Приложение 3.2.1 к ОПОП по специальности 09.02.07
Информационные системы и программирование

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Магнитогорский государственный технический университет им. Г.И. Носова»

Многопрофильный колледж

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ЛАБОРАТОРНЫХ ЗАНЯТИЙ МЕЖДИСЦИПЛИНАРНОГО КУРСА

МДК.02.01 Технология разработки программного обеспечения

для обучающихся специальности

09.02.07 Информационные системы и программирование

Магнитогорск, 2024

ОДОБРЕНО

Предметно-цикловой комиссией «Информатики и вычислительной техники»
Председатель Т.Б.Ремез
Протокол № 5 от «31» 01 2024

Методической комиссией МпК
Протокол № 3 от «21»02 2024

Разработчик (и):
преподаватель отделения №2 "Информационных технологий и транспорта" Многопрофильного колледжа ФГБОУ ВО «МГТУ им. Г.И. Носова» Оксана Викторовна Кобыльская

Методические указания по выполнению лабораторных работ разработаны на основе рабочей программы профессионального модуля ПМ.02 «Осуществление интеграции программных модулей», МДК.02.01 Технология разработки программного обеспечения

Содержание лабораторных работ ориентировано на подготовку обучающихся к освоению профессионального модуля программы подготовки специалистов среднего звена по специальности 09.02.07 Информационные системы и программирование и овладению профессиональными компетенциями

СОДЕРЖАНИЕ

1 ВВЕДЕНИЕ

Важную часть теоретической и профессиональной практической подготовки обучающихся составляют лабораторные занятия.

Состав и содержание лабораторных занятий направлены на реализацию Федерального государственного образовательного стандарта среднего профессионального образования.

Ведущей дидактической целью лабораторных занятий является экспериментальное подтверждение и проверка существенных теоретических положений (законов, зависимостей).

В соответствии с рабочей программой ПМ.02 Осуществление интеграции программных модулей, МДК.02.02 Инструментальные средства разработки программного обеспечения предусмотрено проведение лабораторных занятий. В рамках лабораторного занятия обучающиеся могут выполнять одну или несколько лабораторных работ.

В результате их выполнения, обучающийся должен: ***уметь:***

- У1 использовать выбранную систему контроля версий;
- У2 использовать методы для получения кода с заданной функциональностью и степенью качества;
- У3 анализировать проектную и техническую документацию;
- У4 использовать специализированные графические средства построения и анализа архитектуры программных продуктов;
- У5 организовывать заданную интеграцию модулей в программные средства на базе имеющейся архитектуры и автоматизации бизнес-процессов;
- У6 определять источники и приемники данных;
- У7 использовать приемы работы в системах контроля версий;
- У8 выполнять отладку, используя методы и инструменты условной компиляции (классы Debug и Trace);
- У9 оценивать размер минимального набора тестов;
- У10 разрабатывать тестовые пакеты и тестовые сценарии;
- У11 выявлять ошибки в системных компонентах на основе спецификаций;
- У12 использовать различные транспортные протоколы и стандарты форматирования сообщений;
- У13 выполнять тестирование интеграции;
- У14 организовывать постобработку данных;
- У15 создавать классы- исключения на основе базовых классов;
- У16 выполнять ручное и автоматизированное тестирование программного модуля;
- У17 использовать инструментальные средства отладки программных продуктов;
- Уо 01.01 распознавать задачу и/или проблему в профессиональном и/или социальном контексте;
- Уо 01.02 анализировать задачу и/или проблему и выделять еѐ составные части;
- Уо 01.03 определять этапы решения задачи;
- Уо 01.04 выявлять и эффективно искать информацию, необходимую для решения задачи и/или проблемы;
- Уо 01.05 составлять план действий;
- Уо 01.06 определять необходимые ресурсы;
- Уо 01.07 владеть актуальными методами работы в профессиональной и смежных сферах;
- Уо 01.08 реализовывать составленный план;
- Уо 01.09 оценивать результат и последствия своих действий (самостоятельно или с помощью наставника);
- Уо 01.10 учитывать временные ограничения и сроки при решении профессиональных задач;
- Уо 02.01 определять задачи для поиска информации;
- Уо 02.02 определять необходимые источники информации;

- Уо 02.03 планировать процесс поиска; структурировать получаемую информацию;
- Уо 02.04 выделять наиболее значимое в перечне информации;
- Уо 02.05 оценивать практическую значимость результатов поиска;
- Уо 02.06 оформлять результаты поиска, применять средства информационных технологий для решения профессиональных задач;
- Уо 02.07 использовать современное программное обеспечение;
- Уо 02.08 использовать различные цифровые средства для решения профессиональных задач;
- Уо 02.09 проявлять культуру информационной безопасности при использовании информационно-коммуникационных технологий;
- Уо 03.01 определять актуальность нормативно-правовой документации в профессиональной деятельности;
- Уо 03.02 применять современную научную профессиональную терминологию;
- Уо 03.03 определять и выстраивать траектории профессионального развития и самообразования;
- Уо 03.10 применять исследовательские приемы и навыки, чтобы быть в курсе последних отраслевых решений
- Уо 04.01 организовывать работу коллектива и команды;
- Уо 04.02 взаимодействовать с коллегами, руководством, клиентами в ходе профессиональной деятельности;
- Уо 04.03 эффективно работать в команде;
- Уо 04.04 использовать навыки управления проектами в распределении ресурсов и формировании графика выполнения задач;
- Уо 05.01 грамотно излагать свои мысли и оформлять документы по профессиональной тематике на государственном языке;
- Уо 05.02 проявлять толерантность в рабочем коллективе;
- Уо 05.03 применять техники и приемы эффективного общения в профессиональной деятельности
- Уо 09.01 понимать общий смысл четко произнесенных высказываний на известные темы (профессиональные и бытовые), понимать тексты на базовые профессиональные темы;
- Уо 09.02 участвовать в диалогах на знакомые общие и профессиональные темы;
- Уо 09.03 строить простые высказывания о себе и о своей профессиональной деятельности;
- Уо 09.04 кратко обосновывать и объяснять свои действия (текущие и планируемые);
- Уо 09.05 писать простые связные сообщения на знакомые или интересующие профессиональные темы
- Уо 09.06 читать, понимать и находить необходимые технические данные и инструкции в руководствах в любом доступном формате.

Содержание практических и лабораторных занятий ориентировано на подготовку обучающихся к освоению профессионального модуля программы подготовки специалистов среднего звена по специальности и овладению ***профессиональными компетенциями:***

ПК.2.1 Разрабатывать требования к программным модулям на основе анализа проектной и технической документации на предмет взаимодействия компонент;

ПК.2.2 Выполнять интеграцию модулей в программное обеспечение;

ПК.2.3 Выполнять отладку программного модуля с использованием специализированных программных средств;

ПК.2.4 Осуществлять разработку тестовых наборов и тестовых сценариев для программного обеспечения;

ПК.2.5 Производить инспектирование компонент программного обеспечения на предмет соответствия стандартам кодирования.

А также формированию ***общих компетенций***:

ОК 01 Выбирать способы решения задач профессиональной деятельности применительно к различным контекстам;

ОК 02 Использовать современные средства поиска, анализа и интерпретации информации и информационные технологии для выполнения задач профессиональной деятельности;

ОК 03 Планировать и реализовывать собственное профессиональное и личностное развитие, предпринимательскую деятельность в профессиональной сфере, использовать знания по финансовой грамотности в различных жизненных ситуациях;

ОК 04 Эффективно взаимодействовать и работать в коллективе и команде;

ОК 05 Осуществлять устную и письменную коммуникацию на государственном языке Российской Федерации с учетом особенностей социального и культурного контекста;

ОК 09 Пользоваться профессиональной документацией на государственном и иностранном языках.

Выполнение обучающимися лабораторных работ по ПМ.02 Осуществление интеграции программных модулей, МДК.02.01 Технология разработки программного обеспечения направлено на:

- обобщение, систематизацию, углубление, закрепление, развитие и детализацию полученных теоретических знаний по конкретным темам учебной дисциплины;
- формирование умений применять полученные знания на практике, реализацию единства интеллектуальной и практической деятельности;
- развитие интеллектуальных умений у будущих специалистов: аналитических, проектировочных, конструктивных и др.;
- выработку при решении поставленных задач профессионально значимых качеств, таких как самостоятельность, ответственность, точность, творческая инициатива.

Лабораторные занятия проводятся в рамках соответствующей темы, после освоения дидактических единиц, которые обеспечивают наличие знаний, необходимых для ее выполнения.

2 МЕТОДИЧЕСКИЕ УКАЗАНИЯ

Тема 2.1.2 Описание и анализ требований к программному обеспечению

Лабораторное занятие № 1
Анализ предметной области. Построение диаграммы вариантов использования

Цель: Проанализировать и описать заданную предметную область, сформировать требования к информационной системе, распределить роли в группе разработчиков.

Выполнив задания, Вы будете:

уметь:

– У3 анализировать проектную и техническую документацию;

– У4 использовать специализированные графические средства построения и анализа архитектуры программных продуктов;

– У11 выявлять ошибки в системных компонентах на основе спецификаций.

Материальное обеспечение:

Пакет Microsoft Office, доступ к Internet ресурсам.

Задание:

Лабораторное занятие направлено на ознакомление с процессом описания информационной системы и получение навыков по использованию основных методов анализа ИС.

Рекомендуемые предметные области для разработки информационной системы:

1. Разработка программного комплекса «Управление кредитами в банке».
2. Разработка программного комплекса «Отделение колледжа».
3. Разработка программного комплекса «Обслуживание банкомата».
4. Разработка программного комплекса «Управление гостиницей».
5. Разработка программного комплекса «Магазин по продаже автозапчастей».
6. Разработка программного комплекса «Строительная фирма».
7. Разработка программного комплекса «Управление библиотечным фондом».
8. Разработка программного комплекса «АРМ работника склада»
9. Разработка программного комплекса «АРМ администратора ателье по ремонту оргтехники»
10. Разработка программного комплекса «АРМ администратора автосалона».
11. Разработка программного комплекса «АРМ администратора ресторана».
12. Разработка программного комплекса «АРМ администратора фитнес-клуба».
13. Разработка программного комплекса «АРМ администратора аэропорта».
14. Разработка программного комплекса «АРМ работника отдела кадров».
15. Разработка программного комплекса «АРМ администратора спорткомплекса».

Вы также можете предложить преподавателю свой вариант предметной области.

Требования к результатам выполнения лабораторной работы:

1. наличие описания информационной системы;
2. проведение анализа осуществимости выполнения проекта;
3. наличие заключения о возможности реализации проекта.

В первую очередь необходимо определить источники информации. Это могут быть менеджеры отделов, где система будет использоваться, разработчики программного обеспечения, знакомые с типом будущей системы, технологи, конечные пользователи и т.д.

В ходе анализа осуществимости создания информационной системы целесообразно ответить на вопросы:

- Что произойдет с организацией, если система не будет введена в эксплуатацию?

- Какие текущие проблемы существуют в организации и как новая система поможет их решить?
- Каким образом система будет способствовать целям бизнеса?
- Требует ли разработка системы технологии, которая до этого не использовалась в организации?

Результатом анализа осуществимости создания информационной системы является заключение о возможности реализации проекта. В нем даются рекомендации относительно начала или продолжения разработки системы. Могут быть предложены ориентировочные суммы бюджета или его изменения, графика работ по созданию системы или предъявлены более высокие требования к системе.

Документ, описывающий предметную область, должен содержать рекомендации относительно разработки системы, базовые предложения по объѐму требуемого бюджета и времени, графику работ, числу разработчиков, требуемому программному обеспечению, и может включать следующие разделы:

- Введение
- Организация выполнения проекта
- Анализ рисков и др.

После обработки собранной информации готовится отчет по анализу осуществимости создания системы, а затем выполнить построение диаграммы вариантов использования (она же диаграмма прецедентов, диаграмма Use Case).

Диаграммы вариантов использования

Понятие варианта использования (use case) впервые ввел Ивар Якобсон и придал ему такую значимость, что в настоящее время вариант использования превратился в основной элемент разработки и планирования проекта.

Вариант использования представляет собой последовательность действий (транзакций), выполняемых системой в ответ на событие, инициируемое некоторым внешним объектом (действующим лицом). Вариант использования описывает типичное взаимодействие между пользователем и системой. В простейшем случае вариант использования определяется в процессе обсуждения с пользователем тех функций, которые он хотел бы реализовать. На языке UML вариант использования изображают следующим образом:

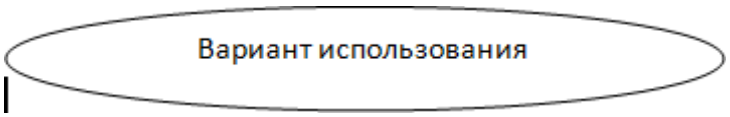


Рисунок 1 – Вариант использования

Действующее лицо (actor) – это роль, которую пользователь играет по отношению к системе. Действующие лица представляют собой роли, а не конкретных людей или наименования работ. Несмотря на то, что на диаграммах вариантов использования они изображаются в виде стилизованных человеческих фигурок, действующее лицо может также быть внешней системой, которой необходима некоторая информация от данной системы. Показывать на диаграмме действующих лиц следует только в том случае, когда им действительно необходимы некоторые варианты использования. На языке UML действующие лица представляют в виде фигур:

Рисунок 2 – Действующее лицо (актер)

Действующие лица делятся на три основных типа:

- пользователи;
- системы;
- другие системы, взаимодействующие с данной;
- время.

Время становится действующим лицом, если от него зависит запуск каких-либо событий в системе.

Связи между вариантами использования и действующими лицами

В языке UML на диаграммах вариантов использования поддерживается несколько типов связей между элементами диаграммы. Это связи коммуникации (communication), включения (include), расширения (extend) и обобщения (generalization).

Связь коммуникации – это связь между вариантом использования и действующим лицом. На языке UML связи коммуникации показывают с помощью однонаправленной ассоциации (сплошной линии).

Рисунок 3– Пример связи коммуникации

Связь включения применяется в тех ситуациях, когда имеется какой-либо фрагмент поведения системы, который повторяется более чем в одном варианте использования. С помощью таких связей обычно моделируют многократно используемую функциональность.

Связь расширения применяется при описании изменений в нормальном поведении системы. Она позволяет варианту использования только при необходимости использовать функциональные возможности другого.

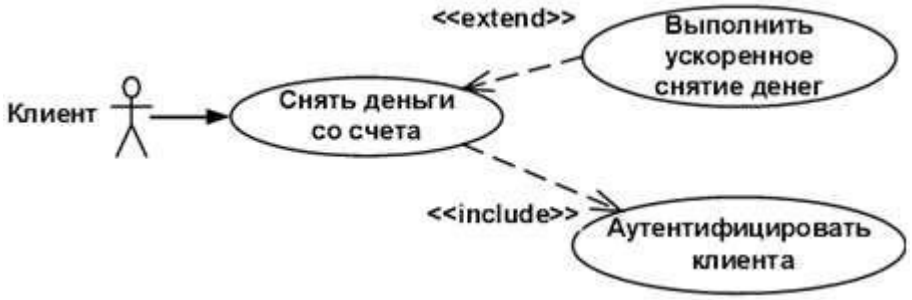


Рисунок 4 – Пример связи включения и расширения

С помощью связи обобщения показывают, что у нескольких действующих лиц имеются общие черты.

Рисунок 5 – Пример связи обобщения

Порядок выполнения работы:

1. Изучить теоретический материал по теме.
2. Составить подробное описание информационной системы.
3. На основании описания системы провести анализ осуществимости создания информационной системы..
4. Распределить роли в группе (руководитель проекта, системный аналитик, разработчик, разработчик, тестер-разработчик).

5. Составить документ, описывающий предметную область.
6. Составить отчет о проделанной работе.

Ход работы:

Краткие теоретические сведения

Общие сведения о разработке программного обеспечения

Проблемы управления программными проектами впервые проявились в 60-х - начале 70-х годов, когда провалились многие большие проекты по разработке программных продуктов. Были зафиксированы задержки в создании ПО, оно было ненадежным, затраты на разработку в несколько раз превосходили первоначальные оценки, созданные программные системы часто имели низкие показатели производительности. Причины провалов коренились в тех подходах, которые использовались в управлении проектами. Применяемая методика была основана на опыте управления техническими проектами и оказалась неэффективной при разработке программного обеспечения.

Важно понимать разницу между профессиональной разработкой ПО и любительским программированием. Необходимость управления программными проектами вытекает из того факта, что процесс создания профессионального ПО всегда является субъектом бюджетной политики организации, где оно разрабатывается, и имеет временные ограничения. Работа руководителя программного проекта по большому счету заключается в том, чтобы гарантировать выполнение этих бюджетных и временных ограничений с учетом бизнес-целей организации относительно разрабатываемого ПО.

Руководители проектов призваны спланировать все этапы разработки программного продукта. Они также должны контролировать ход выполнения работ и соблюдения всех требуемых стандартов. Постоянный контроль за ходом выполнения работ необходим для того, чтобы процесс разработки не выходил за временные и бюджетные ограничения. Хорошее управление не гарантирует успешного завершения проекта, но плохое управление обязательно приведет к его провалу. Это может выразиться в задержке сроков сдачи готового ПО, в превышении сметной стоимости проекта и в несоответствии готового ПО спецификации требований.

Процесс разработки ПО существенно отличается от процессов реализации технических проектов, что порождает определенные сложности в управлении программными проектами:

1. *Программный продукт нематериален.* Программное обеспечение нематериально, его нельзя увидеть или потрогать. Руководитель программного проекта не видит процесс "роста" разрабатываемого ПО. Он может полагаться только на документацию, которая фиксирует процесс разработки программного продукта.
2. *Не существует стандартных процессов разработки ПО.* На сегодняшний день не существует четкой зависимости между процессом создания ПО и типом создаваемого программного продукта. Другие технические дисциплины имеют длительную историю, процессы разработки технических изделий многократно опробованы и проверены. Процессы создания большинства технических систем хорошо изучены. Изучением же процессов создания ПО специалисты занимаются только последнее время. Поэтому пока нельзя точно предсказать, на каком этапе процесса разработки ПО могут возникнуть проблемы, угрожающие всему программному проекту.
3. *Большие программные проекты - это часто "одноразовые" проекты.* Большие программные проекты, как правило, значительно отличаются от проектов, реализованных ранее. Поэтому, чтобы уменьшить неопределенность в планировании проекта, руководители проектов должны обладать очень большим практическим опытом. Но постоянные технологические изменения в компьютерной технике и коммуникационном оборудовании обесценивают предыдущий опыт. Знания и навыки, накопленные опытом, могут не востребоваться в новом проекте.

Перечисленные отличия могут привести к тому, что реализация проекта выйдет из временного графика или превысит бюджетные ассигнования. Программные системы зачастую

оказываются новинками как в "идеологическом", так и в техническом плане. Поэтому, предвидя возможные проблемы в реализации программного проекта, следует всегда помнить, что многим из них свойственно выходить за рамки временных и бюджетных ограничений.

Процесс управления разработкой программного обеспечения

Невозможно описать и стандартизировать все работы, выполняемые в проекте по созданию ПО. Эти работы весьма существенно зависят от организации, где выполняется разработка ПО, и от типа создаваемого программного продукта. Но всегда можно выделить следующие:

- Написание предложений по созданию ПО.
- Планирование и составление графика работ по созданию ПО.
- Оценивание стоимости проекта.
- Подбор персонала.
- Контроль за ходом выполнения работ.
- Написание отчетов и представлений.

Первая стадия программного проекта может состоять из написания предложений по реализации этого проекта. Предложения должны содержать описание целей проектов и способов их достижения. Они также обычно включают в себя оценки финансовых и временных затрат на выполнение проекта. При необходимости здесь могут приводиться обоснования для передачи проекта на выполнение сторонней организации или команде разработчиков.

Написание предложений — очень ответственная работа, так как для многих организаций вопрос о том, будет ли проект выполняться самой организацией или разрабатываться по контракту сторонней компанией, является критическим. Не существует каких-либо рекомендаций по написанию предложений, многое здесь зависит от опыт.

На этапе планирования проекта определяются процессы, этапы и полученные на каждом из них результаты, которые должны привести к выполнению проекта. Реализация этого плана приведет к достижению целей проекта. Определение стоимости проекта напрямую связано с его планированием, поскольку здесь оцениваются ресурсы, требующиеся для выполнения плана.

Контроль за ходом выполнения работ (мониторинг проекта) — это непрерывный процесс, продолжающийся в течение всего срока реализации проекта. Руководитель должен постоянно отслеживать ход реализации проекта и сравнивать фактические и плановые показатели выполнения работ с их стоимостью. Хотя многие организации имеют механизмы формального мониторинга работ, опытный руководитель может составить ясную картину о стадии развитии проекта просто путем неформального общения с разработчиками.

Неформальный мониторинг часто помогает обнаружить потенциальные проблемы, которые в явном виде могут обнаружиться позднее. Например, ежедневное обсуждение хода выполнения работ может выявить отдельные недоработки в создаваемом программном продукте. Вместо ожидания отчетов, в которых будет отражен факт "пробуксовки" графика работ, можно обсудить со специалистами намечающиеся программистские проблемы и не допустить срыва графика работ.

В течение реализации проекта обычно происходит несколько формальных контрольных проверок хода выполнения работ по созданию ПО. Такие проверки должны дать общую картину хода реализации проекта в целом и показать, насколько уже разработанная часть ПО соответствует целям проекта.

Время выполнения больших программных проектов может занимать несколько лет. В течение этого времени цели и намерения организации, заказавшей программный проект, могут существенно измениться. Может оказаться, что разрабатываемый программный продукт стал уже ненужным либо исходные требования к создаваемому ПО просто устарели и их необходимо кардинально менять. В такой ситуации руководство организации-разработчика может принять решение о прекращении разработки ПО или об изменении проекта в целом с тем, чтобы учесть изменившиеся цели и намерения организации-заказчика.

Руководители проектов обычно обязаны сами подбирать исполнителей для своих проектов. В идеальном случае профессиональный уровень исполнителей должен соответствовать той работе, которую они будут выполнять в ходе реализации проекта. Однако во многих случаях

руководители должны полагаться на команду разработчиков, которая далека от идеальной. Такая ситуация может быть вызвана следующими причинами:

1. Бюджет проекта не позволяет привлечь высококвалифицированный персонал. В таком случае за меньшую плату привлекаются менее квалифицированные специалисты.
2. Бывают ситуации, когда невозможно найти специалистов необходимой квалификации как в самой организации-разработчике, так и вне ее. Например, в организации "лучшие люди" могут быть уже заняты в других проектах.
3. Организация хочет повысить профессиональный уровень своих работников. В этом случае она может привлечь к участию в проекте неопытных или недостаточно квалифицированных работников, чтобы они приобрели необходимый опыт и поучились у более опытных специалистов.

Таким образом, почти всегда подбор специалистов для выполнения проекта имеет определенные ограничения и не является свободным. Вместе с тем необходимо, чтобы хотя бы несколько членов группы разработчиков имели квалификацию и опыт, достаточные для работы над данным проектом. В противном случае невозможно избежать ошибок в разработке ПО.

Руководитель проекта обычно обязан посылать отчеты о ходе его выполнения как заказчику, так и подрядным организациям. Это должны быть краткие документы, основанные на информации, извлекаемой из подробных' отчетов о проекте. В этих отчетах должна быть та информация, которая позволяет четко оценить степень готовности создаваемого программного продукта.

В рамках курса «Технология разработки программного обеспечения» выделены следующие роли в группе по разработке ПО:

- Руководитель – общее руководство проектом, написание документации, общение с заказчиком ПО
- Системный аналитик – разработка требований (составление технического задания, проекта программного обеспечения)
- Тестер – составление плана тестирования и аттестации готового ПО (продукта), составление сценария тестирования, базовый пример, проведение мероприятий по плану тестирования
- Разработчик – моделирование компонент программного обеспечения, кодирование

Планирование проекта разработки программного обеспечения

Эффективное управление программным проектом напрямую зависит от правильного планирования работ, необходимых для его выполнения. План помогает руководителю предвидеть проблемы, которые могут возникнуть на каких-либо этапах создания ПО, и разработать превентивные меры для их предупреждения или решения. План, разработанный на начальном этапе проекта, рассматривается всеми его участниками как руководящий документ, выполнение которого должно привести к успешному завершению проекта. Этот первоначальный план должен максимально подробно описывать все этапы реализации проекта.

Процесс планирования начинается, исходя из описания системы, с определения проектных ограничений (временные ограничения, возможности наличного персонала, бюджетные ограничения и т.д.). Эти ограничения должны определяться параллельно с оцениванием проектных параметров, таких как структура и размер проекта, а также распределением функций среди исполнителей. Затем определяются этапы разработки и то, какие результаты документация, прототипы, подсистемы или версии программного продукта) должны быть получены по окончании этих этапов. Далее начинается циклическая часть планирования. Сначала разрабатывается график работ по выполнению проекта или дается разрешение на продолжение использования ранее созданного графика. После этого проводится контроль выполнения работ и отмечаются расхождения между реальным и плановым ходом работ.

Далее, по мере поступления новой информации о ходе выполнения проекта, возможен пересмотр первоначальных оценок параметров проекта. Это, в свою очередь, может привести к изменению графика работ. Если в результате этих изменений нарушаются сроки завершения проекта, должны быть пересмотрены (и согласованы с заказчиком ПО) проектные ограничения.

Конечно, большинство руководителей проектов не думают, что реализация их проектов пройдет гладко, без всяких проблем. Желательно описать возможные проблемы еще до того, как они проявят себя в ходе выполнения проекта. Поэтому лучше составлять "пессимистические" графики работ, чем "оптимистические". Но, конечно, невозможно построить план, учитывающий все, в том числе случайные, проблемы и задержки выполнения проекта, поэтому и возникает необходимость периодического пересмотра проектных ограничений и этапов создания программного продукта.

План проекта должен четко показать ресурсы, необходимые для реализации проекта, разделение работ на этапы и временной график выполнения этих этапов. В некоторых организациях план проекта составляется как единый документ, содержащий все виды планов, описанных выше. В других случаях план проекта описывает только технологический процесс создания ПО. В таком плане обязательно присутствуют ссылки на планы других видов, но они разрабатываются отдельно от плана проекта.

Детализация планов проектов очень разнится в зависимости от типа разрабатываемого программного продукта и организации-разработчика. Но в любом случае большинство планов содержат следующие разделы.

1. *Введение*. Краткое описание целей проекта и проектных ограничений (бюджетных, временных и т.д.), которые важны для управления проектом.
2. *Организация выполнения проекта*. Описание способа подбора команды разработчиков и распределение обязанностей между членами команды.
3. *Анализ рисков*. Описание возможных проектных рисков, вероятности их проявления и стратегий, направленных на их уменьшение.
4. *Аппаратные и программные ресурсы, необходимые для реализации проекта*. Перечень аппаратных средств и программного обеспечения, необходимого для разработки программного продукта. Если аппаратные средства требуется закупать, приводится их стоимость совместно с графиком закупки и поставки.
5. *Разбиение работ на этапы*. Процесс реализации проекта разбивается на отдельные процессы, определяются этапы выполнения проекта, приводится описание результатов ("выходов") каждого этапа и контрольные отметки.
6. *График работ*. В этом графике отображаются зависимости между отдельными процессами (этапами) разработки ПО, оценки времени их выполнения и распределение членов команды разработчиков по отдельным этапам.
7. *Механизмы мониторинга и контроля за ходом выполнения проекта*. Описываются предоставляемые руководителем отчеты о ходе выполнения работ, сроки их предоставления, а также механизмы мониторинга всего проекта.

План должен регулярно пересматриваться в процессе реализации проекта. Одни части плана, например график работ, изменяются часто, другие более стабильны. Для внесения изменений в план требуется специальная организация документопотока, позволяющая отслеживать эти изменения.

Общие сведения о требованиях к информационным системам

Проблемы, которые приходится решать специалистам в процессе создания программного обеспечения, очень сложны. Природа этих проблем не всегда ясна, особенно если разрабатываемая программная система инновационная. В частности, трудно чѐтко описать те действия, которые должна выполнять система. Описание функциональных возможностей и ограничений, накладываемых на систему, называется требованиями к этой системе, а сам процесс формирования, анализа, документирования и проверки этих функциональных возможностей и ограничений – разработкой требований.

Требования подразделяются на пользовательские и системные. Пользовательские требования – это описание на естественном языке (плюс поясняющие диаграммы) функций, выполняемых системой, и ограничений, накладываемых на неѐ. Системные требования – это описание

особенностей системы (архитектура системы, требования к параметрам оборудования и т.д.), необходимых для эффективной реализации требований пользователя.

Первые шаги по разработке требований к информационным системам - анализ осуществимости

Разработка требований — это процесс, включающий мероприятия, необходимые для создания и утверждения документа, содержащего спецификацию системных требований. Для новых программных систем процесс разработки требований должен начинаться с анализа осуществимости. Началом такого анализа является общее описание системы и ее назначения, а результатом анализа — отчет, в котором должна быть четкая рекомендация, продолжать или нет процесс разработки требований проектируемой системы. Другими словами, анализ осуществимости должен осветить следующие вопросы.

1. Отвечает ли система общим и бизнес-целям организации-заказчика и организации-разработчика?
2. Можно ли реализовать систему, используя существующие на данный момент технологии и не выходя за пределы заданной стоимости?
3. Можно ли объединить систему с другими системами, которые уже эксплуатируются?

Критическим является вопрос, будет ли система соответствовать целям организации. Если система не соответствует этим целям, она не представляет никакой ценности для организации. В то же время многие организации разрабатывают системы, не соответствующие их целям, либо не совсем ясно понимая эти цели, либо под влиянием политических или общественных факторов.

Форма представления результата:

1. Цель работы
2. Введение. Краткое описание целей проекта и проектных ограничений (бюджетных, временных и т.д.), которые важны для управления проектом
3. Описание информационной системы (ПО) - наличие заключения о возможности реализации проекта, содержащего рекомендации относительно разработки системы, базовые предложения по объѐму требуемого бюджета, числу разработчиков, времени и требуемому программному обеспечению
4. Анализ осуществимости (согласно требованиям к результатам выполнения), указать возможные проблемы и пути их решения.
5. Роли участников группы разработки ПО.
6. Программно-аппаратные средства, используемые при выполнении работы.
7. Заключение (выводы)
8. Список используемой литературы

Критерии оценки:

Работа выполнена полностью и не содержит ошибок, студент учел и рассмотрел особенности предметной области и грамотно представил отчет – оценка «отлично».

Работа выполнена полностью, но содержит не более двух ошибок, студент учел и рассмотрел особенности предметной области и грамотно представил отчет – оценка «хорошо».

Работа выполнена с ошибками, студент не полностью учел и рассмотрел особенности предметной области и представил краткий отчет – оценка «удовлетворительно».

Работа выполнена с грубыми ошибками, студент не учел особенности предметной области, отчет составлен неграмотно – оценка «неудовлетворительно».

Лабораторное занятие № 2
Моделирование бизнес-процессов предметной области в нотации BPMN

Цель: Ознакомиться с процессом моделирования бизнес-процессов предметной области в нотации BPMN и принципами построения диаграмм в нотации BPMN при разработке программного обеспечения, выполнить моделирование бизнес-процессов заданной предметной области в нотации BPMN.

Выполнив задания, Вы будете:

уметь:

– У3 анализировать проектную и техническую документацию;

– У4 использовать специализированные графические средства построения и анализа архитектуры программных продуктов;

– У11 выявлять ошибки в системных компонентах на основе спецификаций.

Материальное обеспечение:

Пакет Microsoft Office, включая Microsoft Visio, доступ к Internet ресурсам.

Задание:

Выполните построение модели в нотации BPMN для заданнойпредметной области (лабораторное занятие № 1).

Требования к построению модели:

– модель должна отражать весь указанный в описании функционал;

– модель должна содержать не менее трех процессов.

Порядок выполнения работы:

1. Изучить предлагаемый теоретический материал по теме.
2. Построить модель выбранной предметной области в нотации BPMN в соответствии с требованиями.
7. Сформировать отчет о работе, включающий все полученные уровни модели, описание функциональных блоков, хранилищ и внешних объектов.

Ход работы:

Форма представления результата:

1. Цель работы
2. Введение
3. Программно-аппаратные средства, используемые при выполнении работы
4. Описание работы
5. Заключение (выводы)
6. Список используемой литературы

Критерии оценки:

Работа выполнена полностью и не содержит ошибок, студент учел и рассмотрел особенности предметной области и грамотно представил отчет – оценка «отлично».

Работа выполнена полностью, но содержит не более двух ошибок, студент учел и рассмотрел особенности предметной области и грамотно представил отчет – оценка «хорошо».

Работа выполнена с ошибками, студент не полностью учел и рассмотрел особенности предметной области и представил краткий отчет – оценка «удовлетворительно».

Работа выполнена с грубыми ошибками, студент не учел особенности предметной области, отчет составлен неграмотно – оценка «неудовлетворительно».

Лабораторное занятие № 3
Функциональное моделирование программного обеспечения в нотации IDEF0

Цель: Ознакомиться с процессом функционального моделирование программного обеспечения в нотации IDEF0 и принципами построения диаграмм в нотации IDEF0 и IDEF3, построить диаграммы в нотации IDEF0 IDEF3 для заданной предметной области.

Выполнив задания, Вы будете:

уметь:

– У3 анализировать проектную и техническую документацию;

– У4 использовать специализированные графические средства построения и анализа архитектуры программных продуктов;

– У11 выявлять ошибки в системных компонентах на основе спецификаций.

Материальное обеспечение:

Пакет Microsoft Office, доступ к Internet ресурсам.

Задание:

Выполните построение модели в нотации IDEF0 для заданнойпредметной области (лабораторное занятие № 1).

Требования к построению модели:

– модель должна отражать весь указанный в описании функционал;

– модель должна содержать не менее трех уровней декомпозиции в стандарте IDEF0 (контекстная диаграмма А-0, диаграмма А0, диаграммы 2-го уровня А1, А2 и т.д.);

– диаграмма 1-го уровня (А0) должна содержать не менее 4-х функциональных блоков;

– на диаграммах 2-го и далее уровнях декомпозиция функционала системы может быть представлена в нотации в стандарте IDEF0 или IDEF3;

– диаграммы 2-го и далее уровней должны содержать не менее 2-х функциональных блоков.

Порядок выполнения работы:

1. Изучить предлагаемый теоретический материал по теме.
2. Построить модель выбранной предметной области в нотации IDEF0 и IDEF3 в соответствии с требованиями.
3. Сформировать отчет о работе, включающий все полученные уровни модели, описание функциональных блоков, хранилищ и внешних объектов.

Ход работы:

Краткие теоретические сведения:

Основные понятия IDEF0

IDEF0 (Integrated Definition Function Modeling) методология функционального моделирования. В основе IDEF0 методологии лежит понятие блока, который отображает некоторую бизнес-функцию. Четыре стороны блока имеют разную роль: левая сторона имеет значение «входа», правая – «выхода», верхняя – «управления», нижняя – «механизма» (рисунок 1).

Взаимодействие между функциями в IDEF0 представляется в виде дуги, которая отображает поток данных или материалов, поступающий с выхода одной функции на вход другой. В зависимости от того, с какой стороной блока связан поток, его называют соответственно «входным», «выходным», «управляющим».

Рисунок 1 – Функциональный блок

Принципы моделирования в IDEF0

В IDEF0 реализованы три базовых принципа моделирования процессов:

- принцип функциональной декомпозиции;
- принцип ограничения сложности;
- принцип контекста.

Принцип функциональной декомпозиции представляет собой способ моделирования типовой ситуации, когда любое действие, операция, функция могут быть разбиты (декомпозированы) на более простые действия, операции, функции. Другими словами, сложная бизнес-функция может быть представлена в виде совокупности элементарных функций. Представляя функции графически, в виде блоков, можно как бы заглянуть внутрь блока и детально рассмотреть ее структуру и состав (рисунок 2).

Принцип ограничения сложности. При работе с IDEF0 диаграммами существенным является условие их разборчивости и удобочитаемости. Суть принципа ограничения сложности состоит в том, что количество блоков на диаграмме должно быть не менее двух и не более шести. Практика показывает, что соблюдение этого принципа приводит к тому, что функциональные процессы, представленные в виде IDEF0 модели, хорошо структурированы, понятны и легко поддаются анализу.

Принцип контекстной диаграммы. Моделирование делового процесса начинается с построения контекстной диаграммы. На этой диаграмме отображается только один блок - главная бизнес-функция моделируемой системы. Если речь идет о моделировании целого предприятия или даже крупного подразделения, главная бизнес-функция не может быть сформулирована как, например, «продавать продукцию». Главная бизнес-функция системы - это «миссия» системы, ее значение в окружающем мире. Нельзя правильно сформулировать главную функцию предприятия, не имея представления о его стратегии.

При определении главной бизнес-функции необходимо всегда иметь ввиду цель моделирования и точку зрения на модель. Одно и то же предприятие может быть описано по-разному, в зависимости от того, с какой точки зрения его рассматривают: директор предприятия и налоговой инспектор видят организацию совершенно по-разному.

Контекстная диаграмма играет еще одну роль в функциональной модели. Она «фиксирует» границы моделируемой бизнес-системы, определяя то, как моделируемая система взаимодействует со своим окружением. Это достигается за счет описания дуг, соединенных с блоком, представляющим главную бизнес-функцию.

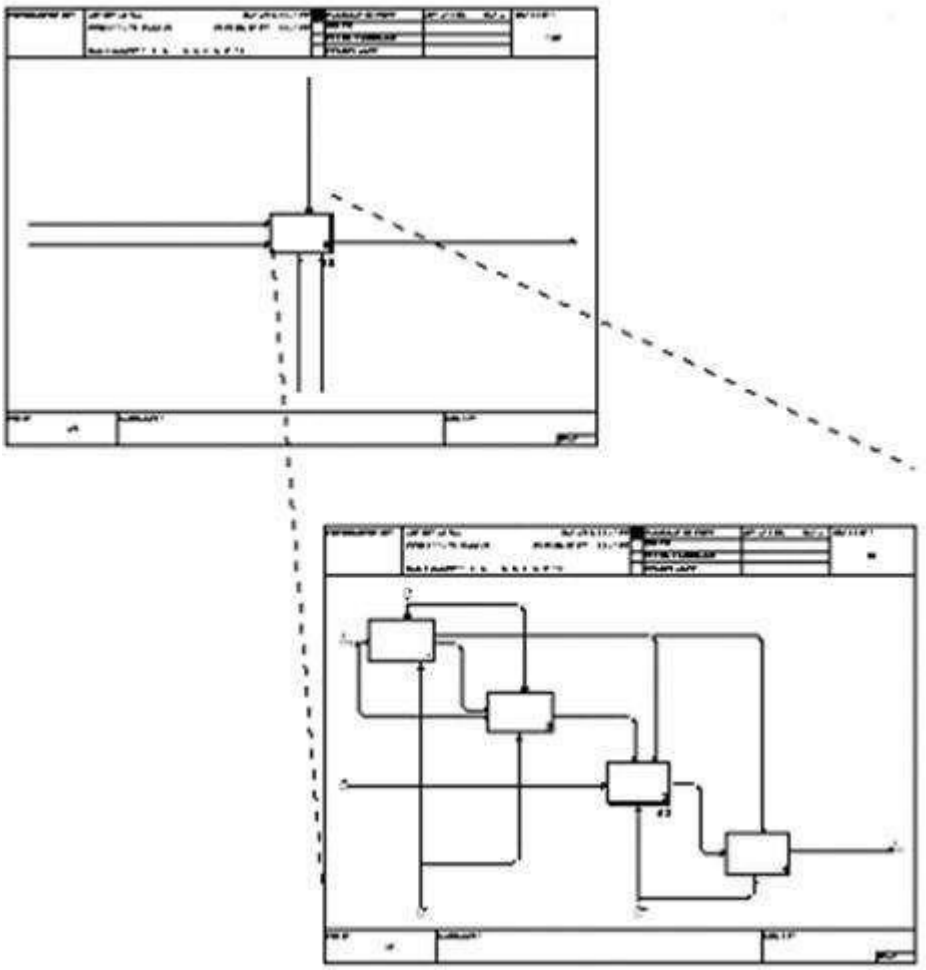


Рисунок 2 – Декомпозиция функционального блока

Пример.

На рисунок 3 и рисунок 4 представлен пример построения функциональной диаграммы, описывающей изготовление изделия. Рисунок 3 - контекстная диаграмма. Рисунок 4 – первый уровень декомпозиции.

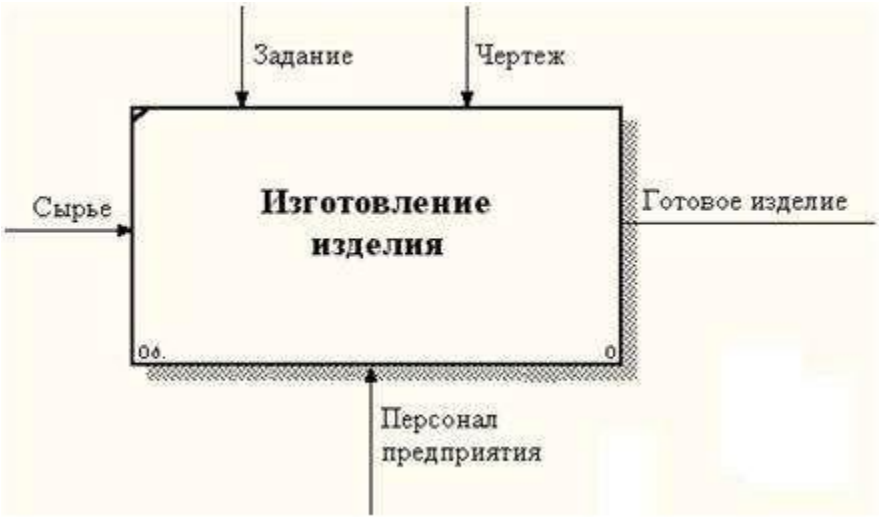


Рисунок 3 – Контекстная диаграмма

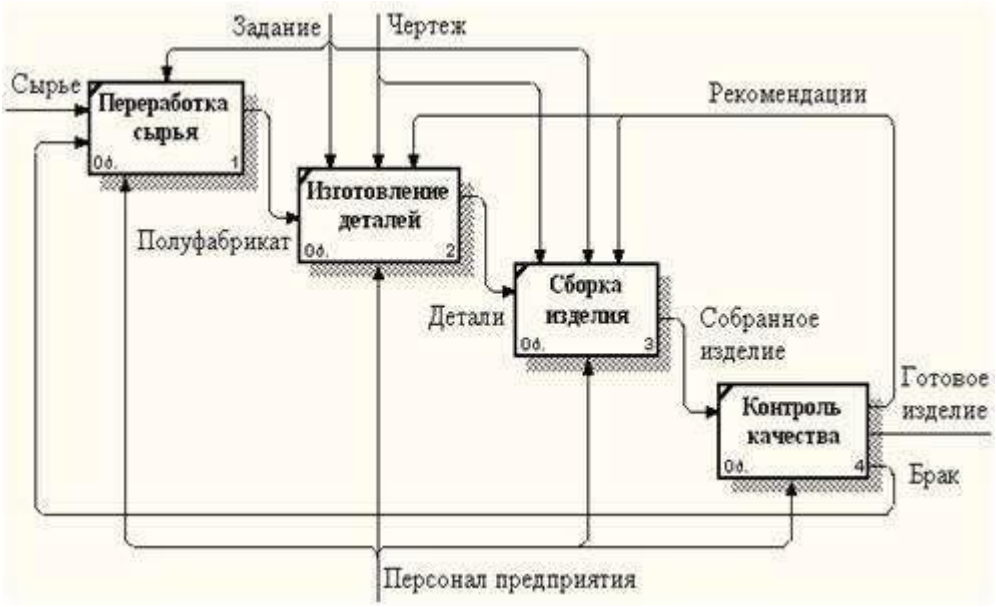


Рисунок 4 – Диаграмма первого уровня декомпозиции

Применение IDEF0

Существует два ключевых подхода к построению функциональной модели: построение «как есть» и построение «как будет».

Построение модели «как есть». Обследование предприятия является обязательной частью любого проекта создания или развития корпоративной информационной системы.

Построение функциональной модели ―как есть‖ позволяет четко зафиксировать, какие деловые процессы осуществляются на предприятии, какие информационные объекты используются при выполнении деловых процессов и отдельных операций. Функциональная модель «как есть» является отправной точкой для анализа потребностей предприятия, выявления проблем и «узких» мест и разработки проекта совершенствования деловых процессов.

Построение модели «как будет». Создание и внедрение корпоративной информационной системы приводит к изменению условий выполнения отдельных операций, структуры деловых процессов и предприятия в целом. Это приводит к необходимости изменения системы бизнес-правил, используемых на предприятии, модификации должностных инструкций сотрудников. Функциональная модель «как будет» позволяет уже на стадии проектирования будущей информационной системы определить эти изменения. Применение функциональной модели «как будет» позволяет не только сократить сроки внедрения информационной системы, но также снизить риски, связанные с невосприимчивостью персонала к информационным технологиям.

IDEF3. Метод описания процессов IDEF3

Для описания логики взаимодействия информационных потоков наиболее подходит IDEF3, называемая также workflow diagramming - методологией моделирования, использующая графическое описание информационных потоков, взаимоотношений между процессами обработки информации и объектов, являющихся частью этих процессов. Диаграммы Workflow могут быть использованы в моделировании бизнес-процессов для анализа завершенности процедур обработки информации. С их помощью можно описывать сценарии действий сотрудников организации, например последовательность обработки заказа или события, которые необходимо обработать за конечное время. Каждый сценарий сопровождается описанием процесса и может быть использован для документирования каждой функции.

IDEF3 – это метод, имеющий основной целью дать возможность аналитикам описать ситуацию, когда процессы выполняются в определенной последовательности, а также описать объекты, участвующие совместно в одном процессе,

Техника описания набора данных IDEF3 является частью структурного анализа. В отличие от некоторых методик описаний процессов IDEF3 не ограничивает аналитика чрезмерно жесткими рамками синтаксиса, что может привести к созданию неполных или противоречивых моделей.

IDEF3 может быть также использован как метод создания процессов. IDEF3 дополняет IDEF0 и содержит все необходимое для построения моделей, которые в дальнейшем могут быть использованы для имитационного анализа.

Каждая работа в IDEF3 описывает какой-либо сценарий бизнес-процесса и может являться составляющей другой работы. Поскольку сценарий описывает цель и рамки модели, важно, чтобы работы именовались отглагольным существительным, обозначающим процесс действия, или фразой, содержащей такое существительное.

Точка зрения на модель должна быть задокументирована. Обычно это точка зрения человека, ответственного за работу в целом. Также необходимо задокументировать цель модели - те вопросы, на которые призвана ответить модель.

Диаграммы. Диаграмма является основной единицей описания в IDEF3.

Единицы работы - Unit of Work (UOW). UOW, также называемые работами (activity), являются центральными компонентами модели. В IDEF3 работы изображаются прямоугольниками с прямыми углами и имеют имя, выраженное отглагольным существительным, обозначающим процесс действия, одиночным или в составе фразы, и номер (идентификатор); другое имя существительное в составе той же фразы обычно отображает основной выход (результат) работы, например, "Изготовление изделия".

Связи. Связи показывают взаимоотношения работ. Все связи в IDEF3 однонаправлены и могут быть направлены куда угодно, но обычно диаграммы IDEF3 стараются построить так, чтобы связи были направлены слева направо. В IDEF3 различают три типа стрелок, изображающих связи, стиль которых устанавливается через меню Edit/Arrow Style:

Старшая (Precedence) - сплошная линия, связывающая единицы работ (UOW), Рисуется слева направо или сверху вниз. Показывает, что работа-источник должна закончиться прежде, чем работа-цель начнется.

Отношения (Relational Link) - пунктирная линия, использующаяся для изображения связей между единицами работ (UOW) а также между единицами работ и объектами ссылок.

Потоки объектов (Object Flow) - стрелка с двумя наконечниками, применяется для описания того факта, что объект используется в двух или более единицах работы, например, когда объект порождается в одной работе и используется в другой.

Старшая связь и поток объектов. Старшая связь показывает, что работа-источник заканчивается ранее, чем начинается работа-цель. Часто результатом работы-источника становится объект, необходимый для запуска работы-цели. В этом случае стрелку, обозначающую объект, изображают с двойным наконечником. Имя стрелки должно ясно идентифицировать отображаемый объект. Поток объектов имеет ту же семантику, что и старшая стрелка.

Перекрестки (Junction). Окончание одной работы может служить сигналом к началу нескольких работ, или же одна работа для своего запуска может ожидать окончания нескольких работ. Перекрестки используются для отображения логики взаимодействия стрелок при слиянии и разветвлении или для отображения множества событий, которые могут или должны быть завершены перед началом следующей работы. Различают перекрестки для слияния (Fan-in Junction) и разветвления (Fan-out Junction) стрелок. Перекресток не может использоваться одновременно для слияния и для разветвления. Для внесения перекрестка служит кнопка в палитре инструментов - добавить в диаграмму перекресток Junction. В диалоге Junction Type Editor необходимо указать тип перекрестка. Смысл каждого типа приведен в таблице 1.

Таблица 1 - Типы перекрестков

Обозначение	Наименование	Смысл в случае слияния стрелок	Смысл в случае разветвления стрелок
	Asynchronous AND	Все предшествующие процессы должны быть завершены	Все следующие процессы должны быть запущены
	Synchronous AND	Все предшествующие процессы завершены одновременно	Все следующие процессы запускаются одновременно
	Asynchronous OR	Один или несколько предшествующих процессов должны	Один или несколько следующих процессов

		быть завершены	должны быть запущены
O	Synchronous OR	Один или несколько предшествующих процессов завершены одновременно	Один или несколько следующих процессов запускаются одновременно
X	XOR (Exclusive OR)	Только один предшествующий процесс завершен	Только один следующий процесс запускается

В отличие от IDEF0 в IDEF3 стрелки могут сливаться и разветвляться только через перекрестки.

Декомпозиция работ. В IDEF3 декомпозиция используется для детализации работ. Методология IDEF3 позволяет декомпозировать работу многократно, т.е. работа может иметь множество дочерних работ. Это позволяет в одной модели описать альтернативные потоки. Возможность множественной декомпозиции предъявляет дополнительные требования к нумерации работ. Так, номер работы состоит из номера родительской работы, версии декомпозиции и собственного номера работы на текущей диаграмме (рисунок 5).

Рисунок 5 – Номер единицы работы (UOW)

Форма представления результата:

1. Цель работы
2. Введение
3. Программно-аппаратные средства, используемые при выполнении работы.
4. Описание работы
5. Заключение (выводы)
6. Список используемой литературы

Критерии оценки:

Работа выполнена полностью и не содержит ошибок, студент грамотно представил отчет – оценка «отлично».

Работа выполнена полностью, но содержит не более двух ошибок, студент грамотно представил отчет – оценка «хорошо».

Работа выполнена с ошибками, студент представил краткий отчет – оценка «удовлетворительно».

Работа выполнена с грубыми ошибками, отчет составлен неграмотно – оценка «неудовлетворительно».

Лабораторное занятие № 4
Построение диаграмм потоков данных DFD

Цель: Ознакомиться с принципами построения диаграмм потоков данных DFD, построить диаграммы потоков данных DFD для заданной предметной области.

Выполнив работу, Вы будете:
уметь:уметь:

– У3 анализировать проектную и техническую документацию;
– У4 использовать специализированные графические средства построения и анализа архитектуры программных продуктов;
– У11 выявлять ошибки в системных компонентах на основе спецификаций.

Материальное обеспечение:
Пакет Microsoft Office, доступ к Internet ресурсам.

Задание:
Ориентируясь на диаграммы, построенные при выполнении лабораторной работы № 3, выполните построение диаграмм потоков данных DFD для заданнойпредметной области.
Требования к построению модели:
– модель должна отражать весь указанный в описании функционал.
–

Порядок выполнения работы:
1. Изучить предлагаемый теоретический материал по теме.
2. Построить диаграмму потоков данных DFD для выбранной предметной области в соответствии с требованиями.
3. Сформировать отчет о работе, включающий все полученные уровни модели, описание функциональных блоков, хранилищ и внешних объектов.

Ход работы:

Форма представления результата:
Цель работы
Введение
Программно-аппаратные средства, используемые при выполнении работы.
Описание работы
Заключение (выводы)
Список используемой литературы

Критерии оценки:
Работа выполнена полностью и не содержит ошибок, студент грамотно представил отчет – оценка «отлично».
Работа выполнена полностью, но содержит не более двух ошибок, студент грамотно представил отчет – оценка «хорошо».
Работа выполнена с ошибками, студент представил краткий отчет – оценка «удовлетворительно».
Работа выполнена с грубыми ошибками, отчет составлен неграмотно – оценка «неудовлетворительно».

Лабораторное занятие № 5
Разработка и оформление технического задания на разработку программного обеспечения

Цель: ознакомиться с правилами написания технического задания на разработку программного обеспечения; разработать техническое задание на создание программного продукта для заданной предметной области.

Выполнив работу, Вы будете:
уметь:
– У3 анализировать проектную и техническую документацию;
– У4 использовать специализированные графические средства построения и анализа архитектуры программных продуктов;
– У11 выявлять ошибки в системных компонентах на основе спецификаций.

Материальное обеспечение:
Пакет Microsoft Office, доступ к Internet ресурсам.

Задание:
Разработайте техническое задание в соответствии с ГОСТ 19.106-78 на создание программного продукта для заданной предметной области (лабораторное занятие № 1).

Порядок выполнения работы:
1. Изучить предлагаемый теоретический материал по теме.
2. Разработайте техническое задание на создание программного продукта для заданной предметной области в соответствии с ГОСТ 19.106-78.
3. Сформировать отчет о работе, включающий все полученные уровни модели, описание функциональных блоков, хранилищ и внешних объектов.

Ход работы:

Краткие теоретические сведения
ГОСТ 19.201-78

Настоящий стандарт устанавливает порядок построения и оформления технического задания на разработку программы или программного изделия для вычислительных машин, комплексов и систем независимо от их назначения и области применения.

Общие положения

1. Техническое задание оформляют в соответствии с ГОСТ 19.106-78 на листах формата А4 и А3 по ГОСТ 2.301-68, как правило, без заполнения полей листа. Номера листов (страниц) проставляют в верхней части листа над текстом.

2. Лист утверждения и титульный лист оформляют в соответствии с ГОСТ 19.104-78. Информационную часть (аннотацию и содержание), лист регистрации изменений допускается в документ не включать.

3. Для внесения изменений и дополнений в техническое задание на последующих стадиях разработки программы или программного изделия выпускают дополнение к нему. Согласование и утверждение дополнения к техническому заданию проводят в том же порядке, который установлен для технического задания.

4. Техническое задание должно содержать следующие разделы:
- название программы и область применения;
- основание для разработки;
- назначение разработки;
- технические требования к программе или программному изделию;
- технико-экономические показатели;
- стадии и этапы разработки:

- порядок контроля и приемки;
- приложения.

В зависимости от особенностей программы или программного изделия допускается уточнять содержание разделов, вводить новые разделы или объединять отдельные из них.

5. Содержание разделов

5.1. В разделе «Наименование и область применения» указывают наименование, краткую характеристику области применения программы или программного изделия и объекта, в котором используют программу или программное изделие.

5.2. В разделе «Основание для разработки» должны быть указаны:

• документ (документы), на основании которых ведется разработка;

• организация, утвердившая этот документ, и дата его утверждения;

• наименование и (или) условное обозначение темы разработки.

5.3. В разделе «Назначение разработки» должно быть указано функциональное и эксплуатационное назначение программы или программною изделия.

5.4. Раздел «Технические требования к программе или программному изделию» должен содержать следующие подразделы:

- требования к функциональным характеристикам;
- требовании к надежности;
- условия эксплуатации;
- требования к составу и параметрам технических средств;
- требования к информационной и программной совместимости;
- требования к маркировке и упаковке;
- требования к транспортированию и хранению;
- специальные требования.

5.5. В подразделе «Требования к функциональным характеристикам» должны быть указаны требования к составу выполняемых функций, организации входных и выходных данных, временным характеристикам и т. п.

5.6. В подразделе «Требования к надежности» должны быть указаны требования к обеспечению надежного функционирования (обеспечение устойчивого функционирования, контроль входной и выходной информации, время восстановления после отказа и т. п.).

5.7. В подразделе «Условия эксплуатации» должны быть указаны условия эксплуатации (температура окружающего воз духа, относительная влажность и т. п. для выбранных типов носителей данных), при которых должны обеспечиваться заданные характеристики, а также вид обслуживания, необходимое количество и квалификация персонала.

5.8. В подразделе «Требования к составу и параметрам технических средств» указывают необходимый состав технических средств с указанием их технических характеристик.

5.9. В подразделе «Требования к информационной и программной совместимости» должны быть указаны требования к информационным структурам на входе и выходе и методам решения, исходным кодам, языкам программирования. При необходимости должна обеспечиваться защита информации и программ.

5.10. В подразделе «Требования к маркировке и упаковке» в общем случае указывают требования к маркировке программного изделия, варианты и способы упаковки.

5.11. В подразделе «Требования к транспортированию и хранению» должны быть указаны для программного изделия условия транспортирования, места хранения, условия хранения, условия складирования, сроки хранения в различных условиях.

5.12. В разделе «Технико-экономические показатели» должны быть указаны: ориентировочная экономическая эффективность предполагаемая годовая потребность, экономические преимущества разработки по сравнению с лучшими отечественными и зарубежными образцами или аналогами.

5.13. В разделе «Стадии и этапы разработки* устанавливают необходимые стадии разработки, этапы и содержание работ (перечень программных документов, которые должны быть

разработаны, согласованы и утверждены), а также, как правило, сроки разработки и определяют исполнителей.

5.14. В разделе «Порядок контроля и приемки» должны быть указаны виды испытаний и общие требования к приемке работы.

5.15. В приложениях к техническому заданию при необходимости приводят:

- •перечень научно- исследовательских и других работ, обосновывающих разработку;
- • схемы алгоритмов, таблицы, описания, обоснования, расчеты и другие
- документы, которые могут быть использованы при разработке;
- • другие источники разработки.

Пример разработки технического задания.

Введение.

Работа выполняется в рамках проекта «Автоматизированная система оперативно-диспетчерского управления электро-, теплоснабжением корпусов института».

1. Основание для разработки

1. Основанием для данной работы служит договор № 1234 от 10 марта 2003г.

2. Наименование работы:

«Модуль автоматизированной системы оперативно-диспетчерского управления теплоснабжением корпусов института».

3. Исполнители: ОАО «Лаборатория создания программного обеспечения».

4. Соисполнители: нет.

5. Назначение разработки

Создание модуля для контроля и оперативной корректировки состояния основных параметров теплообеспечения корпусов Московского института.

6. Технические требования

6.1. Требования к функциональным характеристикам.

6.1.1. Состав выполняемых функций. Разрабатываемое ПО должно обеспечивать:

- сбор и анализ информации о расходовании тепла, горячей и холодной воды по данным теплосчетчиков SA-94 на всех тепловых выходах.;
- сбор и анализ информации с устройств управления системами воздушного отопления и кондиционирования типа РТ1 и РТ2 (разработки кафедры СММЭ и ТЦ);
- предварительный анализ информации на предмет нахождения параметров в допустимых пределах и сигнализирование при выходе параметров за пределы допуска;
- выдачу рекомендаций по дальнейшей работе;
- отображение текущего состояния по набору параметров - циклически постоянно (режим работы круглосуточный), при сохранении периодичности контроля прочих параметров;
- визуализацию информации по расходу теплоносителя:
  - текущую, аналогично показаниям счетчиков;
  - с накоплением за прошедшие сутки, неделю, месяц - в виде почасового графика для информации за сутки и неделю;
  - суточный расход - для информации за месяц.

Для устройств управления приточной вентиляцией текущая информация должна содержать номер приточной системы и все параметры, выдаваемые на собственный индикатор.

По отдельному запросу осуществляются внутренние на стройки.

В конце отчетного периода система должна архивировать данные.

6.1.2. Организация входных и выходных данных.

Исходные данные в систему поступают в виде значений с датчиков, становленных в помещениях института. Эти значения отображаются на компьютере диспетчера. После анализа поступившей информации оператор диспетчерского пункта устанавливает необходимые параметры для устройств, регулирующих отопление и вентиляцию в помещениях.

Возможна также автоматическая установка некоторых параметров для устройств регулирования.

Основной режим использования системы - ежедневная работа.

6.2. Требования к надежности.

Для обеспечения надежности необходимо проверять корректность получаемых данных с датчиков.

6.3. Условия эксплуатации и требования к составу и параметрам технических средств.

Для работы системы должен быть выделен ответственный оператор.

Требования к составу и параметрам технических средств уточняются на этапе эскизного проектирования системы.

6.4. Требования к информационной и программной совместимости.

Программа должна работать на платформах Windows 98/ NT/2000.

6.5. Требования к транспортировке и хранению. Программа поставляется на лазерном носителе информации. Программная документация поставляется в электронном и печатном виде.

6.6. Специальные требования. Программное обеспечение должно иметь дружественный интерфейс, рассчитанный на пользователя (в плане компьютерной грамотности)

средней квалификации.

Ввиду объемности проекта задачи предполагается решать поэтапно, при этом модули ПО, созданные в разнос время, должны предполагать возможность наращивания системы и быть совместимы друг с другом, поэтому документация на принятое эксплуатационное ПО должна содержать полную информацию, необходимую для работы программистов с ним.

Язык программирования – по выбору исполнителя, должен обеспечивать возможность интеграции программного обеспечения с некоторыми видами периферийного оборудования.

7. Требования к программной документации

Основными документами, регламентирующими разработку будущих программ, должны быть документы Единой Системы Программной Документации (ЕСПД); руководство пользователя, руководство администратора, описание применения.

8. Технико-экономические показатели

Эффективность системы определяется удобством использования системы для контроля и управления основными параметрами теплообеспечения помещений института, а также экономической выгодой, полученной от внедрения аппаратно-программного комплекса.

9. Порядок контроля и приемки

После передачи Исполнителем отдельного функционального модуля программы Заказчику, последний имеет право тестировать модуль в течение 7 дней. После тестирования Заказчик должен принять работу по данному этапу или в письменном виде изложить причину отказа от принятия. В случае обоснованного отказа Исполнитель обязуется доработать модуль.

Календарный план работ.

№	Название этапа	Сроки этапа	Чем закачивается этап
1.	Изучение предметной области. Проектирование системы. Разработка предложений по реализации системы	01.02.200_ - 28.02.200_	Предложения по работе системы. Акт сдачи-приѐмки.
2.	Разработка программного модуля по сбору и анализу информации со счѐтчиков и устройств управления. Внедрение системы для одного из корпусов.	01.03.200_ - 31.08.200_	Программный комплекс.
3	Тестирование и отладка модуля. Внедрение системы во всех корпусах.	01.09.200_ - 30.12.200_	Готовая система контроля теплоснабжения, установленная в диспетчерском пункте. Программная документация. Акт сдачи-приѐма работ

Руководитель работ Сидоров А.В.

Форма представления результата:

Техническое задание, разработанное в соответствии с ГОСТ 19.106-78.

Критерии оценки:

Работа выполнена полностью и не содержит ошибок, студент грамотно представил отчет – оценка «отлично».

Работа выполнена полностью, но содержит не более двух ошибок, студент грамотно представил отчет – оценка «хорошо».

Работа выполнена с ошибками, студент представил краткий отчет – оценка «удовлетворительно».

Работа выполнена с грубыми ошибками, отчет составлен неграмотно – оценка «неудовлетворительно».

Тема 2.1.3. Проектирование и разработка программного обеспечения

Лабораторное занятие № 6
Построение диаграммы деятельности и диаграммы состояний

Цель: изучить основные принципы построения диаграмм вариантов использования и диаграмм состояний с использованием языка UML; составить диаграмму вариантов использования и диаграмму состояний для заданной предметной области.

Выполнив работу, Вы будете:

уметь:

- У1 использовать выбранную систему контроля версий;
- У2 использовать методы для получения кода с заданной функциональностью и степенью качества;
- У3 анализировать проектную и техническую документацию;
- У4 использовать специализированные графические средства построения и анализа архитектуры программных продуктов;
- У5 организовывать заданную интеграцию модулей в программные средства на базе имеющейся архитектуры и автоматизации бизнес-процессов;
- У6 определять источники и приемники данных;
- У7 использовать приемы работы в системах контроля версий;
- У8 выполнять отладку, используя методы и инструменты условной компиляции (классы Debug и Trace);
- У12 использовать различные транспортные протоколы и стандарты форматирования сообщений.

Материальное обеспечение:

Пакет Microsoft Office, доступ к Internet ресурсам.

Задание:

Выполните построение диаграммы вариантов использования и диаграммы состояний с использованием языка UML для заданной предметной области (лабораторное занятие № 1).

Требования к построению модели:

– модель должна отражать весь указанный в описании функционал.

Порядок выполнения работы:

1. Изучить предлагаемый теоретический материал по теме.
2. Разработать диаграмму вариантов использования и диаграмму состояний и последовательности с использованием языка UML для заданной предметной области в соответствии с требованиями.
8. Сформировать отчет о работе.

Ход работы:

Краткие теоретические сведения:
Диаграммы состояний

Диаграммы состояний определяют все возможные состояния, в которых может находиться конкретный объект, а также процесс смены состояний объекта в результате наступления некоторых событий.

Существует много форм диаграмм состояний, незначительно отличающихся друг от друга семантикой.

На диаграмме имеются два специальных состояния – начальное (start) и конечное (stop). Начальное состояние выделено черной точкой, оно соответствует состоянию объекта, когда он только что был создан. Конечное состояние обозначается черной точкой в белом кружке, оно

соответствует состоянию объекта непосредственно перед его уничтожением. На диаграмме состояний может быть одно и только одно начальное состояние. В то же время, может быть столько конечных состояний, сколько вам нужно, или их может не быть вообще. Когда объект находится в каком-то конкретном состоянии, могут выполняться различные процессы. Процессы, происходящие, когда объект находится в определенном состоянии, называются действиями (actions).

С состоянием можно связывать данные пяти типов.

1. Деятельность

Деятельностью (activity) называется поведение, реализуемое объектом, пока он находится в данном состоянии. Деятельность – это прерываемое поведение. Оно может выполняться до своего завершения, пока объект находится в данном состоянии, или может быть прервано переходом объекта в другое состояние. Деятельность изображают внутри самого состояния, ей должно предшествовать слово do (делать) и двоеточие.

2. Входное действие

Входным действием (entry action) называется поведение, которое выполняется, когда объект переходит в данное состояние. Данное действие осуществляется не после того, как объект перешел в это состояние, а, скорее, как часть этого перехода. В отличие от деятельности, входное действие рассматривается как непрерываемое. Входное действие также показывают внутри состояния, ему предшествует слово entry (вход) и двоеточие.

3. Выходное действие

Выходное действие (exit action) подобно входному. Однако, оно осуществляется как составная часть процесса выхода из данного состояния. Оно является частью процесса такого перехода. Как и входное, выходное действие является непрерываемым.

Выходное действие изображают внутри состояния, ему предшествует слово exit (выход) и двоеточие.

Поведение объекта во время деятельности, при входных и выходных действиях может включать отправку события другому объекту. В этом случае описанию деятельности, входного действия или выходного действия предшествует знак « ^ ».

Соответствующая строка на диаграмме выглядит как

Do: ^Цель.Событие (Аргументы)

Здесь Цель – это объект, получающий событие, Событие – это посылаемое сообщение, а Аргументы являются параметрами посылаемого сообщения.

Деятельность может также выполняться в результате получения объектом некоторого события. При получении некоторого события выполняется определенная деятельность.

Переходом (Transition) называется перемещение из одного состояния в другое. Совокупность переходов диаграммы показывает, как объект может перемещаться между своими состояниями. На диаграмме все переходы изображают в виде стрелки, начинающейся на первоначальном состоянии и заканчивающейся последующим.

Переходы могут быть рефлексивными. Объект может перейти в то же состояние, в котором он в настоящий момент находится. Рефлексивные переходы изображают в виде стрелки, начинающейся и завершающейся на одном и том же состоянии.

У перехода существует несколько спецификаций. Они включают события, аргументы, ограждающие условия, действия и посылаемые события.

4. События

Событие (event) – это то, что вызывает переход из одного состояния в другое. Событие размещают на диаграмме вдоль линии перехода.

На диаграмме для отображения события можно использовать как имя операции, так и обычную фразу.

Большинство переходов должны иметь события, так как именно они, прежде всего, заставляют переход осуществиться. Тем не менее, бывают и автоматические переходы, не имеющие событий. При этом объект сам перемещается из одного состояния в другое со

скоростью, позволяющей осуществиться входным действиям, деятельности и выходным действиям.

Ограждающие условия

Ограждающие условия (guard conditions) определяют, когда переход может, а когда не может осуществиться. В противном случае переход не осуществится.

Ограждающие условия изображают на диаграмме вдоль линии перехода после имени события, заключая их в квадратные скобки.

Ограждающие условия задавать необязательно. Однако если существует несколько автоматических переходов из состояния, необходимо определить для них взаимно исключающие ограждающие условия. Это поможет читателю диаграммы понять, какой путь перехода будет автоматически выбран.

Действие

Действием (action), как уже говорилось, является непрерываемое поведение, осуществляющееся как часть перехода. Входные и выходные действия показывают внутри состояний, поскольку они определяют, что происходит, когда объект входит или выходит из него. Большую часть действий, однако, изображают вдоль линии перехода, так как они не должны осуществляться при входе или выходе из состояния.

Действие рисуют вдоль линии перехода после имени события, ему предшествует косая черта.

Событие или действие могут быть поведением внутри объекта, а могут представлять собой сообщение, посылаемое другому объекту. Если событие или действие посылается другому объекту, перед ним на диаграмме помещают знак « ^ ».

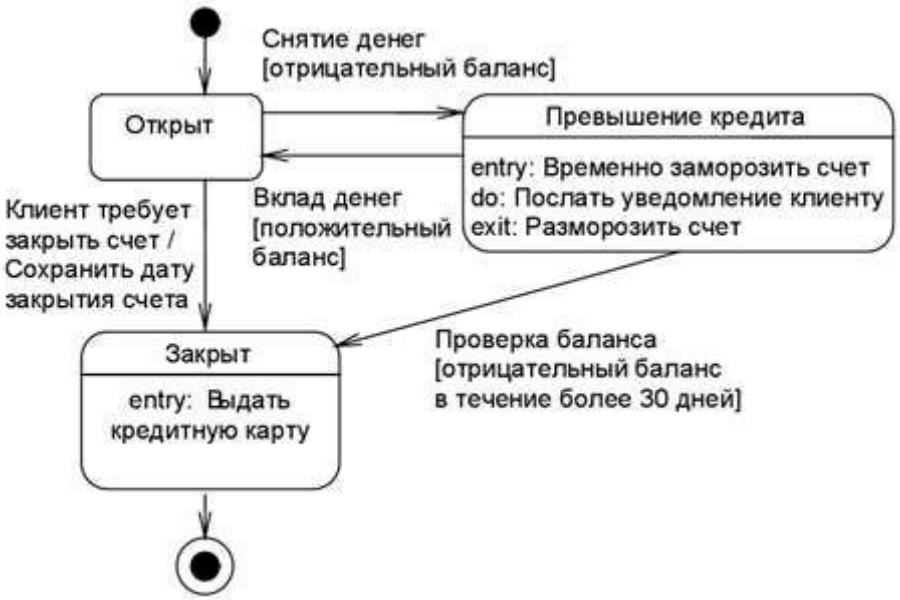


Рисунок 1 – Пример диаграммы состояний

Диаграммы состояний не надо создавать для каждого класса, они применяются только в сложных случаях. Если объект класса может существовать в нескольких состояниях и в каждом из них ведет себя по-разному, для него может потребоваться такая диаграмма.

Форма представления результата:

1. Цель работы
2. Введение
3. Программно-аппаратные средства, используемые при выполнении работы.
4. Описание работы
5. Заключение (выводы)
6. Список используемой литературы

Критерии оценки:

Работа выполнена полностью и не содержит ошибок, студент грамотно представил отчет – оценка «отлично».

Работа выполнена полностью, но содержит не более двух ошибок, студент грамотно представил отчет – оценка «хорошо».

Работа выполнена с ошибками, студент представил краткий отчет – оценка «удовлетворительно».

Работа выполнена с грубыми ошибками, отчет составлен неграмотно – оценка «неудовлетворительно».

Лабораторное занятие № 7
Построение диаграммы последовательности и диаграммы кооперации

Цель: изучить основные принципы построения диаграмм последовательности и диаграмм кооперации с использованием языка UML; составить диаграмму последовательности и диаграмму кооперации для заданной предметной области.

Выполнив работу, Вы будете:

уметь:

- У1 использовать выбранную систему контроля версий;
- У2 использовать методы для получения кода с заданной функциональностью и степенью качества;
- У3 анализировать проектную и техническую документацию;
- У4 использовать специализированные графические средства построения и анализа архитектуры программных продуктов;
- У5 организовывать заданную интеграцию модулей в программные средства на базе имеющейся архитектуры и автоматизации бизнес-процессов;
- У6 определять источники и приемники данных;
- У7 использовать приемы работы в системах контроля версий;
- У8 выполнять отладку, используя методы и инструменты условной компиляции (классы Debug и Trace);
- У12 использовать различные транспортные протоколы и стандарты форматирования сообщений.

Материальное обеспечение:

Пакет Microsoft Office, доступ к Internet ресурсам.

Задание:

Выполните построение диаграммы последовательности и диаграммы кооперации с использованием языка UML для заданной предметной области (лабораторное занятие № 1).

Требования к построению модели:

– модель должна отражать весь указанный в описании функционал.

Порядок выполнения работы:

1. Изучить предлагаемый теоретический материал по теме.
2. Разработать диаграмму вариантов использования и диаграмму состояний и последовательности с использованием языка UML для заданной предметной области в соответствии с требованиями.
3. Сформировать отчет о работе.

Ход работы:

Краткие теоретические сведения:
Диаграмма последовательности (Sequence Diagrams)

Диаграмма последовательности отражает поток событий, происходящих в рамках варианта использования.

Все действующие лица показаны в верхней части диаграммы. Стрелки соответствуют сообщениям, передаваемым между действующим лицом и объектом или между объектами для выполнения требуемых функций.

На диаграмме последовательности объект изображается в виде прямоугольника, от которого вниз проведена пунктирная вертикальная линия. Эта линия называется линией жизни (lifeline) объекта. Она представляет собой фрагмент жизненного цикла объекта в процессе взаимодействия.

Каждое сообщение представляется в виде стрелки между линиями жизни двух объектов. Сообщения появляются в том порядке, как они показаны на странице сверху вниз. Каждое

сообщение помечается как минимум именем сообщения. При желании можно добавить также аргументы и некоторую управляющую информацию. Можно показать самоделегирование (self-delegation) – сообщение, которое объект посылает самому себе, при этом стрелка сообщения указывает на ту же самую линию жизни.

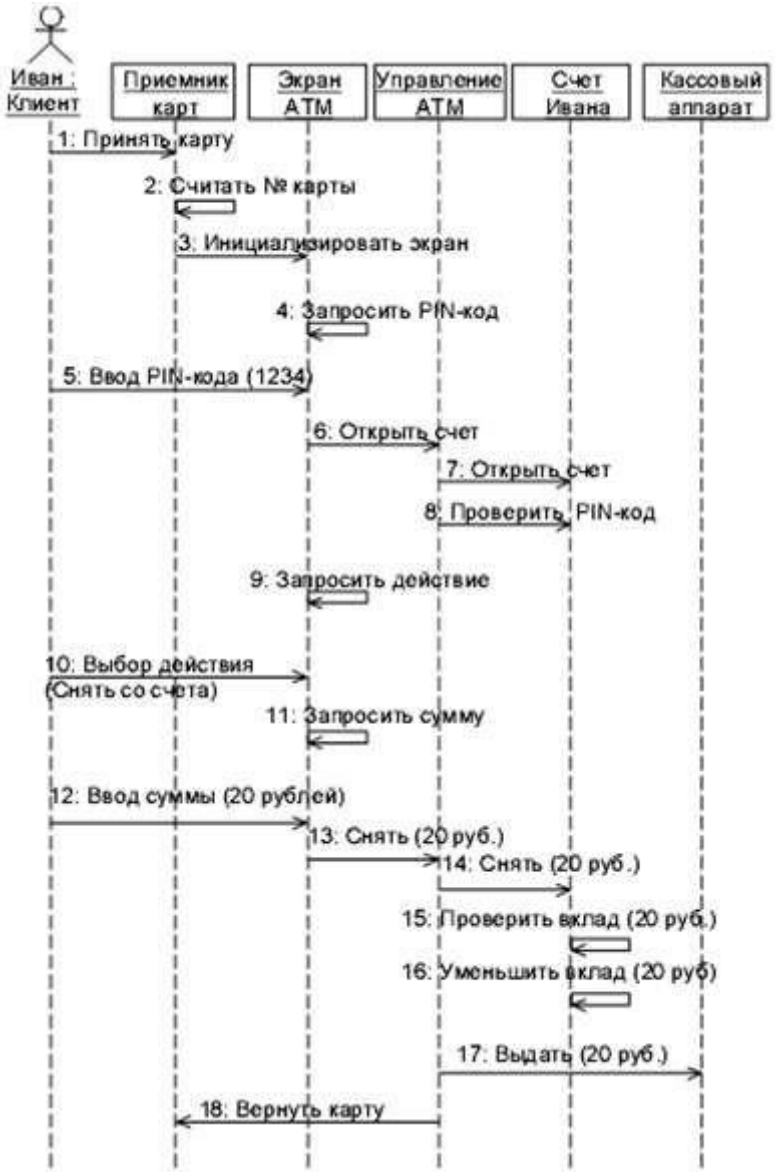


Рисунок 1 – Пример диаграммы последовательности

Диаграмма кооперации (Collaboration Diagram)

Диаграммы кооперации отображают поток событий через конкретный сценарий варианта использования, упорядочены по времени, а кооперативные диаграммы больше внимания заостряют на связях между объектами.

На диаграмме кооперации представлена вся та информация, которая есть и на диаграмме последовательности, но кооперативная диаграмма по-другому описывает поток событий. Из нее легче понять связи между объектами, однако, труднее уяснить последовательность событий.

На кооперативной диаграмме так же, как и на диаграмме последовательности, стрелки обозначают сообщения, обмен которыми осуществляется в рамках данного варианта использования. Их временная последовательность указывается путем нумерации сообщений.

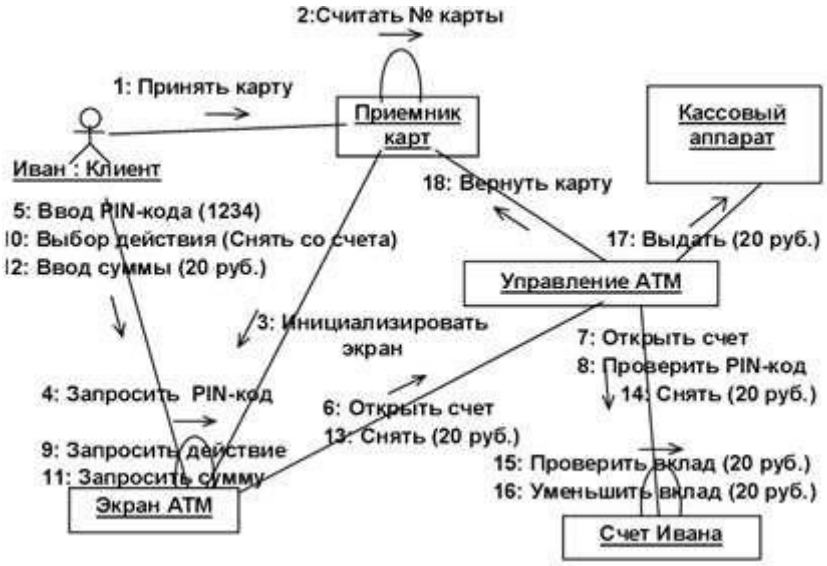


Рисунок 1 – Пример диаграммы кооперации

Форма представления результата:

1. Цель работы
2. Введение
3. Программно-аппаратные средства, используемые при выполнении работы.
4. Описание работы
5. Заключение (выводы)
6. Список используемой литературы

Критерии оценки:

Работа выполнена полностью и не содержит ошибок, студент грамотно представил отчет – оценка «отлично».

Работа выполнена полностью, но содержит не более двух ошибок, студент грамотно представил отчет – оценка «хорошо».

Работа выполнена с ошибками, студент представил краткий отчет – оценка «удовлетворительно».

Работа выполнена с грубыми ошибками, отчет составлен неграмотно – оценка «неудовлетворительно».

Лабораторное занятие № 8
Построение диаграммы классов и диаграммы компонентов

Цель: изучить основные принципы построения диаграмм классов, диаграмм компонентов, диаграмм развертывания с использованием языка UML; составить диаграмму классов, диаграмму компонентов, диаграмму развертывания для заданной предметной области.

Выполнив работу, Вы будете:

уметь:

- У1 использовать выбранную систему контроля версий;
- У2 использовать методы для получения кода с заданной функциональностью и степенью качества;
- У3 анализировать проектную и техническую документацию;
- У4 использовать специализированные графические средства построения и анализа архитектуры программных продуктов;
- У5 организовывать заданную интеграцию модулей в программные средства на базе имеющейся архитектуры и автоматизации бизнес-процессов;
- У6 определять источники и приемники данных;
- У7 использовать приемы работы в системах контроля версий;
- У8 выполнять отладку, используя методы и инструменты условной компиляции (классы Debug и Trace);
- У12 использовать различные транспортные протоколы и стандарты форматирования сообщений.

Материальное обеспечение:

Пакет Microsoft Office, доступ к Internet ресурсам.

Задание:

Выполните построение диаграммы классов, диаграммы компонентов, диаграммы развертывания с использованием языка UML для заданной предметной области (лабораторное занятие № 1).

Требования к построению модели:

– модель должна отражать весь указанный в описании функционал.

Порядок выполнения работы:

4. Изучить предлагаемый теоретический материал по теме.
5. Разработать диаграмму классов, диаграмму компонентов, диаграмму развертывания с использованием языка UML для заданной предметной области в соответствии с требованиями.
9. Сформировать отчет о работе.

Ход работы:

Краткие теоретические сведения:

Диаграммы классов

Диаграмма классов определяет типы классов системы и различного рода статические связи, которые существуют между ними. На диаграммах классов изображаются также атрибуты классов, операции классов и ограничения, которые накладываются на связи между классами.

Диаграмма классов UML - это граф, узлами которого являются элементы статической структуры проекта (классы, интерфейсы), а дугами - отношения между узлами (ассоциации, наследование, зависимости).

На диаграмме классов изображаются следующие элементы:

Пакет (package) - набор элементов модели, логически связанных между собой;

Класс (class) - описание общих свойств группы сходных объектов;

Интерфейс (interface) - абстрактный класс, задающий набор операций, которые объект произвольного класса, связанного с данным интерфейсом, предоставляет другим объектам.

Класс

Класс - это группа сущностей (объектов), обладающих сходными свойствами, а именно, данными и поведением. Отдельный представитель некоторого класса называется объектом класса или просто объектом.

Под поведением объекта в UML понимаются любые правила взаимодействия объекта с внешним миром и с данными самого объекта.

На диаграммах класс изображается в виде прямоугольника со сплошной границей, разделенного горизонтальными линиями на 3 секции:

Верхняя секция (секция имени) содержит имя класса и другие общие свойства (в частности, стереотип).

В средней секции содержится список атрибутов

В нижней - список операций класса, отражающих его поведение (действия, выполняемые классом).

Любая из секций атрибутов и операций может не изображаться (а также обе сразу). Для отсутствующей секции не нужно рисовать разделительную линию и как-либо указывать на наличие или отсутствие элементов в ней.

На усмотрение конкретной реализации могут быть введены дополнительные секции, например, исключения (Exceptions).

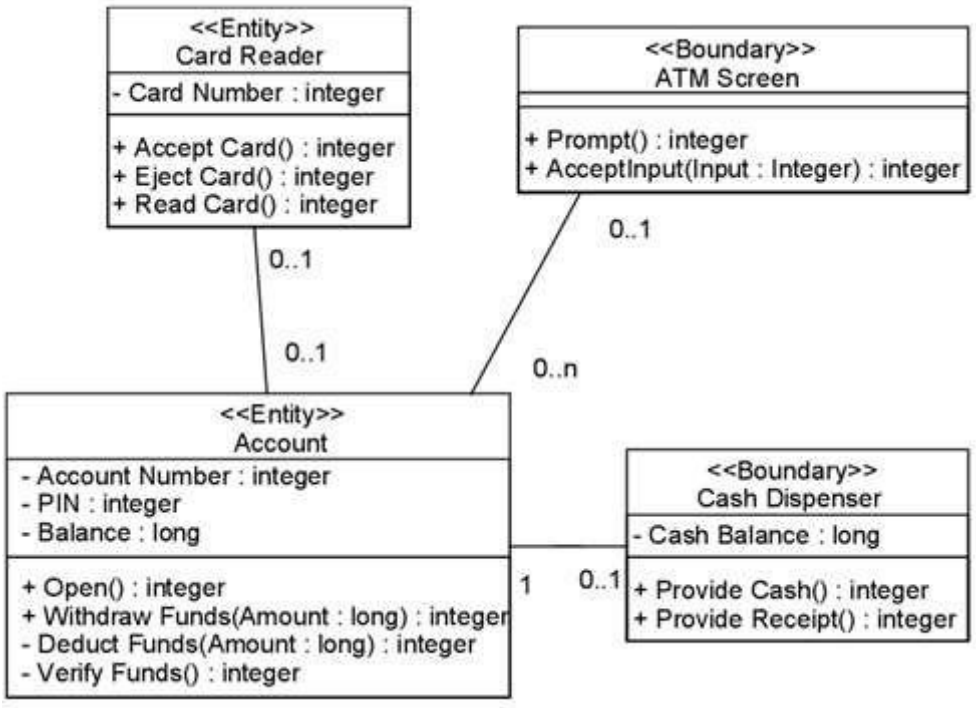


Рисунок 1 – Пример диаграммы классов

Стереотипы классов

Стереотипы классов – это механизм, позволяющий разделять классы на категории.

В языке UML определены три основных стереотипа классов:

- Boundary (граница);
- Entity (сущность);
- Control (управление).

Граничные классы

Граничными классами (boundary classes) называются такие классы, которые расположены на границе системы и всей окружающей среды. Это экранные формы, отчеты, интерфейсы с аппаратурой (такой как принтеры или сканеры) и интерфейсы с другими системами.

Чтобы найти граничные классы, надо исследовать диаграммы вариантов использования. Каждому взаимодействию между действующим лицом и вариантом использования должен соответствовать, по крайней мере, один граничный класс. Именно такой класс позволяет действующему лицу взаимодействовать с системой.

Классы-сущности

Классы-сущности (entity classes) содержат хранимую информацию. Они имеют наибольшее значение для пользователя, и потому в их названиях часто используют термины из предметной области. Обычно для каждого класса-сущности создают таблицу в базе данных.

Управляющие классы

Управляющие классы (control classes) отвечают за координацию действий других классов. Обычно у каждого варианта использования имеется один управляющий класс, контролирующий последовательность событий этого варианта использования. Управляющий класс отвечает за координацию, но сам не несет в себе никакой функциональности, так как остальные классы не посылают ему большого количества сообщений. Вместо этого он сам посылает множество сообщений. Управляющий класс просто делегирует ответственность другим классам, по этой причине его часто называют классом-менеджером.

В системе могут быть и другие управляющие классы, общие для нескольких вариантов использования. Например, может быть класс SecurityManager (менеджер безопасности), отвечающий за контроль событий, связанных с безопасностью. Класс TransactionManager (менеджер транзакций) занимается координацией сообщений, относящихся к транзакциям с базой данных. Могут быть и другие менеджеры для работы с другими элементами функционирования системы, такими как разделение ресурсов, распределенная обработка данных или обработка ошибок.

Помимо упомянутых выше стереотипов можно создавать и свои собственные.

Атрибуты

Атрибут – это элемент информации, связанный с классом. Атрибуты хранят инкапсулированные данные класса.

Так как атрибуты содержатся внутри класса, они скрыты от других классов. В связи с этим может понадобиться указать, какие классы имеют право читать и изменять атрибуты. Это свойство называется видимостью атрибута (attribute visibility).

У атрибута можно определить четыре возможных значения этого параметра:

Public (общий, открытый). Это значение видимости предполагает, что атрибут будет виден всеми остальными классами. Любой класс может просмотреть или изменить значение атрибута. В соответствии с нотацией UML общему атрибуту предшествует знак « + ».

Private (закрытый, секретный). Соответствующий атрибут не виден никаким другим классом. Закрытый атрибут обозначается знаком « – » в соответствии с нотацией UML.

Protected (защищенный). Такой атрибут доступен только самому классу и его потомкам. Нотация UML для защищенного атрибута – это знак « # ».

Package or Implementation (пакетный). Предполагает, что данный атрибут является общим, но только в пределах его пакета. Этот тип видимости не обозначается никаким специальным значком.

В общем случае, атрибуты рекомендуется делать закрытыми или защищенными. Это позволяет лучше контролировать сам атрибут и код.

С помощью закрытости или защищенности удается избежать ситуации, когда значение атрибута изменяется всеми классами системы. Вместо этого логика изменения атрибута будет заключена в том же классе, что и сам этот атрибут. Задаваемые параметры видимости повлияют на генерируемый код.

Операции

Операции реализуют связанное с классом поведение. Операция включает три части – имя, параметры и тип возвращаемого значения.

Параметры – это аргументы, получаемые операцией «на входе». Тип возвращаемого значения относится к результату действия операции.

На диаграмме классов можно показывать как имена операций, так и имена операций вместе с их параметрами и типом возвращаемого значения. Чтобы уменьшить загруженность диаграммы, полезно бывает на некоторых из них показывать только имена операций, а на других их полную сигнатуру.

В языке UML операции имеют следующую нотацию:

Имя Операции (аргумент: тип данных аргумента, аргумент2:тип данных аргумента2,...): тип возвращаемого значения

Следует рассмотреть четыре различных типа операций:

- операции реализации;
- операции управления;
- операции доступа;
- вспомогательные операции.

Операции реализации

Операции реализации (implementor operations) реализуют некоторые бизнес-функции. Такие операции можно найти, исследуя диаграммы взаимодействия. Диаграммы этого типа фокусируются на бизнес-функциях, и каждое сообщение диаграммы, скорее всего, можно соотнести с операцией реализации.

Каждая операция реализации должна быть легко прослеживаема до соответствующего требования. Это достигается на различных этапах моделирования. Операция выводится из сообщения на диаграмме взаимодействия, сообщения исходят из подробного описания потока событий, который создается на основе варианта использования, а последний – на основе требований. Возможность проследить всю эту цепочку позволяет гарантировать, что каждое требование будет реализовано в коде, а каждый фрагмент кода реализует какое-то требование.

Операции управления

Операции управления (manager operations) управляют созданием и уничтожением объектов. В эту категорию попадают конструкторы и деструкторы классов.

Операции доступа

Атрибуты обычно бывают закрытыми или защищенными. Тем не менее, другие классы иногда должны просматривать или изменять их значения. Для этого существуют операции доступа (access operations). Такой подход дает возможность безопасно инкапсулировать атрибуты внутри класса, защитив их от других классов, но все же позволяет осуществить к ним контролируемый доступ. Создание операций Get и Set (получения и изменения значения) для каждого атрибута класса является стандартом.

Вспомогательные операции

Вспомогательными (helper operations) называются такие операции класса, которые необходимы ему для выполнения его ответственностей, но о которых другие классы не должны ничего знать. Это закрытые и защищенные операции класса.

Чтобы идентифицировать операции, выполните следующие действия:

Изучите диаграммы последовательности и кооперативные диаграммы. Большая часть сообщений на этих диаграммах является операциями реализации. Рефлексивные сообщения будут вспомогательными операциями.

Рассмотрите управляющие операции. Может потребоваться добавить конструкторы и деструкторы.

Рассмотрите операции доступа. Для каждого атрибута класса, с которым должны будут работать другие классы, надо создать операции Get и Set.

Связи

Связь представляет собой семантическую взаимосвязь между классами. Она дает классу возможность узнавать об атрибутах, операциях и связях другого класса. Иными словами, чтобы один класс мог послать сообщение другому на диаграмме последовательности или кооперативной диаграмме, между ними должна существовать связь.

Существуют четыре типа связей, которые могут быть установлены между классами: ассоциации, зависимости, агрегации и обобщения.

Ассоциации

Ассоциация (association) – это семантическая связь между классами. Их рисуют на диаграмме классов в виде обыкновенной линии.

Рисунок 2 – Связь ассоциация

Ассоциации могут быть двунаправленными, как в примере, или однонаправленными. На языке UML двунаправленные ассоциации рисуют в виде простой линии без стрелок или со стрелками с обеих ее сторон. На однонаправленной ассоциации изображают только одну стрелку, показывающую ее направление.

Направление ассоциации можно определить, изучая диаграммы последовательности и кооперативные диаграммы. Если все сообщения на них отправляются только одним классом и принимаются только другим классом, но не наоборот, между этими классами имеет место однонаправленная связь. Если хотя бы одно сообщение отправляется в обратную сторону, ассоциация должна быть двунаправленной.

Ассоциации могут быть рефлексивными. Рефлексивная ассоциация предполагает, что один экземпляр класса взаимодействует с другими экземплярами этого же класса.

Зависимости

Связи зависимости (dependency) также отражают связь между классами, но они всегда однонаправлены и показывают, что один класс зависит от определений, сделанных в другом. Например, класс А использует методы класса В. Тогда при изменении класса В необходимо произвести соответствующие изменения в классе А.

Зависимость изображается пунктирной линией, проведенной между двумя элементами диаграммы, и считается, что элемент, привязанный к концу стрелки, зависит от элемента, привязанного к началу этой стрелки.

Рисунок 3 – Связь зависимость

При генерации кода для этих классов к ним не будут добавляться новые атрибуты. Однако, будут созданы специфические для языка операторы, необходимые для поддержки связи.

Агрегации

Агрегации (aggregations) представляют собой более тесную форму ассоциации. Агрегация – это связь между целым и его частью. Например, у вас может быть класс Автомобиль, а также классы Двигатель, Покрышки и классы для других частей автомобиля. В результате объект класса Автомобиль будет состоять из объекта класса Двигатель, четырех объектов Покрышек и т. д. Агрегации визуализируют в виде линии с ромбиком у класса, являющегося целым:

Рисунок 4 – Связь агрегация

В дополнение к простой агрегации UML вводит более сильную разновидность агрегации, называемую композицией. Согласно композиции, объект-часть может принадлежать только

единственному целому, и, кроме того, как правило, жизненный цикл частей совпадает с циклом целого: они живут и умирают вместе с ним. Любое удаление целого распространяется на его части.

Такое каскадное удаление нередко рассматривается как часть определения агрегации, однако оно всегда подразумевается в том случае, когда множественность роли составляет 1..1; например, если необходимо удалить Клиента, то это удаление должно распространиться и на Заказы (и, в свою очередь, на Строки заказа).

Обобщения (Наследование)

Обобщение (наследование) - это отношение типа общее-частное между элементами модели. С помощью обобщений (generalization) показывают связи наследования между двумя классами. Большинство объектно-ориентированных языков непосредственно поддерживают концепцию наследования. Она позволяет одному классу наследовать все атрибуты, операции и связи другого. Наследование пакетов означает, что в пакете-наследнике все сущности пакета-предка будут видны под своими собственными именами (т.е. пространства имен объединяются). Наследование показывается сплошной линией, идущей от класса-потомка к классу-предку (в терминологии ООП - от потомка к предку, от сына к отцу, или от подкласса к суперклассу). Со стороны более общего элемента рисуется большой полый треугольник.

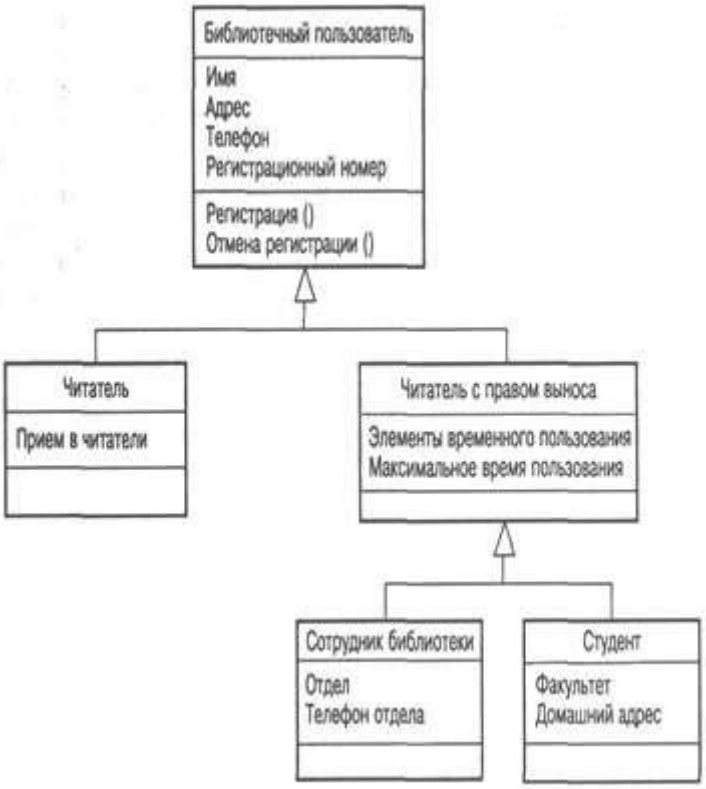


Рисунок 5 – Пример связи наследование

Помимо наследуемых, каждый подкласс имеет свои собственные уникальные атрибуты, операции и связи.

Множественность

Множественность (multiplicity) показывает, сколько экземпляров одного класса взаимодействуют с помощью этой связи с одним экземпляром другого класса в данный момент времени.

Например, при разработке системы регистрации курсов в университете можно определить классы Course (курс) и Student (студент). Между ними установлена связь: у курсов могут быть студенты, а у студентов – курсы. Вопросы, на который должен ответить параметр множественности: «Сколько курсов студент может посещать в данный момент? Сколько студентов может за раз посещать один курс?»

Так как множественность дает ответ на оба эти вопроса, еѐ индикаторы устанавливаются на обоих концах линии связи. В примере регистрации курсов мы решили, что один студент может посещать от нуля до четырех курсов, а один курс могут слушать от 0 до 20 студентов.

В языке UML приняты определенные нотации для обозначения множественности.

Таблица 1 – Обозначения множественности связей в UML

Множественность	Значение
0..*	Ноль или больше
1..*	Один или больше
0..1	Ноль или один
1..1 (сокращенная запись: 1)	Ровно один

Имена связей

Связи можно уточнить с помощью имен связей или ролевых имен. Имя связи – это обычно глагол или глагольная фраза, описывающая, зачем она нужна. Например, между классом Person (человек) и классом Company (компания) может существовать ассоциация. Можно задать в связи с этим вопрос, является ли объект класса Person клиентом компании, еѐ сотрудником или владельцем? Чтобы определить это, ассоциацию можно назвать «employs» (нанимает):

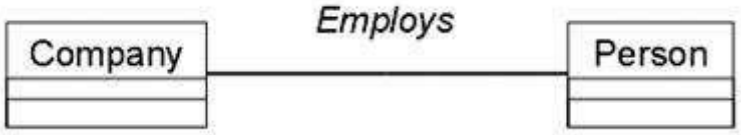


Рисунок 6 – Пример имен связей

Роли

Ролевые имена применяют в связях ассоциации или агрегации вместо имен для описания того, зачем эти связи нужны. Возвращаясь к примеру с классами Person и Company, можно сказать, что класс Person играет роль сотрудника класса Company. Ролевые имена – это обычно имена существительные или основанные на них фразы, их показывают на диаграмме рядом с классом, играющим соответствующую роль. Как правило, пользуются или ролевым именем, или именем связи, но не обоими сразу. Как и имена связей, ролевые имена не обязательны, их дают, только если цель связи не очевидна. Пример ролей приводится ниже:

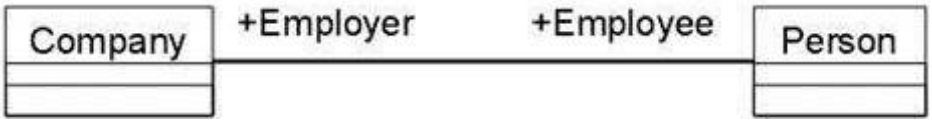


Рисунок 7 – Пример ролей связей

Пакет. Механизм пакетов

В контексте диаграмм классов, пакет - это вместилище для некоторого набора классов и других пакетов. Пакет является самостоятельным пространством имен.

Рисунок 8 – Обозначение пакета в UML

В UML нет каких-либо ограничений на правила, по которым разработчики могут или должны группировать классы в пакеты. Но есть некоторые стандартные случаи, когда такая

группировка уместна, например, тесно взаимодействующие классы, или более общий случай - разбиение системы на подсистемы.

Пакет физически содержит сущности, определенные в нем (говорят, что "сущности принадлежат пакету"). Это означает, что если будет уничтожен пакет, то будут уничтожено и все его содержимое.

Существует несколько наиболее распространенных подходов к группировке.

Во-первых, можно группировать их по стереотипу. В таком случае получается один пакет с классами-сущностями, один с граничными классами, один с управляющими классами и т.д. Этот подход может быть полезен с точки зрения размещения готовой системы, поскольку все находящиеся на клиентских машинах пограничные классы уже оказываются в одном пакете.

Другой подход заключается в объединении классов по их функциональности. Например, в пакете Security (безопасность) содержатся все классы, отвечающие за безопасность приложения. В таком случае другие пакеты могут называться Employee Maintenance (Работа с сотрудниками), Reporting (Подготовка отчетов) и Error Handling (Обработка ошибок). Преимущество этого подхода заключается в возможности повторного использования.

Механизм пакетов применим к любым элементам модели, а не только к классам. Если для группировки классов не использовать некоторые эвристики, то она становится произвольной. Одна из них, которая в основном используется в UML, – это зависимость. Зависимость между двумя пакетами существует в том случае, если между любыми двумя классами в пакетах существует любая зависимость.

Таким образом, диаграмма пакетов представляет собой диаграмму, содержащую пакеты классов и зависимости между ними. Строго говоря, пакеты и зависимости являются элементами диаграммы классов, то есть диаграмма пакетов – это форма диаграммы классов.

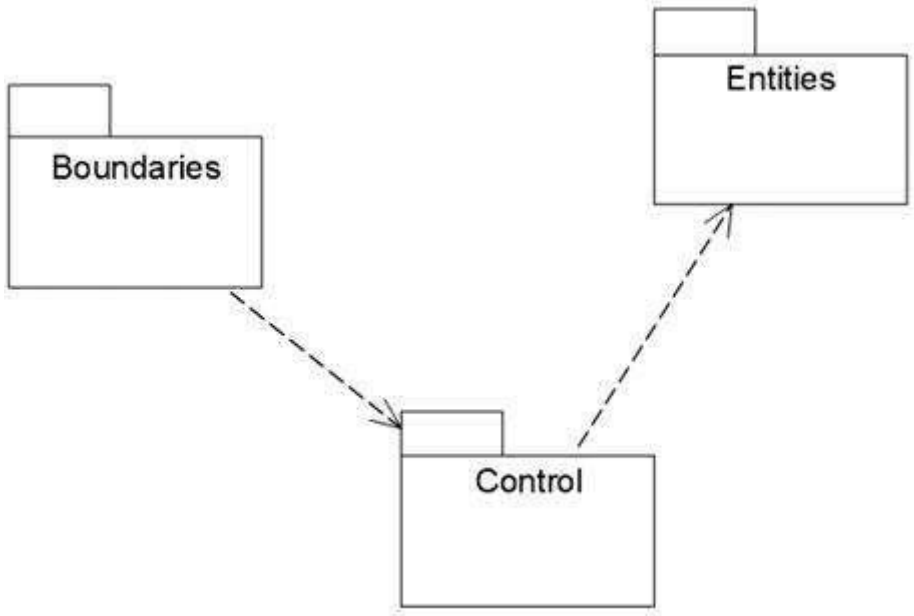


Рисунок 9 – Пример диаграммы пакетов

Зависимость между двумя элементами имеет место в том случае, если изменения в определении одного элемента могут повлечь за собой изменения в другом. Что касается классов, то причины для зависимостей могут быть самыми разными:

- один класс посылает сообщение другому;
- один класс включает часть данных другого класса; один класс использует другой в качестве параметра операции.

Если класс меняет свой интерфейс, то любое сообщение, которое он посылает, может утратить свою силу.

Пакеты не дают ответа на вопрос, каким образом можно уменьшить количество зависимостей в вашей системе, однако они помогают выделить эти зависимости, а после того, как они все окажутся на виду, остается только поработать над снижением их количества. Диаграммы пакетов можно считать основным средством управления общей структурой системы.

Пакеты являются жизненно необходимым средством для больших проектов. Их следует использовать в тех случаях, когда диаграмма классов, охватывающая всю систему в целом и размещенная на единственном листе бумаги формата А4, становится нечитаемой.

Диаграммы компонентов

Диаграммы компонентов показывают, как выглядит модель на физическом уровне. На них изображены компоненты программного обеспечения и связи между ними. При этом на такой диаграмме выделяют два типа компонентов: исполняемые компоненты и библиотеки кода.

Каждый класс модели (или подсистема) преобразуется в компонент исходного кода. После создания они сразу добавляются к диаграмме компонентов. Между отдельными компонентами изображают зависимости, соответствующие зависимостям на этапе компиляции или выполнения программы.

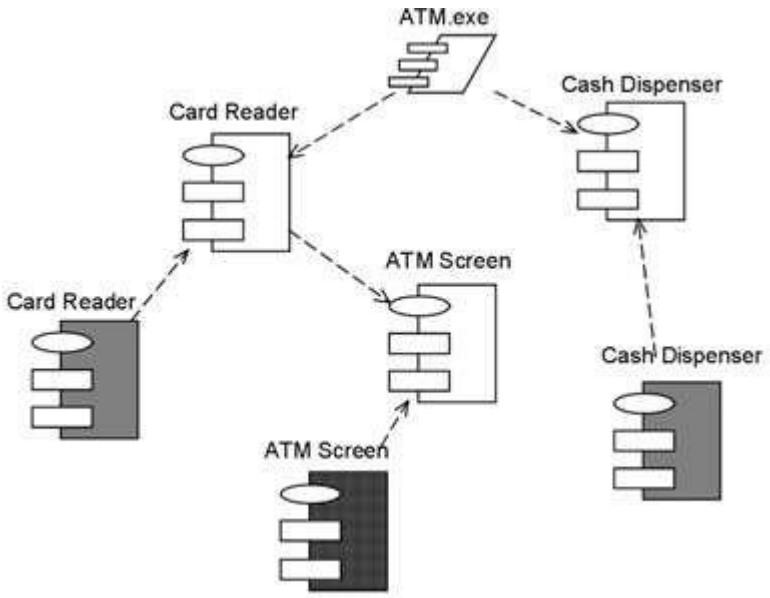


Рисунок 11 – Пример диаграммы компонентов

Диаграммы компонентов применяются теми участниками проекта, кто отвечает за компиляцию системы. Из нее видно, в каком порядке надо компилировать компоненты, а также какие исполняемые компоненты будут созданы системой. На такой диаграмме показано соответствие классов реализованным компонентам. Она нужна там, где начинается генерация кода.

Форма представления результата:

1. Цель работы
2. Введение
3. Программно-аппаратные средства, используемые при выполнении работы.
4. Описание работы
5. Заключение (выводы)
6. Список используемой литературы

Критерии оценки:

Работа выполнена полностью и не содержит ошибок, студент грамотно представил отчет – оценка «отлично».

Работа выполнена полностью, но содержит не более двух ошибок, студент грамотно представил отчет – оценка «хорошо».

Работа выполнена с ошибками, студент представил краткий отчет – оценка «удовлетворительно».

Работа выполнена с грубыми ошибками, отчет составлен неграмотно – оценка «неудовлетворительно».

Лабораторное занятие № 9
Выбор и обоснование архитектуры программного обеспечения, построение диаграмм развертывания

Цель: ознакомиться с основными принципами построения архитектуры программного средства, выполнить построение архитектуры программного обеспечения для заданной предметной области.

Выполнив работу, Вы будете:
уметь:

- У1 использовать выбранную систему контроля версий;
- У2 использовать методы для получения кода с заданной функциональностью и степенью качества;
- У3 анализировать проектную и техническую документацию;
- У4 использовать специализированные графические средства построения и анализа архитектуры программных продуктов;
- У5 организовывать заданную интеграцию модулей в программные средства на базе имеющейся архитектуры и автоматизации бизнес-процессов;
- У6 определять источники и приемники данных;
- У7 использовать приемы работы в системах контроля версий;
- У8 выполнять отладку, используя методы и инструменты условной компиляции (классы Debug и Trace);
- У12 использовать различные транспортные протоколы и стандарты форматирования сообщений.

Материальное обеспечение:
Пакет Microsoft Office, доступ к Internet ресурсам.

Задание:
На основании технического задания, разработанного в лабораторной работе № 5, спроектируйте архитектуру программного продукта.

Порядок выполнения работы:

1. Изучить предлагаемый теоретический материал по теме.
2. Выбрать наиболее подходящий вид архитектуры программного продукта, обосновать свой выбор.
3. Выбрать парадигму программирования, обосновать выбор.
4. Сформировать отчет о работе, включающий описание архитектуры программного продукта в соответствии с техническим заданием на разработку, выбор парадигмы программирования, обоснование выбора.

Ход работы:

Краткие теоретические сведения:
Рассмотрим принципы подхода к решению задачи построения архитектуры ПС по ее программному коду. Одним из способов проведения анализа является формирование графов с различным уровнем отображения внутренних объектов и связей. Модель данных большой программной системы, реализованной на языках программирования C/C++, представляет собой наборы директорий, содержащих различные по использованию файлы: с исходными кодами, с описанием типов объектов и вспомогательные.
Основные принципы построения архитектуры ПС:

1. отображение графа связей файлов проекта (показывает, каким образом исходные файлы проекта подключают друг друга);
2. отображение иерархии наследования классов;
3. отображение диаграммы вызовов функций (диаграмма показывает, каким образом управление попадает в выбранную функцию и куда оно передается из нее; связь на диаграмме соответствует вызову функции);
4. регулирование объема отображаемой информации путем выделения любого элемента программы с отображением только тех элементов, которые взаимодействуют с ним;
5. для любого элемента программы необходима возможность просмотреть участки кода, в которых этот элемент используется;
6. сохранение исходных файлов исследуемой программы в хранилище данных;
7. возможность выделения из всей программы только необходимой функциональной части и сборка только этой части.

Основные задачи анализа:

- выявление описанных, но неиспользуемых элементов;
- изучение исходного кода, на который отсутствует документация;
- исследование исходного кода для дальнейшей модернизации программного продукта;
- исследование исходного кода для дальнейшего переноса программного продукты на другую платформу;
- реструктуризация исходного кода через специально созданное хранилище данных.

Для решения задач было разработано средство анализа исходных текстов. Анализ включает в себя выделение из текстов программ необходимой разработчику информации для поиска процедур и функций, для поиска имен описания типов объектов и самих объектов. На основе выделения формируются структуры различных типов - по функциям и процедурам, по объектам (их именам, их типам) с отображением адреса нахождения искомого объекта и его связей с другими объектами (например, список всех подчиненных функций одного модуля).

Основной подход при разработке средств анализа исходных текстов - создание кросс-платформенного продукта. Использованы скриптовые языки Python, Lua. Это стабильные языки, использующиеся во многих проектах в качестве базовых или для создания расширений. Для построения графического интерфейса пользователя использована библиотека wxWidgets, которая позволяет приложению выглядеть одинаково на всех платформах. Основной код wxWidgets вызывает элемент интерфейса платформы, вместо того чтобы повторно его реализовывать. Это предоставляет быстрый, естественно выглядящий интерфейс на каждой платформе. Для качественной визуализации используется комплекс Graphviz. Данные технологии поддерживают большое количество платформ и распространяются под свободными лицензиями.

Структура данных системы. При анализе исходного текста программа загружает объекты в виртуальное хранилище данных и позволяет отобразить различные представления: функциональную связь (рисунок 1), связи по глобальным объектам (рисунок 2), связь по описателям объектов (рисунок 3).

Схема функциональных связей предоставляет информацию о наличии описания различных функций в расположенных в разных директориях файлах и о том, где именно эти функции вызываются (каждая связь на диаграмме соответствует вызову функции). На диаграмме показано, как будет выглядеть рекурсия, - замыкание функции 7 самой в себя. Каждая функция имеет адрес вида «Директория_1, Файл_1.c, Функция^». Для построения графа связей конкретной функции можно сделать соответствующий запрос.

Схема связей по глобальным объектам похожа на предыдущую, но отображает использование глобальных переменных, констант, объектов. Схема описателей объектов отображает зависимость расположенных в разных файлах описаний классов, типов, структур. Связь вида «Типi – Тип2» обозначает, что Тип2 является одним из составляющих Типi. В классах это отношение «Потомок —> Родитель». Именно такой вид представлений позволяет получить исчерпывающую низкоуровневую информацию о рабочем проекте. При достаточно больших

проектах происходит консолидация всей необходимой для разработчика информации в одном месте.

Это позволяет разработчику объективно увидеть и оценить работу программы, найти неиспользуемые объекты (например, объект_а на рисунок 2), обнаружить рекурсии, увидеть родительские классы и их отличие от потомков.

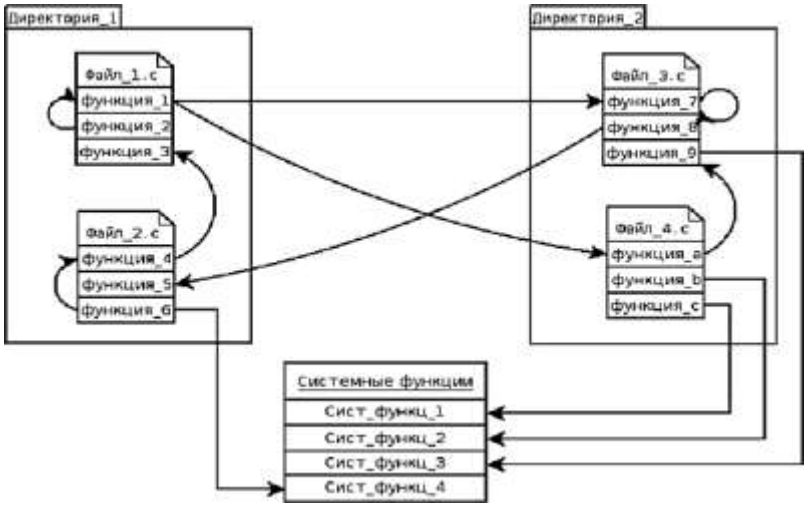


Рисунок 1 – Отображение функциональных связей

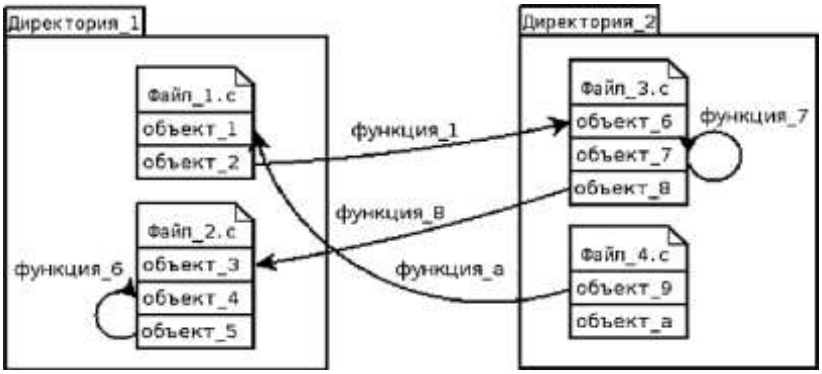


Рисунок 2 – Отображение связей между объектами

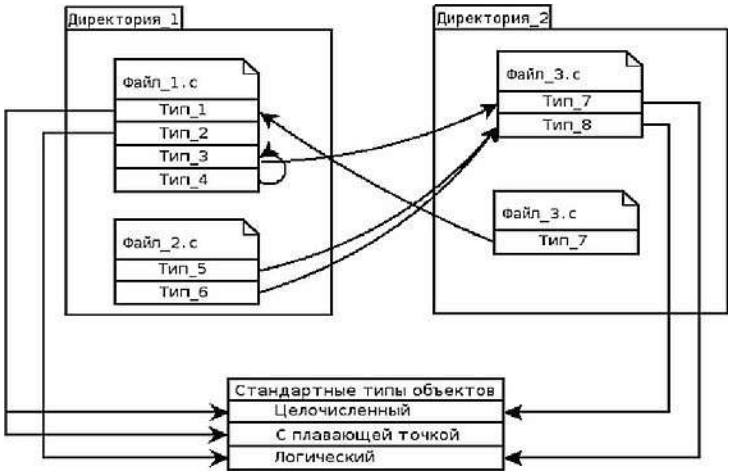


Рисунок 3 – Отображение связей по описателям объектов

Технологические цепочки. Программа работает следующим образом: при выборе проекта для анализа программа производит поиск исходных файлов и при обнаружении начинает загружать их в собственное хранилище данных, разбивая на простейшие объекты и сохраняя их зависимости. На втором этапе происходит работа непосредственно с хранилищем, а именно: в

зависимости от целевого представления анализируются необходимые объекты данных и строятся зависимости, происходит обращение к ОгарИущ для получения графических координат элементов. Третий этап заключается в отображении на экране полученной информации.

В данной программе у программиста есть функционал сборки приложения либо части приложения. При обычном способе сборки, который используется большинством популярных ГОЕ, будут получены исполняемый файл и набор библиотек для работы программы. Библиотеки включают в себя все функции, и вычленить только часть не представляется возможным. Обычная сборка показана на рисунок 4.

Что делать, если требуется собрать программу только с одной или двумя основными функциями? Традиционный способ - переработка исходных текстов, ручной поиск зависимостей вызовов функций и включение в итоговый продукт только выбранных функций. Система позволяет автоматизировать этот процесс. На рисунок 5 отображен процесс сборки только одной функции (функция !).

Рисунок 4 – Обычная сборка

Пользователь системы, выбрав нужную компоновку (требуемые функции) и расположение файлов целевого приложения, запускает процедуру обработки. Происходит обращение к хранилищу данных, подготовка данных, создание новой иерархии директорий и выгрузка необходимых объектов (классов, функций и т.д.) в микросреду для сборки -файлов нового проекта. Файлы имеют расположение, выбранное пользователем. Далее происходят компиляция и линковка. Результатом являются исполняемый файл с требуемыми функциями и библиотека,

которая содержит требуемые зависимости. Возможность выделения необходимых функций полезна при модификации приложения, при коллективной работе, при аудите исходных текстов сторонней организацией.

Для исследования архитектуры программных систем используется нотация структурного моделирования, принятая в инструменте KLOCwork Architect. Этот инструмент предоставляет возможность автоматического извлечения моделей из программного кода и их редактирования. Далее рассматривается эта нотация.

Модели программных систем, используемые в KLOCwork Architect (в дальнейшем- модель), отдаленно напоминают модели типа «сущность - отношение» (entity-relation models). Основными единицами модели являются архитектурные блоки и отношения.

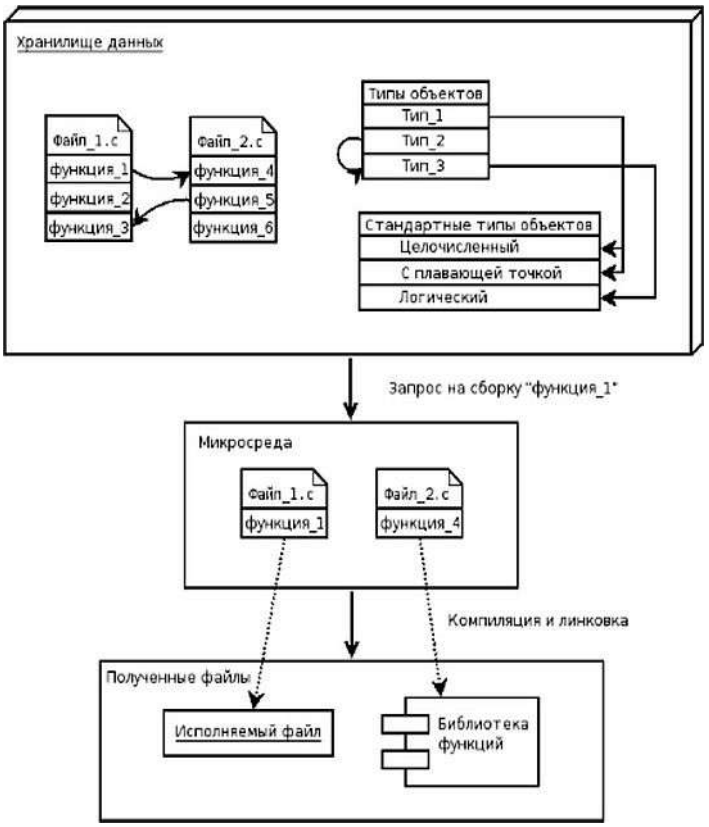


Рисунок 5 – Процесс сборки одной функции

Диаграмма развѐртывания

Корпоративные приложения часто требуют для своей работы некоторой ИТ-инфраструктуры, хранят информацию в базах данных, расположенных где-то на серверах компании, вызывают веб-сервисы, используют общие ресурсы и т. д. В таких случаях полезно иметь графическое представление инфраструктуры, на которую будет развернуто приложение. Для этого и нужны диаграммы развертывания, которые иногда называют диаграммами размещения.

Рисунок 10 – Диаграмм развертывания

Такие диаграммы есть смысл строить только для аппаратно-программных систем, тогда как UML позволяет строить модели любых систем, не обязательно компьютерных.

Польза диаграмм развертывания

Графическое представление ИТ-инфраструктуры может помочь более рационально распределить компоненты системы по узлам сети, от чего зависит в том числе и производительность системы.

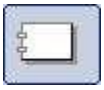
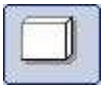
Такая диаграмма может помочь решить множество вспомогательных задач, связанных, например, с обеспечением безопасности.

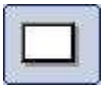
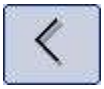
Диаграмма развертывания показывает топологию системы и распределение компонентов системы по ее узлам, а также соединения - маршруты передачи информации между аппаратными узлами. Это единственная диаграмма, на которой применяются ―трехмерные‖ обозначения: узлы системы обозначаются кубиками. Все остальные обозначения в UML - плоские фигуры.

Основные элементы диаграммы развѐртывания

Элементы нотации UML которые можно отобразить на диаграмме развѐртывания, представлены в таблице 1.

Таблиц 1 – Элементы нотации UML для диаграмм развѐртывания

Элемент/Нотация	Предназначение
	Компонент (Component)
	Экземпляр компонента (Component instance)
	Интерфейс (Interface)
	Узел (Node)
	Экземпляр узла (Node instance)
	Объект (Object)

Элемент/Нотация	Предназначение
	Активный объект (Active object)
	Зависимость (Dependency)
	Связь (Connection)
	Точка изгиба связей (Point)
	Комментарий (Note)
	Коннектор комментария (Note connector)

Объединение диаграммы развертывания и диаграммы компонентов

В некоторых случаях допускается размещать диаграмму компонентов на диаграмме развертывания. Это позволяет показать какие компоненты выполняются и на каких узлах.

Форма представления результата:

1. Цель работы
2. Введение
3. Программно-аппаратные средства, используемые при выполнении работы.
4. Описание работы
5. Заключение (выводы)
6. Список используемой литературы

Критерии оценки:

Работа выполнена полностью и не содержит ошибок, студент грамотно представил отчет – оценка «отлично».

Работа выполнена полностью, но содержит не более двух ошибок, студент грамотно представил отчет – оценка «хорошо».

Работа выполнена с ошибками, студент представил краткий отчет – оценка «удовлетворительно».

Работа выполнена с грубыми ошибками, отчет составлен неграмотно – оценка «неудовлетворительно».

Лабораторное занятие № 10
Изучение работы в системе контроля версий git

Цель: изучить основные принципы работы в системе контроля версий.

Выполнив работу, Вы будете:
уметь:

- У1 использовать выбранную систему контроля версий;
- У2 использовать методы для получения кода с заданной функциональностью и степенью качества;
- У3 анализировать проектную и техническую документацию;
- У4 использовать специализированные графические средства построения и анализа архитектуры программных продуктов;
- У5 организовывать заданную интеграцию модулей в программные средства на базе имеющейся архитектуры и автоматизации бизнес-процессов;
- У6 определять источники и приемники данных;
- У7 использовать приемы работы в системах контроля версий;
- У8 выполнять отладку, используя методы и инструменты условной компиляции (классы Debug и Trace);
- У12 использовать различные транспортные протоколы и стандарты форматирования сообщений.

Материальное обеспечение:
Пакет Microsoft Office, доступ к Internet ресурсам, Visual Studio, система контроля версий git, GUI-клиент git.

Задание:
Изучите и настройте систему контроля версий при выполнении проекта в среде Visual Studio для разрабатываемого программного продукта (лабораторное занятие № 10).

Порядок выполнения работы:

1. Изучить предлагаемый теоретический материал по теме.
2. Настройте систему контроля версий для разрабатываемого программного продукта, используя средства системы контроля верси git,.
3. Настройте систему контроля версий для разрабатываемого программного продукта, используя GUI-клиент git.
4. Настройте систему контроля версий для разрабатываемого программного продукта при выполнении проекта в среде Visual Studio.
5. Выбрать модули для рефакторинга кода, провести рефакторинг кода.
6. Сформировать отчет о работе.

Ход работы:

Краткие теоретические сведения

Начиная с Visual Studio 2013 Update 1, пользователям Visual Studio доступен Git-клиент, встроенный непосредственно в IDE. Visual Studio уже в течение достаточно долгого времени имеет встроенные функции управления исходным кодом, но они были ориентированы на централизованные системы с блокировкой файлов, и Git не очень хорошо вписывался в такой рабочей процесс. Поддержка Git в Visual Studio 2013 была существенно переработана по сравнению со старой версией, и в результате удалось добиться лучшей интеграции Visual Studio и Git.

Чтобы воспользоваться этой функциональностью, откройте проект, который управляется Git (или выполните git init для существующего проекта) и выберите пункты View (Вид) > Team Explorer (Командный обозреватель) в главном меню. В результате откроется окно "Connect" ("Подключить"), которое выглядит примерно вот так:

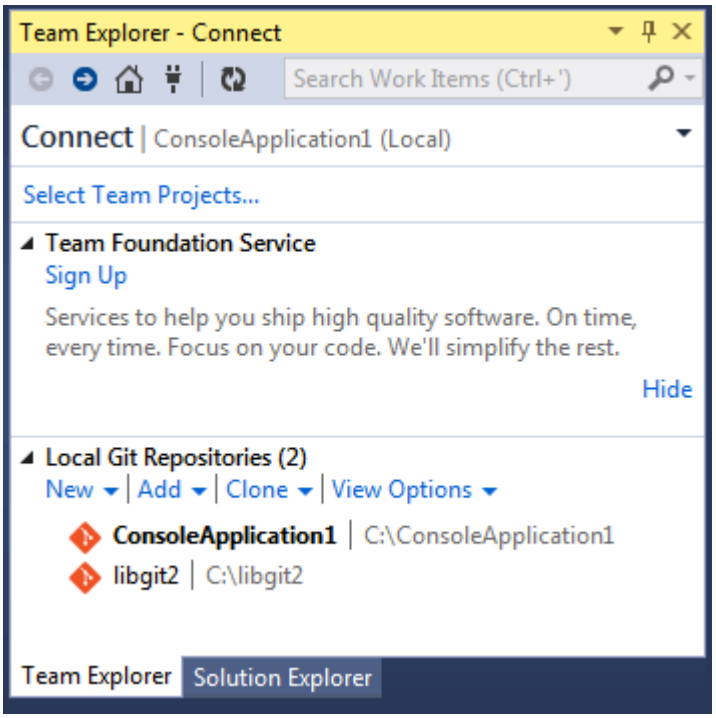


Рисунок 1 – Подключение к Git-репозиторию из окна Team Explorer (Командный обозреватель)

Visual Studio запоминает все проекты, управляемые с помощью Git, которые Вы открыли, и они доступны в списке в нижней части окна. Если в списке нет проекта, который вам нужен, нажмите кнопку "Add" ("Добавить") и укажите путь к рабочей директории. Двойной клик по одному из локальных Git-репозиториев откроет главную страницу репозитория, которая выглядит примерно так "Home" ("Главная") страница Git-репозитория в Visual Studio..

Это центр управления Git; когда вы пишете код, вы, вероятно, проводите большую часть своего времени на странице "Changes" ("Изменения"), но когда приходит время получать изменения, сделанные вашими коллегами по работе, вам необходимо использовать страницы "Unsynced Commits" ("Несинхронизированные коммиты") и "Branches" ("Ветви").

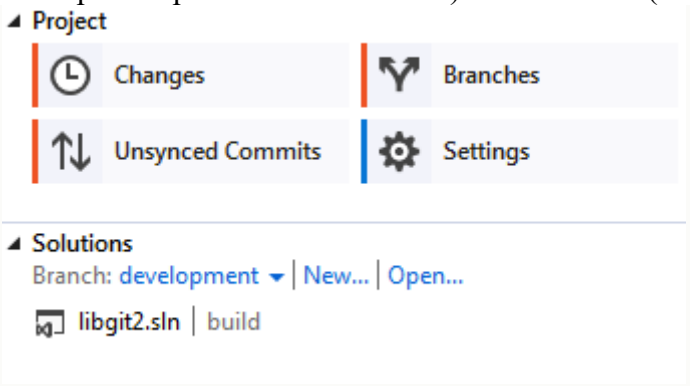


Рисунок 2 – «Home» («Главная») страница Git-репозитория в Visual Studio

В настоящее время Visual Studio имеет мощный задача-ориентированый графический интерфейс для Git. Он включает в себя возможность линейного представления истории, различные средства просмотра, средства выполнения удалѐнных команд и множество других возможностей.

Для просмотра полной документации по данной функциональности (которая здесь не представлена), перейдите на http://msdn.microsoft.com/en-us/library/hh850437.aspx.

Форма представления результата:

Подготовить отчет с линейным представлением истории,

Перечислить различные средства просмотра,

Перечислить средства выполнения удалѐнных команд, с которыми Вы работали при выполнении проекта.

Критерии оценки:

Работа выполнена полностью и не содержит ошибок, студент грамотно представил отчет – оценка «отлично».

Работа выполнена полностью, но содержит не более двух ошибок, студент грамотно представил отчет – оценка «хорошо».

Работа выполнена с ошибками, студент представил краткий отчет – оценка «удовлетворительно».

Работа выполнена с грубыми ошибками, отчет составлен неграмотно – оценка «неудовлетворительно».

Лабораторное занятие № 11
Разработка программного обеспечения с использованием различных паттернов проектирования

Цель: изучить основные паттерны и антипаттерны проектирования программного обеспечения.

Выполнив работу, Вы будете:

уметь:

- У1 использовать выбранную систему контроля версий;
- У2 использовать методы для получения кода с заданной функциональностью и степенью качества;
- У3 анализировать проектную и техническую документацию;
- У4 использовать специализированные графические средства построения и анализа архитектуры программных продуктов;
- У5 организовывать заданную интеграцию модулей в программные средства на базе имеющейся архитектуры и автоматизации бизнес-процессов;
- У6 определять источники и приемники данных;
- У7 использовать приемы работы в системах контроля версий;
- У8 выполнять отладку, используя методы и инструменты условной компиляции (классы Debug и Trace);
- У12 использовать различные транспортные протоколы и стандарты форматирования сообщений.

Материальное обеспечение:

Пакет Microsoft Office, доступ к Internet ресурсам, Visual Studio.

Задание:

Ознакомьтесь с паттернами проектирования и выберите 2 – 3 наиболее подходящих паттернов для разработки программного продукта для заданной предметной области (лабораторное занятие № 1). Обоснуйте свой выбор.

Разработайте обобщенный функционал вашего программного продукта, используя выбранные паттерны проектирования.

Проверьте, не используете ли вы антипаттерны проектирования. Аргументируйте свой ответ.

Порядок выполнения работы:

1. Изучить предлагаемый теоретический материал по теме.
2. Выбрать 2 – 3 наиболее подходящих паттернов для разработки программного продукта для заданной предметной области. Обосновать выбор.
3. Разработать обобщенный функционал программного продукта, используя выбранные паттерны проектирования.
4. Проверить, не используются ли антипаттерны проектирования. Аргументировать ответ.
5. Сформировать отчет о работе.

Ход работы:

Форма представления результата:

1. Цель работы
2. Введение
3. Программно-аппаратные средства, используемые при выполнении работы.
4. Описание работы
5. Заключение (выводы)

6. Список используемой литературы

Критерии оценки:

Работа выполнена полностью и не содержит ошибок, студент грамотно представил отчет – оценка «отлично».

Работа выполнена полностью, но содержит не более двух ошибок, студент грамотно представил отчет – оценка «хорошо».

Работа выполнена с ошибками, студент представил краткий отчет – оценка «удовлетворительно».

Работа выполнена с грубыми ошибками, отчет составлен неграмотно – оценка «неудовлетворительно».

Тема 2.1.4. Обеспечение качества программного обеспечения

Лабораторное занятие № 12
Инспекция программного кода на предмет соответствия стандартам кодирования. Рефакторинг кода

Цель: провести инспекцию программного кода на предмет соответствия стандартам кодирования, провести рефакторинг кода.

Выполнив работу, Вы будете:

уметь:

- У1 использовать выбранную систему контроля версий;
- У2 использовать методы для получения кода с заданной функциональностью и степенью качества;
- У3 анализировать проектную и техническую документацию;
- У4 использовать специализированные графические средства построения и анализа архитектуры программных продуктов;
- У11 выявлять ошибки в системных компонентах на основе спецификаций;
- У12 использовать различные транспортные протоколы и стандарты форматирования сообщений;
- У13 выполнять тестирование интеграции;
- У14 организовывать постобработку данных;
- У15 создавать классы- исключения на основе базовых классов;
- У16 выполнять ручное и автоматизированное тестирование программного модуля;
- У17 использовать инструментальные средства отладки программных продуктов.

Материальное обеспечение:

Пакет Microsoft Office, доступ к Internet ресурсам, Visual Studio.

Задание:

Проведите инспекцию программного кода разработанного программного продукта (лабораторное задание № 11) на предмет соответствия стандартам кодирования, осуществите рефакторинг кода.

Порядок выполнения работы:

1. Изучить предлагаемый теоретический материал по теме.
2. Провести инспекцию программного кода на предмет соответствия стандартам кодирования.
3. Выбрать модули для рефакторинга кода, провести рефакторинг кода.
4. Сформировать отчет о работе.

Ход работы:

Краткие теоретические сведения

Формальные инспекции программного кода

Процесс формальной инспекции программного кода подчиняется всем правилам, определенным для абстрактной формальной инспекции, однако, имеет некоторые особенности, связанные, в первую очередь со структурой инспектируемого программного кода, а также с тем, что обычно инспектируются участки кода, которые невозможно проверить при помощи автоматизированного тестирования, основанного на тестовых примерах.

Особенности этапа просмотра инспектируемого кода

Выделение ключевых точек и построение или использование таблиц трассировки. 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, хранящемся в базе данных проекта, 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.

При помощи трассировочных таблиц в исходном коде определяются инспектируемые функции или методы, соответствующие необходимым требованиям.

Участки кода выделяются и отмечаются меткой или номером соответствующего требования. Если участок кода соответствует требованиям, то необходимо отметить этот факт либо цветом выделения, либо соответствующим текстовым примечанием. Если участок кода имеет проблемы, этот факт должен быть отражен либо цветом выделения, либо ссылкой на соответствующий пункт списка замечаний в бланке инспекции.

В случае отсутствия трассировочных таблиц требований на исходный код рекомендуется делать пометки, поясняющие, почему именно данный участок кода реализует указанные требования. Такие пометки помогут на этапе обсуждения документа.

Проверка стиля кодирования. Отдельным объектом проверки при формальной инспекции программного кода является стиль кодирования. В большинстве проектов существуют стандарты, описывающие правила оформления исходных текстов программ и файлов данных. Неверный стиль кодирования не влияет на работоспособность программы в целом, но значительно затрудняет сопровождение и поддержку изменений в ходе дальнейшего развития системы. Поэтому отклонения от стиля кодирования в инспектируемых участках кода также должны отмечаться в тексте и в списке замечаний.

В некоторых случаях проводят инспекции, целиком направленные на проверку стиля кодирования.

Проверка надежности кода. В некоторых случаях рекомендуется 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.

Особенности этапа проведения собрания

Распределение ролей. В составе инспекторов желательно иметь хотя бы одного специалиста, представляющего себе особенности выполнения инспектируемого кода в реальной установке системы. Это особенно важно при тестировании встроенных систем, тестирование которых проводится на эмуляторах. Во время собрания такой специалист может помогать ведущему определять последовательность рассмотрения замечаний в случае их большого количества.

Управление собранием. При проведении собрания нецелесообразно зачитывать текст инспектируемого файла, как это обычно рекомендуется. Вместо этого ведущему лучше ограничиться перечислением имен функций или методов, либо, в случае, если в ходе инспекции проверяется соответствие исходного кода требованиям – перечислением номеров или идентификаторов требований. Инспектора при наличии замечаний по функции или требованию поднимают руку и зачитывают замечания.

Особенности этапа завершения и повторной инспекции

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

Контроль за внесением изменений. При повторной инспекции исходных текстов рекомендуется использовать специализированные инструментальные средства для сравнения файлов. Изменения по итогам инспекции должны вноситься только в те участки, к которым были высказаны замечания. В случае наличия других изменений ведущий вправе назначить новую инспекцию в полной форме.

Формальные инспекции проектной документации

Процесс формальной инспекции проектной документации подчиняется всем правилам, определенным для абстрактной формальной инспекции, однако, имеет некоторые особенности, связанные, в первую очередь, с тем, что у проектной документации проверяется ее непротиворечивость и полнота. Точное определение этих терминов в рассматриваемом контексте затруднительно, поэтому под непротиворечивостью будем понимать отсутствие в проектной документации требований, с противоположным смыслом, согласно которым возможно несколько совершенно различных вариантов реализации программной системы. Под полнотой будем понимать достаточность требований для однозначной реализации поведения системы.

Особенности этапа просмотра документации

При инспектировании требований к системе, как правило, рассматривается как «внешняя», так и «внутренняя» информация, касающаяся данного документа. Под внешней информацией понимается в первую очередь суть технических решений, принятых при разработке системы, те принципы, которые отличают ее от других систем.

При этом проверяется согласованность требований с этими принципами. Под внутренней информацией понимается прежде всего внутренняя целостность и непротиворечивость документа – свойства, которые позволяют разрабатывать программный код, лишенный двусмысленностей и нестабильных участков. Главный вопрос, на который должен ответить инспектор при проверке внутренних свойств документа: "Определены ли требования так, чтобы коллектив разработчиков смог работать с ним?" или иначе "Эти требования недвусмысленны, полны и авторитетно выражены?". Процесс инспекции может помочь ответить на эти вопросы, а список контрольных вопросов, представленный ниже, может быть использован при проведении инспекций:

1. Является ли каждое требование совершенно недвусмысленным? (Если требование прочесть подряд несколько раз, делая ударение сначала на первом слове, потом - на втором, затем - на третьем и т.д., будет ли при этом меняться смысл этого требования?)
2. Существует ли для каждого из установленных требований некоторый компетентный специалист, который сможет сказать после завершения разработки, выполнено ли данное требование или нет? Определен ли метод решения этой проблемы в документации по требованиям?
3. Существует ли какие-либо не установленные или отсутствующие требования?
4. Существуют ли среди заданных такие требования, которые не являются необходимыми?
5. Если существуют какие-либо конфликтующие требования, известно ли понятное решающее правило, в каких ситуациях какому требованию следует отдавать предпочтение?

Особенности этапа завершения

Влияние несогласованности документации на процесс разработки. Трассировка изменений на программный код. Первичная инспекция проектной документации, как правило, проводится тогда, когда сама программная система еще не написана. Однако при проведении инспекции изменений в требованиях к уже работающей системе (например, при обновлении ее версии), может потребоваться комплексная одновременная инспекция документации и созданного на ее основе программного кода. При этом может возникнуть ситуация, когда изменения, которые требуется внести в документацию по результатам инспекции, требуют соответствующего изменения в программном коде. Решение этой проблемы достигается использованием трассировочных таблиц.

Несколько иная ситуация возникает в случае, когда комплексная инспекция проводится не после изменения требований, а после завершения всей цепочки изменений – после изменения функциональных требований, архитектуры, низкоуровневых требований и программного кода. В этом случае при обнаружении противоречивых требований необходимо выявить все части программной системы, которые реализуют эти требования. В случае, если разработка этих частей выполнялась разными людьми, могла различаться и трактовка противоречивых требований. В этом случае ликвидация противоречивости может повлечь за собой «волну изменений» в проектной документации и исходных текстах системы. Для того, чтобы избежать «волн изменений» по завершению инспекций рекомендуется проводить ее поэтапно до начала

следующего этапа жизненного цикла или до разработки документов следующего уровня детализации.

Форма представления результата:

1. Цель работы
2. Введение
3. Программно-аппаратные средства, используемые при выполнении работы.
4. Описание работы
5. Заключение (выводы)
6. Список используемой литературы

Критерии оценки:

Работа выполнена полностью и не содержит ошибок, студент грамотно представил отчет – оценка «отлично».

Работа выполнена полностью, но содержит не более двух ошибок, студент грамотно представил отчет – оценка «хорошо».

Работа выполнена с ошибками, студент представил краткий отчет – оценка «удовлетворительно».

Работа выполнена с грубыми ошибками, отчет составлен неграмотно – оценка «неудовлетворительно».

Лабораторное занятие № 13
Статический и динамический анализ кода

Цель: провести статический и динамический анализ кода программного обеспечения, провести рефакторинг кода.

Выполнив работу, Вы будете:

уметь:

- У1 использовать выбранную систему контроля версий;
- У2 использовать методы для получения кода с заданной функциональностью и степенью качества;
- У3 анализировать проектную и техническую документацию;
- У4 использовать специализированные графические средства построения и анализа архитектуры программных продуктов;
- У11 выявлять ошибки в системных компонентах на основе спецификаций;
- У12 использовать различные транспортные протоколы и стандарты форматирования сообщений;
- У13 выполнять тестирование интеграции;
- У14 организовывать постобработку данных;
- У15 создавать классы- исключения на основе базовых классов;
- У16 выполнять ручное и автоматизированное тестирование программного модуля;
- У17 использовать инструментальные средства отладки программных продуктов.

Материальное обеспечение:

Пакет Microsoft Office, доступ к Internet ресурсам, Visual Studio.

Задание:

Проведите статический и динамический кода разработанного программного продукта (лабораторное задание № 11), осуществите рефакторинг кода при необходимости.

Порядок выполнения работы:

1. Изучить предлагаемый теоретический материал по теме.
2. Провести статический и динамический анализ кода.
3. Выбрать модули для рефакторинга кода, провести рефакторинг кода.
4. Сформировать отчет о работе.

Ход работы:

Краткие теоретические сведения

Формальные инспекции программного кода

Процесс формальной инспекции программного кода подчиняется всем правилам, определенным для абстрактной формальной инспекции, однако, имеет некоторые особенности, связанные, в первую очередь со структурой инспектируемого программного кода, а также с тем, что обычно инспектируются участки кода, которые невозможно проверить при помощи автоматизированного тестирования, основанного на тестовых примерах.

Особенности этапа просмотра инспектируемого кода

Выделение ключевых точек и построение или использование таблиц трассировки. 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, хранящемся в базе данных проекта, 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.

При помощи трассировочных таблиц в исходном коде определяются инспектируемые функции или методы, соответствующие необходимым требованиям.

Участки кода выделяются и отмечаются меткой или номером соответствующего требования. Если участок кода соответствует требованиям, то необходимо отметить этот факт либо цветом выделения, либо соответствующим текстовым примечанием. Если участок кода имеет проблемы, этот факт должен быть отражен либо цветом выделения, либо ссылкой на соответствующий пункт списка замечаний в бланке инспекции.

В случае отсутствия трассировочных таблиц требований на исходный код рекомендуется делать пометки, поясняющие, почему именно данный участок кода реализует указанные требования. Такие пометки помогут на этапе обсуждения документа.

Проверка стиля кодирования. Отдельным объектом проверки при формальной инспекции программного кода является стиль кодирования. В большинстве проектов существуют стандарты, описывающие правила оформления исходных текстов программ и файлов данных. Неверный стиль кодирования не влияет на работоспособность программы в целом, но значительно затрудняет сопровождение и поддержку изменений в ходе дальнейшего развития системы. Поэтому отклонения от стиля кодирования в инспектируемых участках кода также должны отмечаться в тексте и в списке замечаний.

В некоторых случаях проводят инспекции, целиком направленные на проверку стиля кодирования.

Проверка надежности кода. В некоторых случаях рекомендуется 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.

Особенности этапа проведения собрания

Распределение ролей. В составе инспекторов желательно иметь хотя бы одного специалиста, представляющего себе особенности выполнения инспектируемого кода в реальной установке системы. Это особенно важно при тестировании встроенных систем, тестирование которых проводится на эмуляторах. Во время собрания такой специалист может помогать ведущему определять последовательность рассмотрения замечаний в случае их большого количества.

Управление собранием. При проведении собрания нецелесообразно зачитывать текст инспектируемого файла, как это обычно рекомендуется. Вместо этого ведущему лучше ограничиться перечислением имен функций или методов, либо, в случае, если в ходе инспекции проверяется соответствие исходного кода требованиям – перечислением номеров или идентификаторов требований. Инспектора при наличии замечаний по функции или требованию поднимают руку и зачитывают замечания.

Особенности этапа завершения и повторной инспекции

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

Контроль за внесением изменений. При повторной инспекции исходных текстов рекомендуется использовать специализированные инструментальные средства для сравнения файлов. Изменения по итогам инспекции должны вноситься только в те участки, к которым были высказаны замечания. В случае наличия других изменений ведущий вправе назначить новую инспекцию в полной форме.

Формальные инспекции проектной документации

Процесс формальной инспекции проектной документации подчиняется всем правилам, определенным для абстрактной формальной инспекции, однако, имеет некоторые особенности, связанные, в первую очередь, с тем, что у проектной документации проверяется ее непротиворечивость и полнота. Точное определение этих терминов в рассматриваемом контексте

затруднительно, поэтому под непротиворечивостью будем понимать отсутствие в проектной документации требований, с противоположным смыслом, согласно которым возможно несколько совершенно различных вариантов реализации программной системы. Под полнотой будем понимать достаточность требований для однозначной реализации поведения системы.

Особенности этапа просмотра документации

При инспектировании требований к системе, как правило, рассматривается как «внешняя», так и «внутренняя» информация, касающаяся данного документа. Под внешней информацией понимается в первую очередь суть технических решений, принятых при разработке системы, те принципы, которые отличают ее от других систем.

При этом проверяется согласованность требований с этими принципами. Под внутренней информацией понимается прежде всего внутренняя целостность и непротиворечивость документа – свойства, которые позволяют разрабатывать программный код, лишенный двусмысленностей и нестабильных участков. Главный вопрос, на который должен ответить инспектор при проверке внутренних свойств документа: "Определены ли требования так, чтобы коллектив разработчиков смог работать с ним?" или иначе "Эти требования недвусмысленны, полны и авторитетно выражены?". Процесс инспекции может помочь ответить на эти вопросы, а список контрольных вопросов, представленный ниже, может быть использован при проведении инспекций:

1. Является ли каждое требование совершенно недвусмысленным? (Если требование прочесть подряд несколько раз, делая ударение сначала на первом слове, потом - на втором, затем - на третьем и т.д., будет ли при этом меняться смысл этого требования?)
2. Существует ли для каждого из установленных требований некоторый компетентный специалист, который сможет сказать после завершения разработки, выполнено ли данное требование или нет? Определен ли метод решения этой проблемы в документации по требованиям?
3. Существует ли какие-либо не установленные или отсутствующие требования?
4. Существуют ли среди заданных такие требования, которые не являются необходимыми?
5. Если существуют какие-либо конфликтующие требования, известно ли понятное решающее правило, в каких ситуациях какому требованию следует отдавать предпочтение?

Особенности этапа завершения

Влияние несогласованности документации на процесс разработки. Трассировка изменений на программный код. Первичная инспекция проектной документации, как правило, проводится тогда, когда сама программная система еще не написана. Однако при проведении инспекции изменений в требованиях к уже работающей системе (например, при обновлении ее версии), может потребоваться комплексная одновременная инспекция документации и созданного на ее основе программного кода. При этом может возникнуть ситуация, когда изменения, которые требуется внести в документацию по результатам инспекции, требуют соответствующего изменения в программном коде. Решение этой проблемы достигается использованием трассировочных таблиц.

Несколько иная ситуация возникает в случае, когда комплексная инспекция проводится не после изменения требований, а после завершения всей цепочки изменений – после изменения функциональных требований, архитектуры, низкоуровневых требований и программного кода. В этом случае при обнаружении противоречивых требований необходимо выявить все части программной системы, которые реализуют эти требования. В случае, если разработка этих частей выполнялась разными людьми, могла различаться и трактовка противоречивых требований. В этом случае ликвидация противоречивости может повлечь за собой «волну изменений» в проектной документации и исходных текстах системы. Для того, чтобы избежать «волн изменений» по завершению инспекций рекомендуется проводить ее поэтапно до начала следующего этапа жизненного цикла или до разработки документов следующего уровня детализации.

Форма представления результата:

7. Цель работы

8. Введение
9. Программно-аппаратные средства, используемые при выполнении работы.
10. Описание работы
11. Заключение (выводы)
12. Список используемой литературы

Критерии оценки:

Работа выполнена полностью и не содержит ошибок, студент грамотно представил отчет – оценка «отлично».

Работа выполнена полностью, но содержит не более двух ошибок, студент грамотно представил отчет – оценка «хорошо».

Работа выполнена с ошибками, студент представил краткий отчет – оценка «удовлетворительно».

Работа выполнена с грубыми ошибками, отчет составлен неграмотно – оценка «неудовлетворительно».

Лабораторное занятие № 14
Построение тестовой модели программного обеспечения.
Формирование метрик для оценки качества программного обеспечения.
Составление тест-планов, формирование тестовой документации

Цель: изучить основные принципы разработки тестовой модели программного обеспечения; сформировать метрики для оценки качества программного продукта; составить тест-план; сформировать тестовую документацию.

Выполнив работу, Вы будете:

уметь:

- У1 использовать выбранную систему контроля версий;
- У2 использовать методы для получения кода с заданной функциональностью и степенью качества;
- У3 анализировать проектную и техническую документацию;
- У4 использовать специализированные графические средства построения и анализа архитектуры программных продуктов;
- У11 выявлять ошибки в системных компонентах на основе спецификаций;
- У12 использовать различные транспортные протоколы и стандарты форматирования сообщений;
- У13 выполнять тестирование интеграции;
- У14 организовывать постобработку данных;
- У15 создавать классы- исключения на основе базовых классов;
- У16 выполнять ручное и автоматизированное тестирование программного модуля;
- У17 использовать инструментальные средства отладки программных продуктов.

Материальное обеспечение:

Пакет Microsoft Office, доступ к Internet ресурсам, Visual Studio.

Задание:

Изучите основные принципы разработки тестовой модели программного обеспечения и постройте тестовую модель для разработанного программного продукта (лабораторное задание № 11); сформируйте метрики для оценки качества программного продукта; составьте тест-план; сформируйте тестовую документацию.

Порядок выполнения работы:

1. Изучить предлагаемый теоретический материал по теме.
2. Построить тестовую модель для разработанного программного продукта (лабораторное задание № 11).
3. Сформировать метрику для оценки качества разработанного программного продукта.
4. Составить тест-план для разработанного программного продукта.
5. Сформировать тестовую документацию для разработанного программного продукта.
6. Разработать диаграмму вариантов использования и диаграмму состояний и последовательности с использованием языка UML для заданной предметной области в соответствии с требованиями.
7. Сформировать отчет о работе.

Ход работы:

Краткие теоретические сведения

Для разработки тестовых сценариев и выполнения тестов используются системы управления тестированием, существенно повышающие производительность тест-дизайнеров и тестировщиков, а также обеспечивающие видимость уровня качества приложений среди всех участников проекта.

Тестовые сценарии неразрывно связаны с требованиями, изменения в которых должны своевременно отражаться в тестовой документации, что позволяет сделать система управления жизненным циклом разработки приложений, при помощи механизма трассировок.

При выполнении теста тестировщик отмечает результат прохождения одного шага или всего тестового сценария, прикрепляет обнаруженные ошибки и другую вспомогательную информацию: скриншоты, дампы, логи и т.п.

Тестовые сценарии удобно объединять в тест-планы по назначению:

- тестирование релиза, то есть очередной версии продукта;
- тестирование развертывания;
- тестирование удобства использования;
- конфигурационное тестирование;
- тестирование безопасности и т.п.

Зачастую ручное тестирование превращается в рутину и занимает значительное время, что отрицательно сказывается на скорости выпуска релизов. Автоматизация тестирования позволяет:

- высвободить ресурсы для проведения более сложных видов тестирования;
- снизить количество дефектов, доходящих до стадии контроля качества;
- ускорить выпуск релизов.

Сведение результатов автоматических и ручных тестов в системе управления качеством, позволяет всем участникам проекта видеть уровень качества очередного релиза, контролировать его изменение и опираться на эту информацию при планировании своей работы.

Получение результатов тестирования и их анализ

Получение результатов тестирования напрямую зависит от средств тестирования. В моем случае это был встроенный в 1С механизм отладки и система контроля ошибок.

Перед получением результата программа проходит несколько уровней:

1. Тестирование компонентов — тестируется минимально возможный для тестирования компонент, например, отдельный класс или функция. Часто тестирование компонентов осуществляется разработчиками программного обеспечения.
2. Интеграционное тестирование — тестируются интерфейсы между компонентами, подсистемами или системами. При наличии резерва времени на данной стадии тестирование ведѐтся итерационно, с постепенным подключением последующих подсистем.
3. Системное тестирование — тестируется интегрированная система на еѐ соответствие требованиям.
4. Альфа-тестирование — имитация реальной работы с системой штатными разработчиками, либо реальная работа с системой потенциальными пользователями/заказчиком. Чаще всего альфа-тестирование проводится на ранней стадии разработки продукта, но в некоторых случаях может применяться для законченного продукта в качестве внутреннего приѐмочного тестирования. Иногда альфа-тестирование выполняется под отладчиком или с использованием окружения, которое помогает быстро выявлять найденные ошибки. Обнаруженные ошибки могут быть переданы тестировщикам для дополнительного исследования в окружении, подобном тому, в котором будет использоваться программа.
5. Бета-тестирование — в некоторых случаях выполняется распространение предварительной версии (в случае проприетарного программного обеспечения иногда с ограничениями по функциональности или времени работы) для некоторой большей группы лиц с тем, чтобы убедиться, что продукт содержит достаточно мало ошибок. Иногда бета-тестирование выполняется для того, чтобы получить обратную связь о продукте от его будущих пользователей.

Часто для свободного и открытого программного обеспечения стадия альфа-тестирования характеризует функциональное наполнение кода, а бета-тестирования — стадию исправления ошибок. При этом как правило на каждом этапе разработки промежуточные результаты работы доступны конечным пользователям.

Метрики для оценки качества программного продукта

В настоящее время в программной инженерии еще не сформировалась окончательно система метрик. Действуют разные подходы к определению их набора и методов измерения [10.11-10.13].

Система измерения включает метрики и модели измерений, которые используются для количественной оценки качества ПО.

При определении требований к ПО задаются соответствующие им внешние характеристики и их атрибуты (подхарактеристики), определяющие разные стороны управления продуктом в заданной среде. Для набора характеристик качества ПО, приведенных в требованиях, определяются соответствующие метрики, модели их оценки и диапазон значений мер для измерения отдельных атрибутов качества.

Согласно стандарту [1.16] метрики определяются по модели измерения атрибутов ПО на всех этапах ЖЦ (промежуточная, внутренняя метрика) и особенно на этапе тестирования или функционирования (внешние метрики) продукта.

Остановимся на классификации метрик ПО, правилах для проведения метрического анализа и процесса их измерения.

Типы метрик.

Существует три типа метрик:

- метрики программного продукта, которые используются при измерении его характеристик - свойств;
- метрики процесса, которые используются при измерении свойства процесса ЖЦ создания продукта.

метрики использования.

Метрики программного продукта включают:

- внешние метрики, обозначающие свойства продукта, видимые пользователю;
- внутренние метрики, обозначающие свойства, видимые только команде разработчиков.

Внешние метрики продукта - это метрики:

- надежности продукта, которые служат для определения числа дефектов;
- функциональности, с помощью которых устанавливаются наличие и правильность реализации функций в продукте;
- сопровождения, с помощью которых измеряются ресурсы продукта (скорость, память, среда);применимости продукта, которые способствуют определению степени доступности для изучения и использования;
- стоимости, которыми определяется стоимость созданного продукта.

Внутренние метрики продукта включают:

- метрики размера, необходимые для измерения продукта с помощью его внутренних характеристик;
- метрики сложности, необходимые для определения сложности продукта;
- метрики стиля, которые служат для определения подходов и технологий создания отдельных компонентов продукта и его документов.

Внутренние метрики позволяют определить производительность продукта и являются релевантными по отношению к внешним метрикам.

Внешние и внутренние метрики задаются на этапе формирования требований к ПО и являются предметом планирования и управления достижением качества конечного программного продукта.

Метрики продукта часто описываются комплексом моделей для установки различных свойств, значений модели качества или прогнозирования. Измерения проводятся, как правило, после калибровки метрик на ранних этапах проекта. Общая мера - степень трассируемости, которая определяется числом трасс, прослеживаемых с помощью моделей сценариев типа UML и оценкой количества:

- требований;

- сценариев и действующих лиц;
- объектов, включенных в сценарий, и локализация требований к каждому сценарию;
- параметров и операций объекта и др.

Стандарт ISO/IEC 9126-2 определяет следующие типы мер:

мера размера ПО в разных единицах измерения (число функций, строк в программе, размер дисковой памяти и др.);

- мера времени (функционирования системы, выполнения компонента и др.);
- мера усилий (производительность труда, трудоемкость и др.);
- мера учета (количество ошибок, число отказов, ответов системы и др.).

Специальной мерой может служить уровень использования повторных компонентов и измеряется как отношение размера продукта, изготовленного из готовых компонентов, к размеру системы в целом. Данная мера используется также при определении стоимости и качества ПО. Примеры метрик:

- общее число объектов и число повторно используемых;
- общее число операций, повторно используемых и новых операций;
- число классов, наследующих специфические операции;
- число классов, от которых зависит данный класс;
- число пользователей класса или операций и др.

При оценке общего количества некоторых величин часто используются среднестатистические метрики (среднее число операций в классе, наследников класса или операций класса и др.).

Как правило, меры в значительной степени являются субъективными и зависят от знаний экспертов, производящих количественные оценки атрибутов компонентов программного продукта.

Примером широко используемых внешних метрик программ являются метрики Холстеда - это характеристики программ, выявляемые на основе статической структуры программы на конкретном языке программирования: число вхождений наиболее часто встречающихся операндов и операторов; длина описания программы как сумма числа вхождений всех операндов и операторов и др.

На основе этих атрибутов можно вычислить время программирования, уровень программы (структурированность и качество) и языка программирования (абстракции средств языка и ориентация на проблему) и др.

В качестве метрик процесса могут быть время разработки, число ошибок, найденных на этапе тестирования и др. Практически используются следующие метрики процесса:

- общее время разработки и отдельно время для каждой стадии;
- время модификации моделей;
- время выполнения работ на процессе;
- число найденных ошибок при инспектировании;
- стоимость проверки качества;
- стоимость процесса разработки.

Метрики использования служат для измерения степени удовлетворения потребностей пользователя при решении его задач. Они помогают оценить не свойства самой программы, а результаты ее эксплуатации - эксплуатационное качество. Примером может служить - точность и полнота реализации задач пользователя, а такжезатраченные ресурсы (трудозатраты, производительность и др.) на эффективное решение задач пользователя. Оценка требований пользователя проводится с помощью внешних метрик.

Стандартная оценка значений показателей качества

Оценка качества ПО согласно четырехуровневой модели качества начинается с нижнего уровня иерархии, т.е. с самого элементарного свойства оцениваемого атрибута показателя качества согласно установленных мер. На этапе проектирования устанавливают значения

оценочных элементов для каждого атрибута показателя анализируемого ПО, включенного в требования.

По определению стандарта ISO/IES 9126-2 метрика качества ПО представляет собой "модель измерения атрибута, связываемого с показателем его качества". При измерении показателей качества данный стандарт позволяет определять следующие типы мер:

- меры размера в разных единицах измерения (количество функций, размер программы, объем ресурсов и др.);
- меры времени - периоды реального, процессорного или календарного времени (время функционирования системы, время выполнения компонента, время использования и др.);
- меры усилий - продуктивное время, затраченное на реализацию проекта (производительность труда отдельных участников проекта, коллективная трудоемкость и др.);
- меры интервалов между событиями, например, время между последовательными отказами;
- счетные меры - счетчики для определения количества обнаруженных ошибок, структурной сложности программы, числа несовместимых элементов, числа изменений (например, число обнаруженных отказов и др.).

Метрики качества используются при оценке степени тестируемости с помощью данных (безотказная работа, выполнимость функций, удобство применения интерфейсов пользователей, БД и т.п.) после проведения испытаний ПО на множестве тестов.

Наработка на отказ как атрибут надежности определяет среднее время между появлением угроз, нарушающих безопасность, и обеспечивает трудноизмеримую оценку ущерба, которая наносится соответствующими угрозами.Очень часто оценка программы проводится по числу строк. При сопоставлении двух программ, реализующих одну прикладную задачу, предпочтение отдается короткой программе, так как еѐ создает более квалифицированный персонал и в ней меньше скрытых ошибок и легче модифицировать. По стоимости она дороже, хотя времени на отладку и модификацию уходит больше. Т.е. длину программы можно использовать в качестве вспомогательного свойства для сравнения программ с учетом одинаковой квалификации разработчиков, единого стиля разработки и общей среды.

Если в требованиях к ПО было указано получить несколько показателей, то просчитанный после сбора данных показатель умножается на соответствующий весовой коэффициент, а затем суммируются все показатели для получения комплексной оценки уровня качества ПО.

На основе измерения количественных характеристик и проведения экспертизы качественных показателей с применением весовых коэффициентов, нивелирующих разные показатели, вычисляется итоговая оценка качества продукта путем суммирования результатов по отдельным показателям и сравнения их с эталонными показателями ПО (стоимость, время, ресурсы и др.).

Т.е. при проведении оценки отдельного показателя с помощью оценочных элементов просчитывается весомый коэффициент k -метрика, j -показатель, i -атрибут. Например, в качестве j -показателя возьмем переносимость. Этот показатель будет вычисляться по пяти атрибутам ($i = 1, \ldots, 5$), причем каждый из них будет умножаться на соответствующий коэффициент k_i .

Все метрики j -атрибута суммируются и образуют i -показатель качества. Когда все атрибуты оценены по каждому из показателей качества, производится суммарная оценка отдельного показателя, а потом и интегральная оценка качества с учетом весовых коэффициентов всех показателей ПО.

В конечном итоге результат оценки качества является критерием эффективности и целесообразности применения методов проектирования, инструментальных средств и методик оценивания результатов создания программного продукта на стадиях ЖЦ.

Для изложения оценки значений показателей качества используется стандарт [10.4], в котором представлены следующие методы: измерительный, регистрационный, расчетный и экспертный (а также комбинации этих методов). Измерительный метод базируется на использовании измерительных и специальных программных средств для получения информации о характеристиках ПО, например, определение объема, числа строк кода, операторов, количества ветвей в программе, число точек входа (выхода), реактивность и др.

Регистрационный метод используется при подсчете времени, числа сбоев или отказов, начала и конца работы ПО в процессе его выполнения.

Расчетный метод базируется на статистических данных, собранных при проведении испытаний, эксплуатации и сопровождении ПО. Расчетными методами оцениваются показатели надежности, точности, устойчивости, реактивности и др.

Экспертный метод осуществляется группой экспертов - специалистов, компетентных в решении данной задачи или типа ПО. Их оценка базируется на опыте и интуиции, а не на непосредственных результатах расчетов или экспериментов. Этот метод проводится путем просмотра программ, кодов, сопроводительных документов и способствует качественной оценки созданного продукта. Для этого устанавливаются контролируемые признаки, которые коррелированны с одним или несколькими показателями качества и включены в опросные карты экспертов. Метод применяется при оценке таких показателей, как анализируемость, документируемость, структурированность ПО и др.

Для оценки значений показателей качества в зависимости от особенностей используемых ими свойств, назначения, способов их определения используются:

- шкала метрическая (1.1 - абсолютная, 1.2 - относительная, 1.3 - интегральная);
- шкала порядковая (ранговая), позволяющая ранжировать характеристики путем сравнения с опорными;
- классификационная шкала, характеризующая наличие или отсутствие рассматриваемого свойства у оцениваемого программного обеспечения.

Показатели, которые вычисляются с помощью метрических шкал, называются количественные, а определяемые с помощью порядковых и классификационных шкал - качественные.

Атрибуты программной системы, характеризующие ее качество, измеряются с использованием метрик качества. Метрика определяет меру атрибута, т.е. переменную, которой присваивается значение в результате измерения.Для правильного использования результатов измерений каждая мера идентифицируется шкалой измерений.

Стандарт ISO/IES 9126-2 рекомендует применять 5 видов шкал измерения значений, которые упорядочены от менее строгой к более строгой:

- номинальная шкала отражает категории свойств оцениваемого объекта без их упорядочения;
- порядковая шкала служит для упорядочивания характеристики по возрастанию или убыванию путем сравнения их с базовыми значениями;
- интервальная шкала задает существенные свойства объекта (например, календарная дата);
- относительная шкала задает некоторое значение относительно выбранной единицы;
- абсолютная шкала указывает на фактическое значение величины (например, число ошибок в программе равно 10).

Из всех областей программной инженерии надежность ПС является самой исследованной областью. Ей предшествовала разработка теории надежности технических средств, оказавшая влияние на развитие надежности ПС. Вопросами надежности ПС занимались разработчики ПС, пытаясь разными системными средствами обеспечить надежность, удовлетворяющую заказчика, а также теоретики, которые, изучая природу функционирования ПС, создали математические модели надежности, учитывающие разные аспекты работы ПС (возникновение ошибок, сбоев, отказов и др.) и позволяющие оценить реальную надежность. В результате надежность ПС сформировалась как самостоятельная теоретическая и прикладная наука [10.5-10.10, 10.16-10.24].

Надежность сложных ПС существенным образом отличается от надежности аппаратуры. Носители данных (файлы, сервер и т.п.) обладают высокой надежностью, записи на них могут храниться длительное время без разрушения, поскольку физическому разрушению они не подвергаются.

С точки зрения прикладной науки надежность - это способность ПС сохранять свои свойства (безотказность, устойчивость и др.), преобразовывать исходные данные в результаты в течение определенного промежутка времени при определенных условиях эксплуатации. Снижение надежности ПС происходит из-за ошибок в требованиях, проектировании и выполнении. Отказы и ошибки зависят от способа производства продукта и появляются в программах при их исполнении на некотором промежутке времени.

Для многих систем (программ и данных) надежность - главная целевая функция реализации. К некоторым типам систем (реального времени, радарные системы, системы безопасности, медицинскоеоборудование со встроенными программами и др.) предъявляются высокие требования к надежности, такие, как отсутствие ошибок, достоверность, безопасность и др.

Таким образом, оценка надежности ПС зависит от числа оставшихся и не устраненных ошибок в программах. В ходе эксплуатации ПС ошибки обнаруживаются и устраняются. Если при исправлении ошибок не вносятся новые или, по крайней мере, новых ошибок вносится меньше, чем устраняется, то в ходе эксплуатации надежность ПС непрерывно возрастает. Чем интенсивнее проводится эксплуатация, тем интенсивнее выявляются ошибки и быстрее растет надежность системы и соответственно ее качество.

Надежность является функцией от ошибок, оставшихся в ПС после ввода его в эксплуатацию. ПС без ошибок является абсолютно надежным. Но для больших программ абсолютная надежность практически недостижима. Оставшиеся необнаруженные ошибки проявляют себя время от времени при определенных условиях (например, при некоторой совокупности исходных данных) сопровождения и эксплуатации системы.

Для оценки надежности ПС используются такие статистические показатели, как вероятность и время безотказной работы, возможность отказа и частота (интенсивность) отказов. Поскольку в качестве причин отказов рассматриваются только ошибки в программе, которые не могут самоустраниться, то ПС следует относить к классу невосстанавливаемых систем.

При каждом проявлении новой ошибки, как правило, проводится ее локализация и исправление. Строго говоря, набранная до этого статистика об отказах теряет свое значение, так как после внесения изменений программа, по существу, является новой программой в отличие от той, которая до этого испытывалась.

В связи с исправлением ошибок в ПС надежность, т.е. ее отдельные атрибуты, будут все время изменяться, как правило, в сторону улучшения. Следовательно, их оценка будет носить временный и приближенный характер. Поэтому возникает необходимость в использовании новых свойств, адекватных реальному процессу измерения надежности, таких, как зависимость интенсивности обнаруженных ошибок от числа прогонов программы и зависимость отказов от времени функционирования ПС и т.п.

К факторам гарантии надежности относятся:

- риск как совокупность угроз, приводящих к неблагоприятным последствиям и ущербу системы или среды;
- угроза как проявление неустойчивости, нарушающей безопасность системы;
- анализ риска - изучение угрозы или риска, их частота и последствия;
- целостность - способность системы сохранять устойчивость работы и не иметь риска;

Риск преобразует и уменьшает свойства надежности, так как обнаруженные ошибки могут привести к угрозе, если отказы носят частотный характер.

Основные понятия в проблематике надежности ПС

Формально модели оценки надежности ПС базируются на теории надежности и математическом аппарате с допущением некоторых ограничений, влияющих на эту оценку. Главным источником информации, используемой в моделях надежности, является процесс тестирования, эксплуатации ПС и разного вида ситуации, возникающие в них. Ситуации

порождаются возникновением ошибок в ПС, требуют их устранения для продолжения тестирования.

Базовыми понятиями, которые используются в моделях надежности ПС, являются [10.5-10.10].

Отказ ПС (failure) - это переход ПС из работающего состояния в нерабочее или когда получаются результаты, которые не соответствуют заданным допустимым значениям. Отказ может быть вызван внешними факторами (изменениями элементов среды эксплуатации) и внутренними - дефектами в самой ПС.

Дефект (fault) в ПС - это последствие использования элемента программы, который может привести к некоторому событию, например, в результате неверной интерпретации этого элемента компьютером (как ошибка (fault) в программе) или человеком (ошибка (error) исполнителя). Дефект является следствием ошибок разработчика на любом из процессов разработки - в описании спецификаций требований, начальных или проектных спецификациях, эксплуатационной документации и т.п. Дефекты в программе, не выявленные в результате проверок, являются источником потенциальных ошибок и отказов ПС. Проявление дефекта в виде отказа зависит от того, какой путь будет выполнять специалист, чтобы найти ошибку в коде или во входных данных. Однако не каждый дефект ПС может вызвать отказ или может быть связан с дефектом в ПС или среды. Любой отказ может вызвать аномалию от проявления внешних ошибок и дефектов.

Ошибка (error) может быть следствием недостатка в одном из процессов разработки ПС, который приводит к неправильной интерпретации промежуточной информации, заданной разработчиком или при принятии им неверных решений.

Интенсивность отказов - это частота появления отказов или дефектов в ПС при ее тестировании или эксплуатации.

При выявлении отклонения результатов выполнения от ожидаемых во время тестирования или сопровождения осуществляется поиск, выяснение причин отклонений и исправление связанных с этим ошибок.

Модели оценки надежности ПС в качестве входных параметров используют сведения об ошибках, отказах, их интенсивности, собранных в процессе тестирования и эксплуатации.

Классификация моделей надежности

Как известно, на данный момент времени разработано большое количество моделей надежности ПС и их модификаций. Каждая из этих моделей определяет функцию надежности, которую можно вычислить при задании ей соответствующих данных, собранных во время функционирования ПС. Основными данными являются отказы и время. Другие дополнительные параметры связаны с типом ПС, условиями среды и данных.

Ввиду большого разнообразия моделей надежности разработано несколько подходов к классификации этих моделей. Такие подходы в целом основываются на истории ошибок в проверяемой и тестируемой ПС на этапах ЖЦ. Одной из классификаций моделей надежности ПО является классификация Хетча [10.10]. В ней предлагается разделение моделей на прогнозирующие, измерительные и оценочные (рисунок 1).

Прогнозирующие модели надежности основаны на измерении технических характеристик создаваемой программы: длина, сложность, число циклов и степень их вложенности, количество ошибок на страницу операторов программы и др.

Например, модель Мотли-Брукса основывается на длине и сложности структуры программы (количество ветвей, циклов, вложенность циклов), количестве и типах переменных, а также интерфейсов. В этих моделях длина программы служит для прогнозирования количества ошибок, например, для 100 операторов программы можно смоделировать интенсивность отказов.

Рисунок 1 – Классификация моделей надежности

Модель Холстеда прогнозирует количество ошибок в программе в зависимости от ее объема и таких данных, как число операций (n_1) и операндов (n_2), а также их общее число (N_1, N_2).

Время программирования программы предлагается вычислять по следующей формуле:

$$T = \frac{n_1 N_2 (n_1 \log_2 n_1 + n_2 \log_2 n_2) \log_2 n_1}{2 n_2 S},$$

где S - число Страуда (Холстед принял равным 18 - число умственных операций в единицу времени).

Объем вычисляется по формуле:

$$V = (2 + n_2^*) \log_2 (2 + n_2^*),$$

где n_2^* - максимальное число различных операций.

Измерительные модели предназначены для измерения надежности программного обеспечения, работающего с заданной внешней средой. Они имеют следующие ограничения:

программное обеспечение не модифицируется во время периода измерений свойств надежности;

обнаруженные ошибки не исправляются;

измерение надежности проводится для зафиксированной конфигурации программного обеспечения.

Типичным примером таких моделей являются модели Нельсона и РамамуртиБастани и др.Модель оценки надежности Нельсона основывается на выполнении k-прогонов программы при тестировании и позволяет определить надежность

$$R(k) = exp[-\sum \nabla t_j \lambda(t)],$$

где t_j - время выполнения j -прогона, $\lambda(t) = -[\ln (1 - q_i)]$ и при $q_i \leq 1$ она интерпретируется как интенсивность отказов.

В процессе испытаний программы на тестовых n_l прогонах оценка надежности вычисляется по формуле

$$R(l) = \frac{1 - n_l}{k},$$

где k - число прогонов программы.

Таким образом, данная модель рассматривает полученные количественные данные о проведенных прогонах.

Оценочные модели основываются на серии тестовых прогонов и проводятся на этапах тестирования ПС. В тестовой среде определяется вероятность отказа программы при ее выполнении или тестировании.

Эти типы моделей могут применяться на этапах ЖЦ. Кроме того, результаты прогнозирующих моделей могут использоваться как входные данные для оценочной модели. Имеются модели (например, модель Муссы), которые можно рассматривать как оценочную и в то же время как измерительную модель [10.16, 10.17].

Другой вид классификации моделей предложил Гоэл [10.18, 10.19], согласно которой модели надежности базируются на отказах и разбиваются на четыре класса моделей:

без подсчета ошибок;

с подсчетом отказов;

с подсевом ошибок;

модели с выбором областей входных значений.

Модели без подсчета ошибок основаны на измерении интервала времени между отказами и позволяют спрогнозировать количество ошибок, оставшихся в программе. После каждого отказа оценивается надежность и определяется среднее время до следующего отказа. К таким моделям относятся модели Джелински и Моранды, Шика Вулвертона и Литвуда-Вералла [10.20, 10.21].

Модели с подсчетом отказов базируются на количестве ошибок, обнаруженных на заданных интервалах времени. Возникновение отказов в зависимости от времени является стохастическим процессом с непрерывной интенсивностью, а количество отказов является случайной величиной. Обнаруженные ошибки, как правило, устраняются и поэтому количество ошибок в единицу времени уменьшается. К этому классу моделей относятся модели Шумана, Шика- Вулвертона, Пуассоновская модель и др. [10.21-10.24].

Модели с подсевом ошибок основаны на количестве устраненных ошибок и подсеве, внесенном в программу искусственных ошибок, тип и количество которых заранее известны. Затем определяется соотношение числа оставшихся прогнозируемых ошибок к числу искусственных ошибок, которое сравнивается с соотношением числа обнаруженных действительных ошибок к числу обнаруженных искусственных ошибок. Результат сравнения используется для оценки надежности и качества программы. При внесении изменений в программу проводится повторное тестирование и оценка надежности. Этот подход к организации тестирования отличается громоздкостью и редко используется из-за дополнительного объема работ, связанных с подбором, выполнением и устранением искусственных ошибок.

Модели с выбором области входных значений основываются на генерации множества тестовых выборок из входного распределения, и оценка надежности проводится по полученным отказам на основе тестовых выборок из входной области. К этому типу моделей относится модель Нельсона и др.

Таким образом, классификация моделей роста надежности относительно процесса выявления отказов, фактически разделена на две группы:

модели, которые рассматривают количество отказов как марковский процесс;

модели, которые рассматривают интенсивность отказов как пуассоновский процесс.

Фактор распределения интенсивности отказов разделяет модели на экспоненциальные, логарифмические, геометрические, байесовские и др.

Марковские и пуассоновские модели надежности

Марковский процесс характеризуется дискретным временем и конечным множеством состояний. Временной параметр пробегает неотрицательные числовые значения, а процесс (цепочка) определяется набором вероятностей перехода $p_{ij}(n)$, т.е. вероятностью перейти на n -шаге из состояния i в состояние j . Процесс называется однородным, если он не зависит от n. В моделях, базирующихся на процессе Маркова, предполагается, что количество дефектов, обнаруженных в ПС, в любой момент времени зависит от поведения системы и представляется в виде стационарной цепи Маркова [10.5, 10.7, 10.10]. При этом количество дефектов конечное, но

является неизвестной величиной, которая задается для модели в виде константы. Интенсивность отказов в ПС или скорость прохода по цепи зависит лишь от количества дефектов, которые остались в ПС. К этой группе моделей относятся: Джелински- Моранды [10.20], Шика-Вулвертона, Шантикумера [10.21] и др.

Ниже рассматриваются некоторые модели надежности, которые обеспечивают рост надежности ПО (модели роста надежности [10.7, 10.10]), находят широкое применение на этапе тестирования и описывают процесс обнаружения отказов при следующих предположениях:

все ошибки в ПС не зависят друг от друга с точки зрения локализации отказов;

интенсивность отказов пропорциональна текущему числу ошибок в ПС (убывает при тестировании программного обеспечения);

вероятность локализации отказов остается постоянной;

локализованные ошибки устраняются до того, как тестирование будет продолжено;

при устранении ошибок новые ошибки не вносятся.

Приведем основные обозначения величин при описании моделей роста надежности:

m - число обнаруженных отказов ПО за время тестирования;

X_i - интервалы времени между отказами $i-1$ и i, при $i = 1, \ldots, m$;

S_i - моменты времени отказов (длительность тестирования до i -отказа), $S_i = X_k$ при $i = 1, \ldots, m$;

T - продолжительность тестирования ПО (время, для которого определяется надежность);

N - оценка числа ошибок в ПО в начале тестирования;

M - оценка числа прогнозированных ошибок;

$M\mathrm{T}$ - оценка среднего времени до следующего отказа;

$E(T_p)$ - оценка среднего времени до завершения тестирования;

$Var(T_p)$ - оценка дисперсии;

$R(t)$ - функция надежности ПО;

$Z_i(t)$ - функция риска в момент времени t между $i-1$ и i -отказами;

c - коэффициент пропорциональности;

b - частота обнаружения ошибок.

Далее рассматриваются несколько моделей роста надежности, основанные на этих предположениях и использовании результатов тестирования программ в части отказов, времени между ними и др.

Модель Джелинского-Моранды. В этой модели используются исходные данные, приведенные выше, а также:

m - число обнаруженных отказов за время тестирования;

X_i - интервалы времени между отказами;

T - продолжительность тестирования.

Функция риска $Z_i(t)$ в момент времени t расположена между $i-1$ и i имеет вид:

$Z_i(t) = c(N - n_{i-1}),$

где $i = 1, \ldots, m$; $T_{i-1} < t < T_i.$

Эта функция считается ступенчатой кусочнопостоянной функцией с постоянным коэффициентом пропорциональности и величиной ступени - c. Оценка параметров c и N производится с помощью системы уравнений:

$$\sum_{i=1}^{m} \frac{1}{N - N_{i-1}} - \sum_{i=1}^{m} cX_i = 0,$$
$$\frac{n}{c} - NT - \sum_{i=1}^{m} X_i n_{i-1} = 0$$

При этом суммарное время тестирования вычисляется так: $T = \sum_{i=1}^{m} X_i$

Выходные показатели для оценки надежности относительно указанного времени T включают:

число оставшихся ошибок $M_m = N - m$;

среднее время до текущего отказа $M\mathrm{T}_m = 1/(N - m)c$;

среднее время до завершения тестирования и его дисперсию

$$E(T_p) = \sum_{i=1}^{N-n} \frac{1}{ic},$$

$$Var(T_p) = \sum_{i=1}^{N-n\}} \frac{1}{(ic)^2}.$$

При этом функция надежности вычисляется по формуле:

$$Rm(t) = exp(-(N - m)ct),$$

при $t > 0$ и числе ошибок, найденных и исправленных на каждом интервале тестирования, равным единице.

Модель Шика-Вулвертона. Модель используется тогда, когда интенсивность отказов пропорциональна не только текущему числу ошибок, но и времени, прошедшему с момента последнего отказа. Исходные данные для этой модели аналогичны выше рассмотренной модели Джелински-Моранды:

m - число обнаруженных отказов за время тестирования,

X_i - интервалы времени между отказами,

T - продолжительность тестирования.

Функции риска $Z_i(t)$ в момент времени между $i - 1$ и $i - m$ отказами определяются следующим образом:

$$Z_i(t) = c(N - n_{i-1}), \text{где } i = 1, \ldots, m;\ T_{i-1} < t < T_i$$
$$T = \sum_{i=1}^{m} X_i$$

Эта функция является линейной внутри каждого интервала времени между отказами, возрастает с меньшим углом наклона. Оценка с и N вычисляется из системы уравнений:

К выходным показателям надежности относительно продолжительности T относятся:

число оставшихся ошибок $Mm = N - m$;

среднее время до следующего отказа MTт = (p / (2 (N - m) c))1/2;

среднее время до завершения тестирования и его дисперсия

$$E(T_p) = \sum_{i=1}^{N-m} \sqrt{\frac{\pi}{2ic}},$$

$$Var(T_p) = \sum_{i=1}^{N-m} \frac{2 - \pi/2}{ic}.$$

Функция надежности вычисляется по формуле:

$$R_T(t) = exp(-\frac{(N-m)ct^2}{2}), t \geq 0.$$

Модели пуассоновского типа базируются на выявлении отказов и моделируются неоднородным процессом, который задает $\{M(t), t \geq 0\}$ - неоднородный пуассоновский процесс с функцией интенсивности $\lambda(t)$, что соответствует общему количеству отказов ПС за время его использования t.

Модель Гоело-Окумото. В основе этой модели лежит описание процесса обнаружения ошибок с помощью неоднородного пуассоновского процесса, ее можно рассматривать как модель экспоненциального роста. В этой модели интенсивность отказов также зависит от времени. Кроме того, в ней количество выявленных ошибок трактуется как случайная величина, значение которой зависит от теста и других условных факторов.

Исходные данные этой модели:

m - число обнаруженных отказов за время тестирования;

X_i - интервалы времени между отказами;

T - продолжительность тестирования.

Функция среднего числа отказов, обнаруженных к моменту t, имеет вид

$$m(t) = N(1 - e^{-bt}),$$

где b - интенсивность обнаружения отказов и показатель роста надежности $q(t) = b$.

Функция интенсивности $\lambda(t)$ в зависимости от времени работы до отказа равна

$$\lambda(t) = Nb^{-b}, t \geq 0.$$

Оценка b и N получаются из решения уравнений:

$$m/N - 1 + \exp\{(-bT)\} = 0$$
$$m/b - \sum_{i=1}^{m}\{t_i\} - N_m \exp\{(-bT)\} = 0.$$

Выходные показатели надежности относительно времени T определяют:

среднее число ошибок, которые были обнаружены в интервале $[0, T]$, по формуле $E(N_T) = Nexp(-bT)$,

функцию надежности

$$R_T(t) = exp(N(e^{-bt} - e^{-bt(t+\mathrm{T})})), t \geq 0.$$

В этой модели обнаружение ошибки трактуется как случайная величина, значение которой зависит от теста и операционной среды.

В других моделях количество обнаруженных ошибок рассматривается как константа.В моделях роста надежности исходной информацией для расчета надежности являются интервалы времени между отказами тестируемой программы, число отказов и время, для которого определяется надежность программы при отказе. На основании этой информации по моделям определяются показатели надежности вида:

вероятность безотказной работы;

среднее время до следующего отказа;

число необнаруженных отказов (ошибок);

среднее время дополнительного тестирования программы.

Модель анализа результатов прогона тестов использует в своих расчетах общее число экспериментов тестирования и число отказов. Эта модель определяет только вероятность безотказной работы программы и выбрана для случаев, когда предыдущие модели нельзя использовать (мало данных, некорректность вычислений). Формула определения вероятности безотказной работы по числу проведенных экспериментов имеет вид

$$P = 1 - Nex/N,$$

где Nex - число ошибочных экспериментов, N - число проведенных экспериментов для проверки работы ПС.

Таким образом, можно сделать вывод о том, что модели надежности ПС основаны на времени функционирования и/или количестве отказов (ошибок), полученных в программах в процессе их тестирования или эксплуатации. Модели надежности учитывают случайный марковский и пуассоновский характер соответственно процессов обнаружения ошибок в программах, а также характер и интенсивность отказов.

Форма представления результата:

1. Цель работы
2. Введение
3. Программно-аппаратные средства, используемые при выполнении работы.
4. Описание работы
5. Заключение (выводы)
6. Список используемой литературы

Критерии оценки:

Работа выполнена полностью и не содержит ошибок, студент грамотно представил отчет – оценка «отлично».

Работа выполнена полностью, но содержит не более двух ошибок, студент грамотно представил отчет – оценка «хорошо».

Работа выполнена с ошибками, студент представил краткий отчет – оценка «удовлетворительно».

Работа выполнена с грубыми ошибками, отчет составлен неграмотно – оценка «неудовлетворительно».

Лабораторное занятие № 15
Разработка тестовых сценариев (тест-кейсов), оценка необходимого количества тестов, организация тестирования

Цель: оценить необходимое количество тестов, разработать тестовые сценарии (тест-кейсы), провести тестирование, сформировать тестовую документации.

Выполнив работу, Вы будете:

уметь:

- У1 использовать выбранную систему контроля версий;
- У2 использовать методы для получения кода с заданной функциональностью и степенью качества;
- У3 анализировать проектную и техническую документацию;
- У4 использовать специализированные графические средства построения и анализа архитектуры программных продуктов;
- У11 выявлять ошибки в системных компонентах на основе спецификаций;
- У12 использовать различные транспортные протоколы и стандарты форматирования сообщений;
- У13 выполнять тестирование интеграции;
- У14 организовывать постобработку данных;
- У15 создавать классы- исключения на основе базовых классов;
- У16 выполнять ручное и автоматизированное тестирование программного модуля;
- У17 использовать инструментальные средства отладки программных продуктов.

Материальное обеспечение:

Пакет Microsoft Office, доступ к Internet ресурсам, Visual Studio.

Задание:

1. Выполните оценку необходимого количества тестов для определения качества программного приложения для тест кейса:

Действие	Ожидаемый результат
1. Открываем форму отправки сообщения	Форма открыта Все поля по умолчанию пусты Обязательные поля помечены - * Кнопка "Отправить" не активна
2. Заполняем поля формы: Тип обращения Контактное лицо Контактный телефон Сообщение	Поля заполнены Кнопка "Отправить" - активна (Enabled)
3. Нажимаем кнопку "Отправить"	Если введенные данные корректны - Сообщение "Заявка отправлена"выведено на экран. Новая заявка появилась в списке на странице "Заявки". Если введенные данные НЕ корректны -; Валидационное сообщение со всеми ошибками выведено на экран. Заявка НЕ появилась в списке на странице "Заявки".

2. Оцените необходимое количество тестов для разработанного программного продукта (лабораторное задание № 11), разработайте тестовые сценарии (тест-кейсы), проведите тестирование. Определите задачи доработки программного продукта. Сформируйте тестовую документацию.

Порядок выполнения работы:

1. Выполнить оценку необходимого количества тестов для определения качества программного приложения для тест кейса
2. Изучить предлагаемый теоретический материал по теме.
3. Определить необходимые виды тестирования для разработанного программного продукта (лабораторное занятие № 10).
4. Оценить необходимое количество тестов.
5. Разработать тестовые сценарии (тест-кейсы).
6. Провести тестирование:
7. Определить задачи доработки программного продукта.
8. Сформировать тестовую документацию.
9. Сформировать отчет о работе.

Ход работы:

Краткие теоретические сведения

Тестовое Покрытие (Test Coverage)

Тестовое Покрытие- это одна из метрик оценки качества тестирования, представляющая из себя плотность покрытия тестами требований либо исполняемого кода.

Если рассматривать тестирование как "проверку соответствия между реальным и ожидаемым поведением программы, осуществляемая на конечном наборе тестов", то именно этот конечный набор тестов и будет определять тестовое покрытие:

Чем выше требуемый уровень тестового покрытия, тем больше тестов будет выбрано, для проверки тестируемых требований или исполняемого кода.

Сложность современного программного обеспечения и инфраструктуры сделало невыполнимой задачу проведения тестирования со 100% тестовым покрытием. Поэтому для разработки набора тестов, обеспечивающего более менее высокий уровень покрытия можно использовать специальные инструменты либо техники тест дизайна.

Существует 2 широко применяемых подхода к оценке и измерению тестового покрытия:

Покрытие требований (Requirements Coverage)- оценка покрытия тестами функциональных и нефункциональных требований к продукту путем построения матриц трассировки (traceability matrix).

Покрытие кода (Code Coverage)- оценка покрытия исполняемого кода тестами, путем отслеживания непроверенных в процессе тестирования частей программного обеспечения.

Различия: Метод покрытия требований сосредоточен на проверке соответствия набора проводимых тестов требованиям к продукту, в то время как анализ покрытия кода - на полноте проверки тестами, разработанной части продукта (исходного кода).

Ограничения: Метод оценки покрытия кода не выявит нереализованные требования, так как работает не с конечным продуктом, а с существующим исходным кодом. Метод покрытия требований может оставить непроверенными некоторые участки кода, потому что не учитывает конечную реализацию.

Покрытие требований (Requirements Coverage)

Расчет тестового покрытия относительно требований проводится по формуле:

Tcov = (Lcov/Ltotal) * 100%,

где: Tcov- тестовое покрытие, Lcov - количество требований, проверяемых тест кейсами, Ltotal- общее количество требований

Для измерения покрытия требований, необходимо проанализировать требования к продукту и разбить их на пункты. Опционально каждый пункт связывается с тест кейсами, проверяющими

его. Совокупность этих связей - и является матрицей трассировки. Проследив связи, можно понять какие именно требования проверяет тестовый случай.

Тесты, не связанные с требованиями не имеют смысла. Требования, не связанные с тестами - это "белые пятна", т.е. выполнив все созданные тест кейсы, нельзя дать ответ реализовано данное требование в продукте или нет.

Для оптимизации тестового покрытия при тестировании на основании требований, наилучшим способом будет использование стандартных техник тест дизайна. Пример разработки тестовых случаев по имеющимся требованиям рассмотрен в разделе: "Практическое применение техник тест дизайна при разработке тест кейсов"

Покрытие кода (Code Coverage)

Расчет тестового покрытия относительно исполняемого кода программного обеспечения проводится по формуле:

Tcov = (Ltc/Lcode) * 100%,

где: Tcov- тестовое покрытие, Ltc- кол-ва строк кода, покрытых тестами, Lcode- общее кол-во строк кода.

В настоящее время существует инструментарий (например, Clover), позволяющий проанализировать в какие строки были вхождения во время проведения тестирования, благодаря чему можно значительно увеличить покрытие, добавив новые тесты для конкретных случаев, а также избавиться от дублирующих тестов. Проведение такого анализа кода и последующая оптимизация покрытия достаточно легко реализуется в рамках тестирования белого ящика (white-box testing) при модульном, интеграционном и системном тестировании; при тестировании же черного ящика (black-box testing) задача становится довольно дорогостоящей, так как требует много времени и ресурсов на установку, конфигурацию и анализ результатов работы, как со стороны тестировщиков, так и разработчиков.

Техники тест дизайна (Test Design Technics)

Многие люди тестируют и пишут тестовые случаи (test cases), но не многие пользуются специальными техниками тест дизайна. Постепенно, набираясь опыта они осознают, что постоянно делают одну и ту же работу, поддающуюся конкретным правилам. И тогда они находят, что все эти правила уже описаны.

Предлагаю вам ознакомиться с кратким описанием наиболее распространенных техник тест дизайна:

Эквивалентное Разделение (**Equivalence Partitioning - EP**). Как пример, у вас есть диапазон допустимых значений от 1 до 10, вы должны выбрать одно верное значение внутри интервала, скажем, 5, и одно неверное значение вне интервала - 0.

Анализ Граничных Значений (**Boundary Value Analysis - BVA**). Если взять пример выше, в качестве значений для позитивного тестирования выберем минимальную и максимальную границы (1 и 10), и значения больше и меньше границ (0 и 11). Анализ Граничный значений может быть применен к полям, записям, файлам, или к любого рода сущностям, имеющим ограничения.

Причина / Следствие (**Cause/Effect - CE**). Это, как правило, ввод комбинаций условий (причин), для получения ответа от системы (Следствие). Например, вы проверяете возможность добавлять клиента, используя определенную экранную форму. Для этого вам необходимо будет ввести несколько полей, таких как "Имя", "Адрес", "Номер Телефона" а затем, нажать кнопку "Добавить" - эта "Причина". После нажатия кнопки "Добавить", система добавляет клиента в базу данных и показывает его номер на экране - это "Следствие".

Предугадывание ошибки (**Error Guessing - EG**). Это когда тест аналитик использует свои знания системы и способность к интерпретации спецификации на предмет того, чтобы "предугадать" при каких входных условиях система может выдать ошибку. Например, спецификация говорит: "пользователь должен ввести код". Тест аналитик, будет думать: "Что, если я не введу код?", "Что, если я введу неправильный код? ", и так далее. Это и есть предугадывание ошибки.

Исчерпывающее тестирование (**Exhaustive Testing - ET**) - это крайний случай. В пределах этой техники вы должны проверить все возможные комбинации входных значений, и в принципе,

это должно найти все проблемы. На практике применение этого метода не представляется возможным, из-за огромного количества входных значений.

Практическое применение техник тест дизайна при разработке тест кейсов

Многие знают, что такое тест дизайн, но не все умеют его применять. Чтобы немного прояснить ситуацию, мы решили предложить Вашему вниманию последовательный подход к разработке тестовых случаев (тест кейсов), используя самые простейшиетехники тест дизайна:

Эквивалентное Разделение (Equivalence Partitioning), далее в тексте -EP

Анализ Граничных Значений (Boundary Value Analysis), далее в тексте -BVA

Предугадывание ошибки (Error Guessing), далее в тексте -EG

Причина / Следствие (Cause/Effect), далее в тексте -CE

План разработки тест кейсовпредлагается следующий:

Анализ требований.

Определение набора тестовых данных на основании EP,BVA,EG.

Разработка шаблона теста на основании CE.

Написание тест кейсов на основании первоначальных требований, тестовых данных и шагов теста.

Далее на примере, рассмотрим предложенный подход.

Пример:

Протестировать функциональность формы приема заявок, требования к которой предоставлены в следующей таблице:

Элемент	Тип элемента	Требования
Тип обращения	combobox	Набор данных: Консультация Проведение тестирования Размещение рекламы Ошибка на сайте *- на процесс выполнения операции приема заявок не влияет.
Контактное лицо	editbox	1. Обязательное для заполнения 2. Максимально 25 символов 3. Использование цифр и спец символов не допускается
Контактный телефон	editbox	Обязательное для заполнения Допустимые символы "+" и цифры "+" можно использовать только в начале номера Допустимые форматы: начинается с плюса - 11-15 цифр +31612361264 +375291438884 без плюса - 5-10 цифр, например: 0613261264 2925167
Сообщение	text area	1. Обязательное для заполнения 2. Максимальная длина 1024 символа
Отправить	button	Состояние: 1. По умолчанию - не активна (Disabled) 2. После заполнения обязательных полей становится активна (Enabled) Действия после нажатия 1. Если введенные данные корректны - отправка сообщения 2. Если введенные данные НЕ корректны - валидационное сообщение

Вариант использования (иногда его может и не быть) представлен на рисунке 1.

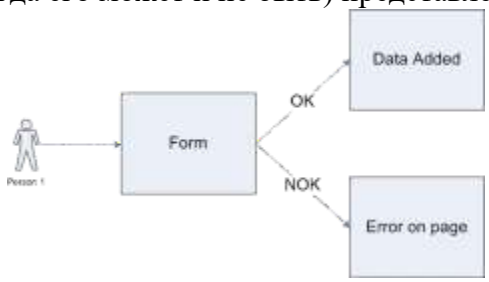


Рисунок 1 – Вариант использования

1. Анализ требований

Читаем, анализируем требования и выделяем для себя следующие нюансы:

какие из полей обязательные для заполнения?

имеют ли поля ограничения по длине или по размерности (границы)?

какие из полей имеют специальные форматы?

2. Определение набора тестовых данных

Отталкиваясь от требований к полям, используя техники тест дизайна начинаем определение набора тестовых данных:

в зависимости от того обязательное поле или нет, определим какие поля необходимо проверить на пустое значение, так как оно может вызывать ошибку (В результирующей таблице оранжевыйцвет)

т.к. исчерпывающее тестирование не представляется возможным из-за огромного числа всевозможных комбинаций значений, в первую очередь необходимо определитьминимальный набор данных. Это можно сделать используя такиетехники, какЕРиBVA. (В результирующей таблицеголубойцвет)

На форме присутствует поле, имеющее составной тип (цифры используются совместно с символами), обладает специальным форматом данных и поэтому выделение тестовых данных для него - это достаточно трудоемкая задача. В пределах данной статьи ограничимся только простой проверкой форматов и основных требований описанных в форме приема заявок.

По завершению генерации данных используя стандартные техники, можно добавить некоторое количество значений на основании личного опыта (техника EG) - это будет использование спец. символов, очень длинных строк, разных форматов данных, регистров в строках (Upper, Lowwer, Mixed cases), отрицательные и нулевые значения, кейворды Null - NaN - Infinity и т.д. Сюда можно включить все, что вы полагаете может вывести приложение из строя (В результирующей таблицефиолетовыйцвет)

Примечание:

Отметим, что количество тестовых данных после окончательной генерации будет достаточно большим, даже при использовании специальных техник тест дизайна. Поэтому ограничимся лишь несколькими значениями для каждого поля, так как цель данной статьи показать именно процесс создания тест кейсов, а не процесс получения конкретных тестовых данных.

2.1 Выбор тестовых данных для каждого отдельно взятого поля

Поле Тип обращения. Так как все данные входят в 1 класс эквивалентности, то есть не изменяют сам процесс выполнения приема заявки, берем любою (1-ю) позицию в листе с ожидаемым результатом ОК. Но т.к. реализовано поле как лист, имеет также смысл рассмотреть и граничные условия (техника BVA), т.е. берем первый и последний элементы. Итого: 1-я и последняя позиции в листе. Ожидаемый результат при использовании - ОК.

Поле Контактное лицо. Это обязательное поле размером от 1 до 25 символов (включая границы). Проверка на обязательность добавляет к тестовым данным пустое значение. Проведем анализ граничных условий (BVA), получим набор: 0, 1, 2, 24, 25 и 26 символов. Пустое значение (0 символов) уже было добавлено при анализе обязательности поля для ввода, поэтому при BVA мы не будем добавлять его еще раз. (если его добавить второй раз, произойдет дублирование тестовых данных, которое не приведет к нахождению новых дефектов, а значит повторное

добавление в домен не имеет смысла). В связи с тем, что значения 2 и 24 символа являются, с нашей точки зрения, некритичными, их можно не добавлять. В итоге получаем, что минимальный набор данных для тестирования поля - это строки 1 и 25 - ОК, и 0 (пустое значение), 26 символов - NOK.

поле Контактный телефонсостоит из нескольких частей: код страны, код оператора, номер телефон (который может быть составной и разделенный дефисами). Для определения правильного набора тестовых данных необходимо рассматривать каждую составную часть по-отдельности. Применяя BVA и EP, получим:

для номеров с плюсомПо BVA получим номера с 10, 11, 12 и 14, 15, 16 цифрами, где 10 и 16 - NOK, а 11, 12, 14, 15 - OK Рассматривая полученные данные с позиции EP выделим, что 11, 12, 14, 15 входят в один класс эквивалентности. Поэтому при тестировании мы можем использовать любое из них, но так как 11 и 15 - это границы интервала, то на наш взгляд их пропускать нельзя. Следовательно мы можем уменьшить набор значений до двух, исключив 12 и 14, а оставив 11 и 15 для проверки граничных условий. Итого имеем: 11 и 15 цифр - OK, (+12345678901, +123456789012345) 10 и 16 цифр - NOK; (+1234567890, +12345678901234 56)

для номеров без плюса:По BVA получим номера с 4, 5, 6 и 9, 10, 11 цифрами. Действуя аналогично примеру для номеров телефонов с плюсом, исключим значения 6 и 9, оставив 5 и 10. Итого имеем: 5 и 10 цифр - OK, (12345, 1234567890) 4 и 11 цифр - NOK; (1234, 12345678901)

поле Сообщение. подбор данных проводим по аналогии с полем Контактное лицо. На выходе получаем значения: строки 1 и 1024 - ОК, и 1025 символов - NOK.

Результирующая таблица данных, для использования при последующем составлении тест кейсов

Поле	OK/NOK	Значение	Комментарий
Тип обращения	OK	Консультация	первый в списке
		Ошибка на сайте	последний в списке
	NOK		
Контактное лицо	OK	йцукенгшщзйцукенгшщзйцуке	25 символов нижний регистр
		а	1 символ
		ЙЦУКЕНГШЩЗФЫВАПРОЛДЖЯЧСМИ	25 символов ВЕРХНИЙ регистр
		ЙЦУКЕНГШЩЗфывапролджЯЧСМИ	25 символов СМеШаННыЙ регистр
	NOK		пустое значение
		йцукенгшщзйцукенгшщзйцукей	длина больше максимальной(26 символов
		@#$%^&;.?,>|\/№"!()_{}[<~	спец. символы (ASCII)
		123456789012345678901234 5	только цифры
		adsadasdasdas dasdasd asasdsads(...)sas	очень длинная строка (~1Mb)
Контактный	OK	+12345678901	с плюсом - минимальная

телефон			длина
		+123456789012345	с плюсом - максимальная длина
		12345	без плюса - минимальная длина
		1234567890	без плюса - максимальная длина
	NOK		пустое значение
		+1234567890	с плюсом - < минимальной длины
		+1234567890123456	с плюсом - > максимальной длины
		1234	без плюса - < минимальной длины
		12345678901	без плюса - > максимальной длины
		+YYYXXXyyyxxzz	с плюсом - буквы вместо цифр
		yyyxxxxzz	без плюса - буквы вместо цифр
		+###-$$$-%^-&^-&!	спец. символы (ASCII)
		12323123231232132312 32(...)99	очень длинная строка (~1Mb)
Сообщение	OK	йццуйцуйц(...)йцу	максимальная длина (1024 символа)
	NOK		пустое значение
		йццуйцуйц(...)йцуц	длина больше максимальной (1025 символов)
		adsadasdasdas dasdasd asasdsads(...)sas	очень длинная строка (~1Mb)
		@##$$$%^&^&	только спец. символы (ASCII)

Пакеты тестов

Как правило, модульные тесты и тестовые конфигурации разрабатываются самими разработчиками, основная задача тестера в этом случае – организовать все тесты в упорядоченную структуру пакетов и указать, какие пакеты должны исполняться в каких условиях. Вся иерархия тестов хранится в так называемом файле метаданных, имеющем расширение vsmdi. Этот файл находится в системе контроля версий и может быть включен в решение как отдельный элемент (рисунок 2).

Для создания тестового пакета можно воспользоваться меню "Create New Test List", как показано на рисунке 3, и после этого откроется окно для задания имени нового пакета и определения его места в иерархии тестовых пакетов (рисунок 4). В том случае, если для решения уже создан файл метаданных, пакет тестов будет добавлен к нему, если же файла метаданных еще создано не было, он будет создан автоматически.

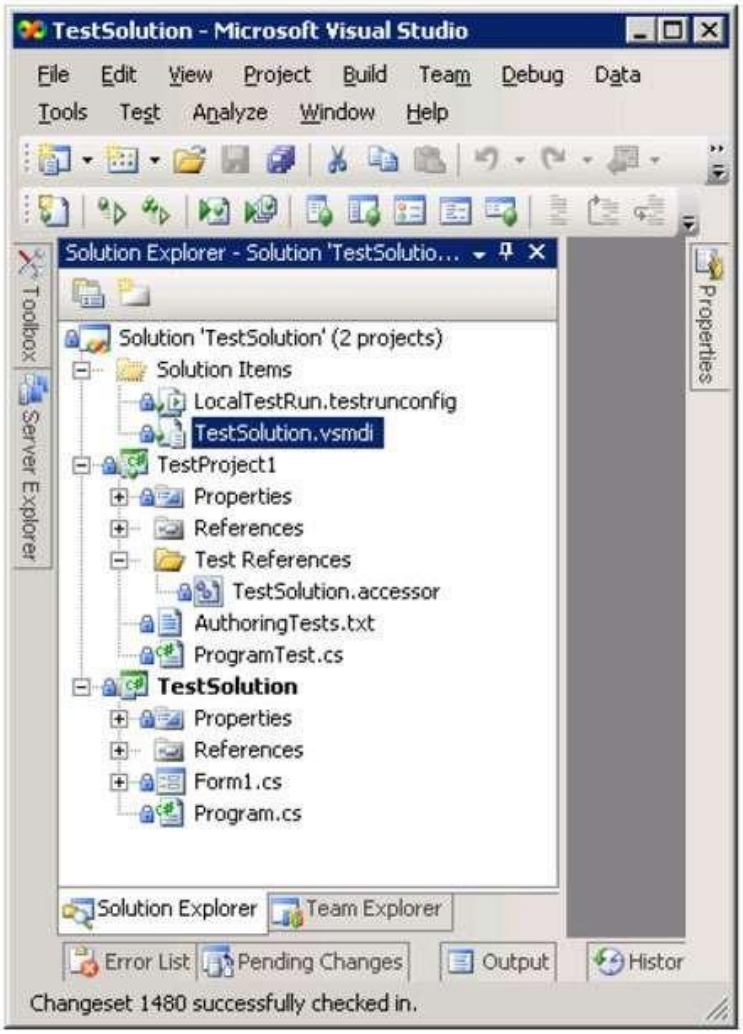


Рисунок 2 – Пакет с иерархией тестов

Рисунок 3 – Создание списка тестов

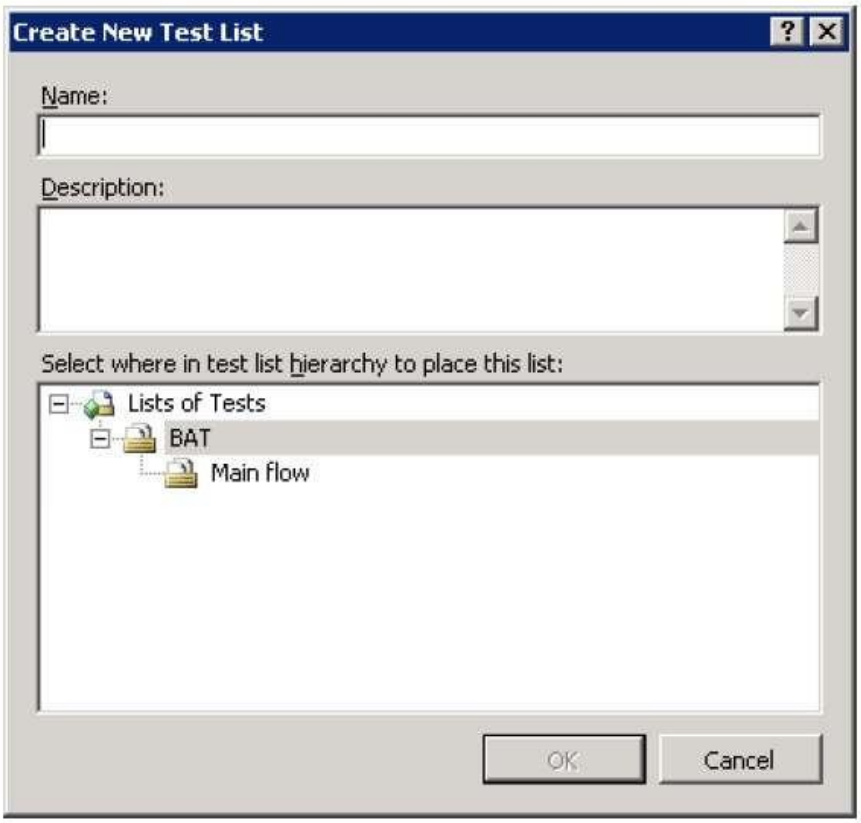


Рисунок 4 – Свойства нового списка тестов

Содержимое пакетов тестов редактируется с помощью специального редактора, показанного на рисунке 5. В пакеты могут быть включены тесты, находящиеся в одном из проектов текущего решения.

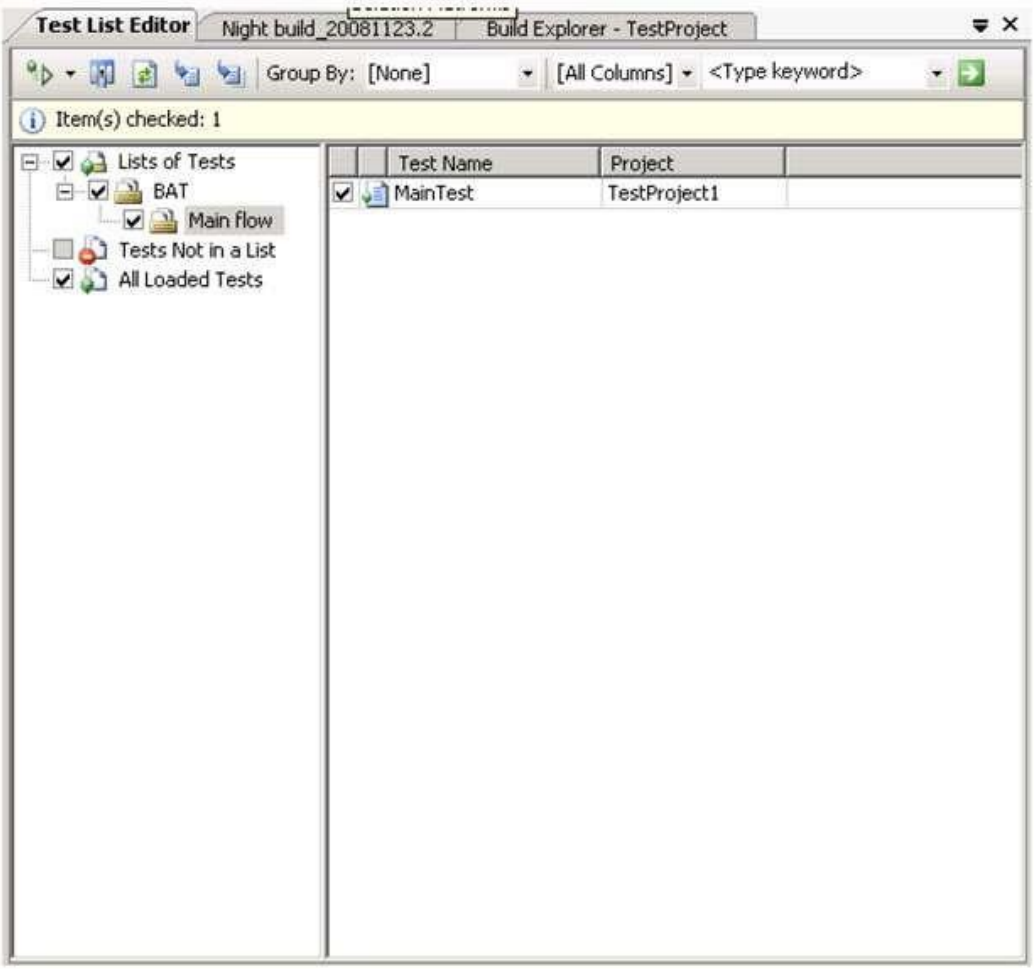


Рисунок 5 – Редактор список тестов

Тестовые пакеты могут использоваться как для ручного прогона тестов определенной тематики (команда "Run checked", рисунок 6), так и для автоматического прогона в рамках автоматической сборки.

Рисунок 6 – Ручной" запуск пакета тестов

Указать тесты, которые будут запускаться при определенной сборке можно при создании файла с описанием сборки MsBuild, или в последствии через модификацию проекта MsBuild. В первом случае достаточно на соответствующем шаге мастера выбрать файл метаданных и отметить галочками интересующие пакеты тестов (рисунок 4.6). Во втором случае необходимо открыть проект MsBuild в редакторе XML, найти элемент MetaDataFile, или вписать необходимые пакеты вручную:

```
<MetaDataFile
Include="$(BuildProjectFolderPath)/../../TestSolution/TestSolution.vsmdi">
<TestList>BAT/Main flow</TestList>
</MetaDataFile>
```

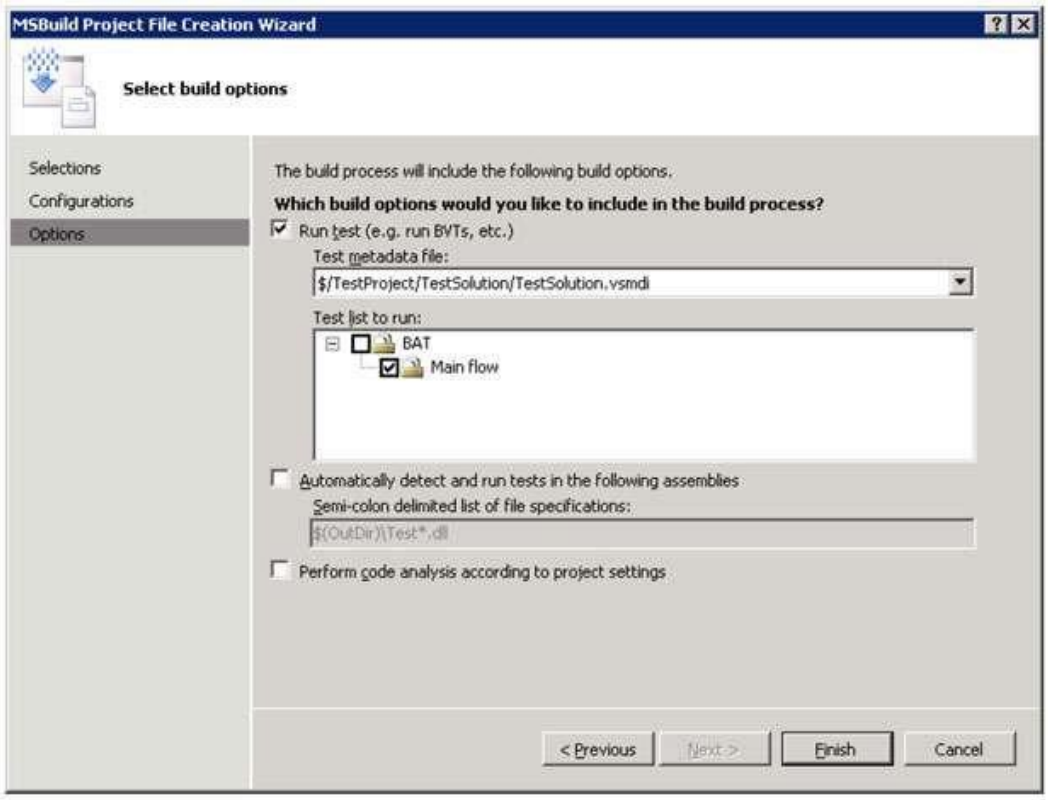


Рисунок 7 – Выбор пакета тестов при автоматической сборке

Автоматическое тестирование Web-приложений

Capture & Playback подход. Этот подход к тестированию пользовательских интерфейсов выглядит очень эффектно и основан на следующей идее. Тестировщик проходится мышкой по окнам, пунктам меню и другим элементам интерфейса. Специальная программа записывает его шаги и потом их воспроизводит в пакетном режиме. То есть очень просто получаются повторяемые тесты. Технически это устраивается так.

Специальное тестовое окружение с той или иной точностью распознает, куда именно было нажато мышкой на экране и создает соответствующий код в специальном скрипте. Потом этот скрипт "прогоняется" в пакетном режиме, воспроизводя действия тестировщиков. Весь вопрос в том, каким образом распознается клик тестировщика мышкой. Идеально, когда этот клик связывается с соответствующим элементом управления интерфейса. То есть если тестировщик нажал кнопку в каком-то диалоге, то в скрипте эта информация сохраняется в полном объеме. Другая, более грубая ситуация имеет место тогда, когда тестовое окружение не может распознать, какой элемент пользовательского интерфейса активировал тестировщик. Тогда в скрипт заносится информация о тех координатах на экране, куда был клик мышкой.

Чем плоха последняя ситуация? Дело в том, что при малейшем изменении пользовательского интерфейса (а это типичная ситуация, ведь ПО развивается, дорабатывается и тестируется одновременно) автоматический тест-скрипт, созданный таким образом, выходит из строя. Там, куда раньше клик мышкой попадал, например, на нужную кнопку, теперь находится совсем другой элемент управления.

Если же тестовое окружение распознало элемент интерфейса, то такой тест-скрипт оказывается более "живучим". Это происходит на уровне перехвата сообщений на уровне операционной системы (в частности, Windows). Но для того, чтобы это было возможно, код приложения должен быть написан "правильным" образом. Далеко не все интерфейсные приложения написаны "правильно".

То, насколько успешно для конкретного приложения можно применить данный подход, определяется несколькими факторами. Основным является то, какая используется платформа (например, Java Swing, AWT, Windows Forms, WPF, etc.) и дополнительные библиотеки с

элементами пользовательского интерфейса. Наиболее зрелой платформой с этой точки зрения на данный момент является MS Windows Forms.

Capture & Playback при тестировании Web-интерфейсов. В случае с тестированием Web-интерфейса ситуация с точностью распознавания элементов управления (interface controls) значительно проще, чем при тестировании произвольного пользовательского интерфейса. Взаимодействие с сервером происходит по строго описанному протоколу HTTP, что позволяет при записи перехватить отправляемые и получаемые сообщения. Кроме того, визуальное представление страницы задано в структурированном формате HTML, что позволяет легко опознать отдельные элементы на странице.

В издание Visual Studio Team Edition for Software Testers включен дополнительный пакет, облегчающий автоматизацию тестирования Web-приложений методом Capture & Playback. Он позволяет, как автоматически генерировать простые тестовые сценарии на основе записи действия пользователя, так и писать более точные тесты на любом языке платформы .NET.

Добавить новый Web-тест к решению можно с помощью команды "Test/New Test", выбрав в возникшем диалоге (рисунок 8) тип теста Web Test.

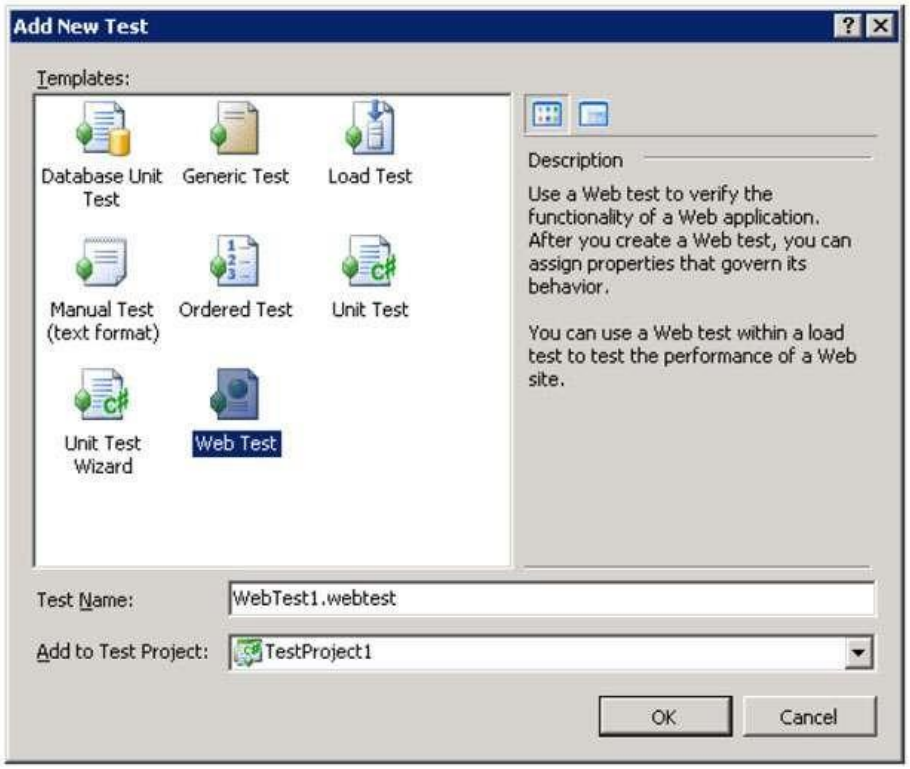


Рисунок 8 – Создание Web-теста

После создания нового теста автоматически будет запущена процедура записи сценария теста в браузере (рисунок 9). На этом этапе достаточно ввести www-адрес приложения, которое следует протестировать, после чего выполнить тестовый "проход" по Web-интерфейсу непосредственно в браузере.

Рисунок 9 – Запись шагов тестировщика Web-приложения в Internet Explorer

После окончания записи будет автоматически сгенерирован тест, включающий все отправленные на сервер http-запросы и все полученные ответы (рисунок 10). При этом генератор автоматически добавит некоторые правила, по которым будет проверяться корректность работы теста.

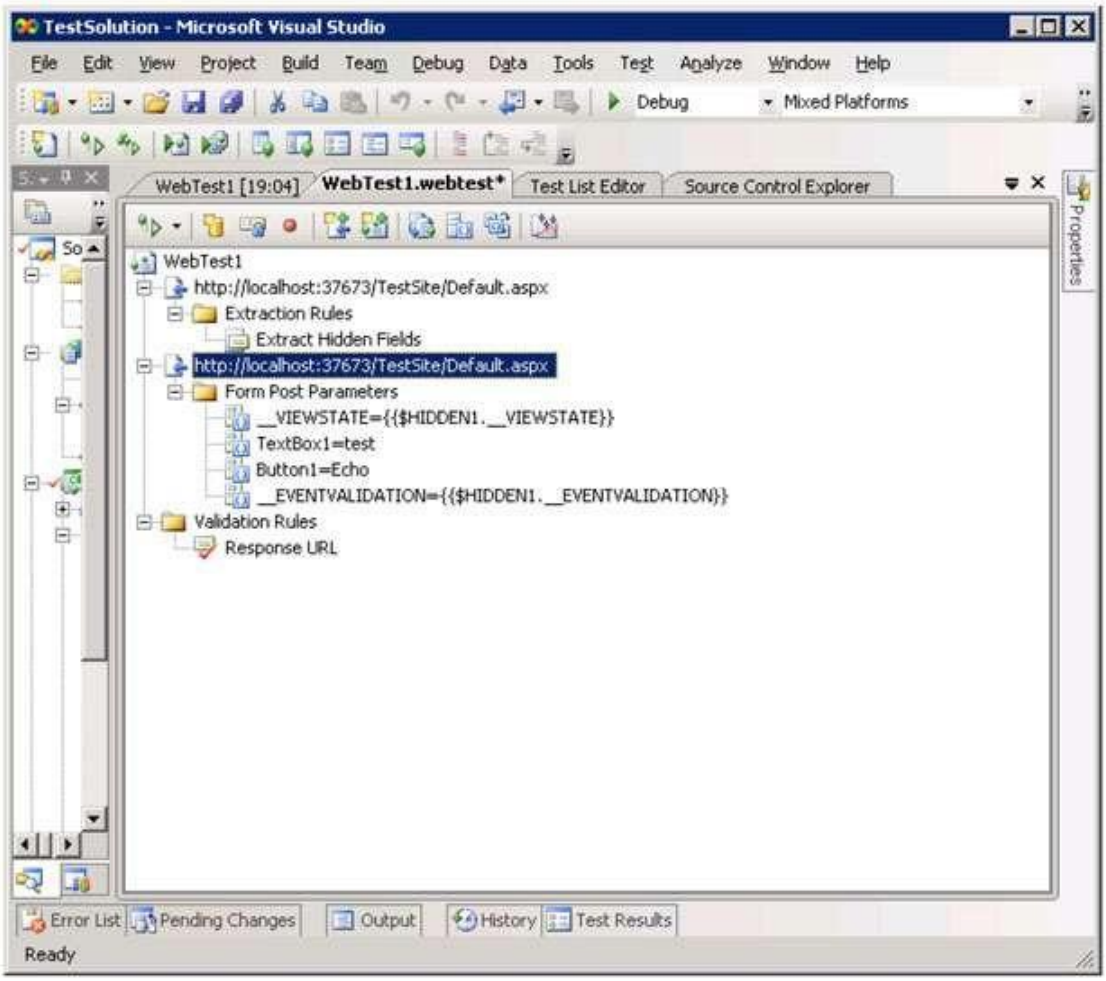


Рисунок 10 – Редактор Web-теста

Для каждого из шагов теста можно, с помощью визуального редактора, добавить дополнительные проверки или опции, управляющие ходом выполнения: поиск подстроки в ответе сервера (тексте полученного HTML), валидацию HTML через задание регулярного выражения, проверка наличия или отсутствия определенных тегов или атрибутов на странице и т.д. Допускается также возможность разработки собственных правил на любом .NET языке. В тех же случаях, когда гибкости редактора недостаточно, можно сгенерировать C# код для данного теста и реализовать необходимую логику "вручную".

При написании правил валидации очень важно правильно выбрать необходимый уровень детализации. Чем более детально сформулировано правило и чем более специфично оно для данного HTML, тем больше вероятность того, что тест придется изменять при изменении кода тестируемого приложения, даже если эти изменения не касались напрямую этой части. Хорошее правило валидации должно проверять только то, что является важной частью бизнес логики приложения или то, что является ключевым свойством данной HTML-страницы. При этом правило не должно проверять детали верстки и дополнительные визуальные эффекты. К сожалению, автоматически сгенерированные правила далеко не всегда оказываются наиболее эффективными.

Созданный в редакторе или в ручную тест является полноправным тестом и может быть включен в тестовый пакет, а следовательно, и в процедуры автоматической сборки (рисунок 11). Однако, для того, чтобы автоматическое тестирование было возможным, необходимо соответствующим образом настроить сервер автоматических сборок – на нем должен быть развернут сервер IIS с тестовым сайтом, а одним из этапов сборки должно быть обновление кода этого сайта.

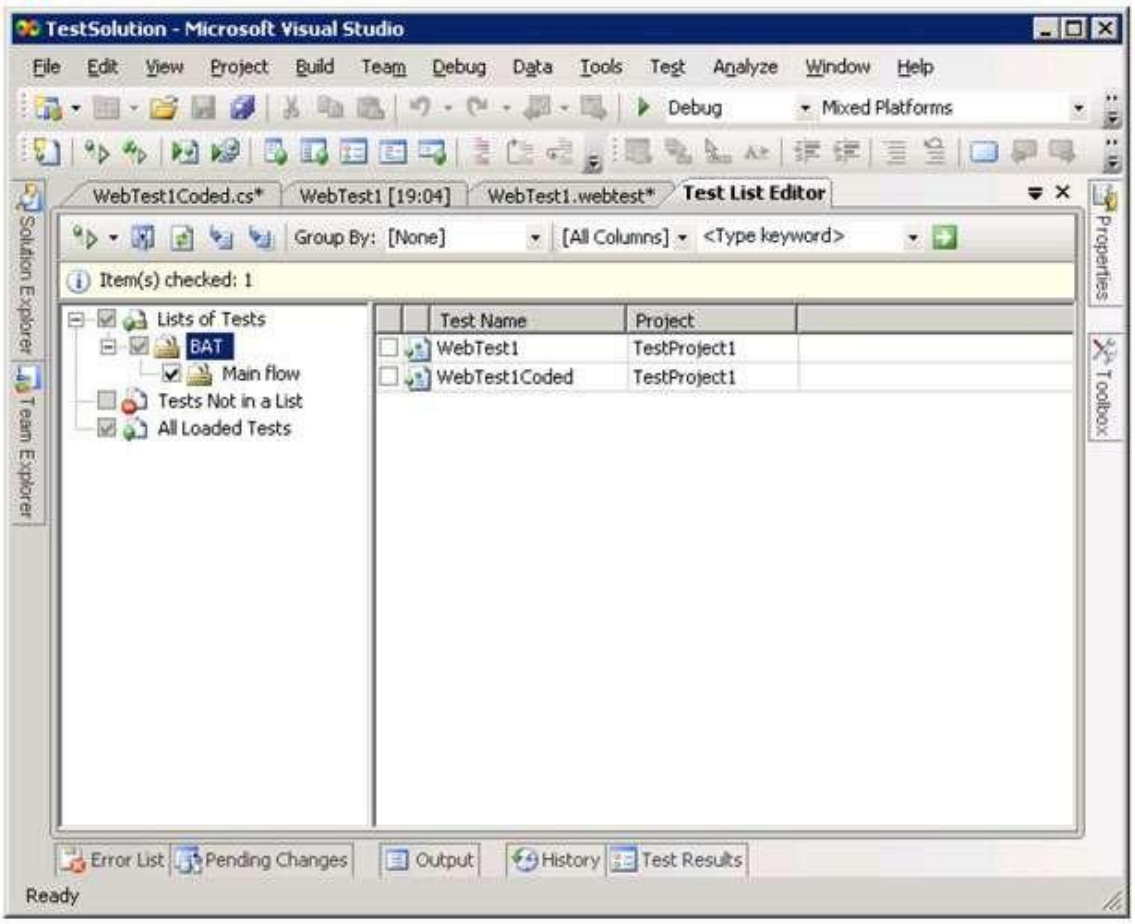


Рисунок 11 – Web-тесты в пакетах тестов

Форма представления результата:

1. Цель работы
2. Введение
3. Программно-аппаратные средства, используемые при выполнении работы.
4. Описание работы
5. Заключение (выводы)
6. Список используемой литературы

Критерии оценки:

Работа выполнена полностью и не содержит ошибок, студент грамотно представил отчет – оценка «отлично».

Работа выполнена полностью, но содержит не более двух ошибок, студент грамотно представил отчет – оценка «хорошо».

Работа выполнена с ошибками, студент представил краткий отчет – оценка «удовлетворительно».

Работа выполнена с грубыми ошибками, отчет составлен неграмотно – оценка «неудовлетворительно».