). Для проведения модульного тестирования класса после таких изменений потребуется изменение драйвера и, возможно, заглушек. Но только модульного тестирования в данном случае недостаточно, необходимо также проводить и интеграционное тестирование данного класса вместе со всеми классами, которые связаны с ним по данным или по управлению.

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

12.2.3 Подходы к проектированию тестового окружения

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

Первый подход к модульному тестированию основывается на предположении, что функциональность каждого вновь разработанного модуля должна проверяться в автономном режиме без его интеграции с системой. Здесь для каждого вновь разрабатываемого модуля создается тестовый драйвер и заглушки, при помощи которых выполняется набор тестов. Только после устранения всех дефектов в автономном режиме производится интеграция модуля в систему и проводится тестирование на следующем уровне. Достоинством данного подхода является более простая

локализация ошибок в модуле, поскольку при автономном тестировании исключается влияние остальных частей системы, которое может вызывать маскировку дефектов (эффект четного числа ошибок). Основной недостаток данного метода – повышенная трудоемкость написания драйверов и заглушек, поскольку заглушки должны адекватно моделировать поведение системы в различных ситуациях, а драйвер должен не только создавать тестовое окружение, но и имитировать внутреннее состояние системы, в составе которой должен функционировать модуль.

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

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

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

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

12.3 ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Данное занятие предполагает выполнение следующих этапов:

- изучить методические указания;
- выполнить выданное задание;
- ответить на контрольные вопросы.

12.4 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Каковы цель и основные задачи модульного тестирования?
2. Назовите две основные проблемы, возникающие при модульном тестировании.
3. Дайте понятие модуля и его границ.
4. Характеристика процесса тестирования классов как модулей.
5. Определение степени полноты тестирования класса.
6. Протоколирование состояний объектов и их изменений.
7. Тестирование изменений, внесённых в исходный код.
8. Подходы к проектированию тестового окружения.

13 ВЫПОЛНЕНИЕ ФУНКЦИОНАЛЬНОГО ТЕСТИРОВАНИЯ

13.1 ЦЕЛЬ РАБОТЫ

Цель работы – изучение вопросов, связанных с функциональным тестированием.

13.2 ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ

Функциональное тестирование – это проверка программного обеспечения (ПО) на правильность функционирования в идеальных условиях, в отличие от нефункционального, где либо условия неидеальны (нагрузочное тестирование), либо тестируется не правильность функционирования, а что-то иное (тестирование удобства пользования или структура программы).

Есть много приложений, для которых производительность и удобство пользования некритичны. Во всяком случае, часто требования к ПО содержат только функциональную часть. И практически не бывает требований к ПО без функциональной части. Отсюда делаем вывод, что функциональное тестирование проводить нужно для любого ПО.

Как правило, функциональное и нефункциональное тестирование ПО можно проводить параллельно, поэтому обычно это делается разными людьми или командами. В большинстве источников указывается, что функциональное тестирование – это синоним black-box тестирования (при котором программа рассматривается как черный ящик). По степени доступа к коду различают два типа тестирования: тестирование «черного ящика» (black-box), или функциональное тестирование – без доступа к коду, и «белого ящика» (white-box), или структурное тестирование – тестирование кода.

В случае «черного ящика» программа рассматривается как конечный автомат, с входными и выходными данными, набором внутренних состояний и переходов между ними. Тестируется правильность поведения программы при различных входных данных и внутреннем состоянии. Правильность определяется ис-

ходя главным образом из спецификации, а также любыми другими способами, кроме изучения кода (см. лекцию 1).

В случае «белого ящика» тестировщик пишет тест кейсы, основываясь исключительно на коде программы (тесты на правильность кода).

Расширение black-box тестирования, в котором также применяется изучение кода, называется тестированием «серого ящика» (gray-box). В этом случае правильность поведения определяется любым удобным способом, в том числе изучением кода, что позволяет писать более эффективные black-box тесты (то есть тесты на правильность поведения).

Терминология приведена по книге A Practitioner's guide to Sofware Test Design (Lee Copeland).

Методы отбора тестов для black-box тестирования.

Любая программа может рассматриваться как конечный автомат, с входными и выходными данными, набором внутренних состояний и переходов между ними.

Чтобы провести полное тестирование программы, нужно проверить правильность ее поведения при всех возможных комбинациях входных данных и во всех возможных внутренних состояниях. Это невозможно из-за огромного числа комбинаций даже в простейших случаях. Поэтому на практике отбираются только наиболее важные тесты, такой отбор можно производить несколькими методами.

Тестирование сценариев использования – юз-кейсов (use-cases). Чтобы удостовериться в правильности перехода программы между различными внутренними состояниями, в идеале следует протестировать все возможные переходы между каждым из состояний (не только одношаговые, но и многошаговые, то есть все пути в графе состояний). Чтобы уменьшить число тестов, можно проверить только те переходы, которые имеют смысл для пользователя. Use-case – это логически завершенная последовательность действий. Например, открытие файла в Notepad – это use-case, а выбор пункта меню «Открыть файл» в Notepad – это не use-case, а лишь первый шаг юз-кейса «открытие файла». Тестирование сценариев является самым необходимым видом тестирования. Программа должна выполнять операции, для которых она предназначена. Если пользователь может выбрать пункт меню, но

файл не открывается – это очень серьезный баг. Здесь проверяется правильность перехода программы между внутренними состояниями при выполнении определенных операций (т.е. при определенных входных данных)

Тестирование классов эквивалентности. Чтобы удостовериться в правильности поведения программы при различных входных данных, в идеале следует протестировать все возможные значения для каждого элемента этих данных. (а также все возможные сочетания входных параметров).

Например, пусть мы тестируем программу для отдела кадров, в ней есть поле «Возраст соискателя».

Требования по возрасту у нас будут такие:

– 0–13 лет – не нанимать;
– 14–17 лет – можно нанимать на неполный день;
– 18–54 года – можно нанимать на полный день;
– 55–99 лет – не нанимать.

Чтобы проверить все возможные разрешенные данные (только разрешенные!) нам нужно протестировать ввод чисел от 0 до 99. (Возможен ведь еще ввод отрицательных чисел и нечисловых данных.) Так ли необходимо тестировать все числа от 0 до 99? В случае если программа анализирует каждое число по отдельности, вот таким образом, то, видимо, да:

```
...
if (age == 13) hireStatus=«NO»;
if (age == 14) hireStatus=«PART»;
if (age == 15) hireStatus=«PART»;
if (age == 16) hireStatus=«PART»;
if (age == 17) hireStatus=«PART»;
if (age == 18) hireStatus=«FULL»;
...
```

Но к счастью, программы обычно пишут по-другому:

```
if (age >= 0 && age <=13)
        hireStatus=«NO»;
if (age >= 14 && age <=17)
        hireStatus=«PART»;
if (age >= 18 && age <=54)
        hireStatus=«FULL»;
if (age >= 55 && age <=99)
```

hireStatus=«NO»;

Становится очевидным, что можно протестировать одно из чисел каждого диапазона. Например: 5, 15, 20, 60. А также граничные значения (первое и последнее значения из каждого диапазона): 0, 13, 14, 17, 18, 54, 55, 99.

Чтобы уменьшить количество тестируемых значений, производится разбиение множества всех значений входной переменной на подмножества (классы эквивалентности), а затем тестирование одного любого значения из каждого класса.

Все значения из каждого подмножества должны быть эквивалентны для наших тестов. То есть, если тест проходит успешно для одного значения из класса эквивалентности, он должен проходить успешно для всех остальных. И наоборот, если тест не проходит для одного значения, он не должен проходить для всех остальных.

В данном случае имеем 12 классов эквивалентности (каждое из 8 граничных значений по сути является отдельным классом).

Чтобы проверить правильность работы программы на всех разрешенных данных, нужно провести 12 тестов.

Запрещенные данные тестируются аналогично – можно выделить классы эквивалентности «дробное число от 0 до 99», «отрицательное число», «число больше 99», «набор букв», «пустая строка» и т. д.

Таким образом, метод классов эквивалентности можно разделить на три этапа:

1. Тестирование разрешенных значений
2. Тестирование граничных значений
3. Тестирование запрещенных значений

Часто в литературе второй и третий этапы называют отдельными методами, но сути это не меняет.

Попарное тестирование. Метод классов эквивалентности применяется для тестирования каждого входного параметра по отдельности.

Пусть наша программа принимает на вход десяток параметров. Баги, возникающие при определенном сочетании всех десяти параметров, довольно редки. Вообще, взаимное влияние параметров, о котором пользователь не знает – это баг интерфейса (интерфейс интуитивно не понятен).

Чаще всего будут встречаться ситуации, в которых один параметр влияет на один из оставшихся, т. е. самыми частыми будут баги, возникающие при определенном сочетании двух каких-то параметров. Таким образом, можно упростить себе задачу и протестировать все возможные значения для каждой из пар параметров. Такой подход называется попарным тестированием (pairwise testing).

Пример. Пусть имеется 3 двоичных входных параметра (3 чекбокса). Количество всех возможных комбинаций – 2 в степени 3 = 8 , значит, нужно произвести 8 тестов. Давайте попробуем сэкономить, тестируя чекбоксы попарно.

Выпишем все комбинации для первого и второго чекбоксов (таблица 13.1).

Таблица 13.1

1-й	2-й
0	0
0	1
1	0
1	1

Добавим третий столбец так, чтобы во втором и третьем столбце получились все 4 двоичные комбинации. Это можно сделать разными способами, мы сделаем так (на первый столбец можно не обращать внимания) (таблица 13.2).

Таблица 13.2

1-й	2-й	3-й
0	0	0
0	1	0
1	0	1
1	1	1

Итак, с помощью четырех наборов входных данных (четырех тестов) мы протестируем две пары параметров: первый со вторым и второй с третьим. Осталось протестировать пару «первый с третьим».

Выпишем отдельно 1 и 3 столбцы (таблица 13.3).

Таблица 13.3

1-й	3-й
0	0
0	0
1	1
1	1

Как видно, мы имеем здесь две из четырех возможных комбинаций. Комбинации «01» и «10» здесь отсутствуют, а комбинации «00» и «11» присутствуют два раза. Ну что же, добавим еще 2 строки (еще два теста) (таблица 13.4).

Таблица 13.4

1-й	3-й
0	0
0	0
1	1
1	1
0	1
1	0

Вернем второй столбец на его законное место (таблица 13.5)

Таблица 13.5

1-й	2-й	3-й
0	0	0
0	1	0
1	0	1
1	1	1
0		1
1		0

Выходит, что последние два теста можно проходить при любых значениях второго параметра. Можно дописать для определенности нули в эти пустые места (таблица 13.6).

Таблица 13.6

1-й	2-й	3-й
0	0	0
0	1	0
1	0	1
1	1	1
0	0	1
1	0	0

Получаем 6 тестов вместо 8 при полном переборе.

Можно ли сэкономить еще? Оказывается, можно.

Вернемся к 1 шагу (таблица 13.1).

Давайте допишем третий столбец другим способом, поменяв порядок комбинаций (таблица 13.7).

Таблица 13.7

1-й	2-й	3-й
0	0	1
0	1	0
1	0	0
1	1	1

Все комбинации для 1 и 2, а также для 2 и 3 параметра здесь есть. Отлично.

Посмотрим теперь на комбинации 1 и 3 параметра (таблица 13.8)

Таблица 13.8

1-й	3-й
0	1
0	0
1	0
1	1

Изменив порядок значений в третьем столбце, мы одним махом убили двух зайцев: скомбинировали и 2-й с 3-м, и 1-й с 3-м параметры.

Итого имеем всего 4 строки, то есть 4 теста, эквивалентные первоначальным шести (таблица 13.9).

Таблица 13.9

1-й	2-й	3-й
0	0	1
0	1	0
1	0	0
1	1	1

Полный перебор всех комбинаций в третьем столбце гарантированно даст минимальное количество тестов. Однако, судя по тому, что алгоритмы такой минимизации разрабатываются до сих пор, полный перебор неприемлем из-за большого времени исполнения. Существуют программы, дающие приемлемый результат в приемлемое время, например, программа PICT от Microsoft.

Вот строгое определение ортогонального массива:

Ортогональный массив OA (N, k, s, t) – это двумерный массив из N рядов (итераций) и k колонок (факторов) из набора S (т. е. факторы могут принимать любое из s значений), обладающий свойством: выбрав любые t колонок (0<=t<=k) мы получим в рядах все комбинации сочетаний из s по t (Количество повторений одинаковых комбинаций обозначают через λ. Чаще всего рассматривают массивы, где λ = 1, т. е. каждая комбинация встречается только один раз). Параметр t называют мощностью ортогонального массива.

В попарном тестировании применяется ортогональный массив мощности 2 – это двумерный массив такой, что любые 2 колонки этого массива содержат все возможные комбинации (пары) значений, хранящихся в массиве.

13.3 ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Данное занятие предполагает выполнение следующих этапов:

- изучить методические указания;
- выполнить выданное задание;
- ответить на контрольные вопросы.

13.4 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что понимается под функциональным тестированием?

2. В чём сущность тестирования «черного ящика» (black-box)?
3. В чём сущность тестирования «белого ящика» (white-box)?
4. В чём сущность тестирования «серого ящика» (gray-box)?
5. Методы отбора тестов для black-box тестирования.
6. Тестирование сценариев использования – юз-кейсов (use-cases).
7. Тестирование классов эквивалентности.
8. В чём сущность попарного тестирования?

14 ТЕСТИРОВАНИЕ ИНТЕГРАЦИИ

14.1 ЦЕЛЬ РАБОТЫ

Цель работы – изучение вопросов, связанных с интеграционным тестированием.

14.2 ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ

Исходя из различий между модульным тестированием и системным тестированием, интеграционное тестирование является переходным этапом между представлением программы в виде отдельных модулей в вид полностью функциональной системы.

Интеграционное тестирование – вид тестирования, при котором на соответствие требований проверяется интеграция модулей, их взаимодействие между собой, а также интеграция подсистем в одну общую систему. Для интеграционного тестирования используются компоненты, уже проверенные с помощью модульного тестирования, которые группируются в множества. Данные множества проверяются в соответствии с планом тестирования, составленным для них, а объединяются они через свои интерфейсы.

Так как модули соединяются между собой с помощью предусмотренных реализацией интерфейсов и в процессе тестирования у нас нет потребности рассматривать внутреннюю структуру компонентов, можно утверждать, что интеграционное тестирование выполняется методом «черного ящика».

Существует несколько подходов к интеграционному тестированию:

– Снизу вверх. Сначала собираются и тестируются модули самих нижних уровней, а затем по возрастанию к вершине иерархии. Данный подход требует готовности всех собираемых модулей на всех уровнях системы.

– Сверху вниз. Данный подход предусматривает движение с высокоуровневых модулей, а затем направляется вниз. При этом используются заглушки для тех модулей, которые находятся ниже по уровню, но включение которых в тест еще не произошло.

– Большой взрыв. Все модули всех уровней собираются воедино, а затем тестируется. Данный метод экономит время, но требует тщательной проработки тест кейсов.

При автоматизации тестирования используется Система непрерывной интеграции. Принцип ее действия заключается в следующем:

1) Система непрерывной интеграции производит мониторинг системы контроля версий.

2) При изменении исходных кодов в репозитории производится обновление локального хранилища.

3) Выполняются необходимые проверки и модульные тесты.

4) Исходные коды компилируются в готовые выполняемые модули.

5) Выполняются тесты интеграционного уровня.

6) Генерируется отчет о тестировании.

Это позволяет тестировать систему сразу после внесения изменений, что существенно сокращает время обнаружения и исправления ошибок.

Интеграционные тесты проверяют, чтобы различные части приложения корректно работали вместе. В отличии от «модульного тестирования» интеграционные тесты часто запрашивают компоненты инфраструктуры приложения, например, БД, файловую систему, веб-запросы и ответы и так далее. На месте этих компонентов юнит-тесты используют mock-объекты, а интеграционные тесты должны проверять, хорошо ли эти компоненты работают в системе.

Поскольку интеграционные тесты работают с большими сегментами кода и поскольку им нужны элементы инфраструктуры, они медленнее, чем юнит-тесты. Поэтому вам не стоит использовать слишком много интеграционных тестов, особенно если вы можете проверить некоторый функционал с помощью юнит-теста.

Если одно и то же поведение можно протестировать с помощью юнит-теста и интеграционного теста, выберите юнит-тест, поскольку он фактически всегда будет быстрее. У вас могут быть сотни юнит-тестов, но всего лишь несколько интеграционных тестов, покрывающих наиболее важные сценарии.

Тестирование логики внутри методов обычно является задачей юнит-тестов. А интеграционные тесты входят в игру тогда, когда надо проверить, как приложение работает во фреймворке (например, ASP.NET Core) или с БД. Вам не нужно создавать слишком много интеграционных тестов, чтобы проверить, что вы можете добавить запись или извлечь ее из БД. Вам не нужно тестировать каждое возможное изменение в коде для доступа к данным – вы просто должны убедиться, что приложение работает должным образом.

Интеграционное тестирование в ASP.NET Core.

Чтобы настроить интеграционные тесты, вам надо создать тестовый проект, связанный с вашим ASP.NET Core веб-проектом, и установить механизм запуска тестов. Этот процесс описан в документации Unit-testing, наряду с более подробными инструкциями по запуску тестов и рекомендациями по именованию тестов и тестовых классов.

Юнит-тесты и интеграционные тесты стоит хранить в разных проектах. Тогда вы случайно не включите элементы инфраструктуры в юнит-тесты, и тогда вы сможете запускать либо все тесты, либо какой-то определенный набор тестов.

Тестовый хост. ASP.NET Core включает тестовый хост, который может быть добавлен в проекты интеграционных тестов и используется для хостинга ASP.NET Core приложений, обрабатывая тестовые запросы без необходимости реального веб-хостинга. В данный пример мы включили проект интеграционных тестов, который будет использовать xUnit и тестовый хост, как вы видите ниже:

```
"dependencies": {
  "PrimeWeb": "1.0.0",
  "xunit": "2.1.0",
  "dotnet-test-xunit": "1.0.0-rc2-build10025",
  "Microsoft.AspNetCore.TestHost": "1.0.0"
}
```

После того как пакет Microsoft.AspNet.TestHost будет включен в проект, вы сможете создать и настроить TestServer. Следующий тест показывает, как проверить, то запрос, отправленный к корневой директории сайта, возвращает “Hello, World!”, и

этот тест должен пройти успешно для шаблона по умолчанию ASP.NET Core Empty.

```
private readonly TestServer _server;
private readonly HttpClient _client;
public PrimeWebDefaultRequestShould()
{
   // Arrange
   _server = new TestServer(new WebHostBuilder()
      .UseStartup<Startup>());
   _client = _server.CreateClient();
}

[Fact]
public async Task ReturnHelloWorld()
{
   // Act
   var response = await _client.GetAsync("/");
   response.EnsureSuccessStatusCode();

   var responseString = await response.Content.ReadAsStringAsync();

   // Assert
   Assert.Equal("Hello World!",
      responseString);
}
```

14.3 ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Данное занятие предполагает выполнение следующих этапов:

– изучить методические указания;
– выполнить выданное задание;
– ответить на контрольные вопросы.

14.4 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что понимается под интеграционным тестированием?
2. Дайте характеристику подходам к интеграционному тестированию: снизу вверх; сверху вниз; большой взрыв.
3. В чём заключается принцип действия системы непрерывной интеграции?
4. В чём заключается интеграционное тестирование ASP.NET Core?

15 ДОКУМЕНТИРОВАНИЕ РЕЗУЛЬТАТОВ ТЕСТИРОВАНИЯ

15.1 ЦЕЛЬ РАБОТЫ

Цель работы – изучить правила и последовательность составления итогового отчета о результатах тестирования web-приложения.

15.2 ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ

Итоговый отчет можно разделить на части с соответствующей информацией:

- Общая информация (Common Information).
- Тестовое окружение (Test Platform).
- Рекомендации QA (QA Recommendations).
- Детализированная информация (Detailed Information).
- Окончание содержимого.

Общая информация (Common Information). В данной части отчета описывается, какие виды тестов проводились. Зачастую указываются модули, которые тестировались или функционал. Стоит удостовериться, не забыта ли какая-то часть функционала, особенно это актуально, когда нужно собрать итоговый отчет, соединив в себе данные о разных видах тестов и функционале.

Тестовое окружение (Test Platform). Как правило, в этой части указываются: название проекта; номер сборки; ссылка на проект (сборку). Необходимо убедиться, что зайдя по ссылке вы действительно попадаете на проект или можете установить приложение.

При указании данных в этой части отчета нужно быть очень внимательным, т. к. неправильная ссылка на сборку или неверный номер сборки не дают достоверной информации всем заинтересованным людям, а также затрудняют работу человеку, собирающему финальный отчет.

Рекомендации QA (QA Recommendations). Данная часть отчета является наиболее важной, т. к. здесь отражается общее состояние сборки. Здесь показывается аналитическая работа тестировщика, его рекомендации по улучшению функционала, наибо-

лее слабые места и наиболее критичные дефекты, динамика изменения качества проекта.

В этом разделе должна быть информация о следующем:

– Указан функционал (часть функционала), который заблокирован для проверки. Даны пояснения, почему этот функционал не проверен (указаны наиболее критичные дефекты).

– Произведен анализ качества проверенного функционала. Следует указать, улучшилось оно или ухудшилось по сравнению с предыдущей версией, какое качество на сегодняшний момент, какие факторы повлияли на выставление именно такого качества сборки.

– Если качество сборки ухудшилось, то обязательно должны быть указаны регрессионные места.

– Наиболее нестабильные части функционала следует выделить и указать причину, по которой они таковыми являются.

– Даны рекомендации по тому функционалу и дефектам, скорейшее исправление которых является наиболее приоритетным.

– Список наиболее критичных для сборки дефектов, с указанием названия и их критичности.

– Для отчета уровня Smoke обязательно указать весь нестабильный функционал. Если сборка является релизной или предрелизной, то любое ухудшение качества является критичным и важно об этом сообщить менеджеру как можно раньше. Помимо всего вышеуказанного для релизных и предрелизных сборок в отчете о качестве продукта важно указывать следующее:

– Дана информация о всех проблемах, характерных сборке. Проведен анализ, насколько оставшиеся проблемы являются критичными для конечного пользователя.

– Указаны дефекты, которые следует исправить, чтобы качество конечной сборки было выше.

Детализированная информация (Detailed Information). В данной части отчета описывается более подробная информация о проверенных частях функционала, устанавливается качество каждой проверенной части функционала (модуля) в отдельности. В зависимости от типа проводимых тестов, эта часть отчета будет отличаться.

Smoke. При оценке качества функционала на уровне Smoke теста, оно может быть либо Приемлемым, либо Неприемлемым. Качество сборки зависит от нескольких факторов:

– Если это релизная или предрелизная сборка, то для выставления Приемлемого качества на уровне Smoke не должно быть найдено функциональных дефектов.

– Наличие нового функционала. Новый функционал, который впервые поставляется на тестирование, не должен содержать дефектов уровня Smoke для выставления Приемлемого качества всей сборки.

– Чтобы установить сборке Приемлемое качество, не должно быть дефектов уровня Smoke у того функционала, по которому планируется проводить полные тесты.

– Все наиболее важные части функционала отрабатывают корректно, тогда качество всего функционала на уровне Smoke может быть оценено, как приемлемое.

В части о детализированной информации качества сборки следует более подробно описать проблемы, которые были найдены во время теста.

DV. В этой части отчета указывается качество о проведении валидации дефектов.

Здесь должна быть следующая информация:

– Общее количество всех дефектов, поступивших на проверку.

– Количество неисправленных дефектов и их процент от общего количества.

– Список дефектов, которые не были проверены и причины, по которым этого не было сделано.

– Наглядная таблица с неисправленными дефектами.

По вышеуказанным результатам выставляется качество теста. Если процент неисправленных дефектов < 10%, то качество – приемлемое, если > 10%, то качество – неприемлемое.

NFT. При проведении полного теста нового функционала качество отдельно проверенного функционала может быть: высокое, среднее, низкое.

В отчете следует отдельно указывать информацию о качестве каждой части нового функционала. В этой части отчета должна быть следующая информация:

– Дана общая оценка реализации нового функционала (сгруппированная по качеству).

– Подробная (детальная) информация о качестве каждой из частей новой функциональности.

– Проведен анализ каждой из новых функций в отдельности.

– Даны ясные пояснения о выставлении соответствующего качества.

– Даны рекомендации по улучшению качества (какие проблемы следует исправить).

– Показана таблица с новыми функциями (название), их качеством, статусом функции из CQ.

AT, MAT, Regression. Если проводились тесты указанных уровней, то в первую очередь при написании отчета нужно анализировать динамику изменения качества проверенной функциональности в сравнении с более ранними версиями сборки.

Также как и у предыдущего вида тестов, качество этих может быть: высокое, среднее, низкое.

Для указанных видов тестов в данной части отчета должна быть описана информация следующего характера:

– Дана сравнительная характеристика каждой из частей функционала в сравнении с предыдущими версиями сборки.

– Подробная (детальная) информация о качестве каждой из частей проверенной функциональности.

– Даны ясные пояснения о выставлении соответствующего качества каждой функции в отдельности.

– Даны рекомендации по улучшению качества (какие проблемы следует исправить).

В завершении содержимое отчета должно включать в себя информацию следующего характера:

– Ссылка на тест-план.

– Ссылка на документ feature matrix (если таковой имеется).

– Ссылка на документ со статистикой (если таковой имеется).

– Общее количество всех новых дефектов.

– Подпись высылающего отчет.

Данные ссылки должны быть корректными, необходимо проверить достоверную ли информацию получает пользователь, открывший ссылку. Следует обращать особое внимание на подпись, удостоверьтесь, что указана именно ваша подпись либо какая-то универсальная для определенного проекта подпись.

15.3 ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Данное занятие предполагает выполнение следующих этапов:

- изучить методические указания;
- выполнить выданное задание;
- ответить на контрольные вопросы.

15.4 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какая структура итогового отчета о результатах тестирования?
2. Что содержится в разделе Общая информация?
3. Что содержится в разделе Тестовое окружение?
4. Что содержится в разделе Рекомендации QA?
5. Что содержится в разделе Детализированная информация?
6. Что содержится в окончании отчета?